

# THINK Pascal™

*The Fastest Way to Finished Software*

## **Object-Oriented Programming Manual**





# THINK Pascal

---



*The Fastest Way to Finished Software*

## ***Object-Oriented Programming Manual***



## Credits

<b>User Manual</b>	Philip Borenstein and Jeff Mattson
<b>THINK Pascal</b>	Rich Siegel, John McEnerney, David Neal
<b>THINK Class Library</b>	Dan Podwall and Gregory H. Dow
<b>Quality Assurance</b>	David Allcott, Michael Rockhold, and Paul Vetri
<b>Technical Support</b>	Michael Carland, Mark Geschelin, and Phil Shapiro
<b>Marketing Manager</b>	Susan Smith
<b>Product Manager</b>	Philip Borenstein

Copyright © 1989, 1991 Symantec Corporation.  
All Rights Reserved. Printed in U.S.A.

Symantec Corporation  
10201 Torre Avenue  
Cupertino, CA 95014  
408/253-9600

THINK C and THINK Pascal are trademarks of Symantec Corporation. Other brands and their products are trademarks of their respective holders.

ResEdit, SAREz, and SAdRez are copyrighted programs of Apple Computer, Inc. licensed to Symantec Corp. to distribute for use only in combination with THINK C. Apple software shall not be copied onto another diskette (except for archive purposes) or into memory unless as part of execution of THINK C. When THINK C has completed execution, Apple Software shall not be used by any other program.

The *THINK C Object-Oriented Programming Manual* is copyrighted and all rights reserved. Information in this document is subject to change without notice and does not represent a commitment on the part of Symantec Corporation. The software described in this document is furnished under a license agreement. The document may not, in whole or in part, be copied, photocopied, reproduced, translated, or reduced to any electronic medium or machine-readable form without prior consent, in writing, from Symantec Corporation.

SYMANTEC CORPORATION MAKES NO WARRANTIES, EITHER EXPRESS OR IMPLIED, REGARDING THE ENCLOSED COMPUTER SOFTWARE PACKAGE, ITS MERCHANTABILITY, OR ITS FITNESS FOR ANY PARTICULAR PURPOSE. THE EXCLUSION OF IMPLIED WARRANTIES IS NOT PERMITTED BY SOME STATES. THE ABOVE EXCLUSION MAY NOT APPLY TO YOU. THIS WARRANTY PROVIDES YOU WITH SPECIFIC LEGAL RIGHTS. THERE MAY BE OTHER RIGHTS THAT YOU MAY HAVE WHICH VARY FROM STATE TO STATE.

# Contents

## *Getting Started with Objects*

1	Welcome . . . . .	3
	What's in this Manual . . . . .	5
	What's New in the THINK Class Library . . . . .	6
	Long coordinates . . . . .	6
	Improved error handling . . . . .	7
	Dependent objects . . . . .	7
	Director owners . . . . .	8
	Abstract text . . . . .	8
	Compatibility with earlier versions . . . . .	8
2	Installing the THINK Class Library . . . . .	9
	Installation . . . . .	11
	Building the Starter Project . . . . .	12
	Installing the TMPL Resources into ResEdit . . . . .	13

## *Object-Oriented Programming*

3	Object-Oriented Programming . . . . .	17
	Overview . . . . .	19
	Objects and Messages . . . . .	19
	Classes . . . . .	20
	Inheritance and Polymorphism . . . . .	21
	Objects and the Macintosh Interface . . . . .	22
	Working with Objects . . . . .	23
	What classes should I define? . . . . .	23
	When should I create a subclass? . . . . .	23
	What should be a method? . . . . .	24
	When should I use procedural programming? . . . . .	24
	Where to Go Next . . . . .	24
4	Object Pascal . . . . .	25
	Overview . . . . .	27
	Declaring a Class . . . . .	27

Declaring and Using Objects . . . . .	29
Creating and deleting objects . . . . .	30
Referring to instance variables . . . . .	30
Class membership . . . . .	30
Defining and Using Methods . . . . .	31
Referring to the current object . . . . .	31
Calling a method . . . . .	32
Calling an inherited method within a method . . . . .	32
Using TObject. . . . .	33
Tips and Techniques . . . . .	35
Organizing your classes . . . . .	35
Objects and segments. . . . .	35
Objects and the Macintosh . . . . .	35
Keywords. . . . .	37
Summary . . . . .	37

<b>5 Tutorial: LearnOOP . . . . .</b>	<b>39</b>
What Does LearnOOP Do? . . . . .	41
How Does LearnOOP Work? . . . . .	43
How Do Objects Know How to Draw Themselves? . . . . .	44
How Do You Create an Object? . . . . .	49
How Do You Define a New Class? . . . . .	50
But How Does It Know? . . . . .	52
Where to Go Next . . . . .	52

<b>6 The Class Browser . . . . .</b>	<b>53</b>
Using the Class Browser . . . . .	55
Finding the Declaration of a Class . . . . .	55
Finding a class declaration with the Class Browser . . . . .	55
Finding a class declaration with the editor . . . . .	56
Finding the Definition of a Method. . . . .	57
Finding a method definition with the Class Browser. . . . .	57
Finding a method definition with the editor . . . . .	58
Keyboard Shortcuts in the Class Browser. . . . .	59
Class Browser Summary . . . . .	60

### ***The THINK Class Library***

<b>7 The THINK Class Library . . . . .</b>	<b>63</b>
Introduction . . . . .	65
What you should know . . . . .	65
Conventions . . . . .	65
Types . . . . .	65
Naming conventions . . . . .	66
Object-oriented terminology . . . . .	66

Overview . . . . .	67
The class hierarchy . . . . .	67
The visual hierarchy . . . . .	69
The chain of command . . . . .	70
The flow of control . . . . .	71
Writing an Application with the TCL . . . . .	72
Creating the project in THINK Pascal . . . . .	73
Creating the project in THINK C . . . . .	74
Creating the application subclass . . . . .	74
Creating the document subclass . . . . .	75
Creating the pane subclass . . . . .	76
Working with Panes . . . . .	77
Windows and panes . . . . .	78
Coordinate systems . . . . .	79
Drawing in a pane . . . . .	80
Properties of panes . . . . .	81
Panoramas . . . . .	83
Scroll panes . . . . .	86
Cursor tracking . . . . .	87
Initializing views from resources . . . . .	87
Working with Menus . . . . .	88
Using MENU resources . . . . .	89
Building menus on the fly . . . . .	90
Dimming and checking menu items . . . . .	90
Handling Low Memory Situations . . . . .	92
Undoing and Mouse Tracking . . . . .	93
Undoing . . . . .	93
Mouse tracking . . . . .	94
Segmentation and the THINK Class Library . . . . .	94
Debugging and the THINK Class Library . . . . .	95
Debugging aids in THINK C . . . . .	95
Debugging aids in THINK Pascal . . . . .	96
THINK Class Library Resources . . . . .	96
Alerts . . . . .	96
Controls . . . . .	97
Error message strings . . . . .	97
Menus . . . . .	97
Menu bars . . . . .	98
Small icon . . . . .	98
Strings and string lists . . . . .	98
Window template . . . . .	99
Modifying the THINK Class Library . . . . .	99
Where to Go Next . . . . .	100
 8 Exception Handling . . . . .	 101
What Is an Exception Handler? . . . . .	103
Using the Exception Mechanism . . . . .	104
Exception handling conventions . . . . .	105
Displaying error messages . . . . .	105

Exception Handling in THINK C . . . . .	106
The basic form . . . . .	106
Use handlers only when necessary . . . . .	107
Returning values . . . . .	108
Special cases. . . . .	109
Exception Handling in THINK Pascal . . . . .	109
The basic form . . . . .	109
Use handlers only when necessary . . . . .	111
Using Exit with exception handlers . . . . .	113
Special cases. . . . .	113
Exception Handler Routines . . . . .	113
Exception handler routines . . . . .	113
Exception raising routine . . . . .	114
Error detection routines . . . . .	115
Utility routines . . . . .	115
 9 CAbstractText . . . . .	117
Introduction . . . . .	117
Heritage . . . . .	117
Using CAbstractText . . . . .	117
Variables . . . . .	118
Methods . . . . .	119
Construction and destruction methods . . . . .	119
Accessing method . . . . .	120
Command methods . . . . .	122
Display methods . . . . .	124
Text specification methods . . . . .	125
Text characteristics methods . . . . .	127
Calibrating methods . . . . .	129
Cursor method . . . . .	130
 10 CAppleEvent . . . . .	131
Introduction . . . . .	131
Heritage . . . . .	131
Using CAppleEvent . . . . .	131
How the THINK Class Library handles AppleEvents . . . . .	132
Variables . . . . .	132
Methods . . . . .	133
Creation and destruction. . . . .	133
Accessing methods . . . . .	133
 11 CApplication . . . . .	137
Introduction . . . . .	137
Heritage . . . . .	137
Using CApplication . . . . .	137
The application and the chain of command . . . . .	137
Writing the main program . . . . .	137
Handling low memory situations . . . . .	138



Variables . . . . .	139
Global variable . . . . .	140
Event related instance variables . . . . .	140
Phase related instance variables . . . . .	140
Memory related instance variables . . . . .	141
Standard file instance variables . . . . .	142
Methods . . . . .	142
Initialization methods . . . . .	142
Advanced initialization methods . . . . .	147
Accessing methods . . . . .	148
Command methods . . . . .	148
Memory management methods . . . . .	151
Execution methods . . . . .	154
Document methods . . . . .	157
Periodic task methods . . . . .	158
Class resources . . . . .	158
 12 CArray . . . . .	 159
Introduction . . . . .	159
Heritage . . . . .	159
Using CArray . . . . .	159
Variables . . . . .	161
Methods . . . . .	161
Creation and destruction methods . . . . .	161
Accessing methods . . . . .	161
Insertion and deletion methods . . . . .	162
Membership method . . . . .	162
Moving methods . . . . .	163
Resizing methods . . . . .	164
Temporary storage methods . . . . .	164
Offset method . . . . .	164
 13 CBartender . . . . .	 165
Introduction . . . . .	165
Heritage . . . . .	165
Using CBartender . . . . .	165
Creating standard menus . . . . .	165
Resource based menus . . . . .	168
Creating hierarchical menus . . . . .	170
Dimming and checking menu items . . . . .	171
Variables . . . . .	172
Global variable . . . . .	172
Instance variables . . . . .	172
Methods . . . . .	172
Construction methods . . . . .	173
Insertion and deletion methods . . . . .	173
Item manipulation methods . . . . .	174
Item insertion and deletion . . . . .	175
Look-up methods . . . . .	175
Appearance methods . . . . .	176
Command Extraction methods . . . . .	178

<b>14 CBitmap . . . . .</b>	<b>. 179</b>
Introduction . . . . .	. 179
Heritage . . . . .	. 179
Using CBitmap . . . . .	. 179
Methods . . . . .	. 181
Construction and destruction methods . . . . .	. 181
Accessing methods . . . . .	. 181
Image copying methods . . . . .	. 182
Drawing preparation methods . . . . .	. 182
<b>15 CBitmapPane . . . . .</b>	<b>. 183</b>
Introduction . . . . .	. 183
Heritage . . . . .	. 183
Using CBitmapPane . . . . .	. 183
Variables . . . . .	. 184
Methods . . . . .	. 184
Construction and destruction methods . . . . .	. 184
Accessing methods . . . . .	. 185
Drawing method . . . . .	. 185
<b>16 CBureaucrat . . . . .</b>	<b>. 187</b>
Introduction . . . . .	. 187
Heritage . . . . .	. 187
Using CBureaucrat . . . . .	. 187
Variables . . . . .	. 188
Global variable . . . . .	. 188
Instance variable . . . . .	. 188
Methods . . . . .	. 188
Construction and destruction methods . . . . .	. 188
Accessing method . . . . .	. 188
Command methods . . . . .	. 188
Change notification methods . . . . .	. 191
<b>17 CButton . . . . .</b>	<b>. 193</b>
Introduction . . . . .	. 193
Heritage . . . . .	. 193
Using CButton . . . . .	. 193
Variables . . . . .	. 194
Methods . . . . .	. 194
<b>18 CCharGrid . . . . .</b>	<b>. 197</b>
Introduction . . . . .	. 197
Heritage . . . . .	. 197
Using CCharGrid . . . . .	. 197
Variables . . . . .	. 199
Methods . . . . .	. 199
Construction methods . . . . .	. 199
Drawing methods . . . . .	. 200

<b>19</b>	<b>CCheckBox</b>	<b>201</b>
	Introduction	201
	Heritage	201
	Using CCheckBox	201
	Variables	202
	Methods	202
<b>20</b>	<b>CChore</b>	<b>205</b>
	Introduction	205
	Heritage	205
	Using CChore	205
	Idle chores	205
	Urgent chores	206
	Using chores	206
	Variables	206
	Methods	206
<b>21</b>	<b>CClipboard</b>	<b>209</b>
	Introduction	209
	Heritage	209
	Using CClipboard	209
	Implementing a CClipboard subclass	210
	Variables	210
	Global variable	210
	Instance variables	211
	Methods	211
	Construction and destruction methods	211
	Suspend and resume methods	212
	Appearance methods	212
	Accessing methods	213
	Scrap conversion methods	214
	Class resources	216
<b>22</b>	<b>CCluster</b>	<b>217</b>
	Introduction	217
	Heritage	217
	Using CCluster	217
	Variables	218
	Methods	218
	Construction and destruction	218
	Insertion and deletion	218
	Membership	219
	Iteration	220
<b>23</b>	<b>CCollaborator</b>	<b>223</b>
	Introduction	223
	Heritage	223
	Using CCollaborator	223
	Variables	226

Methods . . . . .	226
Creation and destruction . . . . .	226
Creating dependency method . . . . .	227
Change notification methods . . . . .	227
List update methods . . . . .	227
24 CCollection . . . . .	229
Introduction . . . . .	229
Heritage . . . . .	229
Using CCollection . . . . .	229
Variables . . . . .	229
Methods . . . . .	229
25 CControl. . . . .	231
Introduction . . . . .	231
Heritage . . . . .	231
Using CControl . . . . .	231
Variables . . . . .	231
Methods . . . . .	231
Construction and destruction methods . . . . .	232
Accessing methods . . . . .	232
Appearance methods . . . . .	233
Click response methods . . . . .	235
26 CDataFile . . . . .	237
Introduction . . . . .	237
Heritage . . . . .	237
Using CDataFile . . . . .	237
Variables . . . . .	237
Methods . . . . .	238
Construction and destruction methods . . . . .	238
Accessing methods . . . . .	238
Open and close methods . . . . .	239
Read and write methods . . . . .	239
27 CDecorator. . . . .	241
Introduction . . . . .	241
Heritage . . . . .	241
Using CDecorator . . . . .	241
Variables . . . . .	241
Global variable . . . . .	241
Instance variables . . . . .	242
Methods . . . . .	242
28 CDesktop . . . . .	243
Introduction . . . . .	243
Heritage . . . . .	243
Using CDesktop . . . . .	243

Variables . . . . .	243
Global variable . . . . .	243
Instance variables . . . . .	244
Methods . . . . .	244
Construction and destruction methods . . . . .	244
Appearance methods . . . . .	244
Mouse methods . . . . .	245
Window methods . . . . .	247
Accessing methods . . . . .	248
Calibration methods . . . . .	248
Cleanup method . . . . .	248
 29 CDirector . . . . .	 249
Introduction . . . . .	249
Heritage . . . . .	249
Using CDirector. . . . .	249
Variables . . . . .	250
Global variable . . . . .	250
Instance variables . . . . .	250
Methods . . . . .	250
Creation and destruction . . . . .	250
Accessing method . . . . .	251
Command methods . . . . .	251
Appearance methods . . . . .	252
Window methods . . . . .	253
Change notification method . . . . .	254
 30 CDirectorOwner . . . . .	 255
Introduction . . . . .	255
Heritage . . . . .	255
Using CDirectorOwner . . . . .	255
Variables . . . . .	256
Methods . . . . .	256
Creation and destruction . . . . .	256
Insertion and deletion methods . . . . .	256
Appearance methods . . . . .	256
 31 CDocument . . . . .	 259
Introduction . . . . .	259
Heritage . . . . .	259
Using CDocument . . . . .	259
Variables . . . . .	261
Global variables . . . . .	261
Instance variables . . . . .	261



Methods	262
Construction and destruction methods	262
Command methods	262
Appearance Methods	264
File Creation	264
Printing Methods	265
Filing Methods	266
Undo methods	267
Class Resources	267
<b>32 CEditText</b>	<b>269</b>
Introduction	269
Heritage	269
Using CEditText	269
Variables	270
Methods	270
Construction and destruction methods	270
Mouse and Keystrokes methods	271
Command methods	271
Display methods	271
Text specification methods	272
Text characteristics methods	273
Calibrating methods	275
Printing methods	276
Cursor methods	276
<b>33 CEnvironment</b>	<b>279</b>
Introduction	279
Heritage	279
Using CEnvironment	279
Variables	279
Methods	279
<b>34 CError</b>	<b>281</b>
Introduction	281
Heritage	281
Using CError	281
Variables	281
Global variable	281
Instance variables	281
Methods	282
Error reporting methods	282
Functions	282
Class resources	283
<b>35 CFile</b>	<b>285</b>
Introduction	285
Heritage	285
Using CFile	285

Variables . . . . .	286
Methods . . . . .	286
Construction and destruction methods . . . . .	286
Specifying methods . . . . .	286
Open and close methods . . . . .	287
Accessing methods . . . . .	287
Filing methods . . . . .	287
<b>36 CFWDesktop . . . . .</b>	<b>289</b>
Introduction . . . . .	289
Heritage . . . . .	289
Using CFWDesktop . . . . .	289
Variables . . . . .	290
Methods . . . . .	290
Construction and destruction methods . . . . .	290
Appearance methods . . . . .	290
Mouse methods . . . . .	291
Window methods . . . . .	291
Cleanup method . . . . .	292
<b>37 CGridSelector . . . . .</b>	<b>293</b>
Introduction . . . . .	293
Heritage . . . . .	293
Using CGridSelector . . . . .	293
Variables . . . . .	293
Methods . . . . .	294
Construction methods . . . . .	294
Drawing methods . . . . .	294
Accessing methods . . . . .	295
<b>38 CList . . . . .</b>	<b>297</b>
Introduction . . . . .	297
Heritage . . . . .	297
Using CList . . . . .	297
Variables . . . . .	297
Methods . . . . .	297
Construction methods . . . . .	297
Insertion and deletion methods . . . . .	298
Ordering methods . . . . .	298
Membership methods . . . . .	299
<b>39 CMBBarChore . . . . .</b>	<b>301</b>
Introduction . . . . .	301
Heritage . . . . .	301
Using CMBBarChore . . . . .	301
Variables . . . . .	301
Methods . . . . .	301

<b>40</b>	<b>CMenuDefProc</b>	<b>303</b>
	Introduction	303
	Heritage	303
	Using CMenuDefProc	303
	Creating the stub MDEF	303
	Writing the CMenuDefProc subclass	304
	Using your CMenuDefProc subclass	304
	Examples of MDEF objects	305
	Variables	306
	Methods	306
	Functions	308
<b>41</b>	<b>CMouseEvent</b>	<b>309</b>
	Introduction	309
	Heritage	309
	Using CMouseEvent	309
	Defining a mouse task	309
	Using the mouse task	310
	Variables	311
	Methods	311
	Construction and destruction methods	311
	Mouse tracking methods	311
	Class resources	312
<b>42</b>	<b>CObject</b>	<b>313</b>
	Introduction	313
	Heritage	313
	Using CObject	313
	Variables	313
	Methods	313
	Creation and destruction	314
<b>43</b>	<b>CPane</b>	<b>315</b>
	Introduction	315
	Heritage	315
	Using CPane	315
	Coordinate Systems in Panes	316
	Drawing in Panes	318
	Variables	320
	Methods	321
	Construction/Destruction methods	321
	Accessing methods	323
	Appearance methods	325
	Size and location methods	326
	Adapting methods	327
	Drawing methods	328
	Printing methods	329
	Calibration methods	330
	Cursor method	331
	Coordinate transformation methods	331



<b>44</b>	<b>CPaneBorder</b>	<b>335</b>
	Introduction . . . . .	335
	Heritage . . . . .	335
	Using CPaneBorder . . . . .	335
	Variables . . . . .	337
	Methods . . . . .	338
	Creation and destruction . . . . .	338
	Accessing methods . . . . .	338
	Drawing method . . . . .	339
	Calibration method . . . . .	340
<b>45</b>	<b>CPaneMDEF</b> . . . . .	<b>341</b>
	Introduction . . . . .	341
	Heritage . . . . .	341
	Using CPaneMDEF . . . . .	341
	Variables . . . . .	342
	Methods . . . . .	342
	Construction and destruction methods . . . . .	342
<b>46</b>	<b>CPanorama</b> . . . . .	<b>345</b>
	Introduction . . . . .	345
	Heritage . . . . .	345
	Using CPanorama . . . . .	345
	Variables . . . . .	346
	Methods . . . . .	346
	Construction and destruction methods . . . . .	346
	Accessing methods . . . . .	347
	Calibrating methods . . . . .	349
	Scrolling methods . . . . .	349
	Printing methods . . . . .	350
<b>47</b>	<b>CPatternGrid</b>	<b>351</b>
	Introduction . . . . .	351
	Heritage . . . . .	351
	Using CPatternGrid . . . . .	351
	Variables . . . . .	353
	Methods . . . . .	353
	Construction methods . . . . .	353
	Drawing methods . . . . .	354
	Accessing methods . . . . .	354
<b>48</b>	<b>CPictFile</b> . . . . .	<b>355</b>
	Introduction . . . . .	355
	Heritage . . . . .	355
	Using CPictFile . . . . .	355
	Variables . . . . .	355
	Methods . . . . .	355

<b>49</b>	<b>CPicture .</b>	<b>. 357</b>
	Introduction . . . . .	. 357
	Heritage . . . . .	. 357
	Using CPicture . . . . .	. 357
	Variables . . . . .	. 357
	Methods . . . . .	. 358
	Construction/Destruction	. 358
	Appearance methods . .	. 359
	Accessing methods . .	. 359
	Calibration methods . .	. 359
<b>50</b>	<b>CPNTGFile .</b>	<b>. 361</b>
	Introduction . . . . .	. 361
	Heritage . . . . .	. 361
	Using CPNTGFile . . . . .	. 361
	Variables . . . . .	. 361
	Methods . . . . .	. 361
<b>51</b>	<b>CPrinter .</b>	<b>. 363</b>
	Introduction . . . . .	. 363
	Heritage . . . . .	. 363
	Using CPrinter . . . . .	. 363
	Variables . . . . .	. 365
	Methods . . . . .	. 365
	Construction and destruction methods .	. 365
	Accessing methods . . . . .	. 366
	Printing methods . . . . .	. 368
<b>52</b>	<b>CRadioControl .</b>	<b>. 371</b>
	Introduction . . . . .	. 371
	Heritage . . . . .	. 371
	Using CRadioControl . . . . .	. 371
	Variables . . . . .	. 371
	Methods . . . . .	. 372
<b>53</b>	<b>CRadioGroupPane .</b>	<b>. 373</b>
	Introduction . . . . .	. 373
	Heritage . . . . .	. 373
	Using CRadioGroupPane . . . . .	. 373
	Variables . . . . .	. 374
	Methods . . . . .	. 374
	Construction and destruction methods .	. 374
	Accessing methods . . . . .	. 375
	Change notification method. . . . .	. 375
<b>54</b>	<b>CResFile .</b>	<b>. 377</b>
	Introduction . . . . .	. 377
	Heritage . . . . .	. 377

Using CResFile . . . . .	377
Variables . . . . .	377
Methods . . . . .	377
<b>55 CRunArray . . . . .</b>	<b>379</b>
Introduction . . . . .	379
Heritage . . . . .	379
Using CRunArray . . . . .	379
Variables . . . . .	380
Methods . . . . .	380
Creation method . . . . .	380
Accessing method . . . . .	380
Insertion and deletion methods . . . . .	380
Membership method . . . . .	381
Summing methods . . . . .	381
Run-handling methods . . . . .	381
<b>56 CScrollBar . . . . .</b>	<b>383</b>
Heritage . . . . .	383
Using CScrollBar . . . . .	383
Variables . . . . .	384
Methods . . . . .	384
Construction and destruction methods . . . . .	384
Accessing methods . . . . .	384
Drawing methods . . . . .	385
Click response methods . . . . .	385
<b>57 CScrollPane . . . . .</b>	<b>387</b>
Introduction . . . . .	387
Heritage . . . . .	387
Using CScrollPane . . . . .	387
Variables . . . . .	388
Methods . . . . .	388
Construction and destruction methods . . . . .	388
Accessing methods . . . . .	389
Scroll bar maintenance methods . . . . .	390
Scroll performance methods . . . . .	390
Functions . . . . .	391
<b>58 CSelector . . . . .</b>	<b>393</b>
Introduction . . . . .	393
Heritage . . . . .	393
Using CSelector . . . . .	393
Variables . . . . .	394
Methods . . . . .	394
Construction and destruction methods . . . . .	394
Mouse methods . . . . .	395
Accessing methods . . . . .	395

<b>59</b>	<b>CSelectorMDEF</b>	<b>399</b>
	Introduction	399
	Heritage	399
	Using CSelectorMDEF	399
	Variables	399
	Methods	399
	Construction and destruction methods	399
<b>60</b>	<b>CSizeBox</b>	<b>401</b>
	Introduction	401
	Heritage	401
	Using CSizeBox	401
	Variables	401
	Methods	401
	Construction and destruction methods	401
	Appearance methods	402
	Class resources	402
<b>61</b>	<b>CStack</b>	<b>403</b>
	Introduction	403
	Heritage	403
	Using CList	403
	Variables	403
	Methods	403
<b>62</b>	<b>CSwitchboard</b>	<b>405</b>
	Introduction	405
	Heritage	405
	Using CSwitchboard	405
	Variables	405
	Methods	406
	Initialization methods	406
	Mouse methods	406
	Key methods	406
	Disk methods	406
	Window event methods	407
	Suspend/Resume methods	407
	Event processing methods	407
<b>63</b>	<b>CTask</b>	<b>411</b>
	Introduction	411
	Heritage	411
	Using CTask	411
	Variables	412
	Methods	412
	Initialization methods	412
	Accessing methods	412
	Action methods	413
	Class resources	413

---

<b>64</b>	<b>CTearChore . . .</b>	<b>415</b>
	Introduction . . .	415
	Heritage . . . . .	415
	Using CTearChore .	415
	Variables . . . . .	415
	Methods . . . . .	415
<b>65</b>	<b>CTearOffMenu .</b>	<b>417</b>
	Introduction . . . . .	417
	Heritage . . . . .	417
	Using CTearOffMenu .	417
	Variables . . . . .	418
	Methods . . . . .	418
<b>66</b>	<b>CTextEditTask . . . . .</b>	<b>421</b>
	Introduction . . . . .	421
	Heritage . . . . .	421
	Using CTextEditTask . . . . .	421
	Variables . . . . .	422
	Methods . . . . .	422
	Creation and destruction methods .	422
	Accessing method . . . . .	423
	Action methods . . . . .	423
	Internal methods . . . . .	424
<b>67</b>	<b>CTextStyleTask.</b>	<b>427</b>
	Introduction . . . . .	427
	Heritage . . . . .	427
	Using CTextStyleTask. .	427
	Variables . . . . .	428
	Methods . . . . .	428
	Construction method	428
	Action methods . .	428
	Internal methods . .	429
<b>68</b>	<b>CTextEnvirons .</b>	<b>431</b>
	Introduction . . . . .	431
	Heritage . . . . .	431
	Using CTextEnvirons .	431
	Variables . . . . .	432
	Methods . . . . .	433
<b>69</b>	<b>CView . . .</b>	<b>435</b>
	Introduction .	435
	Heritage . . .	435

Using CView . . . . .	435
Views and the visual hierarchy . . . . .	435
Views and the chain of command . . . . .	436
Using Balloon Help with views . . . . .	436
Variables . . . . .	437
Methods . . . . .	437
Construction and destruction methods . . . . .	437
Accessing methods . . . . .	438
Appearance methods . . . . .	440
Mouse methods . . . . .	441
Cursor methods . . . . .	442
Subview Management methods . . . . .	445
Class resources . . . . .	447
 70 CWindow . . . . .	 449
Introduction . . . . .	449
Heritage . . . . .	449
Using CWindow . . . . .	449
Variables . . . . .	450
Global variable . . . . .	450
Instance variables . . . . .	450
Methods . . . . .	450
Construction and destruction methods . . . . .	450
Accessing methods . . . . .	452
Appearance methods . . . . .	454
Size and location methods . . . . .	455
Drawing methods . . . . .	456
Mouse methods . . . . .	457
Conversion methods . . . . .	457
 71 TCL Library Routines . . . . .	 459
Introduction . . . . .	459
Toolbox Utilities . . . . .	459
QuickDraw Utilities . . . . .	459
Window Manager Utilities . . . . .	460
Dialog Manager Utilities . . . . .	460
Font Manager Utility . . . . .	461
Keyboard Utilities . . . . .	461
String Utilities . . . . .	461
Operating System Utilities . . . . .	462
Long Coordinate Utilities . . . . .	462
Long point utilities . . . . .	463
Long rectangle utilities . . . . .	464
THINK Class Library Utilities . . . . .	465
Error reporting utility . . . . .	466
Memory allocation utilities . . . . .	466
Memory disposal utilities . . . . .	468

Long Coordinate QuickDraw Utilities . . . .	468
Long rectangle drawing routines. . . . .	469
Long oval drawing routines . . . . .	469
Long rounded rectangle drawing routines	470
Long bit transfer routine . . . . .	471
Long point pen routines . . . . .	471
Long region utility routines . . . . .	471

<b>72 Global Variables . . . . .</b>	<b>473</b>
Introduction . . . . .	473
Global Objects . . . . .	473
Mouse Click Globals . . . . .	474
Cursors . . . . .	475
System Globals . . . . .	475
Error Globals . . . . .	477
Utility Globals . . . . .	477

## **Appendix**

<b>A MacApp and THINK Pascal . . . . .</b>	<b>481</b>
Contents . . . . .	481
What is MacApp? . . . . .	483
Before you begin. . . . .	483
Take your time . . . . .	483
Minimum Requirements . . . . .	484
Memory Requirements . . . . .	484
Running MacApp with MultiFinder . . . .	484
Running MacApp with the Finder . . . .	484
Compatibility . . . . .	484
Installing MacApp Files for THINK Pascal . .	484
Converting MacApp . . . . .	486
What the Pascal Source Converter does .	486
Make sure you have enough disk space .	486
Convert MacApp . . . . .	486
Clean up . . . . .	489
Building Your Seed Projects . . . . .	490
What's in the MacApp Seeds folder? . . .	490
Build the projects. . . . .	491
The compiler variables . . . . .	492
Segmentation . . . . .	493
Using the Seed Projects . . . . .	493
Creating Resource Files for MacApp . . . . .	494
Using SAREz . . . . .	494
Using the prebuilt resource files . . . .	495
Environment Differences . . . . .	496
Toolbox initialization . . . . .	496
Busy cursor. . . . .	496
Segmentation in MacApp . . . . .	496
UObject and instantiation by name . . . .	497



---

Index . . . . .

. 499



# THINK Pascal



## *Getting Started with Objects*

### *Part One*

- 1 Welcome
- 2 Installing the THINK  
Class Library



# Welcome

## 1

**T**his manual is about object-oriented programming in THINK Pascal. It describes the object extensions to Pascal and the THINK Class Library. The THINK Class Library is a collection of classes that define a generic Macintosh application.

### **If you're new to object-oriented programming**

You should read this chapter and Chapter 3, "Object-Oriented Programming." Then read the first part of Chapter 4, "Object Pascal," and follow the tutorial in Chapter 5, "Tutorial: LearnOOP."

### **If you're familiar with object-oriented programming**

Read this chapter, Chapter 4, "Object Pascal," and Chapter 7, "The THINK Class Library."

### **If you know the THINK Class Library**

Be sure to read "What's New in the THINK Class Library" on page 6 in this chapter. Then read Chapter 4, "Object Pascal" to learn about the object extensions to THINK Pascal. You should scan through Chapter 7, "The THINK Class Library," and read Chapter 8, "Exception Handling."

## **Contents**

What's in this Manual . . . . .	5
What's New in the THINK Class Library . . . . .	6
Long coordinates . . . . .	6
Improved error handling . . . . .	7
Dependent objects . . . . .	7
Director owners . . . . .	8
Abstract text . . . . .	8
Compatibility with earlier versions . . . . .	8

## 1 *Welcome*

---



## **What's in this Manual**

This manual is organized in four sections: Getting Started, Object-Oriented Programming in THINK Pascal, The THINK Class Library, and Appendix.

### **Getting Started**

This is the section you're reading. It contains this chapter and instructions for installing the THINK Class Library.

### **Object-Oriented Programming**

This section describes object-oriented programming with THINK Pascal.

Object-Oriented Programming	This chapter is a general introduction to object-oriented programming. You'll learn the terminology and basic concepts of object-oriented programming.
Object Pascal	This chapter describes Object Pascal, the object extensions to THINK Pascal.
Tutorial: LearnOOP	LearnOOP is a hands-on demonstration of the basic concepts of object-oriented programming.
The Class Browser	This chapter shows you how to use the THINK Pascal Class Browser. The browser is a tool that lets you examine your classes and look up definitions of instance variables and methods.

### **The THINK Class Library**

This section contains 66 chapters that describe the THINK Class Library and each class in the library. This section is designed for both THINK C and THINK Pascal programmers.

The THINK Class Library	This chapter introduces you to the class library. It gives you an overview, and tells you how the different parts of the THINK Class Library fit together. Be sure to read this chapter even if you've
-------------------------	--

	used the THINK Class Library before.
Exception Handling	This chapter describes the exception handling mechanism used in the THINK Class Library. The exception handling mechanism lets you handle errors in a consistent and graceful way.
Chapter 9—Chapter 70	These chapters describe each class in the THINK Class Library.
TCL Library Routines	This chapter describes library routines used throughout the THINK Class Library.
Global Variables	This chapter lists all of the global variables that the THINK Class Library uses.

## **Appendix**

This is the section contains an appendix about using THINK Pascal with MacApp 2.0.

## **What's New in the THINK Class Library**

This version of the THINK Class Library is version 1.1. If you've used earlier versions of the THINK Class Library, you'll find that the fundamental model behind the TCL has not changed. The visual hierarchy and the chain of command work just as they did before. This section describes the most important new features of this version of the THINK Class Library and how they affect your existing programs.

### **Long coordinates**

The visual hierarchy now uses 32-bit (long) coordinates. This change means that your panes and panoramas are no longer limited to the QuickDraw coordinate space. You don't have to use 32-bit coordinates. In fact, the default is to make turn long coordinates off. However, this change will require you to modify your programs.

The THINK Class Library uses the types `LongRect` and `LongPt` throughout the CView family. These types are defined like this in C:

```
typedef struct LongPt
{
    long v, h;
} LongPt;

typedef struct LongRect
{
    long top, left, bottom, right;
} LongRect;
```

And like this in Pascal:

```
type
    LongPt = record
        case integer of
            1: ( v, h: longint
                );
            2: ( vh: array[VHSelect] of longint
                );
        end;

    LongRect = record
        case integer of
            1: ( top, left,
                bottom, right: longint
                );
            2: ( topLeft: LongPt;
                botRight: LongPt
                );
        end;
```

Please be sure to read “Working with Panes” on page 77 and the description of `CPane`. To help make the transition easier, you should also read “Long Coordinate Utilities” on page 462.

### **Improved error handling**

The TCL now uses an exception handling mechanism to catch errors. This mechanism lets you separate the normal flow of your code from the error handling code. Most of the classes in the THINK Class Library use the exception handling mechanism. Be sure to read Chapter 8, “Exception Handling.” For a good example of exception handling, look at the `CDataFile` class.

### **Dependent objects**

All of the chain of command classes are now descendants of a new class called `CCollaborator`. This class lets you set up dependencies between ob-

jects, so one object can announce a change to its dependents. Read the chapter on CCollaborator.

## Director owners

In earlier versions of the THINK Class Library, only an application could a director's supervisor. This version introduces a new class, CDirectorOwner, for objects that own directors. CApplication and CDirector are director owners. This class lets you create multi-window documents by creating directors that are supervised by other directors or documents.

## Abstract text

The class CAbstractText defines characteristics of text classes. The class CStaticText has been removed. Instead, you specify whether text is editable, styleable, or selectable. See the chapters on CEditText and CAbstractText.

## Compatibility with earlier versions

Older versions of the classes that have changed substantially in this version of the THINK Class Library are in the Compatibility classes folder.

### Old Class

CBarOwner

CBorder

CColorWindow

CRadioButton

CRadioGroup

CStaticText

CCluster

CList

CFile

CDataFile

### Replaced because...

The functionality that CBarOwner provided is now part of CBartender.

Borders are now handled by CPaneBorder which is not a subclass of CPane.

This functionality is now in CWindow.

The new class CRadioGroupPane is a pane that groups radio controls. CRadioControl replaces CRadioButton.

All text classes are now descendants of CAbstractText. For static text, use CEditText, but specify that it is not editable.

These classes are now descendants of CArray. The old classes are named OCluster and OList.

These classes now use the exception handling mechanism instead of returning error codes. The old classes are named OFile and ODataFile.



# *Installing the THINK Class Library*

---

## *2* ♦

**T**his chapter tells you how to install the THINK Class Library on your disk. The THINK Class Library is a set of classes that implement a generic Macintosh application. Chapter 7, “The THINK Class Library,” talks about the THINK Class Library in detail.

You don’t need to use the THINK Class Library to use Object Pascal. If you just want to learn how to use object-oriented programming with THINK Pascal, read Chapter 3, “Object-Oriented Programming,” and Chapter 4, “Object Pascal.”

### **Contents**

Installation . . . . .	. 11
Building the Starter Project . . . . .	. 12
Installing the TMPL Resources into ResEdit .	. 13

## ◆ 2 *Installing the THINK Class Library*

---



## Installation

Before you install the THINK Class Library on your disk, be sure that you followed the instructions in *THINK Pascal User Manual*, Chapter 2, "Installing THINK Pascal." Also, make sure you have enough space on your hard disk. The files that make up the core of the THINK Class Library take up nearly 1300K of disk space.

---

### Note

You'll want to install the THINK Class Library demos even if you're familiar with the previous version of the THINK Class Library. The demos show you how to use new classes and take advantage of changes in old classes.

---

To install the THINK Class Library, run the self-extracting archive THINK Class Library 1.1.sea, and choose your Development folder as the destination folder. This archive creates a folder called THINK Class Library 1.1 and puts these files and folders in it:

<b>This file or folder...</b>	<b>Contains...</b>
Core classes	The basic classes all applications need.
TCL Libraries	Utility routines.
Control classes	Classes for buttons and pop-up menus.
Dialog classes	Classes for dialogs.
File classes	Classes for specific kinds of files. CFile is in Core classes.
FW/Tearoffs	Classes for floating windows and tear-off menus.
Table classes	Classes for tables, like spreadsheets and lists.
Text classes	Classes for editing and displaying text.
More classes	Classes that don't fall into any of the above categories, like CBit-Map, CStack and CTextEnvirons.
Compatibility classes	Classes from THINK Class Library 1.0 that have been replaced with new classes.

## 2 Installing the THINK Class Library

---

<b>This file or folder...</b>	<b>Contains...</b>
TCL 1.1 doc	Documentation on some parts of the THINK Class Library, including classes for dialogs and tables.
TCL Resources	The standard resources you need to use the THINK Class Library.
TCL TMPLs	A set of TMPL resources that make it easy to create resources for pane initialization. The next set of instructions tell you how to install these TMPLs into ResEdit.

To install the demos for the THINK Class Library, run the self-extracting archive `TCL 1.1 Pascal Demos.sea`, and choose your Development folder as the destination folder. This archive creates a folder called `TCL 1.1 Pascal Demos` and puts folders in it:

<b>This demo in this folder...</b>	<b>Is...</b>
Art Class Demo	A drawing program that uses tear-off menus and floating windows.
NewClassDemo Folder	A demonstration of some of the THINK Class Library's new features, including classes for dialogs, tables, pop-up menus, and styled text editing
Starter Folder	A minimal project that you can use as a starting point for your own project. It just open and closes empty windows.
TinyEdit Folder	A small text editor based on the Starter project.

### Building the Starter Project

You'll use the `Starter.pi` and `Starter.Build.pi` projects as the basis for all your THINK Class Library-based applications. If you build this project now you'll save a lot of time later since THINK Pascal will only need to recompile the new files that you add to the project.

THINK Pascal stores the compiled code for all your source files in the project document, so when you duplicate the `Starter Folder` to create your own projects, the THINK Class Library files will be already compiled. Since building can take anywhere from five minutes on a Macintosh IIci to over half an hour on a Macintosh Plus, you can save yourself a lot of time later by

spending a few minutes building the projects now. The built projects require about 850K of disk space.

To build the `Starter.π` and `Starter.Build.π` projects, follow these steps:

1. Open the project `Starter.π`, in the `Starter Folder`.
2. Choose the **Build** command from the **Run** menu. THINK Pascal loads all the libraries and compiles all the source files.
3. Choose **Close Project** from the **Project** menu.
4. Choose **Open Project** from the **Project** menu and open the project `Starter.Build.π`.
5. Choose the **Build** command from the **Run** menu.
6. Choose **Quit** from the **File** menu to return to the Finder.

THINK Pascal uses two projects—one for debugging and one for building the final application. When you build an application, THINK Pascal has to turn the debugging options off. That means that the entire THINK Class Library would need to be recompiled. By using two projects and switching to the `.Build.π` project when you want to build the final application, only your own files will need to be recompiled.

## Installing the TMPL Resources into ResEdit

If you use ResEdit to create and edit your resource files, you should install the TMPL resources from TCL TMPLs into your copy of ResEdit. The TMPL resources are resource editor templates for ResEdit. After you install them, it will be easy for you to create certain THINK Class Library resources with ResEdit. Follow these steps to install the TMPL resources into ResEdit.

1. Make a duplicate copy of ResEdit.
2. Double-click on the duplicate copy of ResEdit you just made.
3. Open the file `TCL TMPLs`.
4. Select the item `TMPL`.
5. Choose **Copy** from the **Edit** menu.
6. Open the original copy of ResEdit.
7. Choose **Paste** from the **Edit** menu.
8. Choose **Quit** from the **File** menu. When ResEdit asks you to confirm the changes, click on the Yes button.
9. Delete your copied version of ResEdit.

ResEdit is included in your THINK Pascal package in the `Resource Utilities.sea` archive. To install it, follow the instructions in the *THINK Pascal User Manual*, Chapter 2, "Installing THINK Pascal."

## ◆ 2 *Installing the THINK Class Library*

---

# THINK Pascal

## *Object-Oriented Programming*

### *Part Two*

- 3 Object-Oriented Programming
- 4 Object Pascal
- 5 Tutorial: LearnOOP
- 6 The Class Browser





# Object-Oriented Programming 3 ♦

**O**bject-oriented programming is not hard to learn. In fact, the hardest thing about object-oriented programming is unlearning what you already know about procedural (traditional) programming. To make object-oriented programming easier to learn, this chapter uses examples and comparisons to procedural programming. The examples should make some concepts more concrete, and the comparisons to procedural programming will help you relate what you're learning to something you already know.

## Contents

Overview . . . . .	. 19
Objects and Messages . . . . .	. 19
Classes . . . . .	. 20
Inheritance and Polymorphism . . . . .	. 21
Objects and the Macintosh Interface . . . . .	. 22
Working with Objects. . . . .	. 23
What classes should I define? . . . . .	. 23
When should I create a subclass? . . . . .	. 23
What should be a method? . . . . .	. 24
When should I use procedural programming? . . . . .	. 24
Where to Go Next . . . . .	. 24

## 3 *Object-Oriented Programming*

---

## Overview

The basic difference between procedural and object-oriented programming is in the way the two disciplines treat data and action. In procedural programming, data and action are two separate things. You define your data structures, and then you define some routines to operate on them. For each data structure you define, you need a new set of routines.

In object-oriented programming, action and data are closely coupled. When you define your data—your objects—you also define their actions. Instead of a set of routines that do something to data, you have a set of objects interacting with each other.

## Objects and Messages

An **object** is an entity that contains some data and an associated set of actions that operate on the data. To make an object perform one of its actions, you send it a **message**.

For example, you might create an object that represents a rectangle. Its data contains the locations of the rectangle's four corner points, and its actions might include drawing, erasing, and moving. To draw a rectangle, you send the rectangle a draw message.

---

### Note

For the time being, don't worry about *how* to send a message to an object. The important thing is to start thinking of sending messages as requesting an object to do something.

---

Contrast this with the way you'd do the same thing in procedural programming. First you define a record that represents the four corners of the data structures, and then you define three routines to draw, to erase, and to move a rectangle. Each of the routines would take a rectangle data structure as an argument.

So the first big advantage of object-oriented programming over procedural programming is that you can keep the routines that operate on a data structure together with the data structure they're operating on. This "keeping together" is called **encapsulation**.

### Classes

Encapsulation is a minor advantage of object-oriented programming. A more significant advantage is that objects can inherit data and behavior from other objects.

Every object belongs to a **class**, which defines the implementation of a particular kind of object. A class describes the object's data and the messages it responds to.

Classes are very much like record declarations. You define the private data for the class the same way as you would the fields of a record. In classes, though, the fields are called **instance variables**. Each instance of a class, each object, has its own instance variables just as each variable of a record type has the same fields.

When you send an object a message, it invokes a routine that implements that message. These routines are called **methods**. The class definition includes the method implementations.

One important thing to keep in mind is that message and method are not the same thing. A message is what you send to an object. How an object responds to a message is the method.

You can think of a class as a template for creating objects. It describes an object's data and the messages it responds to. An object is called an **instance** of a class. You can also say that an object is a **member** of a class.

The rest of this chapter uses pictures like this to represent classes. This is what the rectangle class in the example above looks like:

#### **Class**

Rectangle

#### **Instance variables**

top  
left  
bottom  
right

#### **Messages**

Draw  
Erase  
Move

#### **Methods**

draw line from point to point  
erase lines  
offset points



## Inheritance and Polymorphism

You can define a class in terms of an existing class. The new class is called the **subclass**, and the existing class is called the **superclass**. A class without a superclass is said to be a **root class**. A subclass inherits all the instance variables and methods of its superclass. Subclasses can define additional instance variables and methods. Subclasses can also **override** methods defined by the superclass.

Overriding a method means that the subclass responds to the same message as its superclass, but it uses its own method to respond to the message.

For example, suppose you want to create a class to represent employees. Two of the instance variables might be the person's name and birth date. You might define three methods, `GetName`, `GetAge` and `GetWkPay`, to return the person's name, to calculate the person's age from the birth date, and to return the person's weekly salary.

**Class**

Employee

**Superclass**

none

**Instance variables**

Name

Birthdate

**Messages**

`GetName`

`GetAge`

`GetWkPay`

**Methods**

return Name

return (Today - Birthdate)

return 0

Now you can use the `Employee` class to create two new subclasses: `HourlyEmployee` and `ExemptEmployee`. Both of these classes inherit all the instance variables from the `Employee` class. Each class, however, defines a new instance variable to store the salary, and each class overrides the `GetWkPay` method to return the appropriate weekly salary.

This is what the `HourlyEmployee` class looks like.

**Class**

`HourlyEmployee`

### 3 Object-Oriented Programming

---

#### **Superclass**

Employee

#### **Instance variables**

HourlySalary

#### **Messages**

GetWkPay

#### **Methods**

return (HourlySalary \* 40)

And this is what the ExemptEmployee class looks like.

#### **Class**

ExemptEmployee

#### **Superclass**

Employee

#### **Instance variables**

YearlySalary

#### **Messages**

GetWkPay

#### **Methods**

return (YearlySalary / 52)

As long as your object is a member of any of the Employee classes, you can send it GetName, GetAge, and GetWkPay messages, and the object will respond properly.

This ability to send the same message to objects of different classes is called **polymorphism**. If you need to create a new kind of employee (a salesperson on commission), all you need to do is define a subclass of Employee and override the GetWkPay method. Any routine that sends GetWkPay messages to employee objects would still work.

Consider for a moment how you would do the same thing in procedural programming. Your employee record would need a flag field to specify whether it was an HourlyEmployee or an exempt employee. Then, your GetWkPay function would check this field and calculate the weekly salary appropriately. If you added a salesperson, you would have to change the GetWkPay function to handle the salary calculation for the new kind of employee.

## Objects and the Macintosh Interface

The Macintosh interface lends itself to object-oriented programming. For instance, in the Finder, the effect of the **Open** command depends on the icons you've selected. If you select a folder, it opens the folder. If you select an ap-

plication it launches the application. If you select a document, it launches the application and tells it to open the selected document. What you're doing is sending an Open message to different Finder objects. Each object uses its own Open method to carry out the request.

Think of the visual entities on the Macintosh screen—windows, controls, the menu bar, icons—as objects. They're things that are waiting to do something. Think of your actions—clicking, dragging, typing—as messages. You're sending messages to the Macintosh objects.

The THINK Class Library, included with your THINK Pascal package, provides a set of classes that let you work with these Macintosh elements from an object-oriented point of view. You can read more about the THINK Class Library in Chapter 7, "The THINK Class Library."

## **Working with Objects**

If you've never worked with object-oriented programming, this section will give you some general guidelines for designing classes. As you get more comfortable with object-oriented programming, you'll build up a collection of classes you can use in any of your applications. If you want to see how extensive a library of classes can be, look at the THINK Class Library.

### **What classes should I define?**

Usually, you define only one root-level class and then define all the other classes as descendants of that class. The advantage of this approach is that you can define behavior that applies to all the objects in your application.

This root-level class is an example of an abstract class. You don't create instances of abstract classes. Instead, you use them to give a family of objects common behavior. The Employee class in the example above is an abstract class. Its function is to provide the common instance variables and methods you'll need for all employees.

In general, you should define a class (or a subclass of your root class) for each concept your application deals with.

### **When should I create a subclass?**

Create a subclass whenever you need to change the behavior of a method or when you need to add more instance variables. If you write a method that does different things depending on the value of some instance variable, it's probably time to create a subclass.

## ◆ 3 *Object-Oriented Programming*

---

### **What should be a method?**

Typically, any action an object can make should be a method. It's considered good practice to provide methods to set and get the values of instance variables rather than allowing other functions to access them directly. If you find yourself passing an object as a parameter to a function to manipulate its instance variables, you should turn the function into a method.

### **When should I use procedural programming?**

Object-oriented programming isn't the answer to all programming problems. Some problems are solved best by "old fashioned" procedural programming. In particular, procedural programming is better for procedures that are highly algorithmic or computationally intensive.

### **Where to Go Next**

The next chapter describes how THINK Pascal implements object-oriented programming. As you'll see, it's a fairly straightforward extension of record declarations.

If you're not familiar with object-oriented programming, scan through Chapter 4, then try the "LearnOOP" tutorial in Chapter 5.

Chapter 7 is about the THINK Class Library, a collection of classes you can use to implement Macintosh applications.

Chapters 9–70 describe the main classes of the THINK Class Library in detail. Though these chapters are meant for reference, you might want to leaf through some of the class descriptions to get a sense of how the THINK Class Library is put together.



# Object Pascal 4

**T**his chapter shows you how to use the object-oriented extensions to THINK Pascal. You'll learn how to declare a class, how to declare an object, and how to define and call a method. The object-oriented extensions to THINK Pascal are based on Object Pascal defined in *Object Pascal Report* by Larry Tesler, Apple Technical Report No. 1, Apple Computer 1985.

## What you should know

Before you read this chapter, you should know about object-oriented programming. You should know about classes, instances, instance variables, and methods. If you need to learn about these things, look at Chapter 3. THINK Pascal implements objects as handles, so you should be familiar with the Macintosh Memory Manager. If you need to learn about the Memory Manager, see *Inside Macintosh II*, Chapter 1, "The Memory Manager."

## Contents

Overview . . . . .	. 27
Declaring a Class . . . . .	. 27
Declaring and Using Objects . . . . .	. 29
Creating and deleting objects . . . . .	. 30
Referring to instance variables . . . . .	. 30
Class membership . . . . .	. 30
Defining and Using Methods . . . . .	. 31
Referring to the current object . . . . .	. 31
Calling a method . . . . .	. 32
Calling an inherited method within a method . . . . .	. 32
Using TObject . . . . .	. 33
Tips and Techniques . . . . .	. 35
Organizing your classes . . . . .	. 35
Objects and segments . . . . .	. 35
Objects and the Macintosh . . . . .	. 35
Keywords . . . . .	. 37
Summary . . . . .	. 37

## ◆ 4 *Object Pascal*

---



## Overview

The Object Pascal extensions are built into THINK Pascal. You don't have to add any files to your project. You can use Object Pascal in applications and in desk accessories. When you use Object Pascal in a desk accessory or a code resource, be sure that you check the Multi-Segment option in the **Set Project Type...** dialog.

To use objects, you need to declare a **class**. A class declaration describes the properties of a class and names its instance variables and methods. A class declaration doesn't allocate any space; it just lets the compiler know what a class looks like.

---

### Note

Some implementations of Object Pascal use the term **object type** instead of class and the term **field** instead of instance variable.

---

After you declare a class, you need to define the methods associated with the class. A method is a procedure or function that operates on objects of a particular class. A method definition is almost identical to a procedure or function definition.

To use objects in a program, you declare a variable, called an **object reference**, that will point to the actual object. Then you use the new procedure to allocate memory for the object.

Once you've created an object, you can access its instance variables and send it messages. When you're finished using an object, you use the dispose procedure to expunge it from memory.

The sections below talk about each of these steps in detail.

## Declaring a Class

A class declaration is like a record declaration. You give the class a name and specify its superclass, then you declare the instance variables and methods like this:

```
class = object (superclass)
    instance variable declarations
    method declarations
end;
```

The class declaration must specify a class name. If the class has no superclass, it's called a **root class**. To specify that a class is a root class, leave out the superclass name after the `object` keyword.

After the class name, declare the instance variables and methods. To declare an instance variable, use the same syntax you use to declare a field in a record. To declare a method, use the same syntax you use to declare a procedure or function.

Here are some class declarations:

```
Window = object theWindow: WindowPtr;
    growRect: Rect;
    dragRect: Rect;

    procedure Init;
    procedure Destroy;
    procedure Hit(where: Point);
    procedure Drag(where: Point);
    procedure Draw;
end;

MyWindow = object (Window)
    windowData: Handle;
    subWindow: ToolWindow;

    procedure Hit(where: Point);
    override;
    procedure Draw;
    override;
end;
```

*You might want to make all your objects be descendants of TObject. See "Using TObject" on page 33.*

The first class declaration specifies a root class called `Window`. It has three instance variables and five methods. The second class declaration specifies a subclass of `Window` called `MyWindow`. The second class inherits all of the instance variables that `Window` declares, as well as all of its methods.

`MyWindow` also declares an instance variable of its own and overrides the `Hit` and `Draw` methods.

When you override a method in Object Pascal, you must follow the method declaration with the word `override`. If you don't, you will get an error. Object Pascal assumes that all method declarations introduce new methods. If you don't use the word `override`, Object Pascal thinks that you're trying to introduce a new method with the same name as an inherited method.

An instance variable in a class declaration can be an object reference. The `subWindow` instance variable in the example above is of class `ToolWindow`,



which hasn't been defined yet. It may not even be defined in this file. A forward reference like this is always allowed in a class declaration.

---



---

#### Warning

A forward reference to a class *must* resolve to a class. It should not resolve to any other type

---



---

## Declaring and Using Objects

Once you've defined a class, you can use its name to define an **object reference**. An object reference is just like a pointer, but you don't use the ^ symbol. This is how you define an object reference:

```
var
    object: class;
```

*You might want to think of object references as pointers to objects.*

An object reference definition doesn't allocate memory for the object. To allocate memory for an object, you use the new procedure described later.

THINK Pascal actually implements object references as handles. You can use any of the Macintosh Memory Manager calls that work on handles on objects if you type cast the object to a generic handle. The section "Objects and the Macintosh" on page 35 tells you why you might want to do this.

Two objects of different classes are type-compatible if one class is an ancestor of the other. For assignment, this works in only one direction: an object may be assigned to an ancestor, but not to a descendant.

For example, assume that TCircle is a subclass of the class TShape and that aCircle is an object reference of type TCircle and aShape is an object reference of type TShape. This assignment is OK:

```
aShape := aCircle
```

since the class of aShape is an ancestor of the class of aCircle. This assignment is *not* OK:

```
aCircle := aShape
```

since the class of aCircle is not an ancestor of the class of aShape. This assignment may be OK:

```
aCircle := TCircle(aShape)
```

Type casting makes a `Shape` appear to be of class `TCircle`, **but** if range checking is on, you'll get an error at run time if a `Shape` isn't really of class `TCircle`.

### Creating and deleting objects

Before you can use an object, you must create it to allocate space for it in the heap. When you're done with an object, you'll probably want to delete it to reclaim its space. The standard procedures `new` and `dispose` create and delete objects.

To create an object use the `new` procedure like this:

```
var
  myObject: TClass;
begin
  ...
  new(myObject);
  ...
end;
```

*If your object is a descendant of `TObject`, you can send it a `Free` message to delete it. See "Using `TObject`" on page 33.*

To delete an existing object, use the procedure `dispose` like this:

```
dispose(myObject);
```

These two procedures are defined in `Runtime.lib`.

### Referring to instance variables

To refer to an instance variable of an object, you use the same syntax as when you refer to a field of a record:

*object.member*

Since an object reference is implemented as a handle, THINK Pascal does the necessary dereferencing for you.

As a rule, only the methods of a class should access an object's instance variables. Your class declaration should provide methods for getting and setting the values of the most important instance variables.

### Class membership

To test whether an object is a member of a class, use the `member` function:

```
member(anObject, aClass);
```

This function returns `TRUE` if the object belongs to the specified class or if the class is an ancestor of the object's class.



## Defining and Using Methods

You define a method the same way you define a procedure or function, except that you include the class name before the method name like this:

```
procedure class.method(parameter declarations) ;  
begin  
    procedure body  
end;
```

or like this:

```
function class.method(parameter declarations) : return-type;  
begin  
    function body  
end;
```

If you use a function to implement a method, use the name of the method without the class name to set the return value.

### Referring to the current object

Within a method, the compiler supplies an implicit declaration so you can refer to the object that received the message:

```
var  
    self: class;
```

The symbol `self` refers to the object receiving the message. Within a method definition, you can omit `self` when you refer to an instance variable or when you call a method in the same class as `self`. Think of the body of a method as being enclosed like this:

```
with self do  
begin  
    method body  
end;
```

---

---

#### Warning

Because of this implicit `with` statement, it is never safe to take the address of an instance variable. See "Objects and the Macintosh" on page 35.

---

---

## 4 Object Pascal

Here's what a method declaration for the Window class above might look like:

```
procedure Window.Drag (where: Point);
var
    myDragRect: Rect;
begin
    myDragRect := dragRect;
    DragWindow(theWindow, where, myDragRect)
end;
```

In this example, `dragRect` and `theWindow` are both instance variables. You could refer to them as `self.dragRect` and `self.theWindow`, but since the body of the method is surrounded in an implicit `with` statement, you don't have to use `self`.

---

### Note

If you declare a local variable with the same name as an instance variable, you won't be able to refer to it because of the implicit `with` statement.

---

### Calling a method

To call a method (to send a message to an object), you use this syntax:

```
object . method(parameters)
```

**A monomorphic method is a method that does not override another method and is never overridden itself.**

When you send a message to an object, THINK Pascal generates a call to a run-time dispatcher to find out which method it should use. If the method is monomorphic, the linker bypasses the dispatcher and generates a call directly to the method.

### Calling an inherited method within a method

Within a method definition, you may want to use the superclass's method. For example, to add functionality to a method, you use the superclass's method, then add more code to augment the behavior. This is how you call a superclass's method:

```
inherited method(args);
```

---

### Note

When you call an inherited method, THINK Pascal always bypasses the message dispatcher and generates a direct call.

---



## Using TObject

Your THINK Pascal package includes a file called `ObjIntf.p` which contains the definition of a root class called `TObject`. This class is an abstract class that implements methods for copying and deleting any object. You might want to make all your classes descendants of `TObject`.

This is what the declaration of the `TObject` class looks like:

```
type TObject = object
  function ShallowClone:TObject: TObject;
    { Lowest level method for }
    { copying an object; }
    { Should not be overridden }
    { except in very unusual cases. }
    { Simply calls HandToHand }
    { to copy the object data. }
  function Clone:TObject: TObject;
    { Defaults to calling ShallowClone; }
    { Can be overridden to copy objects }
    { referred to by fields. }
  procedure ShallowFree:TObject;
    { Lowest level method for }
    { freeing an object; }
    { Should not be overridden }
    { except in very unusual cases. }
    { Simply calls DisposHandle }
    { to free the object data. }
  procedure Free:TObject;
    { Defaults to calling ShallowFree; }
    { Can be overridden to free }
    { objects referred to by fields. }
end;
```

If you make all your classes descendants of `TObject`, you can send any object a `Clone` message to copy it or a `Free` message to delete it.

---

### Warning

If you send an object a `Free` message to delete it, do not use the `dispose` procedure on the same object. If you do, your program will crash. Conversely, if you clone an object, you don't need to use the `new` procedure.

---

## ◆ 4 Object Pascal

---

Your own classes can override the `Clone` and `Free` methods to copy or to free memory that your objects allocate. For instance, suppose you have a class that allocates a table as a handle:

```
FixedTable = object (TObject)
  theTable: Handle;
  numItems: integer;

  procedure Init;
  function Clone: FixedTable;
  override
  procedure Free;
  override
  ...
end;
```

Assume that the `Init` method allocates a block of 1K bytes like this:

```
procedure FixedTable.Init;
begin
  theTable := NewHandle(1024);
  numItems := 0;
end;
```

Your `Clone` and `Free` methods might look like this:

```
function FixedTable.Clone: FixedTable;
var
  objCopy: FixedTable;
  tableCopy: Handle;
begin
  objCopy := nil;

  { HandToHand may move memory, }
  { and it takes a VAR parameter, }
  { so make a copy of the handle. }
  tableCopy := Handle(theTable);

  { now see if there's enough room }
  { for the table. }
  if HandToHand(tableCopy) = noErr then
  begin
    { now copy the object if possible }
    objCopy := inherited Clone;
    if objCopy <> nil then
      { assign the table if the copy is OK }
      objCopy.theTable := tableCopy
    else
      { otherwise get rid of the table }
      DisposHandle (tableCopy);
  end;
end;
```



```

    { return the result }
    Clone := objCopy;
end;

procedure FixedTable.Free;
begin
    { Release any storage that }
    { this object allocated }
    DisposHandle(theTable);

    { Now use the inherited method }
    { to delete this object }
    inherited Free;
end;

```

Note that the Clone method is careful about not passing a VAR parameter to a routine that may move memory.

## Tips and Techniques

This section describes some general tips to make writing object-oriented programs easier. It describes how to organize your classes, how to initialize objects easily, and how to make sure that your objects behave correctly under the Macintosh Memory Manager.

### Organizing your classes

It's a good idea to isolate classes to make them easy to use and reuse. The best way to do this is to make each class a unit. If you have several classes that are all related, you can put them all into one large unit. Another way to organize your classes is to put the class declarations in one or two interface files and to put the implementation for each class in a separate file.

*To learn about units, see Chapter 10, "Units and Libraries," in the THINK Pascal User Manual.*

For example, the THINK Class Library included with your THINK Pascal package contains about 60 core classes. The class definitions are in the file TCL.p. The implementations for each of the methods are in separate files.

### Objects and segments

THINK Pascal creates two directive entries, «%\_MethTables» and «%\_SelfProcs», that contain the routines and data for the message dispatcher. By default, these directive entries are in the main segment. If you're using a lot of objects, it's a good idea to move each of these two segments into their own segment and to set their attributes to Locked.

*To learn more about segmentation and setting attributes, see Chapter 7, "Working with Projects," of the THINK Pascal User Manual.*

### Objects and the Macintosh

THINK Pascal implements objects as handles. You can pass an object reference to virtually any Memory Manager routine that works on handles.

## 4 Object Pascal

An important side effect of this implementation is that any reference to an instance variable is implicitly a handle reference.

---

**Warning**

Your program should not rely on the addresses of instance variables, particularly in calls to Toolbox routines that may move memory.

---

Look at the definition for the Drag method for the Window class declared earlier in this chapter:

```
procedure Window.Drag(where: Point);
var
  myDragRect: Rect;
begin
  myDragRect := dragRect;
  DragWindow(theWindow, where, myDragRect);
end;
```

The more obvious definition would be:

```
procedure Window.Drag(where: Point);
begin
  DragWindow(theWindow, where, dragRect);
end;
```

Because the third argument to DragWindow is a var parameter, dragRect refers to the address of the instance variable self.dragRect. Since the DragWindow routine may move memory—including the handle self—you can't rely on the address of dragRect being correct when DragWindow needs it.

The first definition, which copies the instance variable into a local variable, avoids the problem since the address of a local variable never changes within the method body.

It is OK to assign values to instance variables, even when the expression on the right hand side may move memory. For instance, consider this class declaration:

```
FixedTable = object theTable: Handle;
               numItems: integer;

               procedure Init;
               ...
               end;
```



Assume that the `Init` method allocates a block of 1K bytes like this:

```
procedure FixedTable.Init;  
begin  
    theTable := NewHandle(1024);  
    numItems := 0;  
end;
```

In this case, the assignment to `theTable` is safe. THINK Pascal calls `NewHandle` before it resolves the reference to `theTable`.

Of course, you can use the Memory Manager routines `HLock` and `HUnlock` and `HGetState` and `HSetState` to keep the handle `self` from moving around. Keep in mind that using the Memory Manager will slow your program down.

---

**Note**

If you use the Memory Manager routines to lock and unlock an object, be sure to coerce it to the generic `Handle` type like this: `HLock(Handle(self))`.

---

You can use the Macintosh Toolbox routine `GetHandleSize` to find out how big an object is. The size of an object is the sum of the sizes of all of its instance variables, including its ancestors' instance variables, plus 2 bytes (4 bytes if you're using the Far Code option).

---

**Warning**

Your program should not rely on the size of an object or the representation of an object in memory.

---

**Keywords**

Object Pascal reserves the keywords `object` and `inherited`. The word `override` is not a keyword, but THINK Pascal interprets it specially in context.

**Summary**

This is a summary of the functions and procedures you use to work with objects. These routines create objects, destroy objects, and test membership.

```
procedure new(objectReference) ;
```

Create a new instance of an object with the class of the variable *objectReference*.

## ◆ 4 Object Pascal

---

`procedure dispose(objectReference) ;`

Delete the specified object. This procedure releases the memory that the object occupies. It does not release any memory that an instance variable might point to.

`function member(objectReference, class) : Boolean;`

Return TRUE if *objectReference* is a member of *class*. An object is a member of a class if it is of type *class* or if *class* is a superclass of the object's actual class.

If your classes are descendants of TObject, you can use these methods to copy and delete objects.

`function TObject.Clone: TObject;`

Return a new object that is a copy of the object receiving a Clone message. For example:

```
var
  myObject: MyClass;
  { MyClass is a descendant of TObject }
  anObject: MyClass;
begin
  new(myObject);
  { Create the object with new }

  some operation on myObject

  anObject := myObject.Clone;
  { Create a copy of myObject }
end;
```

`procedure TObject.Free;`

Delete the object that receives the Free message. Don't call dispose on an object that you've freed with this message. For example:

```
var
  myObject: MyClass;
  { MyClass is a descendant of TObject }
begin
  new(myObject); { Create the object }
  myObject.Free; { Delete the object }
end;
```

# Tutorial: LearnOOP

---

## 5

**L**earnOOP is a shape-drawing program that introduces you to the basics of object-oriented programming. Each of the shapes that LearnOOP draws is an object. You'll see how the routines that operate on objects are encapsulated in the class declaration. You'll learn how dynamic binding and how inheritance work. Then you'll define a new class yourself.

If you already know object-oriented programming, you don't have to read this chapter. You should skip on to the next chapter to learn about the class browser.

### Before you begin

Make sure that you've installed THINK Pascal on your Macintosh as described in Chapter 2 of the *THINK Pascal User Manual*. You don't need to install the THINK Class Library to run this tutorial.

### What you should know

This tutorial assumes that you know how THINK Pascal works. You should know how to open a project, how to run a project, and how to use the editor. If you need to learn how to do any of these things, see the tutorial in Chapter 3 of your *THINK Pascal User Manual*.

## Contents

What Does LearnOOP Do? .	. 41
How Does LearnOOP Work? . . . . .	. 43
How Do Objects Know How to Draw Themselves? .	. 44
How Do You Create an Object? . . . . .	. 49
How Do You Define a New Class?	. 50
But How Does It Know? .	. 52
Where to Go Next .	. 52

## ◆ 5 *Tutorial: LearnOOP*

---



## What Does LearnOOP Do?

To begin this tutorial, double-click on the `LearnOOP.π` project in the `LearnOOP` folder. If you're already in THINK Pascal, close any existing project, and choose `LearnOOP.π` from the standard file dialog. The project window looks like Figure 5-1.

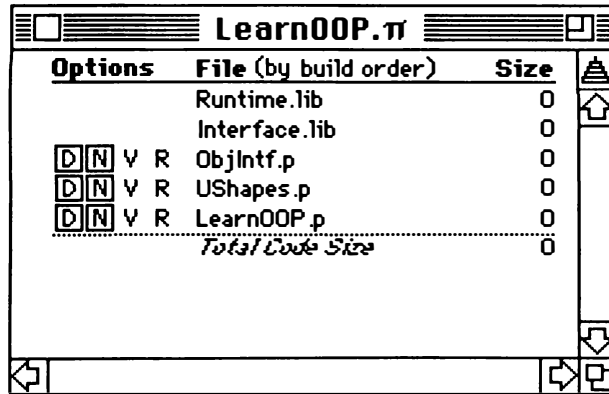


Figure 5-1 The `LearnOOP.π` project

Choose **Go** from the **Run** menu to start the program. THINK Pascal will load the libraries and compile the source files before it starts running. It will take a minute or so.

The `LearnOOP` program uses the Text window to prompt you for something to do, and it draws in the Drawing window. When you run the program, `LearnOOP` asks you if you want to create an Oval or a Rectangle, like in Figure 5-2.

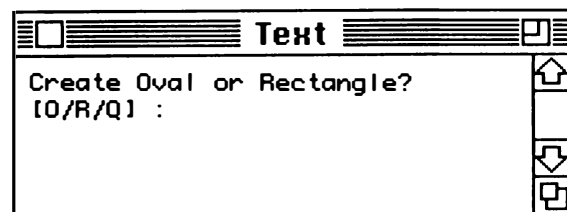
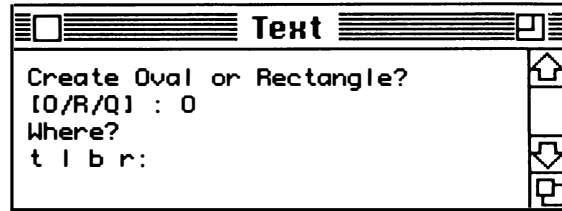


Figure 5-2 Asking for a shape

If you want to draw an oval, type an `O`; if you want to draw a rectangle, type an `R`. To quit the program altogether, type a `Q`. Right now, type an `O`.

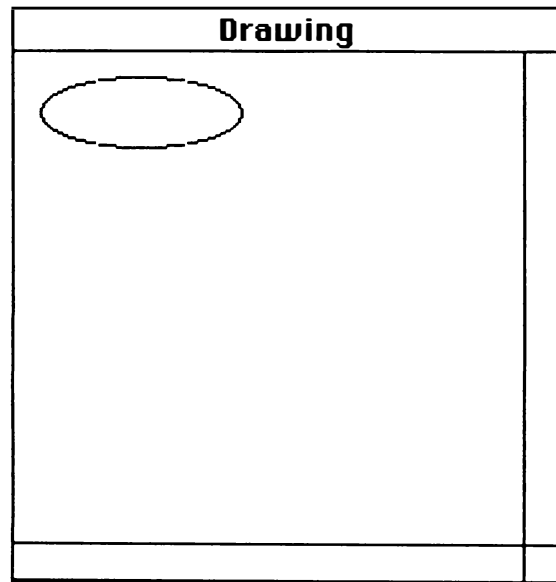
## 5 Tutorial: LearnOOP

After you tell LearnOOP what kind of shape to draw, it asks you to give it the coordinates of the shape, like in Figure 5-2.



**Figure 5-3** Asking for coordinates

The letters `t l b r` represent the coordinates of the top, left, bottom, and right corners of a rectangle that completely encloses the shape. The point 0,0 is at the top left corner of the window. To see how this works, try typing these values: 10 10 40 90. LearnOOP draws the shape in the drawing window like Figure 5-2.



**Figure 5-4** Drawing the shape

After it draws the first shape, LearnOOP asks what kind of shape to draw next. Try a couple of more shapes before you look at how the program works.

## How Does LearnOOP Work?

If LearnOOP is still asking you for a kind of shape, type a Q to make it stop.

Double-click on the file named `LearnOOP.p` in the project window. The program that draws shapes is pretty simple. Look at the main part of the program at the end of the file:

```
{ Main Program }
begin
  ShowText;
  ShowDrawing;

  while ChooseShape(aShape) do
    if aShape = nil then
      writeln('•• I don''t know that shape.')
    else
      begin
        writeln('Where?');
        write('t l b r: ');
        readln(aTop, aLeft, aBottom, aRight);
        aShape.SetBounds(aTop, aLeft, aBottom,
          aRight);
        aShape.Draw
      end;
      writeln('•• FINISHED ••')
    end.
end.
```

The first two procedures just make sure that the Text and Drawing windows are visible.

The next part of the program in the while loop actually does the drawing. The function `ChooseShape` is what asks you for the kind of shape to draw. If you choose to quit, `ChooseShape` returns `FALSE`, and the program ends. If you do choose a shape, `ChooseShape` creates a shape and stores it in the variable `aShape`.

After you choose a shape, the program asks you for the coordinates of the shape and stores them in four variables.

The next two lines are the most interesting lines of the program.

The line

```
aShape.SetBounds(aTop, aLeft, aBottom,
  aRight);
```

sends a `SetBounds` message to the shape `aShape` to give it its coordinates.

The line

```
aShape.Draw;
```

sends aShape a message to draw the shape.

The important thing here is that the program uses the same two lines to set the coordinates and to draw the shapes regardless of the kind of shape. The part of the program that draws the shape doesn't know what kind of shape to draw. It just sends the shape object a Draw message. The object knows how to draw itself.

### How Do Objects Know How to Draw Themselves?

The best way to find out what's going on is to step through the program.

If LearnOOP is asking you for a shape, type Q to make the program end. Then double-click in the project window on the files LearnOOP.p and UShapes.p to open them. If you don't see a little Stop Sign in the lower left corner of the editing windows, choose the **Stops In** command from the **Debug** menu. Next, in the file LearnOOP.p, put a stop sign on the line that sends the Draw message. The window should look like Figure 5-5.

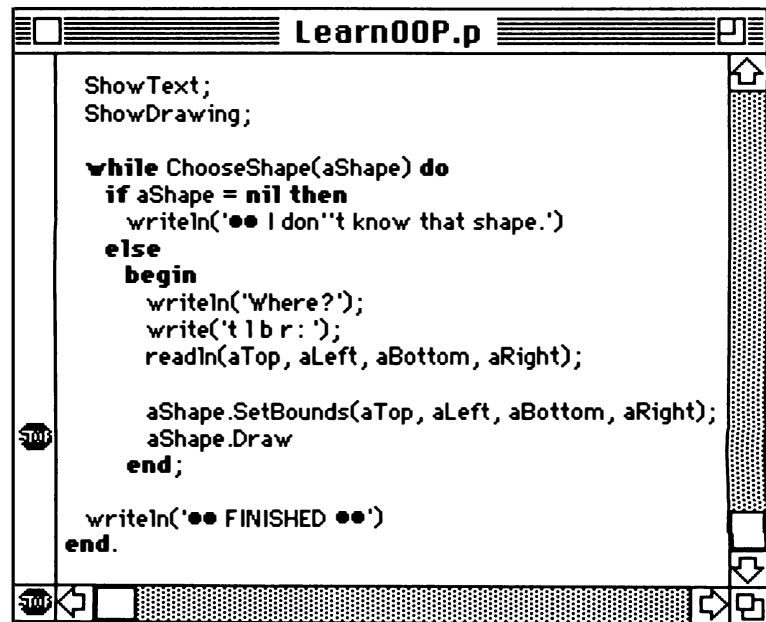


Figure 5-5 Putting a stop sign in LearnOOP.p

Now choose the **Go** command from the **Run** menu.

THINK Pascal starts the program again, and LearnOOP asks you to choose a shape. Type R for rectangle. LearnOOP asks you to type in the bounds for the shape. Type in whatever you like. (If you can't think of anything, try 20 30 45 66.)

As soon as you type the Return key, LearnOOP stops at the line that sends a Draw message to aShape. To find out what's going on, choose the **Step Into** command from the **Run** menu. You'll see the THINK Pascal execution finger point to the beginning of a procedure called `CRectangle.Draw`, like in Figure 5-6.

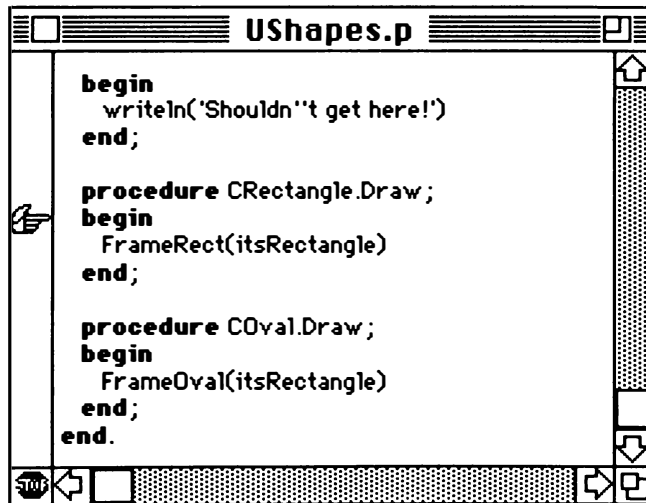


Figure 5-6 Stepping into `CRectangle.Draw`

This procedure is actually a method. In fact, it's the method that handles Draw messages sent to objects of class `CRectangle`. In a moment, you'll see what this class looks like, but first, see what happens when you choose a different kind of shape.

Choose the **Go** command from the **Run** menu to let LearnOOP draw the shape. When it asks you for another shape, type O for oval. Type whatever you want for the location. Because the stop sign is still in, the program stops at the line that sends a Draw message to aShape. Choose the **Step Into**

## 5 Tutorial: LearnOOP

command from the **Run** menu again. This time, you'll see the execution finger at the beginning of another procedure, as in Figure 5-7.

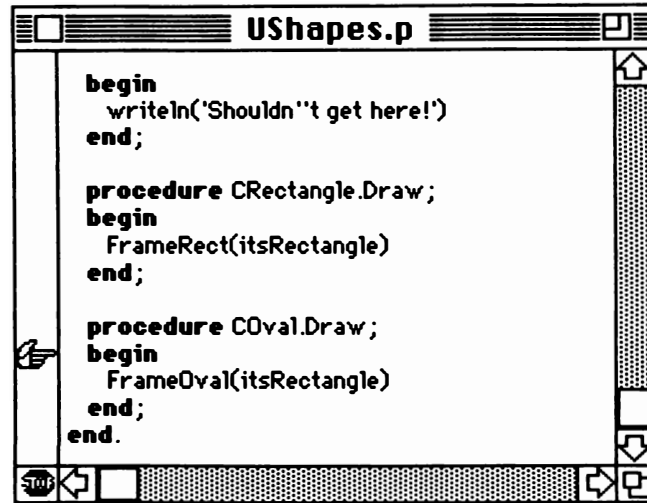


Figure 5-7 Stepping into COval.Draw

The procedure `COval.Draw` is the method that handles `Draw` messages for objects of class `COval`.

The declaration of a class determines how an object will respond to a message. Scroll to the beginning of the file `UShapes.p`. The class declarations for the objects are in the interface part:

```
type
{ CShape is the abstract class that }
{ describes all shapes. All shape classes }
{ are a subclasses of this class. }
CShape = object(TObject)

    { The rectangle that encloses the shape. }
    itsRectangle: Rect;

    { The SetBounds method sets values for }
    { itsRectangle. Most subclasses won't }
    { need to override this method. }
    procedure SetBounds (aTop, aLeft, aBottom,
        aRight: integer);
```



```
{ The Draw method draws the shape. }
{ All subclasses should override }
{ this method. }
procedure Draw;
end;

CRectangle = object (CShape)
    procedure Draw;
    override;
end;

COval = object (CShape)
    procedure Draw;
    override;
end;
```

These declarations say that this file has three classes. The first class, `CShape`, is a generic shape. Every object of class `CShape` has one instance variable: a rectangle called `itsRectangle`. Every object of class `CShape` also implements two methods: `SetBounds` to set the size of the shape, and `Draw` to actually draw the shape.

`CShape` is an abstract class. You use an abstract class as the common ancestor of a family of classes that share certain traits and behaviors. The common trait that all shapes share is that they need a bounding rectangle. The common behaviors that all shapes share are being drawn and setting the bounding rectangle.

The next two classes, `CRectangle` and `COval`, are subclasses of `CShape`. They both inherit the `itsRectangle` instance variable and the `SetBounds` method. Both classes override the `Draw` method. This means that a `CRectangle` is identical to a generic shape, but it has its own drawing behavior. `COval` is also identical to a generic shape, but it has its own drawing behavior different from `CRectangle`'s.

Now scroll down to look at the implementation of the methods. These are the method definitions for the abstract class CShape:

```
procedure CShape.SetBounds (aTop, aLeft,
    aBottom, aRight: integer);
begin
    with itsRectangle do
        begin
            top := aTop;
            left := aLeft;
            bottom := aBottom;
            right := aRight
        end
    end;
end;

procedure CShape.Draw;
begin
    writeln('Shouldn't get here!')
end;
```

The first method, SetBounds, just sets up the boundary rectangle `itsRectangle`. Since `itsRectangle` is an instance variable for this class, you don't need to declare it in the var section of the procedure.

The second method is the generic Draw method. Because CShape is an abstract class, there shouldn't be an object of class CShape. Every subclass of CShape should override the Draw method to draw a particular kind of shape.

The Draw methods for CRectangle and COval override the generic Draw method with the specific behavior for their class.

```
procedure CRectangle.Draw;
begin
    FrameRect (itsRectangle)
end;

procedure COval.Draw;
begin
    FrameOval (itsRectangle)
end;
```

These are the two procedures you saw when you stepped through the program. They're what give CRectangle and COval their unique drawing behavior.

If you send an object of class CRectangle a Draw message, the `CRectangle.Draw` method gets called, and the `FrameRect` procedure draws a rectangle. If you send an object of class COval a Draw message, the



`COval.Draw` method gets called, and the `FrameOval` procedure draws an oval.

## How Do You Create an Object?

The specific `Draw` method that gets called when you send an object a `Draw` message depends on the class of the object. So, how do you create objects of different classes?

Look at the `ChooseShape` function in the file `LearnOOP.p`. This is the function that asks you to choose the shape:

```
function ChooseShape (var theShape: CShape):
  Boolean;
begin
  ChooseShape := TRUE;

  writeln('Create Oval or Rectangle?');
  write('[0/R/Q] : ');
  readln(answer);

  case answer[1] of
    'Q', 'q':
      ChooseShape := FALSE;
    'R', 'r':
      new(CRectangle(theShape));
    'O', 'o':
      new(COval(theShape));
    otherwise
      theShape := nil
  end
end;
```

After it asks you for the kind of shape to draw, this function looks at what you typed and returns an object of the appropriate class.

If you typed an `R` or an `O`, it uses the Pascal procedure `new` to create a new object.

Note that the `var` parameter `theShape` is declared as class `CShape`. When it creates the shape, the function first type casts it to class `CRectangle` or to class `COval`. So this line:

```
new(CRectangle(theShape));
```

means “Treat `theShape` as a `CRectangle` and create an object of that class.”

The rules for assignments of objects say that you can *assign an object* of a subclass to a variable declared as a superclass. What this means is that you

## 5 Tutorial: LearnOOP

---

can assign a CRectangle object to a variable of class CShape, but you can't assign an object of class CShape to variable of class CRectangle.

After it creates the shape, ChooseShape returns the new object, and the object gets set to the global variable aShape in the while loop:

```
while ChooseShape(aShape) do
```

The class of aShape is CShape. Doesn't this mean that when you send a Draw message to aShape it should call the generic CShape.Draw method? No, because the class of an object is determined at run time. This is called **dynamic binding**.

So if you create an object of class CRectangle and send it a Draw message, the method that gets called is CRectangle.Draw.

What happens when you send the SetBounds message? Since neither the CRectangle class nor the COval class override the SetBounds method, the generic CShape.SetBounds method gets called when you send either class of object a SetBounds message. This is called **inheritance**. The CRectangle and COval classes inherit the SetBounds method from their superclass, CShape.

### How Do You Define a New Class?

Now, why don't you try creating a new class that's a subclass of one of the existing three classes. Here's an example of how to do it. You'll create a subclass for drawing squares.

The first thing to do is to decide which class to subclass. Since a square is just like a rectangle—except that it has sides of equal lengths—try making the square a subclass of CRectangle. You can use the same drawing method, but you'll have to override the SetBounds method. Here's how you'd declare the new CSquare class:

*If you want to follow along, put this class declaration right under the COval declaration in the file UShapes.p.*

```
CSquare = object(CRectangle)
  procedure SetBounds (aTop, aLeft, aBottom,
    aRight: integer);
    override;
end;
```



And here's one way of defining the `SetBounds` method for the `CSquare` class:

*If you want to follow along, put this method definition right under the `COval.Draw` definition in the file `UShapes.p`.*

```
procedure CSquare.SetBounds (aTop, aLeft,
  aBottom, aRight: integer);
begin
  aRight := aLeft + (aBottom - aTop);
  inherited SetBounds(aTop, aLeft, aBottom,
    aRight)
end;
```

This procedure says to reset the right edge of the bounding rectangle to be the same distance from the left edge as the bottom edge is from the top edge. (It ignores whatever value you give it for `aRight`.) The next line says to use the superclass' `SetBounds` method. In this case, it means to call `CRectangle`'s `SetBounds` method which it inherits from `CShape`.

The only thing left to do is to edit the `ChooseShape` function in the file `LearnOOP.p` to handle the new shape. Here's how:

```
function ChooseShape (var theShape: CShape):
  Boolean;
begin
  ChooseShape := TRUE;

  writeln('Create Oval, Rectangle, Square?');
  write('[0/R/S/Q] : ');
  readln(answer);

  case answer[1] of
    'Q', 'q':
      ChooseShape := FALSE;
    'R', 'r':
      new(CRectangle(theShape));
    'O', 'o':
      new(COval(theShape));
    'S', 's':
      new(CSquare(theShape));
    otherwise
      theShape := nil
  end
end;
```

Try running the program now with the new class. When you ask `ChooseShape` to create a square, it should ignore the value you give for the right edge of the bounding rectangle.

### Note

Because of the way LearnOOP reads the coordinates of the bounding rectangle, you still have to provide a value for the right edge.

---

## But How Does It Know?

Even though you've seen object-oriented programming work, you may still be wondering who or what in the LearnOOP program is calling the right method for a given message. Yes, there is a mechanism that figures out which method goes with which message for a particular object. But this mechanism is abstracted away in the compiler and becomes part of the language.

Here's an example. You probably know that most microprocessors use different machine language instructions for multiplying integers and for multiplying floating point numbers. But when you write a Pascal program you know that you can write statements like:

```
floatVar := 1961.0102 * floatVal;  
count := 24 * intVal;
```

Though there's a "hidden mechanism" that knows when to use floating point multiplication and when to do integer multiplication, you don't worry about it. The inner workings of the multiplication operator are abstracted away and become a part of the language.

It's the same thing with object-oriented programming. The mechanism that matches messages and methods with objects is a part of the language. The power of object-oriented programming comes not from how it works but from the layer of abstraction that it adds to the low level mechanics of data structures and algorithms.

## Where to Go Next

Since this small program declares only three classes, this is a good time to learn how to use the Class Browser. Chapter 6 describes the Class Browser in detail.

If you want, you can try adding more methods or classes. Try adding a method that asks where to put the shape so you don't have to provide an extra value for the CSquare class. Or you might want to try adding methods for filled shapes or for shapes with different line thicknesses.

# *The Class Browser*

---

6



**T**his chapter shows you how to use the class browser. The class browser is a tool that displays all the classes defined in your project as a tree chart. You can use the Class Browser to look at the declaration of classes or to find the definitions of methods.

## **Contents**

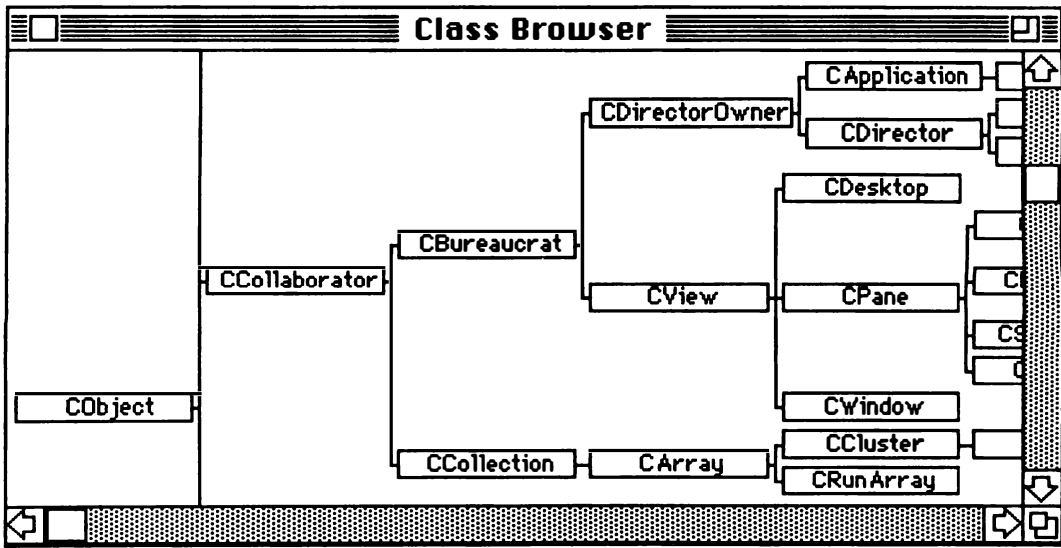
Using the Class Browser . . . . .	. 55
Finding the Declaration of a Class. . . . .	. 55
Finding a class declaration with the Class Browser . . . . .	. 55
Finding a class declaration with the editor . . . . .	. 56
Finding the Definition of a Method . . . . .	. 57
Finding a method definition with the Class Browser . . . . .	. 57
Finding a method definition with the editor . . . . .	. 58
Keyboard Shortcuts in the Class Browser . . . . .	. 59
Class Browser Summary . . . . .	. 60

## 6 *The Class Browser*

---

## Using the Class Browser

To use the browser, choose the **Class Browser** command from the **Windows** menu. The Class Browser is shown in Figure 6-1.



**Figure 6-1** The Class Browser

The boxes represent the class hierarchy of the classes in your project. Each box represents one class. Subclasses are connected to their superclasses like an organizational chart.

In this picture, you can see that the classes CDesktop, CPane, and CWindow, are all descendants of CView.

## Finding the Declaration of a Class

You can find the definition of a class two ways: with the Class Browser or with the editor.

### Finding a class declaration with the Class Browser

To find the declaration of a class, double-click on the box that represents the class. The editor opens the file that contains the class declaration and scrolls to the declaration of the class.

## 6 The Class Browser

For instance, to see the declaration of the CBureaucrat class, double-click on the CBureaucrat box. The editor opens the file CBureaucrat.h and scrolls to the declaration of CBureaucrat, as shown in Figure 6-2.

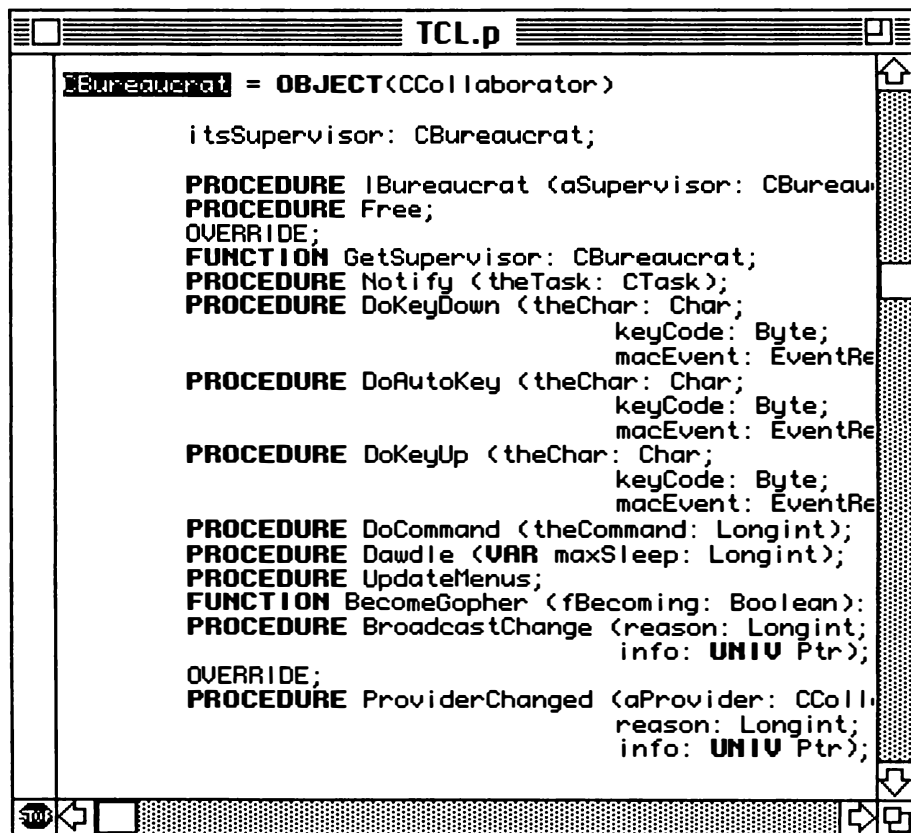


Figure 6-2 The declaration of CBureaucrat

### Finding a class declaration with the editor

You can find the declaration of a class in the editor the same way you find the definition of a procedure or function. When you hold down the Option key as you double-click on the name of a class, the editor opens the file that declares that class.

For example, if you're looking at the CEditPane.IEditPane method, and you want to see the declaration of the CBureaucrat class, you can Option-double-click on the name CBureaucrat in the parameter list.

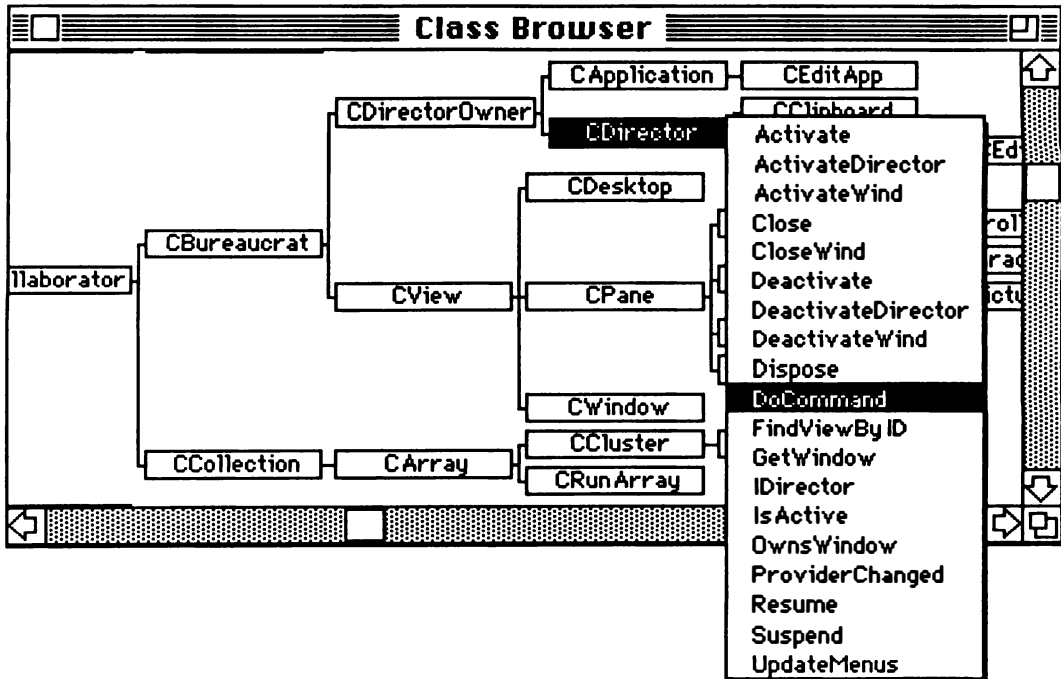


## Finding the Definition of a Method

You can find the definition of a method two ways: with the Class Browser or with the editor.

### Finding a method definition with the Class Browser

To find all the methods that a class defines or overrides, hold the mouse down on the class name. A pop-up menu appears next to the class.



**Figure 6-3** The methods of CDirector

If you choose a method from the pop-up menu, the editor opens the file that defines the method.

#### Note

The pop-up menu shows only the methods that a class defines or overrides. The pop-up menu does not include the methods that the class inherits from its superclass.

## 6 The Class Browser

### Finding a method definition with the editor

You can find the definition of a method from the editor the same way you can find the definition of a procedure or function. When you hold down the Option key as you double-click on the name of a method, the editor opens the file that defines that method and scrolls to the method definition.

Since several classes may define or override the same method, it's not clear which file the editor should open. When a method is defined in more than one class, the editor opens the Class Browser window and highlights the classes that define the method with a bold border.

For example, suppose that you Option-double-click on the method name `ChangeSize` in the `CEditPane.IEditPane` method, like in Figure 6-4

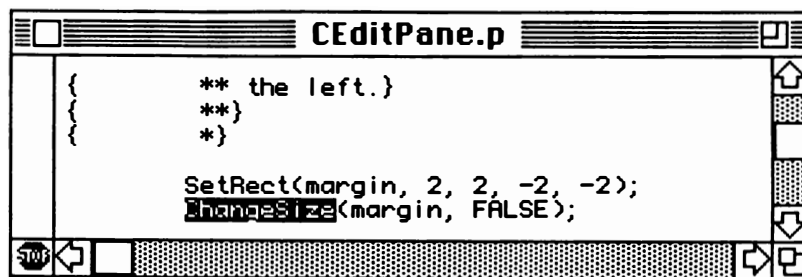


Figure 6-4 Option-double-clicking on the call to the `ChangeSize` method.

Since several classes define the `ChangeSize` method, the editor opens the Class Browser window and highlights the classes that define the method as shown in Figure 6-5.

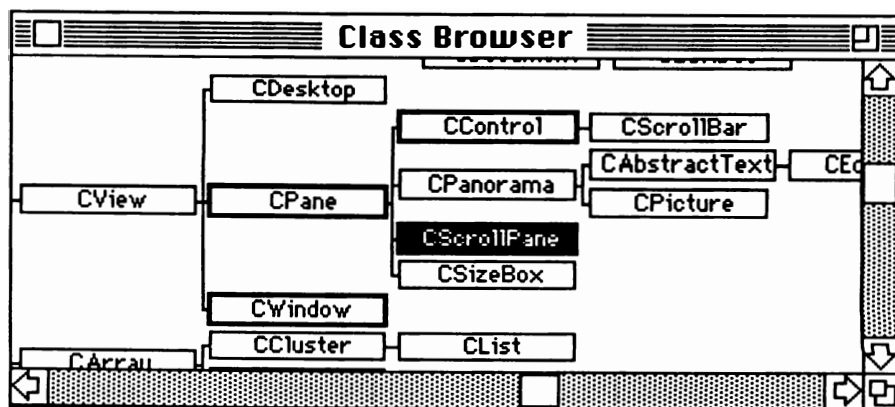


Figure 6-5 The classes that define a `ChangeSize` method.

To see the definition of the method, double-click on one of the highlighted classes. The editor opens the file that defines that method.

If you picked the wrong class or if you want to see another definition of the method, you can Option-double-click on the method name again to display the Class Browser window again.

## Keyboard Shortcuts in the Class Browser

You can use the keyboard to navigate through the classes in the Class Browser window. The keyboard commands let you select classes or find the definitions of classes without using the mouse.

If you type the name of a class, the Class Browser highlights the class whose name matches what you've typed so far. For instance, as you type "CControl", the browser highlights CObject, CChore, CCollaborator, and finally CControl.

This is what other keys do in the Class Browser:

<b>Key</b>	<b>Action</b>
Enter or Return	Open the file that contains the definition of the selected class. This is the same as double clicking on a class name.
Up Arrow	Select the previous sibling of the current class.
Down Arrow	Select the next sibling of the current class.
Left Arrow	Select the superclass of the current class.
Right Arrow	Select a subclass of the current class.
Tab	Traverse each class in the class hierarchy.

### Class Browser Summary

This table summarizes how to use the Class Browser:

<b>If you're in the...</b>	<b>and you want to find a...</b>	<b>do this:</b>
Class Browser	class declaration	double-click on the class name.
Editor	class declaration	Option-double-click on the class name.
Class Browser	method definition	hold down the mouse to see a pop-up menu of all the methods the class defines or overrides. Choose a method name from the pop-up menu to see its definition.
Editor	method definition	Option-double-click on the method name. If there is more than one definition, the Class Browser highlights all the classes that define the method. Double-click on the class name to see the definition of the method.

# THINK Pascal



## *The THINK Class Library*

### *Part Three*

- 7 The THINK Class Library
- 8 Exception Handling
- 9-70 The Classes
- 71 TCL Library Routines
- 72 Global Variables



# The THINK Class Library

---



7

**T**he THINK Class Library is a collection of classes that implement a standard Macintosh application. It takes care of things like handling menu commands, updating windows, dispatching events, dealing with MultiFinder, maintaining the Clipboard, and so on.

## Contents

Introduction . . . . .	. 65
What you should know . . . . .	. 65
Conventions . . . . .	. 65
Types . . . . .	. 65
Naming conventions . . . . .	. 66
Object-oriented terminology . . . . .	. 66
Overview . . . . .	. 67
The class hierarchy . . . . .	. 67
The visual hierarchy . . . . .	. 69
The chain of command . . . . .	. 70
The flow of control . . . . .	. 71
Writing an Application with the TCL . . . . .	. 72
Creating the project in THINK Pascal . . . . .	. 73
Creating the project in THINK C . . . . .	. 74
Creating the application subclass . . . . .	. 74
Creating the document subclass . . . . .	. 75
Creating the pane subclass . . . . .	. 76
Working with Panes . . . . .	. 77
Windows and panes . . . . .	. 78
Coordinate systems . . . . .	. 79
Drawing in a pane . . . . .	. 80
Properties of panes . . . . .	. 81
Panoramas . . . . .	. 83
Scroll panes . . . . .	. 86
Cursor tracking . . . . .	. 87
Initializing views from resources . . . . .	. 87
Working with Menus . . . . .	. 88
Using MENU resources . . . . .	. 89
Building menus on the fly . . . . .	. 90

## 7 The THINK Class Library

---

Dimming and checking menu items . . . . .	90
Handling Low Memory Situations . . . . .	92
Undoing and Mouse Tracking . . . . .	93
Undoing . . . . .	93
Mouse tracking . . . . .	94
Segmentation and the THINK Class Library . . . . .	94
Debugging and the THINK Class Library . . . . .	95
Debugging aids in THINK C . . . . .	95
Debugging aids in THINK Pascal . . . . .	96
THINK Class Library Resources . . . . .	96
Alerts . . . . .	96
Controls . . . . .	97
Error message strings . . . . .	97
Menus . . . . .	97
Menu bars . . . . .	98
Small icon . . . . .	98
Strings and string lists . . . . .	98
Window template . . . . .	99
Modifying the THINK Class Library . . . . .	99
Where to Go Next . . . . .	100





## Introduction

This chapter introduces you to the THINK Class Library and talks about some of the general topics in application building. After you read this chapter, you should try running the demonstration applications. Be sure to follow the instructions in Chapter 2 of your user manual, and in this manual's Chapter 2, "Installing the THINK Class Library" to install the THINK Class Library and its demonstration programs.

### What you should know

You should be comfortable working with THINK C or THINK Pascal. You should be familiar with the fundamentals of Macintosh programming. Particularly, you should know about QuickDraw, the Window Manager, and the Memory Manager.

If you're using THINK C, you should be familiar with the THINK C implementation of objects. If you're using THINK Pascal, you should know Object Pascal. Earlier chapters in this manual cover these topics in detail. If you need a fundamental introduction to object-oriented programming, see Chapter 3, "Object-Oriented Programming." As you go through this chapter, remember that this is all new territory, so don't be discouraged if it takes a while for all the pieces to fall into place.

## Conventions

This manual is designed for both the THINK C and THINK Pascal versions of the THINK Class Library. The chapters that follow describe the classes in the THINK Class Library. The chapters are organized like this:

Section	Description
Introduction	A brief description of the class.
Heritage	The class's superclass and its subclasses.
Using	A detailed description of how to use the class, along with examples.
Variables	A list of the instance variables of the class.
Methods	A description of each method that the class implements or overrides.

### Types

Most of the types used to declare instance variables are the same in C and in Pascal. For 32-bit integers and 16-bit integers, this manual uses the Pascal names. C programmers should assume the corresponding C type:

## 7 The THINK Class Library

Pascal type	C type
integer	short
longint	long

Hexadecimal values use C syntax: 0x0FA is the same as \$0FA in Pascal.

In Pascal a pointer that doesn't point to anything has a special value called NIL. In C the same value is called NULL. This manual uses both terms.

### Naming conventions

The THINK Class Library follows these naming conventions:

Name	Description
CName	The name of a class.
OName	The name of a class that was defined in the original version of the THINK Class Library, but replaced in this version.
aName	A formal parameter.
cName	A class variable.
fName	A flag. Usually a Boolean instance variable.
gName	A global variable.
kName	A constant. In Pascal these are defined as <code>const</code> . In C these are defined with <code>#define</code> or <code>enum</code> .
itsName	An instance variable.
theName	A variable. Usually a local variable or an instance variable. Sometimes used for formal parameters.
macName	A Macintosh data structure used either as an instance variable or as a local variable.

### Object-oriented terminology

This manual uses the terminology established in Chapter 3, "Object-Oriented Programming" to talk about object-oriented programming.

The syntax for calling a method inherited from the superclass is slightly different between C and Pascal, so this manual uses the phrase "call the inherited method." In THINK C, you would call it like this:

```
inherited::MethodName();
```

In THINK Pascal, you'd call it this way:

```
inherited MethodName;
```



## Overview

The THINK Class Library is organized into three distinct, interacting structures: the class hierarchy, the visual hierarchy, and the chain of command. The class hierarchy is the set of all the classes that make up the THINK Class Library. The visual hierarchy describes the organization of all visible entities. The chain of command specifies which objects get to handle commands.

The THINK Class Library converts Macintosh events into **visual messages** and **direct commands**. A visual message is an event that affects the visual hierarchy. Mouse clicks, activate events, and update events are all visual messages. A direct command is a request that an object perform an action. Direct commands are usually the result of menu commands.

To convert Macintosh events into messages and commands, the THINK Class Library uses an object called a **switchboard**. The switchboard calls `GetNextEvent` or `WaitNextEvent` repeatedly and, depending on the kind of event, sends a message to the appropriate object. Each application has only one instance of a switchboard object. Another object, called the **bartender**, takes care of converting menu selections into direct commands.

### The class hierarchy

The class hierarchy describes the relationships among all the classes in the THINK Class Library. All of the classes are descendants of the root class `CObject`. The class hierarchy is in Figure 7-1 on page 68. To make the drawing easier to read, the initial 'C' of each class is omitted.

---

#### Note

The classes with a heavy outline constitute the core classes that must be present in any program that uses the THINK Class Library.

---

Do not think of the class hierarchy as a functional description of the THINK Class Library. You might think from the drawing of the class hierarchy that the `CRadioGroup` class somehow interacts with the `CBureaucrat` class, but that's not the case.

Instead, think of the class hierarchy as a family tree. Each class inherits all of the behavior (methods) and all of the attributes (instance variables) of its ancestor. So the `CButton` class *is* a control. It behaves like a control and responds to all of the messages a control responds to.

Some of the classes in the class hierarchy are **abstract classes**. The THINK Class Library never creates objects of these classes. They're used to give a

## 7 The THINK Class Library

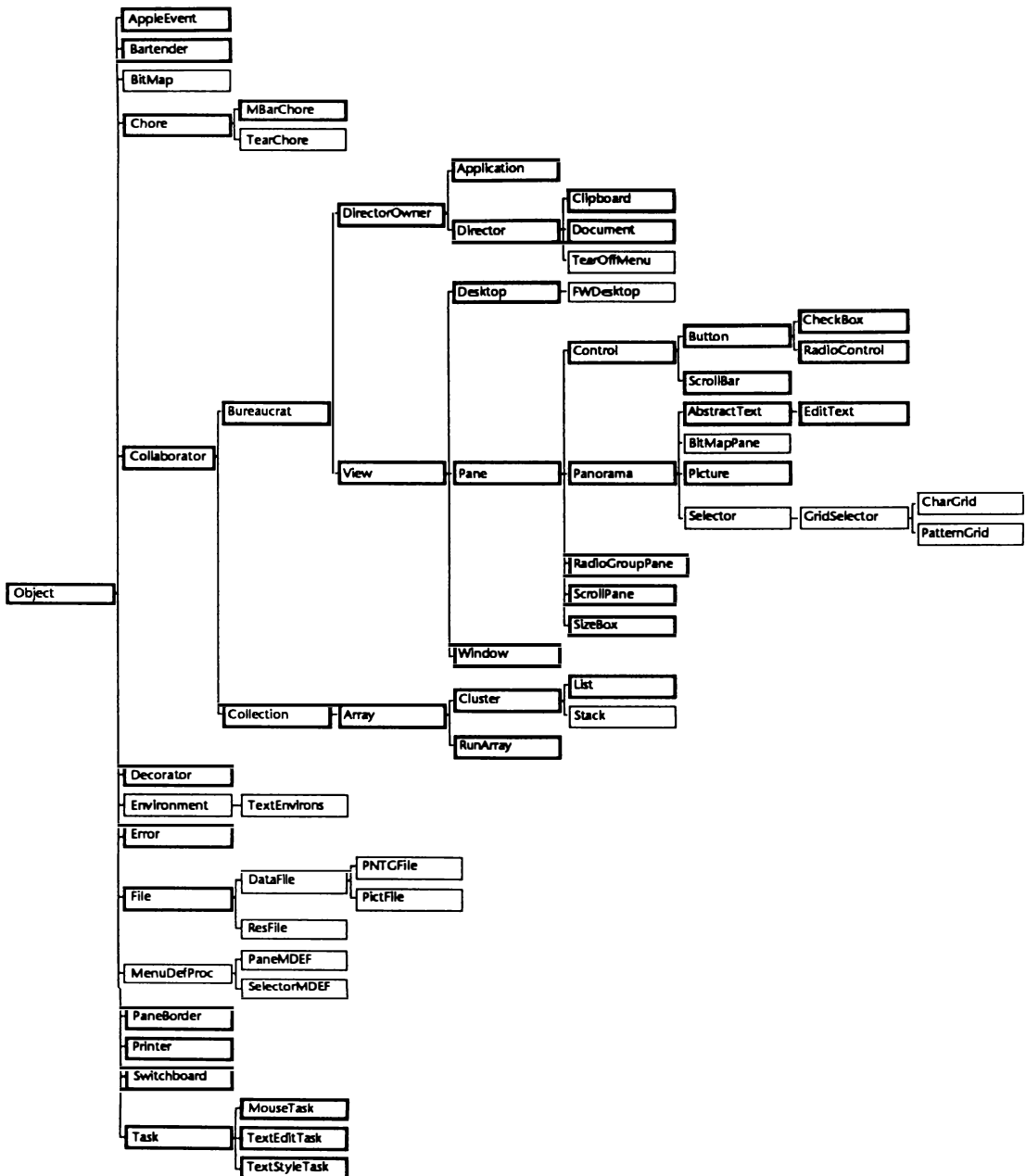


Figure 7-1 The THINK Class Library's class hierarchy

family of objects common behaviors. The most important abstract classes in the THINK Class Library are CObject, CCollaborator, CBureaucrat, CView, CDirectorOwner, and CApplication.

### The visual hierarchy

The visual hierarchy describes all the visible objects, or **views**, that the THINK Class Library knows about. The visual hierarchy is built around the idea of **enclosures**. Everything that you see on the screen belongs to—is enclosed by—another visual entity.

At the top of the visual hierarchy is the **desktop**. The desktop encloses all of the windows in an application. Each window encloses one or more panes, and those panes may enclose other panes. Figure 7-2 shows a typical visual hierarchy.

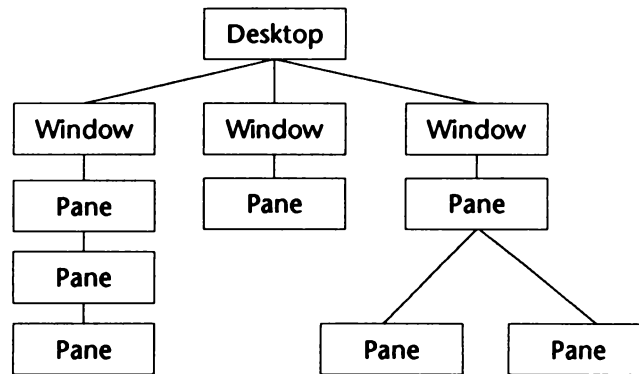


Figure 7-2 A typical visual hierarchy

Unlike the class hierarchy, the visual hierarchy is dynamic. It changes as your program runs. When you open a new document, you add another window to the desktop, and you add another set of panes to the new window.

Panes are the most important kinds of views. All drawing takes place in a pane. The THINK Class Library defines different kinds of panes designed for different kinds of displays. Every pane has its own drawing environment, so you can draw in a pane without worrying about where it is on the screen.

Visual events work their way down the visual hierarchy. Since Macintosh update and activate events always have a window associated with them, these messages get sent directly to a window object. Mouse clicks and cursor adjustment messages always work their way down *from* the desktop to the active window to the appropriate pane.

## 7 The THINK Class Library

Panes can handle visual commands like mouse clicks. When you click in a pane, the switchboard determines which pane the mouse went down in and sends it a `DoClick` message. Because they're descendants of `CBureaucrat`, panes can be in the chain of command and respond to direct commands.

### The chain of command

The chain of command specifies which object handles a direct command. The chain of command is based on the idea of **supervisors** and **bureaucrats**. Every object in the chain of command is a bureaucrat. If a bureaucrat can't handle a direct command, it passes the command on to its supervisor. The application is the only bureaucrat that does not have a supervisor. If the application doesn't handle the command, no object will.

Objects in the chain of command are descendants of the class `CBureaucrat`. Every bureaucrat has a `DoCommand` method that the subclass can override to handle specific commands. The default `DoCommand` method just sends a `DoCommand` to its supervisor.

The first object to get a chance to handle a command is called the **gopher**. If the object that the gopher points to can't handle the command, it sends the command on to its supervisor. Your application is responsible for sending a `BecomeGopher` message to a bureaucrat that should be the gopher.

In Figure 7-3, the gopher points to a pane whose supervisor is a document. If the pane can't handle a direct command, it passes the command on to the document. If the document can't handle the command, it gets passed up to the application.

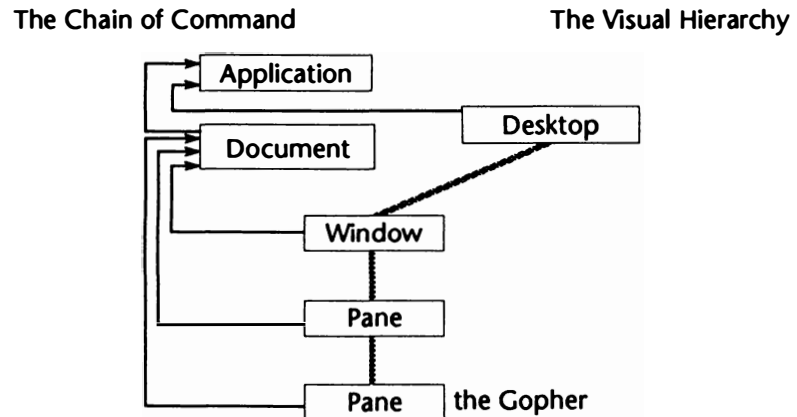


Figure 7-3 The gopher, the chain of command, and the visual hierarchy

The enclosures in the visual hierarchy are drawn in gray. Note that the chain of command is separate from the visual hierarchy.

`CDirector` is an important subclass of `CBureaucrat` that you need to know about. A director is a bureaucrat that supervises a window. Directors handle the communication between the visual hierarchy and the chain of command. For instance, when a window gets an activate event, it sends an `ActivateWind` message to its supervisor, which is always a director. The director can then take some action as a result of becoming active.

Another descendant of `CBureaucrat` is `CDocument`. A document is a director that has a file associated with it. Documents manage the communication between windows, files, and menu commands. The default document class handles common commands like **Save**, **Save As...**, **Print**, etc. You can think of a document as a file that you view through a window. A better way to think about a document is that it is the essence of a Macintosh application. It is anything that you can display and manipulate inside a window.

Every bureaucrat is a descendant of the class `CCollaborator`. A **collaborator** is an object that can let other objects know that something has happened. One collaborator is called the **provider**, and another is called the **dependent**. For instance, a provider might be a document that displays data in several windows. Its dependents might be the windows. If the data changes, the provider lets the dependents know, so the windows redisplay the data correctly.

### The flow of control

The chain of command and the visual hierarchy get their direction from the switchboard. The switchboard gets events from the Macintosh Event Manager and converts the event into messages for either the chain of command or the visual hierarchy. Messages for the chain of command usually go to the gopher, the first bureaucrat in the chain. Messages for the visual hierarchy go to the active window or to the desktop.

When you press the mouse, the switchboard sends a `DispatchClick` message to the desktop. If the click was in the menu bar, the desktop sends the bartender a message to update the state of the menus before they appear. The bartender sends a message to the chain of command to enable and disable the appropriate menu items. Then the desktop uses the bartender to convert the menu selection into a direct command and sends a `DoCommand` message to the gopher.

If the click was in a window, the desktop sends a `DispatchClick` message to the window, which eventually sends a `DoClick` message to the

## 7 The THINK Class Library

---

pane the mouse went down in. The pane's `DoClick` method can then do whatever's appropriate for the pane. It might even send a `DoCommand` message to an object in the chain of command.

When the switchboard gets an activate or an update event, it sends an `Activate`, `Deactivate`, or `Update` message to the window. The window sends a similar message to its director.

When you type, the switchboard sends a `DoKeyDown` or a `DoAutoKey` message to the gopher. If you hold down the Command key when you type, the switchboard asks the bartender to convert the key into a command and sends a `DoCommand` message to the gopher.

If you're running under `MultiFinder`, and you bring another application to the foreground, the switchboard sends a `Suspend` message to the application (not the gopher) which sends `Suspend` messages to all of its directors. A similar thing happens when your application comes to the foreground.

---

### Note

The THINK Class Library treats desk accessories as if they were in their own layer, even if you're not using `MultiFinder`, so your application will still get suspend and resume "events" when you bring up a desk accessory.

---

## Writing an Application with the TCL

This section tells you how to write an application with the THINK Class Library. The easiest way to do this, is to take the Starter program and build from it.

To create an application with the THINK Class Library, you create subclasses of existing classes. The three classes you need to override are `CApplication`, `CDocument`, and `CPane`. Your application subclass determines the overall structure of your application. The document subclass implements the way your application handles its files, and the pane subclass implements how the information in your file appears in the document windows.

In addition to the subclasses, you also need a resource file for your project. This resource file must contain the standard THINK Class Library resources as well as your own. The standard THINK Class Library resources are in the file `TCL Resources`. When you use the THINK Class Library, make a copy of this resource file and name it the same as your project and append `.rsrc` to it. Then add your own resources to it.





---

**Note**

For more information about resource files and the THINK Class Library, see "THINK Class Library Resources" on page 96. To learn about using resources with a project, see Chapter 7, "The Project," of the *THINK C User Manual* or Chapter 7 "Working with Projects" in the *THINK Pascal User Manual*.

---

**Creating the project in THINK Pascal**

Use the Starter project as the beginning of your project. The Starter project is in the Starter Folder in the TCL 1.1 Demos Folder.

1. Copy the Starter Folder and change its name to the name of your application
2. Open the new folder
3. Rename Starter. $\pi$  to the name of your application
4. Rename Starter.Build. $\pi$  to the name of your application, but keep the .Build. $\pi$  suffix.
5. Rename Starter. $\pi$ .rsrc to the name of your project plus .rsrc
6. Open the project without the .Build. $\pi$  suffix.
7. Use the **Run Options...** dialog to choose the resource file you renamed in step 5.
8. Use the **Find...** command in the **Search** menu to change Starter to the name of your application
9. Use the **Save As...** command to save the files under your own names (the **Save As...** command also changes the names of the files in the project)

THINK Pascal uses two projects—one for debugging and one for building the final application. When you build an application, THINK Pascal has to turn the debugging options off. That means that the entire THINK Class Library would need to be recompiled. By using two projects and switching to the .Build. $\pi$  project when you want to build the final application, only your own files will need to be recompiled.

### Creating the project in THINK C

Use the Starter project as the beginning of your project. The Starter project is in the Starter Folder in the TCL 1.1 Demos Folder.

1. Copy the Starter Folder and change its name to the name of your application
2. Open the new folder
3. Rename Starter. $\pi$  to the name of your application
4. Rename Starter. $\pi$ .rsrc to the name of your project plus .rsrc
5. Open the project
6. Use the **Find...** command in the **Search** menu to change Starter to the name of your application
7. Use the **Save As...** command to save the files under your own names (the **Save As...** command also changes the names of the files in the project)

Use the **Set Project Type...** dialog to make sure that your project is Multi-Finder Aware and will receive Suspend & Resume events.

### Creating the application subclass

If your application is a standard Macintosh application, your application subclass must override these methods:

#### *Initialization method*

SetUpFileParameters	CreateDocument
OpenDocument	DoCommand

Your initialization method should initialize any instance variables that your subclass declares. It must call CApplication's IApplication method.

The SetUpFileParameters method sets up the standard file parameters that specify which files are visible in the standard file box when the user chooses **Open...** from the **File** menu.

The CreateDocument method creates a new, untitled document. The document it creates is one that you define as a subclass of CDocument. After creating the document, your CreateDocument method should send it a NewFile message. This is the method that gets called when the user chooses **New** from the File menu.

The OpenDocument method is like the CreateDocument method. Instead of sending the newly created document a NewFile message, though, this document should send it an OpenFile message. The OpenDocument



method takes one parameter, an SFReply record, that contains the information about the file that the user chose to open.

The DoCommand method handles all the application-specific commands. Most of the commands should be handled at the document level. Some commands, like **New**, **Open**, and **Quit**, are handled by the default application class.

### Creating the document subclass

The document is where your application draws and displays its data. All documents have windows associated with them. Most documents also have an associated file. Neither the window nor the file are created automatically. You must create them yourself.

Your document class should override these methods:

<i>Initialization method</i>	OpenFile
Dispose	DoSave
DoCommand	DoSaveAs
NewFile	Revert

Your document class must have an initialization method. If your subclass defines new instance variables, this is the method that sets them up. By convention, the name of your initialization method should be `I YourDoc` where *YourDoc* is the name of your document class. Your initialization method should call the inherited method. The supervisor of a document is always gApplication.

If your document allocates memory, you should also override the Dispose method to deallocate it. Be sure that your method calls `inherited::Dispose` (inherited `Free` in Pascal) to make sure that the document is disposed of properly.

---

#### Note

You do not need to dispose of the `itsWindow` or the `itsFile` instance variables. The default Dispose or Free method does that for you.

---

Your document class's DoCommand method does most of the work in your application. When a window is active, the switchboard will send all commands to the document first (it's the gopher), and if the document can't handle it, the application class tries to handle it. Your document class should handle all the commands it knows about, and call the inherited method when it can't.

---

### Note

Be sure that your `DoCommand` method or that one of the methods it invokes sets the instance variable `dirty` to `TRUE` when there has been a change to the document.

---

Your document class will get a `NewFile` message when the user chooses **New** from the **File** menu. This method needs to create a window and attach the panes for it. The `NewFile` method doesn't need to create a file until the user tries to save the document.

Your document gets an `OpenFile` message when the user chooses **Open...** from the **File** menu. The `OpenFile` method has one argument: a pointer to a Macintosh `SFReply` record. When you get the `OpenFile` message, you can be sure that the `SFReply` record is properly filled in. Your `OpenFile` message needs to create an instance of a file object (usually of class `CDataFile`). You can send your file any of several read messages to obtain its contents. Your `OpenFile` method also needs to create a window to display the contents of the file, just as your `NewFile` method does.

When the user chooses **Save** from the **File** menu, your document gets a `DoSave` message. Your `DoSave` method should write the contents of its file to disk. The file object is stored in the instance variable `itsFile`.

When the user chooses **Save As...** from the **File** menu, your document gets a `DoSaveAs` message. This method takes an `SFReply` record, and you can be sure that it is properly filled in. Your document class needs to override this method to write its data to a file.

If your application supports the **Revert** command, you should implement it in the `DoRevert` method. Your implementation might do the same thing as closing without saving, then opening the file again.

### Creating the pane subclass

Once you've created your windows and opened your files, you need to display them somewhere. In the THINK Class Library, you don't write directly onto the window. Instead, you create a subclass of class `pane`.

When you create a subclass of `Pane`, you need to override these methods:

<code>I YourPane</code>	<code>Dispose</code>
<code>Draw</code>	<code>DoClick</code>

Your pane class should have an initialization method. If your subclass defines new instance variables, this is the method that sets them up. By con-



vention, the name of your initialization method should be `IYourPane` where *YourPane* is the name of your pane class. Your initialization method should call the inherited method of whatever pane class you're overriding. The supervisor of a pane should be either the pane that encloses it or the director its window belongs to.

The pane initialization method is where you set the pane's location in its enclosure and its characteristics. If you want your pane to receive clicks, be sure to send the pane a `SetWantsClicks (TRUE)` message, otherwise mouse clicks in your pane are ignored.

If your pane allocates memory, you should also override the `Dispose` or `Free` method to deallocate it. Be sure that your method calls inherited method to make sure that the pane is disposed of properly.

The `Draw` message tells your pane to draw its contents. You can assume that the port, clip region, and coordinate system have been set up correctly. See section "Drawing in a pane" on page 80 to learn all about drawing in a pane.

When you click in a pane, it will get a `DoClick` message. Your `DoClick` method can handle the mouse click itself to draw something, to drag an object, or to select something. Some panes, like the edit text pane implemented by `CEditText`, have built-in `DoClick` methods.

If you want your mouse action to be undoable, you need to create a subclass of `CMouseDown` and send it in a `TrackMouseDown` message. The section "Mouse tracking" on page 94 goes into more detail about undoable mouse actions.

## Working with Panes

In the THINK Class Library, all your drawing takes place in a **pane**. A pane is a rectangular region of the screen completely enclosed by a window. A window may have several panes, and each pane may have several subpanes. Each pane has its own drawing environment and can handle its own visual commands.

Every pane has an **enclosure** that completely encloses the pane. A pane also has a supervisor—an object in the chain of command. A pane's enclosure and supervisor can be the same object, particularly if the pane belongs to another pane. In most the common case, though, the supervisor is the director that owns the window or the pane.

Every pane has its own drawing environment. The rectangle that describes the edges of a pane is the **frame**. The frame defines a local coordinate sys-

## 7 The THINK Class Library

tem for the pane. In most cases, the top, left corner of the pane is the point (0, 0).

### Windows and panes

All panes are subclasses of the abstract class `CView` which defines the behavior of visual entities. A window is a view, but it is not a pane. Other visual entities are subclasses of panes; they include things like controls, borders, pictures, and size boxes.

Every pane belongs to a window or to another pane. The outer, or owning pane, is the enclosure. Every pane has one enclosure. For example, in the window in Figure 7-4, there are seven panes (the window is not a pane):

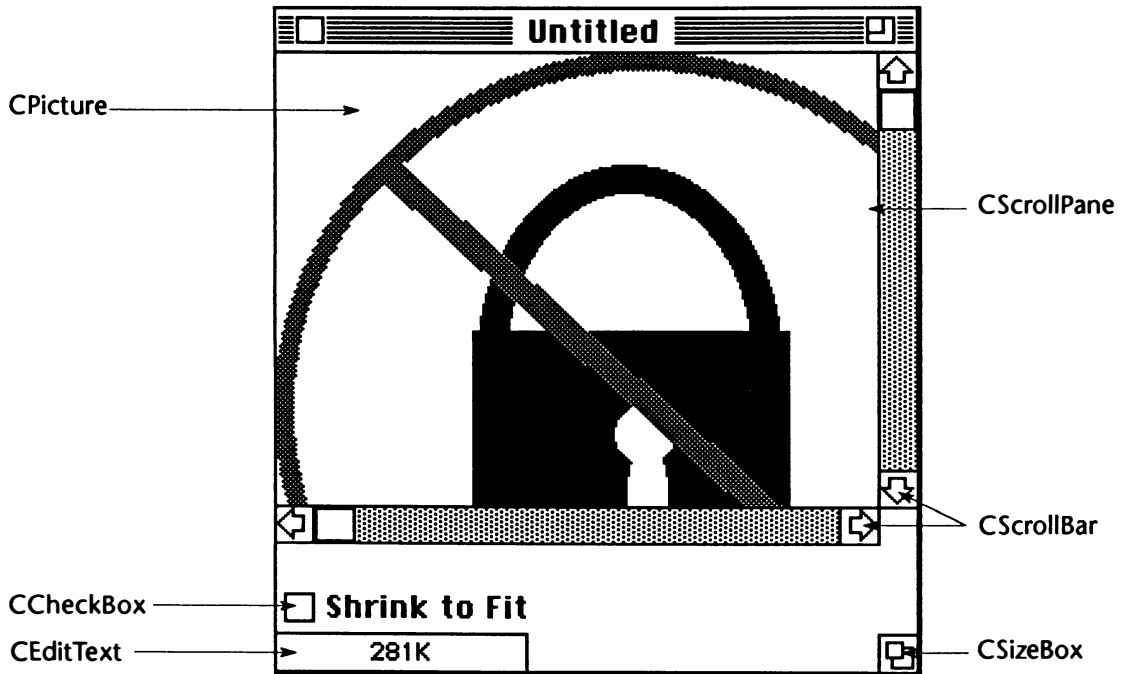


Figure 7-4 The panes in a window

The size box, the check box, the text, and the scroll pane are enclosed by the window. The two scroll bars belong to the scroll pane. The picture is enclosed by the window.



### Coordinate systems

When you're working with panes in the THINK Class Library, you need to know about four coordinate systems:

- Global coordinates
- Window coordinates
- Frame coordinates
- QuickDraw coordinates

The desktop, some internal methods, and some Toolbox routines use **global coordinates**. In this coordinate system, all units are in pixels, and (0, 0) is at the top, left corner of the main screen. You rarely use global coordinate in the THINK Class Library.

In **window coordinates** the top, left corner of the window's content region is (0, 0) and each unit is a pixel. The only time you need to use window coordinates is to set the position of a pane whose enclosure is a window. Window coordinates are also useful as a common point of reference for two different panes in the same window.

**Frame coordinates** provide a local coordinate system for a pane. Units in frame coordinates are in pixels, and the point (0, 0) is usually the top left corner of the pane. If the pane moves within its enclosure, the coordinate system does not change; the top left corner is still (0, 0). The only time this origin point changes is when you scroll the pane. Each pane can choose to use long coordinates or short coordinates.

All drawing and mouse tracking is done in **QuickDraw coordinates**. This is the coordinate system that the Macintosh Toolbox uses for its drawing operations. QuickDraw coordinates are only valid after a call to `Prepare`. The relationship between QuickDraw coordinates and the other coordinate systems depends on whether a pane is using long frame coordinates or short frame coordinates.

**Short coordinates** map directly to QuickDraw coordinates. Each element in a short coordinate uses 16-bit values, so a pane that uses short coordinates is limited to the rectangle (-32768, -32768, 32767, 32767). **Long coordinates** layer a 32-bit coordinate system on top of the QuickDraw 16-bit coordinates. The long coordinate system lets you use a much larger coordinate area for your pane. Since all drawing takes place in QuickDraw coordinates you have to map the long coordinates to QuickDraw coordinates when you draw in a pane.

## 7 The THINK Class Library

---

The CPane class defines several methods that transform coordinates from one system to another.

### Drawing in a pane

Every pane has its own drawing environment. The rectangle that describes the edges of a pane is the pane's **frame**. The frame defines a local coordinate system for the pane. In most cases, the top, left corner of the pane is the point (0, 0).

To draw in a pane, you override its Draw method. The THINK Class Library sends your pane a Draw message whenever the pane needs to be updated. You can send a Refresh message yourself if you want to force an update event.

To draw in a pane use the standard QuickDraw routines. The pane's Prepare method sets up the QuickDraw port. If your pane uses short coordinates, you the coordinate system is set up correctly. If your pane uses long coordinates, you need to transform frame coordinates to QuickDraw coordinates before you draw. You can CPane methods FrameToQD and FrameToQDR convert frame points and rectangles to QuickDraw points and rectangles.

The THINK Class Library uses the types LongRect and LongPt for both long and short coordinates. You'll notice that most of the descendants of CView that work with points and rectangles use these types.

---

#### Note

If you've worked with earlier versions of the THINK Class Library, this is the biggest change you'll notice since it will require some changes to your programs.

---

These two types are defined like this in LongCoordinates.h in C:

```
typedef struct LongPt
{
    long v, h;
} LongPt;

typedef struct LongRect
{
    long top, left, bottom, right;
} LongRect;
```





And like this in TCL.p in Pascal:

```

type
  LongPt = record
    case integer of
      1: (
        v, h: longint
      );
      2: (
        vh: array[VHSelect] of longint
      );
    end;

  LongRect = record
    case integer of
      1: (
        top, left,
        bottom, right: longint
      );
      2: (
        topLeft: LongPt;
        botRight: LongPt
      );
    end;

```

If the pane uses short coordinates, frame coordinates and QuickDraw coordinates are identical, so the values stored in a LongRect or in a LongPt are in QuickDraw coordinates. To use them with QuickDraw routines, however, you'll need to convert them to the QuickDraw types Rect and Point. The THINK Class Library provides several utility routines to do these conversions. For a complete list, see "Long Coordinate Utilities" on page 462.

To draw directly in a pane as a result of a mouse click, you need to override the DoClick method of the pane. Just as in drawing, if your pane uses short coordinates, you do not need to transform frame coordinates to QuickDraw coordinates. If you use long coordinates, you must transform the coordinates before you draw. To make your mouse action undoable, use a mouse task. See the section "Undoing and Mouse Tracking" on page 93.

### Properties of panes

When a pane moves or changes size, all of the panes that it encloses change as well. The way a pane changes depends on its **sizing characteristics**. When you create a pane, you specify its horizontal and vertical sizing characteristics.

## 7 The THINK Class Library

---

The horizontal sizing characteristic specifies how the pane's left and right edges change.

Horizontal sizing	Meaning
<code>sizeFIXEDLEFT</code>	The left edge of the pane is always the same number of pixels from the left edge of the enclosing pane as when it was originally placed.
<code>sizeFIXEDRIGHT</code>	The right edge of the pane is always the same number of pixels from the right edge of the enclosing pane as when it was originally placed.
<code>sizeFIXEDSTICKY</code>	The left and right edges stick to their original locations in the enclosing pane. If the enclosure scrolls horizontally, the pane will scroll with it.
<code>sizeELASTIC</code>	The width of the pane grows or shrinks by the same amount as the width of the enclosing pane.

The vertical sizing characteristic specifies how the pane's top and bottom edges change.

Vertical sizing	Meaning
<code>sizeFIXEDTOP</code>	The top edge of the pane is always the same number of pixels from the top edge of the enclosing pane as when it was originally placed.
<code>sizeFIXEDBOTTOM</code>	The bottom edge of the pane is always the same number of pixels from the bottom edge of the enclosing pane as when it was originally placed.
<code>sizeFIXEDSTICKY</code>	The top and bottom edges stick to their original locations in the enclosing pane. If the enclosure scrolls vertically, the pane will scroll with it.
<code>sizeELASTIC</code>	The height of the pane grows or shrinks by the same amount as the height of the enclosing pane.

A couple of examples might help: A vertical scroll bar in a window has the characteristics `sizeFIXEDRIGHT` and `sizeELASTIC`. It has a fixed horizontal length and remains anchored to the right edge of the window. Vertically, it changes with the height of the window. A status box in the lower left cor-

ner of a window would be `sizeFIXEDLEFT` horizontally and `sizeFIXEDBOTTOM` vertically. It has a constant size and remains anchored to the bottom left corner of the window.

In Figure 7-5, the dark line represents the window. It contains a main pane that takes up most of the window and several other panes. The two panes that hold the scroll bars are fixed to the edges of the window. When the window is resized, they grow or shrink by the same amount as the window. The panes that hold the grow box and the status box are anchored to the bottom corners of the window. The square pane in the upper left portion of the main pane will always be there, regardless of how the window changes. Its dimensions will not change automatically.

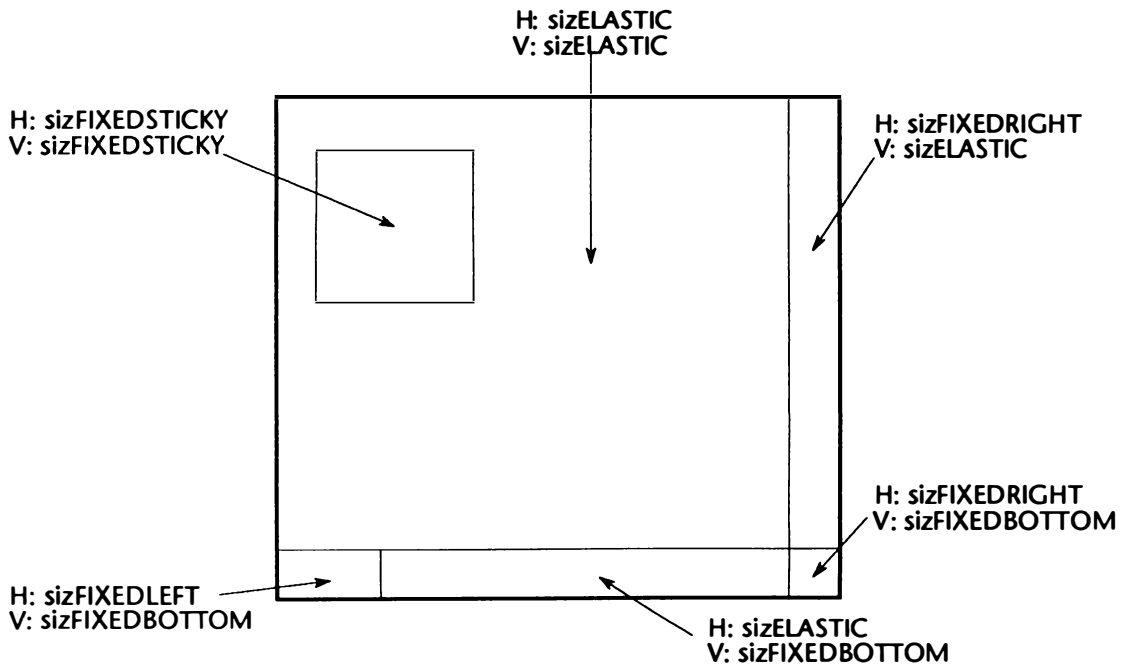


Figure 7-5 Horizontal and vertical sizing

### Panoramas

Almost everything you want to display is bigger than a pane. Graphics and text, for instance, frequently take up more room than what you can fit in a pane. The THINK Class Library provides a **panorama** class, `CPanorama`, that lets you display portions of a large graphic in a pane. You might say a panorama is a pane that scrolls.

Think of a panorama as a sheet of paper glued to a desk. The frame moves over the paper. The only part of the panorama you can see is what's inside the frame. To scroll, you move the frame around on the panorama to see different parts of it.

The rectangle that completely encloses the panorama is called the **bounds rectangle**. The bounds rectangle defines the size and coordinate system of the panorama. Usually, the top, left corner of the bounds rectangle is the point (0, 0), and the units in its coordinate system are pixels.

The coordinate system of the bounds rectangle specifies how the frame moves over the panorama when you scroll. In the usual case, when you scroll up, you move the pane up a pixel. In some applications, though, you want to scroll more than one pixel. In a text editing application or in a spreadsheet application, you want to move up by an entire line.

You can specify a **scale** that says how many pixels make up a single panorama unit. You can set different scales for horizontal and for vertical units. In a graphics application, each unit might be one pixel. In a spreadsheet, a vertical unit might be 12 pixels, and a horizontal unit might be 60 pixels.

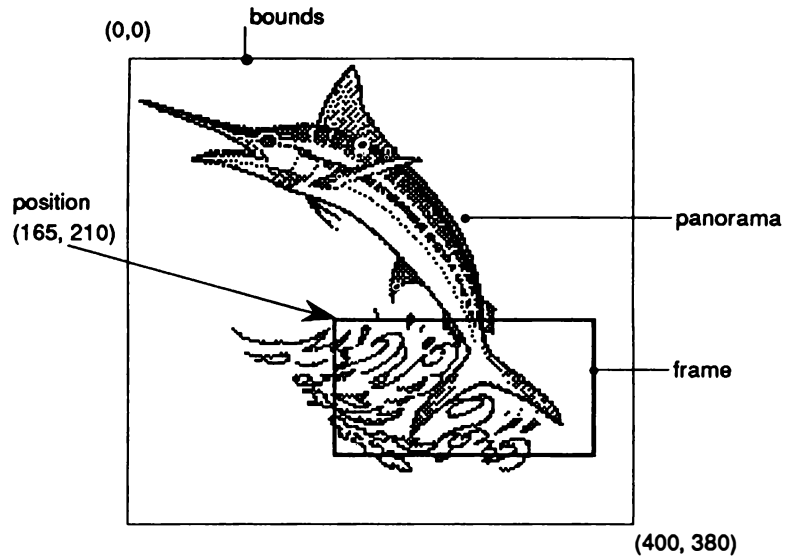
The units of the panorama bounds are for scrolling only. For drawing, you'll use the frame coordinates which are always pixel units.

There are two ways to talk about the top, left corner of the frame of the pane. The top, left corner of the pane, expressed in panorama units, is the **position** of the frame in the panorama. The top, left corner, expressed in frame coordinates, is the **origin** of the frame.

The key thing to remember is that scrolling always happens in panorama units. Drawing always takes place in frame coordinates. As you scroll, the origin of the frame changes. A couple of examples might help.

In Figure 7-6, both the horizontal and vertical scales are set to 1 pixel per panorama unit. The bounds rectangle of the panorama is (0, 0, 400, 380).

The portion of the picture you can see in the frame of the pane, the fish's tail, starts at (165, 210) in the panorama.



**Figure 7-6** A graphic panorama and its scales

Since the panorama units match the frame units, the position of the frame in the panorama and the origin of the frame are the same. If you were to draw a line from (220, 230) to (270,230), it would cut across the tail of the fish.

In Figure 7-7, which shows a panorama with text, the horizontal scale is 6 pixels per unit, and the vertical scale is 12 pixels per unit. The bounds rectangle of the panorama is (0, 0, 40, 9). In the panorama scale, this means 8 lines of 40 characters each. The position of the frame in the panorama is (0, 3), the beginning of the fourth line. The origin of the frame, however, is at

## 7 The THINK Class Library

(0, 36). If you wanted to draw a line to strike out the word “And,” you would draw it from (0, 42) to (18, 42)

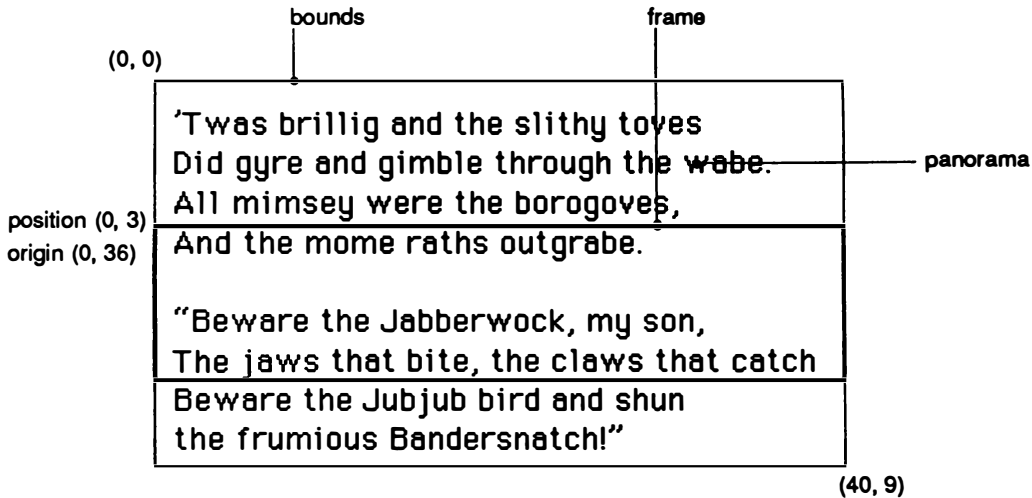


Figure 7-7 A text panorama and its scales

### Note

The top, left of the bounds rectangle doesn't have to be (0, 0). You can define the bounds coordinate system that's most convenient for the kind of data you're displaying in the panorama.

### Scroll panes

To make it easy to use panoramas, the THINK Class Library provides a class called `CScrollPane` that implements a **scroll pane**. Scroll panes give you an easy way to attach scroll bars to your panorama.

You create a scroll pane the same way as any other pane. You can request a vertical scroll bar, an horizontal scroll bar, and a size box. Then you use the `InstallPanorama` method to associate a panorama with the scroll pane. The scroll pane examines the panorama and adjusts the scroll bars appropriately. You can specify how many panorama units to scroll when you click on different parts of the scroll bar.

The scroll bars and the panorama communicate through the scroll pane. When you click in one of the scroll bars, it tells the scroll pane which tells the panorama how much to scroll.



### Cursor tracking

The `AdjustCursor` method that all panes inherit from `CView` lets you change the cursor when it moves into your pane. Most of the time you'll use only one cursor in the whole pane. In this case, all you have to do is set the cursor with the Toolbox routine `SetCursor`. Look at the `AdjustCursor` method in `CEditText` for an example.

Sometimes, though, you might want to use different cursors within the same pane. The `AdjustCursor` method lets you do this as well, but it takes a little more work. See the description of the `CView` class in Chapter 52.

### Initializing views from resources

The `IViewRes` method lets you initialize any descendant of `CView` from a resource template.

Resource	Class
Pane	CPane
Pano	CPanorama
PctP	CPicture
ScPn	CScrollPane
AbTx	CAbstractText, CEditText
View	CView

You can use `ResEdit` to create the resource templates for each class.

---

#### Note

Your THINK C package includes a file `TCL.TMPLs` that contains `TMPL` resources you can install into `ResEdit`. These `TMPLs` let you create and edit the resources above. See the instructions in Chapter 2, "Installing the THINK Class Library," to learn how to install these `TMPLs` into `ResEdit`.

---

When you use `ResEdit` to create view resource templates, keep in mind these values for sizing and clipping mnemonics.

#### Sizing values

```

sizFIXEDLEFT = 0
sizFIXEDRIGHT = 1
sizFIXEDTOP = 2
sizFIXEDBOTTOM = 3
sizFIXEDSTICKY = 4
sizELASTIC = 5

```

#### Clipping values

```

clipAPERTURE = 0
clipFRAME = 1
clipPAGE = 2

```

### Working with Menus

The THINK Class Library lets you think of your menu commands more abstractly than the Macintosh Toolbox. Instead of identifying a menu command by its menu ID and item number, the THINK Class Library lets you assign unique **command numbers** to each item of the menu. With command numbers, you can reassign functions to different menu items without having to rebuild the application.

Command numbers are positive long integers in the range 1 to 2,147,483,647. Command numbers in the range 1 to 1023 are reserved for the THINK Class Library. Command number 0 is reserved for `cmdNull`, the null command. All other command numbers (1024 to 2,147,483,647) are available for your application.

The reserved commands are for the most common Macintosh application commands like **Open**, **Save**, **Quit**, **Page Setup...**, etc. Be sure you use the reserved command number if you expect to get the default behavior from the THINK Class Library. For a list of all the reserved commands see the description for `CBartender` on page 165.

When you choose a command from the menu bar, the desktop sends a `FindCmdNumber` message to the bartender. The bartender matches the menu ID and the item number to a command number and sends it to the gopher in a `DoCommand` message. If the you use a Command key equivalent, the switchboard sends the `DoCommand` message to the gopher.

---

#### Note

Remember, the gopher is a pointer to a command object. Usually, the gopher points to a pane in the active window.

---



### Using MENU resources

When you create your menus with ResEdit, append the command number to the menu item. The menu item and the command number are separated by the character #. For example, Figure 7-8 shows the **File** menu.

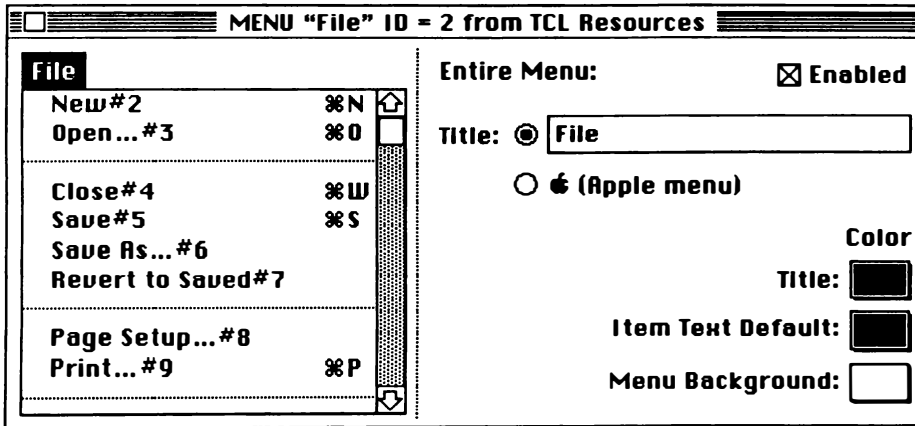


Figure 7-8 The File menu in ResEdit

#### Note

If you don't append a command number to a menu item, the bartender automatically assigns it cmdNull.

The MENU resources for these menus are in the file TCL Resources.

You can use any menu ID for your application's menus. The THINK Class Library reserves the following menu IDs for certain menus:

Menu title	Menu ID	Mnemonic
Apple	1	MENUapple
File	2	MENUfile
Edit	3	MENUedit
Font	10	MENUfont
Size	11	MENUsize

After you create all the menus that your application needs, create an MBAR 1 resource that contains all of the IDs of the menus your application uses. The application's SetUpMenus message creates the bartender (stored in the global variable gBartender) to read the MBAR 1 resource. The bartender creates the tables that match command numbers to menu items.

### Note

Your application's menus must be in `MBAR 1` unless you change the definition of `MBARapp` in `Constants.h` or `TCL.p`.

If you want the bartender to return the menu ID and item number of a particular menu item, use the special command number `-1` in your `MENU` resource. The bartender will return the negative of the menu ID in the high word and the menu item number in the low word. This is the same as menus that you build on the fly, described next.

### Building menus on the fly

Menus that you create as your program is running, like Font menus, won't have command numbers associated with them. In this case, the bartender's `FindCmdNumber` method returns the negative of the menu ID in the high word and the item ID in the low word. When your `DoCommand` method gets a negative command number, you know you have to figure out the command from the menu ID and item number.

For example, if `DoCommand` gets a command `-655369 (0xFFFF5FFF7)`, it means that there is no command number associated with the menu item. To get the menu ID and the menu item, negate the value and split it into two words. In this example, the return value becomes `655369 (0x000A0009)`, which means that the menu ID is 10 and the menu item is 9.



**Figure 7-9** How `FindCmdNumber` builds command numbers.

You can add menu items to existing menus if you like. For example, you might want to add the Font menu to a general text-handling menu, or you might want to have a menu with the names of all the documents your application has opened. The important thing to remember is to add all these menus at the **end** of the existing menu. Otherwise, the bartender will get confused.

### Dimming and checking menu items

The bartender includes methods to let you enable and disable and check and uncheck menu items. When you click in the menu bar, the bartender sends an `UpdateMenus` message to all of the bureaucrats in the chain of command. In the general case, all the items in the menu start out dimmed and unchecked. Then each bureaucrat enables the menu items that pertain



to it. Once the appropriate items have been enabled and checked, the Toolbox routine `MenuSelect` displays all the menus.

---

#### Note

The dimming, undimming, and unchecking take very little time. You won't notice a delay between the time you click on the menu bar and when the menu is displayed.

---

Suppose you click in the menu bar of a text processing application. When you click on the menu bar, but before the Toolbox displays the menu, the bartender disables all the menu items. Then, the application enables all the application related menu items: **New**, **Open...**, **Quit**, for example. The document enables all the document related items: **Save**, **Save As...**, **Revert** (if the document's been changed), and so on. A pane might check the current font and size in the **Font** menu. Finally the menu appears on the screen with the correct items checked and enabled.

The `UpdateMenus` method of your application, document, and pane need to enable each item. To make sure that item enabling happens from the general (application) to the specific (pane), be sure to call the inherited method first in your own `UpdateMenus` method.

You can use the bureaucrat methods `SetDimOption` and `SetUnchecking` in your application `SetUpMenus` method to modify this behavior. `SetDimOption` lets you specify whether the bartender should dim all, some, or none of the items when you click on the menu bar. For Font menus, for instance, it doesn't make sense to dim all the font names only to reenable them again.

#### Dim Option

`dimNONE`

`dimSOME`

`dimALL`

#### Meaning

Never dim any of the menu items.

Dim only the menu items that have command numbers associated with them.

Dim all of the menu items. Each bureaucrat's `UpdateMenus` method must enable the items for the commands it handles. This is the default.

There are only two options for `SetUnchecking`. The bartender either unchecks all the items or it doesn't.

Unchecking option	Meaning
TRUE	Uncheck all the menu items at menu selection. Your <code>UpdateMenus</code> method should check the appropriate items. Set this option for menus like Font menus or Style menus.
FALSE	Don't uncheck any menu items at menu selection. This is the default since most menu items never need to be checked.

*The term "rainy day fund" comes from the expression "saving for a rainy day." It means to save money in case of disaster.*

### Handling Low Memory Situations

The `CApplication` class provides several methods that deal with low memory situations. These methods use a memory reserve called the **rainy day fund**.

---

#### Note

For details about the methods and instance variables described in this section, see Chapter 11, "CApplication."

---

When you call `IApplication`, you specify how much memory your application should allocate for the rainy day fund. You also specify how many bytes of that fund should be available to satisfy Toolbox routine memory requests and how many bytes of that fund should be available to satisfy critical operation requests. These values are stored in the instance variables `toolBoxBalance` and `criticalBalance`.

If the Macintosh Memory Manager gets a request for more memory than is available, it calls a grow zone function. In the THINK Class Library, the grow zone function sends the application a `GrowMemory` message.

The `GrowMemory` method tries several strategies to free memory in the heap. First, it sends the application a `MemoryShortage` message. Your application subclass should override this method to release memory that is not crucial to execution. For instance, your `MemoryShortage` method might dispose of a buffer it no longer needs. If the `MemoryShortage` method wasn't able to release enough memory, `GrowMemory` starts using the rainy day fund.

`GrowMemory` looks first at the `inCriticalOperation` instance variable to release memory from the rainy day fund:



<b>If inCriticalOperation is</b>	<b>leave this much in reserve</b>
TRUE	toolboxBalance
FALSE	criticalBalance

A critical operation is a temporary operation that takes place in a single Macintosh event, that might take a lot of memory, and that should never fail. For instance, saving a file is a critical operation. You can use the routine `SetCriticalOperation` to set the `inCriticalOperation` flag.

If `GrowMemory` still wasn't able to release enough memory and the `canFail` flag is TRUE, `GrowMemory` returns without trying to allocate any more memory. If the `canFail` flag is TRUE, the application is saying that it can deal with a failing memory request.

If the `canFail` flag is FALSE, `GrowMemory` tries to release all the memory left in the rainy day fund because the application is not prepared to deal with a failing memory request. If there is not enough memory, even after using the rainy day fund, it is likely that the application will crash.

You can use the routine `SetAllocation` to set and reset the `canFail` flag. In general, your application should be prepared to handle failing memory requests.

## Undoing and Mouse Tracking

The THINK Class Library provides a class that lets you implement the **Undo** command easily. In the default implementation, each document has its own undo history.

### Undoing

The THINK Class Library uses the abstract class `CTask` to implement undoable actions. For every undoable action you want in your application, you need to create a subclass of `CTask`.

After you perform an action, you store enough information to undo it in your task's instance variables. Then you send the task to your supervisor in a `Notify` command. The `CDocument` class implements the `Notify` method to store a task in one of the document's instance variables. When you choose **Undo** from the **Edit** menu, the document's `DoCommand` method sends an Undo message to the task it stored.

Here's an example. Suppose you've defined a subclass of `CTask` to change the font in an edit text pane. Before passing the command on to the edit pane's `DoCommand` method, you create a task and store the current font in

an instance variable. After you pass the font command to the edit text pane, you send the task in a `Notify` message to the document.

Your `Undo` method would simply send the font change command to the document. Since the command goes through the regular command chain, your `DoCommand` method would create a task to let you undo what you were undoing.

### Mouse tracking

The THINK Class Library uses the undo mechanism to make mouse tracking easier and undoable. The `CMouseTask` class is an abstract class that defines some methods specifically for mouse tracking.

To implement a mouse tracking task, define a subclass of `CMouseTask` and override the `KeepTracking` and `EndTracking` methods. The `KeepTracking` method does whatever you want to happen while the mouse is down. The `EndTracking` method does whatever you want to happen when the mouse is released.

For example, if you're moving a rectangle from one place in a pane to another, the `KeepTracking` method might draw a gray outline that moves as you move the mouse. The `EndTracking` method would erase the rectangle from its old location and redraw it in the new location.

If you want to make your mouse task undoable, you need to store enough information in the object to undo the effects of mouse tracking. You must also override the `Undo` method (inherited from `CTask`) to use this information to undo the effects of the mouse task.

After tracking the mouse, you can send the task in a `Notify` message to the document. When you choose **Undo** from the **Edit** menu, the document sends an `Undo` message to the task to undo the effects of mouse tracking.

### Segmentation and the THINK Class Library

You can segment your application any way you want. The only thing to keep in mind is that certain files and libraries must be in a resident segment, that is, a segment that is never purged.



In THINK C, these files must be in a resident segment:

- MacTraps
- MacTraps2
- oops (or oopsDebug)
- TCLUtilities.c
- LongCoordinates.c
- Exceptions.c

*See Chapter 7 of your THINK Pascal User Manual for more information about segmentation in THINK Pascal.*

In THINK Pascal the following files must be in a resident segment. You can set the attributes of their segment to be Locked.

- TCL.p
- Exceptions.p
- TCLRuntime.lib
- Interface.lib
- TCL.lib

---

**Note**

The directive segments «%\_MethTables» and «%\_SelProcs» must also be in a resident segment. If you are using Far Code, «%\_MethTables» must be in a segment by itself.

---

## Debugging and the THINK Class Library

The THINK C and THINK Pascal versions of the THINK Class Library use some compiler-specific features to help you debug programs that use the TCL.

### Debugging aids in THINK C

When you're debugging your application, you should use the library oops-Debug. This library contains a version of the message dispatcher that checks for NULL objects or possibly missing methods. Though there's no problem in building an application with this library, it is slightly less efficient than the regular oops library.

If the preprocessor symbol `__TCL__DEBUG__` is defined, certain error-checking functions like the `ASSERT` macro are enabled. The `ASSERT` macro takes a Boolean expression as an argument and raises an exception if the assertion is FALSE.

In THINK C, the global variables `gBreakFailure` and `gAskFailure` let you examine and simulate exceptions. If `gBreakFailure` is TRUE, THINK

## 7 The THINK Class Library

---

C calls the Toolbox routine `Debugger` when an exception is raised. If `gAskFailure` is `TRUE`, THINK C calls `Debugger` when the utility routines like `FailOSErr` that may raise exceptions are called. You can then change the arguments to the utility routine to simulate an exception.

There are two underscores at the beginning and end of `__TCL_DEBUG__`. You can set the value of `__TCL_DEBUG__` in the Prefix page of the **Options...** dialog.

### Debugging aids in THINK Pascal

The THINK Pascal version of the THINK Class Library uses three compiler variable to control debugging. If the compiler variable `TCL_DEBUG` is `TRUE`, the `Assert` procedure is enabled. The `Assert` procedure evaluates a Boolean expression and raises an exception if the result is `FALSE`.

---

#### Note

In `Starter.π`, `TCL_DEBUG` is set to `TRUE`. In `Starter.Build.π` it's set to `FALSE`.

---

If the compiler variable `ResumeAfterError` is `TRUE`, any time an exception is raised, an alert box appears, and the program ends.

Another compiler variable, `DebugExceptions` is used only when `ResumeAfterError` is `FALSE`. If `DebugExceptions` is `TRUE`, the THINK Class Library tries to handle exceptions in the THINK Pascal environment. The debugging option must be on for any file that may raise an exception.

If `ResumeAfterError` is `FALSE` and `DebugExceptions` is `FALSE`, the THINK Class Library handles exceptions in the normal way, but not in the THINK Pascal environment. These are the settings in `Starter.Build.π`.

Use the **Compile Options...** command to set these compiler variables.

## THINK Class Library Resources

The THINK Class Library requires certain resources in your project's resource file. All of the resources described in this section are in the file `TCLResources`. The mnemonic constants for all of these resources are in the file `Constants.h` in the `Core Headers` folder in THINK C and in `TCL.p` in THINK Pascal.

### Alerts

The `ALRT` and `DITL` resources always have matching IDs. You can change these resources to suit your application.



**ALRT/DITL**

128

**Used for**

General. A handy, all-purpose alert box. The DITL contains only “^0”, so you can use the Toolbox routine `ParamText` to set up the text.

150

Confirm to revert to last saved version.

151

Confirm to save changes before closing or quitting.

200

Severe Macintosh error has occurred.

250

No printer selected.

251

Error alert. The DITL says “Couldn’t complete the last command because ^0.”

252

Error alert. The DITL says “Couldn’t successfully startup or quit the application because ^0”

253

Assertion failed. Used by assertion in exception handling.

300

Macintosh OS error alert.

**Controls**

The THINK Class Library uses this CNTL template for all the scroll bars it creates.

**CNTL**

300

**Used for**

Scroll bar

**Error message strings**

The THINK Class Library uses a resource of type `Est r` to report Macintosh errors. `Est r` resources have exactly the same format as `STR` resources. You can use the ResEdit command **Open as Template...** in the **File** menu to open and edit an `Est r` resource as a `STR`.

The ID of the `Est r` resource is the error code you want to identify. The file `TCL Resource` includes one `Est r`. The error handling class `CError` uses `Est r` resources to display messages. You should create an `Est r` for every error your application reports to the user.

**Est r**

-108

**Used for**

Out of memory

-192

Tried to get nonexistent resource

**Menus**

The THINK Class Library reserves these menu IDs for the standard menus. The **File** and **Edit** menus contain all the standard items. You can remove the

## 7 The THINK Class Library

ones that don't apply to your application. The bartender builds the desk accessory menu for you automatically, but you'll have to build the **Font** and **Size** menus yourself in the `SetUpMenus` method of your application. For an example of Font and Size menus, see the `TinyEdit` project in the `TCL 1.1 Demos` folder.

MENU	Used for
1	Apple
2	File
3	Edit
10	Font
11	Size

---

### Note

In THINK C the mnemonics for these menus are in `Commands.h`, not in `Constants.h`. In THINK Pascal, these constants are in `TCL.p`.

---

### Menu bars

The THINK Class Library uses this resource to install all the menus in your application. The `MBAR` resource in `TCL Resources` automatically includes the **Apple**, **File**, and **Edit** menus.

MBAR	Used for
1	List of all menus to install at application startup.

### Small icon

Earlier versions of THINK Class Library used this small icon to draw a grow box instead of the Toolbox routine `DrawGrowIcon`. If your application uses `SICNs`, you can use the routine `DrawSICN` to draw it in a pane. See page 459 for details.

SICN	Used for
200	Grow box

### Strings and string lists

The THINK Class library uses these strings for various prompts and messages. You can modify these to suit your application.



<b>STR</b>	<b>Used for</b>
150	Prompt for the <b>Save As...</b> dialog box.
300	Generic operating system error message used when no <code>ErrStr</code> resource is available.
301	Generic error suffix. "of a Mac OS Error"
<b>STR#</b>	<b>Used for</b>
128	List of common Macintosh words. This list includes the words: quitting, closing, Undo, Redo, Untitled, Show Clipboard, Hide Clipboard.
129	Strings used for low memory warnings.
130	Task names for changing the wording of the Undo menu item text. This resource has no strings. Your application should add strings to this list if it supports Undo. See the descriptions of <code>CTask</code> and <code>CMouseDownTask</code> .
131	Strings used by exception handler.

### Window template

The THINK Class Library requires only one window template for the Clipboard window. Your application will probably define one or more additional WIND templates.

<b>WIND</b>	<b>Used for</b>
200	Clipboard. Window template used for displaying the clipboard.

## Modifying the THINK Class Library

You can modify the THINK Class Library classes to suit your particular needs. To change the behavior of one of the THINK Class Library classes, create a new subclass of the class you want to modify, and add new methods or override the methods you need to change. In some cases, it might be a better idea to change the source code for a class.

Be aware that there are dangers in changing the source code of a class. From time to time, Symantec may release new versions of the THINK Class Library. Changes that you make in the source of a class may make it difficult to use new classes or updates of existing classes. If you do make changes to the source of a class, be sure to keep an archival copy of the original class and to mark your changes clearly.

## 7 The THINK Class Library

---

The second danger is a little more subtle. As you use the THINK Class Library, you may want to create new classes and subclasses to implement some kind of behavior. If you want to be able to use these classes in other programs, and especially if you want others to be able to use these classes, you should not rely on any features of your modified classes.

---

### Note

Under your license agreement, you may distribute new subclasses of the classes in the THINK Class Library. You may not, however, distribute modified sources of the classes in the THINK Class Library.

---

## Where to Go Next

Learning to use the THINK Class Library takes time and experimentation. Start with the `TinyEdit` example in the `TinyEdit` Folder. It was built from the Starter application you should use to create your own applications. You might start by adding an **About...** box to the `TinyEdit` application. (Hint: Since it's an application-wide command, implement it in the application's `DoCommand` method.) As you explore how the `TinyEdit` application was put together, look at the next chapters to understand how the classes of the THINK Class Library work.

# Exception Handling

## 8

**T**he THINK Class Library includes a new exception handling mechanism to detect and respond to failures. The exception handling mechanism lets you divide your routines into two parts: one to handle the normal operation and another to handle abnormal situations.

---

### Note

Although the exception handling routines are designed to work with the THINK Class Library, you can use it in a limited way with non-TCL programs.

---

## Contents

What Is an Exception Handler? . . . . .	103
Using the Exception Mechanism . . . . .	104
Exception handling conventions . . . . .	105
Displaying error messages . . . . .	105
Exception Handling in THINK C . . . . .	106
The basic form . . . . .	106
Use handlers only when necessary . . . . .	107
Returning values . . . . .	108
Special cases . . . . .	109
Exception Handling in THINK Pascal . . . . .	109
The basic form . . . . .	109
Use handlers only when necessary . . . . .	111
Using Exit with exception handlers . . . . .	113
Special cases . . . . .	113
Exception Handler Routines . . . . .	113
Exception handler routines . . . . .	113
Exception raising routine . . . . .	114
Error detection routines . . . . .	115
Utility routines . . . . .	115

## ◆ 8 *Exception Handling*

---

## What Is an Exception Handler?

An exception is the occurrence of an abnormal condition during the execution of a program. For example, your program may open a file, write and read data to it, and close it. That's the normal operation for your program. In the course of operation, though, your program might encounter several exceptions: the file may not exist, the file may be unavailable, there might be media errors, or your program may not be able to close the file.

The exception handling mechanism in the THINK Class Library lets you separate the normal execution part of a routine from the error handling part of the routine. Without an exception handler, your program might have this kind of structure:

```
create a file object
if creation failed
    report an error
else
    open the file
    if open failed
        delete the file object
        report an error
    else
        allocate a buffer
        if allocation failed
            delete the file object
            report an error
        else
            ...
```

As you can see, the normal flow of control is hidden in all the error-checking code. With an exception handler, your code looks more like this:

```
TRY TO DO THE FOLLOWING:
    create a file object
    open the file
    allocate a buffer
    read the contents of the file
    close the file
CATCH ANYTHING THAT WENT WRONG:
    if the file object was created, delete it
    if the buffer was allocated, release it
    if the file is still open, close it
```

The normal portion is called the **try handler**, and the portion that handles abnormal conditions is called the **catch handler**. A routine in the try block can **raise an exception** to stop normal operation and go to a catch block.

## 8 Exception Handling

---

The routine that raises an exception is called `Failure`. The exception handling mechanism has several utility routines that test their parameters and call `Failure` for you if an error actually occurred. These routines are described later on page 115.

The exception handling mechanism keeps a stack of catch handlers. If a routine raises an exception, and there is no catch handler in the routine, control passes to the most recent catch handler. The THINK Class Library has a catch handler in `CApplication's` `Run` method, so any exception is always caught.

Suppose you have three functions, and each with a catch and try handler:

```
function A
  TRY
    call B
  CATCH
    clean up A
end

function B
  TRY
    call C
  CATCH
    clean up B
end

function C
  TRY
    do something
  CATCH
    clean up C
end
```

If an operation in function C's try handler raises an exception, control passes to C's catch handler. When that handler is finished, control passes to B's catch handler, and then finally to A's catch handler. If B had no catch handler, control would go from C's catch handler to A's catch handler.

Not every routine needs to have try and catch handlers, just those that would need to clean up if an error had occurred. Some of the time, you can rely on a caller's catch handler to do the cleanup for you.

### Using the Exception Mechanism

The exception handling mechanism is conceptually the same in both THINK C and in THINK Pascal, but, because of differences in the languages, you use them differently. In THINK C, the exception handler uses C macros to di-



vide the normal portion of your functions from the exception portions. In THINK Pascal, it uses nested procedures.

This section discusses some aspects of exception handlers common to both languages. The next two sections cover specific aspects of the exception handler for THINK C and for THINK Pascal.

### Exception handling conventions

Regardless of the language you're using, you should follow certain conventions for writing try and catch handlers.

If an object cannot be created (that is, if a call to `new` fails) the THINK Class Library raises an exception with `FailNIL`. In THINK C, `CObject's` `operator new` is overridden. In THINK Pascal, you must use `TCLRuntime.lib`.

Once you've created an object successfully, you need to make sure that it will be released if the initialization fails. You can set up a catch handler to dispose of the object, or, if the object reference is an instance variable of another class, you can rely on its catch handler to dispose of the object. The examples below show you how to do this.

Make sure that you set all object references to `NIL` before you create them with `new`. This way you can tell whether the object was created because the object reference will be non-`NIL`.

The memory management methods are designed to protect your application from running out of memory in extreme circumstances. In normal operation, you should check all of your memory allocations without relying on the memory reserves. See page 138 for more information about handling low memory situations.

### Displaying error messages

Since catch handlers propagate, you can rely on the top-level catch handler in `CApplication's` `Run` method to display an error message. If you want to display your own message, you have to make sure that the catch handler in `Run` does not display a message.

When a try handler wants to raise an exception, it calls `Failure`, which takes two arguments: the error that caused the failure and a message. These two values are saved in the globals `gLastError` and `gLastMessage`. The error is typically an operating system error. The message is optional. It's used by the `ErrorAlert` routine to find an error string to display.

*See page 466 for a description of the `ErrorAlert` utility routine.*

## 8 Exception Handling

### Note

In Pascal, you can look at the parameters of the catch handler procedure to determine the error and message values.

You can display your own error message in your catch handler. If you do, be sure to call `SetFailInfo` with an error code `kSilentError`. This special error code tells the top-level handler not to display an error message at all. Whatever you pass to `SetFailInfo` as the message is ignored.

## Exception Handling in THINK C

In THINK C, you use the `TRY`, `CATCH`, and `ENDTRY` macros to write exception handlers. The macros are defined in `Exceptions.h`.

### The basic form

This is the basic form for writing an exception handler in C:

```
void Example(void)
{
    TRY {
        This is the normal flow.
        Anything here can raise an exception that will be
        caught in the catch handler.
    }
    CATCH {
        This is the catch handler.
        Clean up after errors here: close files, release memory, etc.
    }
    ENDTRY;
}
```

Here's a concrete example:

```
void Example (void)
{
    CThing *anObject = NULL;

    TRY {
        anObject = new CThing;
        anObject->IThing();
    }
    CATCH {
        ForgetObject (anObject);
    }
    ENDTRY;
}
```

**Example 8-1** A simple exception handler in C

In Example 8-1, there are two places where an exception can be raised. First, when you create anObject, there may not be enough memory. If that's the case, new fails and raises an exception. Second, an operation in the IThing initialization method may raise an exception. In either case, control passes to the catch handler.

If new failed, anObject will be NULL, so the call ForgetObject won't do anything, and there's no other cleanup to do. If new succeeded, then the exception was caused by some operation in IThing. In that case, ForgetObject sends a Dispose message to anObject to release it.

### Use handlers only when necessary

You don't need a catch handler every time you create an object if you know that it will be disposed of by another handler. Here's an example of that:

```
void CMyApp::CreateDocument(void)
{
    CMyDoc *newDoc = NULL;

    TRY {
        newDoc = new CMyDoc;
        newDoc->IMyDoc(...);
        newDoc->NewFile();
    }
    CATCH {
        ForgetObject(newDoc);
    }
    ENDTRY;
}

void CMyDoc::NewFile(void)
{
    BuildWindow();
    itsWindow->Select();
}

void CMyDoc::BuildWindow()
{
    CCoolPane *myPane;

    itsWindow = new CWindow;
    itsWindow->IWindow(...);

    myPane = new CCoolPane;
    myPane->ICoolPane(...);
    ...
}
```

**Example 8-2** A set of routines with only one catch handler in C

## 8 Exception Handling

In Example 8-2 only `CMyApp::CreateDocument` needs a catch handler. Here's why. `BuildWindow` creates a window and immediately stores into `itsWindow`, which is an instance variable in `CMyDoc`. `BuildWindow` also creates a pane, which is immediately added into the view hierarchy by `ICoolPane`. If an exception is raised, it will be caught in `CreateDocument`'s catch handler, which calls `ForgetObject`. `ForgetObject` sends a `Dispose` message to `newDoc`, which sends a `Dispose` message to `itsWindow`, which in turn disposes of all the panes that the window encloses.

### Returning values

If your function returns a value, you must not return from the try handler. Instead, your function should return after `ENDTRY`. For example, look at `CDataFile::ReadAll`:

```
Handle CDataFile::ReadAll (void)
{
    register OSErr errCode;
    long length;
    Handle contents = NULL;

    TRY
    {
        FailOSErr(GetEOF(refNum, &length));
        contents = NewHandleCanFail(length);
        FailNIL(contents);
        FailOSErr(SetFPos(refNum,
                        fsFromStart, 0L));
        FailOSErr(FSRead(refNum, &length,
                        *contents));
    }
    CATCH
    {
        ForgetHandle(contents);
    }
    ENDTRY;

    return contents;
}
```

### Example 8-3 Returning from a function with try and catch handlers

The `ReadAll` method in Example 8-3 is a good example of using the exception handling mechanism. Note how all of the File Manager routines that return error codes are wrapped in calls to `FailOSErr`. This utility function checks the error return value, and if it is not `noErr`, it raises an exception. Note also how the function uses `NewHandleCanFail` to allocate a handle. That function ensures that the memory manager won't use any of the memo-

ry reserves to get memory. The call to FailNIL raises an exception if the handle couldn't be allocated.

### Special cases

In some abnormal cases, you may want to keep catch handlers from propagating to the top-level handler in CApplication::Run. If so, just use the macro NO\_PROPAGATE somewhere in your catch handler.

If you think you can fix whatever raised the exception in your catch handler, you can use the RETRY macro to retry the try handler:

```
void Example(void)
{
    TRY {
        something that might raise an exception
    }
    CATCH {
        fix the problem
        RETRY;
    }
    ENDTRY;
}
```

### Example 8-4 Using RETRY

## Exception Handling in THINK Pascal

In THINK Pascal, you use a nested procedure to write an exception handler. Before the main flow of control, you use the CatchFailures procedure to set up the nested procedure as the exception handler. Your routine must also declare a variable of type FailInfo to hold the information the exception handling mechanism needs.

### Warning

For the exception handler to work correctly under the THINK Pascal environment, you must have debugging enabled for all procedures and functions that may raise an exception. The exception mechanism works correctly when you build an application.

### The basic form

This is the basic form for writing an exception handler in Pascal:

```
procedure Example;
var
    fi: FailInfo; { info for exception handler }
```

## 8 Exception Handling

---

```
procedure HandleFailure (error: integer;  
                        message: longint);  
begin  
    This is the catch handler.  
    Clean up after errors here:  
    close files, release memory, etc.  
end;  
  
begin  
    CatchFailures(fi, HandleFailure);  
  
    This is the normal flow.  
    Anything here can raise  
    an exception that will be  
    caught in the catch handler.  
    Success;  
end;
```

Note that you must declare a variable of type `FailInfo` in the main procedure. By convention, this variable is called `fi`. To set up the exception handler, you call the procedure `CatchFailures` and pass it the `fi` variable and the name of the nested procedure that serves as the catch handler. Finally, you must call `Success` to let the exception handler know that the main flow of code executed successfully.

The catch handler is always a nested procedure declared like this:

```
procedure HandleFailure (error: integer;  
                        message: longint);
```

Error and message are the values that were passed to the `Failure` routine to raise the exception. Typically, error is a Macintosh error code, and message is an optional application-specific description of what went wrong.

Here's a concrete example:

```
procedure Example;
  var
    anObject: CThing;
    fi: FailInfo;

    procedure HandleFailure (error: integer;
                           message: longint);
    begin
      ForgetObject (anObject);
    end;

  begin
    anObject := NIL;

    CatchFailures(fi, HandleFailure);

    new (anObject);
    anObject.IThing;

    Success'
  end;
```

#### **Example 8-5 A simple exception handler in Pascal**

In Example 8-1, there are two places where an exception can be raised. First, when you create `anObject`, there may not be enough memory. If that's the case, `new` fails and returns `NIL`. Second, an operation in the `IThing` initialization method may raise an exception. In either case, control passes to the catch handler.

If `new` failed, `anObject` will be `NIL`, so the call `ForgetObject` won't do anything, and there's no other cleanup to do. If `new` succeeded, then the exception was caused by some operation in `IThing`. In that case, `ForgetObject` sends a `Dispose` message to `anObject` to release it.

#### **Use handlers only when necessary**

You don't need a catch handler every time you create an object if you can be sure that it will be disposed of properly by another handler. Here's an example of that:

```
procedure CMyApp.CreateDocument;
begin
  var
    newDoc: CMyDoc;
    fi: FailInfo;

    procedure HandleFailure (error: integer;
```

## 8 Exception Handling

```
message: longint);  
begin  
    ForgetObject(newDoc);  
end;  
  
begin  
    newDoc := NIL;  
  
    CatchFailures(fi, HandleFailure);  
  
    new(newDoc);  
    newDoc.IMyDoc(...);  
    newDoc.NewFile;  
  
    Success;  
end;  
  
procedure CMyDoc.NewFile;  
begin  
    BuildWindow;  
    itsWindow.Select;  
end;  
  
procedure CMyDoc.BuildWindow;  
var  
    myPane: CCoolPane;  
    tmpWin: CWindow;  
begin  
    new(tmpWin);  
    itsWindow := tmpWindow;  
    itsWindow.IWindow(...);  
  
    new(myPane);  
    myPane.ICoolPane(...);  
    ...  
end;
```

### Example 8-6 A set of routines with only one catch handler in Pascal

In Example 8-2 only `CMyApp.CreateDocument` needs a catch handler. Here's why. `BuildWindow` creates a window and immediately stores into `itsWindow`, which is an instance variable in `CMyDoc`. `BuildWindow` also creates a pane, which is immediately added into the view hierarchy by `ICoolPane`. If an exception is raised, it will be caught in `CreateDocument`'s catch handler, which calls `ForgetObject`. `ForgetObject` sends a `Dispose` message to `newDoc`, which sends a `Dispose` message to `itsWindow`, which in turn disposes of all the panes that the window encloses.





### Using Exit with exception handlers

If you use the Pascal keyword `Exit` to leave a function or procedure that has a catch handler, be sure that you call `Success` first. Otherwise, the exception mechanism will get confused.

### Special cases

In some abnormal cases, you may want to keep catch handlers from propagating to the top-level handler in `CApplication.Run`. If so, you should insert a label after the call to `Success` in the try handler and goto it from your catch handler.

If you think you can fix whatever raised the exception in your catch handler, you can use the `RetryException` procedure to retry the try handler:

```

procedure Example;
var
    fi: FailInfo;

    procedure HandleFailure (error: integer;
                             message: longint);
    begin
        fix the problem
        RetryException(fi);
    end;

begin
    something that might raise an exception
end;
```

### Example 8-7 Retrying the try handler

## Exception Handler Routines

These routines form the exception handling mechanism. Your program should use the error detection routines to test the return values of Macintosh Toolbox routines.

### Exception handler routines

If you're a Pascal programmer, you should only use `CatchFailures`, `Success`, and `RetryException`. If you're a C programmer, the `TRY`, `CATCH`, and `ENTRY` macros call these routines for you.

---

## ◆ 8 Exception Handling

---

### CatchFailures

```
procedure CatchFailures (var fi: FailInfo;  
    procedure handler (error: integer;  
        message: longint));
```

#### *Pascal only*

This procedure sets up a catch handler. *fi* is a variable of type *FailInfo* declared in the outer procedure or function. *Handler* is the name of a nested procedure that will serve as the catch handler. The catch handler is always a nested procedure declared like this:

```
    procedure HandleFailure (error: integer;  
        message: longint);
```

Error and message are the values that were passed to the *Failure* routine to raise the exception. Typically, error is a Macintosh error code, and message is an optional application-specific description of what went wrong.

### PushTryHandler

#### *C only*

```
void PushTryHandler (FailInfo *fi);
```

Push an exception handler on the exception stack. This function is called by the TRY macro. You should not need to call this function yourself.

### Success

```
procedure Success;  
void Success (void);
```

Pop the most recent exception handler off the exception stack. In THINK Pascal, you should call this procedure after all the code that could have failed has executed successfully. In THINK C, the CATCH macro calls this function for you.

### RetryException

```
procedure RetryException (var fi: FailInfo);  
void RetryException (FailInfo *fi);
```

Retry the try handler. In THINK Pascal you can call this procedure if you can fix the problem in the catch handler. In THINK C, you should use the RETRY macro in the catch handler.

#### **Exception raising routine**

### Failure

```
procedure Failure (error: integer; message: longint);  
void Failure (short error, long message);
```

Raise an exception. Set *gLastError* to error and *gLastMessage* to message. If error is not *kSilentError*, the catch handler in *CApplication's* Run method displays an error alert.



For an explanation of the message parameter, see the description of `ErrorAlert` on page 466

### Error detection routines

These routines test their arguments or check Toolbox error-reporting functions to raise an exception if an error has occurred. You should use these routines any time you call any Toolbox routines that returns an error code, that deals with resources, or moves memory.

#### FailMemError

```
procedure FailMemError;
void FailMemError (void);
```

Raise an exception if the Toolbox function `MemError` returns anything but `noErr`. Calls `Failure(memoryError, 0)`.

#### FailNIL

```
procedure FailNIL (p: UNIV Ptr);
void FailNIL (void *p);
```

Raise an exception if `p` is `NIL`. Calls `Failure(memFullError, 0)`.

#### FailResError

```
procedure FailResError;
void FailResError (void);
```

Raise an exception if the Toolbox function `ResError` returns anything but `noErr`. Calls `Failure(resourceError, 0)`.

#### FailNILRes

```
procedure FailNILRes (resH: UNIV Handle);
void FailMemError (void resH);
```

Raise an exception if `resH`, a resource handle, is `NIL`. If `ResError` returns `noErr`, call `Failure(resNotFound, 0)`. Otherwise, calls `Failure(resourceError, 0)`.

#### FailOSErr

```
procedure FailOSErr (errCode: OSErr);
void FailOSErr (OSErr errCode);
```

Raise an exception if `errCode` is anything but `noErr`. This function calls `Failure(errCode, 0)`.

### Utility routines

#### SpecifyMsg

```
function SpecifyMsg (strListID: integer;
                    strIndex: integer): longint;
long SpecifyMsg (short strListID, short strIndex);
```

Create a message value. `StrListID` is the resource ID of a `STR#` resource, and `strIndex` is the index into that resource. The value returned should be

## ◆ 8 *Exception Handling*

---

passed to `Failure` (or one of the routines that calls `Failure`). See the description of `ErrorAlert` on page 466 for more information.

### **SetFailInfo**

```
procedure SetFailInfo (newError: integer;  
    newMessage: longint);
```

```
void SetFailInfo (short newError, long newMessage);
```

Set the global variables `gLastError` and `gLastMessage`. If `gLastMessage` is not zero, this routine doesn't reset it. `GLastError` is always set.

Although it's usually used internally by the exception handling mechanism, you should call `SetFailInfo(kSilentError, 0)` to prevent the top-level exception handler from displaying an error message.

# CAbstractText

---

9



## Introduction

CAbstractText is an abstract class that provides you with a template for creating a pane that displays text.

## Heritage

Superclass  
Subclasses

CPanorama  
CEditText

## Using CAbstractText

CAbstractText is an abstract class for creating panes that display text. A text pane is usually the panorama in a scroll pane so you can scroll through the text. The `Specify` method, on page 120, lets you choose whether the user can edit and copy your text pane's text. The `DoCommand` method, on page 122, handles all the common text editing commands such as cutting and pasting, font selection, line spacing, etc.

To handle the **Undo** command, CAbstractText creates task objects for many common commands and sends messages to the object telling it to perform or undo its task. `CTextEditTask` on page 421 handles typing, cutting, copying, and pasting. `CTextStyleTask` on page 427 handles font, size, and style commands.

When you perform a task that can be undone, the name of the **Undo** command changes, for example **Undo Typing**. The **Undo** labels for this CAbstractText are stored in the resource STR# 130. The index of the first string is stored in the class variable `cFirstTaskIndex`. By default, it's 1. If you put the strings in a different place in STR# 130, you should set this variable to the index of the first string. If you don't use **Undo** labels, set this variable to 0. The labels must be in this order: Typing, Cut, Copy, Paste, Clear, Formatting.

## 9 CAbstractText

Since it's an abstract class, you can't create an instance of CAbstractText. You can use the CEditText or CStyleText classes that come with the THINK Class Library.

These classes are based on the Macintosh TextEdit (TE) routines and become slow if they contain over 4000 characters. If your text pane needs to handle more text, create your own subclass of CAbstractText.

Most of CAbstractText's methods do nothing. They are "place holders" for the methods your subclass must override. The rest of CAbstractText's methods call them. These are the methods you must override:

DoClick	SetTextPtr
GetTextHandle	SetFontNumber
SetFontStyle	SetFontSize
SetTextMode	SetAlignCmd
SetAlignCmd	SetSpacingCmd
GetHeight	GetCharOffset
GetCharPoint	GetTextStyle
GetCharStyle	FindLine
GetLength	TypeChar
SetSelection	GetSpacingCmd
GetSelection	GetNumLines
CopyTextRange	PerformEditCommand
InsertTextPtr	Dawdle

These two methods assume that the text in your pane is stored in a single contiguous buffer. If it's not, you should override them:

GetCharBefore	GetCharAfter
---------------	--------------

### Variables

This is an instance variable that any class can use.

Variable	Type	Description
itsTypingTask	CTextEditTask	Active typing task.

CFirstTaskIndex is a class variable that only CAbstractText and its subclasses should use. In THINK C, it's a class variable.

Variable	Type	Description
cFirstTaskIndex	short	Index in STR# 130 resource of first undo label

These are internal instance variables which only subclasses of CAbstractText should use. In THINK C, they are protected.

Variable	Type	Description
lineWidth	integer	Width of a text line in pixels. If negative, lines are as wide as the pane.
fixedLineHeights	Boolean	TRUE if all lines same height.
wholeLines	Boolean	TRUE if lines aren't cut off vertically.
lastFontNum	integer	Last font number.
lastFontCmd	longint	Last font command number.
lastTextSize	integer	Last seen text size
lastSizeCmd	longint	Last size command number.
editable	Boolean	TRUE if user can edit text.
stylable	Boolean	TRUE if user can change font, size, or style of text.

## Methods

### Construction and destruction methods

#### IAbstractText

```

procedure IAbstractText (anEnclosure: CView;
    aSupervisor: CBureaucrat;
    aWidth, aHeight: integer;
    aHEncl, aVEncl: integer;
    aHSizing, aVSizing: SizingOption;
    aLineWidth: integer);

void IAbstractText (CView *anEnclosure,
    CBureaucrat *aSupervisor,
    short aWidth, short aHeight,
    short aHEncl, short aVEncl,
    SizingOption aHSizing, SizingOption aVSizing,
    short aLineWidth);

```

Initialize an abstract text pane. Most of the arguments to this method are identical to pane initialization. ALineWidth specifies how wide the lines should be. If it's less than zero, the text wraps to the width of the pane.

## 9 *CAbstractText*

---

### Note

The descriptions of the other arguments are in CPane.

---

### IViewRes

```
procedure IViewRes (rType:ResType; resID: integer;  
    anEnclosure: CView; aSupervisor: CBureaucrat);
```

```
void IViewRes (ResType rType, short resID,  
    CView *anEnclosure, CBureaucrat *aSupervisor);
```

Initialize edit text pane from a resource template. RType is the resource type for the CView subclass you want to initialize. ResID is the resource ID of the resource. AnEnclosure and aSupervisor are the enclosure and supervisor of the pane.

To initialize a text pane from a resource file, use a 'AbTx' resource.

---

### Note

Earlier versions of the THINK Class Library used 'StTx' resources to initialize static text and edit text panes. You should make sure that any old programs do not use 'StTx'.

---

### Free/Dispose

```
procedure Free;
```

```
void Dispose (void);
```

Dispose of an abstract text object.

### Accessing method

### Specify

```
procedure Specify (fEditable, fSelectable, fStylable:  
    Boolean);
```

```
void Specify (Boolean fEditable,  
    Boolean fSelectable, Boolean fStylable);
```

Specify whether the user can edit, select, and style text. This method sets editable to fEditable, wantsClicks to fSelectable, and stylable to fStylable. By default, text is editable, selectable, and stylable. If editing is allowed, you can insert and delete text. If selecting is allowed, you can select text to copy it to the clipboard. If styling is allowed, the user can change have change the font, style, or size of the text.

To make your Specify calls more understandable, the THINK Class Library defines some constants you can use instead of TRUE and FALSE:

kEditable	kNotEditable
kSelectable	kNotSelectable
kStylable	kNotStylable.





This list describes some common ways to call `Specify`:

- To let the user edit and style text, call `Specify(kEditable, kSelectable, kStylable)`.
- To prevent the user from editing, styling, or selecting text, call `Specify(kNotEditable, kNotSelectable, kNotStylable)`.
- To let the user select text (to copy it, for example) but not edit or style it, call `Specify(kNotEditable, kSelectable, kNotStylable)`.
- To let the user edit text, but not style it, call `Specify(kEditable, kSelectable, kNotStylable)`.
- To let the user style text, but not edit it, call `Specify(kNotEditable, kSelectable, kStylable)`.

---

**Note**

If `fSelectable` is `kNotSelectable`, `Specify` disables editing and styling, regardless of the values of `fEditable` and `fStylable`.

---

You can call `Specify` at any time. For example you could lock text (that is, disable editing) to prevent accidental editing, then let the user unlock it later to edit it.

**GetSpecification**

```
procedure GetSpecification (var fEditable,  
    fSelectable, fStylable: Boolean);  
void GetSpecification (Boolean *fEditable,  
    Boolean *fSelectable, Boolean *fStylable);  
Return whether this pane is editable, stylable, and selectable.
```

### Command methods

#### DoCommand

```
procedure DoCommand (theCommand: longint);  
void DoCommand (long theCommand);
```

This method handles the common commands that apply to text: cutting and pasting, font selection, styling, alignment, and spacing. DoCommand handles these commands: (It passes all other commands to its supervisor.)

- Editing commands: cmdCut, cmdCopy, cmdPaste, cmdClear, cmdSelectAll
- Style commands: cmdPlain, cmdBold, cmdItalic, cmdUnderline, cmdOutline, cmdShadow, cmdCondense, cmdExtend
- Alignment commands: cmdAlignLeft, cmdAlignCenter, cmdAlignRight
- Spacing commands: cmdSingleSpace, cmd1HalfSpace, cmdDoubleSpace.
- Font selection from the **Font** menu (MENUfont)
- Size selection from the **Size** menu (MENUsize)

To let the user undo actions, this method creates a task object for most commands. It creates a CTextEditTask object for an editing command and a CTextStyleTask object for a style, alignment, spacing, font, or size command.

#### UpdateMenus

```
procedure UpdateMenus;  
void UpdateMenus (void);
```

Update the menus that have to do with text processing right before the menu appears. This method enables the **Cut**, **Copy**, and **Clear** commands if there is a current selection and enables the **Paste** command if there is text in the clipboard. This menu also checks the current font, size, style, alignment, and spacing commands.

---

#### Note

See the implementation of this method if you want an example of how to write your own UpdateMenus method.

---

**DoKeyDown**

```
procedure DoKeyDown (theChar: char; keyCode: Byte;
    macEvent: EventRecord);
```

```
void DoKeyDown (char theChar, Byte keyCode,
    EventRecord *macEvent);
```

Handle a key down in a text pane. Usually, this passes function and command keys to its superclass and inserts other characters into the text. This table shows how it handles command keys and function keys:

**If the key is...**

Any command key  
Home, Page Up, Page Down  
End

A cursor key

Any other key

**This method...**

Passes it to the superclass.

Passes it to the superclass.

Scrolls to the bottom left of the text.

Calls TypeChar and notifies the editing task of the change in the selection.

Passes it to the editing task, which should insert the character if the text pane is editable.

**DoAutoKey**

```
procedure DoAutoKey (theChar: char; keyCode: Byte;
    macEvent: EventRecord);
```

```
void DoAutoKey (char theChar, Byte keyCode,
    EventRecord *macEvent);
```

Handle an auto-key in a text pane. If the command key is down, this method does nothing. Otherwise, this method calls DoKeyDown.

**PerformEditCommand**

```
procedure PerformEditCommand (theCommand: longint);
```

```
void PerformEditCommand (long theCommand);
```

Perform the standard cut, copy, paste, and clear commands on the text. The task classes call this method to perform and undo an edit command. The default method does nothing. Your subclass must override this method.

**TypeChar**

```
procedure TypeChar (theChar: char;
    theModifiers: integer);
```

```
void TypeChar (char theChar, short theModifiers);
```

Process a keystroke. This method does not need to set up for an **Undo** command and should handle the key directly. The task classes call this method to perform a key stroke. The default method does nothing. Your subclass must override this method.

## ◆ 9 CAbstractText

---

<b>SelectionChanged</b>	<pre>procedure SelectionChanged; void SelectionChanged (void);</pre> <p>The selection has just changed. This method sends a BroadcastChange message with textSelectionChanged as the reason. If this text pane has an editing task, it sends the task a SelectionChanged message.</p>
<b>MakeEditTask</b>	<pre>function MakeEditTask (editCmd: longint):     CTextEditTask; CTextEditTask *MakeEditTask (long editCmd);</pre> <p>Create a task to handle typing and the cut, copy, paste, and clear commands. This method creates a CTextEditTask class. If you override this method, it must create a subclass of CTextEditTask. In THINK C MakeEditTask is a protected method.</p>
<b>MakeStyleTask</b>	<pre>function MakeStyleTask (styleCmd: longint):     CTextStyleTask; CTextStyleTask *MakeStyleTask (long editCmd);</pre> <p>Create a task to handle font, style, size, alignment, and spacing commands. This method creates a CTextStyleTask class. If you override this method, it must create a subclass of CTextStyleTask. In THINK C MakeStyleTask is a protected method.</p>
<b>BecomeGopher</b>	<pre>procedure BecomeGopher (fBecoming: Boolean); void BecomeGopher (Boolean fBecoming);</pre> <p>If fBecoming is TRUE, this text pane is becoming the gopher. This method activates it and enables the <b>Font</b>, <b>Size</b>, and <b>Style</b> menus. Otherwise, this text pane is no longer the gopher. This method deactivates the pane and disables the <b>Font</b>, <b>Size</b>, and <b>Style</b> menus.</p>
	<b>Display methods</b>
<b>ScrollToSelection</b>	<pre>procedure ScrollToSelection; void ScrollToSelection (void);</pre> <p>Scroll so the current selection is visible.</p>
<b>SetSelection</b>	<pre>procedure SetSelection (selStart, selEnd: long;     fRedraw: Boolean); void SetSelection (long selStart, long selEnd,     Boolean fRedraw);</pre> <p>Set the selection to the range corresponding to character positions selStart through selEnd. The default method does nothing. Your subclass must override this method.</p>



<b>GetSelection</b>	<pre>procedure GetSelection (var selStart, selEnd:     longint); void GetSelection (long *selStart, long *selEnd);</pre> <p>Return the start and end of the current selection. The default method does nothing. Your subclass must override this method.</p>
<b>SelectAll</b>	<pre>procedure SelectAll (fRedraw: Boolean); void SelectAll (Boolean fRedraw);</pre> <p>Select all the text in this text pane. This method uses <code>GetLength</code> to determine the length of the text.</p>
<b>Paginate</b>	<pre>procedure Paginate (aPrinter CPrinter;     pageWidth, pageHeight: integer); void Paginate (struct CPrinter *aPrinter,     short pageWidth, short pageHeight);</pre> <p>Split the text pane into pages. This method makes sure that lines aren't cut in half horizontally at the ends of pages.</p>
<b>Text specification methods</b>	
<b>SetTextString</b>	<pre>procedure SetTextString (textStr: Str255); void SetTextString (Str255 textStr);</pre> <p>Use the specified string as the text pane's text. This method uses <code>SetTextPtr</code> to set the text.</p>
<b>SetTextHandle</b>	<pre>procedure SetTextHandle (textHand: Handle); void SetTextHandle (Handle textHand);</pre> <p>Use the text <code>textHand</code> as the text for this text pane. This method uses <code>SetTextPtr</code> to set the text.</p>
<b>SetTextPtr</b>	<pre>procedure SetTextPtr (textPtr: Ptr;     numChars: longint); void SetTextPtr (Ptr textPtr, long numChars);</pre> <p>Use the first <code>numChars</code> characters that <code>textPtr</code> points to as the text for this abstract text object. The default method does nothing. Your subclass must override this method so it makes a copy of the text.</p>
<b>GetTextHandle</b>	<pre>function GetTextHandle: CharsHandle; CharsHandle GetTextHandle (void);</pre> <p>Return a handle to the text of the abstract text. The default method does nothing. Your subclass must override this method. In most cases, the text</p>

## 9 *CAbstractText*

---

class should keep its text in a handle, and this method should return that handle—not a copy of it.

If your implementation requires that this method return a copy of the text in a handle, your subclass should override any methods that call `GetTextHandle` to dispose of the handle. In `CAbstractText`, the methods `GetCharBefore` and `GetCharAfter` expect `GetTextHandle` to return a handle to the actual text.

### **CopyTextRange**

```
function CopyTextRange (start, end: long): Handle;  
Handle CopyTextRange (long start, long end);
```

Return a copy of the text specified by start and end. The default method does nothing. Your subclass must override this method.

### **InsertTextPtr**

```
procedure InsertTextPtr (text: Ptr; length: longint;  
    fRedraw: Boolean);  
void InsertTextPtr (Ptr text, long length,  
    Boolean fRedraw);
```

Insert a copy of the given text at the start of the selection. Text is a pointer to the text. The default method does nothing. Your subclass must override this method.

### **InsertTextHandle**

```
procedure InsertTextHandle (text: Handle;  
    fRedraw: Boolean);  
void InsertTextHandle (Handle text, Boolean fRedraw);  
Insert a copy of the given text at the start of the selection. Text is a handle to the text.
```

### **GetCharBefore**

```
procedure GetCharBefore (var aPosition: longint;  
    var charBuf: tCharBuf);  
void GetCharBefore (long *aPosition,  
    tCharBuf charBuf);
```

Return the character before aPosition. The character and its size are returned in charBuf, and aPosition is updated with the starting position of the character. If there is no character after aPosition, then this method sets the length of the character to 0 (zero) and does not update aPosition.

### **GetCharAfter**

```
procedure GetCharAfter (var aPosition: longint;  
    var charBuf: tCharBuf);  
void GetCharAfter (long *aPosition,  
    tCharBuf charBuf);
```

Return the character after aPosition. The character and its size are returned in charBuf, and aPosition is updated with the starting position

of the character. If there is no character before `aPosition`, then this method sets the length of the character to 0 (zero) and does not update `aPosition`.

### Text characteristics methods

#### **SetFontNumber**

```
procedure SetFontNumber (aFontNumber: integer);  
void SetFontNumber (short aFontNumber);
```

Set the font by font number. The default method does nothing. Your subclass must override this method. If your subclass supports multiple fonts in a pane, this method should set the font for the current selection. Otherwise, it should set the font for the whole text pane.

#### **SetFontName**

```
procedure SetFontName (aFontName: Str255);  
void SetFontName (Str255 aFontName);
```

Set the font by font name.

#### **SetFontStyle**

```
procedure SetFontStyle (aStyle: Style);  
void SetFontStyle (short aStyle);
```

Set the font style. `AStyle` may be one of: bold, italic, underline, outline, shadow, condense, or extend. The default method does nothing. Your subclass must override this method. If your subclass supports multiple styles in a pane, this method should set the style for the current selection. Otherwise, it should set the style for the whole text pane.

#### **SetFontSize**

```
procedure SetFontSize (aSize: integer);  
void SetFontSize (short aSize);
```

Set the font size to the specified size. The default method does nothing. Your subclass must override this method. If your subclass supports multiple styles in a pane, this method should set the size for the current selection. Otherwise, it should set the size for the whole text pane.

#### **SetTextMode**

```
procedure SetTextMode (aMode: integer);  
void SetTextMode (short aMode);
```

Set the text mode to the specified mode. `AMode` can be one of `srcOr`, `srcXor`, or `srcBic`. The default method does nothing. Your subclass must override this method.

#### **SetAlignCmd**

```
procedure SetAlignCmd (anAlignment: long);  
void SetAlignCmd (long anAlignment);
```

Set the text alignment. `AnAlignment` can be one of `cmdAlignLeft`, `cmdAlignRight`, or `cmdAlignCenter`. The default method does nothing.

## ◆ 9 *CAbstractText*

---

ing. Your subclass must override this method. If your subclass supports multiple alignments in a pane, this method should set the alignment for the current selection. Otherwise, it should set the alignment for the whole text pane.

### **GetAlignCmd**

```
function GetAlignCmd: longint;  
long GetAlignCmd (void);
```

Return the current alignment. It can be one of `cmdAlignLeft`, `cmdAlignRight`, `cmdAlignCenter`, or `cmdNull`. The default method does nothing. Your subclass must override this method. If your subclass allows different alignments within a text pane and the current selection contains more than one alignment, this method should return `cmdNull`.

### **SetSpacingCmd**

```
procedure SetSpacingCmd (aSpacingCmd: longint);  
void SetSpacingCmd (long aSpacingCmd);
```

Set the space between lines of text. `ASpacingCmd` can be one of `cmdSingleSpace`, `cmdHalfSpace`, or `cmdDoubleSpace`. The default method does nothing. Your subclass must override this method.

### **GetSpacingCmd**

```
function GetSpacingCmd: longint;  
long GetSpacingCmd (void);
```

Return the space between lines of text. It can be one of `cmdSingleSpace`, `cmdHalfSpace`, or `cmdDoubleSpace`. The default method does nothing. Your subclass must override this method.

### **GetHeight**

```
function GetHeight (startLine endLine: longint):  
    longint;  
long GetHeight (long startLine, long endLine);
```

Return the total height in pixels of the indicated lines of text. The default method does nothing. Your subclass must override this method.

### **Get1Height**

```
function Get1Height (aLineNum: longint): integer;  
short Get1Height (long aLineNum);  
Return the height in pixels of the indicated line of text.
```

### **GetCharOffset**

```
function GetCharOffset (aPt: LongPt): longint;  
long GetCharOffset (LongPt *aPt);
```

Return the character position nearest the coordinate `aPt`. `APt` must be in frame coordinates. The default method does nothing. Your subclass must override this method.





<b>GetCharPoint</b>	<pre>procedure GetCharPoint (offset: long;     var aPt: LongPt); void GetCharPoint( long offset, LongPt *aPt);</pre> <p>Return the coordinates of the character position offset. APT is in frame coordinates. The default method does nothing. Your subclass must override this method.</p>
<b>GetCharStyle</b>	<pre>procedure GetCharStyle (charOffset: long;     var theStyle: TextStyle); void GetCharStyle (long charOffset,     TextStyle *theStyle);</pre> <p>Return style information for the character at position charOffset.</p>
<b>GetTextStyle</b>	<pre>procedure GetTextStyle (var whichAttributes: integer;     var aStyle: TextStyle); void GetTextStyle (short *whichAttributes,     TextStyle *aStyle);</pre> <p>Return current style information. WhichAttributes is a flag that indicates which text attributes to report on. The default method does nothing. Your subclass must override this method. If your subclass supports multiple styles in a text pane, this method should report only on the attributes that are continuous over the current selection, and set whichAttributes to those attributes.</p> <p>The attributes flags and TextStyle record are described in <i>Inside Macintosh VI</i>, Chapter 15, "TextEdit."</p>
<b>Calibrating methods</b>	
<b>SetWholeLines</b>	<pre>procedure SetWholeLines (aWholeLines: Boolean); void SetWholeLines (Boolean aWholeLines);</pre> <p>If aWholeLines is TRUE, this method will resize the frame so it will display only whole lines. The text pane is not redrawn.</p>
<b>GetWholeLines</b>	<pre>function GetWholeLines: Boolean; Boolean GetWholeLines (void);</pre> <p>Return the value of the wholeLines instance variable. If TRUE, the frame is set so it displays only whole lines and not partial lines.</p>

## 9 *CAbstractText*

---

### **ResizeFrame**

```
procedure ResizeFrame (delta: Rect);  
void ResizeFrame (Rect *delta);
```

Resize the frame when the size of its pane changes. The delta rectangle specifies how each side changes. Positive values mean down and to the right. Negative values mean up and to the left.

### **FindLine**

```
function FindLine (charPos: integer): longint;  
long FindLine (short charPos);
```

Return the line number containing the specified character position. The default method does nothing. Your subclass must override this method.

Both line and character numbering start at zero. If the character position is before the start of the text (a negative number), this method should return zero. If the character position is beyond the end of the text, this method should return the number of the last line.

### **GetLength**

```
function GetLength: longint;  
long GetLength (void);
```

Return the length in bytes of the text buffer. The default method does nothing. Your subclass must override this method.

### **GetNumLines**

```
function GetNumLines: longint  
long GetNumLines (void);
```

Return the total number of lines in the text buffer. The default method does nothing. Your subclass must override this method.

### **Cursor method**

### **AdjustCursor**

```
procedure AdjustCursor (where: Point;  
    mouseRgn: RgnHandle);  
void AdjustCursor (Point where, RgnHandle mouseRgn);
```

Turn the cursor into an I-beam when the mouse is in the edit text pane.

# CAppleEvent

## 10

### Introduction

CAppleEvent is a class that contains an AppleEvent and its default reply. The THINK Class Library handles the four required AppleEvents (Open Application, Open Documents, Print Documents, and Quit Application) and lets you use others.

### Heritage

Superclass	CObject
Subclasses	None

### Using CAppleEvent

When your application receives an AppleEvent, the THINK Class Library puts in a CAppleEvent object and passes it to the gopher. You don't need to write any code to handle the four required AppleEvents. The application handles them by default.

If your application uses an AppleEvent besides the four required ones, you need to write a DoAppleEvent method to handle it. That method should be in the class that handles the event: your subclass of CPane, CDocument or CApplication. You don't need to subclass CAppleEvent to handle new AppleEvents.

To extract the parameters from your AppleEvent, you can send it either a GetDescList message, on page 134, which returns items in a AEDescList, or an ExtractFromDescList message, on page 134, which returns its items in a CArray object. After you extract all the parameters you know about, send your AppleEvent a GotRequiredParam message, on page 134, which returns TRUE if you extracted all the required parameters.

If your AppleEvent needs to interact with the user (e.g., display a dialog), send it the RequestInteraction message, on page 134. The THINK

## ◆ 10 CAppleEvent

---

Class Library assumes that the AppleEvents Open Application and Open Documents do not require user interaction. If they do, you must override CApplication's DoAppleEvent method

When your AppleEvent needs to return a result, send it a SetErrorResult message, described on page 135. If you need to store an error string or return data, send your AppleEvent object a GetAEReply message, described on page 134, and store that information directly in the reply.

### How the THINK Class Library handles AppleEvents

When you initialize your application, it checks to see if the AppleEvent manager is installed. If it is, the switchboard installs a handler in the AppleEvents dispatch table, called AppleEventHandler, that handles all AppleEvents.

When your application is running and the switchboard receives a high-level event, it assumes the event is an AppleEvent and calls AEPProcessAppleEvent. That function, in turn, calls the handler AppleEventHandler, which the switchboard installed earlier. The handler sends the switchboard a DoAppleEvent message. That method packages the AppleEvent and sends it to the gopher, which finds who will handle it.

### Variables

These are internal instance variables which only subclasses of CAppleEvent should use. In THINK C, they are protected.

Variable	Type	Description
theEvent	AppleEvent	The AppleEvent.
theReply	AppleEvent	The default reply.
theRefCon	longint	The reference value.
eventClass	DescType	The event class; e.g., 'aevt'.
eventID	DescType	The event ID; e.g., 'oapp'.
canInteract	Boolean	TRUE if interaction with the user has previously been requested and approved.
errCode	OSErr	The error code for this event.



Variable	Type	Description
notificationRec	NMRec	The Notification Manager record for AEInteractWithUser. By default, it's NULL
idleProc	IdleProcPtr	The idle procedure for AEInteractWithUser. If NULL, uses CSwitchboard's idle procedure

## Methods

### Creation and destruction

#### IAppleEvent

```
procedure IAppleEvent (theEvent, theReply: AppleEvent,
    theRefCon: longint, eventClass, eventID: DescType);
void IAppleEvent (const AppleEvent *theEvent,
    const AppleEvent *theReply, long theRefCon,
    DescType eventClass, DescType eventID);
```

Initialize a CAppleEvent object. TheEvent is the AppleEvent that CSwitchboard received. TheReply is the default reply that the AppleEvent manager generates for an AppleEvent. TheRefCon is the reference value for this event's class and ID. EventClass and eventID are the event class and ID from theEvent.

### Accessing methods

#### GetEventClass

```
function GetEventClass: DescType;
DescType GetEventClass (void);
```

Return the event class, e.g. 'aevt'. The event class and event ID uniquely identify an AppleEvent.

#### GetEventID

```
function GetEventID: DescType;
DescType GetEventID (void);
```

Return the event ID, e.g. 'oapp'. The event class and event ID uniquely identify an AppleEvent

#### GetAEEEvent

```
function GetAEEEvent: AppleEvent;
AppleEvent GetAEEEvent (void);
```

Return the AppleEvent.

## ◆ 10 CAppleEvent

---

<b>GetAEReply</b>	<pre>function GetAEReply: AppleEvent; AppleEvent GetAEReply (void);</pre> <p>Return the AppleEvent reply.</p>
<b>GetAERefCon</b>	<pre>function GetAERefCon: longint; long GetAERefCon (void);</pre> <p>Return the AppleEvent reference value for this event's class and ID. By default, this is always zero in the THINK Class Library. To use other values, override the ISwitchboard method in CSwitchboard, and change its call to AEInstallEventHandler.</p>
<b>GetDescList</b>	<pre>procedure GetDescList (whichParam: AEKeyword; var descList: AEDescList); void GetDescList (AEKeyword whichParam, AEDescList *descList);</pre> <p>Return the AEDescList associated with the keyword whichParam.</p>
<b>ExtractFromDescList</b>	<pre>function ExtractFromDescList (whichParam: AEKeyword; itemType: DescType; itemSize: Size): CArray; CArray *ExtractFromDescList (AEKeyword whichParam, DescType itemType, Size itemSize);</pre> <p>Get the descriptor list associated with the parameter whichParam, and puts its items into a CArray. This method is especially useful if all the items in the list are the same size and the data in the list is not keyword separated. This method does not use the keywords.</p>
<b>GotRequiredParams</b>	<pre>function GotRequiredParams: Boolean; Boolean GotRequiredParams (void);</pre> <p>Does the recommended check to determine if all the required parameters of an AppleEvent have already been extracted. You should call this method after you extract all the parameters you know about and before you actually handle the event.</p>
<b>RequestInteraction</b>	<pre>function RequestInteraction (timeOutTicks: longint): OSErr; OSErr RequestInteraction (long timeOutTicks);</pre> <p>Indicates that you need to interact with the user (e.g. put up a dialog) to handle the event. If interaction is allowed and the application is brought forward before the timeout period has elapsed, then this method returns noErr and you can proceed with the interaction. If any other value is returned, then your attempt to interact failed.</p>

This method calls `AEInteractWithUser`, with `timeOutTicks`, an idle procedure, and a pointer to a Notification Manager record. By default, the idle procedure sends the switchboard an `AppleEventIdle` message, described on page 408. To use a different idle procedure, assign it to the `idleproc` instance variable. Also, the Notification Manager record pointer is `NIL` by default, so `AEInteractWithUser` supplies its own default record. To use your own record, assign it to the `notificationRec` instance variable.

**SetErrorResult**

```
procedure SetErrorResult (anErrorCode: OSErr);  
void SetErrorResult (OSErr anErrorCode);
```

Set the result code for this event. If you successfully handle this function, set the result code to `noErr`. If you try to handle this function and fail, set the result code to appropriate error code.

**GetErrorResult**

```
function GetErrorResult: OSErr;  
OSErr GetErrorResult (void);
```

Return the error code for this event. If the event was handled successfully, it will be `noErr`. If the event was not handled, it will be `errAEEventNotHandled`. If someone attempted to handle it, failed, and stored a result code, it will be some other result code. `CSwitchboard` uses a special exception handler to trap exceptions raised during the handling of an `AppleEvent`, and will return the exception error instead.

## ◆ 10 CAppleEvent

---



# Application

---

# 11



## Introduction

Every program must create a subclass of `CApplication` and create only one instance of this class. The application is the highest level in the chain of command. It is the only bureaucrat without a supervisor.

## Heritage

Superclass  
Subclasses

`CDirectorOwner`  
You must define a subclass of this class.

## Using `CApplication`

`CApplication` is one of the classes you must override to implement an application with the THINK Class Library. An application usually has one or more subordinate **directors** which handle the interaction between commands and windows. The most common kind of director is a **document** which, in addition to a window, also handles a file.

### The application and the chain of command

The application is the root of the chain of command. If the current command object, or **bureaucrat**, (stored in the global variable `gGopher`) can't handle a command, it passes the command to its supervisor. If no bureaucrat can handle the message, it ends up with the application. If the application doesn't handle a command, the message is ignored.

### Writing the main program

Your main program must create an application object and assign it to the global variable `gApplication`. After initializing your application, send it application a `Run` message to get it started. Finally, send your application an

## 11 CApplication

Exit (or ExitApp in Pascal) message to give it a chance to clean up after itself. In THINK Pascal, your main routine should look like this:

*In this example, CYourApp is the name of your application subclass.*

```
begin
    new(CYourApp(gApplication));
    CYourApp(gApplication).IYourApp;
    gApplication.Run;
    gApplication.ExitApp;
end.
```

And in THINK C, your main routine should look like this:

```
void main()
{
    gApplication = new(CYourApp);
    ((CYourApp *)gApplication)->IYourApp();
    gApplication->Run();
    gApplication->Exit();
}
```

Your application subclass must override or define these five methods:

*initialization method* (IYourApp in the example)  
OpenDocument  
SetUpFileParameters  
DoCommand  
CreateDocument

Of course, your subclass can override additional methods if it needs to.

### Handling low memory situations

*The term "rainy day fund" comes from the expression "saving for a rainy day." It means to save money in case of disaster.*

The CApplication class provides several methods that deal with low memory situations. These methods use a memory reserve called the **rainy day fund**.

When you call IApplication, you specify how much memory your application should allocate for the rainy day fund. You also specify how many bytes of that fund should be available to satisfy Toolbox routine memory requests and how many bytes of that fund should be available to satisfy critical operation requests. These values are stored in the instance variables `toolBoxBalance` and `criticalBalance`.

If the Macintosh Memory Manager gets a request for more memory than is available, it calls a grow zone function. In the THINK Class Library, the grow zone function sends the application a `GrowMemory` message.

The `GrowMemory` method tries several strategies to free memory in the heap. First, it sends the application a `MemoryShortage` message. Your ap-



plication subclass should override this method to release memory that is not crucial to execution. For instance, your `MemoryShortage` method might release a code segment it's not using, or it might dispose of a buffer it no longer needs. If the `MemoryShortage` method wasn't able to release enough memory, `GrowMemory` starts using the rainy day fund.

`GrowMemory` looks first at the `inCriticalOperation` instance variable to release memory from the rainy day fund:

<b>If <code>inCriticalOperation</code> is</b>	<b>leave this much in reserve</b>
TRUE	<code>toolboxBalance</code>
FALSE	<code>criticalBalance</code>

A critical operation is a temporary operation that takes place in a single Macintosh event, that might take a lot of memory, and that should not cause the application to crash. For instance, saving a file is a critical operation. You can use the routine `SetCriticalOperation` to set the `inCriticalOperation` flag.

If `GrowMemory` still wasn't able to release enough memory and the `canFail` flag is TRUE, `GrowMemory` returns without trying to allocate any more memory. If the `canFail` flag is TRUE, the application is saying that it can deal with a failing memory request.

If the `canFail` flag is FALSE, `GrowMemory` tries to release all the memory left in the rainy day fund because the application is not prepared to deal with a failing memory request. If there is not enough memory, even after using the rainy day fund, it is likely that the application will crash.

You can use the routine `SetAllocation` to set and reset the `canFail` flag. In general, your application should be prepared to handle failing memory requests.

## Variables

The global variable `gApplication` points to your application object. The method `IAApplication` sets this variable.

The instance variables of the application class handle events, memory shortage situations, standard file parameters, and flow of control. Your application subclass may define additional instance variables.

## 11 CApplication

### Global variable

Variable	Type	Description
gApplication	CApplication	The single instance of your application object.
gSystem	tSystem	A global record with information about the Macintosh and System that your program is running with.

Only methods of CApplication and its subclasses should use this variable. In THINK C, it's a class variable.

Variable	Type	Description
cMaxSleepTime	long	default for WaitNextEvent

### Event related instance variables

The event related instance variables handle the interaction between the Macintosh Toolbox Event Manager and your application. Your application subclass should not manipulate these variables.

Variable	Type	Description
itsSwitchboard	CSwitchboard	Points to the event-processing object.
itsIdleChores	CList	Chores to perform at idle time.
itsUrgentChores	CCluster	Chores to perform after the current event.
urgentsToDo	Boolean	Are any urgent chores pending?
running	Boolean	If TRUE, the program is still running.
unhandledTask	CTask	Task that no document handled

### Phase related instance variables

This variable tells you what phase your application is in.



Variable	Type	Description
phase	integer	The phase the application is in.

These are its possible values for phase:

Value	Description
appInitializing	The application is initializing.
appRunning	The application is running.
appQuitting	The application is quitting.

### Memory related instance variables

The memory related variables are used when your application is running out of memory. The `rainyDayFund` variable specifies the number of bytes to set aside to use when your application runs into a critical memory situation. You specify this amount in your application initialization method.

Variable	Type	Description
rainyDayFund	Size	Bytes of memory to set aside for critical memory situations.
criticalBalance	Size	Portion of rainyDayFund to use for critical operations.
toolboxBalance	Size	Portion of rainyDayFund to use for Toolbox operations that must not fail.
tempAllocation	Size	Portion of rainyDayFund that a routine has borrowed.
rainyDay	Handle	Handle to the reserve memory.
rainyDayUsed	Boolean	Has rainy day fund been used?
memWarningIssued	Boolean	Has the user been alerted?
canFail	Boolean	Is it OK for memory request to fail?
inCriticalOperation	Boolean	Is it OK to use up to criticalBalance bytes from reserve?

## 11 CApplication

### Standard file instance variables

The standard file variables are used in the ChooseFile method as parameters to the Macintosh Toolbox SFPGetFile routine. You set these variables in your SetUpFileParameters method.

Variable	Type	Description
sfNumTypes	integer	Number of file types recognized
sfFileTypes	SFTypeList	File types that the application recognizes
sfFileFilter	FileFilterProcPtr	Filter for files to display
sfGetDLOGHook	DlgHookProcPtr	Hook for handling get dialog
sfGetDLOGid	integer	Dialog resource ID for get file. Default is getDlgID (-4000).
sfGetDLOGFilter	ModalFilterProcPtr	Filter for get dialog events

### Methods

The methods of the CApplication class deal with application initialization and document handling. Since the application is the ultimate supervisor for all the command objects, it also serves as the end of the chain of command.

#### Initialization methods

The initialization methods set up the application's memory and file parameters. Your application subclass should implement its own initialization method and call the default initialization method.

#### IApplication

```
procedure IApplication (extraMasters: integer;  
    aRainyDayFund, aCriticalBalance,  
    aToolboxBalance: Size);  
  
void IApplication (short extraMasters,  
    Size aRainyDayFund, Size aCriticalBalance,  
    Size aToolboxBalance);
```

This is the main initialization method. In your application subclass, you should implement a method called *IMyAppClass*, where *MyAppClass* is the name of your application subclass.



The `extraMasters`, `aCriticalBalance`, and `aToolboxBalance` parameters are passed to the `InitMemory` method described later on.

Your application subclass must call `CApplication::IApplication` (in THINK Pascal) or `CApplication::IApplication` (in THINK C) in its initialization method. If your application subclass defines instance variables, your initialization method should initialize the variables.

This method also initializes these global variables. All the global variables are described in detail in Chapter 72, "Global Variables."

<code>gApplication</code>	<code>gSignature</code>
<code>gSleepTime</code>	<code>gIBeamCursor</code>
<code>gWatchCursor</code>	<code>gUtilRgn</code>
<code>gGopher</code>	<code>gLastViewHit</code>
<code>gClicks</code>	<code>gSystem</code>

Finally, this method sends the following messages to your application object. The rest of this section goes into detail for each method.

<code>InitToolbox</code>	<code>InitMemory</code>
<code>InspectSystem</code>	<code>InstallPatches</code>
<code>MakeSwitchboard</code>	<code>MakeError</code>
<code>MakeDesktop</code>	<code>MakeClipboard</code>
<code>MakeDecorator</code>	<code>SetUpFileParameters</code>
<code>SetUpMenus</code>	

## InitToolbox

```
procedure InitToolbox;
void InitToolbox (void);
```

This method initializes all of the Macintosh Toolbox managers. Your application subclass should not need to override this method.

## InitMemory

```
procedure InitMemory (extraMasters: integer;
    aRainyDayFund, aCriticalBalance,
    aToolboxBalance: Size);
void InitMemory (short extraMasters,
    Size aRainyDayFund, Size aCriticalBalance,
    Size aToolboxBalance);
```

This method initializes the Macintosh memory manager and sets up the `GrowZone` function used in low memory situations. `ExtraMasters` is the number of times to call the Toolbox routine `MoreMasters`. `ARainyDayFund` is the number of bytes to set aside in the application heap to deal with low memory situations. `ACriticalBalance` is the number of bytes to use from the `rainyDayFund` for critical operations. `AToolboxBalance` is the

## 11 Application

number of bytes to use from the `rainyDayFund` for Toolbox operations that can't fail.

`ARainyDayFund` must be greater than or equal to `aCriticalBalance`, and `aCriticalBalance` must be greater than `aToolboxBalance`.

The values that you use as arguments to this method depend on the memory requirements of your application. You can use these values to get started:

Argument	Suggested value
<code>extraMasters</code>	10
<code>aRainyDayFund</code>	45000
<code>aCriticalBalance</code>	40000
<code>aToolboxBalance</code>	20000

### MakeSwitchboard

```
procedure MakeSwitchboard;  
void MakeSwitchboard (void);
```

This method creates the application's switchboard and stores it in the instance variable `itsSwitchboard`. If you create your own subclass of `CBartender`, you should override this method in your application subclass.

### MakeError

```
procedure MakeError;  
void MakeError (void);
```

This method creates the error-handler and stores it in the global variable `gError`. If you create your own error-handler, you can create a subclass of `CError` and override this method to use your error-handler.

### MakeDesktop

```
procedure MakeDesktop;  
void MakeDesktop (void);
```

*CFWDesktop, which implements a desktop with floating windows, is described on page 289.*

This method creates the desktop and stores it in the global variable `gDesktop`. The default method creates the standard desktop `CDesktop`. If your application uses a non-standard desktop, such as `CFWDesktop`, you should override this method in your application subclass.

### MakeClipboard

```
procedure MakeClipboard;  
void MakeClipboard (void);
```

This method creates the clipboard and stores it in the global variable `gClipboard`. If you create your own subclass of `CClipboard`, you should override this method to create a clipboard of your class.



**MakeDecorator**

```
procedure MakeDecorator;
void MakeDecorator (void);
```

This method creates the window decorator and stores it in the global variable `gDecorator`. The window decorator is responsible for the default sizes of windows and arranges them neatly on the desktop. If you want to implement a different decorator, you can create a subclass of `CDecorator` and override this method to use your decorator.

**SetUpFileParameters**

```
procedure SetUpFileParameters;
void SetUpFileParameters (void);
```

This method lets you set up the parameters that specify what kinds of files your application works on. The THINK Class Library passes some of these parameters to the Toolbox `SFPGetFile` function which displays the standard get file dialog. Your application subclass should override this method and call the inherited method to set up the default values.

Your own method should set the following instance variables and globals:

<code>sfNumTypes</code>	The number of different types of files your application deals with. If your application can open any kind of file, use <code>-1</code> .
<code>sfFileTypes</code>	The file types of the files that your application deals with. One type for each element of this array.
<code>gSignature</code>	The four-character signature of your application. Although it's not an instance variable, this method is the place to set up this global variable.
<code>sfFileFilter</code>	Optional. A pointer to a function that filters the file names your application can deal with.
<code>sfGetDLOGHook</code>	Optional. A pointer to a dialog hook function for the standard get file dialog.
<code>sfGetDLOGid</code>	Optional. The resource ID of the standard get file dialog you're using. Do not change this variable if you want to use the default dialog.
<code>sfGetDLOGFilter</code>	Optional. A pointer to a dialog filter proc.

To learn about file filter functions, dialog hook functions, and dialog filters, see *Inside Macintosh I*, Chapter 20, "The Standard File Package."

## ◆ 11 CApplication

---

### SetUpMenus

```
procedure SetUpMenus;  
void SetUpMenus (void);
```

SetUpMenus calls MakeBartender and reads all the menu information from your application's MBAR 1 resource and builds the menus. It also builds the desk accessory menu.

If your application uses menus that need to be built on the fly, you should override this method. For instance, if your application uses a **Font** menu, you would build it in this method. Be sure to call `inherited SetUpMenus` to make sure all the regular menus get built. See the description of CBartender, especially section "Resource based menus" on page 168, for more information about building these kinds of menus.

### MakeBartender

```
procedure MakeBartender;  
void MakeBartender (void);
```

This method creates the bartender object, which handles all interactions with the menu bar and menu items, and stores it in the global variable `gBartender`. If you create your own subclass of CBartender, you should override this method in your application subclass.

### InspectSystem

```
procedure InspectSystem;  
void InspectSystem (void);
```

This method fills in the fields of the `gSystem` global variable which gives you information about the Macintosh and the System Software that your application is running under.

In THINK C, `tSystem` is declared like this:

```
typedef struct  
{  
    Boolean    hasWNE           : 1;  
    Boolean    hasColorQD      : 1;  
    Boolean    hasGestalt       : 1;  
    Boolean    hasAppleEvents   : 1;  
    Boolean    hasAliasMgr      : 1;  
    Boolean    hasEditionMgr    : 1;  
    Boolean    hasHelpMgr       : 1;  
    Boolean    hasScriptMgr     : 1;  
    Boolean    hasFPU           : 1;  
    short      scriptsInstalled;  
    short      systemVersion;  
} tSystem;
```



In THINK Pascal it's defined like this:

```

type
  tSystem = packed record of
    hasWNE,
    hasColorQD,
    hasGestalt,
    hasAppleEvents,
    hasAliasMgr,
    hasEditionMgr,
    hasHelpMgr,
    hasScriptMgr,
    hasFPU           : Boolean;
    scriptsInstalled,
    systemVersion    : integer;
  end;

```

The field `scriptsInstalled` tells you how many scripts are in use. The field `systemVersion` gives you the version of the Macintosh System that's running. The version number is given as two byte-long numbers. For example, if `systemVersion` is `0x0607`, the System version number is 6.0.7.

### Advanced Initialization methods

You should not need to override these methods. If you are an advanced Macintosh programmer, you may want to take advantage of the fact that they're called in `IApplication`.

#### InstallPatches

```

procedure InstallPatches;
void InstallPatches (void);

```

Install patches to Toolbox routines. The default method patches `LoadSeg` to make sure that CODE resources can be loaded. If a code resource can't be loaded, the patch raises an exception. The default method also patches `ExitToShell` to call `RemovePatches`.

#### InstallPatches

```

procedure RemovePatches;
void RemovePatches (void);

```

Remove the Toolbox patches made in `InstallPatches`.

#### ForceClassReferences

```

procedure ForceClassReferences;
void ForceClassReferences (void);

```

If there are classes in your application that are instantiated by name, add a dummy reference to them here to prevent the smart linker from stripping them out.

## ◆ 11 *CApplication*

---

### Accessing methods

#### GetPhase

```
function GetPhase: integer;  
short GetPhase (void);
```

Return current application phase. The possible values are `appInitializing`, `appRunning`, and `appQuitting`.

### Command methods

The command methods handle events that the application takes care of. Since messages frequently get passed up the chain of command— from the pane, to the document, to the application—the application is the last chance to handle command messages. Most of the command methods don't do anything. They're null methods that keep the message from being passed on.

#### Notify

```
procedure Notify (theTask: CTask);  
void Notify (CTask *theTask);
```

When a subordinate object finishes a task, it sends the task to its supervisor. Documents usually handle `Notify` messages. This method is here in case a document tries to pass the `Notify` message on to the application. It places `theTask` into the instance variable `unhandledTask`. The task is disposed of at the end of the current event.

#### DoKeyDown

```
procedure DoKeyDown (theChar: char; keyCode: Byte;  
    macEvent: EventRecord);  
void DoKeyDown (char theChar, Byte keyCode,  
    EventRecord *macEvent);
```

You should handle key-down events within a pane or a document. For applications, this method doesn't do anything. It's here in case the `DoKeyDown` message gets passed up the chain of command. Your application subclass should not override this method.

#### DoAutoKey

```
procedure DoAutoKey (theChar: char; keyCode: Byte;  
    macEvent: EventRecord);  
void DoAutoKey (char theChar, Byte keyCode,  
    EventRecord *macEvent);
```

You should handle auto-key events within a pane or a document. For applications, this method doesn't do anything. It's here in case the `DoAutoKey` message gets passed up the chain of command. Your application subclass should not override this method.



## DoKeyUp

```
procedure DoKeyUp (theChar: char; keyCode: Byte;
  macEvent: EventRecord);
void DoKeyUp (char theChar, Byte keyCode,
  EventRecord *macEvent);
```

You should handle key-up events within a pane or a document. For applications, this method doesn't do anything. It's here in case the DoKeyUp message gets passed up the chain of command. Your application subclass should not override this method.

### Note

The Macintosh system event mask is usually set to mask out key-up events. If you need to get key-up events, be sure to reset the system event mask with the Toolbox routine SetEventMask.

## DoCommand

```
procedure DoCommand (theCommand: longint);
void DoCommand (long theCommand);
```

This is the only application command method that really does anything. DoCommand handles application commands that the user chooses from the menu. These are the commands that the default DoCommand method handles:

Command	Action
cmdNew	Sends a CreateDocument message to the application.
cmdOpen	Sends a ChooseFile message to your application. If the reply is good, sends an OpenDocument message to your application. The default ChooseFile message calls SFPGetFile with the values you specified in the SetUpFileParameters method. Your application must override the OpenDocument method.
cmdClose	If the front window is a desk accessory, close it. Your document class should handle this command to close documents.
cmdQuit	Sends a Quit message to your application. The default Quit method sends a

## 11 CApplication

	Quit message to the supervisor (a director) of each open window
cmdUndo	
cmdCut	
cmdCopy	
cmdPaste	
cmdClear	If the front window is a desk accessory, always call <code>SystemEdit</code> . Your document class should handle these commands to edit documents.
cmdToggleClip	Sends a <code>Toggle</code> message to the clipboard.

If your application defines its own application-related events, your application class should override this method. If your `DoCommand` method gets a command that it does not handle, it should call inherited `DoCommand`.

### UpdateMenus

```
procedure UpdateMenus;  
void UpdateMenus (void);
```

This method enables the appropriate menu items right before a menu selection. The default method enables the **Quit** command. If the application is in the foreground, it enables the **Show/Hide Clipboard** command. If the application is in the background, this method enables the `cmdClose`, `cmdUndo`, `cmdCut`, `cmdCopy`, `cmdPaste`, and `cmdClear` commands for desk accessories.

Your document's `UpdateMenus` method should enable the appropriate **Edit** menu commands for the document.

### PackageAppleEvent

```
function PackageAppleEvent (  
    theEvent, theReply: AppleEvent; long theRefCon;  
    eventClass, eventID: DescType): CAppleEvent;  
  
CAppleEvent *PackageAppleEvent (  
    struct AppleEvent *theEvent,  
    struct AppleEvent *theReply, long theRefCon,  
    DescType eventClass, DescType eventID);
```

Package an incoming `AppleEvent` and its default reply into a `CAppleEvent` object. This method returns a `CAppleEvent` object of class. If you create a subclass of `CAppleEvent`, you need to override this class. `CAppleEvent` is described on page 131.

**DoAppleEvent**

```
procedure DoAppleEvent (anAppleEvent: CAppleEvent);  
void DoAppleEvent (CAppleEvent *anAppleEvent);
```

Respond to an AppleEvent. This method handles the four required AppleEvents: `kAEOpenApplication`, `kAEOpenDocuments`, `kAEPrintDocuments`, and `kAEQuitApplication`. If you create a subclass of `CAppleEvent`, you need to override this class. `CAppleEvent` is described on page 131.

**Memory management methods**

These methods deal with critical memory situations in your application. The `InitMemory` method sets the function `GrowZoneFunc` as the function to call in low-memory situations. This function sends a `GrowMemory` message to your application to try to reclaim enough memory to continue.

**Note**

`GrowZoneFunc` is defined in `TCL.p` in THINK Pascal and in `CError.h` in THINK C.

**RequestMemory**

```
procedure RequestMemory (aCanFail: Boolean);  
void RequestMemory (Boolean aCanFail);
```

Change the `canFail` instance variable which controls how `GrowMemory` releases memory from the rainy day fund. If `aCanFail` is `FALSE`, the `GrowMemory` method is more conservative about releasing memory for a failing memory request. If `aCanFail` is `TRUE`, `GrowMemory` will exhaust the rainy day fund to satisfy the request.

Instead of using this method, you should use the utility routine `SetAllocation` (described on page 466). `SetAllocation` sends a `RequestMemory` message to the application and returns the previous value, so you can set it back.

Here's an example, in THINK Pascal:

## 11 CApplication

```
procedure AClass.AMethod;
var
    oldAlloc: Boolean;
    myHandle: Handle;
begin
    oldAlloc := SetAllocation(TRUE);
    myHandle := NewHandle(50000);
    SetAllocation(oldAlloc);

    if myHandle := nil then
    begin
        couldn't get 50000 bytes
    end
    else
    begin
        Go on with this operation
    end
end;
```

And here's the same example in THINK C:

```
void AClass::AMethod(void);
{
    Boolean oldAlloc;

    oldAlloc = SetAllocation(TRUE);
    myHandle = NewHandle(50000L);
    SetAllocation(oldAlloc);

    if (myHandle == NULL) {
        couldn't get 50000 bytes
    }
    else {
        Go on with this operation
    }
}
```

Remember that if the argument you pass to `SetAllocation` or `RequestMemory` is `FALSE`, `GrowMemory` will do whatever it can to get the memory. If there still isn't enough memory after the rainy day fund has been exhausted, your program will probably crash.

### SetCriticalOperation

```
procedure SetCriticalOperation (isCritical: Boolean);
void SetCriticalOperation (Boolean isCritical);
```

Set the `inCriticalOperation` instance variable which controls how `GrowMemory` releases memory from the rainy day fund. If `inCriticalOperation` is `TRUE`, and there is a memory shortage that causes `GrowMemory` to be called, `grow memory` tries to release as much memory in the rainy day fund that would leave `toolboxBalance` bytes free.



Your application should call the utility routine `SetCriticalOperation` (described on page 467) instead of sending a `SetCriticalOperation` message to `gApplication`.

Use this routine when there is a operation that must complete and for which your application might need to use some of the memory in the rainy day fund. The critical operation should release any memory that it allocated.

For example, when your application is saving a file, you may need to allocate an extraordinary amount of memory. In this case, you would call `SetCriticalOperation` to let `GrowMemory` know that, if necessary, you're willing to use more of the rainy day fund. Once the operation is complete, you would dispose of the memory.

Note that the `Run` method resets `inCriticalOperation` to `FALSE` every time through the event loop.

## GrowMemory

```
function GrowMemory(bytesNeeded: Size): longint;  
long GrowMemory (Size bytesNeeded);
```

The `GrowZoneFunc` function that the Memory Manager calls in low memory situations invokes this method to try to reclaim memory. `GrowMemory` sends a `MemoryShortage` message to the application.

If `MemoryShortage` wasn't able to release enough memory, `GrowMemory` starts to use the rainy day fund. If `inCriticalOperation` is `TRUE`, it uses as much of the rainy day fund that would still leave `toolboxBalance` bytes in the fund. If `inCriticalOperation` is `FALSE`, it uses as much that would leave `criticalBalance` bytes in the fund.

If `GrowMemory` still wasn't able to release enough memory from the rainy day fund, and `canFail` is `FALSE`, the entire rainy day fund is released. If the entire rainy day fund has been exhausted, this method calls `OutOfMemory`.

## MemoryShortage

```
procedure MemoryShortage (bytesNeeded: Size);  
void MemoryShortage (Size bytesNeeded);
```

The `GrowMemory` method sends a `MemoryShortage` message to your application to try to free memory. The default `MemoryShortage` method does nothing, so your application subclass should override this method. Your `MemoryShortage` method should try to free `bytesNeeded` bytes of memory, and it should disable menu commands that won't work in low memory situations.

## ◆ 11 Application

---

---

### Warning

Your `MemoryShortage` method must not allocate any memory.

---

### MemoryReplenished

```
procedure MemoryReplenished;  
void MemoryReplenished (void);
```

Your application will get this message when the memory situation is no longer critical. The default `MemoryReplenished` method does nothing, so your application subclass should override this method. Your `MemoryReplenished` method should enable the commands you disabled in the `MemoryShortage` method. You might also want to reallocate memory that you released in that method.

### OutOfMemory

```
function OutOfMemory (bytesNeeded: Size): longint;  
long OutOfMemory (Size bytesNeeded);
```

A memory request cannot be satisfied, and `canFail` was `FALSE`, so the application won't handle the memory allocation failure. This method calls `Failure(memFullError, 0)`. If there is any memory left, the top-level handler will display an alert, but it is more likely that the application will crash.

You can override this method if you can free up more memory, or if you can exit the application gracefully without allocating memory. Normally, you would try to free enough memory in your `MemoryShortage` method. If there are more drastic ways to release memory, implement them here. This method should return 1 if it successfully released memory or zero if it couldn't.

---

### Warning

This method should not allocate any memory or call any routines that allocate memory.

---

### Execution methods

The execution methods are invoked while your application is running. Most of these methods handle system-related events. The only execution method your application should override is the `Exit` method.

### Run

```
procedure Run;  
void Run (void);
```

This method runs your application until the user quits. This method sends the application `ProcessEvent` messages until the user chooses to quit.

Before running your program, this method sends a `Preload` message to your application to open or print documents that the user selected and opened from the Finder.

Your application should not override this method.

### **ProcessEvent**

```
procedure Process1Event;
void Process1Event (void);
```

Process and dispatch one event. This method sends a `ProcessEvent` message to the switchboard (stored in the instance variable `itsSwitchboard`). If you installed an urgent chore, this method sends it a `Perform` message.

### **Preload**

```
procedure Preload;
void Preload (void);
```

If the user opened or chose to print files from the Finder, this method sends the application an `OpenDocument` message for each document. If the user chose the **Print** command, this method sends a `DoCommand(cmdPrint)` message to the gopher. After processing all the files, this method sends the application a `StartupAction` message.

### **StartupAction**

```
procedure StartupAction (numPreloads: integer);
void StartupAction (short numPreloads);
```

This method gives you an opportunity to perform any startup actions. `NumPreloads` is the number of files that the user selected from the Finder. If `numPreloads` is zero, the default method sends a `DoCommand(cmdNew)` message to the gopher (usually the application). The effect of the default method is to open an untitled document at startup.

### **Suspend**

```
procedure Suspend;
void Suspend (void);
```

Your application is about to be suspended under `MultiFinder`. The default method calls the inherited method and sets the global variable `gInBackground` to `TRUE`.

### **Resume**

```
procedure Resume;
void Resume (void);
```

Your application has come back to the foreground under `MultiFinder`. The default method calls the inherited method and sets the global variable `gInBackground` to `FALSE`.

## ◆ 11 CApplication

---

### SwitchToDA

```
procedure SwitchToDA;  
void SwitchToDA (void);
```

A desk accessory is becoming active. The default method sends the application a Suspend message. Your application should not override or use this method.

### SwitchFromDA

```
procedure SwitchFromDA;  
void SwitchFromDA (void);
```

Your application is becoming active after a desk accessory was active. The default method sends your application a Resume message. Your application should not override or use this method.

### Idle

```
procedure Idle (macEvent: EventRecord);  
void Idle (EventRecord *macEvent);
```

This method handles periodic tasks. It also checks to see if a critical memory situation is no longer critical. Idle sends Dawdle messages to the gopher and to each of the gopher's supervisors. It also sends Perform messages to all chores. Your application should not override this method.

### Quit

```
procedure Quit;  
void Quit (void);
```

The user wants to quit the application. This method sends a Quit message to the supervisor (a director) of each open window. Any of the directors can cancel quitting. Your application should not override this method.

---

#### Note

The Quit method for directors returns a Boolean value. If it returns FALSE, quitting is canceled.

---

### Exit/ExitApp

```
procedure ExitApp;  
void Exit (void);
```

The application is about to exit. Your main program should send an Exit message to the application before it terminates; the THINK Class Library will not send the message for you. The default method does nothing.

The Exit method is the last chance you have to clean up after your application. For example, this is a good place to delete any temporary files you've created.

**JumpToEventLoop**

```
procedure JumpToEventLoop;  
void JumpToEventLoop (void);
```

This method is for compatibility with earlier versions of the THINK Class Library. It calls `Failure(kSilentErr, 0)` to force a return to the top-level exception handler in `Run`.

**Document methods**

The document methods of an application deal with creating and opening documents. In your application subclass, the only document methods you'll need to override are `CreateDocument` and `OpenDocument`. All the other document methods are used internally to implement standard behavior.

**CreateDocument**

```
procedure CreateDocument;  
void CreateDocument (void);
```

This method creates a new document. The default `DoCommand` method sends a `CreateDocument` message to the application when the user chooses **New** from the **File** menu. Your application must override this method. Your `CreateDocument` method needs to create a new document, initialize it, and then send it a `NewFile` message.

**OpenDocument**

```
procedure OpenDocument (macSFReply: SFReply);  
void OpenDocument (SFReply *macSFReply);
```

This method opens an existing document. The `macSFReply` record specifies which document to open. The default `DoCommand` method sends an `OpenDocument` message to the application when the user chooses **Open...** from the **File** menu.

---

**Note**

`DoCommand` actually sends a `ChooseFile` message to the application first, and if the reply is good, it sends an `OpenDocument` message. Your `OpenDocument` method can assume that the `macSFReply` record is valid.

---

Your application subclass must override this method. Your `OpenDocument` method needs to create a new document, initialize it, and then send it an `OpenFile(macSFReply)` message.

**ChooseFile**

```
procedure ChooseFile (var macSFReply: SFReply);  
void ChooseFile (SFReply *macSFReply);
```

This method displays a standard get file dialog and places the reply in the `macSFReply` record. This method calls the Toolbox routine `SFPGetFile`

## 11 CApplication

---

with the file parameters you specified in the `SetUpFileParameters` method.

You can send the application a `ChooseFile` message from other methods to get the name and location of a file. Don't forget to check the `macSFReply`.good field to make sure that the information in the record is valid.

Your application subclass should not override this method unless it uses a different technique to open a document.

### Periodic task methods

#### AssignIdleChore

```
procedure AssignIdleChore (theChore: CChore);  
void AssignIdleChore (CChore *theChore);
```

This method adds `theChore` to the application's list of chores. The application sends a `Perform` message to each chore at idle time. Your application should not override this method.

---

#### Note

If you want to remove the chore from the list later on, you must keep a pointer to the chore object.

---

#### CancelIdleChore

```
procedure CancelIdleChore (theChore: CChore);  
void CancelIdleChore (CChore *theChore);
```

This method removes `theChore` from the application's list of chores. Your application should not override this method.

#### AssignUrgentChore

```
procedure AssignUrgentChore (theChore CChore);  
void AssignUrgentChore (CChore *theChore);
```

This method adds `theChore` to the application's list of urgent chores. The application sends a `Perform` message to each urgent chore after handling the current event and then removes the chore. Your application should not override this method.

### Class resources

#### Resource

MBAR 1  
STR# 129

#### Description

The application's menu bar  
Low memory warning strings.

# CArray ♦

## 12

### Introduction

CArray implements a resizable array.

### Heritage

Superclass  
Subclasses

CCollection  
CCluster  
CRunArray

### Using CArray

Use CArray when you want an array you can resize. The elements may be any size, as long as they're all the same size.

To insert a new element into the array, use `InsertAtIndex`. It moves the element in that slot and all the element below it down one slot. It's illustrated in Figure 12-1.

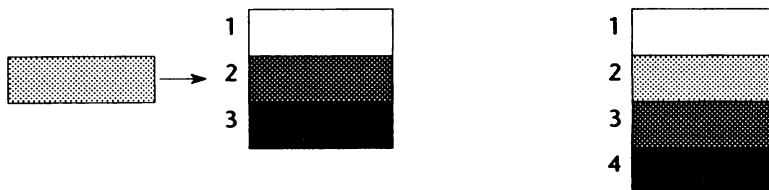
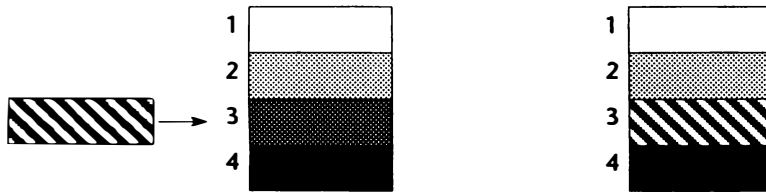


Figure 12-1 Before and after `InsertAtIndex`

## 12 CArray

To update the value of an existing element, use `SetItem`. It at that slot with the new element. It's illustrated in Figure 12-2.



**Figure 12-2** Before and after `SetItem`

To delete an element, use `DeleteItem`. It deletes the element and moves all the elements below it up one slot.

`CArray` stores the elements in its array in one large piece of memory that it allocates dynamically. It uses the instance variable `blockSize` to control how to grow and shrink this memory. If there are no empty slots when you try to insert an element, `CArray` grows memory by `blockSize` slots. When you delete an element and there are `blockSize` empty slots left, `CArray` shrinks the memory by `blockSize` slots. By default, `blockSize` is 3.

`CArray` has two methods that move elements in the array: `Swap` and `MoveItemToIndex`. To store an element while moving it, `CArray` has a temporary storage slot at the end of the array. To make sure an element in temporary storage isn't accidentally overwritten, those methods set the flag `usingTemporary` whenever they put something in the temporary storage slot. This slot is not counted in the `slots` instance variable, which is the total number of slots that can permanently store elements.





## Variables

Variable	Type	Description
blockSize	integer	Number of slots to allocate when more space is needed.
slots	longint	Total number of slots allocated.
hItems	Handle	Items in the array
elementSize	longint	Size of each element in bytes
lockChanges	Boolean	If TRUE, you can't insert or delete in array.
usingTemporary	Boolean	If TRUE, temporary element storage buffer is in use.

## Methods

### Creation and destruction methods

#### IArray

```
procedure IArray (elementSize: longint);
void IArray (long elementSize);
```

Initialize the array. ElementSize is the size of each element in the array. This method sets the size of each element in the array to ElementSize, sets blockSize to 3, and allocates enough space for the temporary storage slot.

#### Free/Dispose

```
procedure Free;
void Dispose (void);
```

Dispose of the array and the block of memory it allocated.

### Accessing methods

#### SetBlockSize

```
procedure SetBlockSize (aBlockSize: integer);
void SetBlockSize (short aBlockSize);
```

Set the number of elements to add to the array when CArray allocates more space. IArray initializes this value to 3.

#### SetLockChanges

```
function SetLockChanges (fLockChanges: Boolean):
    Boolean;
```

```
Boolean SetLockChanges (Boolean fLockChanges);
```

If fLockChanges is TRUE, you can't use the InsertAtIndex and DeleteItem methods. This method sets lockChanges to

## ◆ 12 CArray

---

fLockChanges and returns the previous value of lockChanges. CCluster uses this method to prevent you from corrupting the array when you iterate over its elements with DoForEach.

### Insertion and deletion methods

#### InsertAtIndex

```
procedure InsertAtIndex (itemPtr: Ptr;  
    index: longint);
```

```
void InsertAtIndex (void *itemPtr, long index);
```

Insert the element at itemPtr at the index index. If index is already occupied, move the element in that slot and all the element below it down one slot. If index is beyond the last position, add the item to the end of the array. If necessary, this method allocates a larger block of memory for the array. This method sends a BroadcastChange message with arrayInsertElement as the reason.

#### DeleteItem

```
procedure DeleteItem (index: longint);
```

```
void DeleteItem (long index);
```

Delete the element at index index. This method moves all the elements below index up one slot. If the number of free slots is greater than blockSize, this method reallocates a block of memory that's smaller by blockSize slots.

#### SetItem

```
procedure SetItem (itemPtr: Ptr; index: longint);
```

```
void SetItem (void *itemPtr, long index);
```

Put the element at itemPtr into the array at position index. Index must be within the array. This method sends a BroadcastChange message with arrayElementChanged as the reason.

#### Store

```
procedure Store (itemPtr: Ptr; index: longint);
```

```
void Store (void *itemPtr, long index);
```

Put the element at itemPtr into the array at position index. Whenever possible, use Set instead. This is an internal method that performs no error-checking. In THINK C, this is a protected method.

### Membership method

#### GetItem

```
procedure GetItem (itemPtr: Ptr; index: longint);
```

```
void GetItem (void *itemPtr, long index);
```

Return the element at position index in the block memory at itemPtr. Index must be within the array. ItemPtr must point to a block large enough to hold one element.

**Retrieve**

```
procedure Retrieve (itemPtr: Ptr; index: longint);
void Retrieve (void *itemPtr, long index);
```

Return the element at position `index` in the block memory at `itemPtr`. Whenever possible, use `Get` instead. This is an internal method that performs no error-checking. In THINK C, this is a protected method.

**Search**

```
function Search (itemPtr: Ptr; function compare (
    item1, item2: Ptr): integer): Ptr;
```

```
void *Search (void *itemPtr, CompareFunc compare);
```

Find an element that satisfies a condition. The function `compare` tests for the condition. It takes two arguments. The first is `itemPtr` and the second is a pointer to an element in the array. `Compare` should return 0 when an element satisfies the condition. `ItemPtr` is a pointer to some data that you can use in the comparison. It can be a pointer to data like that in the array, a pointer to another type of data, or NIL

In THINK C, declare `compare` like this:

```
int compare (void *item1, void *item2);
```

In THINK Pascal, declare `compare` like this:

```
function compare (item1, item2: Ptr): integer;
```

**Moving methods****MoveItemToIndex**

```
procedure MoveItemToIndex (currentIndex, newIndex:
    longint);
```

```
void MoveItemToIndex (long currentIndex,
    long newIndex);
```

Move the element at `currentIndex` to `newIndex`. This method moves the elements between the old and new positions up or down slot. This method sends a `BroadcastChange` message with `arrayMoveElement` as the reason.

**Swap**

```
procedure Swap (index1, index2: longint);
```

```
void Swap (long index1, long index2);
```

Swap the two items: move the item at `index1` to `index2`, and move the item at `index2` to `index1`. For both items, this method sends a `BroadcastChange` message with `arrayElementChanged` as the reason.

## ◆ 12 CArray

---

### Resizing methods

CArray uses these internal methods to resize its block of memory. You should need to use them only if you are creating a subclass of CArray. In THINK C, only subclasses of CArray can use them.

#### Resize

```
procedure Resize (numSlots: longint);  
void Resize (long numSlots);
```

Resize the array so that it has numSlots slots. This method does not change the number of items stored in the array.

#### MoreSlots

```
procedure MoreSlots;  
void MoreSlots (void);  
Add blockSize slots to the array.
```

### Temporary storage methods

CArray uses these methods to move elements into and out of the temporary storage slot. You should need to use them only if you are creating a subclass of CArray. In THINK C, only subclasses of CArray can use them.

#### CopyToTemporary

```
procedure CopyToTemporary (index: longint);  
void CopyToTemporary (long index);
```

Place the element in slot index into temporary storage. This method also sets usingTemporary to TRUE.

#### CopyFromTemporary

```
procedure CopyFromTemporary (index: longint);  
void CopyFromTemporary (long index);
```

Place an item from temporary storage into slot index. This method also sets usingTemporary to FALSE.

### Offset method

CArray uses this method to find the offset into the array of an element. You should need to use it only if you are creating a subclass of CArray. In THINK C, only subclasses of CArray can use it.

#### ItemOffset

```
function ItemOffset (itemIndex: longint): longint;  
long ItemOffset (long itemIndex);
```

Return the offset into the array of the slot itemIndex in bytes.

# *CBartender* ♦

## 13

---

### Introduction

The bartender is the object that manages the menu bar, menus, and menu items. The bartender is an object of class `CBartender`. There is only one object of this class stored in a global variable.

---

#### Note

Earlier versions of the THINK Class Library used `CBarOwner`, a subclass of `CBartender`, to prevent excessive menu bar flashing. This version of `CBartender` includes this improvement.

---

### Heritage

Superclass  
Subclasses

`CObject`  
This class has no subclasses.

### Using `CBartender`

In the THINK Class Library, almost every menu item has a command number associated with it. The bartender maintains a table that maps menu items with command numbers. You should never try to access the bartender's data structures directly. Instead, use the access methods.

You should not need to subclass the `CBartender` class. If, for some reason, you do need to create a subclass, you'll also need to override the `SetUpMenus` method in your application class to initialize your bartender.

To manipulate the menus or menu items, send messages to the global bartender object stored in the global variable `gBartender`.

#### Creating standard menus

To associate a command number with a menu item, append the command number to the end of the menu item in your `MENU` resource. The menu item

## 13 CBartender

text and the command number are separated by the character #. Figure 13-1 shows what the **File** menu looks like in ResEdit:

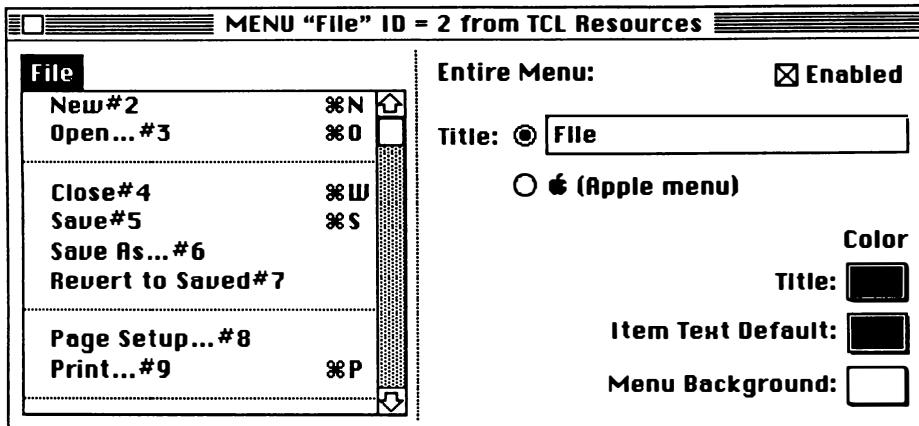


Figure 13-1 The File menu in ResEdit

If you don't append a command number to a menu item, the bartender automatically associates the command number `cmdNull` with it.

The bartender builds its tables from the information in the MBAR resource you pass to its initialization method. Your application's MBAR resource should contain the menu IDs of all the menus that will appear in the menu bar.

### Note

By default, the application's `SetUpMenus` method uses the MBAR resource with an ID = 1.

*The MENU resources for these menus are in the file TCL Resources.*

You can use any menu ID for your application's menus. The THINK Class Library reserves the following menu IDs for certain menus:

Menu title	Menu ID	Mnemonic
Apple	1	MENUapple
File	2	MENUfile
Edit	3	MENUedit
Font	10	MENUfont
Size	11	MENUsize



These are the commands that the THINK Class Library defines. In THINK Pascal they're in the `TCL.p`, and in THINK C they're in the file `Commands.h`. Remember that the THINK Class Library reserves the commands in the range 1 to 1023.

These are standard **File** menu commands:

<b>Command</b>	<b>ID</b>
<code>cmdQuit</code>	1
<code>cmdNew</code>	2
<code>cmdOpen</code>	3
<code>cmdClose</code>	4
<code>cmdSave</code>	5
<code>cmdSaveAs</code>	6
<code>cmdRevert</code>	7
<code>cmdPageSetup</code>	8
<code>cmdPrint</code>	9

These are the standard **Edit** menu commands:

<b>Command</b>	<b>ID</b>
<code>cmdUndo</code>	16
<code>cmdCut</code>	18
<code>cmdCopy</code>	19
<code>cmdPaste</code>	20
<code>cmdClear</code>	21
<code>cmdToggleClip</code>	22
<code>cmdSelectAll</code>	23

These are the text style commands:

<b>Command</b>	<b>ID</b>
<code>cmdPlain</code>	30
<code>cmdBold</code>	31
<code>cmdItalic</code>	32
<code>cmdUnderline</code>	33
<code>cmdOutline</code>	34
<code>cmdShadow</code>	35
<code>cmdCondense</code>	36
<code>cmdExtend</code>	37

## 13 CBartender

---

These are the text alignment commands:

Command	ID
cmdAlignRight	40
cmdAlignLeft	41
cmdAlignCenter	42
cmdJustify	43

These are the line spacing commands:

Command	ID
cmdSingleSpace	50
cmdHalfSpace	51
cmdDoubleSpace	52

These are miscellaneous commands:

Command	ID	Description
cmdNull	0	Command which does nothing
cmdOK	100	OK button in dialog box
cmdCancel	101	Cancel button in dialog box
cmdAbout	256	About Application request

If you want the bartender to return the menu ID and item number of a particular menu item, use the special command -1 in your MENU resource. The bartender will return the negative of the menu ID in the high word and the menu item number in the low word. This is the same as menus that you build on the fly, described next.

### Resource based menus

Your application may use menus that don't have menu commands associated with each item. The most common example is a **Font** menu. Use the reserved **Font** menu in your MBAR resource, and add the resources with the AddResMenu Toolbox routine. Here's how you might write your SetUpMenus method in THINK Pascal to include a font menu.

```
procedure CYourApp.SetUpMenus;
begin
  inherited SetUpMenus;
  AddResMenu(GetMHandle(MENUfont), 'FONT');
  SetDimOption(MENUfont, dimNONE);
  SetUnchecking(MENUfont, true);
end;
```



And here's how you might write your the method in THINK C:

```
void CYourApp::SetUpMenus()
{
    MenuHandle macMenu;

    inherited::SetUpMenus();
    AddResMenu(GetMHandle(MENUfont), 'FONT');
    SetDimOption(MENUfont, dimNONE);
    SetUnchecking(MENUfont, TRUE);
}
```

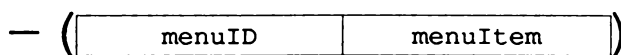
#### Note

To learn about SetDimOption and SetUnchecking, see section "Dimming and checking menu items" on page 171.

When you build a menu on the fly like this, the bartender does not associate a command number with the menu items. In this case, the bartender's FindCmdNumber returns the negative of the menu ID in the high word and the item ID in the low word.

*The values that FindCmd returns in these cases are the negative of what MenuSelect would have returned*

For example, if you choose the ninth font in the **Font** menu, FindCmdNumber returns -655369 (0xFFFF5FFF7). Since the number is negative, you know that there is no command number associated with the item. To get the menu ID and the menu item, negate the value and split it into two words. In this example, the return value becomes 655369 (0x000A0009), which means that the menu ID is 10 and the menu item is 9.



**Figure 13-2** How FindCmdNumber builds command numbers.

#### Note

There are functions to extract the high word and low word of a long integer. In THINK Pascal, the functions are HiWord and LoWord. In THINK C, the functions are HiShort and LoShort (defined in Global.h).

You can add menu items to existing menus if you like. You might want to add the **Font** menu to a general text-handling menu, or you might want to have a menu with the names of all the documents your application has

## ◆ 13 CBartender

---

*For a detailed description of hierarchical menus, see Inside Macintosh V, Chapter 13, "The Menu Manager."*

opened. The important thing to remember is to add all these menu items at the **end** of the existing menu. Otherwise, the bartender will get confused.

### Creating hierarchical menus

Most of the time you'll be able to set up your hierarchical menus in your resource file. The key thing to remember is that the item that the hierarchical menu is attached to uses the command key equivalent and the item mark differently. The command key value must be 0x1B (Control-[]), and the value of the item mark is the resource ID of the hierarchical menu.

Sometimes, you can't specify a hierarchical menu in a resource. In these cases, use the `InsertHierMenu` method. This method adds the hierarchical menu to an existing menu in the menu bar and to the bartender's table. Generally, you'll specify hierarchical menus in your resource file. Use this method to insert hierarchical menus from your program.

---

#### Note

For hierarchical menus to work correctly, you must set them up either in the resource file or with a call to `InsertHierMenu`. If you use only the Macintosh Toolbox routines, the bartender's item checking methods will not work.

---

This example shows how to write a method in THINK Pascal to insert a hierarchical **Font** menu as the third item in a **Text** menu from your program:

```
procedure CYourApp.SetUpMenus;
var
  macMenu: MenuHandle;
  MENUtext, itemNo: integer;
begin
  inherited SetUpMenus;
  gBartender.InsertHierMenu(MENUfont, cmdNull,
    MENUtext, 2);
  AddResMenu(GetMHandle(MENUfont), 'FONT');
  gBartender.SetDimOption(MENUfont, dimNONE);
  gBartender.SetUnchecking(MENUfont, true);
end;
```

And this example shows how to write a method in THINK C to do the same thing:

```
void CYourApp::SetUpMenus ()
{
    MenuHandle    macMenu;
    short         MENUtext, itemNO;

    inherited::SetUpMenus ();
    gBartender->InsertHierMenu (MENUfont, cmdNull,
        MENUtext, 2);
    AddResMenu (GetMHandle (MENUfont), 'FONT');
    SetDimOption (MENUfont, dimNONE);
    SetUnchecking (MENUfont, TRUE);
}
```

*The dimming, undimming, and unchecking take very little time. You won't notice a delay between the time you click on the menu bar and when the menu is displayed.*

### Dimming and checking menu items

The bartender includes methods to let you enable and disable and check and uncheck menu items. When you click in the menu bar, the bartender sends an UpdateMenus message to all of the bureaucrats in the Chain of Command. In the general case, all the items in the menu start out dimmed and unchecked. Then each bureaucrat enables the menu items that pertain to it. Once the appropriate items have been enabled and checked, the Toolbox routine MenuSelect displays all the menus.

Suppose you click in the menu bar of a text processing application. When you click on the menu bar, but before the Toolbox displays the menu, the bartender disables all the menu items. Then, the application enables all the application related menu items: **New**, **Open...**, **Quit**, for example. The document enables all the document related items: **Save**, **Save As...**, **Revert** (if the document's been changed), and so on. A pane might check the current font and size in the **Font** menu. Finally the menu appears on the screen with the correct items checked and enabled.

The UpdateMenus method of your application, document, and pane need to enable each item. To make sure that item enabling happens from the general (application) to the specific (pane), be sure to call `inherited UpdateMenus` first in your own UpdateMenus method.

You can use the bureaucrat methods `SetDimOption` and `SetUnchecking` in your application `SetUpMenus` method to modify this behavior. `SetDimOption` lets you specify whether the bartender should dim all, some, or none of the items when you click on the menu bar. For **Font** menus, for instance, it doesn't make sense to dim all the font names only to re-enable them again.

### Variables

The global variable `gBartender` points to the single instance of the bartender. Whenever you want to manipulate menus, you'll send messages to this object. The bartender uses its instance variables to maintain the mapping between command numbers and menu items. You should not need to access or alter the instance variables.

#### Global variable

Variable	Type	Description
<code>gBartender</code>	<code>CBartender</code>	The global bartender.

#### Instance variables

Variable	Type	Description
<code>numMenus</code>	integer	The number of menus available.
<code>theMenus</code>	<code>MenuEntryH</code>	A table with an entry for each available menu.
<code>choreAssigned</code>	Boolean	TRUE if there's a pending chore to redraw the menu bar
<code>forceMBarUpdate</code>	Boolean	TRUE if <code>UpdateMenuBar</code> always draws the menu bar.

### Methods

To use these methods to manipulate menus and menu items, send messages to the bartender object stored in the global `gBartender`.

For example, to disable the **Open...** command you would write, in THINK Pascal:

```
gBartender.DisableCmd(cmdOpen);
```

And in THINK C you would write:

```
gBartender->DisableCmd(cmdOpen);
```

To change the name of the **Copy** command to **Copy Picture**, you might write, in THINK Pascal:

```
gBartender.SetCmdText(cmdCopy, 'Copy Picture');
```

And in THINK C you might write:

```
gBartender->SetCmdText (cmdCopy,  
    "\pCopy Picture");
```

In the rare case where you need the actual menu handle or other information about the menu, use the `FindMacMenu` or `FindMenuItem` methods.

### Construction methods

#### **IBartender**

```
procedure IBartender (MBARid: integer);  
void IBartender (short MBARid);
```

Initialize a Bartender object. The application method `SetUpMenus` sends this message to initialize the bartender and stores the bartender in the global variable `gBartender`. `MBARid` is the ID of the MBAR resource that the bartender uses to build its menu tables. The default `SetUpMenus` method uses 1 as its `MBARid`.

### Insertion and deletion methods

#### **AddMenu**

```
procedure AddMenu (MENUid: integer;  
    install: Boolean; beforeID: integer);  
void AddMenu (short MENUid, Boolean install,  
    short beforeID);
```

This method adds a menu to the bartender's list. `MENUid` is the resource ID of the menu. If `install` is `TRUE`, install it into the menu bar as well as into the bartender's list. `BeforeID` specifies before which menu this menu is installed. If `beforeID` is 0, the menu is added to the end of the menu bar. A `beforeID` of -1 is a hierarchical or pop-up menu.

#### **RemoveMenu**

```
procedure RemoveMenu (MENUid: integer);  
void RemoveMenu (short MENUid);
```

Remove the specified menu from the menu bar and from the bartender's list. `MENUid` is the resource ID of the menu.

#### **InsertInBar**

```
procedure InsertInBar (MENUid, beforeID: integer);  
void InsertInBar (short MENUid, short beforeID);
```

Insert a menu in the menu bar and in the bartender's list. This message is the same as `AddMenu (MENUid, TRUE, beforeID)`.

#### **DeleteFromBar**

```
procedure DeleteFromBar (MENUid: integer);  
void DeleteFromBar (short MENUid);
```

Remove the specified menu from the menu bar, but leave it in the list. After deleting the menu, this method creates a chore of class `CMBArChore` and

## ◆ 13 CBartender

---

sends it to `gApplication` as an urgent chore. The chore will redraw the menu bar the next time through the event loop.

### **InsertHierMenu**

```
procedure InsertHierMenu (hMENUId: integer;  
    cmdNo: longint; inMENUId, afterItem: integer);  
void InsertHierMenu (short hMENUId, long cmdNo,  
    short inMENUId, short afterItem);
```

Insert a hierarchical menu in another menu. `hMENUId` is the menu ID of the hierarchical menu. If you want to be able to dim a hierarchical menu, provide a command number in `cmdNo`; if you're not going to dim the hierarchical menu, pass in `cmdNull` for `cmdNo`. `inMENUId` is the menu ID of the menu that will contain the hierarchical menu. `AfterItem` is the item number after which the hierarchical menu should be inserted. The title of the menu is used as the item name for the hierarchical menu.

### **Item manipulation methods**

#### **EnableCmd**

```
procedure EnableCmd (cmdNo: longint);  
void EnableCmd (long cmdNo);  
Enable the menu item associated with cmdNo.
```

#### **DisableCmd**

```
procedure DisableCmd (cmdNo: longint);  
void DisableCmd (long cmdNo);  
Disable the menu item associated with cmdNo.
```

#### **EnableMenu**

```
procedure EnableMenu (MENUId: integer);  
void EnableMenu (short MENUId);  
Enable the menu with resource ID MENUId.
```

#### **DisableMenu**

```
procedure DisableMenu (MENUId: integer);  
void DisableMenu (short MENUId);  
Disable the menu with resource ID MENUId.
```

#### **EnableMenuBar**

```
procedure EnableMenuBar;  
void EnableMenuBar (void);  
Enable all the menus currently in the menu bar.
```

#### **DisableMenuBar**

```
procedure DisableMenuBar;  
void DisableMenuBar (void);  
Disable all the menus currently in the menu bar.
```



<b>SetCmdText</b>	<pre>procedure SetCmdText (cmdNo: longint;   theText: Str255);</pre> <pre>void SetCmdText (long cmdNo, Str255 theText);</pre> <p>Change the text of the menu item associated with cmdNo to theText.</p>
<b>GetCmdText</b>	<pre>procedure GetCmdText (cmdNo: longint;   var theText: Str255);</pre> <pre>void GetCmdText (long cmdNo, Str255 theText);</pre> <p>Get the text of the menu item associated with cmdNo. If cmdNo is not associated with a menu item, the contents of theText are unchanged.</p>
<b>CheckMarkCmd</b>	<pre>procedure CheckMarkCmd (cmdNo: longint;   checked: Boolean);</pre> <pre>void CheckMarkCmd (long cmdNo, Boolean checked);</pre> <p>Place (or remove) a check mark next to the menu item associated with cmdNo.</p>
<b>Item insertion and deletion</b>	
<b>InsertMenuCmd</b>	<pre>procedure InsertMenuCmd (cmdNo: longint;   theText: Str255; MENUid, afterItem: integer);</pre> <pre>void InsertMenuCmd (long cmdNo, Str255 theText,   short MENUid, short afterItem);</pre> <p>Insert a new item in a menu. CmdNo is the command number of the new item. TheText is the text of the new item. MENUid is the menu ID you're inserting the new item into. AfterItem specifies after which item to insert the new item.</p>
<b>RemoveMenuCmd</b>	<pre>procedure RemoveMenuCmd (cmdNo: longint);</pre> <pre>void RemoveMenuCmd (long cmdNo);</pre> <p>Remove the menu item associated with the command cmdNo from its menu.</p>
<b>Look-up methods</b>	
<p>These methods let you know which command number is associated with which menu item and which menu contains the menu item associated with a particular command number.</p>	
<b>FindCmdNumber</b>	<pre>function FindCmdNumber (MENUid, itemNo: integer):   longint;</pre> <pre>long FindCmdNumber (short MENUid, short itemNo);</pre> <p>Return the command number associated with the menu MENUid and itemNo. If the item has no command associated with it, this method returns -MENUid in the high word and itemNo in the low word.</p>

## 13 CBartender

---

This is the message that the desktop and switchboard send to the bartender to convert a menu selection or a menu key into a command number. Some menu items, like font menus, don't have commands associated with them. For these menus, `FindCmdNumber` returns the menu id and item number.

### **FindMenuItem**

```
procedure FindMenuItem (cmdNo: longint;  
    var MENUid: integer; var macMenu: MenuHandle;  
    var itemNo: integer);  
  
void FindMenuItem (long cmdNo, short *MENUid,  
    MenuHandle *macMenu, short *itemNo);
```

Given a command number, return the menu ID, the handle to the menu, and the item number. If the command number isn't associated with a menu item, all three items are set to zero.

### **FindItemText**

```
function FindItemText; (MENUid: integer;  
    itemStr: Str255): integer;  
  
short FindItemText (short MENUid, Str255 itemStr);
```

Return the item number in `MENUid` with the specified text. Return 0 if the menu doesn't have an item called `itemStr`. This method is useful for keeping track of items that don't have command numbers associated with them.

### **FindMacMenu**

```
function FindMacMenu (MENUid: integer): MenuHandle;  
  
MenuHandle FindMacMenu (short MENUid);
```

Return a handle to the Macintosh menu with ID `MENUid`. If the menu is not in the bartender's table, return `NIL`.

### **FindMenuIndex**

```
function FindMenuIndex (MENUid: integer): integer;  
  
short FindMenuIndex (short MENUid);
```

Return the index of the menu `MENUid` in the bartender's table. This is an internal method. You should not use this method.

### **Appearance methods**

### **SetDimOption**

```
procedure SetDimOption (MENUid: integer;  
    aDimming: DimOption);  
  
void SetDimOption (short MENUid, DimOption aDimming);
```

Set the dim option for the specified menu. By default, all the items in the menu will be dimmed when you click in the menu bar. The `UpdateMenus` method of each of your bureaucrats needs to enable the appropriate items.



**Dim Option**

dimNONE

dimSOME

dimALL

**Meaning**

Never dim any of the menu items.

Dim only the menu items that have command numbers associated with them.

Dim all of the menu items. Each bureaucrat's UpdateMenus method must enable the items for the commands it handles.

This is the default.

**SetUnchecking**

```
procedure SetUnchecking (MENUId: integer,
    anUnchecking: Boolean);
```

```
void SetUnchecking (short MENUId,
    Boolean anUnchecking);
```

Set the checking option for the specified menu. By default, none of the items are unchecked when you click in the menu bar. If you set this option to TRUE, all the items will be unchecked, and your UpdateMenus methods must check the appropriate menu items.

**If anUnchecking is... Do this...**

TRUE

Uncheck all the menu items at menu selection. Your UpdateMenus method should check the appropriate items. Set this option for menus like **Font** menus or **Style** menus.

FALSE

Don't uncheck any menu items at menu selection. This is the default since most menu items never need to be checked.

**UpdateAllMenus**

```
procedure UpdateAllMenus;
```

```
void UpdateAllMenus (void);
```

The bartender gets this message before menu selection. First it disables and unchecks all the items in the menus according to the dimming and unchecking options. Then it sends an UpdateMenus message to the gopher to enable the appropriate menu items. You should not use or override this method.

**UpdateMenuBar**

```
procedure UpdateMenuBar;
```

```
void UpdateMenuBar (void);
```

If the menu bar needs to be redrawn, call DrawMenuBar. The menu bar needs to be redrawn if the forceMBarUpdate flag is TRUE, or if the enabled state of any menu in the menu bar doesn't match the last recorded state.

## ◆ 13 *CBartender*

---

### **Command Extraction methods**

These three methods are internal methods that the bartender uses to find commands associated with menu items and to build its internal data structures. You should not use these methods. These are the THINK Pascal methods:

```
procedure ExtractCommands: (var theEntry: MenuEntry);  
procedure ParseItemString (var itemStr: Str255;  
    var cmdNo: longint);  
procedure ExtractHierMenus (macMenu: MenuHandle;  
    index: integer);
```

And these are the THINK C methods:

```
void ExtractCommands (MenuEntryP theEntry);  
void ParseItemString (Str255 itemStr, long *cmdNo);  
void ExtractHierMenus (MenuHandle macMenu,  
    short index);
```

# CBitmap

## 14

## Introduction

CBitmap is a class for working with Macintosh bit maps. You should be familiar with QuickDraw bit maps to get the most out of this class.

## Heritage

Superclass  
Subclasses

CObject  
None

## Using CBitmap

CBitmap gives you methods to draw into a bit map, to stamp a bit map on the current drawing port, and to get a bit map from the current drawing port. You may notice that this class doesn't provide a drawing method. That's because this class is only for *manipulating* bit maps. The class CBitmapPane gives you a pane for drawing bit maps.

*To learn more about transfer modes see Inside Macintosh I, Chapter 6, "QuickDraw."*

Use the CopyFrom method to copy bits from your bit map object to the bit map of the current QuickDraw port. To copy bits from the current port to your bit map object, use the CopyTo method. The transfer mode determines how the bit mapped is copied. Use the GetXferMode method to examine the current transfer mode and the SetXferMode method to set the transfer mode. The default transfer mode is srcCopy.

To draw into an off-screen bit map, bracket your drawing routines with calls to BeginDrawing and EndDrawing. When you create the bit map object, you have the choice of creating a drawing port for it or not. If you choose to create a port, your bit map will have a complete QuickDraw environment when you draw to it. If you don't, the drawing routines use the drawing environment of the current port.

In general, you should create a port for a bit map whenever you intend to draw to a bit map. If you use a bit map only to hold an image that you copied from or want to copy to the current port, you don't need to create a port.

## 14 CBitmap

For example, this Pascal procedure draws a diagonal line in an off-screen bit map.:

```
procedure DrawIntoOffScreenBitMap;
var
  myBitMap: CBitmap;
begin
  new(myBitMap);
  myBitMap.IBitMap(300, 300, TRUE);
  myBitMap.BeginDrawing;
  MoveTo(10, 10);
  LineTo(200, 200);
  myBitMap.EndDrawing;
end;
```

This is what the same routine looks like in C:

```
void DrawIntoOffScreenBitMap(void)
{
  CBitmap *myBitMap;

  myBitMap = new(CBitmap);
  myBitMap->IBitMap(300, 300, TRUE);
  myBitMap->BeginDrawing;
  MoveTo(10, 10);
  LineTo(200, 200);
  myBitMap->EndDrawing;
};
```

Once you finish drawing into your bit map, you can use the CopyFrom method to stamp it into the current port. To save the bit map as a paint (PNTG) file, you can use the CPNTGFile class.

### Variables

Variable	Type	Desktop
macPort	GrafPtr	Grafport for the bit map if requested.
savePort	GrafPtr	Used internally to save the original drawing port if necessary.
macBitMap	BitMap	The bit map.
saveBitMap	BitMap	Used internally to save the original drawing port's bit map if necessary.
xferMode	Integer	Bitmap transfer mode. Initially srcCopy.



## Methods

### Construction and destruction methods

#### **IBitMap**

```
procedure IBitmap (width, height: Integer;
  makePort: Boolean);

void IBitmap (short width, short height,
  Boolean makePort);
```

Initialize the bit map. This method allocates memory for the bit map. The origin is set to (0, 0), and the transfer mode is set to `srcCopy`. If `makePort` is `TRUE`, `IBitmap` creates a `QuickDraw` grafport whose port bits point to the bit map. In general, you should pass `TRUE` for `makePort` if you intend to draw to the bit map and `FALSE` if you use the bit map only to hold an image.

#### **Free/Dispose**

```
procedure Free;

void Dispose (void);
```

Dispose of the bit map. This method releases memory allocated to the bit map and closes the `macPort` if it was opened.

### Accessing methods

#### **GetBounds**

```
procedure GetBounds (var theBounds: LongRect);

void GetBounds (theBounds *LongRect);
```

Get the bounding rectangle of the bit map. This rectangle describes the size and coordinate system of the bit map.

#### **SetBoundsOrigin**

```
procedure SetBoundsOrigin (hOrigin, vOrigin: integer);

void SetBoundsOrigin (short hOrigin, short vOrigin);
```

Set the coordinates of the top left corner of the bit map. This method changes the coordinate system of the bit map.

#### **SetXferMode**

```
procedure SetXferMode (aXferMode: Integer);

void SetXferMode (short aXferMode);
```

Set the transfer mode of the bitmap.

#### **GetXferMode**

```
function GetXferMode: Integer;

short GetXferMode (void);
```

Return the transfer mode of the bit map.

#### **PixelIsBlack**

```
function PixelIsBlack (pixelPos: LongPt): Boolean;

Boolean PixelIsBlack (LongPt pixelPos);
```

Return `TRUE` if the pixel specified by `pixelPos` is black. Return `FALSE` if it isn't or if the specified position is not in the bit map.

### Image copying methods

#### CopyTo

```
procedure CopyTo (fromRect, toRect: LongRect;  
    maskRgn: RgnHandle);  
  
void CopyTo (LongRect *fromRect, LongRect *toRect,  
    RgnHandle maskRgn);
```

Copy bits to this bit map from the current port (thePort). FromRect is a rectangle in thePort's bit map, and toRect is a rectangle in this bit map. If maskRgn is NIL, this method does no clipping. Otherwise, the transfer is clipped to maskRgn, a region in this bit map's coordinates. The transfer mode is stored in the xferMode instance variable

#### CopyFrom

```
procedure CopyFrom (LongRect, LongRect: Rect;  
    maskRgn: RgnHandle);  
  
void CopyFrom (LongRect *fromRect, LongRect *toRect,  
    RgnHandle maskRgn);
```

Copy bits from this bit map to the current port (thePort). FromRect is a rectangle in this bit map, and toRect is a rectangle in thePort's bit map. If maskRgn is NIL, this method does no clipping. Otherwise, the transfer is clipped to maskRgn, a region in thePort's coordinates. The transfer mode is stored in the xferMode instance variable

### Drawing preparation methods

#### BeginDrawing

```
procedure BeginDrawing;  
  
void BeginDrawing (void);
```

Prepare for drawing to this bit map. If the bit map doesn't have its own port, set the port bits of the current port to this bit map. If the bit map has its own port, this method saves the current port and sets the bit map's port as the current port.

#### EndDrawing

```
procedure EndDrawing;  
  
void EndDrawing (void);
```

Reset the port to the state before drawing. If the bit map doesn't have its own port, restore thePort's port bits from saveBitMap, otherwise reset the port to savePort.

# CBitmapPane 15 ♦

---

## Introduction

CBitmapPane is a subclass of CPanorama for drawing bit maps in a pane.

## Heritage

Superclass	CPanorama
Subclasses	None

## Using CBitmapPane

CBitmapPane is a pane with an associated bit map. The drawing method of a bit map pane copies from the bit map to the pane's drawing port. That means that your program should draw into the bit map pane's bit map before the drawing method gets called. The Art Class demonstration program, which uses a subclass of CBitmapPane, draws directly on the screen and on the associated bit map in a mouse task. When the Macintosh generates an update event, the pane's drawing method copies the associated bit map right on the screen, giving a very smooth update.

Though you can use CBitmapPane directly, you will probably want to use a subclass of it so you can add instance variables and methods tailored for your application. Your program can use whatever technique is appropriate to fill the bit map with an image. If the image doesn't change, your program can just read the bit map from a file and rely on the update event to stamp the image on the screen. If it's an interactive image, you might use a mouse task like Art Class does. If it's a complex image and you want to give the illusion of a quick update, you do the drawing to the bit map in your subclass's Draw method with CBitmap's BeginDrawing and EndDrawing methods, then call the inherited Draw method to stamp the image on the screen.

*For static images, a picture pane is easier to use. See the CPicture class on page 183.*

### Variables

Variable	Type	Desktop
itsBitMap	CBitmap	The bitmap to stamp on the pane.

### Methods

#### Construction and destruction methods

##### **IBitMapPane**

```
procedure IBitmapPane (anEnclosure: CView;  
    aSupervisor: CBureaucrat;  
    aWidth, aHeight: integer;  
    aHEncl, aVEncl: integer;  
    aHSizing, aVSizing: SizingOption;  
    aBounds: LongRect;  
    aBitMap: CBitmap; makePort: Boolean);  
  
void IBitmapPane (CView *anEnclosure,  
    CBureaucrat *aSupervisor,  
    short aWidth, short aHeight,  
    short aHEncl, short aVEncl,  
    SizingOption aHSizing, SizingOption aVSizing,  
    LongRect *aBounds,  
    CBitmap *aBitMap, Boolean makePort);
```

Initialize a panorama. The first eight arguments to this routine are identical to the ones for IPane.

The aBounds rectangle is a long rectangle that defines the bounds of the panorama. The bounds define the height and width of aBitMap as well as its coordinate system.

If aBitMap is NIL, IBitmapPane creates a new bit map object for the pane, otherwise it uses aBitMap as the pane's bit map. When you use an existing bit map, be sure to get pass the correct bounds to IBitmapPane. Send the bit map a GetBounds message to get its bounds, and pass the results to IBitmapPane.

IBitmapPane passes the makePort parameter to CBitmap's IBitmap method. If aBitMap is not NIL, this parameter is ignored. The makePort parameter specifies whether the bit map should have a drawing environment. In general, you should always pass TRUE for this parameter.

---

#### Note

The descriptions of the other arguments are in CPane on page 321.

---



**Free/Dispose**

```
procedure Free;
```

```
void Dispose (void);
```

Dispose of the bit map. This method releases memory allocated to the bit map and closes the macPort if it was opened.

**Accessing methods****SetBitMap**

```
procedure SetBitMap (aBitMap: CBitMap);
```

```
void SetBitMap (CBitMap *aBitMap);
```

Get the bounding rectangle of the bit map. This rectangle describes the size and coordinate system of the bit map.

**GetBitMap**

```
function GetBitMap: CBitMap;
```

```
CBitMap *GetBitMap (void);
```

Get the bit map associated with this pane.

**Drawing method****Draw**

```
procedure Draw (var area: Rect);
```

```
void Draw (Rect *area);
```

Draw the bit map. Copy the bits in the bit map to the bit map of the current port.

## ◆ 15 CBitmapPane

---

# CBureaucrat

---

## Introduction

CBureaucrat is an abstract class that implements a link in the chain of command. Any object in the THINK Class Library that responds to a menu command or a mouse click is a bureaucrat.

A bureaucrat can either respond to a command, or it can pass the command to its supervisor. All of the default methods for bureaucrats pass the message on to the supervisor, so if a particular subclass cannot handle the message, calling the inherited method will cause the message to be passed on up the chain of command.

## Heritage

Superclass	CCollaborator
Subclasses	CDirectorOwner
	CView

## Using CBureaucrat

You generally won't need to create a subclass of CBureaucrat because most of the objects you'll use are already descendants of this class. However, you may find that you need to create a class that's a link in the chain of command that's not one of the common objects.

The global variable `gGopher` points to the current bureaucrat which is the first object in the chain of command. Whenever the switchboard responds to an event, it sends an appropriate message to the gopher. For instance, in response to a key-down event, the switchboard sends a `DoKeyDown` message to the gopher.

If the object `gGopher` refers to cannot handle the command, it passes the message up the chain of command until the message reaches an object that handles the command or until it reaches the end of the chain of command—the application.

## ◆ 16 CBureaucrat

---

When there are no documents open, gGopher points to the application. When you open a document, gGopher points to the document. Your document subclass should use the BecomeGopher method to change gGopher to point to the main pane.

Be sure to use the BecomeGopher method to change the gopher instead of setting the gGopher variable yourself. BecomeGopher uses CCollaborator's provider/dependent mechanism to let dependent objects know that the gopher has changed.

### Variables

#### Global variable

Variable	Type	Description
gGopher	CBureaucrat	The current bureaucrat.

#### Instance variable

A bureaucrat has only one instance variable.

Variable	Type	Description
itsSupervisor	CBureaucrat	The supervisor.

### Methods

#### Construction and destruction methods

##### IBureaucrat

```
procedure IBureaucrat (aSupervisor: CBureaucrat);  
void IBureaucrat (CBureaucrat *aSupervisor);  
Initialize the bureaucrat. This method sets aSupervisor to be  
itsSupervisor
```

##### Free/Dispose

```
procedure Free;  
void Dispose (void);  
If the bureaucrat is the gopher, send a BecomeGopher (TRUE) message to  
the supervisor before disposing of the object.
```

#### Accessing method

##### GetSupervisor

```
function GetSupervisor: CBureaucrat;  
CBureaucrat *GetSupervisor (void);  
Return the bureaucrat's supervisor
```

#### Command methods

These are the messages that every bureaucrat accepts. Unless a bureaucrat (or one of its ancestors) overrides one of these methods, it just passes the



message to its supervisor. If you follow the chain of supervisors—the chain of command—all the way to the end, you'll end up at your application class.

**Notify**

```
procedure Notify (theTask: CTask);  
void Notify (CTask *theTask);
```

A task has been completed. This method passes the `Notify` message to the bureaucrat's supervisor.

**DoKeyDown**

```
procedure DoKeyDown (theChar: char; keyCode: Byte;  
    macEvent: EventRecord);  
void DoKeyDown (char theChar, Byte keyCode,  
    EventRecord *macEvent);
```

Handle a key-down event. This method passes the `DoKeyDown` message to the bureaucrat's supervisor.

**DoAutoKey**

```
procedure DoAutoKey (theChar: char; keyCode: Byte;  
    macEvent: EventRecord);  
void DoAutoKey (char theChar, Byte keyCode,  
    EventRecord *macEvent);
```

Handle an auto-key event. This method passes the `DoAutoKey` message to the bureaucrat's supervisor.

**DoKeyUp**

```
procedure DoKeyUp (theChar: char; keyCode: Byte;  
    macEvent: EventRecordPtr);  
void DoKeyUp (char theChar, Byte keyCode,  
    EventRecord *macEvent);
```

Handle a key-up event. This method passes the `DoKeyUp` message to the bureaucrat's supervisor.

---

**Note**

By default, the Toolbox Event Manager masks out key-up events.

---

**DoCommand**

```
procedure DoCommand (theCommand: longint);  
void DoCommand (long theCommand);
```

Handle a command. This method passes the `DoCommand` message to the bureaucrat's supervisor.

### DoAppleEvent

```
procedure DoAppleEvent (anAppleEvent: CAppleEvent);  
void DoAppleEvent (CAppleEvent *anAppleEvent);
```

Handle an AppleEvent. This method passes the AppleEvent to the bureaucrat's supervisor.

### Dawdle

```
procedure Dawdle (var maxSleep: longint);  
void Dawdle (long *maxSleep);
```

Perform periodic actions at idle time. The default method does nothing. CApplication's Idle method sends a Dawdle message to all of the bureaucrats in the chain of command.

If your bureaucrat requires periodic actions, perform them in the Dawdle method. Set the value of maxSleep to the largest number of ticks that your application can tolerate between Dawdle messages. If your application's Dawdle message doesn't have any time constraints, you can ignore maxSleep.

The application class uses the value of maxSleep to let WaitNextEvent know that it should yield an event after at most maxSleep ticks. For instance, the Dawdle method for CEditText calls TEIdle to blink the insertion point and sets maxSleep to GetCaretTime which is the rate at which the insertion point blinks.

A blinking insertion point is a good example. A clock display is another good example. You would update the display from your document's Dawdle method. If your clock displays seconds, set maxSleep to 60 since you need to update the display at least every 60 ticks.

On a null event, CApplication's Idle method sends a Dawdle message to every bureaucrat in the chain of command. Idle sets the global variable gSleepTime to the smallest sleep time that all bureaucrats requested. The switchboard uses this global variable in its call to WaitNextEvent.

### UpdateMenus

```
procedure UpdateMenus;  
void UpdateMenus (void);
```

Update the menus that have to do with this bureaucrat. Your bureaucrat (usually a document or a pane) should override this method to enable the menu items that pertain to it. The important thing to remember when you write an UpdateMenus method is to call inherited UpdateMenus before you enable your own menus. In this way, your supervisor's menus will be updated first. See CBartender, especially section "Dimming and checking menu items" on page 171, for a discussion of menu updating.

**BecomeGopher**

```
function BecomeGopher (fBecoming: Boolean): Boolean;  
Boolean BecomeGopher (Boolean fBecoming);
```

If `fBecoming` is `TRUE`, make this bureaucrat the gopher and call `BroadcastChange` with `bureaucratIsGopher` as the reason. If `fBecoming` is `FALSE`, call `BroadcastChange` with `bureaucratIsNotGopher` as the reason.

If the current gopher refuses to relinquish control, this method returns `FALSE` and does not call `BroadcastChange` at all.

**Change notification methods**

These methods override methods in `CCollaborator`. To learn how to use them, see `CCollaborator` on page 223.

**BroadcastChange**

```
procedure BroadcastChange (reason: longint;  
    info: Ptr);  
void BroadcastChange (long reason, void* info);
```

In addition to notifying this bureaucrat's dependents of a change, notify its supervisor as well. Reason is an integer that describes the type of change. You must define reasons for your subclass. Info is a pointer to any additional information needed to respond to the change. If you want objects that aren't in a collaborator's list of dependents to know about a change, override this method

**ProviderChanged**

```
procedure ProviderChanged (aProvider: CCollaborator;  
    reason: longint; info: Ptr);  
void ProviderChanged (CCollaborator *aProvider,  
    long reason, void* info);
```

One of this bureaucrat's providers has just changed. This method passes the change notification to the bureaucrat's supervisor. If you want your subclass to respond to change notifications, you should override this method. `aProvider` is the provider that changed. Reason is an integer that describes the type of change. You must define reasons for your subclass. Info is a pointer to any additional information that your subclass might need.

## ◆ 16 *CBureaucrat*

---



# *CButton* ♦

## 17

---

### Introduction

CButton implements a standard Macintosh push button. You can use a button in any pane or window.

### Heritage

Superclass	CControl
Subclasses	CCheckBox CRadioControl

### Using CButton

CButton implements a standard Macintosh push button. You should not override this class unless you want to implement a push button that works differently from the default button. (The other kinds of Macintosh buttons, check boxes and radio buttons, are subclasses of this class.)

The CButton class handles all the mouse tracking for you. When you click and release the mouse button within a button, the button sends a DoCommand message to its supervisor. To specify which command number the button will send in the DoCommand message, send it a SetClickCmd message.

This is what happens when you click in a button. The DoClick method that CButton inherits from CControl calls the Macintosh Toolbox routine TrackControl to highlight the button and track mouse movement. When you release the mouse within a button, the DoClick method sends the control a DoGoodClick message. The DoGoodClick method sends the DoCommand message to the button's supervisor.

### Variables

Variable	Type	Description
clickCmd	longint	Command number to send after a click. The default is cmdNull.

### Methods

Although CButton implements only these methods, you can use the methods it inherits from CControl and CView to manipulate a button's title and location.

#### IButton

```
procedure IButton (CNTLid: integer;
  anEnclosure: CView; aSupervisor: CBureaucrat);
void IButton (short CNTLid, CView *anEnclosure,
  CBureaucrat *aSupervisor);
```

Initialize a button from a CNTL resource (a control template). CNTLid is the resource ID of the CNTL resource. AnEnclosure is the pane or window that the control belongs to. ASupervisor is the control's supervisor, typically a document.

IButton places the control at the location specified in the CNTL resource. If you want to position the button from your program, send it a Place message. (Controls inherit the Place method from CPane.)

#### INewButton

```
procedure INewButton (aWidth, aHeight: integer;
  aHEncl, aVEncl: integer; title: StringPtr;
  fVisible: Boolean; procID: integer;
  anEnclosure: CView; aSupervisor: CBureaucrat);
void INewButton (short aWidth, short aHeight,
  short aHEncl, short aVEncl, StringPtr title,
  Boolean fVisible, short procID,
  CView *anEnclosure, CBureaucrat *aSupervisor);
```

Initialize a button from the parameters in the argument list. Title is the name of the button. FVisible specifies whether the button is drawn initially. ProcID specifies what type of button to create. Its possible values are defined in Controls.h and include pushButProc, checkBoxProc, and radioButProc.

---

#### Note

The descriptions of the other arguments are in CPane on page 321.

---



<b>GetClickCmd</b>	<pre>function GetClickCmd: longint; long GetClickCmd (void);</pre> <p>Return the command which is sent to a button's supervisor when the user clicks in a button.</p>
<b>SetClickCmd</b>	<pre>procedure SetClickCmd (aClickCmd: longint); void SetClickCmd (long aClickCmd);</pre> <p>Set the command number to be sent when the user clicks in a button.</p>
<b>SetDefault</b>	<pre>procedure SetDefault (fDefault: Boolean); void SetDefault (Boolean fDefault);</pre> <p>Specify whether the button is the default button. The default button has a three-pixel thick rounded rectangle as its border.</p>
<b>SimulateClick</b>	<pre>procedure SimulateClick; void SimulateClick (void);</pre> <p>Simulate a click in this button. Use this method when you want to provide visual feedback for a Command key shortcut. This method highlights the button momentarily, then sends a DoGoodClick message to the button.</p>
<b>DoGoodClick</b>	<pre>procedure DoGoodClick (whichPart: integer); void DoGoodClick (short whichPart);</pre> <p>When you press and release the mouse within the button, this method sends a DoCommand message to its supervisor. This is an internal method. You should not use or override this method.</p>

## ◆ 17 CButton

---

# CCharGrid

## 18

### Introduction

CCharGrid is a subclass of CGridSelector that displays characters in a table and lets you choose one. CCharGrid is useful for implementing tool palettes like MacPaint and HyperCard.

### Heritage

Superclass	CGridSelector
Subclasses	None

### Using CCharGrid

The CCharGrid class lets you create panes that display characters in a table. The most common use for this kind of table is a tool palette. You can use a CCharGrid as a tool palette that's part of a window or as a custom tear-off menu. The Art Class demonstration program uses CCharGrid to display its Tool tear-off menu.

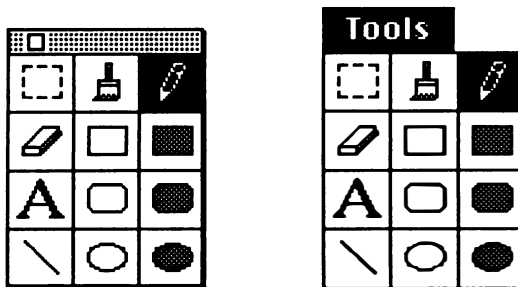


Figure 18-1 Art Class uses CCharGrid as a palette and as a menu

To use a CCharGrid, you need to create a special font that contains your tools. You can use ResEdit to create the font, or you can use a commercial a bit-map font editor. Apple recommends that the family ID for these kinds of fonts be in the range 32256 to 32767. Check with the Font Manager chapters

## 18 CCharGrid

in *Inside Macintosh I* and *Inside Macintosh IV* for detailed information about working with fonts..

Unlike other subclasses of CPane that optionally let you initialize an object from a resource, you *must* use a resource to initialize a CCharGrid. The first argument to the initialization method, ICharGrid, is the resource ID of a ChGd resource which contains the values that ICharGrid passes up to CGridSelector's initialization method. The ChGd resource looks like this:

Field	Size	Description
Rows	integer	The number of rows in the grid.
Columns	integer	The number of columns in the grid.
Box Width	integer	The width in pixels of each box.
Box Height	integer	The height in pixels of each box.
Horiz. Sizing	integer	Horizontal sizing option. Usually <code>sizeFIXEDLEFT</code> (0)
Vert. Sizing	integer	Vertical sizing option. Usually <code>sizeFIXEDTOP</code> (2)
Horiz. Location	integer	Horizontal location of grid in its enclosure.
Vert. Location	integer	Vertical location of grid in its enclosure.
Command base	integer	The command base for turning selections into command numbers.
Font Size	integer	The size in points of the font.
Font Name	Pascal string	The name of the font to display.
theCharacters	Pascal string	Handle to a Pascal string of the characters to display in the grid.

*If the first character of your font name is "%" or ".", it won't appear in your Font menu.*

The first nine fields are the same as the arguments that you pass to IGridSelector. The last three are specific to character grids. The Font Size and Font Name fields specify the size and the name of the font to use. The font name is a Pascal string. The last field contains the characters to display. It is also a Pascal string.

The file `TCL TMPLs` contains ResEdit templates ('TMPL' resources) that help you create and edit 'ChGd' resources. This is what the 'ChGd' resource in Art Class looks like when you edit it with ResEdit.

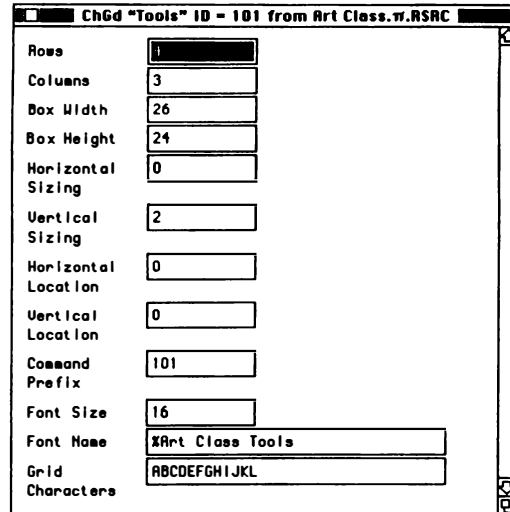


Figure 18-2 The ChGd resource in Art Class

## Variables

Variable	Type	Description
<code>theCharacters</code>	Handle	Handle to a Pascal string of the characters to display in the grid.

## Methods

### Construction methods

#### ICharGrid

```
procedure ICharGrid. (ChGdid: integer;
    anEnclosure: CView; aSupervisor: CBureaucrat);
void ICharGrid (short ChGdid, CView *anEnclosure,
    CBureaucrat *aSupervisor);
```

Initialize the character grid. `ChGdid` is the resource ID of the 'ChGd' resource that describes the location of the pane and the size and name of the font to use for the grid. See "Using CCharGrid" above for the details on 'ChGd' resources. `ICharGrid` creates a text environment object of class `CTextEnvirons` to maintain the text characteristics whenever the characters are drawn. The text transfer mode is initially `srcOr`.

## ◆ 18 CCharGrid

---

### Free/Dispose

```
procedure Free;  
void Dispose (void);  
Dispose of the character list and the object.
```

### Drawing methods

### DrawItem

```
procedure DrawItem (theItem: integer;  
    theBox: Rect);  
void DrawItem (short theItem, Rect *theBox);  
Draw character numbered theItem so it is centered within the box. Char-  
acters are numbered from 1.
```



# CCheckBox

---

# 19



## Introduction

CCheckBox implements a standard Macintosh check box. You can use a check box in any pane or window.

## Heritage

Superclass	CButton
Subclasses	None

## Using CCheckBox

CCheckBox implements a standard Macintosh check box. You should not override this class unless you want to implement a check box that works differently from the default check box.

The CCheckBox class handles all the mouse tracking for you. When you click and release the mouse button within a check box, it toggles the value of the check box and sends a DoCommand message to its supervisor. To specify which command number the check box will send in the DoCommand message, send it a SetClickCmd message.

---

### Note

CCheckBox inherits the SetClickCmd method from CButton.

---

This is what happens when you click in a check box. The DoClick method that CCheckBox inherits from CControl calls the Macintosh Toolbox routine TrackControl to highlight the check box and track mouse movement. When you release the mouse within a check box, the DoClick method sends the control a DoGoodClick message. The DoGoodClick method toggles the value of the control and sends the DoCommand message to the check box's supervisor.

To find out whether the check box is checked, send it an `IsChecked` message. You can set the value of the check box from your program by sending it a `SetValue` message. Use the values `BUTTON_ON` and `BUTTON_OFF` to specify whether to check or uncheck the check box.

---

### Note

Setting the value with `SetValue` will not send a `DoCommand` message to the check box's supervisor.

---

## Variables

This class has no instance variables.

## Methods

Although `CCheckBox` implements only these methods you can use the methods it inherits from `CControl` and `CView` to manipulate a check box's title and location.

You should use the `SetClickCmd` method that `CCheckBox` inherits from `CButton` to specify the command the check box sends to its supervisor when you click in it.

### ICheckBox

```
procedure ICheckBox (CNTLid: integer;  
    anEnclosure: CView; aSupervisor: CBureaucrat);  
void ICheckBox (short CNTLid, CView *anEnclosure,  
    CBureaucrat *aSupervisor);
```

Initialize a check box from a CNTL resource (a control template). `CNTLid` is the resource ID of the CNTL resource. `AnEnclosure` is the pane or window that the control belongs to. `ASupervisor` is the control's supervisor, typically a document.

`ICheckBox` places the control at the location specified in the CNTL resource. If you want to position the check box from your program, send it a `Place` message. (Controls inherit the `Place` method from `CPane`.)

### INewCheckBox

```
procedure INewCheckBox (aWidth, aHeight: integer;  
    aHEncl, aVEncl: integer;  
    title: StringPtr; fVisible: Boolean;  
    anEnclosure: CView; aSupervisor: CBureaucrat);  
void INewCheckBox (short aWidth, short aHeight,  
    short aHEncl, short aVEncl,
```



---

```
StringPtr title, Boolean fVisible,  
CView *anEnclosure, CBureaucrat *aSupervisor);
```

Initialize a check box from the parameters in the argument list. Title is the name of the check box. FVisible specifies whether the check box is drawn initially.

---

**Note**

The descriptions of the other arguments are in CPane on page 321.

---

**DoGoodClick**

```
procedure DoGoodClick (whichPart: integer);  
void DoGoodClick (short whichPart);
```

When you press and release the mouse within the check box, this method toggles the state of the check box and sends a DoCommand message to its supervisor.

This is an internal method. You should not use or override this method.

**IsChecked**

```
function IsChecked: Boolean;  
Boolean IsChecked (void);  
Returns TRUE if the check box is checked.
```

## ◆ 19 CCheckBox

---

# CChore 20 ♦

---

## Introduction

CChore is an abstract class for implementing periodic or urgent actions. The THINK Class Library executes periodic chores at idle time and urgent chores after processing the current event.

## Heritage

Superclass	CObject
Subclasses	CBarChore CTearChore

## Using CChore

You can use this class to create chores that run at idle time or that run at the next possible opportunity. Your application maintains a list of chores and implements the methods to install and remove them.

To implement a chore, you need to define a subclass of this class. In your subclass, override the `Perform` method. This method implements the action you want your chore to take.

### Idle chores

To install an idle chore, send it to the application in an `AssignIdleChore` message. Your application sends a `Perform` message to all the idle chores at idle time. To remove an idle chore, send the chore you want to remove to the application in a `CancelIdleChore` message. This example shows how to install an idle chore in THINK Pascal:

```
gApplication.AssignIdleChore(theChore);
```

And this example shows how to install an idle chore in THINK C:

```
gApplication->AssignIdleChore(theChore);
```

---

### Note

To remove an idle chore, keep it in a variable since `CancelIdleChore` requires a reference to a `CChore` object.

---

### Urgent chores

An urgent chore is one that gets executed immediately after the application processes the current event. Typically, it's the same event that caused the urgent chore to be installed. Urgent chores are executed only once. They're removed from the urgent chore list immediately after they're executed. To install an urgent chore, send it to the application in an `AssignUrgentChore` message.

### Using chores

Chores are attached to the application and not to any particular document. The things that a chore does, then, should pertain to the entire application.

---

### Note

If you need your chore to run at a specific interval, you can use the Macintosh Toolbox routine `Ticks` to store the last time the chore was performed.

---

For periodic tasks that apply to a particular document or pane, use the `Dawdle` method. All bureaucrats, including documents and panes, inherit a `Dawdle` method. The application sends a `Dawdle` message to each bureaucrat in the chain of command. For instance, you would use a `Dawdle` method to blink an insertion point in a pane.

## Variables

This class has no instance variables. Your `CChore` subclass may define its own instance variables. For example, if you want your idle chore to run at specified intervals, you could use an instance variable to store the period.

## Methods

`CChore` implements only one method, which you must override. Your `CChore` subclass may implement additional methods.

### Perform

```
procedure Perform (var maxSleep: longint);  
void Perform (long *maxSleep);
```

This is the method that executes your chore. You must override this method in your `CChore` subclass. The default method does nothing.



The `maxSleep` parameter specifies how many ticks your chore can tolerate passing before it gets a `Perform` message. For example, if you need to do a chore at least once every tenth of a second, you would set `maxSleep` to 6. (Each tick is one-sixtieth of a second.)

For a more detailed discussion of `maxSleep`, see the description of the `Dawdle` method in `CBureaucrat` on page 190.





# CClipboard ♦

## 21

---

### Introduction

CClipboard implements a standard Macintosh clipboard, or scrap. The default clipboard handles TEXT and PICT scraps. To deal with other kinds of scraps or to implement a private scrap, create a subclass of this class.

### Heritage

Superclass  
Subclasses

CDirector  
To implement a private scrap, you should define a subclass.

### Using CClipboard

This class implements a standard Macintosh clipboard. CClipboard uses the desk scrap to store its data and supports a window to display the contents of the scrap. This implementation supports only TEXT and PICT data.

When you initialize your application, the default initialization method, `IAApplication`, sends a `MakeClipboard` message. This method creates an instance of CClipboard and stores it in the global variable `gClipboard`.

The default application `DoCommand` method handles the `cmdToggleClip` command which shows and hides the Clipboard window. When you're running under MultiFinder, the Clipboard window is hidden when your application is suspended. When the application resumes, the Clipboard window becomes visible again.

## 21 CClipboard

---

### Note

In the THINK Class Library, desk accessories behave as if they were in their own layer even if your application isn't running under MultiFinder. When a desk accessory window is frontmost, the Clipboard window will be hidden. When one of your application windows is frontmost, the Clipboard window will be visible again.

---

### Implementing a CClipboard subclass

If you want your application to support a private scrap, or if you want to display other types of data, you'll need to implement a subclass of CClipboard. You'll also need to override the `MakeClipboard` method in your application class.

---

### Note

If you create a subclass of this class, be sure you understand how the Macintosh scrap mechanism works. See *Inside Macintosh I*, Chapter 15, "The Scrap Manager."

---

In your CClipboard subclass you'll need to override these methods:

Method	Action
<code>PutData</code>	Place data in the private scrap.
<code>GetData</code>	Retrieve data from the private scrap.
<code>ConvertGlobal</code>	Place the contents of the desk scrap into the private scrap.
<code>ConvertPrivate</code>	Place the contents of the private scrap into the desk scrap.
<code>MakeClipView</code>	Create a view to display data other than plain text or pictures.

## Variables

The global variable `gClipboard` contains a pointer to the single instance of the clipboard object. The value of this variable is set in the application method `MakeClipboard`.

### Global variable

Variable	Type	Description
<code>gClipboard</code>	<code>CClipboard</code>	The global clipboard.

**Instance variables**

Variable	Type	Description
<code>itsContents</code>	<code>CPanorama</code>	Pane for displaying contents
<code>itsScrollPane</code>	<code>CScrollPane</code>	Contents can be scrolled
<code>theLength</code>	<code>longint</code>	Length from last Get operation
<code>theOffset</code>	<code>longint</code>	Offset from last Get operation
<code>lastScrapCount</code>	<code>integer</code>	Count at the last conversion between global and private scraps.
<code>privateNewer</code>	<code>Boolean</code>	TRUE if the private scrap has changed since the last conversion to the desk scrap.
<code>windowVisible</code>	<code>Boolean</code>	TRUE if the Clipboard window is visible.

**Methods****Construction and destruction methods****IClipboard**

```
procedure IClipboard (aSupervisor: CApplication;
  hasWindow: Boolean);
```

```
void IClipboard (CApplication *aSupervisor,
  Boolean hasWindow);
```

Initialize the clipboard. The application's `MakeClipboard` method sends this message and stores the clipboard object in the global variable `gClipboard`.

**Note**

If you create a subclass of `CClipboard`, be sure to override the `MakeClipboard` method in your application subclass as well so it creates an object of the clipboard subclass. See the implementation of `MakeClipboard` in `CApplication` for an example.

## 21 CClipboard

---

### Suspend and resume methods

These methods handle scrap conversion when your application moves from foreground to background under MultiFinder. In most cases, you won't need to use or override these methods.

#### Suspend

```
procedure Suspend;  
void Suspend (void);
```

The application is about to be switched into the background under MultiFinder. If the application's private scrap is newer than the desk scrap, this method sends `ConvertPrivate` and `ScrapConverted` messages to the clipboard. This method then sends the clipboard window a `HideSuspend` message.

#### Resume

```
procedure Resume;  
void Resume (void);
```

The application is about to be brought to the foreground under MultiFinder. If the desk scrap is newer than the private scrap, this method sends the clipboard `ConvertGlobal`, `ScrapConverted`, and `UpdateDisplay` messages to update the contents of the clipboard. This method then sends the window a `ShowResume` message to make it visible.

### Appearance methods

These methods handle the appearance of the Clipboard window and how the contents of the clipboard appear in the window. If you implement your own clipboard class, you'll need to override the `UpdateDisplay` method.

#### Close

```
function Close (quitting: Boolean): Boolean;  
Boolean Close (Boolean quitting);
```

The user chose **Close** from the **File** menu. `Close` sends a `CloseWind` message and returns `TRUE`.

#### CloseWind

```
procedure CloseWind (theWindow: CWindow);  
void CloseWind (CWindow *theWindow);
```

The user clicked in the window's close box. This method hides the Clipboard window and changes the text in the menu item.

#### Toggle

```
procedure Toggle;  
void Toggle (void);
```

This method opens the Clipboard window if it's closed or closes it if it's open. The application's default `DoCommand` method sends this message.

**UpdateDisplay**

```
procedure UpdateDisplay;
void UpdateDisplay (void);
```

Display the contents of the scrap in the Clipboard window. This method supports only TEXT and PICT types of data. This method will work correctly with a private scrap, but if you want to display other kinds of data, you'll need to override this method.

**Accessing methods**

Use these methods to place data in and retrieve data from the desk scrap. If your application implements a private scrap, use the PutData and GetData methods instead.

**PutGlobalScrap**

```
procedure PutGlobalScrap (theType: ResType;
    theData: Handle);
void PutGlobalScrap (ResType theType, Handle theData);
```

Put theData of type theType into the desk scrap. Be sure to send an EmptyGlobalScrap message before you call this method.

**GetGlobalScrap**

```
function GetGlobalScrap (theType: ResType;
    theData: Handle): Boolean;
Boolean GetGlobalScrap (ResType theType,
    Handle theData);
```

Get the data of type theType from the desk scrap and put it in the block that theData is a handle to. Return TRUE if the method was able to fulfill the request, FALSE otherwise.

TheData must be an allocated handle. The size of the allocated memory will grow as needed to fit the data. This is how you would get the TEXT data from the desk scrap in THINK Pascal:

```
var
    myData: Handle;
begin
    { Create a zero-sized handle }
    myData := NewHandle(0);
    gClipboard.GetGlobalData ('TEXT', myData);
end
```

## 21 CClipboard

And this is how you would get TEXT data from the desk scrap in THINK C:

```
Handle myData;
...
/* Create a zero-sized handle */
myData = NewHandle(0);
gClipboard->GetGlobalData('TEXT', myData);
```

### DataSize

```
function DataSize (theType: ResType): longint;
long DataSize (ResType theType);
```

Return the number of bytes of data in the clipboard of type theType. If the clipboard doesn't contain any data of the specified type, this method returns zero.

### Status

```
function Status: ScrapStatus;
ScrapStatus Status (void);
```

Return the status of the scrap. This method returns a value that describes the relationship between the private scrap and the desk scrap.

Value	Meaning
PRIVATE_SCRAP_NEWER	The information in the private scrap is newer than the information in the desk scrap.
GLOBAL_SCRAP_NEWER	The information in the desk scrap is newer than the information in the private scrap.
SCRAPS_THE_SAME	The information in both scraps is the same.

### ScrapConverted

```
procedure ScrapConverted;
void ScrapConverted (void);
Set the internal flags after the scrap has been converted.
```

#### Scrap conversion methods

Use the PutData and GetData methods to implement **Cut**, **Copy**, and **Paste** commands for your application. If your application uses only the desk scrap, you can use the PutGlobalScrap and GetGlobalScrap methods.

### PutData

```
procedure PutData (theType: ResType; theData: Handle);
void PutData (ResType theType, Handle theData);
Put theData of type theType into the scrap. The default method puts the data in the desk scrap. If your subclass supports a private scrap, you must
```



override this method. After you store your data in the private scrap, be sure to send a `PrivateChanged` message. Before you call this method, be sure you send an `EmptyScrap` method to clear your scrap.

### **GetData**

```
function GetData (theType: ResType,
                 var theData: Handle): Boolean;
```

```
Boolean GetData (ResType theType, Handle *theData);
```

Get the data of type `theType` and put it into a newly created handle. This method returns `TRUE` if the request was successful, `FALSE` otherwise. If your application supports a private scrap, you must override this method.

Unlike `GetGlobalScrap` you should not allocate the handle to `theData`. This method will allocate the memory. This is how you'd get data of type `TEXT` in THINK Pascal:

```
var
    Handle    myData;
begin
    gClipboard.GetData('TEXT', myData);
    ...
end;
```

And this is how you'd get data of type `TEXT` in THINK C:

```
Handle myData;
gClipboard->GetData('TEXT', &myData);
```

### **ConvertGlobal**

```
procedure ConvertGlobal;
void ConvertGlobal (void);
```

Convert data in the desk scrap and put it in the private scrap. The default method does nothing. If your application supports a private scrap, you must override this method.

### **ConvertPrivate**

```
procedure ConvertPrivate;
void ConvertPrivate (void);
```

Convert data in the private scrap and put it into the desk scrap. The default method does nothing. If your application supports a private scrap, you must override this method.

### **EmptyGlobalScrap**

```
procedure EmptyGlobalScrap;
void EmptyGlobalScrap (void)
```

Clear the contents of the desk scrap.

## ◆ 21 CClipboard

---

### EmptyScrap

```
procedure EmptyScrap;  
void EmptyScrap (void)
```

Clear the contents of the scrap that is the destination of PutData. You should call this method before you call PutData. The default method clears the global scrap.

### PrivateChanged

```
procedure PrivateChanged;  
void PrivateChanged (void);
```

The data in the private scrap has changed. If you implement a private scrap, be sure to send this message to the clipboard after you put new data in your private scrap. This method sets the internal flags and sends an UpdateDisplay message to the clipboard. Your subclass should not override this method.

### MakeClipView

```
function MakeClipView (dataType: longint;  
    dataHandle: Handle): CPanorama;  
CPanorama *MakeClipView (long dataType,  
    Handle dataHandle)
```

Make a view to display the clipboard data dataHandle of type dataType. This method disposes dataHandle when appropriate. UpdateDisplay calls this method to create the right kind of pane for the data you want to display in the Clipboard window.

If you create a subclass of CClipboard, you need to override this method to create the kind of pane that you need to display your data. If dataType is 'PICT' or 'TEXT', you can call the inherited method.

## Class resources

The WIND resource for the Clipboard window is in the file TCL Resources which contains all of the resources the THINK Class Library requires.

Resource	Description
WIND 200	Window template for Clipboard window.



# CCluster

---

## Introduction

CCluster implements an unordered list of objects.

## Heritage

Superclass  
Subclasses

CArray  
CList  
CStack

## Using CCluster

Use a CCluster object whenever you need to maintain an unordered list of objects. Typically, the elements of the list are object references, though you can store anything you want in a cluster. The default size for an item in a cluster is four bytes, which is large enough to hold a pointer or a handle. Several objects in the THINK Class Library use CCluster objects or descendants of CCluster to maintain lists.

---

### Note

If you need an ordered list of objects, use the CList class instead.

---

CCluster allocates space for each object reference in blocks. By default, a block holds three slots. Each slot holds one object reference. CCluster takes care of allocating blocks automatically, though you can change the number of slots per block if you like.

### Variables

This instance variable has the same value as `hItems` but a different type. It's kept for compatibility with the previous version of the THINK Class Library.

Variable	Type	Description
<code>items</code>	<code>LongHandle</code>	Handle to the items in the cluster.

### Methods

In addition to the insertion, deletion, and searching methods, CCluster implements iteration methods that let you apply a function to all the objects in a cluster.

#### Construction and destruction

##### ICluster

```
procedure ICluster;  
void ICluster (void);  
Initialize the cluster.
```

##### Free/Dispose

```
procedure Free;  
void Dispose (void);  
Dispose of a cluster, but not the items in it.
```

##### DisposeAll

```
procedure DisposeAll;  
void DisposeAll (void);  
Dispose of a cluster and all the items in it. This method sends a Free (in THINK Pascal) or Dispose (in THINK C) message to every item in the cluster.
```

##### DisposeItems

```
procedure DisposeItems;  
void DisposeItems (void);  
Dispose of the items in a cluster, but not the cluster.
```

#### Insertion and deletion

These methods add objects to and remove objects from the cluster. Remember that the items in the cluster are not in any particular order.

##### Add

```
procedure Add (theObject: CObject);  
void Add (CObject *theObject);  
Add theObject to the cluster.
```

**Remove**

```
procedure Remove (theObject: CObject);
void Remove (CObject *theObject);
```

Remove theObject from the collection if it is already in the cluster.

**Membership**

These methods let you determine whether a particular item is in the cluster.

**Includes**

```
function Includes (theObject: CObject): Boolean;
Boolean Includes (CObject *theObject);
```

Return TRUE if theObject is in the cluster.

**FindItem**

```
function FindItem (function MyFunc(obj: CObject):
    Boolean): CObject;
CObject* FindItem (TestFunc MyFunc);
```

Look for an item that satisfies a function. FindItem applies the function MyFunc to each item in the list and returns the first one that causes the function to return TRUE.

In THINK C, you must declare MyFunc like this:

```
Boolean MyBoolFunc (CObject *obj);
```

**FindItem1**

```
function FindItem1 (function MyFunc1(obj: CObject;
    theParam: Ptr); param: Ptr): CObject;
CObject* FindItem1 (TestFunc1 MyFunc1, long param);
```

Look for an item that satisfies a function. FindItem1 applies the function MyFunc1 to each item in the list and returns the first one that causes the function to return TRUE. Param is an extra parameter you can pass to FindItem1 if MyFunc needs more information.

In THINK C, you must declare MyFunc like this:

```
Boolean MyFunc1 (CObject *obj, long param);
```

**Offset**

```
function Offset (theObject: CObject): longint;
long Offset (CObject *theObject);
```

Returns the index of theObject in the collection. If theObject is not in the collection, Offset returns BAD\_INDEX. This is an internal method. In most cases you won't need to use this method.

### Iteration

These methods let you apply a function to all of the objects in a cluster. The function you apply to the objects in the cluster should not affect the cluster itself.

To apply a message to all the items in a cluster, you need to write a message-sending function. It's up to you to make sure that all the items in the cluster can respond to the message.

For example, this is how you send a Draw message to all the items in a cluster, in THINK Pascal:

```
procedure Perform_Draw (thePane: CPane;  
    area: Ptr)  
begin  
    thePane.Draw (RecPtr (area) ^);  
end  
  
theCluster.DoForEach1 (Perform_Draw, @area);
```

---

### Note

RectPtr is defined in TCL.p as ^Rect.

---

And this is how you send a Draw message to all the items in a cluster, in THINK C:

```
void Perform_draw (CPane *thePane, Rect *area)  
{  
    thePane->Draw (area);  
}  
  
myCluster->DoForEach1 (Perform_Draw,  
    (long) (&myRect));
```

### DoForEach

```
procedure DoForEach (procedure Proc (theObject:  
    CObject));
```

```
void DoForEach (EachFunc Proc);
```

Apply the procedure Proc to each object in the collection. In THINK C, you must declare MyFunc like this:

```
void MyFunc (CObject *theObject);
```

**DoForEach1**

```
procedure DoForEach1 (procedure Proc (theObject:
    CObject; theParam: Ptr); param: Ptr);
```

```
void DoForEach1 (EachFunc1 Proc, long param);
```

Apply the procedure Proc to each object in the collection along with the parameter param. In THINK C, you must declare MyFunc like this:

```
void MyFunc1 (CObject *theObject, long param);
```



# CCollaborator

---

## 23

### Introduction

CCollaborator is an abstract class that implements objects that depend on one another. This class provides a mechanism for an object to announce changes to other objects.

### Heritage

Superclass	CObject
Subclasses	CBureaucrat CCollection

### Using CCollaborator

This class implements a mechanism that lets an object announce changes to other objects. The object that other objects depend on is called the **provider**. The objects that depend on the provider are called **dependents**.

For example, suppose that your program displays the contents of a file as different kinds of graphs in different windows. Whenever the data in the file change, the windows that display the data should also change. The data file is the provider. The graph display windows are the dependents.

To specify a dependency, send the dependent object a `DependUpon` message with the provider as the argument. When the provider changes, send the provider a `BroadcastChange` message. That method sends each of the dependents a `ProviderChanged` message.

The `BroadcastChange` method takes a parameter called `reason` that lets the dependents know why they're being notified. Reasons are implemented as long integers. In C, you can use `#define` directives or enums. In Pascal, you can use `const` declarations.

Reason names follow two conventions. First, reason names begin with the name of their class, without the C. For example, all of `CArray`'s reasons begin

## 23 CCollaborator

with array. Second, each collaborator subclass has a reason named *classnameLastChange*. For example, CArray defines a reason named *arrayLastChange*. Using *LastChange* ensures that your class and all its subclasses have unique reason numbers. Set *LastChange* to the last reason your class defines. When you define a subclass, set the subclass's first reason to the superclass's *LastChange* plus one. For example, these are the reason definitions for the CArray class in THINK C:

```
enum
{
    arrayInsertElement = 1,
    arrayDeleteElement,
    arrayMoveElement,
    arrayElementChanged,

    arrayLastChange = arrayElementChanged
};
```

Note that *arrayLastChange* is set to *arrayElementChanged*, the last reason.

These are the reasons for CRunArray, a subclass of CArray:

```
enum
{
    runArraySizeChanged = arrayLastChange + 1,
    runArrayLastChange = runArraySizeChanged
};
```

Note that CArray's first (and only) reason is *arrayLastChange + 1*.

The examples below illustrate how you might implement the graphing program described above. When the data in the data file change, the data file's *UpdatedData* method sends a *BroadcastChange* message with the reason *dataUpdated*. The *BroadcastChange* method sends each of the data file's graphs, the data file's dependents, a *ProviderChanged* method, which updates the picture of the data in the graph.



This is what it might look like in THINK C:

```
enum
{   myFileUpdated = 1,
    myFilePointAdded,
    myFilePointDeleted,

    myFileLastChange = myFileDeleted
};
. . .

void CMyFile::UpdateData (MyDataPtr oldData,
    MyDataPtr newData)
{
    . . .
    BroadcastChange(myFileUpdated,
        (void *) newData );
}

void CMyGraphPane::ProviderChanged (
    CCollaborator *aProvider,
    long reason, void *info)
{
    if (aProvider == itsDataFile)
        switch (reason)
        {
            case myFileUpdated:
                UpdateGraphData(info);
                break;
            case myFilePointAdded:
                AddGraphData(info);
                break;
            case myFilePointDeleted:
                DeleteGraphData(info);
                break;
        }
}
```

*Be sure to check the class of the provider. You can make sure that the provider is a known object as in the example here, or you can check for membership with the member(). The preferred way is to check that it is a known object.*

This is the same example in THINK Pascal:

```
const
    myFileUpdated = 1;
    myFilePointAdded = myFileUpdated + 1;
    myFilePointDeleted = myFilePointAdded + 1;

    myFileLastChange = myFileDeleted;
. . .
```

## 23 CCollaborator

```
procedure CMyFile.UpdateData (oldData,
    newData: MyDataPtr)
begin
    BroadcastChange (longint (myFileUpdated),
        Ptr (newData) );
end;

procedure CMyGraphPane.ProviderChanged (
    aProvider: CCollaborator;
    reason: longint; info: Ptr)
begin
    if aProvider = itsDataFile then
        case MyReason (reason) of
            myFileUpdated:
                UpdateGraphData (info);
            myFilePointAdded:
                AddGraphData (info);
            myFilePointDeleted:
                DeleteGraphData (info);
        end
    end;
end;
```

*Be sure to check the class of the provider. You can make sure that the provider is a known object as in the example here, or you can check for membership with the member (). The preferred way is to check that it is a known object.*

### Variables

Variable	Type	Description
itsProviders	CList	The objects this collaborator depends upon.
itsDependents	CList	The objects that depend upon this collaborator.

### Methods

#### Creation and destruction

##### ICollaborator

```
procedure ICollaborator;
void ICollaborator (void);
```

Initialize a collaborator. This method sets both itsProviders and itsDependents to NIL.

##### Free/Dispose

```
procedure Free;
void Dispose (void);
```

Remove this object from each of its provider's list of dependents and from each of its dependent's list of providers. Then dispose of its own lists of providers and dependents.



<b>DependUpon</b>	<b>Creating dependency method</b> <pre>procedure DependUpon (aProvider: CCollaborator); void DependUpon (CCollaborator *aProvider);</pre> <p>This collaborator depends on aProvider. Add this collaborator to aProvider's list of dependents, and add aProvider to this collaborator's list of providers. If either of those lists are NIL, this method creates them before adding to them.</p>
<b>CancelDependency</b>	<pre>procedure CancelDependency (aProvider: CCollaborator); void CancelDependency (CCollaborator *aProvidoer);</pre> <p>This collaborator no longer depends upon aProvider. Remove this collaborator from the provider's dependent list and remove the provider from this collaborator's provider list.</p>
<b>BroadcastChange</b>	<b>Change notification methods</b> <pre>procedure BroadcastChange (reason: long; info: Ptr); void BroadcastChange (long reason, void* info);</pre> <p>Notify this object's dependents that this object has changed. Reason is an integer that describes the type of change. You must define reasons for your subclass. Info is a pointer to any additional information needed to respond to the change.</p> <p>If you want objects that aren't in a collaborator's list of dependents to know about a change, override this method. For example, CBureaucrat overrides this method to notify the bureaucrat's supervisor, in addition to its dependents, of the change.</p>
<b>ProviderChanged</b>	<pre>procedure ProviderChanged (aProvider: CCollaborator;     reason: longint; info: Ptr); void ProviderChanged (CCollaborator *aProvider,     long reason, void* info);</pre> <p>One of this object's providers has just changed. Your subclass must override this method to respond to each reason it can handle. The default method does nothing. AProvider is the provider that changed. Reason is an integer that describes the type of change. You must define reasons for your subclass. Info is a pointer to any additional information that your subclass might need.</p> <b>List update methods</b> <p>CCollaborator uses these methods internally to maintain the lists of dependents and providers. Whenever possible, use the DependUpon method instead of these methods. When a dependent is added or removed,</p>

## ◆ 23 CCollaborator

---

DependUpon and update *both* the dependent's list of providers *and* the provider's list of dependents. They do the same when a provider is added or removed. If you use the functions below, you might damage the collaborators' lists so that they don't match.

### **AddDependent**

```
procedure AddDependent (aDependent: CCollaborator);  
void AddDependent (CCollaborator *aDependent);  
Add aDependent to this collaborator's list of dependents.
```

### **RemoveDependent**

```
procedure RemoveDependent (aDependent: CCollaborator);  
void RemoveDependent (CCollaborator *aDependent);  
Remove aDependent from this collaborator's list of dependents.
```

### **AddProvider**

```
procedure AddProvider (aProvider: CCollaborator);  
void AddProvider (CCollaborator *aProvider);  
Add aProvider to this collaborator's list of providers.
```

### **RemoveProvider**

```
procedure RemoveProvider (aProvider: CCollaborator);  
void RemoveProvider (CCollaborator *aProvider);  
Remove aProvider from this collaborator's list of providers.
```

# CCollection

---

## 24

### Introduction

CCollection is an abstract class for implementing collections of things.

### Heritage

Superclass	CCollaborator
Subclasses	CArray

### Using CCollection

CCollection is an abstract class that helps you implement collections of things. The class doesn't provide the implementation of the collection. In your subclass, you specify the data structures that implement the list.

### Variables

The only instance variable in this abstract class is the number of items in the collection. Your subclass will need to add at least the instance variable that contains or points to the memory that contains the items in your collection.

Variable	Type	Description
numItems	longint	The number of items in the collection.

### Methods

#### ICollection

```
procedure ICollection;  
void ICollection (void);
```

Initialize the collection. The default method sets the initial number of items to 0.

#### GetNumItems

```
function GetNumItems: longint;  
long GetNumItems (void);
```

Return the number of items in a collection

## ◆ 24 CCollection

---

**IsEmpty**

```
function IsEmpty: Boolean;  
Boolean IsEmpty (void);  
Return TRUE if the collection has no items.
```

# CControl

---

## 25

### Introduction

CControl is an abstract class for implementing Macintosh controls. The two built-in descendant classes, CButton and CScrollBar, implement the standard Macintosh controls.

### Heritage

Superclass	CPane
Subclasses	CButton CScrollBar

### Using CControl

This class describes the common methods for all controls. Scroll bars, buttons, check boxes, and radio buttons are descendants of CControl. Although you won't be creating subclasses of CControl (unless you define your own kind of control) you should become familiar with these methods.

### Variables

The only instance variable for this class is a handle to the actual Macintosh control.

Variable	Type	Description
macControl	ControlHandle	Handle to the Macintosh control.

### Methods

You can use these methods with any kind of control, but some methods don't really apply to all controls. For example, it's possible to set the value of a push button or to give a title to a scroll bar, but in most cases these actions don't make sense.

### Construction and destruction methods

The CControl class does not define an initialization method. Each descendant of CControl defines an initialization method that uses CNTL resources to specify the type of control.

#### Free/Dispose

```
procedure Free;  
void Dispose (void);  
Dispose of a control.
```

### Accessing methods

#### SetValue

```
procedure SetValue (aValue: integer);  
void SetValue (short aValue);  
Set the value of a control. This method sends a BroadcastChange message to all of this object's dependents with controlValueChanged as the reason.
```

#### GetValue

```
function GetValue: integer;  
short GetValue (void);  
Get the value of a control
```

#### SetMaxValue

```
procedure SetMaxValue (aMaxValue: integer);  
void SetMaxValue (short aMaxValue);  
Set the maximum value a control can have. For push buttons, check boxes, and radio buttons this value should be 1.
```

#### GetMaxValue

```
function GetMaxValue: integer;  
short GetMaxValue (void);  
Get the maximum value a control can take on.
```

#### SetMinValue

```
procedure SetMinValue (aMinValue: integer);  
void SetMinValue (short aMinValue);  
Set the minimum value a control can have. Note that for push buttons, check boxes, and radio buttons this value should be 0.
```

#### GetMinValue

```
function GetMinValue: integer;  
short GetMinValue (void);  
Get the minimum value a control can take on.
```



**SetTitle**

```
procedure SetTitle (aTitle: Str255);  
void SetTitle (Str255 aTitle);
```

Set the title of a control. Scroll bars can have titles, but the title is not displayed.

**GetTitle**

```
procedure GetTitle (var aTitle: Str255);  
void GetTitle (Str255 aTitle);
```

Get the title of a control

**SetActionProc**

```
procedure SetActionProc (anActionProc: ProcPtr);  
void SetActionProc (VoidFunc anActionProc);
```

Set the action proc of a control. The action proc is called repeatedly while the mouse is held down in a control. The control manager distinguishes between an action proc called when the mouse goes down in a moving indicator (like a scroll box) and one called when the mouse goes down in a stationary part of a control. The default DoClick method sends the action proc to the control manager only for mouse hits that are not in a moving indicator. In THINK Pascal, you must declare the action proc like this:

```
procedure MyAction (macControl: ControlHandle;  
                   whichPart: integer);
```

In THINK C, you must declare the action proc like this:

```
pascal void MyAction (ControlHandle macControl,  
                     short whichPart);
```

If you want an action proc called when the mouse is in a moving indicator, you'll need to override the DoClick method.

---

**Note**

Be sure you understand action procs and the control manager before you use this method.

---

**Appearance methods**

These methods control the appearance of controls on the screen. They hide and show the control, draw it, move it, and change its size.

**Hide**

```
procedure Hide;  
void Hide (void);
```

Hide the control.

## ◆ 25 CControl

---

<b>Show</b>	<pre>procedure Show; void Show (void);</pre> <p>Show the control.</p>
<b>Activate</b>	<pre>procedure Activate; void Activate (void);</pre> <p>Make the control active. The control gets an Activate message automatically when its enclosure gets an Activate message.</p>
<b>Deactivate</b>	<pre>procedure Deactivate; void Deactivate (void);</pre> <p>Make the control inactive. The control gets a Deactivate message automatically when its enclosure gets a Deactivate message.</p>
<b>Offset</b>	<pre>procedure Offset (hOffset, vOffset: longint; redraw: Boolean); void Offset (long hOffset, long vOffset, Boolean redraw);</pre> <p>Move the control by hOffset, vOffset pixels. If redraw is TRUE, redraw the control after the move.</p>
<b>ChangeSize</b>	<pre>procedure ChangeSize (delta: Rect; redraw: Boolean); void ChangeSize (Rect *delta, Boolean redraw);</pre> <p>Change the size of a control. Each component of the rectangle specifies by how many pixels to offset each point. Positive numbers mean down and to the right. Negative numbers mean up and to the left. If redraw is TRUE, redraw the control after the change.</p>
<b>Draw</b>	<pre>procedure Draw (var area: Rect); void Draw (Rect *area);</pre> <p>Draw the control. The area parameter is ignored.</p>
<b>DrawAll</b>	<pre>procedure DrawAll (var area: Rect); void DrawAll (Rect *area);</pre> <p>Draw the control and all its subviews.</p>
<b>Prepare</b>	<pre>procedure Prepare; void Prepare (void);</pre> <p>Prepare the port and the coordinate system before drawing. Generally, you don't need to use or override this method.</p>



### Click response methods

The CControl class takes care of all the mouse tracking within a control.

#### DoClick

```
procedure DoClick (hitPt: Point;
  modifierKeys: integer; when: longint);
void DoClick (Point hitPt, short modifierKeys,
  long when);
```

Handle a click in a control. HitPt is the point in frame coordinates. ModifierKeys is the same as the modifier field of an event record. When is the time that the mouse went down in ticks.

The default method handles both simple controls and controls with moving indicators and behaves a little differently in each case.

If the mouse goes down in a part other than a moving indicator:

- The default method calls the Toolbox routine `TrackControl` with the action proc you specified with the `SetActionProc` method.
- If `TrackControl` returns `TRUE` (the user clicks and releases the mouse in the same part of the control), this method sends a `DoGoodClick` message to the control.

If the mouse goes down in a moving indicator, like the thumb of a scroll bar:

- The default method calls the Toolbox routine `TrackControl` with no action proc.
- If the value of the control has changed (the user moved the indicator to a new place), this method sends a `DoThumbDragged` message to the control.

#### DoThumbDragged

```
procedure DoThumbDragged (delta: integer);
void DoThumbDragged (short delta);
```

An indicator in a control has been moved. The `DoClick` method sends a `DoThumbDragged` message to the control when the user changes the position of a control's indicator.

The default method does nothing. Controls with indicators should override this method. For an example of how to override this method, see `CScrollbar.DoThumbDragged` on page 385.

## ◆ 25 CControl

---

### **DoGoodClick**

```
procedure DoGoodClick (whichPart: integer);
```

```
void DoGoodClick (short whichPart);
```

The mouse went down and up in the same part of a control.

The default method does nothing. Subclasses should override this method. See the `CButton.DoGoodClick` on page 195 for an example.

# *CDataFile* ◆

## 26

---

### Introduction

This class implements the methods you need to work with a Macintosh data file. You can use this class without creating a subclass if you need to read raw bytes. If you require a structured file, you should create a subclass of CDataFile.

### Heritage

Superclass	CFile
Subclasses	CPNTGFile CPictFile

### Using CDataFile

You can use this class without creating a subclass to read and write the data fork of a Macintosh file. After creating an instance of CDataFile, use one of the specification methods inherited from CFile to indicate which file you're working with. Then use the Open method to open the file, and use the reading and writing methods to read and write the file.

Some CDataFile methods use the THINK Class Library's exception handling library if something goes wrong in a file operation. For more information on exception handling, see Chapter 8, "Exception Handling."

---

#### Note

This version of CDataFile handles errors differently than the original version. The old version is now named ODataFile and is in the Compatibility folder.

---

### Variables

The only instance variable for this class is the refNum of the file. The Macintosh file system uses this number to identify the volume, directory, and file

on disk. To learn more about the Macintosh file system, see *Inside Macintosh IV*, Chapter 19, “The File Manager.”

Variable	Type	Description
refNum	integer	The file's refNum when open.

### Methods

These methods let you open and close data files, read and write data, and get information about a file. You need to use the methods inherited from CFile to specify which file you want to work with.

#### Construction and destruction methods

##### IDataFile

```
procedure IDataFile;
void IDataFile (void);
```

Initialize the data file.

#### Accessing methods

##### SetLength

```
procedure SetLength (aLength: longint);
void SetLength (long aLength);
```

Set the end-of-file marker for this file at aLength.

##### GetLength

```
function GetLength: longint;
long GetLength (void);
```

Return the length of the file. The file must be open already.

##### SetMark

```
procedure SetMark (howFar: longint;
    fromWhere: integer);
void SetMark (long howFar, short fromWhere);
```

Specify where subsequent read/write operations will take place. This position is called “the mark.”

HowFar specifies how far in bytes from the fromWhere parameter to set the mark. Positive values are offsets toward the end of the file. Negative values are offsets toward the beginning of the file. FromWhere is one of the following:

fromWhere value	Meaning
fsFromStart	from the beginning of the file
fsFromLEOF	from the end of the file
fsFromMark	from the current position

**GetMark**

```
function GetMark: longint;
long GetMark (void);
```

Return the current position of the mark in markPos.

**Open****Open and close methods**

```
procedure Open (permission: SignedByte);
void Open (SignedByte permission);
```

Open the file with the given permission. Permission can be one of:

<b>Permission</b>	<b>Meaning</b>
fsCurPerm	same as the current permission
fsRdPerm	read permission
fsWrPerm	write permission
fsRdWrPerm	read/write permission
fsRdWrShPerm	shared read/write permission

See *Inside Macintosh IV*, Chapter 19, "The File Manager" for a discussion of permissions.

**Close**

```
procedure Close;
void Close (void);
```

Write out any unwritten data and close the file.

**Read and write methods****ReadAll**

```
function ReadAll: Handle;
Handle ReadAll (void);
```

Read the entire contents of the file into a handle. This method allocates the handle. You must pass in a reference to a handle. This example shows how to use ReadAll in THINK Pascal:

```
var
  theData: Handle;
begin
  ...
  myDataFile.ReadAll(theData);
  ...
end
```

And this example shows how to use ReadAll in THINK C:

```
Handle theData;
...
myDataFile->ReadAll(&theData);
```

## ◆ 26 CDataFile

---

### ReadSome

```
procedure ReadSome (info: Ptr; var howMuch: longint);  
void ReadSome (Ptr info, long howMuch);
```

Read howMuch bytes into a buffer starting at info. You must allocate the space to read the information into. This example shows how to use ReadSome in THINK Pascal::

```
var  
    myBuffer = packed array [1..128] of char;  
begin  
    ...  
    myDataFile.ReadSome(@myBuffer, 128);  
    ...  
end
```

And this example shows how to use ReadSome in THINK C:

```
char myBuffer[128];  
...  
myDataFile->ReadSome(myBuffer, 128L);
```

### WriteAll

```
procedure WriteAll (contents: Handle);
```

```
void WriteAll (Handle contents);
```

Write the contents of the handle to the file.

### WriteSome

```
procedure WriteSome (info: Ptr; var howMuch: longint);
```

```
void WriteSome (Ptr info, long howMuch);
```

Write howMuch bytes from the buffer starting at info.



# CDecorator

## 27

### Introduction

CDecorator implements an object that arranges windows on the screen. There is only one instance of this class.

### Heritage

Superclass	CObject
Subclasses	None

### Using CDecorator

CDecorator gives you methods for arranging windows on the screen. There is only one instance of CDecorator which is stored in the global variable `gDecorator`.

After creating and initializing a new window, send it in a `PlaceNewWindow` message to the decorator. The decorator makes the window a bit smaller than the main screen and offsets it down and to the right. Using the decorator is entirely optional.

You can create a subclass of CDecorator if you like. For instance, you might want a decorator that implements window tiling. Be sure to override the `MakeDecorator` method in your application subclass to initialize your decorator and to store the decorator in the global variable `gDecorator`.

### Variables

The global decorator object, created in the application method `MakeDecorator`, is stored in the global variable `gDecorator`.

#### Global variable

Variable	Type	Description
<code>gDecorator</code>	<code>CDecorator</code>	The window-dressing object.

### Instance variables

Variable	Type	Description
wCount	integer	The number of new windows placed
index	integer	Index for offsetting windows
wWidth	integer	Width of a window in pixels
wHeight	integer	Height of a window in pixels
hLocation	integer	Horizontal location for next window
vLocation	integer	Vertical location for next window

### Methods

<b>IDecorator</b>	<pre>procedure IDecorator; void IDecorator (void);</pre> <p>Initialize the decorator. The application method <code>MakeDecorator</code> creates the global decorator and sends it this message.</p>
<b>PlaceNewWindow</b>	<pre>procedure PlaceNewWindow (theWindow: CWindow); void PlaceNewWindow (CWindow *theWindow);</pre> <p>Place <code>theWindow</code> on the screen. The default method makes it fill up most of the screen and calls <code>StaggerWindow</code> to position it. .</p>
<b>StaggerWindow</b>	<pre>procedure StaggerWindow (theWindow: CWindow); void StaggerWindow (CWindow *theWindow);</pre> <p>Position a new window on the screen, offset down and to the right of the last window the decorator placed.</p>
<b>CenterWindow</b>	<pre>procedure CenterWindow (theWindow: CWindow ); void CenterWindow (CWindow *theWindow);</pre> <p>Center the window specified by <code>theWindow</code> on the main screen. If <code>theWindow</code> is a modal dialog, this method tries to put it in the top third of the main screen. This method does not increment <code>wCount</code> or change any of the instance variables.</p>
<b>GetWCount</b>	<pre>function GetWCount: integer; short GetWCount (void);</pre> <p>Return the number of windows the decorator has put on the screen. You can use this value to give your untitled windows sequential numbers like "Untitled-1."</p>

# CDesktop ♦

## 28

### Introduction

CDesktop implements a view that occupies the entire screen. This view serves as the top of the visual hierarchy and the enclosure for all windows. Normally, there is only one instance of CDesktop.

### Heritage

Superclass	CView
Subclasses	CFWDesktop

### Using CDesktop

The most common way you'll use the desktop is as the enclosure for your windows. Your application should not send messages to the desktop. Instead, you should rely on higher level objects like windows and directors to send messages to it.

CFWDesktop is a subclass of CDesktop which implements a desktop that supports floating windows. CFWDesktop is described on page 289.

The desktop is stored in the global variable `gDesktop`. The `CApplication` method `MakeDesktop` creates an instance of the `CDesktop` and stores it in that variable.

### Variables

The global variable `gDesktop` holds a pointer to the single instance of the desktop. Use this variable to specify the enclosure for your application's windows.

#### Global variable

Variable	Type	Desktop
<code>gDesktop</code>	<code>CDesktop</code>	The single instance of the desktop

### Instance variables

Variable	Type	Desktop
bounds	Rect	Boundaries of the desktop
itsWindows	CList	List of windows
topWindow	CWindow	Top-most application window

### Methods

#### Construction and destruction methods

#### IDesktop

```
procedure IDesktop (aSupervisor: CBureaucrat);
```

```
void IDesktop (CBureaucrat *aSupervisor);
```

Initialize the desktop. The default method opens a GrafPort that your application uses as its desktop. The desktop's supervisor should be the application, stored in the global variable gApplication.

#### Free/Dispose

```
procedure Free;
```

```
void Dispose (void);
```

Dispose of the desktop and send this message to all of the desktop's windows.

#### Appearance methods

#### Show

```
procedure Show;
```

```
void Show (void);
```

Makes a desktop visible by showing all of its windows. This method sends a Show message to all of the desktop's windows.

#### Hide

```
procedure Hide;
```

```
void Hide (void);
```

Hide a desktop by hiding all of its windows. This method sends a Hide message to all of the desktop's windows.

#### Activate

```
procedure Activate;
```

```
void Activate (void);
```

Activate the desktop by activating the top window. This method sends an Activate message to the desktop's top window.

#### Deactivate

```
procedure Deactivate;
```

```
void Deactivate (void);
```

Deactivate a desktop by deactivating its top window. This method sends a Deactivate message to the desktop's top window.

**ReallyVisible**

```
function ReallyVisible: Boolean;
Boolean ReallyVisible (void);
Return TRUE if the desktop is visible.
```

**DispatchClick****Mouse methods**

```
procedure DispatchClick (var macEvent: EventRecord);
void DispatchClick (EventRecord *macEvent);
```

If the user clicks the mouse anywhere, find out where the mouse went down, and send the appropriate message to the object the click is intended for. If the top window is modal, and the mouse did not go down in the menu bar, this method beeps and does not process the click.

This method uses what the Macintosh Toolbox routine FindWindow returns to determine what message to send to which object.

Part	Action
inDesk	If the mouse goes down in the desktop, this method sends a DoClick message to the desktop. The default DoClick method does nothing.
inSysWindow	If the mouse goes down in a system window, this method calls the Toolbox routine SystemClick.
inMenuBar	If the mouse goes down in the menu bar, this method calls the Toolbox routine MenuSelect. This method then sends a FindCmdNumber message to the bartender to convert the menu selection into a command number. This command number is sent to the gopher in a DoCommand message.
inContent	If the mouse goes down in the content region of a window and the window is not active, this method sends the window a Select message. If the window is already active and can receive clicks, this method sends the window a DispatchClick message which will

	eventually send a DoClick message to a pane or the window itself.
inDrag	If the mouse goes down in the drag region (the title bar), this method sends the window a Drag message.
inGrow	If the mouse goes down in the window's grow region (the lower right corner), this method sends the window a Resize message.
inGoAway	If the mouse goes down and up in the window's close box, this method sends it a Close message.
inZoomIn, inZoomOut	If the mouse goes down and up in the window's zoom box, this method sends it a Zoom message.

<b>DoMouseUp</b>	<pre>procedure DoMouseUp (macEvent: EventRecord); void DoMouseUp (EventRecord *macEvent);</pre> Handle a mouse up event in the desktop. The default method does nothing.
<b>DispatchCursor</b>	<pre>procedure DispatchCursor (where: Point;     mouseRgn: RgnHandle); void DispatchCursor (Point where, RgnHandle mouseRgn);</pre> Set the cursor for the view the cursor is in. If the cursor is over an active window, this method sends a DispatchCursor message to the window. If the cursor is in the menu bar, this method sets it to the arrow. If the cursor is not in the menu bar or in an active window, this method sends an AdjustCursor message to the desktop. You should not override this method.
<b>AdjustCursor</b>	<pre>procedure AdjustCursor (where: Point;     mouseRgn: RgnHandle); void AdjustCursor (Point where, RgnHandle mouseRgn);</pre> Adjust the cursor shape when the mouse is in an insignificant part of the desktop. The default method sets the cursor to an arrow.
<b>Contains</b>	<pre>function Contains (thePoint: Point): Boolean; Boolean Contains (Point thePoint);</pre> Return TRUE if thePoint is within the bounds of the desktop.



**HitSamePart**                    `function HitSamePart (var pointA, pointB: Point): Boolean;`  
                                 `Boolean HitSamePart (Point pointA, Point pointB);`  
Return TRUE if pointA and pointB are reasonably close together. "Reasonably close" means that the two points are within REASONABLY\_CLOSE pixels.

### **Window methods**

You do not have to use any of these methods yourself. Directors, objects that manage the interaction between windows and the desktop, take care of sending most of these messages.

**AddWind**                      `procedure AddWind (theWindow: CWindow);`  
                                 `void AddWind (CWindow *theWindow);`  
Add theWindow to the desktop's window list.

**RemoveWind**                  `procedure RemoveWind (theWindow: CWindow);`  
                                 `void RemoveWind (CWindow *theWindow);`  
Remove theWindow from the desktop's window list.

**SelectWind**                   `procedure SelectWind (theWindow: CWindow);`  
                                 `void SelectWind (CWindow *theWindow);`  
Select theWindow and bring theWindow to the front.

**ShowWind**                    `procedure ShowWind (theWindow: CWindow);`  
                                 `void ShowWind (CWindow *theWindow);`  
Make theWindow visible on the desktop.

**HideWind**                    `procedure HideWind (theWindow: CWindow);`  
                                 `void HideWind (CWindow *theWindow);`  
Hide theWindow.

**DragWind**                    `procedure DragWind (theWindow: CWindow;`  
                                 `macEvent: EventRecord);`  
                                 `void DragWind (CWindow *theWindow,`  
                                 `EventRecord *macEvent);`  
Drag theWindow on the desktop.

**UpdateWindows**              `procedure UpdateWindows;`  
                                 `void UpdateWindows (void);`  
Send an Update message to each of the desktop's windows.

## ◆ 28 CDesktop

---

### Accessing methods

#### GetTopWindow

```
function GetTopWindow: CWindow;  
CWindow* GetTopWindow (void);  
Get the frontmost window.
```

#### GetBounds

```
procedure GetBounds (theBounds: Rect);  
void GetBounds (Rect *theBounds);  
Get the bounds of the desktop.
```

#### GetAperture

```
procedure GetAperture (var theAperture: LongRect);  
void GetAperture (LongRect *theAperture);  
Get the visible portion of the desktop. This method returns the bounds of  
the desktop since all of the desktop is always visible.
```

### Calibration methods

#### Prepare

```
procedure Prepare;  
void Prepare (void);  
Set the port to the desktop port. You should not use or override this method.
```

### Cleanup method

#### Cleanup

```
procedure Cleanup;  
This method is in THINK Pascal only. This method does nothing, but  
CFWDesktop overrides it to move THINK Pascal windows to the back. For  
more information see CFWDesktop.Cleanup on page 292. THINK C  
doesn't use this method since CFWDesktop doesn't need to move THINK C  
windows to the back.
```



# CDirector 29 ◆

---

## Introduction

CDirector is an abstract class for implementing a window that can handle commands. Directors implement communication between the application and a window.

## Heritage

Superclass	CDirectorOwner
Subclasses	CDocument CClipboard CTearOffMenu

## Using CDirector

A director is an abstract class that manages the communication between the application and a window. Any time you want to display a window, it must be related to a director.

In most cases, you'll use the CDocument subclass to display and manipulate data stored in a file in a window. The only time you'll create a subclass of CDirector is when you need a special kind of window like a status window, a subwindow, or a tear-off menu.

---

### Note

The class CTearOffMenu implements a tear-off menu as a subclass of CDirector.

---

When a window belonging to a director becomes the active window, the gopher points to the bureaucrat specified by the director's `itsGopher` instance variable. When the window becomes inactive, the gopher points to the application.

When the switchboard processes a window-related event (such as an update event), it sends the message to the window which in turn sends the message

to the director. When the user chooses a command from the menu, the switchboard sends the `DoCommand` message to the gopher.

The supervisor of a director must be the application. The supervisor of a window must be a subclass of `CDirector`.

### Variables

#### Global variable

The global variable `gGopher` points to the current bureaucrat which is usually a descendant of `CDirector`. When a window becomes active, `gGopher` points to the bureaucrat specified in the director's `itsGopher` instance variable. When there are no active directors, `gGopher` points to the application.

Variable	Type	Description
<code>gGopher</code>	<code>CBureaucrat</code>	The current bureaucrat.

#### Instance variables

The instance variable `itsGopher` usually points to the main pane of a window. When the director becomes active, the bureaucrat specified in `itsGopher` becomes the gopher.

Variable	Type	Description
<code>itsWindow</code>	<code>CWindow</code>	Window that the director controls
<code>active</code>	<code>Boolean</code>	TRUE if the director is active
<code>itsGopher</code>	<code>CBureaucrat</code>	Bureaucrat to make the gopher when the director is activated.
<code>activateWindowOnResume</code>	<code>Boolean</code>	TRUE, if the director activates its window when the program is resumed

### Methods

#### Creation and destruction

##### IDirector

```
procedure IDirector (aSupervisor: CDirectorOwner);
void IDirector (CDirectorOwner *aSupervisor);
```

Initialize the director. This method adds the director to `aSupervisor`'s list of directors. By default, the director has no window and is not active. The



itsGopher instance variable is set to this object. Your director subclass must create the window for the director. Your director subclass should call the inherited method to initialize the director.

### Free/Dispose

```
procedure Free;
```

```
void Dispose (void);
```

Dispose of the director. This method sends a **Free** or **Dispose** message to the director's window (if it has one) and removes the director from the supervisor's list of directors. Your subclass must call the inherited method to make sure that the director is disposed of properly.

### Accessing method

### GetWindow

```
function GetWindow: CWindow;
```

```
CWindow* GetWindow (void);
```

Return the window that the director controls.

### FindViewByID

```
function FindViewByID (aViewID: longint): CView;
```

```
CView* FindViewByID (long aViewID);
```

Within the director's window, locate the view with id aViewID.

### OwnsWindow

```
function OwnsWindow (aWindow: CWindow): Boolean;
```

```
Boolean OwnsWindow (CWindow *aWindow);
```

Return TRUE if this director owns aWindow. The director owns the window if the window's supervisor is this object.

### Command methods

### DoCommand

```
procedure DoCommand (theCommand: longint);
```

```
void DoCommand (long theCommand);
```

Handle a command. The default method handles this command:

Command	Description
cmdClose	Send a <b>Close (FALSE)</b> message to the director

Your director class will usually override this method. You should handle your own commands first, then call the inherited method to get the generic effects.

### UpdateMenus

```
procedure UpdateMenus;  
void UpdateMenus (void);
```

If a window is associated with the director, this method enables the **Close** command (cmdClose).

### Appearance methods

### Activate

```
procedure Activate;  
void Activate (void);
```

A director is becoming active. If the director's window is not a floating window, sends a BecomeGopher (TRUE) message to itsGopher. This method sets gSleepTime to 0 to force an idle event and sends an ActivateDirector message to the director's supervisor.

### Deactivate

```
procedure Deactivate;  
void Deactivate (void);
```

A director is becoming inactive. If the director is the gopher, send a BecomeGopher (TRUE) message to the director's supervisor. This method sends a DeactivateDirector message to the director's supervisor.

### ActivateDirector

```
procedure ActivateDirector (aDirector: CDirector);  
void ActivateDirector (CDirector *aDirector);
```

A director owned by this director is becoming active. This method calls the inherited method to make aDirector the frontmost director and sends an ActivateDirector message to this director's supervisor.

### DeactivateDirector

```
procedure DeactivateDirector (aDirector: CDirector);  
void DeactivateDirector (CDirector *aDirector);
```

A director owned by this director is becoming inactive. This method calls the inherited method and sends a DeactivateDirector message to this director's supervisor.

### Suspend

```
procedure Suspend;  
void Suspend (void);
```

The application is being suspended. This method calls the inherited method to send a Suspend message to all of the supervisor's directors. If this director is active, and if it owns an active window, it sends the window a Deactivate message. The director remains active even though the application is suspended.

**Resume**

```
procedure Resume;  
void Resume (void);
```

The application is being resumed. This method calls the inherited method to send a `Resume` method to all of the supervisor's directors. If the director is active and the director owns a window, send the window an `Activate` message.

The default method sends the director an `Activate` message.

**Close**

```
function Close (quitting: Boolean): Boolean;  
Boolean Close (Boolean quitting);
```

Close a director. This method calls the inherited method to close all of the directors that the supervisor owns (including this one). If all of the directors closed, this method calls `Free` or `Dispose` and returns `TRUE`, otherwise it returns `FALSE`.

**Window methods****CloseWind**

```
procedure CloseWind (theWindow: CWindow);  
void CloseWind (CWindow *theWindow);
```

This method calls `Close (FALSE)`. If `theWindow` is not the window owned by this director, it sends a `Dispose` or a `Free` message to `theWindow`.

If you want a click in the window's close box to mean something other than closing the director, override this method. For instance, you might want to override this method so it hides the window instead of closing it. See `CClipboard`'s `CloseWind` method on page 212 for an example.

**ActivateWind**

```
procedure ActivateWind (theWindow: CWindow);  
void ActivateWind (CWindow *theWindow);
```

The window owned by the director has been activated. You generally won't need to use or override this method in your director subclasses unless you want to do something to `theWindow` (and not the director) when it becomes active. To perform specific actions at activate time, override the `Activate` method instead.

**DeactivateWind**

```
procedure DeactivateWind (theWindow: CWindow);  
void DeactivateWind (CWindow *theWindow);
```

The window owned by the director is becoming inactive. You generally won't need to use or override this method in your director subclasses unless you want to do something to `theWindow` (and not the director) when it becomes deactivated. To perform some specific actions at deactivate time, override the `Deactivate` method instead.

### IsActive

```
function IsActive: Boolean;  
Boolean IsActive (void);
```

Returns TRUE if the director is active; FALSE otherwise. You should not override this method.

### Change notification method

This method overrides a method in CCollaborator. For more information, see CCollaborator on page 223.

### ProviderChanged

```
procedure ProviderChanged (aProvider: CCollaborator;  
    reason: longint; info: Ptr);  
void ProviderChanged (CCollaborator *aProvider,  
    long reason, void* info);
```

One of this director's providers or subordinates has just changed. This method handles the reason `bureaucratIsGopher` by updating the instance variable `itsGopher`. If your subclass handles other reasons, it should override this method and call the inherited method.

`AProvider` is the provider or subordinate that changed. In this case it is the bureaucrat that is becoming the gopher. `Reason` is an integer that describes the type of change. In this case, `bureaucratIsGopher` is the only reason handled. `Info` is a pointer to any additional information that a subclass might need. This method does not use `info`, and passes it along in a call to the inherited method.

# ***CDirectorOwner*** **30**

---

## **Introduction**

CDirectorOwner is an abstract class for objects that own directors, which are objects that manage windows.

## **Heritage**

Superclass	CBureaucrat
Subclasses	CApplication
	CDirector

## **Using CDirectorOwner**

CDirectorOwner is an abstract class for objects that own directors. A director is an object that manages the communication between an application and a window.

CDirectorOwner has two subclasses: CApplication and CDirector. An application needs to inform its directors when the application is suspending, resuming, or quitting. If your application implements multi-window documents, use an object of class CDocument as the main document and objects of class CDirector as the subwindows. The subwindows should be owned by the document object.

### Variables

Variable	Type	Description
itsDirectors	CList	List of the directors this object owns.
active	Boolean	TRUE if any director is active.

### Methods

#### Creation and destruction

**IDirectorOwner** procedure IDirectorOwner (aSupervisor: CDirectorOwner);  
 void IDirectorOwner (CDirectorOwner \*aSupervisor);  
 Initialize the owner. This method sets itsDirectors to NULL.

**Dispose/Free** procedure Free;  
 void Dispose (void);  
 Dispose of this object and all the directors it owns.

#### Insertion and deletion methods

**AddDirector** procedure AddDirector (aDirector: CDirector);  
 void AddDirector (CDirector \*aDirector);  
 Add a new director to the director list.

**RemoveDirector** procedure RemoveDirector (aDirector: CDirector);  
 void RemoveDirector (CDirector \*aDirector);  
 Remove a director from the director list.

#### Appearance methods

**ActivateDirector** procedure ActivateDirector (aDirector: CDirector);  
 void ActivateDirector (CDirector \*aDirector);  
 A director that this object owns has been activated. Sets active to TRUE.

**DeactivateDirector** procedure DeactivateDirector (aDirector: CDirector);  
 void DeactivateDirector (CDirector \*aDirector);  
 A director that this object owns has been deactivated. Sets active to FALSE.

**Suspend** procedure Suspend;  
 void Suspend (void);  
 Notify this object's directors that the application is suspending. Send a Suspend message to all directors.



**Resume**

```
procedure Resume;
```

```
void Resume (void);
```

Notify this object's directors that the application is resuming. Send a Resume message to all the directors.

**Quit**

```
function Quit: Boolean;
```

```
Boolean Quit (void);
```

The application is about to quit. This method is the same as calling Close (TRUE).

**Close**

```
function Close (fQuitting: Boolean): Boolean;
```

```
Boolean Close (Boolean fQuitting);
```

Try to close all the directors that this object owns. If they all close, this method returns TRUE. If any director doesn't close, this method stops closing directors and returns FALSE. Set fQuitting to TRUE if you are quitting the application.



# CDocument

---

## Introduction

CDocument is the main class for presenting and manipulating information. You can think of a document as the association of a window, a file, and a set of panes.

Your application must create a subclass of CDocument.

## Heritage

Superclass  
Subclasses

CDirector  
You must create a subclass of CDocument

## Using CDocument

CDocument is one of the classes you need to override to implement an application in the THINK Class Library. You can think of a document as a file that you view through a window. A better way to think about a document is that it is the essence of a Macintosh application. It is anything that you can display and manipulate inside a window.

The document is where your application draws and displays its data. Since documents are descendants of CDirector, all documents have windows associated with them. Most documents also have an associated file. Neither the window nor the file are created automatically. You must create them yourself in your document's initialization method.

Your document class should override these methods:

<i>initialization method</i>	OpenFile
Free	DoSave
DoCommand	DoSaveAs
NewFile	Revert

Your document class must have an initialization method. If your subclass defines new instance variables, this is the method that sets them up. By con-

## ◆ 31 CDocument

---

vention, the name of your initialization method should be `IYourDoc`, where *YourDoc* is the name of your document class. Your initialization method should call `IDocument`. The supervisor of a document is always `gApplication`.

If your application allocates memory, you should also override the `Free` or `Dispose` method to deallocate it. Be sure that your method calls the inherited method to make sure that the document is disposed of properly.

---

### Note

You do not need to dispose of the window or file associated with a document. The default `Free` or `Dispose` method does that for you.

---

Your document class's `DoCommand` method does most of the work in your application. When a window is active, the switchboard sends all commands to the document first (it's the gopher), and if the document can't handle it, the application class tries to handle it. Your document class should handle all the commands it knows about, and call the inherited method when it can't.

---

### Note

Be sure that your `DoCommand` method or that one of the methods it invokes sets the instance variable `dirty` to `TRUE` when there has been a change to the document.

---

Your document class gets a `NewFile` message when you choose **New** from the **File** menu. This method needs to create a window and attach the panes for it. The `NewFile` method doesn't need to create a file until you try to save the document.

Your document gets an `OpenFile` message when you choose **Open...** from the **File** menu. The `OpenFile` method has one argument: a Macintosh `SFReply` record. When you get the `OpenFile` message, you can be sure that the `SFReply` record is properly filled in. Your `OpenFile` message needs to create an instance of a file object (usually of class `CDataFile`). You can send your file any of several read messages to get its contents. Your `OpenFile` method also needs to create a window to display the contents of the file, just as your `NewFile` method does.

When the you choose **Save** from the **File** menu, your document gets a `DoSave` message. Your `DoSave` method should write the contents of its file to disk. The file object is stored in the instance variable `itsFile`.



When the you choose **Save As...** from the **File** menu, your document gets a `DoSaveAs` message. This method takes an `SFReply` record, and you can be sure that it is properly filled in. Your document class needs to override this method to write its data to a file.

If your application supports the **Revert** command, you should implement `DoRevert` method. Your implementation might close the file without saving, then open the file again.

## Variables

### Global variables

The global variable `gGopher` points to the current bureaucrat which is usually a document. When a window becomes active, `gGopher` points to the document that owns it. When there are no active documents, `gGopher` points to the application.

The supervisor of every document is the application.

Variable	Type	Description
<code>gGopher</code>	<code>CBureaucrat</code>	The current bureaucrat.
<code>gApplication</code>	<code>CApplication</code>	The application

### Instance variables

Variable	Type	Description
<code>itsMainPane</code>	<code>CPane</code>	The document's main pane. <code>NIL</code> if the document has no main pane. The main pane's enclosure should be <code>itsWindow</code> , an instance variable inherited from <code>CDirector</code> .
<code>itsFile</code>	<code>CFile</code>	The file associated with this document. <code>NIL</code> if document has no file.
<code>lastTask</code>	<code>CTask</code>	The last task this document was notified as being completed
<code>undone</code>	<code>Boolean</code>	<code>TRUE</code> if the last task was undone.
<code>itsPrinter</code>	<code>CPrinter</code>	The printer object associated with this document. <code>NIL</code> if document is not printable

## 31 CDocument

---

Variable	Type	Description
dirty	Boolean	TRUE if document has been altered.
pageWidth	integer	The width of a page.
pageHeight	integer	The height of a page.

### Methods

#### Construction and destruction methods

##### IDocument

```
procedure IDocument (aSupervisor: CApplication;  
    printable: Boolean);  
  
void IDocument (struct CApplication *aSupervisor,  
    Boolean printable);
```

Initialize the document. `aSupervisor` must be the `gApplication`. If the value of `printable` is `TRUE`, this method calls `MakePrinter` to create an instance of `CPrinter` and stores it in the `itsPrinter` instance variable.

##### Free/Dispose

```
procedure Free;  
void Dispose (void);
```

Dispose of the document and memory it allocated. This method sends a `Free` or a `Dispose` message to `itsFile`, `lastTask`, and `itsPrinter` if they were allocated. The `Close` method usually calls this method. If your document subclass allocates memory (such as a pane for `itsMainPane`) your method must dispose of it, then you must call the inherited method to dispose of the rest of the document.

#### Command methods

##### Notify

```
procedure Notify (theTask: CTask);  
void Notify (CTask *theTask);
```

A subordinate has completed a task. The default method disposes the current `lastTask` and stores `theTask` in the instance variable `lastTask`. This method sets `undone` to `FALSE` and `dirty` to `TRUE`.



## DoCommand

```
procedure DoCommand (theCommand: longint);
void DoCommand (long theCommand);
```

Handle a document-related command. The default method handles these commands:

Command	Action
cmdClose	Send a Close message to the document.
cmdSave	Set the cursor to the watch cursor and send a DoSave message to the document.
cmdSaveAs	Send a DoSaveFileAs message to the document. The default DoSaveFileAs method sets the cursor to a watch.
cmdRevert	Display a "Do you really want to revert?" alert. If you respond OK, set the cursor to the watch cursor and send a DoRevert message to the document.
cmdPageSetup	If the document is printable, send a DoPageSetup to the document.
cmdPrint	If the document is printable, send a DoPrint message to the document.
cmdUndo	If there is a last task, send either a Redo or an Undo message to the task. Then send an UpdateUndo message to the document.

Your document class will usually override this method. You should handle your own commands first, then call the inherited method to handle the default commands.

## UpdateMenus

```
procedure UpdateMenus;
void UpdateMenus (void);
```

Update the menu items right before they appear on the screen. The default method enables the following commands:

Command	Enabled if...
cmdSaveAs	always
cmdSave	the document is dirty.
cmdRevert	a file is associated with the document and it's dirty.
cmdPageSetup	the document is printable.
cmdPrint	the document is printable.
cmdUndo	there is a last task to undo.

Your document class should override this method to enable the appropriate commands for your document. Be sure you call the inherited method before you enable your document's commands.

### Appearance Methods

#### Close

```
function Close (quitting: Boolean): Boolean;  
Boolean Close (Boolean quitting);
```

A document is being closed. This method sends a `ConfirmClose` message to the document. If the result is `TRUE`, it sends a `Close` message to `itsFile` (if it has one). The `quitting` parameter tells the `ConfirmClose` method whether to ask to save before “closing” or “quitting.” If the document was actually closed, this method returns `TRUE`. Otherwise it returns `FALSE`.

#### CloseWind

```
procedure CloseWind (theWindow: CWindow);  
void CloseWind (CWindow *theWindow);
```

Close the specified window. The default method sends a `Close` message to the document. If you want to be able to close windows without actually closing a file, you should override this method. Otherwise, your document class should not override this method.

#### ConfirmClose

```
function ConfirmClose (quitting: Boolean): Boolean;  
Boolean ConfirmClose (Boolean quitting);
```

Display a “Save before closing?” or “Save before quitting?” dialog. If you answer yes, send a `DoSave` to the document and return `TRUE`. If you answer no, just return `TRUE`. If you answer Cancel, return `FALSE`. If your document does not have a file, your document class should override this method with a method that always returns `TRUE`.

### File Creation

#### NewFile

```
procedure NewFile;  
void NewFile (void);
```

Open a new file. The default method does nothing. Your `CreateDocument` method in your application class should send a `NewFile` message to the document it creates. Your document class should override this method to do the following:

- Create a new window and assign it to `itsWindow`
- Create a new file and assign it to `itsFile`
- Create the panes you need and assign the main pane to `itsMainPane`



**OpenFile**

```
procedure OpenFile (macSFReply: SFReply);
void OpenFile (SFReply *macSFReply);
```

Open an existing file. The default method does nothing. Your `OpenDocument` method in your application class should send an `OpenFile` message to the document it creates. Your document class should override this method and do the following:

- Create a new window and assign it to `itsWindow`
- Open the file specified in `macSFReply` and assign it to `itsFile`
- Create the panes you need and assign the main pane to `itsMainPane`
- Display the contents of the file in the pane

**Note**

To learn how to specify a file, see the class `CFile`. To learn how to read and write from a data file, see the class `CDataFile` on page 237.

**Printing Methods****MakePrinter**

```
procedure MakePrinter;
void MakePrinter (void);
```

Make the printer object for this document. For more information on printer objects, see page 363. If you create subclass of `CPrinter`, you should override this method to create an object of your class.

**Paginate**

```
procedure Paginate;
void Paginate (void);
```

Send a `Paginate` message to `itsMainPane` if its not `NIL`, otherwise send a `ResetPagination` message to `itsPrinter`.

**PageCount**

```
function PageCount: integer;
short PageCount (void);
```

Return the number of pages in the document. This method sends a `GetStripCount` message to `itsPrinter` to find out how many pages the document will take. If there are too many pages (more than 999), it raises an exception.

If the document doesn't have a `CPrinter` object associated with it, this method returns zero.

## ◆ 31 CDocument

---

### AboutToPrint

```
procedure AboutToPrint (var firstPage, lastPage:
    integer);
void AboutToPrint (short *firstPage, short *lastPage);
```

Check the range of pages to be printed. The default method changes lastPage to be equal to PageCount. Your document class should override this method to do whatever is appropriate for your application. This is the place where the document can request information about the page size from itsPrinter.

### PrintPageOfDoc

```
procedure PrintPageOfDoc (pageNum: integer);
void PrintPageOfDoc (short pageNum);
```

Print the specified page. The default method sends a PrintPage message to itsMainPane if it's not NIL.

### DonePrinting

```
procedure DonePrinting;
void DonePrinting (void);
```

Printing is complete. The PrintPageRange method in CPrinter sends this message when the print loop is over. This method sends a DonePrinting message to itsMainPane.

### Filing Methods

### DoSave

```
function DoSave: Boolean;
Boolean DoSave (void);
```

Save the document under its current name and return TRUE if successful. The current file is available through the instance variable itsFile. The default method does nothing. Your document class must override this method.

### DoSaveAs

```
function DoSaveAs (macSFReply: SFReply): Boolean;
Boolean DoSaveAs (SFReply *macSFReply);
```

Save the document under a new name and return TRUE if successful. The macSFReply record specifies where to write the file. The default method does nothing. Your document class must override this method.

### DoRevert

```
procedure DoRevert;
void DoRevert (void);
```

Revert to the last saved version of this document. The default method does nothing. If you want your application to support the **Revert** command, you'll have to override this method.

**DoSaveFileAs**

```
function DoSaveFileAs: Boolean;
Boolean DoSaveFileAs (void);
```

Respond to a **Save As...** command and return TRUE if successful. This method sends a `PickFileName` message to the document. If you provide a good file name, it sends a `DoSaveAs` message to the document. Since this method implements the standard **Save As...** command through two other messages, you won't need to override this method.

**PickFileName**

```
procedure PickFileName (var macSFReply: SFReply);
void PickFileName (SFReply *macSFReply);
```

Display a standard get file dialog to get a new file name. `PickFileName` uses the `GetName` method to specify the default name.

**GetName**

```
procedure GetName (var theName: Str255);
void GetName (Str255 theName);
```

Get the name of the document. If there is a file associated with the document, this method returns the name of the file. If there is no file associated with the document, but there is a window associated with it, it returns the title of the window. If there is neither a file nor a window associated with the document, this method returns a null string.

**Undo methods****UpdateUndo**

```
procedure UpdateUndo;
void UpdateUndo (void);
```

Update the **Undo/Redo** menu item to have the correct wording. Default method sends a `GetNameIndex` message to the `lastTask` to find its string in the `STRtaskNames STR#` resource. Your document class should not override this method.

**Class Resources****Resource**

```
STRprompt 150
ALRTrevert 150
ALRTsaveChanges 151
STRtaskNames 130
STRcommon 128
```

**Description**

```
STR resource ID for
PickFileName prompt string
Revert to saved alert
Save changes before close/quit
alert
STR# resource ID for task names
STR# resource for commonly
used strings
```

## 31 *CDocument*

---

# CEditText

---

## 32

### Introduction

CEditText implements a pane that displays text. This class uses the Macintosh TextEdit routines.

### Heritage

Superclass	CAbstractText
Subclasses	CStaticText

### Using CEditText

Use CEditText whenever you need to display unformatted text. An edit text pane is usually the panorama in a scroll pane so you can scroll through the text. The DoCommand method of an edit text pane handles all the common text editing commands such as cutting and pasting, font selection, line spacing, etc. The Specify method, inherited from CAbstractText, on page 120, lets you choose whether the user can edit and copy your text pane's text.

CEditText does not use the Styled Text Edit routines described in *Inside Macintosh V*. To create a pane of styled text, use CStyleText.

To make sure that the edit text pane responds to commands, you must place it in the chain of command. The best way to do this is to set the value of your document's itsGopher instance variable to the edit pane.

Because CEditText is based on the Macintosh TextEdit (TE) routines, it has some limitations. It's designed to edit small amounts of text. The maximum number of characters you can store in a CEditText record is around 32,000, but you'll notice performance degradation long before it gets that big. To create a pane to display more text (for a text editor, for example), create a subclass of CAbstractText, described on page 117.

### Variables

Variable	Type	Description
macTE	TEHandle	TextEdit record handle.
spacingCmd	longint	Line spacing command number.
alignCmd	longint	Alignment command number.

### Methods

#### Construction and destruction methods

##### IEditText

```

procedure IEditText (anEnclosure: CView;
    aSupervisor: CBureaucrat;
    aWidth, aHeight: integer;
    aHEnc, aVEnc: integer;
    aHSizing, aVSizing: SizingOption;
    aLineWidth: integer);

void IEditText (CView *anEnclosure,
    CBureaucrat *aSupervisor,
    short aWidth, short aHeight,
    short aHEnc, short aVEnc,
    SizingOption aHSizing, SizingOption aVSizing,
    short aLineWidth);

```

Initialize an edit text pane. Most of the arguments to this method are identical to the pane initialization. ALineWidth specifies how wide the lines should be. If it's less than zero, the width is the same as the Macintosh TE record's viewRect.

#### Note

The descriptions of the other arguments are in CPane on page 321.

##### IViewRes

```

procedure IViewRes (rType:ResType; resID: integer;
    anEnclosure: CView; aSupervisor: CBureaucrat);

void IViewRes (ResType rType, short resID,
    CView *anEnclosure, CBureaucrat *aSupervisor);

```

Initialize edit text pane from a resource template. RType is the resource type for the CView subclass you want to initialize. ResID is the resource ID of the resource. AnEnclosure and aSupervisor are the enclosure and supervisor of the pane.

To initialize a text pane from a resource file, use a 'AbTx' resource.

**Note**

Earlier versions of the THINK Class Library used 'St Tx' resources to initialize static text and edit text panes. You should make sure that any old programs do not use 'StTx'.

**MakeMacTE**

```
procedure MakeMacTE;
void MakeMacTE (void);
```

Create a TextEdit record and set macTE to it.

**Free/Dispose**

```
procedure Free;
void Dispose (void);
```

Dispose the TextEdit record macTE.

**Mouse and Keystrokes methods****DoClick**

```
procedure DoClick (hitPt: Point;
  modifierKeys: integer; when: longint);
void DoClick (Point hitPt, short modifierKeys,
  long when);
```

Handle a mouse down in an edit text pane. This method sends a SelectionChanged message after processing the click.

**Command methods****PerformEditCommand**

```
procedure PerformEditCommand (theCommand: longint);
void PerformEditCommand (long theCommand);
```

Perform the standard cut, copy, paste, and clear commands on the text. The task classes call this method to undo an edit command.

**TypeChar**

```
procedure TypeChar (theChar: char;
  theModifiers: integer);
void TypeChar (char theChar, short theModifiers);
```

Process a keystroke. This method does not need to set up for an **Undo** command and should handle the key directly. The task classes call this method to undo key strokes.

**Display methods****Draw**

```
procedure Draw (var area: Rect);
void Draw (Rect *area);
```

Draw the text pane.

## ◆ 32 CEditText

---

<b>Scroll</b>	<pre>void Scroll (long hDelta, long vDelta,             Boolean redraw);</pre> <pre>procedure Scroll (hDelta, vDelta: longint;                 redraw: Boolean);</pre> <p>Scroll the text within the pane by hDelta characters and vDelta lines.</p>
<b>Activate</b>	<pre>procedure Activate;</pre> <pre>void Activate (void);</pre> <p>Activate the text pane. This method enables the editing commands and either highlights the selection or shows the text insertion caret.</p>
<b>Deactivate</b>	<pre>procedure Deactivate;</pre> <pre>void Deactivate (void);</pre> <p>Deactivate the text pane. This method disables the editing commands and either unhighlights the selection or hides the text insertion caret.</p>
<b>SetSelection</b>	<pre>procedure SetSelection (selStart, selEnd: long;                         fRedraw: Boolean);</pre> <pre>void SetSelection (long selStart, long selEnd,                     Boolean fRedraw);</pre> <p>Set the selection to the range corresponding to character positions selStart through selEnd.</p>
<b>GetSelection</b>	<pre>procedure GetSelection (var selStart, selEnd:                         longint);</pre> <pre>void GetSelection (long *selStart, long *selEnd);</pre> <p>Return the start and end of the current selection.</p>
	<b>Text specification methods</b>
<b>SetTextPtr</b>	<pre>procedure SetTextPtr (textPtr: Ptr;                       numChars: longint);</pre> <pre>void SetTextPtr (Ptr textPtr, long numChars);</pre> <p>Use the first numChars characters that textPtr points to as the text for this abstract text object. This method makes a copy of the text.</p>
<b>GetTextHandle</b>	<pre>function GetTextHandle: CharsHandle;</pre> <pre>CharsHandle GetTextHandle (void);</pre> <p>Return a handle to the text of the text. This method returns a handle to the actual text, not to a copy of the text.</p>





<b>CopyTextRange</b>	<pre>function CopyTextRange (start, end: long): Handle; Handle CopyTextRange (long start, long end);</pre> <p>Return a copy of the range of text specified by start and end.</p>
<b>InsertTextPtr</b>	<pre>procedure InsertTextPtr (text: Ptr; length: longint;     fRedraw: Boolean); void InsertTextPtr (Ptr text, long length,     Boolean fRedraw);</pre> <p>Insert a copy of the given text and length at the start of the current selection. If fRedraw is TRUE, redraw the pane at the next update event.</p>
<b>CheckInsertion</b>	<pre>procedure CheckInsertion (insertLen: longint;     useSelection: Boolean); void CheckInsertion (long insertLen,     Boolean useSelection);</pre> <p>Check whether an insertion of insertLen characters would exceed TextEdit's capacity. If useSelection is TRUE, this method deducts the size of the selection from the total length. If the insertion would fail, this method calls Failure.</p>
<b>Text characteristics methods</b>	
<b>SetFontNumber</b>	<pre>procedure SetFontNumber (aFontNumber: integer); void SetFontNumber (short aFontNumber);</pre> <p>Set the font for this text pane by font number.</p>
<b>SetFontStyle</b>	<pre>procedure SetFontStyle (aStyle: Style); void SetFontStyle (short aStyle);</pre> <p>Toggle the font style for this text pane. AStyle may be one of: bold, italic, underline, outline, shadow, condense, or extend</p>
<b>SetFontSize</b>	<pre>procedure SetFontSize (aSize: integer); void SetFontSize (short aSize);</pre> <p>Set the font size for this text pane to the specified size.</p>
<b>SetTextMode</b>	<pre>procedure SetTextMode (aMode: integer); void SetTextMode (short aMode);</pre> <p>Set the text mode for this text pane to the specified mode. AMode can be one of srcOr, srcXor, or srcBic.</p>

## 32 CEditText

---

<b>SetAlignCmd</b>	<pre>procedure SetAlignCmd (anAlignCmd: long); void SetAlignCmd (long anAlignCmd);</pre> <p>Set the text alignment for this text pane. This method uses the THINK Class Library's names for the alignment choices: <code>cmdAlignLeft</code>, <code>cmdAlignRight</code>, or <code>cmdAlignCenter</code>.</p>
<b>GetAlignCmd</b>	<pre>function GetAlignCmd: longint; long GetAlignCmd (void);</pre> <p>Return the current alignment for this text pane. It can be one of <code>cmdAlignLeft</code>, <code>cmdAlignRight</code>, <code>cmdAlignCenter</code>, or <code>cmdNull</code>.</p>
<b>SetAlignment</b>	<pre>procedure SetAlignment (anAlignment: long); void SetAlignment (long anAlignment);</pre> <p>Set the text alignment for this text pane. This method uses TextEdit's names for the alignment choices: <code>teFlushDefault</code>, <code>teFlushLeft</code>, <code>teCenter</code>, or <code>teFlushRight</code>.</p>
<b>SetSpacingCmd</b>	<pre>procedure SetSpacingCmd (aSpacingCmd: longint); void SetSpacingCmd (long aSpacingCmd);</pre> <p>Set the space between lines of text. <code>ASpacingCmd</code> can be one of <code>cmdSingleSpace</code>, <code>cmdlHalfSpace</code>, or <code>cmdDoubleSpace</code>.</p>
<b>GetSpacingCmd</b>	<pre>function GetSpacingCmd: longint; long GetSpacingCmd (void);</pre> <p>Return the space between lines of text. It can be one of <code>cmdSingleSpace</code>, <code>cmdlHalfSpace</code>, or <code>cmdDoubleSpace</code>.</p>
<b>GetTEFontInfo</b>	<pre>procedure GetTEFontInfo (var macFontInfo: FontInfo); void GetTEFontInfo (FontInfo *macFontInfo);</pre> <p>Return a QuickDraw FontInfo record for the font that this text pane uses.</p>
<b>GetHeight</b>	<pre>function GetHeight (startLine endLine: longint):     longint; long GetHeight (long startLine, long endLine);</pre> <p>Return the total height in pixels of the indicated lines of text.</p>
<b>GetCharOffset</b>	<pre>function GetCharOffset (aPt: LongPt): longint; long GetCharOffset (LongPt *aPt);</pre> <p>Return the character position nearest the coordinate <code>aPt</code>. <code>APt</code> must be in frame coordinates.</p>



<b>GetCharPoint</b>	<pre>procedure GetCharPoint (offset: long;     var aPt: LongPt); void GetCharPoint ( long offset, LongPt *aPt);</pre> <p>Return the coordinate of the character position offset. APt is in frame coordinates.</p>
<b>GetCharStyle</b>	<pre>procedure GetCharStyle (charOffset: long;     var theStyle: TextStyle); void GetCharStyle (long charOffset,     TextStyle *theStyle);</pre> <p>Return style information for the character at position charOffset.</p>
<b>GetTextStyle</b>	<pre>procedure GetTextStyle (var whichAttributes: integer;     var aStyle: TextStyle); void GetTextStyle (short *whichAttributes,     TextStyle *aStyle);</pre> <p>Return current style information for this text pane. WhichAttributes is a flag that indicates which text attributes to report on. The attributes flags and TextStyle record are described in <i>Inside Macintosh VI</i>, Chapter 15, "TextEdit"</p>
	<b>Calibrating methods</b>
<b>ResizeFrame</b>	<pre>procedure ResizeFrame (delta: Rect); void ResizeFrame (Rect *delta);</pre> <p>Resize the static text's frame when the size of its pane changes. The delta rectangle specifies how each side changes. Positive values mean down and to the right. Negative values mean up and to the left.</p>
<b>CalcTERects</b>	<pre>procedure CalcTERects; void CalcTERects (void);</pre> <p>Determine the destination and view rectangles for this text pane's TextEdit record.</p>
<b>AdjustBounds</b>	<pre>procedure AdjustBounds; void AdjustBounds (void);</pre> <p>Adjust the bounds of a this text pane to match its TextEdit record. When you do something that could change the line width or the number of lines, send this message.</p>

## ◆ 32 CEditText

---

<b>FindLine</b>	<pre>function FindLine (charPos: longint): longint; short FindLine (long charPos);</pre> <p>Return the line number containing the specified character position. Both line and character numbering start at zero. If the character position is before the start of the text (a negative number), this method returns zero. If the character position is beyond the end of the text, this method return the number of the last line.</p>
<b>GetLength</b>	<pre>function GetLength: longint; long GetLength (void);</pre> <p>Return the length in bytes of this text pane's text buffer.</p>
<b>GetNumLines</b>	<pre>function GetNumLines: longint long GetNumLines (void);</pre> <p>Return the total number of lines in this text pane's text buffer.</p>
	<b>Printing methods</b>
<b>AboutToPrint</b>	<pre>procedure AboutToPrint (var firstPage, lastPage: integer); void AboutToPrint (short *firstPage, short *lastPage);</pre> <p>The specified range of pages is about to be printed. This method deactivates the text pane to unhighlight the current selection.</p>
<b>PrintPage</b>	<pre>procedure PrintPage (pageNum: integer; pageWidth, pageHeight: integer; aPrinter: CPrinter); void PrintPage (short pageNum, short pageWidth, short pageHeight, CPrinter *aPrinter);</pre> <p>Print the specified page.</p>
<b>DonePrinting</b>	<pre>procedure DonePrinting; void DonePrinting (void);</pre> <p>Printing is over. This method re-highlights the current selection if the edit pane is active.</p>
	<b>Cursor methods</b>
<b>Dawdle</b>	<pre>procedure Dawdle (var maxSleep: longint); void Dawdle (long *maxSleep);</pre> <p>This method flashes the insertion point when the edit text pane is active. This method sets maxSleep to the value of GetCaretTime, which is the</p>



rate at which the insertion point blinks. Setting this value ensures that `WaitNextEvent` will generate a null event at least that often.

---

**Note**

To learn more about “sleep time,” see the description of the `Dawdle` message in `CBureaucrat` page 190.

---



# *CEnvironment*

---

## 33



### Introduction

CEnvironment maintains a drawing environment for any pane.

### Heritage

Superclass  
Subclasses

CObject  
CTextEnvironment

### Using CEnvironment

Every pane has an `itsEnvironment` instance variable. If this variable points to an object of this class, the `Prepare` method sends it a `Restore` message to set up the QuickDraw drawing environment for the pane.

You can use this class to make sure that the drawing environment is set up correctly. Or you might want to change the drawing environment according to some saved settings.

### Variables

This class has no instance variables.

### Methods

#### Restore

```
procedure Restore;  
void Restore (void);
```

Restore the drawing environment in the current port. The default method just calls the QuickDraw routine `PenNormal`.

## ◆ 33 *CEnvironment*

---



# CError

## 34

### Introduction

CError is an error-handling class. You can use the global error handler to report errors.

#### Note

This class is included for compatibility with earlier versions of the THINK Class Library. New programs that you write with the THINK Class Library should use the exception handling mechanism described on page 101.

### Heritage

Superclass	CObject
Subclasses	None

### Using CError

The application initialization method `IAplication` creates an instance of CError and stores it in the global variable `gError`. Several objects in the THINK Class Library use the global error handler to post error messages. The default error handler reports messages and gives you a chance to quit or to proceed with the application. Create a subclass of this method to implement more sophisticated error recovery.

### Variables

#### Global variable

Variable	Type	Description
<code>gError</code>	CError	Global error handler.

#### Instance variables

This class has no instance variables.

### Methods

#### Error reporting methods

##### SevereMacError

```
procedure SevereMacError (macErr: OSerr);
void SevereMacError (OSerr macErr);
```

Report an operating system error in a Stop Alert that gives you a chance to quit the application or jump to the beginning of the event loop. This method looks for an `Err` resource with the same ID as an operating system error. If it doesn't find one, it uses the default string (STR 200). If you press the Quit button, this method sends a Quit message to the application. If you press the Proceed button, this method sends the application a `JumpToEventLoop` message.

##### CheckOSError

```
function CheckOSError (macErr: OSerr): Boolean;
Boolean CheckOSError (OSerr macErr);
```

Check for a Macintosh operating system error. Return `TRUE` if everything is OK, `FALSE` otherwise. If `macErr` is an error, this method displays a Stop Alert. This method looks for an `Err` resource with the same ID as the operating system error. If it can't find one, it uses a default string.

##### PostAlert

```
procedure PostAlert (STRid: integer; index: integer);
void PostAlert (short STRid, short index);
```

Post a general alert and return. `STRid` is the resource ID of a `STR#` resource. `Index` is the index into the `STR#` resource. The string is displayed in a generic alert box.

##### MissingResources

```
procedure MissingResources;
void MissingResources (void);
```

Post an alert announcing that the application can't find its resources, and exit the application. The alert suggests that your project doesn't have an associated resource file. The alert can't be in a resource, since this method is called only if the resource file is unavailable.

### Functions

Note that these are functions and procedures, not methods.

##### GrowZoneFunc

```
function GrowZoneFunc (bytesNeeded: Size): longint;
pascal long GrowZoneFunc (Size bytesNeeded);
```

The application method `InitMemory` installs this function as the application's `GrowZone` function which is called in low memory conditions. This



function sends a `GrowMemory` message to the application. See `CApplication` on page 137.

### CheckResource

```
procedure CheckResource (r: Handle);
void CheckResource (Handle r);
```

If the handle `h` is `NIL`, this procedure sends the message `SevereMacError(resNotFound)` to the global `gError`. Call this function after trying to retrieve a resource.

### CheckAllocation

```
procedure CheckAllocation (p: Ptr);
void CheckAllocation (void *p);
```

If `p` is `NIL`, the procedure calls `SevereMacError(MemError)` to report a severe memory error. Call this function after trying to allocate memory.

## Class resources

Resource	Description
ALRT/DITL 128	Generic alert box. Contains ^0 for use with <code>ParamText</code> .
ALRT/DITL 200	Alert box for a <code>SevereError</code>
ALRT/DITL 300	Alert box for a Macintosh operating system error.
STR 300	String that reports a severe Macintosh system error
Estr resources	An <code>Estr</code> resource has the same format as a <code>STR</code> resource. The <code>SevereMacError</code> method looks for an <code>Estr</code> resource with the same ID as an operating system error. For instance, <code>Estr -42</code> might read "Too many files open."



## Introduction

CFile is an abstract class for implementing classes that deal with disk files.

## Heritage

Superclass	CObject
Subclasses	CDataFile
	CResFile

## Using CFile

CFile is an abstract class for dealing with Macintosh disk files. Most of the time you'll use the CDataFile subclass to work with regular data files.

Before you open a file, you must specify it. Specifying means identifying it to the Macintosh file manager. This class gives you three ways to specify a file depending on the type of information you can provide. The most common specification method, `SFSpecify`, lets you use an `SFReply` record from the standard file dialogs to identify a file.

Some CFile methods use the THINK Class Library's exception handling library if something goes wrong in a file operation. For more information on exception handling, see Chapter 8, "Exception Handling."

---

### Note

This version of CFile handles errors differently than the version included with earlier versions of the THINK Class Library. The old version is now named OFile and is in the `CompatibilityClasses` folder.

---

### Variables

Variable	Type	Description
name	Str63	File name
volNum	integer	Volume containing the file
dirID	longint	Directory within the volume

### Methods

#### Construction and destruction methods

##### IFile

```
procedure IFile;
void IFile (void);
```

Initialize the file object.

##### Free/Dispose

```
procedure Free;
void Dispose (void);
```

Close and dispose of the file object.

#### Specifying methods

##### Specify

```
procedure Specify (aName: Str63; aVolNum: integer);
void Specify (Str63 aName, short aVolNum);
```

Specify a file by its name and volume reference number. Use this method to specify files on MFS volumes.

##### SpecifyHFS

```
procedure SpecifyHFS (aName: Str63; aVolNum: integer;
    aDirID: longint);
void SpecifyHFS (Str63 aName, short aVolNum,
    long aDirID);
```

Specify a file by its name, volume number, and directory ID. If the specification is for an alias, this method resolves it.

##### SpecifyFSSpec

```
procedure SpecifyFSSpec (aFileSpec: FSSpec);
void SpecifyFSSpec (const FSSpec *aFileSpec)
```

Specify the File using a File Manager FSSpec record. If the specification is for an alias, this method resolves it.

##### SFSpecify

```
procedure SFSpecify (macSFReply: SFReplyPtr);
void SFSpecify (SFReply *macSFReply);
```

Specify a file from the information in a macSFReply record. If the specification is for an alias, this method resolves it. Use this method to specify a file that the user chose through a standard file dialog.

**ResolveFileAlias**

```
procedure ResolveFileAlias;  
void ResolveFileAlias (void)  
If the file specification is an alias, resolve it.
```

**Open and close methods****Open**

```
procedure Open (permission: SignedByte);  
void Open (SignedByte permission);  
Open the file with the specified permission. This method does nothing. Sub-  
classes must override this method. For an example of how to write an Open  
method, see CDataFile and other descendants of this class.
```

**Close**

```
procedure Close;  
void Close (void);  
Close this file. This method does nothing. Subclasses must override this  
method.
```

**Accessing methods****GetName**

```
procedure GetName (var theName: Str63);  
void GetName (Str63 theName);  
Get the name of the file.
```

**GetFSSpec**

```
procedure GetFSSpec (var aFileSpec: FSSpec);  
void GetFSSpec (FSSpec *aFileSpec)  
Get an FSSpec record for this file.
```

**ExistsOnDisk**

```
function ExistsOnDisk: Boolean;  
Boolean ExistsOnDisk (void);  
Returns TRUE if there's an existing file that matches the current specification.
```

**GetMacFileInfo**

```
procedure GetMacFileInfo (var fileInfo: FInfo)  
void GetMacFileInfo (FInfo *fileInfo);  
Return the Finder information for this file. The information includes the file's  
type, creator, and icon position.
```

**Filing methods****CreateNew**

```
procedure CreateNew (creator, fType: OSType);  
void CreateNew (OSType creator, OSType fType);  
Create new file with the specified creator and file type. This method uses the  
name and volume information you set up with one of the specification
```

## ◆ 35 CFile

---

methods. You can use the application signature in gSignature for the creator.

### **ThrowOut**

```
procedure ThrowOut;  
void ThrowOut (void);  
Close the file and delete it.
```

### **ChangeName**

```
procedure ChangeName (newName: Str255);  
void ChangeName (Str63 newName);  
Give this file a new name.
```



# CFWDesktop

## 36

### Introduction

CFWDesktop is a subclass of CDesktop that supports floating windows.

### Heritage

Superclass	CDesktop
Subclasses	None

### Using CFWDesktop

CFWDesktop is a subclass of CDesktop that supports floating windows. In general, your application shouldn't need to send messages to the desktop.

If you use CFWDesktop, be sure to override MakeDesktop to create a floating window desktop and store the object in gDesktop.

This is how you do it in Pascal:

```
procedure CMyApp.MakeDesktop;
begin
    new(CFWDesktop(gDesktop));
    CFWDesktop(gDesktop).IFWDesktop(SELF);
end;
```

And this is how you do it in C:

```
void CMyApp::MakeDesktop(void)
{
    gDesktop = new(CFWDesktop);
    ((CFWDesktop*)gDesktop)->IFWDesktop(this);
}
```

To make a floating window, just pass TRUE as the aFloating parameter to IWindow, CWindow's initialization method.

### Variables

Variable	Type	Desktop
itsFloats	CList	List of floating windows
topFloat	CWindow	Topmost floating window

### Methods

#### Construction and destruction methods

##### IFWDesktop

```
procedure IFWDesktop (aSupervisor: CBureaucrat);
void IFWDesktop (CBureaucrat *aSupervisor);
```

Initialize the desktop. This method calls CDesktop's initialization method and creates the list of floating window..

##### Free/Dispose

```
procedure Free;
void Dispose (void);
```

Dispose of the desktop and send this message to all of the desktop's windows.

#### Appearance methods

##### Show

```
procedure Show;
void Show (void);
```

Makes a desktop visible by showing all of its windows. This method sends a Show message to all of the desktop's windows and floating windows.

##### Hide

```
procedure Hide;
void Hide (void);
```

Hide a desktop by hiding all of its windows. This method sends a Hide message to all of the desktop's windows and floating windows.

##### Activate

```
procedure Activate;
void Activate (void);
```

Activate the desktop by activating all the windows. This method sends an Activate message to the desktop's top window and to all of the floating windows.

##### Deactivate

```
procedure Deactivate;
void Deactivate (void);
```

Deactivate a desktop by deactivating all the windows. This method sends a Deactivate message to the desktop's top window and to all of the floating windows.

**Mouse methods****AdjustCursor**

```
procedure AdjustCursor (where: Point;  
    mouseRgn: RgnHandle);  
void AdjustCursor (Point where, RgnHandle mouseRgn);
```

Adjust the cursor shape when the mouse is in an insignificant part of the desktop. The default method sets the cursor to an arrow.

**Window methods**

You do not have to use any of these methods yourself. Directors, objects that manage the interaction between windows and the desktop, take care of sending most of these messages.

**AddWind**

```
procedure AddWind (theWindow: CWindow);  
void AddWind (CWindow *theWindow);
```

Add theWindow to the desktop's window list. If the window is a floating window, add it to the beginning of the itsFloats list.

**RemoveWind**

```
procedure RemoveWind (theWindow: CWindow);  
void RemoveWind (CWindow *theWindow);
```

Remove theWindow from the desktop's window list. If the window is a floating window, remove it from the itsFloats list.

**SelectWind**

```
procedure SelectWind (theWindow: CWindow);  
void SelectWind (CWindow *theWindow);
```

Select theWindow and bring theWindow to the front. If the window is a floating window, bring it to the front of the itsFloats list and mark it as the topFloat window.

**ShowWind**

```
procedure ShowWind (theWindow: CWindow);  
void ShowWind (CWindow *theWindow);
```

Make theWindow visible on the desktop.

**HideWind**

```
procedure HideWind (theWindow: CWindow);  
void HideWind (CWindow *theWindow);
```

Hide theWindow.

## ◆ 36 CFWDesktop

---

### DragWind

```
procedure DragWind (theWindow: CWindow;  
    macEvent: EventRecord);
```

```
void DragWind (CWindow *theWindow,  
    EventRecord *macEvent);
```

Drag theWindow on the desktop. This method does not use DragWindow Toolbox routine.

### CalTopFloat

```
procedure CalcTopFloat;
```

```
void CalcTopFloat (void);
```

Set topFloat to the first visible floating window. Send this message to the desktop if you hide or remove a floating window.

### Cleanup method

### Cleanup

```
procedure Cleanup;
```

This method is for THINK Pascal. When you're running in the THINK Pascal environment, all the windows share the same space. As you debug your program and open source windows and debugging windows, the window order changes. When you resume your program, THINK Pascal will move your application's active window to the front, but it doesn't know about floating windows. This method moves all of THINK Pascal's windows to the back, and cleans up your application windows so the floating windows float and the application windows are in the right order.

# CGridSelector

---

## 37

### Introduction

CGridSelector is an abstract class for drawing a pane with several items arranged in a table.

### Heritage

Superclass	CSelector
Subclasses	CCharGrid CPatternGrid

### Using CGridSelector

CGridSelector is a descendant of CSelector that lets you display items in a table. To use CGridSelector, all you have to do in your subclass is override the DrawItem method. CGridSelector takes care of everything else.

The THINK Class Library includes two subclasses of CGridSelector. CPatternGrid displays a pattern palette. CCharGrid displays a set of characters. You can use CCharGrid with a special font to display tool palettes.

Items in a grid are numbered from 1, row first.

### Variables

Variable	Type	Description
rows	integer	The number of rows in the grid.
columns	integer	The number of columns in the grid.
boxWidth	integer	The width in pixels of each box.
boxHeight	integer	The height in pixels of each box.
gridOn	Boolean	True if the grid lines should be drawn.

### Methods

#### Construction methods

##### IGridSelector

```
procedure IGridSelector (anEnclosure: CView;  
    aSupervisor: CBureaucrat;  
    aWidth, aHeight: integer;  
    aHEnc1, aVEnc1: integer;  
    aHSizing, aVSizing: SizingOption;  
    aSelection, aCommandBase: integer  
    aRows, aColumns, aBoxWidth, aBoxHeight: integer);  
  
void IGridSelector (CView *anEnclosure,  
    CBureaucrat *aSupervisor,  
    short aWidth, short aHeight,  
    short aHEnc1, short aVEnc1,  
    SizingOption aHSizing, SizingOption aVSizing,  
    short aSelection, short aCommandBase  
    short aRows, short aColumns,  
    short aBoxWidth, short aBoxHeight);
```

Initialize a selector. The first eight arguments to this routine are identical to the ones for IPane.

*See the description of CSelector's DoClick method on page 395 to learn how selectors generate command numbers.*

ASelection is the initial item to select. ACommandBase is the base value for converting the selected item into a command number. ARows and aColumns determine how many boxes are in the grid. ABoxWidth and aBoxHeight determine the size of each box in pixels. Finally, IGridSelector sets the vertical and horizontal scales to the size of the boxes.

---

#### Note

The descriptions of the other arguments are in CPane on page 321.

---

#### Drawing methods

##### Draw

```
procedure Draw (var area: Rect);  
  
void Draw (Rect *area);
```

Draws the grid selector. If gridOn is true, this method first draws the grid. Then it uses DrawItem to draw each box. Finally, it highlights the current selection.

Since most of the work is done in the DrawItem method, you probably won't need to override this method.

**DrawGrid**

```
procedure DrawGrid;
void DrawGrid (void);
```

Draw the grid lines between items. The Draw method calls DrawGrid if gridOn is TRUE.

**DrawItem**

```
procedure DrawItem (theItem: integer;
  theBox: Rect);
void DrawItem (short theItem, Rect *theBox);
```

Draw the item. TheItem is the number of the item to draw, and theBox is the rectangle to draw it in. Your subclass must override this method.

**Accessing methods****HiliteItem**

```
procedure HiliteItem (theItem: integer;
  state: HiliteState);
void HiliteItem (short theItem, HiliteState state);
```

Highlight the specified item. HiliteItem can take on of three values for the state parameter:

<b>Hilite state value</b>	<b>Behavior</b>
hiliteOFF	Invert the item's box.
hiliteON	Invert the item's box.
hiliteDYNAMIC	Flash the edges of the item's box.

If you want to highlight an item differently, you can override this method.

**FindItem**

```
function FindItem (hitPt: Point): integer;
short FindItem (Point hitPt);
```

Determine which item corresponds to a mouse down at a specified point. If the hitPt isn't in the grid, FindItem returns zero. Grid selector items are numbered from 1, row first like this:

1	2	3
4	5	6
7	8	9

**Figure 37-1** Item numbering in grid selectors

## ◆ 37 CGridSelector

---

### FindItemBox

```
procedure FindItemBox (theItem: integer;  
    var theBox: Rect);  
  
short FindItemBox (short theItem, Rect *theBox);
```

Return the box that corresponds to theItem. HiliteItem uses this method to figure out what box to highlight.

### SetGridOn

```
procedure SetGridOn (aGridOn: Boolean);  
void SetGridOn (Boolean aGridOn);
```

Sets the instance variable gridOn to aGridOn. If gridOn is true, the Draw method calls DrawGrid to draw grid lines between items.



## Introduction

CList implements an ordered list of objects.

## Heritage

Superclass	CCluster
Subclasses	None

## Using CList

Use an object of class CList when you need to maintain an ordered list of objects. Several of the objects in the THINK Class Library use CList to maintain lists. You can use the iteration methods inherited from CCluster to apply functions to each item in a list. Every object in a list has an index. Index values begin at 1.

## Variables

This class has no instance variables.

## Methods

For methods that insert new objects before or after existing objects, be sure that the existing object is actually in the list. If you don't, strange things may happen.

### Construction methods

CList inherits the Free (in THINK Pascal) or Dispose (in THINK C) method from CCluster.

### IList

```
procedure IList;  
void IList (void);  
Initialize the list.
```

<b>Insertion and deletion methods</b>	
<b>Append</b>	<pre>procedure Append (theObject: CObject); void Append (CObject *theObject);</pre> <p>Add theObject to the end of the list.</p>
<b>Prepend</b>	<pre>procedure Prepend (theObject: CObject); void Prepend (CObject *theObject);</pre> <p>Add theObject to the beginning of the list.</p>
<b>InsertAfter</b>	<pre>procedure InsertAfter (theObject: CObject;     afterObject: CObject); void InsertAfter (CObject *theObject,     CObject *afterObject);</pre> <p>Insert the theObject after the object specified by afterObject.</p>
<b>InsertAt</b>	<pre>procedure InsertAt (theObject: CObject;     index: longint); void InsertAt (CObject *theObject, long index);</pre> <p>Make theObject be the indexth object in the list. All subsequent objects move down one position</p>
<b>Ordering methods</b>	
<b>BringFront</b>	<pre>procedure BringFront (theObject: CObject); void BringFront (CObject *theObject);</pre> <p>Make theObject be the first object in the list.</p>
<b>SendBack</b>	<pre>procedure SendBack (theObject: CObject); void SendBack (CObject *theObject);</pre> <p>Make theObject be the last object in the list.</p>
<b>MoveUp</b>	<pre>procedure MoveUp (theObject: CObject); void MoveUp (CObject *theObject);</pre> <p>Move theObject up one slot toward the front of the list. The object that was before it in the list moves down one slot.</p>
<b>MoveDown</b>	<pre>procedure MoveDown (theObject: CObject); void MoveDown (CObject *theObject);</pre> <p>Move theObject down one slot toward the end of the list. The object that was after it in the list moves up one slot.</p>

## MoveToIndex

```
procedure MoveToIndex (theObject: CObject;
    index: longint);
```

```
void MoveToIndex (CObject *theObject, long index);
```

Move theObject to the indexth position in the list. If theObject moves back in the list, the items between its original position and its new position move up one slot. If theObject moves forward in the list, the items between its new position and its original position move down one slot.

## Membership methods

## FirstItem

```
function FirstItem: CObject;
```

```
CObject* FirstItem (void);
```

Return the first item in the list.

## LastItem

```
function LastItem: CObject;
```

```
CObject* LastItem (void);
```

Return the last item in the list.

**NthItem**

```
function NthItem (n: longint): CObject;
```

```
CObject* NthItem (long n);
```

Return the nth item in the list.

## FindIndex

```
function FindIndex (theObject: CObject): longint;
```

```
long FindIndex (CObject *theObject);
```

Return the index of the object. Indexes begin at 1. If the item is not in the list, this method returns 0.

## FirstSuccess

```
function FirstSuccess (function testFunc(theObject:
    CObject): Boolean): CObject;
```

```
CObject* FirstSuccess (TestFunc testFunc);
```

Return the first item that causes the testFunc function to return TRUE.

In THINK C, testFunc must be declared like this:

```
Boolean MyTestFunc (CObject *theObject);
```

## FirstSuccess1

```
function FirstSuccess1 (function testFunc (theObject:
    CObject; theParam: Ptr): Boolean; param: Ptr):
    CObject;
```

```
COBJECT* FirstSuccess1 (TestFunc1 testFunc,
    long param);
```

Return the first item that causes the testFunc function to return TRUE.

In THINK C, testFunc must be declared like this:

```
Boolean MyTestFunc (CObject *theObject,
                    long param);
```

### **LastSuccess**

```
function LastSuccess (function testFunc (theObject:
CObject): Boolean): CObject;
```

```
CObject* LastSuccess (TestFunc testFunc);
```

Return the last item in the list that causes the testFunc function to return TRUE.

In THINK C, testFunc must be declared like this:

```
Boolean MyTestFunc (CObject *theObject);
```

### **LastSuccess1**

```
function LastSuccess1 (function testFunc (theObject:
CObject; theParam: Ptr): Boolean; param: Ptr):
CObject;
```

```
CObject* LastSuccess1 (TestFunc1 testFunc,
long param);
```

Return the last item in the list that causes the testFunc function to return TRUE.

In THINK C, testFunc must be declared like this:

```
Boolean MyTestFunc (CObject *theObject,
                    long param);
```

# *CMBarchore* ◆

---

## 39

### Introduction

CMBarchore is a chore that works with CBartender to redraw the menu bar.

### Heritage

Superclass	CChore
Subclasses	None

### Using CMBarchore

The CBartender class uses a CMBarchore to redraw the menu bar after deleting menus from the menu bar. Your application should not need to use this class.

The CBartender's `DeleteFromBar` method creates a CMBarchore and assigns it as an urgent chore, so the menu bar gets redrawn the next time through the event loop.

### Variables

This class has no instance variables.

### Methods

#### Perform

```
procedure Perform (var maxSleep: Longint);  
void Perform (long *maxSleep);
```

This method calls the Toolbox routine `DrawMenuBar` to redraw the menu bar.

## ◆ 39 *CMBarchore*

---

# *CMenuDefProc* ♦

## 40

---

### Introduction

CMenuDefProc is an abstract class that gives you an object-oriented interface for Macintosh menu definition procedures (MDEFs). The description of this class assumes that you're familiar with the Macintosh Menu Manager in general and with MDEFs in particular. If you need to learn more, see *Inside Macintosh I*, Chapter 11, "The Menu Manager" and *Inside Macintosh V*, Chapter 13, "The Menu Manager."

### Heritage

Superclass	CObject
Subclasses	CPaneMDEF

### Using CMenuDefProc

CMenuDefProc lets you write object-oriented menu definition functions. To write an MDEF object, you need to create a subclass of CMenuDefProc or use one of the ones provided in the THINK Class Library.

CMenuDefProc uses a stub MDEF that contains a jump to a generic menu definition procedure and a reference to an MDEF object. The Menu Manager calls the generic menu definition procedure which dispatches messages to your MDEF object.

### Creating the stub MDEF

The stub MDEF is a 10-byte data structure that looks like this in Pascal:

```
GenericMDEFRec = record
    JMPInstruction: integer;
    defProc: ProcPtr;
    itsMenuDefProc: CMenuDefProc;
end;
```

And in C it looks like this:

```
typedef struct GenericMDEFRec {
    short          JMPinstruction;
    VoidFunc       defProc;
    CMenuDefProc*  itsMenuDefProc;
};
```

To create the MDEF, use your favorite resource editor or resource compiler to create a 10-byte resource that contains these hex values:

```
4EF9 0000 0000 0000 0000
```

*When you create the menu that uses your MDEF, be sure to specify the MDEF ID. Otherwise, the Menu Manager uses the default MDEF.*

Apple reserves IDs 0 to 127 for definition procedures and recommends that you number your MDEF in the range 128 to 4095. In the THINK Class Library, the convention is to give your MDEF the same ID as the menu that you'll be using it with.

### Writing the CMenuDefProc subclass

Your object-oriented MDEF must handles all the messages that a regular MDEF handles. That means that your subclass needs to override all the methods of CMenuDefProc:

Initialization	DrawMenu	ChooseItem
SizeMenu	PlacePopup	

Your subclass can define instance variables to hold whatever information your menu needs to do its work. Your initialization method should allocate memory for the instance variables if necessary. Your subclass must call IMenuDefProc to set up the stub MDEF resource.

If you want to use your MDEF for more than one menu, be careful how you use instance variables. If the instance variables of the MDEF control the state of the menu being displayed, you'll only be able to use it for one menu.

For example, the CPaneMDEF subclass of this class keeps two instance variables to refer to the pane being displayed as a menu and to the tear-off menu. If you were to use the same MDEF based on CPaneMDEF for two menus, both menus would display the same pane.

### Using your CMenuDefProc subclass

The most important thing to keep in mind about using a descendant of CMenuDefProc is that you need to initialize it before the Menu Manager loads the menu associated with it. If you don't, your application will crash.





The reason that it's important to initialize the custom MDEF object before CApplication's `SetUpMenus` or CBartender's `AddMenu` has to do with the way the Menu Manager works and the way `IMenuDefProc` does its job.

*The description for GenericMDEF is near the end of this chapter on page 308*

`IMenuDefProc` loads the MDEF and sets up the fields so `defProc` points to a function called `GenericMDEF`, and the `itsMenuDefProc` points to the current object. `GenericMDEF` is the "real" menu definition procedure that dispatches Menu Manager messages to your MDEF object.

As it's loading the menu, the Toolbox routine `GetMenu` loads the MDEF and sends it an `mSizeMsg` to find out how big the menu is. If you haven't initialized the MDEF, this call to menu definition procedure will jump to random memory, and your program will crash.

### Examples of MDEF objects

If you're including the menu in your application's MBAR resource, override CApplication's `SetUpMenus` method to create the MDEF object and initialize it. Then call CApplication's `SetUpMenus` method. Chances are that you're overriding this method anyway to load resource-based menus like the Font menu and to set up the dimming and checking options for your menus.

This is what your `SetUpMenus` method should look like in Pascal:

```
procedure CMyApp.SetUpMenus;
var
    theMDEF: CMyCustomMDEF;
begin
    new(theMDEF);
    theMDEF.IMyCustomMDEF(CustomID);
    inherited SetUpMenus;
    { load font menu, set check & dim options }
end;
```

In C, it should look like this:

```
void CMyApp::SetUpMenus (void)
{
    CMyCustomMDEF theMDEF;

    theMDEF = new(CMyCustomMDEF);
    theMDEF->IMyCustomMDEF(CustomID);
    inherited::SetUpMenus();
    /* load font menu, set check & dim options */
}
```

Your initialization method (IMyCustomMDEF in this example) must call CMenuDefProc's IMenuDefProc.

If you're using CBartender's AddMenu method to add the menu to the menu bar, you need to create and initialize the MDEF object before you call AddMenu. The next example shows a method that creates a tool menu that uses a custom MDEF and adds it to the end of the menu bar. Note that the example uses the TCL convention of giving the menu and MDEF the same ID.

This is what the method looks like in Pascal:

```
procedure CPaintPane.AddToolMenu;
var
  theToolMDEF: CToolMDEF;
begin
  new(theToolMDEF);
  theToolMDEF.IToolMDEF(ToolMENUID);
  gBartender.AddMenu(ToolMENUID, TRUE, 0);
end;
```

In it looks like this:

```
void CPaintPane::AddToolMenu(void)
{
    CToolMDEF theToolMDEF;

    theToolMDEF = new(CToolMDEF);
    theToolMDEF->IToolMDEF(ToolMENUID);
    gBartender->AddMenu(ToolMENUID, TRUE, 0);
}
```

## Variables

This class has no instance variables.

## Methods

Except for the initialization method, all of CMenuDefProc's methods handle a Menu Manager message to a menu definition procedure. The descriptions given here of what your subclass should do are fairly complete, but for details, be sure to consult *Inside Macintosh I*, Chapter 11, "The Menu Manager" and *Inside Macintosh V*, Chapter 13, "The Menu Manager."

**IMenuDefProc**

```
procedure IMenuDefProc (MDEFid: Integer);
void IMenuDefProc (short MDEFid);
```

Initialize the object. This method loads the stub MDEF and stores the address of the generic MDEF function `GenericMDEF` in the `defProc` field. It stores the reference to this object in the `itsMenuDefProc` field.

**DrawMenu**

```
procedure DrawMenu (macMenu: MenuHandle;
    menuRect: Rect);
void DrawMenu (MenuHandle macMenu, Rect *menuRect);
```

Draw the menu in response to a Menu Manager `mDrawMsg`. `MenuRect` is given in global coordinates, the current port is the Window Manager port, and the clipping region is set to `menuRect`. Your `DrawMenu` method should check whether the menu is enabled and draw it in gray if it's not.

**ChooseItem**

```
procedure ChooseItem (macMenu: MenuHandle;
    menuRect: Rect; hitPt: Point;
    var whichItem: Integer);
void ChooseItem(MenuHandle macMenu, Rect *menuRect,
    Point hitPt, short *whichItem);
```

Choose an item in response to a Menu Manager `mChooseMsg`. Both `hitPt` and `menuRect` are in global coordinates, and `whichItem` is the last item chosen. `WhichItem` is initially 0. Your `ChooseItem` method should check that `hitPt` is within `menuRect`, and if it is, it should unhighlight `whichItem` and highlight the new item (if it's not the same as `whichItem`), then set `whichItem` to the new item. If `hitPt` isn't in `menuRect`, you should unhighlight `whichItem` and set `whichItem` to 0.

**SizeMenu**

```
procedure SizeMenu (macMenu: MenuHandle);
void SizeMenu (MenuHandle macMenu);
```

Set the size of the menu in response to a Menu Manager `mSizeMsg`. Your method should store the height of the menu in `macMenu`'s `menuWidth` field and the height in the `menuHeight` field. Note that the menu manager sends this message to your menu soon after it has been loaded. See "Using your `CMenuDefProc` subclass" on page 304 for more details.

**PlacePopUp**

```
procedure PlacePopUp (macMenu: MenuHandle;
    menuRect: Rect; hitPt: Point;
    var whichItem: integer);
void PlacePopUp (MenuHandle macMenu, Rect *menuRect,
    Point hitPt, short *whichItem);
```

Determine the rectangle for a pop-up menu in response to a Menu Manager `mPopUpMsg`. `WhichItem` contains the previously selected item, and `hitPt` contains the top left corner of the pop-up menu (`hitPt.h` contains

the top and `hitPt.v` contains the left). Your method should set `menuRect` to the rectangle the menu is displayed in. If the menu would scroll, set `whichItem` to the actual top item. *Inside Macintosh V*, Chapter 13, “The Menu Manager” contains additional information you should be aware of.

### Functions

`CMenuDefProc` uses an auxiliary function, `GenericMDEF`, to convert messages from the Menu Manager to object-oriented messages. In THINK C, this function is in `GenericMDEF.c` in the `FW/Tearoffs` folder, and in THINK Pascal it's in `GenericMDEF.p` in the `FW/Tearoffs` folder.

#### GenericMDEF

```
procedure GenericMDEF (theMessage: integer;
    macMenu: MenuHandle;
    menuRect: Rect; hitPt: Point;
    var whichItem: integer);

pascal void GenericMDEF (short theMessage,
    MenuHandle macMenu, Rect *menuRect,
    Point hitPt, short *whichItem);
```

As far as the Menu Manager is concerned, `GenericMDEF` is the “real” menu definition procedure. `GenericMDEF`. This routine dispatches messages to your object-oriented MDEF.

Be aware that the `GenericMDEF` only handles the four standard Menu Manager messages. If you write an MDEF that responds to user messages (as described in *Inside Macintosh V*, Chapter 13, “The Menu Manager”), you will need to write your own version of `GenericMDEF` and write your initialization method accordingly.

# *C*MouseEvent 41 ♦

---

## Introduction

CMouseEvent is an abstract class that implements mouse tracking.

## Heritage

Superclass  
Subclasses

CTask  
You must define a subclass of this class.

## Using CMouseEvent

CMouseEvent is an abstract class that lets you implement undoable mouse-related actions. For example, if you're writing a drawing application, you want to make sure that you can undo anything that you move or draw.

You don't have to use this class to implement mouse-related actions. You can always track the mouse yourself in your pane's `DoClick` method. If you do use CMouseEvent to implement mouse actions, you don't have to make them undoable.

## Defining a mouse task

To implement a mouse tracking task, define a subclass of CMouseEvent and override the `KeepTracking` and `EndTracking` methods. The `KeepTracking` method does whatever you want to happen while the mouse is down. The `EndTracking` method does whatever you want to happen when the mouse is released.

For example, if you're moving a rectangle from one place in a pane to another, the `KeepTracking` method might draw a gray outline that moves as you move the mouse. The `EndTracking` method would erase the rectangle from its old location and redraw it in the new location.

If you want to make your mouse task undoable, you need to store enough information in the object to undo the effects of mouse tracking. You must

also override the Undo method (inherited from CTask) to use this information to undo the effects of the mouse task.

Using the moving rectangle example again, your EndTracking method might keep the old location of the rectangle in an instance variable. The Undo method would erase the rectangle from its current location and redraw it again in the old location.

### Using the mouse task

You use mouse tasks in the DoClick method of a pane. Create an instance of your mouse task, initialize it, and pass it in a TrackMouse message to your pane. This is what part of your DoClick method might look like in THINK Pascal:

```
var
  myMover: CShapeMover;
begin
  new(myMover );
  myMover.IShapeMover(UNDOMoverStr);

  TrackMouse(myMover, hitPt, pinRect);
  itsSupervisor.Notify(myMover);
  ...
end
```

And this is what part of your DoClick method might look like in THINK C:

```
CShapeMover *myMover;

myMover = new(CShapeMover);
myMover->IShapeMover(UNDOMoverStr);

this->TrackMouse(myMover, hitPt, &pinRect);
itsSupervisor->Notify(myMover);
```

The TrackMouse method sends your mouse task a BeginTracking message to give you an opportunity to adjust the starting point. As long as the mouse button is down, TrackMouse repeatedly sends KeepTracking messages to your mouse task. When the mouse button is released, TrackMouse sends your mouse task an EndTracking message.

The value you pass to your initialization method is the index of a string in the STR# 130 resource that describes your task. The document method UpdateUndo uses this string for the wording of the **Undo** command in the **Edit** menu. If your task is not undoable, you can use any value and ignore it.



After you've tracked the mouse, you can send the task in a `Notify` message to the document. The document will store the task in the document's `lastTask` instance variable. When you choose **Undo** from the **Edit** menu, the document sends an Undo message to the task to undo the effects of mouse tracking.

## Variables

This class has no instance variables.

## Methods

### Construction and destruction methods

#### **IMouseTask**

```
procedure IMouseTask (aNameIndex: integer);
void IMouseTask (short aNameIndex);
```

Initialize a mouse tracking task. `ANameIndex` is the index for the undo/redo string in the `STR# 130` resource.

### Mouse tracking methods

#### **BeginTracking**

```
procedure BeginTracking (var startPt: LongPt);
void BeginTracking (LongPt *startPt);
```

Mouse tracking is starting. The document's `TrackMouse` method sends this message at the beginning of mouse tracking. `StartPt` is the starting point (in local coordinates). If your mouse task subclass doesn't alter the starting point, you don't need to override this method.

#### **KeepTracking**

```
procedure KeepTracking (var currPt, prevPt, startPt:
    LongPt);
void KeepTracking (LongPt *currPt, LongPt *prevPt,
    LongPt *startPt);
```

Mouse tracking is under way. The document's `TrackMouse` method sends this message repeatedly as long as the mouse button is down. `CurrPt` is the current mouse location. `PrevPt` is the previous mouse location. `StartPt` is the original mouse location. Your mouse task subclass must override this method.

#### **EndTracking**

```
procedure EndTracking (var currPt, prevPt, startPt:
    LongPt);
void EndTracking (LongPt *currPt, LongPt *prevPt,
    LongPt *startPt);
```

Mouse tracking is over. The document's `TrackMouse` method sends this message when the mouse button is released. `CurrPt` is the current mouse location. `PrevPt` is the previous mouse location. `StartPt` is the original

## ◆ 41 CMouseTask

---

mouse location. Your mouse task subclass must override this method. If your mouse task is undoable, you should store all the information you need to undo it in this method.

---

### Note

If you're implementing an undoable mouse task, you'll need to override the `Undo` method as well.

---

## Class resources

### Resource

STR# 130

### Description

List of strings that describe undoable tasks. For example, if you're implementing a mouse task that moves graphic images, the string for that task might be "move." The item in the **Edit** menu would then read "Undo move" or "Redo move."



# CObject ♦ 42

---

## Introduction

CObject is the abstract root level class. It is the ancestor of all the classes in the THINK Class Library.

## Heritage

Superclass	None
Subclasses	CAppleEvent CBartender CBitMap CChore CCollaborator CDecorator CEnvironment CError CFile CMenuDefProc CPaneBorder CPrinter CSwitchboard CTask

## Using CObject

Every class in the THINK Class Library is a descendant of this class. CObject is the only class with no superclass. If you need to create a new class that can't be a subclass of any other class, make it a subclass of CObject.

## Variables

This class has no instance variables.

## Methods

All classes inherit these methods:

## ◆ 42 CObject

---

<b>Free/Dispose</b>	<b>Creation and destruction</b> procedure Free; void Dispose (void); Dispose of the memory an object occupies. If your subclass allocates memory in its initialization method, be sure you release it in this method.
<b>Clone/Copy</b>	function Clone: CObject; CObject *Copy (void); Make a copy of the object. Note that if a class has instance variables that point to a block of memory, only the reference is copied, not the contents of the block of memory.
<b>Lock</b>	function Lock (fLock: Boolean): Boolean; Boolean Lock (Boolean fLock) When fLock is TRUE, make this object non-relocatable. When fLock is FALSE, make it relocatable. This method returns TRUE if the object was locked.
<b>SubclassResponsibility</b>	procedure SubclassResponsibility; void SubclassResponsibility (void); In some cases, certain methods inherited from abstract classes must be overridden, or the subclass won't work correctly. Most of these methods call SubclassResponsibility rather than leaving the method empty. When the symbol <code>__TCL_DEBUG__</code> is defined, this method displays a debugging message.
<b>GetClassName</b>	procedure GetClassName (var className: Str255); void GetClassName (Str255 className); Place the name of this class in className.

# CPane ♦

## 43

### Introduction

CPane is an abstract class that defines a drawing area within a window or within another pane. In the THINK Class Library, all drawing is done within a pane. Each pane has its own drawing environment and coordinate system.

### Heritage

Superclass	CView
Subclasses	CControl
	CPanorama
	CRadioGroupPane
	CScrollPane
	CSizeBox

### Using CPane

Use a pane when you need a non-scrolling area to draw in within a window. If you need a scrollable area, use CPanorama and CScrollPane. If you need a pane for displaying text, see CAbstractText or CEditText.

Your application must define a subclass of CPane (or one of its descendants) to draw in a window. In your subclass, you need to override or define these methods:

<i>initialization method</i>	Free
Draw	DoClick

Your pane class should have an initialization method. If your subclass defines new instance variables, this is the method that sets them up. By convention, the name of your initialization method should be `1 YourPane` where *YourPane* is the name of your pane class. Your initialization method should call `IPane`. The supervisor of a pane should be either the pane that encloses it or the director its window belongs to.

The pane initialization method is where you set the pane's location in its enclosure and its characteristics. If you want your pane, or any of the panes it encloses, to receive clicks, be sure to send the pane a `SetWantsClicks(true)` message, otherwise mouse clicks in your pane are ignored.

If your pane allocates memory, you should also override the `Free` method to deallocate it. Be sure that your method calls inherited `Free` to make sure that the pane is disposed of properly.

The `Draw` message tells your pane to draw its contents. You can assume that the port, clip region, and coordinate system have been set up correctly. If your pane uses long coordinates, you need to map from frame coordinates to QuickDraw coordinates before you do any drawing.

When the user clicks in your pane, it gets a `DoClick` message. Your `DoClick` method can either handle the mouse click itself, or it can create a task and send it in a `TrackMouse` message. To learn more about mouse tracking in a pane, see `CMouseDownTask` on page 309.

### Coordinate Systems in Panes

Panes use two of the coordinate systems in the THINK Class Library: Frame coordinates and QuickDraw Coordinates.

Frame coordinates provide a local coordinate system for a pane. Units in frame coordinates are in pixels, and the point (0, 0) is usually the top, left corner of the pane. If the pane moves within its enclosure, the coordinate system does not change; the top left corner is still (0, 0). The only time this origin point changes is when you scroll the pane. Each pane can choose to use long coordinates or short coordinates.

All drawing and mouse tracking is done in QuickDraw coordinates. This is the coordinate system that the Macintosh Toolbox uses for its drawing operations. QuickDraw coordinates are only valid after a call to `Prepare`. The relationship between QuickDraw coordinates and the other coordinate systems depends on whether a pane is using long frame coordinates or short frame coordinates.

Short coordinates map directly to QuickDraw coordinates. Each element in a short coordinate uses 16-bit values, so a pane that uses short coordinates is limited to the rectangle (-32768, -32768, 32767, 32767). Long coordinates layer a 32-bit coordinate system on top of the QuickDraw 16-bit coordinates. The long coordinate system lets you use a much larger coordinate area for your pane. Since all drawing takes place in QuickDraw coordinates you

have to map the long coordinates to QuickDraw coordinates when you draw in a pane.

If a pane uses long coordinates, the instance variable `fUsingLongCoord`, which `CPane` inherits from `CView` is `TRUE`. Otherwise, `fUsingLongCoord` is `FALSE`. The default is to not use long coordinates.

The THINK Class Library uses the types `LongRect` and `LongPt` for both long and short coordinates. You'll notice that most of the descendants of `CView` that work with points and rectangles use these types.

---

**Note**

If you've worked with earlier versions of the THINK Class Library, this is the biggest change you'll notice since it will require some changes to your programs.

---

These two types are defined like this in `LongCoordinates.h` in C:

```
typedef struct LongPt
{
    long v, h;
} LongPt;

typedef struct LongRect
{
    long top, left, bottom, right;
} LongRect;
```

And like this in `TCL.p` in Pascal:

```
type
    LongPt = record
        case integer of
            1: (
                v, h: longint
            );
            2: (
                vh: array[VHSelect] of longint
            );
        end;
```

```

LongRect = record
  case integer of
    1: (
      top, left,
      bottom, right: longint
    );
    2: (
      topLeft: LongPt;
      botRight: LongPt
    );
  end;

```

If the pane is using short coordinates, frame coordinates and QuickDraw coordinates are identical, so the values stored in a LongRect or in a LongPt are in QuickDraw coordinates. To use them with QuickDraw routines, however, you'll need to convert them to the QuickDraw types Rect and Point. The THINK Class Library provides several utility routines to do these conversions. For a complete listing, see "Long Coordinate Utilities" on page 462.

### Drawing in Panes

To draw in a pane use the standard QuickDraw routines. The pane's Prepare method sets up the QuickDraw port. If your pane uses short coordinates, the coordinate system is set up correctly. If your pane uses long coordinates, you need to transform frame coordinates to QuickDraw coordinates before you draw. You can CPane methods FrameToQD and FrameToQDR convert frame points and rectangles to QuickDraw points and rectangles.

Here's an example. The following Draw method draws a line from the top, left of the pane to the bottom right, and then draws a rectangle inset 10 pixels from the edges of the frame.

In C it looks like this:

```

void CCoolPane::Draw (Rect *area)
{
    Rect theRect;

    MoveTo(frame.left, frame.top);
    LineTo(frame.right, frame.bottom);
    LongToQDRect(&frame, &theRect);
    InsetRect(&theRect, 10, 10);
    FrameRect(&theRect);
}

```

In Pascal, the same method looks like this:

```
procedure CCoolPane.Draw (var area: Rect);
var
    theRect: Rect;
begin
    MoveTo(frame.left, frame.top);
    LineTo(frame.right, frame.bottom);
    LongToQDRect (frame, theRect);
    InsetRect (theRect, 10, 10);
    FrameRect (theRect)
end;
```

Note that to draw the line from one corner to the other, you can use the values directly since they're QuickDraw values. But to perform an operation on a rectangle, you have to convert the LongRect into a QuickDraw Rect.

If a pane uses long coordinates, all coordinates are in frame coordinates. Before you can do any drawing, you need to map those values to the QuickDraw space on the screen. The CPane methods `FrameToQD` and `FrameToQDR` map frame coordinates into QuickDraw coordinates.

For example, imagine you have a pane that uses long coordinates and you want its `Draw` method to display the text "Hello World." and then draw a rectangle inset 10 pixels from the frame. The location of the string is in an instance variable called `stringLoc` which is declared as a `LongPt`. The location in `stringLoc` could be well outside QuickDraw coordinate space. It might be at (100000, 100000).

This is how you might write the `Draw` method in C:

```
void CMyLongPane::Draw (Rect *area)
{
    Point qdLoc;
    Rect qdRect;

    FrameToQD(&stringLoc, &qdLoc);
    MoveTo(qdLoc.h, qdLoc.v);
    DrawString("\pHello World.");
    FrameToQDR(&frame, &qdRect);
    InsetRect(&qdRect, 10, 10);
    FrameRect(&qdRect);
}
```

In Pascal it would look like this:

```
procedure CMyLongPane.Draw (var area: Rect);
var
    qdLoc: Point;
    qdRect: Rect;
begin
    FrameToQD(stringLoc, qdLoc);
    MoveTo(qdLoc.h, qdLoc.v);
    DrawString('Hello World. ');
    FrameToQDR(frame, qdRect);
    InsetRect(qdRect, 10, 10);
    FrameRect(qdRect)
end;
```

Keep in mind the difference between converting long-coordinate structures to QuickDraw structures and mapping long-coordinate structures to QuickDraw structures. When you're using short coordinates, frame coordinates are the same as QuickDraw coordinates, so all you have to do is translate on data structure to another. When you're using long coordinates, you need to map a portion of long coordinate space into QuickDraw space so you can draw in it.

### Variables

Variable	Type	Description
width	integer	Horizontal size in pixels.
height	integer	Vertical size in pixels.
hEncl	longint	Horizontal location in enclosure.
vEncl	longint	Vertical location in enclosure.
hSizing	SizingOption	Horizontal sizing option.
vSizing	SizingOption	Vertical sizing option.
autoRefresh	Boolean	Refresh after a re-size?
frame	LongRect	Area for displaying the pane which de-





		finest the frame coordinates.
aperture	LongRect	Active drawing area of the pane.
hOrigin	longint	Window left in frame coordinates.
vOrigin	longint	Window top in frame coordinates.
itsEnvironment	CEnvironment	Drawing environment.
printClip	ClipOption	The region to clip to when printing.
printing	Boolean	Is printing in progress?
itsBorder	CPaneBorder	Border of this pane.

## Methods

### Construction/Destruction methods

#### IPane

```
procedure IPane (anEnclosure: CView;
  aSupervisor: CBureaucrat;
  aWidth, aHeight: integer;
  aHEncl, aVEncl: integer;
  aHSizing, aVSizing: SizingOption);
```

```
void IPane (CView *anEnclosure,
  CBureaucrat *aSupervisor,
  short aWidth, short aHeight,
  short aHEncl, short aVEncl,
  SizingOption aHSizing, SizingOption aVSizing);
```

Initialize a pane. Almost all of the descendants of CPane use the same arguments in their initialization methods.

AnEnclosure is the enclosing view that contains this pane. Typically the enclosure is either a window or another pane. If your pane is a panorama, its enclosure should be a scroll pane.

ASupervisor is the bureaucrat that handles all the commands that this pane won't. Typically, the supervisor is the document (or director) associated with this pane's window.

AWidth and aHeight are the width and height of the pane in pixels. AHEncl and aVEncl are the horizontal and vertical position of the pane within its enclosure.

The `aHSizing` and `aVSizing` parameters specify what happens to the pane when the size of its enclosure changes. The length and height of a pane changes relative to its original position in the enclosing pane. These are the values that `aHSizing` can have:

<b>aHSizing value</b>	<b>Meaning</b>
<code>sizFIXEDLEFT</code>	The left edge of the pane is always the same number of pixels from the left edge of the enclosing pane as when it was originally placed.
<code>sizFIXEDRIGHT</code>	The right edge of the pane is always the same number of pixels from the right edge of the enclosing pane as when it was originally placed.
<code>sizFIXEDSTICKY</code>	The left and right edges stick to their original locations in the enclosing pane. If the enclosure scrolls horizontally, the pane will scroll with it.
<code>sizELASTIC</code>	The width of the pane grows and shrinks by the same amount as the enclosing pane.

These are the values the `aVSizing` can have:

<b>aVSizing value</b>	<b>Meaning</b>
<code>sizFIXEDTOP</code>	The top edge of the pane is always the same number of pixels from the top edge of the enclosing pane as when it was originally placed.
<code>sizFIXEDBOTTOM</code>	The bottom edge of the pane is always the same number of pixels from the bottom edge of the enclosing pane as when it was originally placed.
<code>sizFIXEDSTICKY</code>	The top and bottom edges stick to their original locations in the enclosing pane. If the enclosure scrolls vertically, the pane will scroll with it.
<code>sizELASTIC</code>	The height of the pane grows and shrinks by the same amount as the enclosing pane.

A couple of examples will make this clearer: A vertical scroll bar in a window would be horizontally `sizFIXEDRIGHT` and vertically `sizELASTIC`. It has a fixed horizontal length and remains anchored to the right edge of the



*You may want to look at Figure 7-5 on page 83.*

window. Vertically, it stretches and contracts with the height of the window. A status box in the lower left corner of a window would be `szFIXEDLEFT` horizontally and `szFIXEDBOTTOM` vertically. It has a constant size and remains anchored to the bottom left corner of the window.

## **IViewRes**

```
procedure IViewRes (rType: ResType; resID: integer;
    anEnclosure: CView; aSupervisor: CBureaucrat);
void IViewRes (ResType rType, short resID,
    CView *anEnclosure, CBureaucrat *aSupervisor);
```

Initialize a pane from a resource template. `RType` is the resource type for the `CView` subclass you want to initialize. `ResID` is the resource ID of the resource. `AnEnclosure` and `aSupervisor` are the same as for `IBorder`. This method is inherited from `CView`.

To initialize a pane from a resource file, use a 'Pane' resource.

## **IViewTemp**

```
procedure IViewTemp (anEnclosure: CView;
    aSupervisor: CBureaucrat; viewData: Ptr);
void IViewTemp (CView *anEnclosure,
    CBureaucrat *aSupervisor, Ptr viewData);
```

This method is used internally for initializing from a resource template. Each subclass of `CView` overrides this method to use its own resource template.

## **Free/Dispose**

```
procedure Free;
void Dispose (void);
```

Dispose of a pane. If you create a subclass of class `CPane`, be sure your class's `Free` or `Dispose` method calls the inherited method as well.

### **Accessing methods**

## **SetFrameOrigin**

```
procedure SetFrameOrigin (fLeft, fTop: longint);
void SetFrameOrigin (long fLeft, long fTop);
```

Set the coordinates of the top left corner of the pane's frame to determine the frame coordinates of the pane.

## **GetFrame**

```
procedure GetFrame (var theFrame: LongRect);
void GetFrame (LongRect *theFrame);
```

Get the frame of the pane. The frame is the rectangle that encloses the pane in pane coordinates. Usually the top and left of this rectangle are 0,0.

<b>GetLengths</b>	<pre>procedure GetLengths (var theWidth, theHeight:     integer); void GetLengths (short *theWidth, short *theHeight);</pre> <p>Get the width and the height of the pane.</p>
<b>GetOrigin</b>	<pre>procedure GetOrigin (var theHOrigin, theVOrigin:     longint); void GetOrigin (long *theHOrigin, long *theVOrigin);</pre> <p>Get the origin of a pane. The origin of a pane is the top left of the window in frame coordinates. It's the distance from the top left of the window to the (0,0) of the pane. You should not use or override this method.</p>
<b>GetAperture</b>	<pre>procedure GetAperture (var theAperture: LongRect); void GetAperture (LongRect *theAperture);</pre> <p>Get the aperture of the pane. The aperture is the visible portion of a pane where drawing can occur. The aperture is given in frame coordinates.</p>
<b>Contains</b>	<pre>function Contains (windPt: Point): Boolean; Boolean Contains (Point windPt);</pre> <p>Returns TRUE if windPt is in the aperture of the pane. WindPt is given in window coordinates.</p>
<b>ReallyVisible</b>	<pre>function ReallyVisible: Boolean; Boolean ReallyVisible (void);</pre> <p>Return TRUE if the pane is visible and within QuickDraw space. This method actually returns TRUE if the pane and its enclosure is <i>potentially</i> visible. The pane may not actually be on the screen. A pane is on the screen if it's ReallyVisible and the aperture isn't empty.</p>
<b>GetPixelExtent</b>	<pre>procedure GetPixelExtent (var hExtent,     vExtent: longint); void GetPixelExtent (long *hExtent, long *vExtent);</pre> <p>Get the dimensions of the pane in pixels.</p>

**SetPrintClip**

```
procedure SetPrintClip (aPrintClip: ClipOption);
void SetPrintClip (ClipOption aPrintClip);
```

Specify the print option to use when printing. APrintClip can be one of

<b>aPrintClip value</b>	<b>Meaning</b>
clipAPERTURE	Print only what is visible
clipFRAME	Print the contents of the frame
clipPAGE	Print to fill the page

The default is clipFRAME.

**GetWindow**

```
function GetWindow: CWindow;
CWindow *GetWindow (void);
```

Return the window that encloses the pane. Note that this method returns a window object, not a Macintosh window.

**SetBorder**

```
procedure SetBorder (aBorder: CPaneBorder);
void SetBorder (CPaneBorder *aBorder)
```

Set the border for this pane. For more information about borders, see the description of CPaneBorder on page 335.

**SetResBorder**

```
procedure SetResBorder (resID: integer);
void SetResBorder (short resID)
```

Create a border for this pane from a 'PBrd' resource whose ID is resID.

**GetBorder**

```
function GetBorder: CPaneBorder;
CPaneBorder *GetBorder (void)
```

Return the border for this pane.

**GetHelpResID**

```
function GetHelpResID: integer;
short GetHelpResID (void)
```

Return the ID of the 'hrct' resource that has the Balloon Help information for this pane. The resource ID is actually stored in the window object that encloses this pane. For more information about help resources for panes see "Using Balloon Help with views" on page 436.

**Appearance methods****Show**

```
procedure Show;
void Show (void);
```

Show the pane if it was hidden. The default method sends a Refresh message to the pane after making it visible.

### Hide

```
procedure Hide;
void Hide (void);
```

Hide the pane if it was visible. The default method sends a Refresh message to the pane before hiding it.

### Size and location methods

### Place

```
procedure Place (hEncl, vEncl: longint;
    redraw: Boolean);
void Place (long hEncl, long vEncl, Boolean redraw);
```

Place the pane at the point hEncl, vEncl of its enclosure. hEncl and vEncl are given in the coordinate system of the enclosure. If redraw is TRUE, redraw the pane after moving it.

### Offset

```
procedure Offset (hOffset, vOffset: longint;
    redraw: Boolean);
void Offset (long hOffset, long vOffset,
    Boolean redraw);
```

Offset the pane by hOffset pixels horizontally and vOffset pixels vertically. If redraw is TRUE, redraw the pane after moving it.

### ChangeSize

```
procedure ChangeSize (delta: Rect; redraw: Boolean);
void ChangeSize (Rect *delta, Boolean redraw);
```

Change the size of the pane. The values of each field of the delta rectangle specify how each side should change. Positive values mean down and to the right. Negative values mean up and to the left.

### AdjustToEnclosure

```
procedure AdjustToEnclosure (deltaEncl: Rect);
void AdjustToEnclosure (Rect *deltaEncl);
```

Adjust the size or location of the pane when the enclosure has moved or changed size. You should not override or use this method.

### AdjustHoriz

```
procedure AdjustHoriz (deltaEncl: Rect;
    var delta: Rect; var offset: integer;
    var moved: Boolean; var sized: Boolean);
void AdjustHoriz (Rect *deltaEncl, Rect *delta,
    short *offset, Boolean *moved, Boolean *sized);
```

Adjust the horizontal size or location of a pane. You should not override or use this method.

### AdjustVert

```
procedure AdjustVert (deltaEncl: Rect;
    var delta: Rect; var offset: integer;
    var moved: Boolean; var sized: Boolean);
```



```
void AdjustVert (Rect *deltaEncl, Rect *delta,
    short *offset, Boolean *moved, Boolean *sized);
```

Adjust the vertical size or location of a pane. You should not override or use this method.

### EnclosureScrolled

```
procedure EnclosureScrolled (hOffset, vOffset:
    longint);
```

```
void EnclosureScrolled (long hOffset, long vOffset);
```

Adjust the location of a pane after its location has scrolled. This method affects the pane only if it is sticky in the direction it is being moved. You should not use or override this method.

### Adapting methods

### FitToEnclosure

```
procedure FitToEnclosure (horizFit, vertFit: Boolean);
```

```
void FitToEnclosure (Boolean horizFit,
    Boolean vertFit);
```

Make the frame of the pane fit the interior of its enclosure in either the vertical or horizontal direction. If `horizFit` is `TRUE`, the left edge and width of the pane's frame change to coincide with the enclosure's interior. If `vertFit` is `TRUE`, the top edge and height of the pane's frame change to coincide with the enclosure's interior. `FitToEnclosure` sends the enclosure a `Get Interior` message to determine the interior of the pane.

`FitToEnclosure` does not redraw the pane.

### FitToEnclFrame

```
procedure FitToEnclFrame (horizFit, vertFit: Boolean);
```

```
void FitToEnclFrame (Boolean horizFit,
    Boolean vertFit);
```

Fit the frame of the pane to the frame of its enclosure in either the vertical or horizontal direction. If `horizFit` is `TRUE`, the left edge and the width of the pane's frame change to coincide with the enclosure's frame. If `vertFit` is `TRUE`, the top edge and the height of the pane's frame change to coincide with the enclosure's frame. `FitToEnclFrame` does not redraw the pane.

### CenterWithinEnclosure

```
procedure CenterWithinEnclosure (horizCenter,
    vertCenter: Boolean);
```

```
void CenterWithinEnclosure (Boolean horizCenter,
    Boolean vertCenter);
```

Center the pane within its enclosure horizontally or vertically. Only the location of the pane changes. The size of the pane does not change.

`CenterWithinEnclosure` does not redraw the pane.

### Drawing methods

#### Draw

```
procedure Draw (area: Rect);  
void Draw (Rect *area);
```

Draw the contents of the pane. The `area` parameter specifies the portion of the pane that needs to be redrawn. Your subclass must override this method.

If the pane is not using long coordinates, the `area` parameter is given in frame coordinates, which in this case are the same as QuickDraw coordinates.

If the pane uses long coordinates, `area` is given in QuickDraw coordinates. Your `Draw` method can use `QDToFrame` or `QDToFrameR` to convert from QuickDraw coordinates to frame coordinates and `FrameToQD` and `FrameToQDR` to convert from frame coordinates to QuickDraw coordinates.

#### DrawAll

```
procedure DrawAll (var area: Rect);  
void DrawAll (Rect *area);
```

Draw the pane and all its subviews and any borders that the pane or subviews may have. Use this method when you want to force the entire pane to be redrawn without waiting for an update event. Scrolling is a good example. You want to redraw the pane as soon as it has scrolled instead of waiting for an update event. This method prepares the pane before sending Draw messages to it and to its subpanes. You should not override this method. Look at the implementation of `CPanorama's Scroll` method for an example of using `DrawAll`.

#### Refresh

```
procedure Refresh;  
void Refresh (void);
```

Force the pane to redraw itself on the next update event. Default method sends a `RefreshLongRect` message to the pane using the frame as the `area`.

#### RefreshRect

```
procedure RefreshRect (area: Rect);  
void RefreshRect (Rect *area);
```

Force a portion of the pane to redraw itself on the next update event. `Area` is a `Rect` in frame coordinates. Only the portion of the pane that's actually visible will actually be redrawn.



**RefreshLongRect**

```
procedure RefreshLongRect (area: LongRect);  
void RefreshLongRect (LongRect *area);
```

Force a portion of the pane to redraw itself on the next update event. Area is a LongRect in frame coordinates. Only the portion of the pane that's actually visible will actually be redrawn.

**RefreshBorder**

```
procedure RefreshBorder;  
void RefreshBorder (void);
```

Force the pane's border to redraw itself on the next update event.

**Printing methods****Paginate**

```
procedure Paginate (aPrinter: CPrinter;  
    pageWidth, pageHeight: integer);  
void Paginate (CPrinter *aPrinter, short pageWidth,  
    short pageHeight);
```

Set up the pane's pagination. By default, panes print only on one page. The document that a pane belongs to sends this message to the pane. For examples more complex pagination, see CAbstractText and CPanorama.

**AboutToPrint**

```
procedure AboutToPrint (var firstPage, lastPage:  
    integer);  
void AboutToPrint (short *firstPage, short *lastPage);
```

Printing is about to begin. Your pane subclass can override this method to do anything that needs to happen right before printing.

**PrintPage**

```
procedure PrintPage (pageNum, pageWidth, pageHeight:  
    integer; aPrinter: CPrinter);  
void PrintPage (short pageNum, short pageWidth,  
    short pageHeight, CPrinter *aPrinter);
```

Print the specified page of the pane. By default, panes have only on page. APrinter is usually supplied by the document that this pane belongs to. If your pane subclass supports multi-page documents, you must override this method to determine which part of the pane to draw.

**DonePrinting**

```
procedure DonePrinting;  
void DonePrinting (void);
```

Printing is over. If you want to take any action when printing is done, override this method.

### **PrepareToPrint**

```
procedure PrepareToPrint;  
void PrepareToPrint (void);
```

Set up the coordinate system and the clipping region before printing. The default method sets the clipping region to the region described in the `printClip` variable. You can set this variable with the `SetPrintClip` method.

### **Prepare**

#### **Calibration methods**

```
procedure Prepare;  
void Prepare (void);
```

Prepare the pane for drawing. `Prepare` sets up the port and the QuickDraw coordinates for the pane, converting `lon` coordinates to QuickDraw coordinates if necessary. It also sets the clipping region to the aperture (the visible portion of the pane) so drawing is constrained to the visible portion of the pane. If the pane has an environment associated with it, `Prepare` sends a `Restore` message to the environment. If the pane is being printed, the default method sends the pane a `PrepareToPrint` message instead.

The inherited `Prepare` method in `CView` sets `cPreparedView` to this view. If this view gets another `Prepare` message, `Prepare` avoids setting up the drawing environment all over again.

### **RestoreEnvironment**

```
procedure RestoreEnvironment;  
void RestoreEnvironment (void);
```

Restore the pane's drawing environment. If the pane has a drawing environment associated with it, this method sends a `Restore` message to it.

### **CalcFrame**

```
procedure CalcFrame;  
void CalcFrame (void);
```

Calculate the coordinates of the pane's frame based on the frame's width and height and its location within its enclosure. Generally, the superclasses of the pane classes you define will send this message. You should not use or override this method unless your pane subclass needs something other than (0,0) at its top left corner

### **ResizeFrame**

```
procedure ResizeFrame (delta: Rect);  
void ResizeFrame (Rect *delta);
```

Adjust the frame when the size of the pane changes. The `delta` rectangle specifies how each side changes. Positive values mean down and to the right. Negative values mean up and to the left. The default method always sets the top left of the frame to (0,0)



You should not need to use or override this method unless your pane subclass requires that the top, left of the frame be something other than (0,0). See the implementation of `ResizeFrame` in `CPanorama` for an example.

## CalcAperture

```
procedure CalcAperture;
void CalcAperture (void);
```

Calculate the visible (or drawable) portion of a pane. The aperture of a pane is the area that is not obscured by the bounds of the enclosing view. You should not use or override this method. To get the aperture, use `GetAperture` on page 324.

## Cursor method

## TrackMouse

```
procedure TrackMouse (theTask: CMouseEvent;
    startPt: LongPt; var pinRect: LongRect);
void TrackMouse (CMouseTask *theTask, LongPt startPt,
    LongRect *pinRect);
```

*To learn more about mouse tracking and mouse tasks, see the class `CMouseTask` on page 309.*

Track the mouse in this view. TheTask is a mouse task that you create. StartPt is the starting point in frame coordinates. PinRect is a constraining rectangle in frame coordinates. This message is usually sent from a `DoClick` method.

The default method does several things:

- Sends a `Prepare` message to the view
- Sends a `BeginTracking` message to theTask
- Sends a `KeepTracking` message to theTask as long as the mouse is down. The current point is constrained to pinRect. (Your task subclass must override the `KeepTracking` method.)
- Sends an `EndTracking` message to theTask. (Your task subclass must override this method as well.)

## Coordinate transformation methods

## WindToFrame

```
procedure WindToFrame (windPt: Point;
    var framePt: LongPt);
void WindToFrame (Point windPt, LongPt *framePt);
```

Convert the point windPt from window coordinates to frame coordinates and place the converted point in framePt.

### WindToFrameR

```
procedure WindToFrameR (windRect: Rect,
    var frameRect: LongRect);
void WindToFrameR (Rect *windRect,
    LongRect *frameRect);
```

Convert the rectangle `windRect` from window coordinates to frame coordinates and place the converted rectangle in `frameRect`.

### FrameToWind

```
procedure FrameToWind (framePt: LongPt;
    var windPt: Point);
void FrameToWind (LongPt *framePt, Point *windPt);
```

Convert the point `framePt` from frame coordinates to window coordinates and place the converted point in `windPt`.

### FrameToWindR

```
procedure FrameToWindR (frameRect: LongRect;
    var windRect: Rect);
void FrameToWindR (LongRect *frameRect,
    Rect *windRect);
```

Convert the rectangle `frameRect` from frame coordinates to window coordinates and place the converted rectangle in `windRect`.

### EnclToFrame

```
procedure EnclToFrame (var thePoint: LongPt);
void EnclToFrame (LongPt *thePoint);
```

Convert a point from the frame coordinates of its enclosure to the frame coordinates of the pane.

### EnclToFrameR

```
procedure EnclToFrameR (var theRect: LongRect);
void EnclToFrameR (LongRect *theRect);
```

Convert a rectangle from the frame coordinates of its enclosure to the frame coordinates of the pane.

### FrameToEncl

```
procedure FrameToEncl (var thePoint: LongPt);
void FrameToEncl (LongPt *thePoint);
```

Convert a point from frame coordinates to the frame coordinates of its enclosure.

### FrameToEnclR

```
procedure FrameToEnclR (var theRect: LongRect);
void FrameToEnclR (LongRect *theRect);
```

Convert a rectangle from frame coordinates to the frame coordinates of its enclosure.

**FrameToGlobalR**

```
procedure FrameToGlobalR (frameRect: LongRect;  
    globalRect: Rect);  
void FrameToGlobalR (LongRect *frameRect,  
    Rect *globalRect);
```

Convert the rectangle `frameRect` from frame coordinates to global coordinates and place the converted rectangle in `globalRect`.

**QDToFrame**

```
procedure QDToFrame (qdPoint: Point;  
    var framePt: LongPt);  
void QDToFrame (Point qdPoint, LongPt *framePt);
```

Convert `qdPoint` from QuickDraw coordinates to frame coordinates, and put the result in `framePt`. `qdPoint` is assumed to be in the portion of QuickDraw space that the pane is mapping to. If the pane is not using long coordinates, the values in `qdPoint` and `framePt` are the same.

**QDToFrameR**

```
procedure QDToFrameR (qdRect: Rect;  
    var frameRect: LongRect);  
void QDToFrameR (Rect *qdRect, LongRect *frameRect);
```

Convert `qdRect` from QuickDraw coordinates to frame coordinates, and put the result in `frameRect`.

**FrameToQD**

```
procedure FrameToQD (framePt: LongPt;  
    var qdPt: Point);  
void FrameToQD (LongPt *framePt, *Point qdPt);
```

Convert `framePt` from frame coordinates to QuickDraw coordinates, and put the results in `qdPt`.

**FrameToQDR**

```
procedure FrameToQDR (frameRect: LongRect;  
    var qdRect: Rect);  
void FrameToQDR (LongRect *frameRect, Rect *qdRect);
```

Convert `frameRect` from frame coordinates to QuickDraw coordinates, and put the result in `qdRect`.

**SectAperture**

```
function SectAperture (srcRect: LongRect;  
    var destRect: Rect): Boolean;  
Boolean SectAperture (LongRect *srcRect,  
    Rect *destRect);
```

Clip the `srcRect` (in frame coordinate) to the pixels visible through the aperture, and return the result in a QuickDraw rectangle, `destRect`. If `destRect` is not empty, this method returns TRUE.

## ◆ 43 *CPane*

---

# *CPaneBorder* ◆

## 44

---

### Introduction

CPaneBorder is a class that draws a border around a pane.

### Heritage

Superclass	CObject
Subclasses	None

### Using CPaneBorder

CPaneBorder draws a border around a pane. You can choose from several types, including a rectangle, a rectangle with a shadow, and a rounded rectangle. If you need a different type, create a subclass of CPaneBorder. This class doesn't use any coordinate system. Its parameters are described as offsets from the pane's frame.

To give a pane a border, create an instance of CPaneBorder and send the pane a `SetBorder` message, with your border as the argument. The pane takes care of drawing it. When the size of the pane changes, the size of the border changes automatically. If you want to access the border to change it, send the pane a `GetBorder` message.

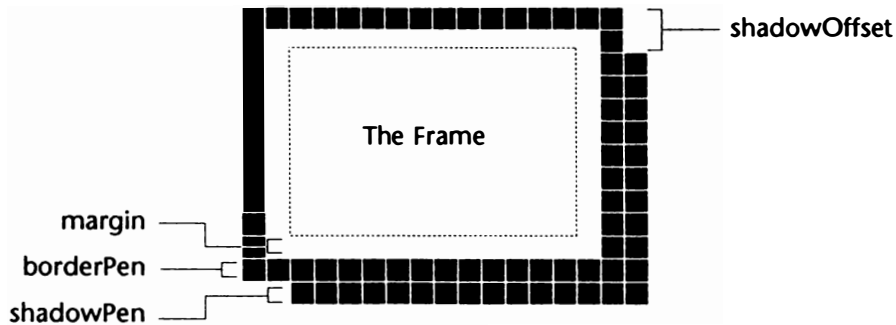
When you initialize the border with `IPaneBorder`, you set its shape. These are the available shapes:

Name	Description
<code>kBorderFrame</code>	Rectangle
<code>kBorderRoundRect</code>	Rounded rectangle
<code>kBorderOval</code>	Oval
<code>kBorderLeft</code>	Line on left side
<code>kBorderTop</code>	Line on top side
<code>kBorderRight</code>	Line on right side
<code>kBorderBottom</code>	Line on bottom
<code>kBorderNone</code>	No border

## 44 CPaneBorder

You can add together `kBorderLeft`, `kBorderTop`, `kBorderRight`, and `kBorderBottom` to create new shapes. For example, `kBorderTop + kBorderRight` gives you a border with lines on the top and right of the pane. To create yet more shapes, you need to subclass `CPaneBorder`.

After you initialize your border, you can change its attributes, like the width of the border outline, the distance between the border outline and the frame, and whether there's a shadow. Figure 44-1 shows you the instance variables that control these attributes:



**Figure 44-1** A pane border

This table describes what these instance variables control and the methods to set them:

Variable	Method	Description
margin	SetMargin	Distance between the frame and the border.
borderPen	SetPenSize	Width of the pen used to draw the border.
doShadow	SetShadow	TRUE, if this border has a shadow.
shadowPen	SetShadow	Width of the pen used to draw the shadow.
shadowOffset	SetShadow	Distance between the border and its shadow.





This table describes other instance variables that control the border's appearance:

Variable	Method	Description
<code>borderFlags</code>	<code>SetBorderFlags</code>	The shape of the border.
<code>penpat</code>	<code>SetPattern</code>	The pattern used to draw the border and shadow.
<code>roundDiameter</code>	<code>SetRounding</code>	The roundness of the border's corners, if the shape is a rounded rectangle.

## Variables

These are internal instance variables. You can access them with the accessing methods described below. Only subclasses of `CPaneBorder` should refer to them directly. In THINK C, the instance variables are protected.

Variable	Type	Description
<code>borderFlags</code>	<code>long</code>	The shape of the border.
<code>borderPen</code>	<code>Point</code>	The width of the pen used to draw the border.
<code>shadowOffset</code>	<code>Point</code>	The distance between the border and its shadow.
<code>shadowPen</code>	<code>Point</code>	The width of the pen used to draw the shadow.
<code>doShadow</code>	<code>Boolean</code>	TRUE if shadow is drawn.
<code>roundDiameter</code>	<code>Point</code>	The roundness of the border's corners, if the shape is a rounded rectangle.
<code>penPat</code>	<code>Pattern</code>	The pattern used to draw the border and shadow..
<code>margin</code>	<code>Rect</code>	The distance between the frame and the border.

### Methods

#### Creation and destruction

##### IPaneBorder

```
procedure IPaneBorder (borderFlags: integer);
void IPaneBorder (short borderFlags);
```

Initialize a pane border. The borderFlags describe the shape of the border. It can be one of these: kBorderNone, kBorderLeft, kBorderTop, kBorderRight, kBorderBottom, kBorderOval, kBorderRoundRect, kBorderFrame. All the defaults may be changed via methods.

##### IResPaneBorder

```
procedure IResPaneBorder (resID: integer);
void IResPaneBorder (short resID);
```

Initialize a border from a 'PBrd' resource.

#### Accessing methods

##### SetPattern

```
procedure SetPattern (aPattern: Pattern);
void SetPattern (Pattern aPattern);
```

Set the pattern used for drawing the border's outline and its shadow.

##### GetPattern

```
procedure GetPattern (var aPattern: Pattern);
void GetPattern (Pattern aPattern);
```

Return the current pattern.

##### SetBorderFlags

```
procedure SetBorderFlags (aBorderFlags: integer);
void SetBorderFlags (short aBorderFlags);
```

Set the border flags. ABorderFlags can be one of these: kBorderNone, kBorderLeft, kBorderTop, kBorderRight, kBorderBottom, kBorderOval, kBorderRoundRect, kBorderFrame.

##### GetBorderFlags

```
function GetBorderFlags: longint;
long GetBorderFlags (void);
```

Return the current border flags.

##### SetPenSize

```
procedure SetPenSize (penWidth, penHeight: integer);
void SetPenSize (short penWidth, short penHeight);
```

Set the pen size used to draw the border.

##### GetPenSize

```
procedure GetPenSize (var penWidth, penHeight:
    integer);
void GetPenSize (short *penWidth; short *penHeight);
```

Return the pen size used to draw the border.

**SetShadow**

```
procedure SetShadow (hOffset, vOffset,
                    width, height: integer);
void SetShadow (short hOffset, short vOffset,
               short width, short height);
```

Set the shadow attributes. *hOffset* and *vOffset* specify the distance of the shadow from the border outline. *width* and *height* specify the pen size used to draw the shadow. This method also sets *doShadow* to TRUE.

**GetShadow**

```
function GetShadow (var hOffset, vOffset,
                   width, height: integer): longint;
long GetPenSize (short *hOffset, short *vOffset,
                short *width, short *height);
```

Return the current shadow attributes.

**SetRounding**

```
procedure SetRounding (hDiameter, vDiameter: integer);
void SetRounding (short hDiameter, short vDiameter);
```

Set the roundness of the corners of rounded rectangles. These are used as parameters to *FrameRoundRect*. This method also sets the border style to *kBorderRoundRect*, if that isn't the border style already.

**GetRounding**

```
procedure GetRounding (var hDiameter, vDiameter:
                      integer);
void GetRounding (short *hDiameter, short *vDiameter);
```

Returns the current rounding diameters.

**SetMargin**

```
procedure SetMargin (aMargin: Rect);
void SetMargin (Rect *aMargin);
```

Set the distance between the border outline and the frame. *aMargin* is a *Rect* with positive offsets for each side of the border. For example, to have a one pixel margin on all sides, use (1, 1, 1, 1).

**GetMargin**

```
procedure GetMargin (var aMargin: Rect);
void GetMargin (Margin *aMargin);
```

Return the current border margin.

**Drawing method****DrawBorder**

```
procedure DrawBorder (paneFrame: Rect);
void DrawBorder (Rect *paneFrame);
```

Draw the border. *PaneFrame* is the pane's frame in the pane's current coordinates.

## ◆ 44 *CPaneBorder*

---

### **Calibration method**

#### **CalcBorderRect**

```
procedure CalcBorderRect (var paneFrame: Rect);  
void CalcBorderRect (Rect *paneFrame);
```

Return the area for the frame and its border. `PaneFrame` is the pane's frame in the pane's current coordinates. This method adds to it the size of the border, including the margin and shadow, on all sides.

# CPaneMDEF

## 45

### Introduction

CPaneMDEF is an abstract class that lets you display any pane as a Macintosh Menu. The class includes methods for handling tear-off menus.

### Heritage

Superclass	CMenuDefProc
Subclasses	CSelectorMDEF

### Using CPaneMDEF

*CTearOffMenu is described on page 417.*

CPaneMDEF's superclass, CMenuDefProc, lets you write object-oriented menu definition procedures. This class and its descendants take that idea a step further to let you use THINK Class Library panes as menus. The pane that you use in a CPaneMDEF subclass should belong to a subclass of CTearOffMenu.

A tear-off menu is a window that appears to float above all the other windows in your application. In that environment, the pane is part of the visual hierarchy, and it will behave just like any other pane in a window.

When it appears as a custom menu, the pane isn't really in the visual hierarchy at all. As far as the pane is concerned, its enclosure is the floating window it appears in when it's torn off. When you draw a custom menu in your Draw method, you have to set up the QuickDraw environment by hand with the SetupQuickDraw method. This method changes the QuickDraw coordinate system so drawing takes place in pane coordinates. Be sure to call RestoreQuickDraw at the end of the Draw method.

Since this class doesn't have a ChooseItem method, you'll need to create a subclass of CPaneMDEF that handles item selection. The class CSelectorMDEF is a subclass of CPaneMDEF that lets you use selectors as menus. It's also a good class to study to learn how to write a ChooseItem

method. Your ChooseItem method should also use the SetupQuickDraw/RestoreQuickDraw methods.

### Variables

Variable	Type	Description
itsPane	CPane	The pane to display as a menu.
itsTearOffMenu	CTearOffMenu	The torn-off menu.
savePort	GrafPtr	Used internally to save the grafport.
saveClip	RgnHandle	Used internally to save the clipping region.

### Methods

#### Construction and destruction methods

##### IPaneMDEF

```
procedure IPaneMDEF (MDEFid: integer;
    aPane: CPane; aTearOffMenu: CTearOffMenu);
void IPaneMDEF (short MDEFid, CPane *aPane,
    CTearOffMenu *aTearOffMenu);
```

Initialize the pane MDEF. This method calls IMenuDefProc with the MDEFid. The instance variable itsPane is set to aPane, which is the pane that you want to display as a menu. ATearOffMenu is a tear-off menu object. You can pass NIL if this menu is not a tear-off menu. See the CTearOffMenu class on page 417.

##### DrawMenu

```
procedure DrawMenu (macMenu: MenuHandle;
    menuRect: Rect);
void DrawMenu (MenuHandle macMenu, Rect *menuRect);
```

This method draws the pane as a menu. MacMenu and menuRect are provided by the Macintosh Menu Manager and are in global coordinates. DrawMenu uses the SetupQuickDraw method to set up the QuickDraw drawing environment so the point (0,0) is the top, left corner of the pane. The method then sends itsPane a RestoreEnvironment message and then a Draw message to display the pane. If the menu is disabled, DrawMenu grays out the menu. Finally, the method restores the QuickDraw environment.

Unless your subclass needs to do something extraordinary to draw your menu, you should not override this method.

**SizeMenu**

```
procedure SizeMenu (macMenu: MenuHandle);
void SizeMenu (MenuHandle macMenu);
```

This method is called by the Macintosh Menu Manager to determine the size of the menu. This method gets the height and width of the pane and stores them in `menuWidth` and `menuHeight` fields of `macMenu`. Your subclass should not override this method.

**TearOffMenu**

```
procedure TearOffMenu (menuRect: Rect;
  mouseLoc: Point);
```

```
void TearOffMenu (Rect *menuRect, Point mouseLoc);
```

*This method doesn't change the QuickDraw coordinates because it expects to be called from ChooseItem which has already set things up.*

`TearOffMenu` drags a gray outline representing the menu when you tear it off the menu bar. The size of the gray outline is the `menuRect` plus the margins specified in `itsTearOffMenu`. You can drag the outline as long as the mouse is still down, and the mouse is in the tear-off region. If the mouse is in the tear-off region when the button is released, `TearOffMenu` sends a `TornOff` message to `itsTearOffMenu` to do the "tearing."

Unless your subclass needs to do something fancy to show that a menu is being torn off, you should not override this method.

Your subclass should call this method in its `ChooseItem` method when the `hitPt` is not in the `menuRect`. You should also check to make sure that there is a tear-off menu (`itsTearOffMenu` is not `NIL`) and that the menu is not disabled.

*For a good example of a ChooseItem method, see the CSelector class.*

Here's what this might look like in Pascal:

```
procedure CMymDEF.ChooseItem(macMenu: MenuHandle
  menuRect: Rect; hitPt: Point;
  var whichItem: integer);
begin
  if PtInRect(hitPt, menuRect) then
    begin
      SetupQuickDraw(menuRect);
      { choose an item }
      RestoreQuickDraw;
    end
  else
    begin
      whichItem = NOTHING;
      if (itsTearOffMenu <> NIL) AND
        MenuEnabled(macMenu)
      then
        TearOffMenu(menuRect, hitPt)
      end
    end
end
```

In C it looks like this:

```
void CMyMDEF::ChooseItem(MenuHandle macMenu,
    Rect *menuRect, Point hitPt,
    short *whichItem)
{
    if (PtInRect(hitPt, menuRect) {
        SetupQuickDraw(menuRect);
        /* choose an item */
        RestoreQuickDraw();
    } else {
        *whichItem = NOTHING;
        if (itsTearOffMenu != NULL &&
            MenuEnabled(macMenu))
            TearOffMenu(menuRect, hitPt);
    }
}
```

### **PtInTearRgn**

```
function PtInTearRgn (hitPt: Point;
    menuRect: Rect): Boolean;
```

```
Boolean PtInTearRgn (Point hitPt, Rect *menuRect);
```

Returns TRUE if hitPt (given in global coordinates) is in the tear-off region. The tear-off region is anywhere except the menu bar and within TEARMARGIN pixels of the menu.

### **SetupQuickDraw**

```
procedure SetupQuickDraw (menuRect: Rect);
void SetupQuickDraw (Rect *menuRect);
```

Set up the QuickDraw environment so the MDEF's pane draws correctly. Since panes associated with MDEFs aren't really part of the visual hierarchy, you have to set up the QuickDraw environment directly. This method sets the port to the desktop's port and sets the origin to the top left of the pane and sets the clipping region to menuRect.

### **RestoreQuickDraw**

```
procedure RestoreQuickDraw;
void RestoreQuickDraw (void);
```

Restore the QuickDraw environment to what it was before the SetupQuickDraw call.



# CPanorama

## 46

### Introduction

CPanorama is an abstract class for implementing displays that may be larger than the frame of a pane. The frame is a viewport through which a portion of the panorama is visible.

### Heritage

Superclass	CPane
Subclasses	CPicture CStaticText

### Using CPanorama

Use a subclass of CPanorama whenever you want to display something that is larger than the pane you want to view it through. For example, some pictures are much bigger than a standard Macintosh screen. For an example of a panorama, look at the CEditText class on page 269. In almost every case, you'll use a panorama with a scroll pane (see CScrollPane on page 387 for a description).

Drawing in a panorama is almost the same as drawing in a pane. The main difference is that panoramas can have their own coordinate system. Each panorama unit can map to one or more pixels. For example, in the CEditText class, the horizontal unit is set to the number of pixels in the widest character, and the vertical unit is set to the number of pixels of the height of a line. Scroll panes use this information to set the scroll bars accurately.

---

#### Note

Panoramas do not support fractional scales or non-linear scales for coordinate systems.

---

The size of the panorama image is called the **bounds**. In most cases, the top, left corner of the bounds is the point (0, 0), but it's not required. The way to

think of the relationship between the panorama and the frame of the pane is that the panorama image is stationary as the frame roves over it. As you move around in a panorama, the coordinates of the top, left corner of the frame will change.

## Variables

Variable	Type	Description
bounds	LongRect	Bounds defining Pane coordinates.
position	LongPoint	Location of frame in panorama.
hScale	integer	Pixels per horizontal unit.
vScale	integer	Pixels per vertical unit.
savePosition	LongPoint	Save for later restoration.
itsScrollPane	CScrollPane	Scroll pane a panorama belongs to, if any.

## Methods

### Construction and destruction methods

#### IPanorama

```
procedure IPanorama (anEnclosure: CView;
  aSupervisor: CBureaucrat;
  aWidth, aHeight: integer;
  aHEnc1, aVEnc1: integer;
  aHSizing, aVSizing: SizingOption);
void IPanorama (CView *anEnclosure,
  CBureaucrat *aSupervisor,
  short aWidth, short aHeight,
  short aHEnc1, short aVEnc1,
  SizingOption aHSizing, SizingOption aVSizing);
```

Initialize a panorama. The arguments to this routine are identical to the ones for IPane.

---

#### Note

The descriptions of the other arguments are in CPane on page 321.

---

**IViewRes**

```
procedure IViewRes (rType: ResType; resID: integer;
  anEnclosure: CView; aSupervisor: CBureaucrat);
void IViewRes (ResType rType, short resID,
  CView *anEnclosure, CBureaucrat *aSupervisor);
```

Initialize a panorama from a resource template. Rtype is the resource type for the CView subclass you want to initialize. ResID is the resource ID of the resource. AnEnclosure and aSupervisor are the same as for IBorder. This method is inherited from CView.

To initialize a panorama from a resource file, use a 'Pano' resource.

**IViewTemp**

```
procedure IViewTemp (anEnclosure: CView;
  aSupervisor: CBureaucrat; viewData: Ptr);
void IViewTemp (CView *anEnclosure,
  CBureaucrat *aSupervisor, Ptr viewData);
```

This method is used internally for initializing from a resource template. Each subclass of CView overrides this method to use its own resource template.

**Accessing methods****GetExtent**

```
procedure GetExtent (var theHExtent, theVExtent:
  longint);
void GetExtent (long *theHExtent, long *theVExtent);
```

Get the size of each side of the panorama in panorama units.

**GetFramePosition**

```
procedure GetFramePosition (var theHPos, theVPos:
  longint);
void GetFramePosition (long *theHPos, long *theVPos);
```

Determine how far the top, left of the frame is from the top, left of the bounds in panorama coordinates. The CScrollPane class uses this method to figure out the maximum settings for the scroll bars. You should not use or override this method.

**GetFrameSpan**

```
procedure GetFrameSpan (var theHSpan, theVSpan:
  integer);
void GetFrameSpan (short *theHSpan, short *theVSpan);
```

Return the number of panorama units that the frame spans.

**SetBounds**

```
procedure SetBounds (aBounds: LongRect);
void SetBounds (LongRect *aBounds);
```

Set the bounds of the panorama. The bounds define the size of the data displayed in the panorama and the panorama coordinates. If the panorama's

enclosure is a scroll pane, this method sends it an `AdjustScrollMax` message to adjust the scroll bars.

**GetBounds**            `procedure GetBounds (var theBounds: LongRect);`  
                         `void GetBounds (LongRect *theBounds);`  
Get the bounds of a panorama.

**SetPosition**           `procedure SetPosition (aPosition: LongPt);`  
                         `void SetPosition (LongPt aPosition);`  
Set the position of the frame in relation to the panorama. The point is in panorama coordinates. If the panorama's enclosure is a scroll pane, this method sends it a `Calibrate` message to adjust the position of scroll box (the scroll bars' "thumb").

**GetPosition**           `procedure GetPosition (var thePosition: LongPt);`  
                         `void GetPosition (LongPt *thePosition);`  
Get the position of the frame in relation to the panorama. In other words, what are the panorama coordinates of the top, left of the frame?

**SetScales**             `procedure SetScales (aHScale, aVScale: integer);`  
                         `void SetScales (short aHScale, short aVScale);`  
Set the horizontal and vertical scales. Specify the scales in pixels per panorama unit. For instance, the `CStaticText` class (a subclass of `CPanorama`) uses the width of the widest character as its horizontal unit and the height of a line as a vertical unit. If the panorama's enclosure is a scroll pane, this method sends it an `AdjustScrollMax` message to adjust the scroll bars.

**GetScales**             `procedure GetScales (var theHScale, theVScale:`  
                                 `integer);`  
                         `void GetScales (short *theHScale, short *theVScale);`  
Get the scale factors of the panorama.

**SetScrollPane**         `procedure SetScrollPane (aScrollPane: CScrollPane);`  
                         `void SetScrollPane (CScrollPane *aScrollPane);`  
Specify the scroll pane that controls this panorama. This method is for use only by the `CPanorama` class. To associate a panorama with a scroll pane, use the `InstallPanorama` method.



<b>GetHomePosition</b>	<pre>procedure GetHomePosition (var theHomePos: LongPt); void GetHomePosition (LongPt *theHomePos);</pre> <p>Get the location of the top, left corner of the panorama in panorama coordinates.</p>
<b>GetPixelExtent</b>	<pre>procedure GetPixelExtent (var hExtent, vExtent:     longint); void GetPixelExtent (long *hExtent, long *vExtent);</pre> <p>Get the size of each side of the panorama in pixels.</p>
	<b>Calibrating methods</b>
<b>ResizeFrame</b>	<pre>procedure ResizeFrame (delta: Rect); void ResizeFrame (Rect *delta);</pre> <p>Adjust the frame of a panorama when its size changes. The delta rectangle specifies the amount of change for each side. Positive numbers mean down and to the right. Negative numbers mean up and to the left.</p>
	<b>Scrolling methods</b>
<b>Scroll</b>	<pre>procedure Scroll (hDelta, vDelta: longint;     redraw: Boolean); void Scroll (long hDelta, long vDelta,     Boolean redraw);</pre> <p>Scroll a panorama by hDelta units horizontally and vDelta units vertically. The units are given in panorama coordinates.</p>
<b>ScrollTo</b>	<pre>procedure ScrollTo (aPosition: LongPt;     redraw: Boolean); void ScrollTo (PointLongPt, Boolean redraw);</pre> <p>Scroll the panorama to a specific position. The units given in aPosition are in panorama coordinates.</p>
<b>ScrollToSelection</b>	<pre>procedure ScrollToSelection; void ScrollToSelection (void);</pre> <p>Scroll the panorama so the selection is visible. The default method does nothing. Your panorama subclass must override this method.</p>
<b>AutoScroll</b>	<pre>function AutoScroll (mouseLoc: Point): Boolean; Boolean AutoScroll (Point mouseLoc);</pre> <p>Scroll automatically during a mouse-down. Returns TRUE if scrolling actually took place. Typically, you would send this message <i>in the same routine</i> that tracks selection.</p>

### DoKeyDown

```
procedure DoKeyDown (theChar: char; keyCode: Byte;
    macEvent: EventRecord)
```

```
void DoKeyDown (char theChar, Byte keyCode,
    EventRecord *macEvent);
```

This method supports the Home, End, Page Up, and Page Down keys of the extended keyboard. The Home and End keys send `ScrollTo` messages to the panorama's scroll pane to scroll to the beginning or end of the panorama. The Page Up and Page Down keys send `DoVertScroll` messages to the panorama's scroll pane to simulate hits on the page up and page down regions of the scroll bar.

### Printing methods

### Paginate

```
procedure Paginate (aPrinter: CPrinter;
    pageWidth, pageHeight: integer);
```

```
void Paginate (CPrinter *aPrinter, short pageWidth,
    short pageHeight);
```

Determine how many pages to print. The panorama is divided into horizontal and vertical strips of equal size. `APrinter` is the printing object. Typically it belongs to the document that owns this panorama.

### AboutToPrint

```
procedure AboutToPrint (var firstPage, lastPage:
    integer);
```

```
void AboutToPrint (short *firstPage, short *lastPage);
```

The specified range of pages are about to be printed. You can override this method if your subclass needs to take some action before printing.

### PrintPage

```
procedure PrintPage (pageNum, pageWidth,
    pageHeight: integer; aPrinter: CPrinter);
```

```
void PrintPage(short pageNum, short pageWidth,
    short pageHeight, CPrinter *aPrinter);
```

Print the specified page. `PageWidth` is the width of the page in pixels. `PageHeight` is the height of the page in pixels. `APrinter` is the printer object. Typically it belongs to the document that owns this panorama.

### DonePrinting

```
procedure DonePrinting;
```

```
void DonePrinting (void);
```

Printing has stopped. Override this method if you need to do some cleanup after printing.

# CPatternGrid

## 47

### Introduction

CPatternGrid is a subclass of CGridSelector that displays patterns in a table and lets you choose one. CPatternGrid is useful for implementing pattern palettes like MacPaint and HyperCard.

### Heritage

Superclass	CGridSelector
Subclasses	None

### Using CPatternGrid

The CPatternGrid class lets you create panes that display patterns in a table. The most common use for this kind of table is a pattern palette for a painting or drawing program. You can use a CPatternGrid as a tool palette that's part of a window or as a custom tear-off menu. The Art Class demonstration program uses CPatternGrid to display its **Patterns** tear-off menu.

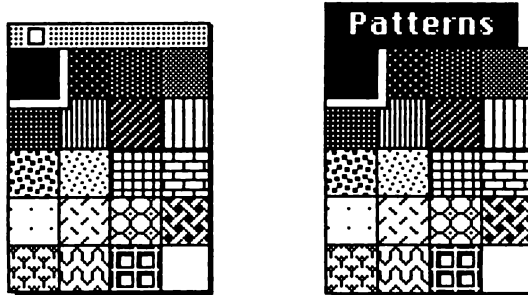


Figure 47-1 Art Class uses CPatternGrid as tear-off menu.

Unlike other subclasses of CPane that optionally let you initialize an object from a resource, you *must* use a resource to initialize a CPatternGrid. The first argument to the initialization method, IPatternGrid, is the resource ID of a 'PtGd' resource which contains the values that IPatternGrid

passes up to CGridSelector's initialization method. The 'PtGd' resource looks like this:

Field	Size	Description
Rows	integer	The number of rows in the grid.
Columns	integer	The number of columns in the grid.
Box Width	integer	The width in pixels of each box.
Box Height	integer	The height in pixels of each box.
Horiz. Sizing	integer	Horizontal sizing option. Usually <code>sizeFIXEDLEFT</code> (0)
Vert. Sizing	integer	Vertical sizing option. Usually <code>sizeFIXEDTOP</code> (2)
Horiz. Location	integer	Horizontal location of grid in its enclosure.
Vert. Location	integer	Vertical location of grid in its enclosure.
Command base	integer	The command base for turning selections into command numbers.
Pattern List ID	integer	The resource ID of the pattern list resource (PAT#) to use.
Number of patterns	integer	The number of patterns to display.
Pattern indices	integer	The index in the PAT# resource to display. One entry for each pattern.

*The standard pattern list is in the System file. Its resource ID is zero.*

The first nine fields are the same as the arguments that you pass to IGridSelector. The last three are specific to pattern grids. The Pattern List ID specifies which pattern list resource ('PAT#') the pattern grid should use. The Number of patterns field is how many of the patterns in the list to use. The Pattern indices are the indices of a pattern in the pattern list.

The file `TCL TMPLs` contains ResEdit templates (TMPL resources) that help you create and edit PtGd resources. This is what the 'PtGd' resource in Art Class looks like when you edit it with ResEdit.



**PtGd "Patterns" ID = 102 from Art Class.w.RSRC**

Rows	5
Columns	4
Box Width	20
Box Height	20
Horizontal Sizing	0
Vertical Sizing	2
Horizontal Location	0
Vertical Location	0
Command Prefix	102
Pattern List ID	0
Number of Patterns	20
1) *****	
Pattern Index	1
2) *****	
Pattern Index	2
3) *****	

Figure 47-2 The PtGd resource in Art Class

## Variables

Variable	Type	Description
patListID	integer	The ID of the 'PAT#' resource to use.
numPats	integer	Number of patterns to use
thePatterns	Handle	Handle to the list of 'PAT#' resource indices.

## Methods

### Construction methods

#### IPatternGrid

```

procedure IPatternGrid (PtGdid: integer;
    anEnclosure: CView; aSupervisor: CBureaucrat);
void IPatternGrid (short PtGdid, CView *anEnclosure,
    CBureaucrat *aSupervisor);

```

Initialize the character grid. PtGdid is the resource ID of the 'PtGd' resource that describes the location of the pane and pattern list to use for the grid. See "Using CPatternGrid" above for the details on 'PtGd' resources.

## ◆ 47 CPatternGrid

---

### Free/Dispose

```
procedure Free;  
void Dispose (void);  
Dispose of the index list and the object.
```

### Drawing methods

### DrawItem

```
procedure DrawItem (theItem: integer;  
    theBox: Rect);  
void DrawItem (short theItem, Rect *theBox);  
Draw pattern numbered theItem so it fills the box. Remember that the  
theItem doesn't specify the theItem'th pattern in the pattern list, but the  
theItem'th index in the index list. Patterns are numbered from 1.
```

### HiliteItem

```
procedure HiliteItem (theItem: integer;  
    state: HiliteState);  
void HiliteItem (short theItem, HiliteState state);  
Highlight the specified item. If state is hiliteON, this method draws a white  
outline around the box. If state is hiliteOFF, it draws the box normally.  
When state is hiliteDYNAMIC, HiliteItem flashes the edges of the box.  
Figure 47-1 shows how CPatternGrid highlights items.
```

### Accessing methods

### GetPattern

```
procedure GetPattern (var thePattern: Pattern);  
void GetPattern (Pattern *thePattern);  
Return the currently selected pattern in thePattern. The current selection  
is stored in the selection instance variable of the CSelector superclass.
```

# CPictFile 48 ♦

---

## Introduction

CPictFile is a class for reading and writing “draw” files like the kind MacDraw produces.

## Heritage

Superclass	CDataFile
Subclasses	None

## Using CPictFile

*To learn the details of the PICT file format, see Tech-Note 27*

You can use this class to read and write files whose type is PICT. These files contain “draw” type graphic images made with applications like MacDraw. A PICT file begins with 512 bytes of header information followed by a Macintosh picture.

Since CPictFile inherits its behavior from CFile, you need to use one of CFile’s specification methods which file the Open method will open.

## Variables

Variable	Type	Description
header	Handle	A handle to store the PICT header. Used internally.

## Methods

### IPictFile

```
procedure IPictFile;  
void IPictFile (void);
```

Initialize the object. This method calls CFile’s initialization method and allocates memory for the header.

## ◆ 48 CPictFile

---

### **Free/Dispose**

```
procedure Free;  
void Dispose (void);  
Dispose of the header, then dispose of the object.
```

### **ReadAll**

```
function ReadAll: Handle;  
Handle ReadAll (void);  
Read the picture from the PICT file into contents. You can use the handle  
with Quickdraw routines that expect a PicHandle or you can pass it to  
CPicture's SetMacPicture method. If there is an error reading the file, this  
method returns NIL.
```

### **WriteAll**

```
procedure WriteAll (contents: Handle);  
void WriteAll (Handle contents);  
Write a picture to a PICT file. The contents handle should be a PicHandle.
```

# CPicture

---

## Introduction

CPicture lets you display Macintosh pictures. The pictures are not editable.

## Heritage

Superclass	CPanorama
Subclasses	None

## Using CPicture

Use CPicture when you want to display a Macintosh picture. You can display the picture in a scroll frame or you can scale it to fit its frame. The picture is not editable. The picture is a standard Macintosh picture and is stored in an instance variable. You can look at the source code for this class to learn how to create your own panorama subclasses.

## Variables

Variable	Type	Description
macPicture	PicHandle	Handle to the QuickDraw picture.
scaled	Boolean	TRUE if picture is scaled to fit in frame.
isResPicture	Boolean	TRUE if picture is read from a resource.
ownsPicture	Boolean	TRUE if this object should release memory for the picture when the object is destroyed.

### Methods

#### Construction/Destruction

##### IPicture

```
procedure IPicture (anEnclosure: CView;
  aSupervisor: CBureaucrat;
  aWidth, aHeight: integer;
  aHEnc, aVEnc: integer;
  aHSizing, aVSizing: SizingOption);
void IPicture (CView *anEnclosure,
  CBureaucrat *aSupervisor,
  short aWidth, short aHeight,
  short aHEnc, short aVEnc,
  SizingOption aHSizing, SizingOption aVSizing);
```

Initialize a picture. The arguments are identical to pane initialization.

---

#### Note

The descriptions of the other arguments are in CPane on page 321.

---

##### IViewRes

```
procedure IViewRes (rType: ResType; resID: integer;
  anEnclosure: CView; aSupervisor: CBureaucrat);
void IViewRes (ResType rType, short resID,
  CView *anEnclosure, CBureaucrat *aSupervisor);
```

Initialize a picture from a resource template. RType is the resource type for the CView subclass you want to initialize. ResID is the resource ID of the resource. AnEnclosure and aSupervisor are the same as for IPicture. This method is inherited from CView.

To initialize a picture from a resource file, use a 'PctP' resource.

##### IViewTemp

```
procedure IViewTemp (anEnclosure: CView;
  aSupervisor: CBureaucrat; viewData: Ptr);
void IViewTemp (CView *anEnclosure,
  CBureaucrat *aSupervisor, Ptr viewData);
```

This method is used internally for initializing from a resource template. Each subclass of CView overrides this method to use its own resource template.

##### Free/Dispose

```
procedure Free;
void Dispose (void);
```

Dispose of this object. If the object owns the picture (ownsPicture is TRUE) the memory for the picture is released. If the picture comes from a resource (isResPicture is TRE), this method uses HPurge to purge the



'PICT' resource. If the picture is not from a resource, this method uses KillPicture to dispose of the picture.

### Appearance methods

#### Draw

```
procedure Draw (var area: Rect);  
void Draw (Rect *area);
```

Draw the picture. This method ignores the area parameter.

### Accessing methods

#### SetMacPicture

```
procedure SetMacPicture (aMacPicture: PicHandle);  
void SetMacPicture (PicHandle aMacPicture);
```

Use aMacPicture as the Macintosh picture for this object. If the picture comes from a resource, and the resource is purgeable, this method sets ownsPicture to FALSE. If the picture does not come from a resource, the picture is set to be un purgeable, and ownsPicture is set to TRUE.

#### UsePICT

```
procedure UsePICT (PICTid: integer);  
void UsePICT (short PICTid);
```

Use the 'PICT' resource with ID PICTid as the Macintosh picture for this object. This method gets the resource and then calls SetMacPicture.

#### GetMacPicture

```
function GetMacPicture: PicHandle;  
PicHandle GetMacPicture (void);
```

Return a handle to the Macintosh picture.

### Calibration methods

#### SetScaled

```
procedure SetScaled (aScaled: Boolean);  
void SetScaled (Boolean aScaled);
```

If aScaled is TRUE, the picture will be scaled to fit its frame.

#### GetScaled

```
function GetScaled: Boolean;  
Boolean GetScaled (void);
```

Return TRUE if the picture is scaled.

#### ResizeFrame

```
procedure ResizeFrame (delta: Rect);  
void ResizeFrame (Rect *delta);
```

Resize the picture's frame by the amount specified.

## ◆ 49 *CPicture*

---

### **FrameToBounds**

```
procedure FrameToBounds;  
void FrameToBounds (void);
```

Make the frame of the picture the same size as the bounds.



# CPNTGFile ♦

## 50

### Introduction

CPNTGFile is a class for reading and writing “paint” files like the kind MacPaint produces.

### Heritage

Superclass	CDataFile
Subclasses	None

### Using CPNTGFile

*To learn the details of the PNTG file format, see Tech-Note 86*

You can use this class to read and write files whose type is PNTG. These files contain “paint” or bitmapped graphic images made with applications like MacPaint. A PNTG file begins with 512 bytes of header information followed by a packed bitmap of the image.

Since CPNTGFile inherits its behavior from CFile, you need to use one of CFile’s specification methods which file the Open method will open.

The reading and writing methods of this class use objects of class CBitMap. That class gives you an object-oriented way to work with Macintosh bit-maps.

### Variables

Variable	Type	Description
header	Handle	A handle to store the PNTG header. Used internally.

### Methods

#### IPNTGFile

```
procedure IPNTGFile;  
void IPNTGFile (void);
```

Initialize the object. This method calls CFile’s initialization method and allocates memory for the header.

### Free/Dispose

```
procedure Free;  
void Dispose (void);  
Dispose of the header, then dispose of the object.
```

### ReadNewBitMap

```
function ReadNewBitMap (makePort: Boolean): CBitMap;  
CBitMap *ReadNewBitMap (Boolean makePort);  
Create a new object of class CBitMap and read the contents of this PNTG file  
into it. The makePort parameter is passed to CBitMap's IBitMap method.  
If it's true, IBitMap creates a Quickdraw grafport. If there was an error  
reading the image, this method returns NIL.
```

Note that this method creates a new bitmap every time you call it. The image is not stored in an instance variable.

### WriteBitMap

```
procedure WriteBitMap (theBitMap: CBitMap);  
void WriteBitMap (CBitMap *theBitMap);  
Write an object of class CBitMap to a PNTG file.
```

# CPrinter ♦

## 51

### Introduction

CPrinter is a class that handles standard Macintosh printing dialogs and calls the appropriate Print Manager routines.

### Heritage

Superclass	CObject
Subclasses	None

### Using CPrinter

The printer object manages communication between a document and the Macintosh print manager. Every document object can have a printer object associated with it. The document's `PrintPageOfDoc` method is the method that actually does the printing.

If your document is long, you may need to paginate your document (that is, divide it into pages). CPrinter lets you split your document into strips, each of which is a row or column of pages. At the intersection of a vertical strip and a horizontal strip, there is a page. Figure 51-1 shows how you might paginate graphic and text documents.

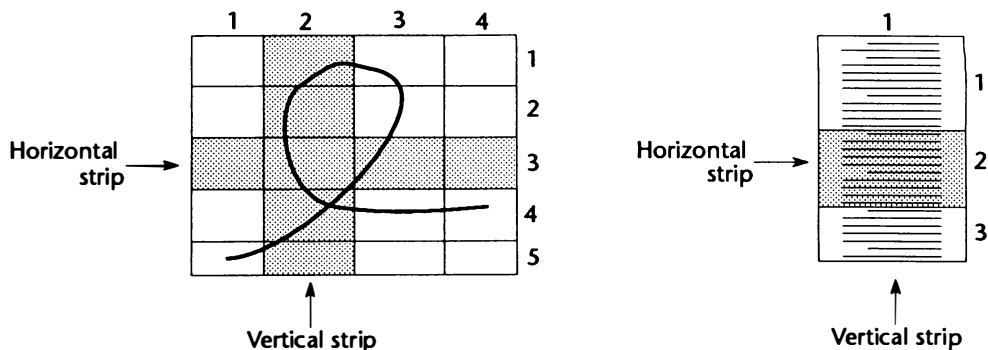


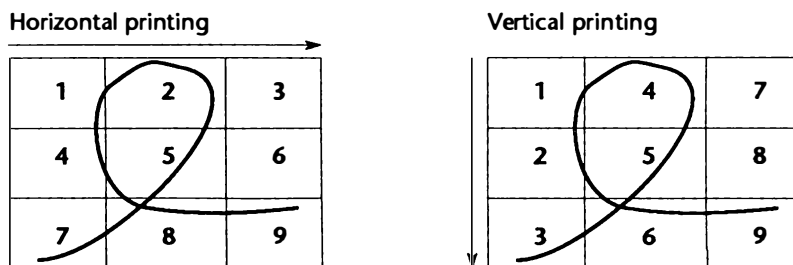
Figure 51-1 Paginating graphic and text documents

Note that strips don't have to be all the same size. Also, notice that each strip has a number. You need this number if you set the size or position of each strip individually.

CPrinter lets you set the size and position of strips in a variety of ways. Here are a few:

- To make all pages exactly the size the user chose in the **Page Setup...** dialog, use `SetStrips` on page 366.
- To insert a page break at a specific place in a document, use `SetHorizPageBreak` and `SetVertPageBreak` on page 367.
- To make all pages the same height or width, use `SetAllStripWidths` or `SetAllStripHeights` on page 367.
- To make a specific strip a specific height or width, use `SetStripWidth` or `SetStripHeight` on page 367.

If your document has more than one horizontal and vertical strip, you can choose the order the pages print in: horizontally (left to right first) or vertically (top to bottom first), as shown in Figure 51-2. Just send the printer the `SetPrintDir` message with either `printHoriz` or `printVert` as the argument.



**Figure 51-2** Printing horizontally and vertically

If you like, you can store a Macintosh print record with your document to preserve defaults. You can pass a handle to this record to the `IPrinter` method when you initialize your document. For methods in this class that return a Boolean value, `TRUE` means that the printer record stored with the printer object has changed. Ordinarily, you won't need to create a subclass of this class.



## Variables

Variable	Type	Description
itsDocument	CDocument	Document using this Printer.
macTPrint	THPrint	Toolbox print record.
printDirection	tPrintDirection	The direction of printing when there is more than one strip.
printMgrOpen	Boolean	TRUE, if the Print Manager is open.
printDocOpen	Boolean	TRUE, if this printer is currently printing the document.
printPageOpen	Boolean	TRUE, if this printer is currently printing a page.
savedResFile	integer	The resource file that this application used before CPrinter opened the Print Manager.
itsStripWidths	CRunArray	List of strip widths.
itsStripHeights	CRunArray	List of strip heights.

## Methods

### Construction and destruction methods

#### IPrinter

```
function IPrinter (aDocument: CDocument;
  aMacTPrint: THPrint): Boolean;
```

```
Boolean IPrinter (CDocument *aDocument,
  THPrint aMacTPrint);
```

Initialize a printer object. ADocument is the document the printer object is associated with. AMacTPrint is a Macintosh print record handle. If aMacTPrint is NIL, this method creates a new print record. This method returns TRUE if it had to update the aMacTPrint record because it was incompatible. The document's initialization method initializes a printer object automatically if the printable parameter to IDocument is TRUE.

## ◆ 51 CPrinter

---

### Free/Dispose

```
procedure Free;  
void Dispose (void);  
Dispose of a printer object.
```

### Accessing methods

### OpenPrintMgr

```
function OpenPrintMgr (fCheckFailure: Boolean):  
    Boolean;  
Boolean OpenPrintMgr (Boolean fCheckFailure);  
Open the Print Manager to get ready to print. If fCheckFailure is TRUE  
and this method could not open the Print Manager, this method displays an  
alert telling the user to choose a printer with the Chooser and returns FALSE.  
This method returns TRUE if there was no error or if fCheckFailure is  
FALSE.
```

### ClosePrintMgr

```
procedure ClosePrintMgr;  
void ClosePrintMgr (void)  
Close the Print Manager. This method aborts the printing of the current page  
and document, if this printer is in the middle of printing.
```

### SetPrintDir

```
procedure SetPrintDir (aPrintDir: tPrintDirection);  
void SetPrintDir (tPrintDirection aPrintDir);  
Set the direction of printing, in case there is more than one horizontal and  
vertical strip. TPrintDirection can be printHoriz or printVert.
```

### HavePagination

```
function HavePagination: Boolean;  
Boolean HavePagination (void)  
Return TRUE if this printer has any pagination information.
```

### ResetPagination

```
procedure ResetPagination;  
void ResetPagination (void)  
Clear the current pagination, setting the number of horizontal and vertical  
strips to zero.
```

### SetStrips

```
procedure SetStrips (numHStrips, numVStrips: integer);  
void SetStrips (short numHStrips, short numVStrips);  
Clear the current pagination and initialize it. This method sets the number of  
strips to numHStrips and numVStrips and sets the width (or height) of  
each strip to the width (or height) of the page size set in Page Setup...
```



<b>SetHorizPageBreak</b>	<pre>procedure SetHorizPageBreak (vStripNum: integer,                              hPos: longint); void SetHorizPageBreak (short vStripNum, long hPos);</pre> <p>Set a horizontal page break. VStripNum is the strip you're changing. HPos is the location of the page break in the frame coordinates of this printer's document.</p>
<b>SetVertPageBreak</b>	<pre>procedure SetVertPageBreak (hStripNum: integer,                              vPos: longint); void SetVertPageBreak (short hStripNum, long vPos);</pre> <p>Set a vertical page break. HStripNum is the strip you're changing. VPos is the location of the page break in the frame coordinates of this printer's document.</p>
<b>SetAllStripWidths</b>	<pre>procedure SetAllStripWidths (aStripWidth: integer); void SetAllStripWidths (short aStripWidth);</pre> <p>Sets the width all vertical page strips to aStripWidth.</p>
<b>SetAllStripHeights</b>	<pre>procedure SetAllStripHeights (aStripHeight: integer); void SetAllStripHeights (short aStripHeight);</pre> <p>Sets the height all horizontal page strips to aStripHeight.</p>
<b>SetStripWidth</b>	<pre>procedure SetStripWidth (pageNum, aStripWidth:                           integer); void SetStripWidth (short pageNum, short aStripWidth)</pre> <p>Sets the width of the vertical strip pageNum to aStripWidth.</p>
<b>SetStripHeight</b>	<pre>procedure SetStripHeight (pageNum, aStripHeight:                            integer); void SetStripHeight (short pageNum,                      short aStripHeight)</pre> <p>Sets the height of the horizontal strip pageNum to aStripHeight.</p>
<b>GetStripCount</b>	<pre>procedure GetStripCount (var hStrips, vStrips:                            integer); void GetStripCount (short *hStrips, short *vStrips);</pre> <p>Get the number of horizontal and vertical page strips.</p>

## 51 CPrinter

---

### PageNumToStrips

```
procedure PageNumToStrips (pageNum: integer;  
    var hStrip, vStrip: integer);  
void PageNumToStrips (short pageNum, short *hStrip,  
    short *vStrip)
```

Get the horizontal and vertical strips that page pageNum falls in.

### GetPageStart

```
procedure GetPageStart (pageNum: integer;  
    var startPos: LongPt);  
void GetPageStart (short pageNum, LongPt *startPos)
```

Get the starting position of page pageNum in the frame coordinates of this printer's document.

### GetPageArea

```
procedure GetPageArea (pageNum: integer;  
    var pageArea: LongRect);  
void GetPageArea (short pageNum, LongRect *pageArea)
```

Get the area of page pageNum.

### GetPrintRecord

```
function GetPrintRecord: THPrint;  
THPrint GetPrintRecord (void);
```

Return the Toolbox print record. This method calls PrValidate to update the record. You should treat the value that this method returns as read-only.

### GetPageInfo

```
procedure GetPageInfo (var paperRect, pageRect:  
    Rect; var hRes, vRes: integer);  
void GetPageInfo (Rect *paperRect, Rect *pageRect,  
    short *hRes, short *vRes);
```

Get information about the paper size and printable area of the page. The paperRect and pageRect are specified in dots. HRes and vRes specify the number of dots per inch.

### DoPageSetup

```
function DoPageSetup: Boolean;  
Boolean DoPageSetup (void);
```

Respond to a **Page Setup...** menu command. This method displays the standard job setup dialog, and returns TRUE if you made changes and pressed the OK button.

### Printing methods

### DoPrint

```
procedure DoPrint;  
void DoPrint (void);
```

Respond to a **Print...** menu command. This method displays the standard print job dialog. If you click on the OK button, this method sends a PrintPageRange message.



**PrintPageRange**

```
procedure PrintPageRange (firstPage, lastPage:
    integer);
void PrintPageRange (short firstPage, short lastPage);
```

Print the specified range of the document associated with this printer object. The `DoPrint` method usually sends this message. Your application can bypass the print dialogs and send this message directly, but Tech Note 122 discourages this practice.

This method sends your document an `AboutToPrint` message to give it an opportunity to adjust the page range. Then, for each page in the page range, it sends your document a `PrintPageofDoc` message.

## ◆ 51 CPrinter

---

# *CRadioControl* 52 ◆

---

## Introduction

CRadioControl implements a standard Macintosh radio button.

---

### Note

Earlier versions of The THINK Class Library used a different scheme to handle radio buttons. This class uses the dependent/provider mechanism implemented by CCollaborator to work with groups of radio buttons. The older class, CRadioButton is provided for backward compatibility, but not recommended.

---

## Heritage

Superclass	CButton
Subclasses	None

## Using CRadioControl

CRadioControl is a class that implements the standard Macintosh radio button. Since radio buttons always come in groups, a button must be a part of a radio group. The class CRadioGroupPane implements radio groups.

Like any other button, a radio button can have a command associated with it. Use the SetClickCmd method to set a radio button's command. You can also use any of the other CControl methods to manipulate the radio button.

See CRadioGroup pane for a discussion about working with groups of radio buttons.

## Variables

This class has no instance variables.

### Methods

#### IRadioControl

```
procedure IRadioControl (CNTLid: integer;  
    anEnclosure: CView; aSupervisor: CBureaucrat);  
void IRadioControl (short CNTLid, CView *anEnclosure,  
    CBureaucrat *aSupervisor);
```

Initialize a radio button from a CNTL resource. CNTLid is the resource ID for the radio button. AnEnclosure is the pane the radio button appears in. The enclosure should be a CRadioGroupPane. ASupervisor is the supervisor of the radio button. The supervisor should be a CRadioGroupPane.

#### INewRadioControl

```
procedure INewRadioControl (aWidth, aHeight: integer;  
    aHEncl, aVEncl: integer; title: StringPtr;  
    fVisible: Boolean, anEnclosed: CView,  
    aSupervisor: CBureaucrat);  
void INewRadioControl (short aWidth, short aHeight,  
    short aHEncl, short aVEncl, StringPtr title,  
    Boolean fVisible, CView *anEnclosure,  
    CBureaucrat *aSupervisor);
```

Initialize a radio button from the parameters in the argument list. AWidth and aHeight are the width and height of the button in pixels. AHEncl and aVEncl are the horizontal and vertical position of the button within its enclosure. Title is the text to write beside the button. And if fVisible is TRUE, the window is drawn immediately after it's created

---

#### Note

The rest of the parameters are described under  
IRadioControl on page 372.

---

#### DoGoodClick

```
procedure DoGoodClick (whichPart: integer);  
void DoGoodClick (short whichPart);
```

When the user presses and releases the mouse within the radio button, and the radio button was off, this method calls SetValue method to turn on the radio button. Note that SetValue in CControl sends a BroadcastChange message. The ProviderChanged method in CRadioGroupPane takes care of turning off the button that was on.

# *CRadioGroupPane* 53 ◆

## Introduction

CRadioGroupPane is a class that manages a group of radio buttons.

---

### Note

Earlier versions of The THINK Class Library used a different scheme to handle radio buttons. This class uses the dependent/provider mechanism implemented by CCollaborator to work with groups of radio buttons. The older class, CRadioGroup is provided for backward compatibility, but not recommended.

---

## Heritage

Superclass	CPane
Subclasses	None

## Using CRadioGroupPane

A radio group pane is a pane specifically designed for grouping radio buttons of class CRadioControl. The radio group pane helps you make sure that only one radio button in a group is on. The button that is on in a radio group is called the **station**.

To add a button to a radio group pane, simply use the group as the button's supervisor when you initialize the button. The group pane assumes that all its subviews are radio buttons. When a radio button is selected, it sends a BroadcastChange message with controlValueChanged as the reason. The group pane turns off the previously selected button and sets currentStation to the newly selected button.

There are two ways you can respond to clicks in radio buttons. One way is to give each radio button a unique ID, and to use SetStationID to set the initial radio button. After that, let the radio group pane manage the radio

buttons. When you want to find out which button is the station, use `GetStationID`.

The other way is to give each radio button its own command. Use the `SetClickCmd` method that `CRadioControl` inherits from `CButton` to give a radio button its own command. When you click on a radio button to turn it on, `CRadioControl`'s `DoGoodClick` method sends the button's supervisor a `DoCommand` message. If the radio button's supervisor is a plain `CRadioGroupPane`, the command should be handled by the group pane's supervisor. Or you can create a subclass of `CRadioGroupPane` with a `DoCommand` method to handle clicks in radio buttons.

### Variables

Only `CRadioGroupPane` and its subclasses should access this instance variable. In THINK C, this variable is protected.

Variable	Type	Description
<code>currentStation</code>	<code>CRadioControl</code>	The currently selected radio button

### Methods

#### Construction and destruction methods

#### IRadioGroupPane

```
procedure IRadioGroupPane (anEnclosure: CView;
    aSupervisor: CBureaucrat;
    aWidth, aHeight: integer;
    aHEncl, aVEncl: integer;
    aHSizing, aVSizing: SizingOption);
void IRadioGroupPane (CView *anEnclosure,
    CBureaucrat *aSupervisor,
    short aWidth, short aHeight,
    short aHEncl, short aVEncl,
    SizingOption aHSizing, SizingOption aVSizing);
```

Initialize a radio group pane. `ASupervisor` is the bureaucrat that owns the radio group pane. Typically, the supervisor is a pane or a window.

#### Note

The descriptions of the other arguments are in `CPane` on page 321.

**Accessing methods****SetStationID**

```
procedure SetStationID (aStationID: longint);  
void SetStationID (long aStationID);
```

Change the current selection to the button with the specified ID number.

**GetStationID**

```
function GetStationID: longint;  
long GetStationID (void);
```

Return the ID number of the currently selected radio button. Returns 0 if no station is selected.

**Change notification method****ProviderChanged**

```
procedure ProviderChanged (aProvider: CCollaborator;  
    reason: longint; info: Ptr);
```

```
void ProviderChanged (CCollaborator *aProvider,  
    long reason, void* info);
```

A radio button in this radio group pane has just been selected and sent a `BroadcastChange` message with `controlValueChanged` as the reason. This method turns off the previously selected button and sets `currentStation` to the newly selected button.

## ◆ 53 *CRadioGroupPane*

---



# CResFile

## 54

## Introduction

CResFile is an abstract class for working with Macintosh resource files.

## Heritage

Superclass	CFile
Subclasses	None

## Using CResFile

*To learn about Macintosh resources and resource files, see Inside Macintosh I, Chapter 5, Inside Macintosh IV, Chapter 3, and TechNote 214.*

If your application reads and writes resources make a subclass of CResFile and give it methods to access the resources. CResFile gives you methods to open and close resource files and to make the file the current resource file.

Since CResFile inherits its behavior from CFile, you need to use one of CFile's specification methods which file the Open method will open.

## Variables

Variable	Type	Description
refNum	Integer	File system reference number of opened file

## Methods

### IResFile

```
procedure IResFile;  
void IResFile (void);
```

Initialize the object. This method calls CFile's initialization method and initializes refNum to 0.

## ◆ 54 CResFile

---

<b>Open</b>	<pre>procedure Open (permission: SignedByte); void Open (SignedByte);</pre> <p>Open the resource file with the specified permission. If the file can't be opened, this method calls <code>FailResError</code>. Remember that you need use one of <code>CFile</code>'s specification methods to specify which file to open.</p>
<b>Close</b>	<pre>procedure Close; void Close (void);</pre> <p>Close this resource file. This method calls <code>FailOSErr</code> if there was a problem closing the file.</p>
<b>MakeCurrent</b>	<pre>procedure MakeCurrent; void MakeCurrent (void);</pre> <p>This method makes the resource file the current resource file.</p>
<b>IsOpen</b>	<pre>function IsOpen: Boolean; Boolean IsOpen (void);</pre> <p>Returns <code>TRUE</code> if the resource fork of this file is open.</p>
<b>Update</b>	<pre>procedure Update; void Update (void);</pre> <p>Updates this resource file by writing out the changed resource.</p>
<b>HasResFork</b>	<pre>function HasResFork: Boolean; Boolean HasResFork (void);</pre> <p>Returns <code>TRUE</code> if this file has a resource fork. The file must have been previously specified.</p>
<b>CreateNew</b>	<pre>procedure CreateNew (creator, fType: OSType); void CreateNew (OSType creator, OSType fType)</pre> <p>Creates a new resource file. If the file already exists but has no resource fork, this method gives it a resource fork.</p>

# *CRunArray* ♦

## 55

### Introduction

CRunArray implements a dynamic array of long integers that can conserve space.

### Heritage

Superclass	CArray
Subclasses	None

### Using CRunArray

CRunArray implements a dynamic array of long integers. It conserves space if your array contains lots of sequences of entries with the same value. These sequences are called runs. A run array consists of runs, which contain a value and the number of consecutive entries that have that value. For example, Figure 8-1 shows how a traditional array and a run array would store the same values.

#### Traditional Array

Index	Value
1	2
2	64
3	64
4	64
5	45

#### Run Array

Run	#Entries	Value
1	1	2
2	3	64
3	1	45

**Figure 8-1** A traditional array and a run array

To put an entry into the array, you can choose between `SetValue` and `InsertValue`. `SetValue` replaces an entry with a new entry. `InsertValue` inserts a run of values into the array. To delete an entry, use

DeleteValue. To sum a range of entries in the array, use the methods SumRange and FindSum.

When used with a run array, many CArray methods operate on runs of entries, not on individual entries. This table shows you some of those CArray methods and gives the CRunArray method you should use instead.

CArray method	CRunArray method
GetItem	GetValue
SetItem	SetValue
InsertAtIndex	InsertValue
DeleteItem	DeleteValue

In a CRunArray, the instance variable numItems contains the number of runs in the array. To find the number of entries, use the method GetNumItems, instead.

### Variables

Variable	Type	Description
itemCount	longint	Number of entries in the array.
hRuns	tRunHndl	Handle to runs. Same as hItems.

### Methods

#### Creation method

#### IRunArray

```
procedure IRunArray;
```

```
void IRunArray;
```

Initialize the array. The number of items and runs is set to 0 (zero).

#### Accessing method

#### GetNumItems

```
function GetNumItems: longint;
```

```
long GetNumItems (void);
```

Return the number of items in the array.

#### Insertion and deletion methods

#### InsertValue

```
procedure InsertValue (item, value, count: longint);
```

```
void InsertValue (long item, long value, long count);
```

Insert a run of values into the array. Item is the index to start the run at. If you specify an index beyond the end of the array, the run is added to the



end of the array. Value is the value for all in the entries in this run. Count is the length of the run.

**SetValue**

```
procedure SetValue (index, value: longint);  
void SetValue (long index, long value);  
Set the value of the entry at index to value.
```

**DeleteValue**

```
procedure DeleteValue (index: longint);  
void DeleteValue (long index);  
Delete the entry at index. All the entries following index move up one position.
```

**DeleteAll**

```
procedure DeleteAll;  
void DeleteAll (void);  
Delete all the entries in this array.
```

**Membership method****GetValue**

```
function GetValue (index: longint): longint;  
long GetValue (long index);  
Return the value of the entry at index.
```

**Summing methods****SumRange**

```
function SumRange (startIndex, endIndex: longint):  
    longint;  
long SumRange (long startIndex, long endIndex);  
Return the sum of the items between startIndex and endIndex, inclusive.
```

**FindSum**

```
function FindSum (aSum: longint): longint;  
long FindSum (long aSum);  
Return the index of the first entry such that all the entries from 1 to that entry have a sum equal to or greater than sum.
```

**Run-handling methods**

CArray uses these methods internally to manipulate runs. You should need to use them only if you are creating a subclass of CRunArray. In THINK C, they are protected.

## ◆ 55 CRunArray

---

### FindRun

```
procedure FindRun (itemIndex: longint;  
    var runIndex, firstInRun: longint);  
  
void FindRun (long itemIndex, long *runIndex,  
    long *firstInRun);
```

Return the number of the run that contains the entry at itemIndex. This method sets runIndex to the run number and firstInRun to the index of the first entry in the run. If itemIndex is not in the array, this method sets runIndex and firstInRun to BAD\_INDEX.

### InsertRun

```
procedure InsertRun (index, runLength, value:  
    longint);  
  
void InsertRun (long index, long runLength,  
    long value);
```

Insert a new run into the array. Index is the index of the first entry in the run. RunLength is the number of entries in the run. Value is the value of all the entries in the run.

### DeleteRun

```
procedure DeleteRun (runIndex: longint);  
void DeleteRun (long runIndex);
```

Delete the run at number runIndex from the array.

# CScrollBar

## 56

CScrollBar implements a standard Macintosh scroll bar.

### Heritage

Superclass	CControl
Subclasses	None

### Using CScrollBar

This class implements a Macintosh scroll bar. To make scroll bars easier to use, this class distinguishes between a mouse click in an indicator (the scroll box or the “thumb”) and a click in any other part of the scroll bar. Both behaviors are implemented in the `DoClick` method of the `CControl` class.

When you click in any part other than an indicator, the `DoClick` method calls the Toolbox routine `TrackControl` with an action procedure, or **action proc**. The action procedure is the routine that adjusts what the scroll bar controls. In most cases, the scroll bar controls a panorama. To set the action proc, use the `SetActionProc` method inherited from `CControl`.

When you click in the indicator, the `DoClick` method sends a `DoThumbDragged` message to the scroll bar. This method calls a **thumb function** that you provide. The thumb function is the routine that adjusts whatever the scroll bar controls. Thumb functions are unique to scroll bars. To set the thumb function, use the `SetThumbFunc` method.

Usually, you’ll use a scroll bar to control a pane. The class `CScrollPane` is a scrollable pane (a panorama) with one or two scroll bars. The `CScrollPane` class handles the usual cases, so you don’t have to provide an action proc or a thumb function.

### Variables

Variable	Type	Description
theOrientation	Orientation	Which way the scroll bar lies, horizontal or vertical.
theThumbFunc	ProcPtr	Function to call after a thumb drag.

### Methods

#### Construction and destruction methods

##### IScrollBar

```

procedure IScrollBar (anEnclosure: CView;
    aSupervisor: CBureaucrat;
    anOrientation: Orientation;
    aLength, aHEncl, aVEncl: integer);

void IScrollBar (CView *anEnclosure,
    CBureaucrat *aSupervisor,
    Orientation anOrientation,
    short aLength, short aHEncl, short aVEncl);

```

Initialize a scroll bar. AnEnclosure is the pane or window the scroll bar belongs to. ASupervisor is the scroll bar's supervisor in the chain of command. Orientation is either HORIZONTAL or VERTICAL. ALength is the length of the scroll bar. AHEncl and aVEncl are the horizontal and vertical position of the upper left corner of the scroll bar.

#### Accessing methods

##### SetThumbFunc

```

procedure SetThumbFunc (aThumbFunc: ProcPtr);
void SetThumbFunc (VoidFunc aThumbFunc);

```

Set aThumbFunc to be the scroll bar's thumb function. The default DoClick method for controls sends a DoThumbDragged message to the control when the user moves an indicator in a control. The DoThumbDragged method for scroll bars calls the thumb function. You should declare a Pascal thumb function like this:

```

procedure MyThumbFunc (theControl: CControl;
    delta: integer);

```

And you should declare a C thumb function like this:

```

void MyThumbFunc (CControl *theControl,
    short delta);

```





TheControl is the control whose indicator was moved. Delta is the amount by which the value changed. To get the current value of the control, you can send it a GetValue message.

### Drawing methods

#### Draw

```
procedure Draw (var area: Rect);  
void Draw (Rect *area);
```

Draw the scroll bar. If the scroll bar is active, this method draws it the normal way. If the scroll bar is inactive, this method draws only the frame of the scroll bar.

#### Activate

```
procedure Activate;  
void Activate (void);
```

Activate the scroll bar.

#### Deactivate

```
procedure Deactivate;  
void Deactivate (void);
```

Deactivate the scroll bar.

### Click response methods

#### DoClick

```
procedure DoClick (hitPt: Point;  
    modifierKeys: integer; when: longint);  
void DoClick (Point hitPt, short modifierKeys,  
    long when);
```

Handle a click in the scroll bar. If the scroll bar's enclosure is a scroll pane, send an AdjustScrollMax message to the scroll pane.

#### DoThumbDragged

```
procedure DoThumbDragged (delta: integer);  
void DoThumbDragged (short delta);
```

If the scroll bar has a thumb function associated with it, call it with the scroll bar and delta as arguments. The default DoClick method for controls sends a DoThumbDragged message to the control when the user moves an indicator in a control. See SetThumbFunc on page 384.

## ◆ 56 CScrollBar

---

# CScrollPane

---

## 57

### Introduction

CScrollPane implements a pane with scroll bars that control a panorama.

### Heritage

Superclass	CPane
Subclasses	None

### Using CScrollPane

A scroll pane is a pane with a panorama and scroll bars to control what is being displayed in the pane. Most of your applications will use a scroll pane that occupies most of the window. After creating a scroll pane, you can send it a `FitToEnclFrame` message to make it as big as the window.

All you have to do to use a scroll pane is install a panorama with the `InstallPanorama` method. The scroll pane uses the scale of the panorama for the values of the scroll bars.

The scroll bars and panorama do not communicate directly. Mouse clicks in the scroll bars are reported to the scroll pane, which then tells the panorama how to scroll or shift its image. Similarly, changes in the panorama which would affect the scroll bars are reported to the scroll pane, which then adjusts the scroll bars.

## Variables

Variable	Type	Description
<code>itsPanorama</code>	<code>CPanorama</code>	The scrollable view. The “content” of the scroll pane.
<code>itsHorizSBar</code>	<code>CScrollBar</code>	The scroll pane’s horizontal scroll bar.
<code>itsVertSBar</code>	<code>CScrollBar</code>	The scroll pane’s vertical scroll bar.
<code>itsSizeBox</code>	<code>CSizeBox</code>	The scroll pane’s size box.
<code>hExtent</code>	<code>longint</code>	For internal use.
<code>vExtent</code>	<code>longint</code>	For internal use.
<code>hUnit</code>	<code>integer</code>	For internal use.
<code>vUnit</code>	<code>integer</code>	For internal use.
<code>hSpan</code>	<code>integer</code>	For internal use.
<code>vSpan</code>	<code>integer</code>	For internal use.
<code>hStep</code>	<code>integer</code>	Number of horizontal units to scroll by when the user clicks on an arrow.
<code>vStep</code>	<code>integer</code>	Number of vertical units to scroll by when the user clicks on an arrow.
<code>hOverlap</code>	<code>integer</code>	Number of units to overlap when the user clicks in a page region.
<code>vOverlap</code>	<code>integer</code>	Number of units to overlap when the user clicks in a page region.

## Methods

### Construction and destruction methods

#### **IScrollPane**

```

procedure IScrollPane (anEnclosure: CView;
    aSupervisor: CBureaucrat;
    aWidth, aHeight, aHEncl, aVEncl: integer;
    aHSizing, aVSizing: SizingOption;
    hasHoriz, hasVert, hasSizeBox: Boolean)

void IScrollPane (CView *anEnclosure,
    CBureaucrat *aSupervisor,
    short aWidth, short aHeight,
    short aHEncl, short aVEncl,
    SizingOption aHSizing, SizingOption aVSizing,

```



```
Boolean hasHoriz, Boolean hasVert,  
Boolean hasSizeBox);
```

Initialize a scroll pane. All but the last three arguments are identical to the arguments to `IPane`. If `hasHoriz` is `TRUE`, the scroll pane has an horizontal scroll bar. If `hasVert` is `TRUE`, the scroll pane has a vertical scroll bar. If `hasSizeBox` is `TRUE`, the scroll pane draws a size box in the lower-right corner of the pane.

---

#### Note

The descriptions of the other arguments are in `CPane` on page 321.

---

### **IViewRes**

```
procedure IViewRes (rType: ResType; resID: integer;  
anEnclosure: CView; aSupervisor: CBureaucrat);
```

```
void IViewRes (ResType rType, short resID,  
CView *anEnclosure, CBureaucrat *aSupervisor);
```

Initialize a scroll pane from a resource template. `RType` is the resource type for the `CView` subclass you want to initialize. `ResID` is the resource ID of the resource. `AnEnclosure` and `aSupervisor` are the same as for `IScrollPane`. This method is inherited from `CView`.

To initialize a scroll pane from a resource file, use a 'ScPn' resource.

### **IViewTemp**

```
procedure IViewTemp (anEnclosure: CView;  
aSupervisor: CBureaucrat; viewData: Ptr);
```

```
void IViewTemp (CView *anEnclosure,  
CBureaucrat *aSupervisor, Ptr viewData);
```

This method is used internally for initializing from a resource template. Each subclass of `CView` overrides this method to use its own resource template.

#### **Accessing methods**

### **InstallPanorama**

```
procedure InstallPanorama (aPanorama: CPanorama);
```

```
void InstallPanorama (CPanorama *aPanorama);
```

Establish `aPanorama` as the panorama associated with this scroll pane. This method sends `AdjustScrollMax` and `Calibrate` messages to the scroll pane.

### **SetSteps**

```
procedure SetSteps (aHStep, aVStep: integer);
```

```
void SetSteps (short aHStep, short aVStep);
```

Set the amount to scroll when the user clicks on the arrows of a scroll bar. The units are in the panorama's units.

## ◆ 57 CScrollPane

---

### GetSteps

```
procedure GetSteps (var theHStep, theVStep: integer);  
void GetSteps (short *theHStep, short *theVStep);
```

Get the amount to scroll when the user clicks on the arrows of a scroll bar. The units are in the panorama's units.

### SetOverlaps

```
procedure SetOverlaps (aHOverlap, aVOverlap: integer);  
void SetOverlaps (short aHOverlap, short aVOverlap);
```

Set the amount of overlap when the user clicks in the page (gray) regions of the scroll bar. The units are in the panorama's units.

### GetInterior

```
procedure GetInterior (var theInterior: LongRect);  
void GetInterior (LongRect *theInterior);
```

Get the interior of the scroll pane. The interior excludes the space that the scroll bars occupy.

### AdjustScrollMax

#### Scroll bar maintenance methods

```
procedure AdjustScrollMax;  
void AdjustScrollMax (void);
```

Adjust the maximum value of the scroll bars from the extent and frame size of the panorama.

### Calibrate

```
procedure Calibrate;  
void Calibrate (void);
```

Adjust the scroll bar's thumb when the position of the frame of the panorama changes.

### ChangeSize

```
procedure ChangeSize (delta: Rect; redraw: Boolean);  
void ChangeSize (Rect *delta, Boolean redraw);
```

Change the size of a scroll pane. Each component of the delta rectangle specifies how each side will change. Positive values mean down and to the right. Negative values mean up and to the left. If redraw is true, the scroll pane is redrawn on the next update event.

### DoHorizScroll

#### Scroll performance methods

```
procedure DoHorizScroll (whichPart: integer);  
void DoHorizScroll (short whichPart);
```

Scroll horizontally. WhichPart specifies which part of the scroll bar was hit. This method sends a DoScroll message.



- DoVertScroll**      `procedure DoVertScroll (whichPart: integer);`  
                          `void DoVertScroll (short whichPart);`  
 Scroll vertically. WhichPart specifies which part of the scroll bar was hit. This method sends a DoScroll message.
- DoThumbDrag**      `procedure DoThumbDrag (hDelta, vDelta: integer);`  
                          `void DoThumbDrag (short hDelta, short vDelta);`  
 Adjust the panorama when the scroll box (thumb) has been dragged. This method sends a DoScroll message to the panorama.
- DoScroll**            `procedure DoScroll (hDelta, vDelta: longint);`  
                          `void DoScroll (long hDelta, long vDelta);`  
 Scroll the panorama belonging to this pane by hDelta units horizontally and vDelta units vertically. The units are given in the panorama's coordinates. All the other methods in this class call this method to handle scrolling.

## Functions

Note that these are procedures, not methods.

- SBarActionProc**    `procedure SBarActionProc (macControl: ControlHandle;`  
                          `whichPart: integer);`  
                          `pascal void SBarActionProc (ControlHandle`  
                          `macControl, short whichPart);`  
 The Toolbox TrackControl routine calls this function continuously while the mouse is down in any part of the scroll bar except the scroll box (thumb). This function sends the scroll bar's supervisor (the scroll pane) DoHorizScroll and DoVertScroll messages that actually scroll the panorama.
- SBarThumbFunc**     `procedure SBarThumbFunc (theSBar: CScrollBar;`  
                          `delta: integer);`  
                          `void SBarThumbFunc (CScrollBar *theSBar, short delta);`  
 The scroll bar's DoThumbDragged method calls this routine after the scroll box of a scroll bar has been moved. A control's DoClick method sends a DoThumbDragged message when the user moves the indicator of a control. This function sends the scroll bar's supervisor (the scroll pane) a DoThumbDrag message.

## ◆ 57 CScrollPane

---



# CSelector 58

## Introduction

CSelector is an abstract class for drawing panes with several items that users can choose from. A tool palette is an example of a selector.

## Heritage

Superclass	CPanorama
Subclasses	CGridSelector

## Using CSelector

CSelector is an abstract class that defines the basic behavior of a pane that lets you choose from several items. A good example of a selector is a tool palette or a pattern palette. The THINK Class Library includes CGridSelector, which is a class for implementing those kinds of palettes, and its descendants CPatternGrid, for displaying the standard patterns, and CCharGrid, for displaying characters in a grid. You can use these classes directly for pattern palettes and tool palettes. If you want to implement a different kind of selector, you'll need to create a subclass of CSelector.

A selector works like menu without command numbers. Each selector has a **command base**, which is like a menu ID. You set the command base when you create a selector, or you can set it once it's been created. A selector contains **items**. You specify the number of items when you create the selector.

Every selector needs to have a Draw method to display the items. How the items appear in the selector and how they're arranged is up to you. You will also need to override the FindItem method to determine which item you clicked on, and you'll need to override the HighlightItem method highlight the selected item. If you want your selector to respond to double-click's you'll need to override the DoDoubleClick method as well.

### Variables

Variable	Type	Description
numItems	integer	The number of items to choose from.
selection	integer	The currently selected item.
commandBase	integer	The base value for converting selections into command numbers.

### Methods

#### Construction and destruction methods

#### ISelector

```

procedure ISelector (anEnclosure: CView;
    aSupervisor: CBureaucrat;
    aWidth, aHeight: integer;
    aHEnc, aVEnc: integer;
    aHSizing, aVSizing: SizingOption;
    aNumItems, aSelection, aCommandBase: integer);

void ISelector (CView *anEnclosure,
    CBureaucrat *aSupervisor,
    short aWidth, short aHeight,
    short aHEnc, short aVEnc,
    SizingOption aHSizing, SizingOption aVSizing,
    short aNumItems, short aSelection,
    short aCommandBase);

```

Initialize a selector. The first eight arguments to this routine are identical to the ones for IPane.

A`NumItems` specifies the number of items in this selector. A`Selection` is the initial item. A`CommandBase` is the base value for converting the selected item into a command number.

---

#### Note

The descriptions of the other arguments are in CPane on page 321.

---



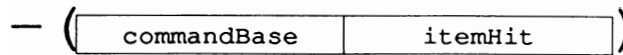
### Mouse methods

#### DoClick

```
procedure DoClick (hitPt: Point;
  modifierKeys: integer; when: longint);
void DoClick (Point hitPt, short modifierKeys,
  long when);
```

This method converts a click into a command number and sends it to the selector's supervisor in a DoCommand message. DoClick sends the selector a FindItem message to find out which item got the click.

Once it has an item, DoClick uses the same scheme as CBartender to build a command number. It puts the command base in the high word of a long integer, the item number in the low word, and negates the resulting long integer.



**Figure 58-1** How DoClick builds the command number.

If the new selection is not the same as the current selection, this method sends a ChangeHilite message to the selector. If the new selection is the same as the current selection, and there was a double-click, this method sends a DoDoubleClick message to the selector.

Since most of the work is done in methods the you must override, your CSelector subclass shouldn't need to override this method. You might want to override this method to handle things like triple-clicks.

#### HitSamePart

```
function HitSamePart (pointA, pointB: Point): Boolean;
Boolean HitSamePart (Point pointA, Point pointB);
```

Returns TRUE if the mouse went down in the same item, FALSE otherwise. The default method checks to see if FindItem(pointA) and FindItem(pointB) return the same item. Your CSelector subclass should not need to override this method.

### Accessing methods

#### ChangeSelection

```
procedure ChangeSelection (aSelection: integer);
void ChangeSelection (short aSelection);
```

Changes the selection from the current selection to aSelection. If aSelection is not the same as the current selection, this method turns off the highlighting of the current selection and turns on the highlighting of the new selection. Your subclass should not need to override this method.

## ◆ 58 CSelector

---

### GetSelection

```
function GetSelection: integer;
```

```
short GetSelection (void);
```

Return the current selection. You shouldn't need to override this method.

### SetCommandBase

```
procedure SetCommandBase (aCommandBase: integer);
```

```
void SetCommandBase (short aCommandBase);
```

This method sets the selector's command base. CSelector uses the command base to build a command number from the item hit. Your subclass should not need to override this method.

### GetCommandBase

```
function GetCommandBase: integer;
```

```
short GetCommandBase (void);
```

Return the value of the selector's command base. Your subclass should not need to override this method.

### HiliteItem

```
procedure HiliteItem (theItem: integer;  
    state: HiliteState);
```

```
void HiliteItem (short theItem, HiliteState state);
```

Hilite the specified item. HiliteItem can take on of three values for the state parameter:

<b>Hilite state value</b>	<b>Behavior</b>
hiliteOFF	Turn highlighting off for the item.
hiliteON	Turn highlighting on for the item.
hiliteDYNAMIC	Flash the selected item without selecting or deselecting.

It's up to you how you highlight the items of your selector. In most cases, it's sufficient to invert the rectangle that encloses the specified item. When highlighting is on, the item should be inverted. When highlighting is off, the item should appear normally. Dynamic highlighting is used when the selector is used as a menu. See the CSelectorMDEF class.

### FindItem

```
function FindItem (hitPt: Point): integer;
```

```
short FindItem (Point hitPt);
```

Determine which item corresponds to a mouse down at a specified point. It's up to you how your selector arranges and displays its items. Typically, items are arranged in a grid or a table. Your subclass must override this method.

**DoDoubleClick**

```
procedure DoDoubleClick;  
void DoDoubleClick (void);
```

Respond to a double-click. The first click will have set the selection, so the double-click will pertain to that item. If your selector subclass responds to double-clicks, you should override this method.



# CSelectorMDEF

## 59



### Introduction

CSelectorMDEF is a class that lets you use descendants of CSelector as custom menus.

### Heritage

Superclass	CPaneMDEF
Subclasses	None

### Using CSelectorMDEF

CSelectorMDEF is a class that handles menu selection for custom menus based on CSelector panes. If you pass CSelectorMDEF's initialization method a descendant of CSelector, you'll be able to use it as a custom menu. CSelectorMDEF's ChooseItem method takes care of selecting items from the CSelector. If you pass ISelectorMDEF a descendant of CTearOffMenu, you can use your custom menu as a tear-off menu.

### Variables

This class has no instance variables.

### Methods

#### Construction and destruction methods

#### ISelectorMDEF

```
procedure ISelectorMDEF (MDEFid: integer;  
    aPane: CPane; aTearOffMenu: CTearOffMenu);  
  
void ISelectorMDEF (short MDEFid, CPane *aPane,  
    CTearOffMenu *aTearOffMenu);
```

Initialize the selector MDEF. This method passes the arguments to CPaneMDEF's initialization method. APane must be a selector pane descended from CSelector. CTearOffMenu should be a descendant of CTearOffMenu where the custom menu will be displayed when it's torn off.

## ◆ 59 CSelectorMDEF

---

### ChooseItem

```
procedure ChooseItem (macMenu: MenuHandle;  
    menuRect: Rect; hitPt: Point;  
    var whichItem: integer);  
  
void ChooseItem (MenuHandle macMenu, Rect *menuRect,  
    Point hitPt, short *whichItem);
```

This method handles menu selection for custom menus.

If the hitPt is within the menuRect, this method sets up the QuickDraw drawing environment to match the pane's drawing environment. The QuickDraw origin is set so the point (0, 0) is the top left of the pane. ChooseItem then sends a FindItem message to the pane associated with this object.



# CSizeBox 60 ◆

---

## Introduction

CSizeBox implements a Macintosh grow icon.

## Heritage

Superclass	CPane
Subclasses	None

## Using CSizeBox

In most cases, you will not need to use this class yourself. It is used by CScrollPane to draw a Macintosh grow icon at the lower left corner of any pane, not just in the lower right corner of a window.

The first version of the THINK Class Library used an 'SICN' resource to draw the grow icon. If the useSICN flag is TRUE, the Draw method uses the resource, otherwise it calls the Toolbox routine DrawGrowIcon.

## Variables

Variable	Type	Description
useSICN	Boolean	TRUE, if this size box uses a SICN resource instead of calling DrawGrowIcon.

## Methods

### Construction and destruction methods

#### ISizeBox

```
procedure ISizeBox (anEnclosure: CView;  
    aSupervisor: CBureaucrat);  
void ISizeBox (CView *anEnclosure,  
    CBureaucrat *aSupervisor);
```

Initialize a size box. This method places a pane containing a size box at the lower right of its enclosure. The sizing options for a size box are sizFIXEDRIGHT and sizFIXEDBOTTOM. By default, useSICN is FALSE.

### Free/Dispose

```
procedure Free;  
void Dispose (void);  
Dispose of a size box.
```

### Appearance methods

### Draw

```
procedure Draw (var area: Rect);  
void Draw (Rect *area);
```

Draw a size box. If the pane is active, this method draws the size box. If the pane is inactive, this method draws a white rectangle. If useSICN is FALSE, this method uses the Toolbox's grow icon, otherwise it uses a 'SICN' resource to draw the grow icon.

### Activate

```
procedure Activate;  
void Activate (void);
```

The enclosure the size box belongs to is becoming active. The default method sends a Draw message to the size box

### Deactivate

```
procedure Deactivate;  
void Deactivate (void);
```

The enclosure the size box belongs to is becoming inactive. The default method sends a Draw message to the size box.

## Class resources

If useSICN is TRUE, this class uses an 'SICN' resource to draw the grow icon.

Resource	Description
SICN 200	The grow icon

## Introduction

CStack implements a stack of objects.

## Heritage

Superclass	CCluster
Subclasses	None

## Using CList

Use an object of class CStack when you need to maintain a stack of objects. You can use the iteration methods that CStack inherits from CCluster to apply functions to each item in the stack,

## Variables

This class has no instance variables.

## Methods

### IStack

```
procedure IStack;  
void IStack (void);
```

Initialize the stack. This method calls the CCluster's initialization method.

### Push

```
procedure Push (theObject: CObject);  
void Push (CObject *theObject);
```

Push theObject on the stack.

### Pop

```
function Pop : CObject;  
CObject *Pop;
```

Pop the item from the top of the stack. If the stack is empty, this method returns NIL.



# CSwitchboard

## 62

### Introduction

CSwitchboard is the class that processes Macintosh Toolbox events and sends the appropriate messages to objects in the THINK Class Library. There is normally only one instance of this class.

### Heritage

Superclass	CObject
Subclasses	None

### Using CSwitchboard

The single instance of this class handles all the Macintosh Toolbox events and sends messages to objects. The application's Run method repeatedly sends `ProcessEvent` messages to this object to dispatch messages to the objects that make up your application.

The application initialization method `IAplication` creates the single instance of CSwitchboard. The switchboard is stored in the application's `itsSwitchboard` instance variable.

---

#### Note

You need to subclass CSwitchboard only if your application handles `app1Evt`, `app2Evt`, or `app3Evt` events. Then you should override the `DoOtherEvent` method.

---

### Variables

<code>mouseRgn</code>	<code>RgnHandle</code>	Argument for <code>WaitNextEvent</code> to handle cursor adjustment.
-----------------------	------------------------	--

### Methods

#### Initialization methods

##### ISwitchboard

```
procedure ISwitchboard;  
void ISwitchboard (void);
```

Initialize the switchboard. The application's `IApplication` method creates the switchboard and sends it this message. This method also installs one handler for all `AppleEvents`. The handler sends the switchboard a `DoAppleEvent` message.

#### Mouse methods

##### DoMouseDown

```
procedure DoMouseDown (macEvent: EventRecord);  
void DoMouseDown (EventRecord *macEvent);
```

The user pressed the mouse button. This method sends a `DispatchClick` message to the desktop, and stores the event in the global `gLastMouseDown`.

##### DoMouseUp

```
procedure DoMouseUp (macEvent: EventRecord);  
void DoMouseUp (EventRecord *macEvent);
```

The user released the mouse button. This message sends a `DoMouseUp` message to the last view hit. Since mouse ups always follow a mouse down, it's not important where the mouse came up, but the message must be sent to the same object that handled the mouse down. This method stores the event in the global `gLastMouseUp`.

#### Key methods

##### DoKeyEvent

```
procedure DoKeyEvent (macEvent: EventRecord);  
void DoKeyEvent (EventRecord *macEvent);
```

The user pressed or released a key. If the user holds down the `Command` key and presses a key at the same time, this method uses the Toolbox routine `MenuKey` to find the menu equivalent. If there is a menu equivalent, this method sends a `DoCommand` message to the gopher. If there is no menu equivalent, this method sends a `DoKeyDown` message to the gopher. For other key events, this method send a `DoKeyDown`, `DoKeyUp`, or `DoAutoKey` messages to the gopher.

#### Disk methods

##### DoDiskEvent

```
procedure DoDiskEvent (macEvent: EventRecord);  
void DoDiskEvent (EventRecord *macEvent);
```

This method calls the Toolbox routine `DIBadMount` to mount a disk. This is the only event that the switchboard handles directly.

**Window event methods****DoUpdate**

```
procedure DoUpdate (macEvent: EventRecord);  
void DoUpdate (EventRecord *macEvent);
```

This method sends an Update message to the window specified in the event record.

**DoActivate**

```
procedure DoActivate (macEvent: EventRecord);  
void DoActivate (EventRecord *macEvent);
```

This method sends an Activate message to the window specified in the event record.

**DoDeactivate**

```
procedure DoDeactivate (macEvent: EventRecord);  
void DoDeactivate (EventRecord *macEvent);
```

This method sends an Deactivate message to the window specified in the event record.

**Suspend/Resume methods****DoSuspend**

```
procedure DoSuspend (macEvent: EventRecord);  
void DoSuspend (EventRecord *macEvent);
```

The application is about to be switched to the background under MultiFinder. This method sends a Suspend message to your application.

**DoResume**

```
procedure DoResume (macEvent: EventRecord);  
void DoResume (EventRecord *macEvent);
```

The application is about to be switched to the foreground under MultiFinder. This method sends a Resume message to your application.

**Event processing methods****DoOtherEvent**

```
procedure DoOtherEvent (macEvent: EventRecord);  
void DoOtherEvent (EventRecord *macEvent);
```

If your application handles `app1Evt`, `app2Evt`, or `app3Evt` events, override this method.

**DoIdle**

```
procedure DoIdle (macEvent: EventRecord);  
void DoIdle (EventRecord *macEvent);
```

This method is invoked during null events. This method sends an Idle message to your application. The application uses its Idle message to perform periodic tasks.

## 62 CSwitchboard

---

### DoHighLevelEvent

```
procedure DoHighLevelEvent (macEvent: EventRecord);  
void DoHighLevelEvent (const EventRecord* macEvent);
```

Handle a high level event. This method assumes all high level events are AppleEvents. If you use a high level event that isn't an AppleEvent, you must override this method. Your method should call inherited DoHighLevelEvent to handle AppleEvents.

### DoAppleEvent

```
function DoAppleEvent (macEvent, theReply: AppleEvent;  
    refCon: longint): OSerr;  
  
OSerr DoAppleEvent (AppleEvent *macEvent,  
    AppleEvent *theReply, long refCon);
```

Respond to an AppleEvent. This method packages the AppleEvent and the default reply into a CAppleEvent object and sends it to the gopher. If an exception occurs, this method returns the error causing the exception. The exception does not propagate beyond this method.

This method calls sends the application a PackageAppleEvent message to package the event and its default reply into a CAppleEvent object. If you subclass CAppleEvent, override this method.

### AppleEventIdle

```
function AppleEventIdle (macEvent: EventRecord;  
    var sleepTime: longint; mouseRgn: RgnHandle):  
    Boolean;  
  
Boolean AppleEventIdle (EventRecord *macEvent,  
    long *sleepTime, RgnHandle *mouseRgn);
```

The switchboard's default AppleEvent idle procedure sends the switchboard this message. The Toolbox functions AEInteractWithUser and AESend use the idle procedure to respond to an event while waiting for the user to respond to an AppleEvent. The possible events are null, update, OS, or activate events.

To use a different idle procedure, set the idleProc instance variable in the CAppleEvent object, described on page 131.

This method returns TRUE if the user aborted by pressing Command-. (Command-Period), and FALSE if the user wants to continue waiting.

### ProcessEvent

```
procedure ProcessEvent;  
void ProcessEvent (void);
```

This method is the heart of your application's event loop. This method gets an event and sends a message to the switchboard to handle it. Before processing the event, ProcessEvent sends a DispatchCursor message to the application to adjust the cursor.



**GetAnEvent**

```
function GetAnEvent (macEvent: EventRecord): Boolean;  
Boolean GetAnEvent (EventRecord *macEvent);
```

Get the next event in the event queue. This method calls one of the Toolbox routines `GetNextEvent` or `WaitNextEvent` to get an event and returns the result. If you need to do something to an event before the switchboard handles it, override this method. Your method should call `inherited GetAnEvent` and then do what you want with the event.

**DispatchEvent**

```
procedure DispatchEvent (macEvent: EventRecord);  
void DispatchEvent (EventRecord *macEvent)
```

This method is the main event dispatcher. Depending on the event, it sends an appropriate message back to itself to handle the event.



# CTask

## 63

### Introduction

CTask is an abstract class for implementing undoable actions.

### Heritage

Superclass	CObject
Subclasses	CMouseTask CTextEditTask CTextStyleTask

### Using CTask

A task is an abstract class for implementing undoable actions. If you want your application to be able to undo an action, you need to define a task subclass for each action.

You can use a task two ways. You can perform your action, create a task object, store enough information in it for its `Undo` method to undo the action, and send it in a `Notify` message to your supervisor (usually the document).

The second way is similar to the first, but you also implement a `Do` method that performs the action. So you create a task, send it a `Do` message to perform the action, and then send it in a `Notify` message to your supervisor. Your `Do` method stores enough information in the task's instance variables to undo the action.

When you notify a document that you've performed a task, it stores the task in the instance variable `lastTask`. When you choose **Undo** from the **Edit** menu, the document's `DoCommand` method sends an `Undo` message to that task.

Every task subclass has a string in the `STR# 130` resource used for the wording of the **Undo/Redo** command. Tasks have an instance variable that is the index of its string in the `STR#` resource. The document's

UpdateUndo method takes care of the wording of the **Undo/Redo** command.

Here's an example. Suppose you've defined a subclass of CTask to change the font in an edit text pane. Before passing the command on to the edit pane's DoCommand method, you create a task and store the current font in an instance variable. After you pass the font command to the edit text pane, you send the task in a Notify message to the document.

Your Undo method would simply send the font change command to the document. Since the command goes through the regular command chain, your DoCommand method would create a task to let you undo what you were undoing.

### Variables

Variable	Type	Description
nameIndex	integer	Index of the <b>Undo/Redo</b> string in the STR# 130 resource.
undone	Boolean	Is this task undone?

### Methods

#### Initialization methods

#### ITask

```
procedure ITask (aNameIndex: integer);
void ITask (short aNameIndex);
```

Initialize a task object. ANameIndex is the index of the task's **Undo** string in the STR# 130 resource. Your subclass's initialization method should call this method in addition to any other initialization it does. If your task subclass allocates memory, you'll also need to implement a Free method to release that memory.

#### Accessing methods

#### GetNameIndex

```
function GetNameIndex: integer;
short GetNameIndex (void);
```

Get the task's index. This method is used by the document's UpdateUndo method.

#### IsUndone

```
function IsUndone: Boolean;
Boolean IsUndone (void);
```

Return whether this task is undone.



<b>DoTask/Do</b>	<p><b>Action methods</b></p> <pre>procedure DoTask; void Do (void);</pre> <p>Perform a task. The default method does nothing. If you want to use a task to implement an action which is not necessarily undoable, your subclass should override this method. The <b>Undo/Redo</b> mechanism doesn't send DoTask messages.</p>
<b>Undo</b>	<pre>procedure Undo; void Undo (void);</pre> <p>Undo a task. The default method toggles the value of undone. Your subclass must store enough information to be able to undo an action. This is the method where you implement the undo.</p>
<b>Redo</b>	<pre>procedure Redo; void Redo (void);</pre> <p>Redo a task that was undone. The default method sends the task an Undo message. This method assumes that a redo is the same as undoing the undo. If your application implements <b>Redo</b> differently, you'll need to override this method.</p>

## Class resources

Resource	Description
STR# 130	List of strings for the wording of the <b>Undo/Redo</b> command. For instance, if you're implementing a "move" action, your string would be "Move". Each task contains the index of its string in this resource.



# CTearChore

## 64

### Introduction

CTearChore is a chore that notifies a tear-off menu that it has been torn off.

### Heritage

Superclass	CChore
Subclasses	None

### Using CTearChore

CTearChore is used in CTearOffMenu's `TornOff` method to let a menu that it has been torn off from the menu bar. `TornOff` creates a tear chore and assigns it as an urgent chore to the application. Your application should not have to use CTearChore directly, but you may find it useful as an example chore.

### Variables

Variable	Type	Description
<code>itsTearOffMenu</code>	CTearOffMenu	The tear-off menu that has been torn off.

### Methods

#### ITearChore

```
procedure ITearChore (aTearOffMenu: CTearOffMenu);  
void ITearChore (CTearOffMenu *aTearOffMenu);
```

Create a tear chore. `aTearOffMenu` is stored in `itsTearOffMenu`. `CTearOffMenu`'s `TornOff` method creates a tear chore and assigns it as an urgent chore.

#### Perform

```
procedure Perform (var maxSleep: longint);  
void Perform (long *maxSleep);
```

Sends a `MoveToCorner` method to `itsTearOffMenu`. This method does not change `maxSleep`.





# *CTearOffMenu*

---

## 65



### Introduction

CTearOffMenu is an abstract class that implements a Macintosh tear-off menu.

### Heritage

Superclass	CDirector
Subclasses	None

### Using CTearOffMenu

CTearOffMenu is a director that holds the pane of a menu that has been torn off from the menu bar. To use a CTearOffMenu, you need to create a subclass and override the initialization method so it creates a pane. Then create an MDEF class (CSelectorMDEF or another descendant of CPaneMDEF) and pass both the pane and the CTearOffMenu subclass to the initialization method.

The window that CTearOffMenu uses to display the tear-off menu is a floating window.

The Art Class example provided with the THINK Class Library uses two subclass of CTearOffMenu for the tool palette and the pattern palette.

### Variables

Variable	Type	Description
itsPane	CPane	The pane being displayed in the menu
corner	Point	Top left of torn-off window.
margins	Rect	Space between window bounds and the pane in the window

### Methods

#### ITearOffMenu

```
procedure ITearOffMenu (aSupervisor: CApplication;
    WINDid: Integer);
```

```
void ITearOffMenu (CApplication *aSupervisor,
    short WINDid);
```

Create a director for a tear-off menu. A tear-off menu's supervisor must be the application. WINDid is resource ID of the window that the tear-off menu appears in. Tear-off menus should use the WIndoid WDEF. The source for this WDEF as well as the WDEF resource itself is in the FW/Tearoffs folder.

ITearOffMenu sets itsPane to NIL, so your subclass needs to create a pane and set itsPane to it. This should be the same pane that you pass to the CPaneMDEF's initialization method.

#### Suspend

```
procedure Suspend;
void Suspend(void);
```

Hides the tear-off menu when the application is suspended.

#### Resume

```
procedure Resume;
void Resume(void);
```

Shows the tear-off menu when the application is reactivated..

#### CloseWind

```
procedure CloseWind(theWindow: CWindow);
void CloseWind(CWindow *theWindow);
```

Close the tear-off menu. To close the tear-off menu, this method moves its window off the screen.



<b>TornOff</b>	<pre>procedure TornOff(aCorner: Point); void TornOff(Point aCorner);</pre> <p>The menu has been torn off to the point aCorner. This method creates a tear chore (see CTearChore on page 415) that moves the window to the appropriate location.</p>
<b>MoveToCorner</b>	<pre>procedure MoveToCorner; void MoveToCorner(void);</pre> <p>Move the window to the corner stored by the TornOff method.</p>
<b>GetMacWindow</b>	<pre>function GetMacWindow(void): WindowPtr; WindowPtr GetMacWindow(void);</pre> <p>Return the window record for the tear-off menu window.</p>
<b>SetMargins</b>	<pre>procedure SetMargins(aMargins: Rect); void SetMargins(Rect *aMargins);</pre> <p>Set the margin between the window outline and the pane in the window. The margin is not a rectangle but the amount by which the pane's frame should be enlarged to create the gray outline of the tear-off menu.</p>
<b>GetMargins</b>	<pre>procedure GetMargins(var theMargins: Rect); void GetMargins(Rect *theMargins);</pre> <p>Return the margins in theMargins.</p>



# *CTextEditTask*

---

## 66



### Introduction

`CTextEditTask` provides undo support for typing and editing in `CAbstractText` subclasses.

### Heritage

Superclass	<code>CTask</code>
Subclasses	None

### Using `CTextEditTask`

`CTextEditTask` implements undo for typing and the **Cut**, **Copy**, **Paste**, and **Clear** commands in `CAbstractText` subclasses. A text pane automatically creates an instance of this class when you type, press Backspace or Forward Delete, or choose an editing command.

You may need to create a subclass of `CTextEditTask` if you create your own subclass of `CAbstractText`, especially if your subclass allows text to have multiple styles or if it doesn't store its text in a single contiguous block of memory. For example, a `CStyleText` text pane uses a task of type `CStyleTextEditTask` that can deal with the style scrap format.

### Variables

These are internal instance variables which only subclasses of CTextEditTask should use. In THINK C, they are protected.

Variable	Type	Description
itsTextPane	CAbstractText	Text pane that this task acts on.
editCmd	longint	Command being performed, cmdNull if typing
inserted	tTextRange	Info about the inserted text.
deleted	tTextRange	Info about the deleted text.
originalScrap	Handle	Contents of text scrap, before this task.
stillTyping	Boolean	TRUE if user is typing.
doText	Boolean	TRUE if this task changes the text in the text pane
doClip	Boolean	TRUE if this task changes the clipboard.
typingEvent	EventRecord	Event record for last keystroke.

### Methods

#### Creation and destruction methods

#### ITextEditTask

```
procedure ITextEditTask (aTextPane: CAbstractText;
    anEditCmd: longint; firstTaskIndex: integer);
void ITextEditTask (CAbstractText *aTextPane,
    long anEditCmd, short firstTaskIndex);
```

Initialize this text edit task. AnEditCmd is the command that this task is responding to, like cmdCut, cmdCopy, cmdPaste, or cmdClear. If the task is responding to typing, anEditCmd is cmdNull. ATextPane is the text pane for this task. FirstTaskIndex is the index of the first text edit **Undo** string in STR# 130. The text edit **Undo** strings are typically "Typing," "Cut," "Copy," "Paste," and "Clear." This method finds the index into STR# 130 for anEditCmd, sets doClip to TRUE if this command will change the contents of the clipboard, or cmdCut, sets doText to TRUE if this com-

mand adds or deletes text to the text pane, and saves the currently selected range.

**Free/Dispose**

```
procedure Free;
void Dispose (void);
```

Dispose of the memory for this object.

**Accessing method****CanStillType**

```
function CanStillType: Boolean;
Boolean CanStillType (void);
```

Returns TRUE if the user's typing doesn't start a new task. The user can type if the user hasn't tried to undo this task and `stillTyping` is TRUE.

**Action methods****DoTask/Do**

```
procedure Do;
void Do (void);
```

Perform an **Edit** menu command by sending the text pane a `PerformEditCommand` message. This method then stores the resulting selection

**DoTyping**

```
procedure DoTyping (theChar: char; keyCode: integer;
    macEvent: EventRecord);
void DoTyping (char theChar, short keyCode,
    EventRecord *macEvent);
```

Type a character. Depending on the character, this methods calls `DoBackspace`, `DoFwdDelete` or `DoNormalChar`. This method is called only when performing a command, not when undoing one.

**Undo**

```
procedure Undo;
void Undo (void);
```

Undo this task. This method saves the text the user inserted, removes the inserted text, and restores the text the user deleted. If you chose **Cut** or **Copy**, it also restores the old clipboard.

**Redo**

```
procedure Redo;
void Redo (void);
```

Redo this task, after it's been undone. This removes the text the user deleted and restores the text the user inserted. If you chose **Copy** or **Cut**, it restores the new clipboard.

### CancelTyping

```
procedure CancelTyping;  
void CancelTyping (void);
```

Stop accumulating characters that the user types. This method is called if this task was created to handle typing and you have just stopped to move the cursor or select **Undo**. After this method is called, you can undo the typing until you start editing again or initiate another task.

### SelectionChanged

```
procedure SelectionChanged;  
void SelectionChanged (void);
```

The selection has changed. This method sends a CancelTyping message.

#### Internal methods

### DoNormalChar

```
procedure DoNormalChar (theChar: char);  
void DoNormalChar (char theChar);
```

Handle a key that is not a Backspace or Forward Delete key, usually a cursor key or a character key. This method sends its text pane a TypeChar message.

### DoBackspace

```
procedure DoBackspace;  
void DoBackspace (void);
```

Handle the Backspace key. If you're performing this task, this method saves the character you're deleting.

### DoFwdDelete

```
procedure DoFwdDelete;  
void DoFwdDelete (void);
```

Handle the Forward Delete key. If you're performing this task, this method saves the character you're deleting.

### SaveRange

```
procedure SaveRange (whichRange: tRangeSelector);  
void SaveRange (tRangeSelector whichRange);
```

If whichRange is kDeletedText, save the text the user is about to delete. If whichRange is kInsertedText, save the text the user just inserted.

### DeleteRange

```
procedure DeleteRange (whichRange: tRangeSelector);  
void DeleteRange (tRangeSelector whichRange);
```

If whichRange is kDeletedText, delete the text the user deleted. If whichRange is kInsertedText, delete the text the user inserted.



**RestoreRange**

```
procedure RestoreRange (whichRange: tRangeSelector;  
    killData: Boolean);
```

```
void RestoreRange (tRangeSelector whichRange,  
    Boolean killData);
```

If whichRange is kDeletedText, restore the text the user deleted. If whichRange is kInsertedText, restore the text the user inserted. If killData is TRUE, this method disposes of this object's copy of the restored text.

**StoreToClip**

```
procedure StoreToClip (whichClip: tClipSelector);
```

```
void StoreToClip (tClipSelector whichClip);
```

If whichClip is kOldClip, store the original scrap text. If whichClip is kNewClip, store the deleted text to the clipboard.



# *CTextStyleTask* 67 ♦

---

## **Introduction**

CTextStyleTask provides undo support for style commands in CAbstractText subclasses.

## **Heritage**

Superclass	CTask
Subclasses	CStyleTETextTask

## **Using CTextStyleTask**

CTextStyleTask provides undo support for font, size, style, alignment, and spacing commands in CAbstractText subclasses. A text pane automatically creates an instance of this class when you choose a style command.

You may need to create a subclass of CTextStyleTask if you create your own subclass of CAbstractText, especially if your subclass allows text to have multiple styles or if it doesn't store its text in a single contiguous block of memory. For example, a CStyleText text pane uses a task of type CStyleTETextTask that can deal with the style scrap format.

### Variables

These are internal instance variables which only subclasses of CTextStyleTask should use. In THINK C, they are protected.

Variable	Type	Description
itsTextPane	CAbstractText	Text pane that this task acts on.
oldStyle	TextStyle	Style before this task.
oldAlignCmd	longint	Alignment before this task.
oldSpacingCmd	longint	Spacing before this task.
styleCmd	longint	Command this task performs.
styleAttribute	integer	Style attributes affected by this task.

### Methods

#### Construction method

#### ITextEditTask

```
procedure ITextStyleTask (aTextPane: CAbstractText;
    aStyleCmd: longint; taskIndex: integer);
void ITextStyleTask (CAbstractText *aTextPane,
    long aStyleCmd, short taskIndex);
```

Initialize this text style task. AStyleCmd is the command that this task is responding to. ATextPane is the text pane for this task. TaskIndex is the index of this command's **Undo** string in STR# 130.

#### Action methods

#### DoTask/Do

```
procedure DoTask;
void Do (void);
```

Save the original formatting, then performs the user's formatting command.

#### Undo

```
procedure Undo;
void Undo (void);
```

Save the current formatting and restore the previously saved formatting. This method handles both **Undo** and **Redo**.

**Internal methods****SaveStyle**

```
procedure SaveStyle;  
void SaveStyle (void);
```

Save the style of the text. Depending on the text pane, this method saves either the style for the whole text pane (like CEditText) or the style for the current selection (like CStyleText).

**RestoreStyle**

```
procedure RestoreStyle;  
void RestoreStyle (void);
```

Restores the previously saved formatting.

## ◆ 67 CTextStyleTask

---

# *CTextEnvirons* ◆

## 68

---

### Introduction

CTextEnvirons maintains a Quickdraw text drawing environment for any pane.

### Heritage

Superclass	CEnvironment
Subclasses	None

### Using CTextEnvirons

Every pane has an `itsEnvironment` instance variable. If this variable points to a descendant of CEnvironment, the `Prepare` method sends it a `Restore` message to set up the drawing environment for the pane.

You can use CTextEnvirons to make sure that a pane's text drawing characteristics are set up correctly. CTextEnvirons maintains the font, the size of the font, the font style, and the drawing transfer mode.

Suppose you have a pane that lets the user set the font and size of a text display. When you create your pane, you read the settings into a `TextInfoRec`, create a CTextEnvirons object, then set the `itsEnvironment` instance variable to point to it. Whenever the pane needs to be drawn, the `Prepare` method sends it a `Restore` message so the drawing mode is set up correctly.

Here's how you might set up a CTextEnvirons object for a pane in Pascal:

```

procedure CSomeDisplayPane.ISomeDisplayPane (
    anEnclosure: CView;
    aSupervisor: CBureaucrat);
var
    aTextInfo: TextInfoRec;
begin
    ...
    aTextInfo.fontNumber := ReadStoredFont;
    aTextInfo.theSize := ReadStoredSize;
    aTextInfo.theStyle := [];
    aTextInfo.theMode := srcCopy;
    new(CTextEnvirons(itsEnvironment));
    CTextEnvirons(itsEnvironment).SetTextInfo
        (aTextInfo);
    ...
end;

```

Here's how you might do the same thing in C:

```

void CSomeDisplayPane::ISomeDisplayPane (
    CView *anEnclosure,
    CBureaucrat *aSupervisor)
{
    TextInfoRec aTextInfo;
    ...
    aTextInfo.fontNumber = ReadStoredFont();
    aTextInfo.theSize = ReadStoredSize();
    aTextInfo.theStyle = 0;
    aTextInfo.theMode = srcCopy;
    itsEnvironment = new CTextEnvirons;
    ((CTextEnvirons *) itsEnvironment)->
        SetTextInfo(&aTextInfo);
    ...
}

```

### Variables

Variable	Type	Description
textInfo	TextInfoRec	Text characteristics

The TextInfoRec looks like this:

Field	Type	Description
fontNumber	Integer	The number of the font
theSize	Integer	Size of the font
theStyle	Style	Style of the font (in THINK C, this type is short)
theMode	Integer	Text transfer mode





## Methods

### **ITextEnvirons**

```
procedure ITextEnvirons;  
void ITextEnvirons (void);
```

Initialize every field of the `textInfo` record to zero. These settings correspond to the default system font, the default system size, plain style, and the `scrCopy` transfer mode.

### **Restore**

```
procedure Restore;  
void Restore (void);
```

Sets the Quickdraw text drawing characteristics to the values previously stored with `SetTextInfo`. This method uses the standard QuickDraw text setting routines: `TextFont`, `TextSize`, `TextFace`, and `TextMode`. This method also calls the Toolbox routine `PenNormal`.

### **SetTextInfo**

```
procedure SetTextInfo (aTextInfo: TextInfoRec);  
void SetTextInfo (TextInfoRec *aTextInfo);
```

Sets the text drawing characteristics to the values in `aTextInfo`. The next time the pane is redrawn, the QuickDraw text drawing characteristics will be set to the values supplied.

### **GetTextInfo**

```
procedure GetTextInfo (var aTextInfo: TextInfoRec);  
void GetTextInfo (TextInfoRec *aTextInfo);;
```

Get the current text drawing characteristics from the object.



# CView 69 ♦

---

## Introduction

CView is an abstract class for implementing objects that have a visual representation. Every object in the visual hierarchy is a descendant of this class.

## Heritage

Superclass	CBureaucrat
Subclasses	CDesktop
	CPane
	CWindow

## Using CView

CView is an abstract class for implementing objects with a visual representation. In other words, anything you can see on the screen is a descendant of CView. Views respond to visual commands involving the mouse. And because CView is a descendant of CBureaucrat, a view can be one of the links in the chain of command.

The standard classes define three subclasses of CView. These are the desktop, windows, and panes. Most of the time, you'll be dealing with panes. As you work with panes and descendants of CPane, keep in mind that all methods that apply to views apply to them as well.

## Views and the visual hierarchy

All views have an enclosure that specifies its place in the visual hierarchy. Each view can enclose several subviews. The top of the visual hierarchy, the desktop, is the only view that does not have an enclosure. The desktop encloses all the windows in your application. Each window encloses one or more panes. Panes can enclose other panes.

The desktop handles some visual commands, like mouse clicks, and sends them on to the appropriate window. The switchboard sends window related

messages, like updates and activates, directly to a window which sends it to its subviews.

By default, a view does not process mouse clicks. If a view can respond to mouse clicks, you need to call `SetWantsClicks (TRUE)` when you create it to let `DispatchClick` and `AdjustCursor` know that they should look for clicks in mouse movement in that view.

### Views and the chain of command

Each view has a supervisor which is the view's boss in the chain of command. The desktop's supervisor is always the application. A window's supervisor is always its director or document. A pane's supervisor is usually its director or document.

The desktop and windows are almost never the first in the chain of command. In other words, they're almost never the gopher. Panes, on the other hand, are frequently made the gopher. For instance, you need to make an edit pane the gopher so it can respond to typing and menu commands.

If a view can be a gopher, you need to call `SetCanBeGopher (TRUE)` so `DispatchClick` can make the view the gopher and let the previous gopher know that it's not the gopher anymore. Although you can force a view to be the gopher by setting the `gGopher` instance variable, you should rely on the gopher-setting mechanism of `BecomeGopher`.

### Using Balloon Help with views

Any view can have a help balloon associated with it. The THINK Class Library uses 'hrcr' resources to specify help balloons. The Macintosh Help Manager, described in *Inside Macintosh VI*, uses a combination of 'hwin' and 'hrcr' resources to display help balloons for stationary windows. The THINK Class Library uses only 'hrcr' resource. They should not be associated with 'hwin' resources.

Each window holds a resource ID to an 'hrcr' resource for help associated with that window. If the window doesn't provide a resource ID for the 'hrcr' resource, it uses the default 'hrcr' resource whose ID is `kDefaultHelpResID (128)`.

If you want to provide help for a pane, set the `helpResIndex` instance variable, which it inherits from `CView`, as an index into the 'hrcr' resource. Otherwise, just set `helpResIndex` to 0.

The `DispatchCursor` method uses `GetBalloonInfo` and `ShowHelp-Balloon` to look for the help resource and to display them.

## Variables

Variable	Type	Description
macPort	GrafPtr	Mac drawing port for the view
itsEnclosure	CView	View which totally encloses this one
itsSubviews	CList	Views contained within this view
visible	Boolean	Is the view visible?
active	Boolean	Is the view active?
wantsClicks	Boolean	Does the view handle mouse clicks?
canBeGopher	Boolean	Can this view be the gopher?
ID	longint	The identifier for this view.
usingLongCoord	Boolean	TRUE if using 32-bit coordinates
helpResIndex	integer	Index into 'hrcr' resource for balloon help

The following three variables are class variables. In Pascal these are global variables. CView uses these class variables for tracking help balloons and for optimizing calls to Prepare. In general, you won't need to use them.

Variable	Type	Description
cCurrHelpView	CView	Used in DispatchCursor to determine whether a help balloon was displayed
cLastHelpView	CView	The view that is showing a help balloon
cPreparedView	CView	Currently prepared view

## Methods

### Construction and destruction methods

#### IView

```
procedure IView (anEnclosure: CView;
  aSupervisor: CBureaucrat);
void IView (CView *anEnclosure,
  CBureaucrat *aSupervisor);
```

Initialize a view. Views start out with no port, no subviews, invisible, and inactive. By default, views don't want clicks. AnEnclosure is the view that

completely encloses this view. ASupervisor is the bureaucrat that gets command messages for the commands this view can't handle.

### IViewRes

```
procedure IViewRes (rType: ResType; resID: integer;
    anEnclosure: CView; aSupervisor: CBureaucrat);
void IViewRes (ResType rType, short resID,
    CView *anEnclosure, CBureaucrat *aSupervisor);
```

Initialize a view from a resource template. Each subclass of CView overrides this method to use its own resource template. RType is the resource type for the CView subclass you want to initialize. Res ID is the resource ID of the resource. AnEnclosure is the view that completely encloses this view. ASupervisor is the bureaucrat that gets command messages when the view can't handle them.

To initialize a view from a resource file, use a 'View' resource.

### IViewTemp

```
procedure IViewTemp (anEnclosure: CView;
    aSupervisor: CBureaucrat; viewData: Ptr);
void IViewTemp (CView *anEnclosure,
    CBureaucrat *aSupervisor, Ptr viewData);
```

The IViewRes method sends an IViewTemp message to initialize a view from a resource template. All subclasses of CView override this method so they can be initialized from resource templates.

### Free/Dispose

```
procedure Free;
void Dispose (void);
```

Dispose of a view. Disposes of all subviews

#### Accessing methods

### IsVisible

```
function IsVisible: Boolean;
Boolean IsVisible (void);
```

Return TRUE if the view is visible.

### IsActive

```
function IsActive: Boolean;
Boolean IsActive (void);
```

Return TRUE if the view is active.

### ReallyVisible

```
function ReallyVisible: Boolean;
Boolean ReallyVisible (void);
```

Return TRUE if the view is visible and if its enclosure is ReallyVisible.



<b>GetMacPort</b>	<pre>function GetMacPort: GrafPtr; GrafPtr GetMacPort (void);</pre> <p>Get the GrafPort associated with this view.</p>
<b>GetOrigin</b>	<pre>procedure GetOrigin (var theHOrigin,     theVOrigin: longint); void GetOrigin (long *theHOrigin, long *theVOrigin);</pre> <p>Get the origin of the view (the top left corner). The default method returns (0, 0).</p>
<b>GetFrame</b>	<pre>procedure GetFrame (var theFrame: LongRect); void GetFrame (LongRect *theFrame);</pre> <p>Get the frame of the view. The default method does nothing. View subclasses (like Pane) must override this message.</p>
<b>GetInterior</b>	<pre>procedure GetInterior (var theInterior: LongRect); void GetInterior (LongRect *theInterior);</pre> <p>Get the interior of the view. The default method returns what GetFrame returns. If the interior of a particular subclass is not the same as the frame, the class must override this method.</p>
<b>GetAperture</b>	<pre>procedure GetAperture (var theAperture: LongRect); void GetAperture (LongRect *theAperture);</pre> <p>Get the aperture of the view. (The aperture is the visible portion of a view.) The default method does nothing. Subclasses must override this method.</p>
<b>Contains</b>	<pre>function Contains (thePoint: Point): Boolean; Boolean Contains (Point thePoint);</pre> <p>Return TRUE if the view contains thePoint. The default method always returns FALSE.</p>
<b>SetWantsClicks</b>	<pre>procedure SetWantsClicks (aWantsClicks: Boolean); void SetWantsClicks (Boolean aWantsClicks);</pre> <p>If aWantsClicks is true, the view will report clicks in itself. By default, views don't want clicks.</p>
<b>GetWantsClicks</b>	<pre>function GetWantsClicks: Boolean; Boolean GetWantsClicks (void);</pre> <p>Returns TRUE if this view wants to receive clicks</p>

<b>SetCanBeGopher</b>	<pre>procedure SetCanBeGopher (fCanBeGopher: Boolean); void SetCanBeGopher (Boolean fCanBeGopher)</pre> <p>If fCanBeGopher is TRUE, this view can become the gopher. By default, a view can't become the gopher.</p>
<b>CanBeGopher</b>	<pre>function CanBeGopher: Boolean; Boolean CanBeGopher (void)</pre> <p>Returns TRUE if this view can become the gopher.</p>
<b>SetID</b>	<pre>procedure SetID (anIdentifier: longint); void SetID (long anIdentifier)</pre> <p>Sets this view's ID to be anIdentifier. By default, the ID is 0 (zero). It is up to you to provide IDs and to guarantee that they're unique. View IDs give you a way to identify particular views within the view hierarchy. You can use Macintosh style identifiers like 'MyVu' or plain numeric constants.</p>
<b>GetID</b>	<pre>function GetID: long; long GetID (void)</pre> <p>Returns the ID for this view.</p>
<b>UseLongCoordinates</b>	<pre>procedure UseLongCoordinates (fUsing: Boolean); void UseLongCoordinates (Boolean fUsing);</pre> <p>Specify whether the view should use 32-bit coordinates (long coordinates) or 16-bit QuickDraw coordinates. If fUsing is TRUE, the view is using 32-bit coordinates.</p>
<b>Appearance methods</b>	
<b>Show</b>	<pre>procedure Show; void Show (void);</pre> <p>Make a view visible.</p>
<b>Hide</b>	<pre>procedure Hide; void Hide (void);</pre> <p>Hide a view. If the view is the gopher, make its supervisor the gopher.</p>
<b>Activate</b>	<pre>procedure Activate; void Activate (void);</pre> <p>Make a view active. Activate all the subviews as well.</p>



**Deactivate**

```
procedure Deactivate;  
void Deactivate (void);
```

Make a view inactive. Deactivate all the subviews as well. If the view is the gopher, make its supervisor the gopher.

**Mouse methods****DispatchClick**

```
procedure DispatchClick (var macEvent: EventRecord);  
void DispatchClick (EventRecord *macEvent);
```

Find out which subview got the click and send it a DispatchClick message. If there are no subviews, the click is for this view. If the view can be the gopher, DispatchClick calls BecomeGopher (TRUE) to make the view the gopher. Then DispatchClick sets calls Prepare to set up the drawing environment. Finally, DispatchClick sends a DoClick message to the view to handle the click. Ordinarily, you should not need to override this method.

**DoClick**

```
procedure DoClick (hitPt: Point;  
    modifierKeys: integer; when: longint);  
void DoClick (Point hitPt, short modifierKeys,  
    long when);
```

*These conversion routines  
are described on page 333*

The mouse went down in this view. If the view is not using long coordinates, hitPt is in frame coordinates. If the view is using long coordinates, hitPt is given in QuickDraw coordinates. You can use CPane's QDToFrame, QDToFrameR to convert from long coordinates to frame coordinates, and FrameToQD and FrameToQDR to convert from long coordinates to QuickDraw coordinates. Subclasses must override this method.

**HitSamePart**

```
function HitSamePart (var pointA, pointB: Point):  
    Boolean;
```

```
Boolean HitSamePart (Point pointA, Point pointB);
```

Check whether two points hit the same part of the view. The default method always returns TRUE. Subclasses that override this method should decide what constitutes a part and how close is close.

**DoMouseUp**

```
procedure DoMouseUp (macEvent: EventRecordPtr);  
void DoMouseUp (EventRecord *macEvent);
```

The mouse went up in a view. Default method does nothing.

### Cursor methods

#### DispatchCursor

```
procedure DispatchCursor (where: Point;
    mouseRgn: RgnHandle);
void DispatchCursor (Point where, RgnHandle mouseRgn);
```

Find which view the cursor is in. If the cursor is in this view, send it an `AdjustCursor` message. If the cursor is in a subview, send it a `DispatchCursor` message. `DispatchCursor` sends `AdjustCursor` and `DispatchCursor` messages only to views that want clicks.

If the balloon help system is available, and the application has help resources, and no other view has already displayed the help text, `DispatchCursor` display the help balloon associated with this view.

Your application should not need to override this class.

#### AdjustCursor

```
procedure AdjustCursor (where: Point;
    mouseRgn: RgnHandle);
void AdjustCursor (Point where, RgnHandle mouseRgn);
```

Adjust the cursor. A view gets an `AdjustCursor` message when the cursor moves into it and if the view wants clicks. The default method sets the cursor to an arrow. Your subclass should override this method to set the cursor to whatever is appropriate for the view. See the `AdjustCursor` method for `CAbstractText` for an example.

If you use only one cursor within a view, you can ignore the two parameters to this method. If you want to use different cursor shapes within a pane, you need use these two parameters.

It's unlikely that you'll want multiple cursors for a view that's not a pane. The `CPane` class inherits this method.

The `where` parameter tells you where the cursor is in window coordinates. You can use the `WindToFrame` pane method to convert it to frame coordinates. The `mouseRgn` parameter is the region in which the cursor shape stays the same. In other words, your pane will not get an `AdjustCursor` message again until the cursor leaves this region. The `mouseRgn` is specified in global coordinates.

Suppose you're displaying a map of the United States in a pane, and you want the cursor to be a star when it's over Texas. You use the `where` parameter, converted from window coordinates to frame coordinates, to determine that you're in Texas, and you change the cursor to a star.



Now, when you leave Texas, you want to set the cursor shape to something else. Since your pane gets `AdjustCursor` messages only when the cursor leaves the `mouseRgn`, you have to change the `mouseRgn`. You create a region with the shape of the state of Texas, convert this region to global coordinates, and make it the `mouseRgn`.

---

**Note**

The mouse region should be the intersection of the original mouse region and the one you're specifying. This way, you preserve any clipping boundaries that the original mouse region had accounted for.

---

This is what the `AdjustCursor` method for this example might look like in Pascal:

```
procedure MapPane.AdjustCursor (where: Point;
                                mouseRgn: RgnHandle);
var
    locWhere: Point;
    TexasRgn: RgnHandle;
    TexasRect: Rect;
    tempRect: Rect;

begin
    TexasRgn := NewRgn;
    locWhere := where;
    WindToFrame(locWhere);

    { If we're not in Texas, }
    { use the default.       }
    if (not PtInTexas(locWhere)) then
    begin
        inherited AdjustCursor(where, mouseRgn);
        exit(AdjustCursor);
    end

    { Set the star cursor. }
    SetCursor(star);

    { Calculate the mouse region }
    { in frame coords.          }
    OpenRgn;
    DrawTexas;
    CloseRgn(TexasRgn);
```

```

        { Convert it to global coords. }
        TexasRect := TexasRgn^^.rgnBBox;
        tempRect := TexasRect;
        FrameToGlobalR(tempRect);
        OffsetRgn(TexasRgn,
            tempRect.left - TexasRect.left
            tempRect.top - TexasRect.top);

        { Set the mouse region. }
        SectRgn(mouseRgn, TexasRgn, mouseRgn);
    end;

```

And this is what the AdjustCursor method might look like in C:

```

void MapPane::AdjustCursor(Point where,
    RgnHandle mouseRgn);
{
    Point        locWhere;
    RgnHandle    TexaxRgn = NewRgn();
    Rect         TexasRect;
    Rect         TempRect;

    locWhere = where;
    WindToFrame(&locWhere);

    /* If we're not in Texas, */
    /* use the default.      */
    if (!ptInTexas(locWhere)) {
        inherited::AdjustCursor(where, mouseRgn);
        return ;
    }

    /* Set the star cursor. */
    SetCursor(star);

    /* Calculate the mouse region */
    /* in frame coords.          */
    OpenRgn();
    DrawTexas();
    CloseRgn(TexasRgn);

    /* Convert it to global coords. */
    TexasRect = (**TexasRgn).rgnBBox;
    tmpRect = TexasRect;
    FrameToGlobalR(&tempRect);
    OffsetRgn(TexasRgn,
        tempRect.left - TexasRect.left,
        tempRect.top - TexasRect.top)

    /* Set the mouse region. */
    SectRgn(mouseRgn, TexasRgn, mouseRgn);
}

```

**GetBalloonInfo**

```
procedure GetBalloonInfo (  
    var helpData: HMMesageRecord; var tip: Point;  
    var altertnateRect: Rect; var tipProc: Ptr;  
    var variant: integer; var method: integer);  
  
void GetBalloonInfo (HMMesageRecord *helpData,  
    Point *tip, Rect *alternateRec, Ptr *tipProc,  
    short *theProc, short *variant, short *method);
```

Set up the parameters needed to show a help balloon. If `helpResIndex` is greater than 0, use it as an index into the 'hrct' resource associated with this view. If the view is a pane, the enclosing window holds the resource id of the 'hrct' resource, otherwise it uses 'hrct' with the ID `kDefaultHelpResID` (128).

This method calls the Help Manager's `HMGetIndHelpMsg` routine to get the information from the 'hrct' resource.

**ShowHelpBalloon**

```
procedure ShowHelpBalloon (helpData: HMMesageRecord;  
    tip: Point; altertnateRect: Rect; tipProc: Ptr;  
    variant: integer; method: integer);  
  
void ShowHelpBalloon (HMMesageRecord *helpData,  
    Point tip, Rect *altRec, Ptr tipProc,  
    short theProc, short variant, short method);
```

This method uses the data from `GetBalloonInfo` and calls the Help Manager routine `HMShowBalloon` to display the help balloon. If there is an error, this method calls `Failure`. `HelpData` comes from the 'hrct' resource. `Tip`, as returned from `GetBalloonInfo` is set to the center of the view, in global coordinates. `AltRect` is the aperture of the view. `TipProc` is `NIL`. `TheProc` is 0, the standard definition function. `Variant` and `method` are both read from the 'hrct' resource.

**GetHelpResID**

```
function GetHelpResID : integer;  
short GetHelpResID (void);
```

Return the resource ID of the 'hrct' resource that has the balloon help information for this view. This method returns the ID of the TCL's default 'hrc' resource: `kDefaultResID`.

**Subview Management methods****AddSubview**

```
procedure AddSubview (theSubview: CView);  
void AddSubview (CView *theSubview);
```

Add theSubview to the view's `itsSubviews` list. You should not use or override this method.

### RemoveSubview

```
procedure RemoveSubview (theSubview: CView);
void RemoveSubview (CView *theSubview);
Remove theSubview from the itsSubviews list.
```

### FindSubview

```
function FindSubview (hitPt: Point): CView;
CView* FindSubview (Point hitPt);
Find the subview that wants clicks and contains hitPt. HitPt is in window coordinates. This method finds the topmost subview. You should not override this method.
```

### FindViewById

```
function FindViewById (anID: longint): CView;
CView* FindViewById (long anID);
Find the subview whose ID is anID. If more than one view has the same ID, this method finds the topmost subview.
```

### MatchView

```
function MatchView (function matchProc(aView: CView;
    matchData: Ptr): Boolean;
    matchData: Ptr): CView;
CView* MatchView (MatchViewProc matchProc,
    void *matchData)
Find the first subview that returns TRUE for matchProc. This method finds the topmost subview.
```

MatchProc is a pointer to function of the form:

```
function matchProc (aView: CView;
    matchData: Ptr);
```

or in C:

```
Boolean matchProc (CView *aView,
    void *matchData);
```

MatchData can be anything that your matchProc needs. For an example of MatchView, see the definition of FindViewById.

### SubpaneLocation

```
procedure SubpaneLocation (hEncl, vEncl: longint;
    var hLocation, vLocation: longint);
void SubpaneLocation (long hEncl, long vEncl,
    long *hLocation, long *vLocation);
```

Return the position of a subview at hEncl, vEncl in hLocation and vLocation in window coordinates.

## Prepare

```
procedure Prepare;
void Prepare (void);
```

Prepare to draw or to do something after a click. The default method sets the class variable `cPreparedView` to `this`. The class variable `cPreparedView` points to the most recently prepared view. `CPane` uses `cPreparedView` to avoid redundant calls to `Prepare`.

View classes must override this method. The overridden methods should set `cPreparedView` to `this` or call the inherited method. See the definition of `Prepare` in `CPane` for an example.

## ForceNextPrepare

```
procedure ForceNextPrepare;
void ForceNextPrepare (void);
```

Clear `cPreparedView` to disable the optimization on the next call to `Prepare`. If you change the port, the clipping region, or the origin of a view, you must call this method to disable the `Prepare` optimization.

## FrameToGlobalR

```
procedure FrameToGlobalR (frameRect: LongRect;
    var globalRect: Rect);
void FrameToGlobalR (LongRect *frameRect,
    Rect *globalRect);
```

Convert `frameRect` from frame coordinates to global coordinates and put the converted coordinates into `globalRect`. The default method does nothing. View subclasses must override this method.

## Class resources

The `IViewRes` method lets you initialize any descendant of `CView` with a resource template.

Resource	Class
Bord	CBorder
Pane	CPane
Pano	CPanorama
PctP	CPicture
ScPn	CScrollPane
StTx	CStaticText, CEditText
View	CView

You can use `ResEdit` to create the resource templates for each class.

---

**Note**

Your package includes a file called `TCL_TMPLs` that contains a set of TMPL resources you can install into ResEdit. These TMPLs let you create and edit the resource above easily. See the instructions in “Installing the TMPL Resources into ResEdit” in Chapter 2 to learn how to install these TMPLs into ResEdit.

---



# CWindow 70 ♦

---

## Introduction

CWindow implements a class necessary to manipulate a Macintosh window.

## Heritage

Superclass	CView
Subclasses	None

## Using CWindow

Virtually every application you write on the Macintosh uses windows. The CWindow class implements methods to manipulate Macintosh windows. In the THINK Class Library, window objects are merely visual entities. They're not designed to interact directly with the application. The class CDirector manages communication between a window and the application. The CDocument class, a subclass of CDirector, manages communication between the application, a window, and a file.

Objects of class CWindow usually belong to directors. Under normal circumstances, you should not need to define a subclass of CWindow or to manipulate one directly. The only thing you really need to do to a window is create it and set its options.

You can specify that a window be modal. If a modal window is the front-most window, CDesktop's `DispatchClick` method will not let a click in another window deactivate the modal window, though mouse clicks in the menu bar are handled. Be sure that your program provides a way to close modal windows.

The Macintosh windows associated with CWindow objects have a window kind of `OBJ_WINDOW_KIND`. If your application uses windows that are not associated with a CWindow object— from a utility library, for instance— you should override the CDesktop methods `DispatchClick` and `DispatchCursor` and the CSwitchboard methods `DoUpdate`,

DoActivate, and DoDeactivate to check the window kind of the Macintosh windows they deal with.

## Variables

The enclosure for all windows must be gDesktop.

### Global variable

Variable	Type	Description
gDesktop	CDesktop	The global desktop which acts as the top of the visual hierarchy and the enclosure for all windows.

### Instance variables

Variable	Type	Description
procID	integer	Window definition ID used to create the window.
sizeRect	Rect	Minimum and maximum size of the window
floating	Boolean	Is this a floating window?
isColor	Boolean	Is this a color window?
isModal	Boolean	Is it currently modal?
actClick	Boolean	Process mouse click which activates window?
location	Point	Current window location, used when "hiding" a window while suspended
helpResID	integer	Resource ID of 'hrct' help resource for panes in this window.

## Methods

### Construction and destruction methods

#### IWindow

```
procedure IWindow (WINDid: integer;
  aFloating: Boolean; anEnclosure: CDesktop;
  aSupervisor: CDirector);

void IWindow (short WINDid, Boolean aFloating,
  CDesktop *anEnclosure,
  CDirector *aSupervisor);
```

Initialize a window object from a WIND resource. If Color QuickDraw is available, this method creates a color window.



WINDid is the id of the WIND resource. If aFloating is true, the window is a floating window. In this case, anEnclosure must refer to a floating window desktop.

AnEnclosure is the view the window belongs to. A window's enclosure should always be gDesktop. ASupervisor is the window's supervisor. Usually a window's supervisor is the director it belongs to. IWindow sends the window a MakeMacWindow message to actually allocate space in memory for the window.

By default, a window is not modal. To make a window modal, use the SetModal method on page 453.

By default, helpResID is set to kDefaultResID. HelpResID is the resource ID of the 'hrct' resource associated with this window. To change it use the SetHelpResID method on page 454. For more information about using Balloon Help with panes, see "Using Balloon Help with views" on page 436.

## INewWindow

```
procedure INewWindow (bounds: Rect;
    fVisible: Boolean; aProcID: integer;
    fFloating: Boolean; Boolean fHasGoAway
    anEnclosure: CDesktop; aSupervisor: CDirector);
void INewWindow (Rect *bounds,
    Boolean fVisible, short aProcID,
    Boolean fFloating, Boolean fHasGoAway,
    CDesktop *anEnclosure,
    CDirector *aSupervisor);
```

Initialize a window from the parameters in the argument list. If Color Quick-Draw is available, this method creates a color window.

Bounds is a rectangle that describes the size and position of the window. AProcID is an integer that describes that type of the window.

If fVisible is TRUE, the window is drawn immediately after it's created. If fHasGoAway is TRUE, the window has a close box in.

The other parameters are identical to IWindow's parameters above.

## Free/Dispose

```
procedure Free;
void Dispose(void);
```

Dispose of the window and all of its subpanes. This method also removes the window from the desktop.

### **MakeMacWindow**

```
procedure MakeMacWindow (WINDid: integer);  
void MakeMacWindow (short WINDid);
```

Create a Macintosh window from a WIND resource. This method lets the Window Manager allocate memory itself. If you want to create your windows differently, override this method. Bear in mind that floating windows should be in front (-1) and that non-floating windows should be behind.

### **MakeNewMacWindow**

```
procedure MakeNewMacWindow (bounds: Rect;  
    aProcID: integer; fHasGoAway: Boolean);  
void MakeNewMacWindow (Rect *bounds, short aProcID,  
    Boolean fHasGoAway)
```

Create a Macintosh window from the parameters in the argument list. Bounds is a rectangle that describes the size and position of the window. AProcID is an integer that describes that type of the window. If FHasGoAway is TRUE, the window will have a close box in the left side of the title bar.

This method lets the Window Manager allocate memory itself. If you want to create your own windows differently, override this method. For example, you might want to perform your own memory allocation or to create color windows. For compatibility with the Desktop class, the convention of putting floating windows in front and non-floating windows in back should be observed. Bear in mind that floating windows should be in front (-1) and that non-floating windows should be behind.

### **Close**

```
procedure Close;  
void Close (void);
```

Close a window. A window object gets this message as a result of a click in the close box. The default method sends the window's supervisor (a director) a CloseWind message.

### **Accessing methods**

### **GetFrame**

```
procedure GetFrame (var theFrame: LongRect);  
void GetFrame (LongRect *theFrame);
```

Get the frame of a window. The frame of the window includes the one pixel border around the window. The top, left of theFrame is always (-1,-1)

### **GetInterior**

```
procedure GetInterior (var theInterior: LongRect);  
void GetInterior (LongRect *theInterior);
```

Get the interior of a window. The interior of the window is the same as its portRect and does not include any of the border pixels. The top left of theInterior is always (0, 0).



<b>GetAperture</b>	<pre>procedure GetAperture (var theAperture: LongRect); void GetAperture (LongRect *theAperture);</pre> <p>Return the drawable area of the window. For windows, this is the entire window. The top, left of theAperture is always (0,0).</p>
<b>IsFloating</b>	<pre>function IsFloating: Boolean; Boolean IsFloating (void);</pre> <p>Return TRUE if this is a floating window.</p>
<b>IsModal</b>	<pre>function IsModal: Boolean; Boolean IsModal (void);</pre> <p>Return TRUE if this is a modal window.</p>
<b>SetModal</b>	<pre>procedure SetModal (fModal: Boolean); void SetModal (Boolean fModal);</pre> <p>Specify whether a window is modal. If fModal is TRUE, the window is modal. If fModal is FALSE, it is not modal. If a modal window is the front-most window, mouse clicks anywhere except in the window and in the menu bar are ignored.</p>
<b>IsColor</b>	<pre>function IsColor: Boolean; Boolean IsColor (void);</pre> <p>Return TRUE if this is a color window.</p>
<b>SetTitle</b>	<pre>procedure SetTitle (theTitle: Str255); void SetTitle (Str255 theTitle);</pre> <p>Set the window's title.</p>
<b>GetTitle</b>	<pre>procedure GetTitle (var theTitle: Str255); void GetTitle (Str255 theTitle);</pre> <p>Get the window's title.</p>
<b>SetActClick</b>	<pre>procedure SetActClick (anActClick: Boolean); void SetActClick (Boolean anActClick);</pre> <p>If anActClick is TRUE, the window processes the mouse click that activated it.</p>
<b>WantsActClick</b>	<pre>function WantsActClick: Boolean; Boolean WantsActClick (void);</pre> <p>Return TRUE if the window processes the mouse click that activates it.</p>

## 70 CWindow

---

### Contains

```
function Contains (thePoint: Point): Boolean;  
Boolean Contains (Point thePoint);
```

Return TRUE if thePoint is inside the window. ThePoint is in global coordinates.

### SetSizeRect

```
procedure SetSizeRect (aSizeRect: Rect);  
void SetSizeRect (Rect *aSizeRect);
```

Set the minimum and maximum size for the window. ASizeRect's top,left are the minimum height and width. ASizeRect's bottom,right are the maximum.

### SetStdState

```
procedure SetStdState (aStdState: Rect);  
void SetStdState (Rect *aStdState);
```

Set the standard state of a window. The standard state is the size and location of the window when it's zoomed out. Before you send this message to a window, make sure that the Macintosh window associated with it supports zooming. If it does, the spareFlag in the window record is true.

### SetHelpResID

```
procedure SetHelpResID (aResID: integer);  
void SetHelpResID (short aResID);
```

Set the resource ID of the 'hrcr' Balloon Help resource associated with this window. For more information about using Balloon Help with the THINK Class Library, see "Using Balloon Help with views" on page 436.

### GetHelpResID

```
function GetHelpResID: integer;  
short GetHelpResID (void);
```

Get the resource ID of the 'hrcr' resource associated with this window. If you do not set one with SetHelpResID, helpResID is set to kDefaultHelpResID by default.

### Appearance methods

### Show

```
procedure Show;  
void Show (void);
```

Show a window. Also sends its enclosure (the desktop) a ShowWind message.

### Hide

```
procedure Hide;  
void Hide (void);
```

Hide a window. Also sends its enclosure (the desktop) a HideWind message.



<b>Activate</b>	<pre>procedure Activate; void Activate (void);</pre> <p>Activate a window and all of its panes. Also sends its supervisor (a director) an <code>ActivateWind</code> message.</p>
<b>Deactivate</b>	<pre>procedure Deactivate; void Deactivate (void);</pre> <p>Deactivate a window and all of its panes. Also sends its supervisor (a director) a <code>DeactivateWind</code> message.</p>
<b>Select</b>	<pre>procedure Select; void Select (void);</pre> <p>Select a window by bringing it to the front and making it active. Sends its enclosure (the desktop) a <code>SelectWind</code> message.</p>
<b>ShowResume</b>	<pre>procedure ShowResume; void ShowResume (void);</pre> <p>Make a window visible when an application resumes. If you hide your windows when your application is suspended, send it this message in its director's <code>Resume</code> method.</p>
<b>HideSuspend</b>	<pre>procedure HideSuspend; void HideSuspend (void);</pre> <p>Hide a window when an application is suspended. To hide a window when the application is being suspended, send the window this message in its director's <code>Suspend</code> method. The <code>HideSuspend</code> method doesn't actually hide the window. Instead, it moves it out of the visible range so the front-to-back ordering of the windows doesn't change.</p>
<b>ShowOrHide</b>	<pre>procedure ShowOrHide (showFlag: Boolean); void ShowOrHide (Boolean showFlag);</pre> <p>Make a window visible or invisible without generating any update or activate events.</p>
<b>Size and location methods</b>	
<b>Drag</b>	<pre>procedure Drag (macEvent: EventRecord); void Drag (EventRecord *macEvent);</pre> <p>Drag a window. This method is invoked when the user clicks in a window's drag region. The default method sends a <code>DragWind</code> message to the desktop.</p>

### Resize

```
procedure Resize (macEvent: EventRecord);  
void Resize (EventRecord *macEvent);
```

Resize a window. This method is invoked when you click and drag the window's grow region. After resizing, this method sends a `ChangeSize` message to the window.

### Zoom

```
procedure Zoom (direction: integer);  
void Zoom (short direction);
```

Zoom a window. This method is invoked when the user clicks in a window's zoom box. Direction is either `inZoomIn` or `inZoomOut`. After changing the size of the window, this method sends an `AdjustToEnclosure` message to its panes.

### Move

```
procedure Move (hGlobal, vGlobal: integer);  
void Move (short hGlobal, short vGlobal);
```

Move the window to the specified location in global coordinates. Note that the new position refers to the top left corner of the window's content region. The title bar will be above the specified coordinates.

### ChangeSize

```
procedure ChangeSize (width, height: integer);  
void ChangeSize (short width, short height);
```

Change the size of the window to the specified width and height. After changing the window size, this method sends an `AdjustToEnclosure` message to its panes. This method respects the maximum and minimum window sizes you set with `SetSizeRect`.

### MoveOffScreen

```
procedure MoveOffScreen;  
void MoveOffScreen (void);
```

Move the window out of the visible area of the desktop.

### Drawing methods

### Update

```
procedure Update;  
void Update (void);
```

Update the window. This method is invoked when the Switchboard processes an update event. This method sends a `Prepare` message to the window and then sends a `Draw` message to each of the window's panes. You should not override this method.

### Prepare

```
procedure Prepare;  
void Prepare (void);
```

Set the port of the window and prepare for drawing.



**Mouse methods****DispatchClick**

```
procedure DispatchClick (var macEvent: EventRecord);  
void DispatchClick (EventRecord *macEvent);
```

Find a pane to handle the mouse click. You should not use or override this method. If you want to handle a click in a particular window subclass, override the DoClick method (inherited from CView) instead.

**DispatchCursor**

```
procedure DispatchCursor (where: Point;  
    mouseRgn: RgnHandle);  
void DispatchCursor (Point where, RgnHandle mouseRgn);
```

Find a pane that handles AdjustCursor messages. This method converts where from global to window coordinates, and then lets the DispatchCursor method in CView take care of displaying Balloon Help if it is enabled. You should not use or override this method.

**Conversion methods****FrameToGlobalR**

```
procedure FrameToGlobalR (frameRect: LongRect;  
    globalRect: Rect);  
void FrameToGlobalR (LongRect *frameRect,  
    Rect *globalRect);
```

Convert frameRect from frame to global coordinates. Place the result in globalRect.



# TCL Library Routines

---

## 71



### Introduction

The THINK Class Library uses some library routines to deal with the Macintosh Toolbox, long coordinates, and memory allocation. The Starter seed projects are set up so you can use these utility routines.

For a detailed description of the exception handling mechanism, see Chapter 8, “Exception Handling.”

For more information about memory management, see “Handling low memory situations” on page 138.

### Toolbox Utilities

These routines make working with parts of the Macintosh Toolbox easier.

#### THINK C programmers

These functions are defined in `TBUtilities.c`. Be sure to include `TBUtilities.h` before you use these functions.

#### THINK Pascal programmers

These routines are defined in `TCL.p`.

#### QuickDraw Utilities

#### DrawSICN

```
procedure DrawSICN (SICNid, index: integer;  
    location: Point);  
void DrawSICN (short SICNid, short index,  
    Point location);
```

Draw a small icon at the point `location`. `SICNid` is the resource id of a SICN resource. `Index` is the number of the small icon within the resource.

## 71 TCL Library Routines

---

### **PinInRect**

```
procedure PinInRect (theRect: LongRect;
    var thePoint: LongPt);
void PinInRect (LongRect *theRect, Point *thePoint);
```

Pin thePoint within theRect. This routine is similar to the Toolbox routine PinRect except that the point is changed in place, and this routine does not subtract 1 at the right and bottom edges.

### **Window Manager Utilities**

### **BringBehind**

```
procedure BringBehind (macWindow,
    behindWindow: WindowPtr);
void BringBehind (WindowPtr macWindow,
    WindowPtr behindWindow);
```

BringBehind moves macWindow so it is behind behindWindow. The CFWDesktop class uses this routine to place the frontmost application window behind all the floating windows.

### **IsDialogWindow**

```
function IsDialogWindow (macWindow: WindowPeek):
    Boolean;
Boolean IsDialogWindow (WindowPeek macWindow);
```

Return TRUE if macWindow is a dialog window.

### **IsMyWindow**

```
function IsMyWindow (macWindow: WindowPeek):
    Boolean);
Boolean IsMyWindow (WindowPeek macWindow);
```

Returns TRUE if macWindow is an application window or a dialog window.

### **IsMyWindow**

```
function IsSystemWindow (macWindow: WindowPeek):
    Boolean);
Boolean IsSystemWindow (WindowPeek macWindow);
```

Returns TRUE if macWindow is a system window (desk accessory window).

### **Dialog Manager Utilities**

### **FindDlogPosition**

```
procedure FindDlogPosition (theType: ResType;
    theID: integer; var corner: Point);
void FindDlogPosition (ResType theType, short theID,
    Point *corner);
```

Return in corner the coordinates of the upper left corner of the dialog or an alert if the dialog were centered in the upper third of the screen. This routine is useful for the standard file dialogs which ask you to supply a corner point. Otherwise, you should use PositionDialog.

**PositionDialog**

```
procedure PositionDialog (theType: ResType;
    theID: integer);

void PositionDialog (ResType theType, short theID);
```

Center the bounding box of a dialog or an alert in the upper third of the screen. TheType is either 'DLOG' or 'ALRT' and theID is the resource ID of the dialog or the alert. PositionDialog does not display the dialog, it just changes its location.

**Font Manager Utility****GetFontNumber**

```
procedure GetFontNumber (fontName: ConstStr255Param;
    var fontNum: integer);

void GetFontNumber (ConstStr255Param fontName,
    short *fontNum);
```

Given a font name, return its font number in fontNum. If the font is not found, return a negative number.

**Keyboard Utilities****KeyIsDown**

```
function KeyIsDown (theKeyCode: integer): Boolean;

Boolean KeyIsDown (short theKeyCode);
```

Return TRUE if the specified key is being held down. Note that the key code is depends on the kind of keyboard. This function does not tell you if a specific key character is being held down.

**AbortInQueue**

```
function AbortInQueue: Boolean;

Boolean AbortInQueue (void);
```

Return TRUE if there is a Command-Period pending in the event queue. If there is, the event is removed from the queue.

**IsCancelEvent**

```
function IsCancelEvent (anEvent: EventRecord):
    Boolean;

Boolean IsCancelEvent (EventRecord *anEvent);
```

Return TRUE if anEvent is a "cancel" event. In US keyboards, a cancel event is Command-Period. For more information about cancelling in international environments, see Tech Note 263.

**String Utilities****ConcatPStrings***C only*

```
void ConcatPString (Str255 first,
    ConstStr255Param second);
```

Concatenate two Pascal strings. The string second is added to the end of first. The result is truncated if it would be longer than 255 bytes. Pascal programmers should use the built-in procedure concat.

## 71 TCL Library Routines

---

### CopyPString

*C only*

```
void CopyPString (ConstStr255Param srcString,  
                  Str255 destString);
```

Copy srcString into destString. The original contents of destString are lost. Pascal programmers should assign one string to another.

## Operating System Utilities

### THINK C programmers

These functions are defined in OSChecks . c. Be sure to include OSChecks . h before you use these functions.

### THINK Pascal programmers

These routines are defined in TCL . p.

### TrapAvailable

```
function TrapAvailable (trapNum: integer): Boolean;  
Boolean TrapAvailable (short trapNum);  
Return TRUE if the specified trap is available.
```

### WNEIsImplemented

```
function WNEIsImplemented: Boolean;  
Boolean WNEIsImplemented (void);  
Return TRUE if the WaitNextEvent is available. WaitNextEvent is  
available in every system from System 6.0.
```

### TempMemCallsAvailable

```
function TempMemCallsAvailable: Boolean;  
Boolean TempMemCallsAvailable (void);  
Return TRUE if the MultiFinder temporary memory calls are available.
```

### ColorQDIsPresent

```
function ColorQDIsPresent: Boolean;  
Boolean ColorQDIsPresent (void);  
Return TRUE if Color QuickDraw is available.
```

## Long Coordinate Utilities

These routines operate on long rectangles and long points. These routines do not map long points from frame coordinates to QuickDraw coordinates or vice versa. To map frame coordinates to QuickDraw coordinates, use the transformation methods in CPane. See "Coordinate transformation methods" on page 331.



### THINK C programmers

These functions are defined in `LongCoordinates.c`. Be sure to include `LongCoordinates.h` before you use these functions.

### THINK Pascal programmers

These functions are defined in `TCL.p`.

### Long point utilities

#### QDToLongPt

```
procedure QDToLongPt (srcPt: Point;
var destPt: LongPt);
void QDToLongPt (Point srcPt, LongPt *destPt);
```

Convert `srcPt` from a QuickDraw point to a long point and place the result in `destPt`.

#### LongToQDPt

```
procedure LongToQDPt (srcPt: LongPt;
var destPt: Point);
void LongToQDPt (LongPt *srcPt, Point *destPt);
```

Convert `srcPt` from a long point to a QuickDraw point and place the result in `destPt`. If `srcPt` is not in QuickDraw space, it is clipped to 16 bits. To convert a long point in frame coordinates to a QuickDraw point, use CPane's `FrameToQD` method.

#### SetLongPt

```
procedure SetLongPt (var pt: LongPt; h, v: longint);
void SetLongPt (LongPt *pt, long h, long v);
```

Set the horizontal and vertical elements of long point `pt` to `h` and `v`.

#### AddLongPt

```
procedure AddLongPt (srcPt: LongPt;
var destPt: LongPt);
void AddLongPt (LongPt *srcPt, LongPt *destPt);
```

Add `srcPt` and `destPt` and return the result in `destPt`.

#### SubLongPt

```
procedure SubLongPt (srcPt: LongPt;
var destPt: LongPt);
void SubLongPt (LongPt *srcPt, LongPt *destPt);
```

Subtract `srcPt` from `destPt` and return the result in `destPt`.

#### EqualLongPt

```
function EqualLongPt (pt1: LongPt;
pt2: LongPt): Boolean;
Boolean EqualLongPt (LongPt *pt1, LongPt *pt2);
```

Return TRUE if `pt1` and `pt2` are the same point.

## 71 TCL Library Routines

---

### **PtInQDSpace**

```
function PtInQDSpace (pt: LongPt): Boolean;  
Boolean PtInQDSpace (LongPt *pt);  
Return TRUE if pt is within 16-bit QuickDraw coordinate space.
```

### **Long rectangle utilities**

### **QDToLongRect**

```
procedure QDToLongRect (srcRect: Rect;  
    var destRect: LongRect);  
void QDToLongRect (Rect *srcRect, LongRect *destRect);  
Convert srcRect from a QuickDraw rectangle to a long rectangle and return the result in destRect.
```

### **LongToQDRect**

```
procedure LongToQDRect (srcRect: LongRect;  
    var destRect: Rect);  
void LongToQDRect (LongRect *srcRect, Rect *destRect);  
Convert srcRect from a long rectangle to a QuickDraw rectangle and return the result in destRect.
```

### **SetLongRect**

```
procedure SetLongRect (var r: LongRect; left, top,  
    right, bottom: longint);  
void SetLongRect (LongRect *r, long left, long top,  
    long right, long bottom);  
Set the elements of the long rectangle r.
```

### **OffsetLongRect**

```
procedure OffsetLongRect (var r: LongRect; dh, dv:  
    longint);  
void OffsetLongRect (LongRect *r, long dh, long dv);  
Offset r by dh and dv. Positive values are to the right and down.
```

### **InsetLongRect**

```
procedure InsetLongRect (var r: LongRect;  
    dh, dv: longint);  
void InsetLongRect (LongRect *r, long dh, long dv);  
Inset the sides of r by dh and dv. Positive values move the sides in.
```

### **SectLongRect**

```
function SectLongRect (src1, src2: LongRect;  
    var destRect: LongRect): Boolean;  
Boolean SectLongRect (LongRect *src1, LongRect *src2,  
    LongRect *destRect);  
Calculate the intersection of src1 and src2, and place the result in destRect. DestRect may be the same as src1 or src2. If destRect is not empty, return TRUE.
```





### UnionLongRect

```
procedure UnionLongRect (src1, src2: LongRect;
    var destRect: LongRect);
void UnionLongRect (LongRect *src1, LongRect *src2,
    LongRect *destRect);
```

Calculate the union of src1 and src2, and place the result in destRect. DestRect may be the same as src1 or src2.

### PtInLongRect

```
function PtInLongRect (pt: LongPt;
    r: LongRect): Boolean;
Boolean PtInLongRect (LongPt *pt, LongRect *r);
```

Return TRUE if the long point pt is in the long rectangle r.

### Pt2LongRect

```
procedure Pt2LongRect (pt1, pt2: LongPt;
    var r: LongRect);
void UnionLongRect (LongPt *pt1, LongPt *pt2,
    LongRect *r);
```

Calculate the smallest rectangle that encloses pt 1 and pt 2 and return the result in the long rectangle r.

### EqualLongRect

```
function EqualLongRect (r1, r2: LongRect): Boolean;
Boolean EqualLongRect (LongRect *r1, LongRect *r1);
```

Return TRUE if the long rectangles r1 and r2 are identical.

### EmptyLongRect

```
function EmptyLongRect (r: LongRect): Boolean;
Boolean EmptyLongRect (LongRect *r);
```

Return TRUE if the long rectangle r encloses no points.

### RectInQDSpace

```
function RectInQDSpace (r: LongRect): Boolean;
Boolean RectInQDSpace (LongRect *r);
```

Return TRUE if the long rectangle r is entirely within QuickDraw space.

## THINK Class Library Utilities

These routines are generally useful for working with the THINK Class Library. Some of them work with the exception handling mechanism (see Chapter 8, "Exception Handling") and with the memory management routines (see "Handling low memory situations" on page 138).

### THINK C programmers

These functions are defined in `TCLUtilities.c`. Be sure to include `TCLUtilities.h` before you use these functions.

### **THINK Pascal programmers**

These routines are defined in `TCL.p`. `EqualMem` is written in assembly language. It is defined in `TCL.lib`.

### **Error reporting utility**

#### **ErrorAlert**

```
procedure ErrorAlert (error: integer;  
    message: longint);
```

```
void ErrorAlert (short error, long message);
```

Display an alert for the given error and message codes. If the low word of message is zero, `ErrorAlert` looks for an 'Estr' resource that matches the error code in `error`. If it can't find one, it displays a generic error message and the error number.

If the low word of message is greater than zero, it is used as an index into a 'STR#' resource. If the high word of message is zero, `ErrorAlert` uses the resource 'STR#' 301. If the high word of message is not zero, its value is added to 1024 to get the resource ID of a 'STR#' resource. You can use the exception handler's `SpecifyMsg` function to build the message.

For instance, if message is 655,363—0x000A0003 hex—, the resource ID of the 'STR#' resource is 1024+10, and the index into it is 3.

If the application is running, `ErrorAlert` uses 'ALRT' `ALRT_Exception` (251), otherwise it uses 'ALRT' `ALRT_ExceptionAbort` (252).

### **Memory allocation utilities**

#### **NewHandleCanFail**

```
function NewHandleCanFail (size: longint): Handle;
```

```
void *NewHandleCanFail (long size);
```

Allocate a handle without drawing on the memory reserve. If the allocation fails, returns NIL.

#### **ResizeHandleCanFail**

```
procedure ResizeHandleCanFail (theHandle: Handle;  
    newSize: longint);
```

```
void ResizeHandleCanFail (void *theHandle, long  
    newSize);
```

Resize a handle without drawing on the memory reserve.

#### **SetAllocation**

```
function SetAllocation (canFail: Boolean): Boolean;
```

```
Boolean SetAllocation( Boolean canFail);
```

Change the parameters that `gApplication` uses when the grow zone function is invoked because the memory manager can't satisfy a request. If `canFail` is TRUE (or `kAllocCanFail`), memory requests are allowed to

fail without drawing on memory reserves. If `canFail` is `FALSE` (or `kAllocCantFail`), the application will try to satisfy the memory request from the reserves. `SetAllocation` returns the previous setting. You should reset the allocation parameters after you make a memory request.

In THINK C, you might do it like this:

```
void AClass::AMethod(void)
{
    Boolean oldAlloc;

    oldAlloc = SetAllocation(kAllocCanFail);
    request memory here
    SetAllocation(oldAlloc);
    fail gracefully if it doesn't succeed
}
```

In THINK Pascal, you could do it this way:

```
procedure AClass.AMethod;
var
    oldAlloc: Boolean;
begin
    oldAlloc := SetAllocation(kAllocCanFail);
    request memory here
    oldAlloc := SetAllocation(oldAlloc);
    fail gracefully if it doesn't succeed
end;
```

This method sends a `RequestMemory` method to `gApplication`. See “Handling low memory situations” on page 138 for more information about working with memory in the THINK Class Library.

## SetCriticalOperation

```
procedure SetCriticalOperation (aCriticalOp: Boolean);
void SetCriticalOperation (Boolean aCriticalOp);
```

Change the parameters that `gApplication` uses when the grow zone function is invoked because the memory manager can't satisfy a request. If `aCriticalOp` is `TRUE`, more of the memory reserve is available to satisfy a failing memory request.

This method sends a `SetCriticalOperation` message to `gApplication`. See “Handling low memory situations” on page 138 for more information about working with memory in the THINK Class Library.

## 71 TCL Library Routines

---

### EqualMem

```
function EqualMem (p1, p2: Ptr; n: longint): Boolean;  
Boolean EqualMem (void *p1, void *p2, long n);
```

Compare two blocks of memory, and return TRUE if they are equal. P1 and p2 point to memory, and n is the number of bytes to compare.

### SetMinimumStack

```
procedure SetMinimumStack (newSize: minSize);  
void SetMinimumStack (long minSize);
```

Set the stack to be at least minSize bytes. If you use this routine, it must appear only once, and it must be the first statement in your program.

#### Memory disposal utilities

These memory utilities release different kinds of memory, and set the variable that references them to NIL. In THINK C, these routines are implemented as macros, which are defined in Global.h. In THINK Pascal they're defined as procedures in TCL.p.

### ForgetHandle

```
procedure ForgetHandle (var handle: Handle);  
void ForgetHandle (Handle h); [MACRO]
```

If h is not NIL, call DisposHandle (h), and set h to NIL.

### ForgetPtr

```
procedure ForgetPtr (var p: Ptr);  
void ForgetPtr (Ptr p); [MACRO]
```

If p is not NIL, call DisposPtr (p), and set p to NIL.

### ForgetResource

```
procedure ForgetResource (var r: Handle);  
void ForgetResource (Handle r); [MACRO]
```

If r is not NIL, call ReleaseResource (r), and set r to NIL.

### ForgetObject

```
procedure ForgetObject (var obj: CObject);  
void ForgetObject (CObject obj); [MACRO]
```

If obj is not NIL, send obj a Dispose or Free message, and set obj to NIL.

## Long Coordinate QuickDraw Utilities

The following routines are long coordinate versions of some common QuickDraw routines. The arguments that these routines take are long points and long rectangles in frame coordinates. If you use these routines, be sure that you call them only in a pane's Draw method because they expect that the pane has already been prepared.



---

**Note**

These routines transform frame coordinates to QuickDraw coordinates every time you call them. It's usually more convenient to do the transformation once, then do all the drawing in QuickDraw coordinates.

---

**THINK C programmers**

These functions are defined in `LongQD.c`. Be sure to include `LongQD.h` before you use these functions.

**THINK Pascal programmers**

These routines are defined in `LongQD.p`.

**Long rectangle drawing routines**

**LEraseRect**

```
procedure LEraseRect (area: LongRect);  
void LEraseRect (LongRect *area);  
Erase the rectangle specified in area.
```

**LFrameRect**

```
procedure LFrameRect (area: LongRect);  
void LFrameRect (LongRect *area);  
Frames the rectangle specified in area.
```

**LInvertRect**

```
procedure LInvertRect (area: LongRect);  
void LInvertRect (LongRect *area);  
Invert the rectangle specified in area.
```

**LPaintRect**

```
procedure LPaintRect (area: LongRect);  
void LPaintRect (LongRect *area);  
Paint area in the current pattern and mode.
```

**LFillRect**

```
procedure LFillRect (area: LongRect; pat: Pattern);  
void LFillRect (LongRect *area, Pattern pat);  
Fill area in the pattern pat in patCopy mode.
```

**Long oval drawing routines**

**LEraseOval**

```
procedure LEraseOval (area: LongRect);  
void LEraseOval (LongRect *area);  
Erase the oval specified in area.
```

## 71 *TCL Library Routines*

---

<b>LFrameOval</b>	<pre>procedure LFrameOval (area: LongRect); void LFrameOval (LongRect *area);</pre> Erase the oval specified in area.
<b>LInvertOval</b>	<pre>procedure LInvertOval (area: LongRect); void LInvertOval (LongRect *area);</pre> Invert the oval specified in area.
<b>LPaintOval</b>	<pre>procedure LPaintOval (area: LongRect); void LPaintOval (LongRect *area);</pre> Paint area in the current pattern and mode.
<b>LFillOval</b>	<pre>procedure LFillOval (area: LongRect; pat: Pattern); void LFillOval (LongRect *area, Pattern pat);</pre> Fill area in the pattern pat in patCopy mode.
<b>Long rounded rectangle drawing routines</b>	
<b>LEraseRoundRect</b>	<pre>procedure LEraseRoundRect (area: LongRect;     ovWidth, ovHeight: integer); void LEraseRoundRect (LongRect *area,     short ovWidth, short ovHeight);</pre> Erase the rounded rectangle specified in area.
<b>LFrameRoundRect</b>	<pre>procedure LFrameRoundRect (area: LongRect;     ovWidth, ovHeight: integer); void LFrameRoundRect (LongRect *area,     short ovWidth, short ovHeight);</pre> Erase the rounded rectangle specified in area.
<b>LInvertRoundRect</b>	<pre>procedure LInvertRoundRect (area: LongRect;     ovWidth, ovHeight: integer); void LInvertRoundRect (LongRect *area,     short ovWidth, short ovHeight);</pre> Invert the rounded rectangle specified in area.
<b>LPaintRoundRect</b>	<pre>procedure LPaintRoundRect (area: LongRect;     ovWidth, ovHeight: integer); void LPaintRoundRect (LongRect *area,     short ovWidth, short ovHeight);</pre> Paint area in the current pattern and mode.



## **LFillRoundRect**

```
procedure LFillRoundRect (area: LongRect;
    ovWidth, ovHeight: integer; pat: Pattern);
void LFillRoundRect (LongRect *area,
    short ovWidth, short ovHeight, Pattern pat);
```

Fill area in the pattern pat in patCopy mode.

### **Long bit transfer routine**

## **LCopyBits**

```
procedure LCopyBits (srcBits, dstBits: BitMap;
    srcRect, dstRect: LongRect;
    mode: integer; mask: RgnHandle);
void LCopyBits (BitMap *srcBits, BitMap *dstBits,
    LongRect *srcRect, LongRect *dstRect,
    short mode, RgnHandle mask);
```

Transfer a bit image from srcBits to dstBits. For a full description of this routine, see the description of CopyBits in *Inside Macintosh I*.

### **Long point pen routines**

## **LMoveTo**

```
procedure LMoveTo (hLoc, vLoc: longint);
void LMoveTo (long hLoc, long vLoc);
```

Move the pen to the point hLoc, vLoc.

## **LLineTo**

```
procedure LLineTo (hLoc, vLoc: longint);
void LLineTo (long hLoc, long vLoc);
```

Draw a line from the current pen location to hLoc, vLoc.

### **Long region utility routines**

## **LRectRgn**

```
procedure LRectRgn (rgn: RgnHandle; rect: LongRect);
void LRectRgn (RgnHandle rgn, LongRect *rect);
```

Create a region whose shape is specified by rect. Rgn must be a handle to an existing region.

## **LClipRect**

```
procedure LClipRect (r: LongRect);
void LClipRect (LongRect *r);
```

Change the clipping region of the current grafPort to the long rectangle r.

## 71 *TCL Library Routines*

---



# Global Variables

## 72

---

### Introduction

This chapter describes the global variables in the THINK Class Library.

### Global Objects

These globals hold pointers to the only instances of some classes. Most of them are initialized in the application initialization routines. You should not need to reset any of these variables yourself, except for `gSleepTime`.

#### **gApplication**

```
gApplication: CApplication;  
CApplication *gApplication;
```

The global application object. You should set this variable to an instance of your application class in your main program. See “Writing the main program” in *CApplication* on page 137 for an example.

#### **gDesktop**

```
gDesktop: CDesktop;  
CDesktop *gDesktop;
```

The desktop. This variable holds the only instance of *CDesktop* (or *CFWDesktop* if you're using floating windows). It's initialized in the application method *MakeDesktop*. The desktop is the enclosure for all the windows in your application. It is the top of the visual hierarchy.

#### **gBartender**

```
gBartender: CBartender;  
CBartender *gBartender;
```

The bartender. This variable holds the only instance of *CBartender*. It's initialized in the application method *SetUpMenus*. The bartender converts menu choices into command numbers and handles most menu operations.

## 72 Global Variables

---

### **gClipboard**

```
gClipboard: CClipboard;  
CClipboard *gClipboard;
```

The clipboard. This variable holds the only instance of CClipboard. It's initialized in the application method `MakeClipboard`. The clipboard is where data is cut and copied to and pasted from.

### **gGopher**

```
gGopher: CBureaucrat;  
Cbureaucrat *gGopher;
```

The gopher is the first bureaucrat to get commands. If the gopher can't handle the command, it passes control to its supervisor. Initially, the gopher is set to the application (`gApplication`). When a document becomes active, the gopher points to the document. A document can set the gopher to point to one of its panes.

### **gError**

```
gError: CError;  
CError *gError;
```

The global error handler. This variable is initialized in `IApplication`.

### **gDecorator**

```
gDecorator: CDecorator;  
CDecorator *gDecorator;
```

The window dresser. This variable holds the only instance of CDecorator. It's initialized in the application method `MakeDecorator`. The decorator takes care of arranging windows on the screen.

## **Mouse Click Globals**

The THINK Class Library uses these globals to count mouse clicks. The only variable you'll need to use is `gClicks`.

### **gLastMouseDown**

```
gLastMouseDown: EventRecord;  
EventRecord gLastMouseDown;  
Event record of the last mouse-down event.
```

### **gLastMouseUp**

```
gLastMouseUp: EventRecord;  
EventRecord gLastMouseUp;  
Event record of the last mouse-up event.
```

### **gLastViewHit**

```
gLastViewHit: CView;  
CView *gLastViewHit;  
The last view the mouse went down in.
```

**gClicks**

```
gClicks: integer;
short gClicks;
```

Click counter. This variable counts multiple clicks. Its value is 1 for a single click, 2 for a double click, 3 for a triple click, etc. To be considered a multiple click the mouse must go down in the same view as the last mouse down within the amount of time specified by `GetDb1Time` and the view method `HitSamePart` must return `TRUE`.

**Cursors**

These cursor handles are initialized in `IApplication`. You can use them as arguments to `SetCursor`. Remember that these are handles, so you'll have to call `SetCursor` like this in THINK Pascal:

```
SetCursor(gWatchCursor^);
```

And like this in THINK C:

```
SetCursor(*gWatchCursor);
```

**gIBeamCursor**

```
gIBeamCursor: CursHandle;
CursHandle gIBeamCursor;
I-beam for text views.
```

**gWatchCursor**

```
gWatchCursor: CursHandle;
CursHandle gWatchCursor;
Watch cursor for waiting.
```

**System Globals**

You can use these globals to get information about the environment that your application is running in, and about the state of your application.

**gSystem**

```
gSystem: tSystem;
tSystem gSystem;
```

This global is a record that contains fields that tell you about the capabilities of Macintosh that your program is running under. It's initialized in the `InspectSystem` method of `CApplication`.

## 72 Global Variables

In THINK C, `tSystem` is declared like this:

```
typedef struct
{
    Boolean    hasWNE           : 1;
    Boolean    hasColorQD      : 1;
    Boolean    hasGestalt       : 1;
    Boolean    hasAppleEvents   : 1;
    Boolean    hasAliasMgr      : 1;
    Boolean    hasEditionMgr    : 1;
    Boolean    hasHelpMgr       : 1;
    Boolean    hasScriptMgr     : 1;
    Boolean    hasFPU           : 1;
    short      scriptsInstalled;
    short      systemVersion;
} tSystem;
```

In THINK Pascal it's defined like this:

```
type
    tSystem = packed record of
        hasWNE,
        hasColorQD,
        hasGestalt,
        hasAppleEvents,
        hasAliasMgr,
        hasEditionMgr,
        hasHelpMgr,
        hasScriptMgr,
        hasFPU           : Boolean;
        scriptsInstalled,
        systemVersion    : integer;
    end;
```

The field `scriptsInstalled` tells you how many scripts are in use. The field `systemVersion` gives you the version of the Macintosh System that's running. The version number is given as two byte-long numbers. For example, if `systemVersion` is `0x0607`, the System version number is 6.0.7.

### **gSleepTime**

```
gSleepTime: longint;
long gSleepTime;
```

The switchboard uses this value to pass to `WaitNextEvent`. It is the maximum time (in ticks) between events. Perform methods in `CChore` subclasses and `Dawdle` methods in `CBureaucrat` subclasses can change this value indirectly to force an idle event.

**gInBackground**

`gInBackground: Boolean;`  
`Boolean gInBackground;`  
 TRUE if the application is in the background.

**Error Globals**

These global variables let you see what error raised the last exception. The values of these variables is valid only within a catch handler. At any other time, they're undefined. For more information about exception handling, see Chapter 8, "Exception Handling."

**gLastError**

`gLastError: integer;`  
`short gSignature;`  
 The error that raised an exception.

**gLastMessage**

`gLastMesage: longint;`  
`long gLastMessage;`  
 The message passed to `Failure` at the last exception.

**Utility Globals****gSignature**

`gSignature: OSType;`  
`OSType gSignature;`  
 The signature of your application. You should initialize this variable in your `SetUpFileParameters` application method. Use this variable whenever your application needs to create a file.

**gUtilRgn**

`gUtilRgn: RgnHandle;`  
`RgnHandle gUtilRgn;`  
 Utility region. This region is initialized with `NewRgn` in `IApplication`. You can use it wherever you need a region, but you should not rely on it being the same across calls to different methods.

**Warning**

If you use `gUtilRgn`, be sure that you never dispose of it. If you've created a complex region, and you want to release some memory, use `SetEmptyRgn` to make it as small as possible.

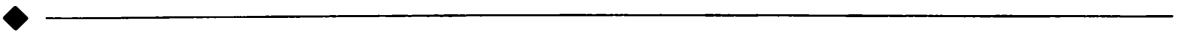


# THINK Pascal ◆

## *Appendix*

### *Part Four*

#### A MacApp and THINK Pascal





# MacApp and THINK Pascal

---

# A

**T**his appendix tells you what you need to know to use Apple's MacApp with THINK Pascal. MacApp was written specifically for Apple's Macintosh Programmer's Workshop (MPW) and, it takes advantage of some idiosyncrasies and features of that environment and its MPW Pascal compiler. To use MacApp with THINK Pascal, you need to modify some of the MacApp files.

---

## Note

THINK Pascal 4.0 only supports MacApp 2.0.1 and MacApp 2.0.2. Make sure that you have the correct version of MacApp from Apple.

---

## Contents

What is MacApp? . . . . .	483
Before you begin . . . . .	483
Take your time . . . . .	483
Minimum Requirements . . . . .	484
Memory Requirements . . . . .	484
Running MacApp with MultiFinder . . . . .	484
Running MacApp with the Finder . . . . .	484
Compatibility . . . . .	484
Installing MacApp Files for THINK Pascal . . . . .	484
Converting MacApp . . . . .	486
What the Pascal Source Converter does . . . . .	486
Make sure you have enough disk space . . . . .	486
Convert MacApp . . . . .	486
Clean up . . . . .	489
Building Your Seed Projects . . . . .	490
What's in the MacApp Seeds folder? . . . . .	490
Build the projects . . . . .	491
The compiler variables . . . . .	492
Segmentation . . . . .	493

## ◆ A *MacApp and THINK Pascal*

---

Using the Seed Projects . . .	. 493
Creating Resource Files for MacApp .	. 494
Using SAREz . . . . .	. 494
Using the prebuilt resource files .	. 495
Environment Differences . . . . .	. 496
Toolbox initialization . . . . .	. 496
Busy cursor . . . . .	. 496
Segmentation in MacApp . . . . .	. 496
UObject and instantiation by name .	. 497

## What is MacApp?

MacApp is a collection of classes that help you build a Macintosh application. MacApp is available directly from Apple through APDA. You don't need to use MacApp to build Macintosh applications with THINK Pascal. The THINK Class Library included with your THINK Pascal package is an alternative class library that you can use as the foundation for your applications.

---

### Note

THINK Pascal 4.0 only supports MacApp 2.0.1 and MacApp 2.0.2. Make sure that you have the correct version of MacApp from Apple.

---

### Before you begin

Make sure that you've installed THINK Pascal according to the instructions in Chapter 2 of your *THINK Pascal User Manual*. You don't need to install the THINK Class Library to use MacApp.

Make sure that you've installed MacApp according to the instructions in your MacApp documentation. You should have a folder called MacApp somewhere on your disk. If you're using MPW, this folder should be in your MPW folder. If not, the MacApp folder can be anywhere on your disk that's convenient for you.

If you don't use MPW, these are the general instructions for installing MacApp:

1. Create a folder called MacApp
2. Copy the contents of all the MacApp disks to the MacApp folder.
3. Combine the contents of all folders named things like *More Thing*, *Even More Thing*, *Even More Thing!* into one folder called *Thing*

Even though the converter doesn't alter your original MacApp source files, be sure that you have a backup copy of your entire MacApp package.

### Take your time

Installing and setting up the files that you need to use MacApp with THINK Pascal will take a couple of hours. You'll be copying files from your THINK Pascal package, moving files from your original MacApp package, converting the MacApp sources, and building seed projects for THINK Pascal. Take your time to make sure you do everything right.

### Minimum Requirements

To use MacApp with THINK Pascal, you need a Macintosh with at least 4 megabytes of RAM. You need a hard disk with at least 2 megabytes for MacApp and 1 megabyte per built project. If you want to keep your original copy of MacApp on disk, you'll need another 2 megabytes.

### Memory Requirements

MacApp takes up quite a bit of memory. To make the best use of MacApp, you need at least 4Mb of RAM. You should set the zone size in the **Run Options...** dialog of the **Run** menu to be at least 1536K. You should set the stack size to be 32K, though you may be able to use as little as 16K.

### Running MacApp with MultiFinder

If you're using System 6.0, you may want to not run under MultiFinder because MacApp with THINK Pascal takes up so much memory. If you do want to run with MultiFinder or System 7.0, be sure that you make the THINK Pascal partition size about 3072K or larger.

### Running MacApp with the Finder

When you use THINK Pascal with the Finder, it will use as much memory as there is on your Macintosh (except for what's in the System heap).

### Compatibility

Since both THINK Pascal and MacApp intercept several Toolbox traps, you'll have an easier time if you remove as many non-essential INITs as possible. Removing INITs not only frees up memory, it also ensures that you're running in a clean environment.

### Installing MacApp Files for THINK Pascal

Your THINK Pascal package includes several files that you need to make MacApp work with THINK Pascal. These files include the Pascal Source Converter, which converts the MacApp sources so they're compatible with THINK Pascal, and prebuilt projects for a generic MacApp application and the example programs that come with MacApp. All of these files are compressed in the file called MacApp 2.0 for THINK Pascal.sea.

To install these files on your disk, double-click on the file MacApp 2.0 for THINK Pascal.sea. When you're asked for a folder to extract to, choose the Development folder that you installed THINK Pascal into.

When the installation is complete, there should be a folder called THINK MacApp in your THINK Pascal 4.0 Folder and a folder called MacApp 2.0 for THINK Pascal 4.0 in your Development folder.



The MacApp 2.0 for THINK Pascal 4.0 folder contains these files and folders:

File or Folder	Description
Pascal Source Converter	An application that converts MPW Pascal programs into THINK Pascal programs.
Generic.Script	A script for the Pascal Source converter that converts MPW Pascal source files into THINK Pascal source files.
MacApp Seeds	A set of pre-segmented MacApp projects ready for you to add your own files. You'll use these projects as the seeds for all your MacApp projects.
MacApp.Script	A script for the Pascal Source Converter that converts MacApp 2.0.1 or MacApp 2.0.2 into a form that THINK Pascal can use.
•Samples From MacApp•	A set of projects for the MacApp sample programs.

The Pascal Source Converter converts your MacApp sources so they're compatible with THINK Pascal. The MacApp.Script is a file that the converter uses to convert the files. The converter places the converted source files in folders within the THINK MacApp folder which is in the THINK Pascal 4.0 Folder.

---

### Warning

Don't change the organization of the THINK MacApp folder or the MacApp 2.0 for THINK Pascal 4.0 folder until after you have converted your MacApp sources. MacApp.Script relies on this folder organization.

---

Next, you need to copy some files from your original MacApp package to the THINK MacApp folder. These files are used to build the resource files that MacApp needs. The main reason for copying them to the THINK MacApp folder is to make them more convenient.

Copy the folder RIncludes from the Interfaces folder in the original MacApp distribution disks to your THINK MacApp folder.

## ◆ A *MacApp and THINK Pascal*

---

Everything is in place, and now you're ready to convert the MacApp sources.

### **Converting MacApp**

This section tells you how to use Pascal Source Converter to make your MacApp sources work with THINK Pascal. The converter is an application that modifies source files written for MPW Pascal so they're compatible with THINK Pascal. It uses the script `MacApp.Script` to convert the MacApp sources.

#### **What the Pascal Source Converter does**

The Pascal Source Converter changes the MacApp sources so they don't need to rely on features specific to MPW Pascal. For instance, it processes the MPW Pascal `$INCLUDE` directive to either include a file or to replace it with a unit name in a `USES` clause. It turns MPW Pascal compiler directives into equivalent THINK Pascal compiler directives. The converter also splits up the file `UMacApp.p` into several units according to classes.

You can use the Pascal Source Converter to convert your own programs that you've written for MPW. To learn how to use the Pascal Source Converter, see Chapter 20 of your *THINK Pascal User Manual*.

#### **Make sure you have enough disk space**

The converter needs around two megabytes of free disk space to work. After you convert your MacApp files, you can remove the original files from your disk.

#### **Convert MacApp**

To begin, double-click on the file `MacApp.Script`. This is what the file's icon looks like in the Finder:



**Figure A-1** The `MacApp.Script` icon

The source converter asks you to find the folder that contains the original MacApp sources:

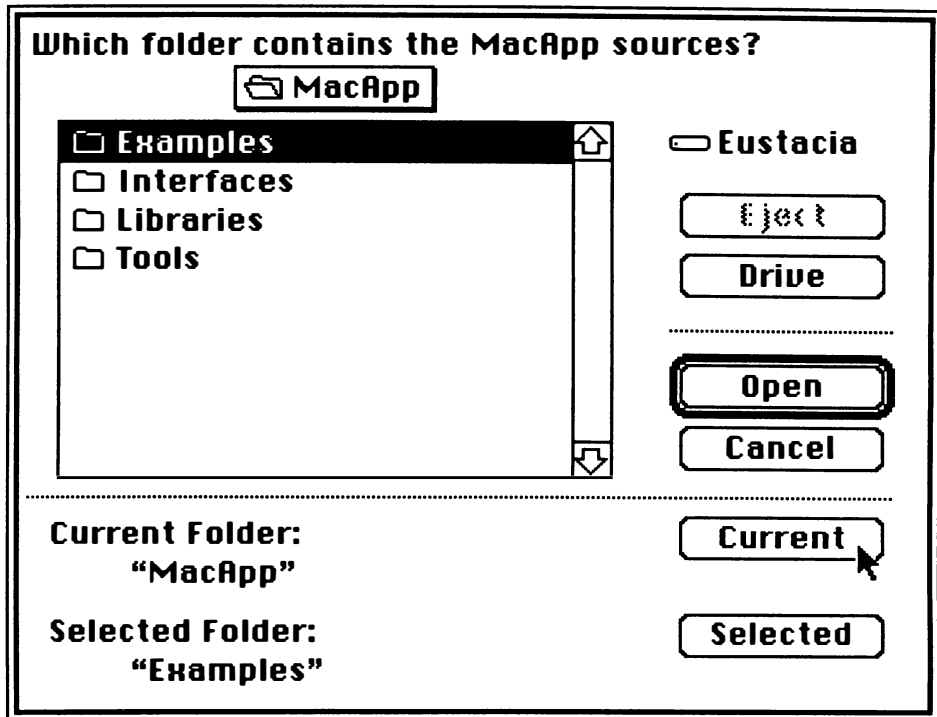


Figure A-2 Pascal Source Converter asking for original MacApp files

This folder may be in your MPW folder. Select the folder and click on the Current or Selected button. The text next to each button tells you which folder the button selects.

#### Note

The Current button chooses the folder that you're in—that is, the folder that's displayed in the pop-up menu in the dialog. The Selected button chooses the folder that's highlighted in the folder list. The prompts let you know which folder each button chooses.

Next, the source converter asks you to find the THINK MacApp folder. The THINK MacApp folder should be in the THINK Pascal 4.0 Folder. This folder contains additional files that the converter uses to convert

## A MacApp and THINK Pascal

MacApp for THINK Pascal. The converted MacApp files will end up in the THINK MacApp folder.

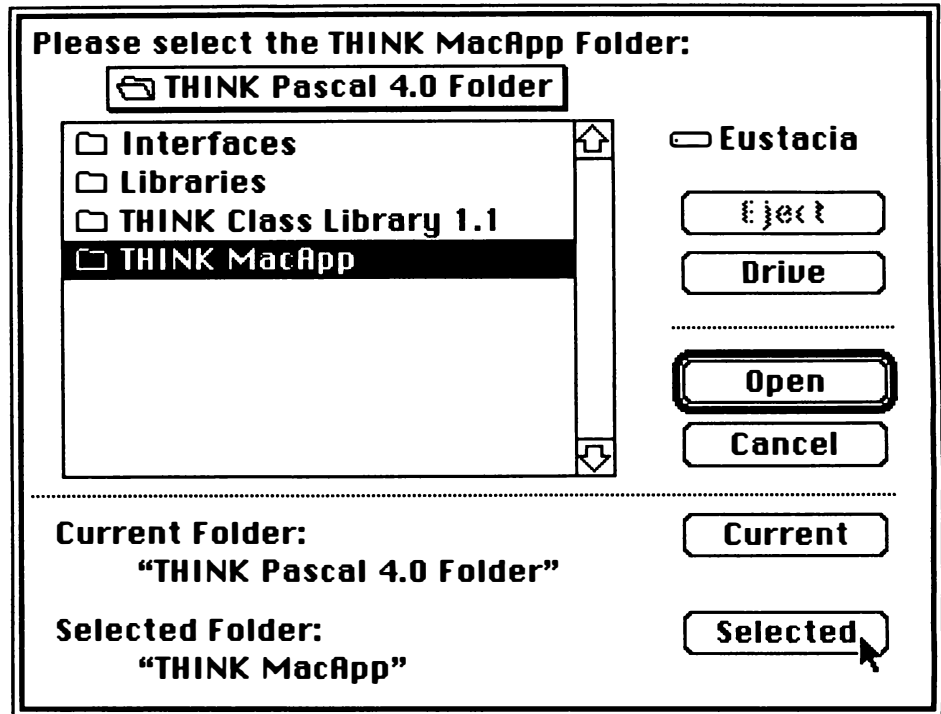


Figure A-3 Pascal Source Converter asking for the THINK MacApp folder

Use the Current or Selected button as before to choose the THINK MacApp folder.

The Pascal Source Converter asks for one more folder, the •Samples From MacApp• folder. Find it on your disk and click on the Current or Selected button.



From here on, the converter does all the work. Conversion takes five to ten minutes depending on the kind of Macintosh you're using. As it's converting the MacApp files, the converter displays a status window like this:

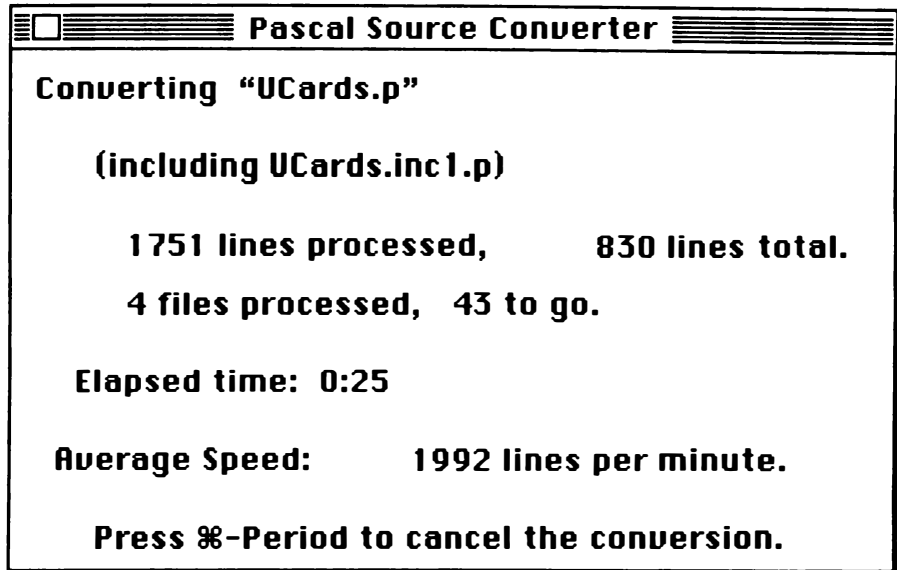


Figure A-4 Pascal Source Converter displaying status

When it finishes converting the MacApp files, the converter displays an alert that tells you that it is done.

### Clean up

Now that the converter is finished, you can delete some files to make more room on your hard disk. You can remove these two folders from the THINK MacApp folder:

- Diffs folder
- Templates folder

You can remove the file MacApp.Script from the THINK Pascal Folder. If you like, you can also delete the Pascal Source Converter from the THINK Pascal Folder, but you may want to use it to convert any existing MPW Pascal source files.

### Note

To learn about the Pascal Source Converter in detail, see Chapter 20 in your *THINK Pascal User Manual*.

---

If you don't use MPW, and you need to make room on your hard disk, you can remove the original MacApp source files. You'll probably want to keep the ViewEdit application and the .R and .Rsrc files from the Libraries folder. Be sure that the original MacApp files are backed up in a safe place.

## Building Your Seed Projects

Now that you've converted the MacApp source files, you're almost ready to use MacApp with THINK Pascal. The next thing you need to do is build your seed projects. The seed projects are prebuilt, precompiled, and preconfigured THINK Pascal projects that you use to build a MacApp-based application.

Building the seed projects may take a couple of hours— particularly if you're on a slower machine like a Macintosh Plus— but it will be time well spent. If you take time to build the seed projects now, you'll save time in the future. Just as important as saving time, building the seed projects now will test that the conversion was completely successful.

### What's in the MacApp Seeds folder?

The MacApp Seeds folder in the MacApp 2.0 for THINK Pascal 4.0 folder contains four THINK Pascal projects and two resource files. The seed projects have all of the MacApp source files already added to them, and the project is segmented correctly. Only the options are different among all the projects.

Since THINK Pascal needs to turn off the Debug option and recompile the entire project when you build your final application, you'll save a lot of time if you build two projects. One project, MacApp.π has the Debug compiler option on for all files. This is the project you'll use while you develop your application in the THINK Pascal environment. The other project, MacApp. - Build.π has the Debug compiler option turned off. This is the project you'll use when you're ready to build the double-clickable final version of your application.

If you plan to use MacApp's built-in debugger, use the MacApp.Debug.π and MacApp.Build.Debug.π projects. MacApp.Debug.π is identical to MacApp.π except that it presets the compiler-variables that control whether to use the MacApp debugger. The MacApp.Build.Debug.π project is just like the MacApp.Build.π project, but with the debugger flags set. The

section “The compiler variables” below describes which compiler-variables are set in each project.

---

**Note**

The MacApp debugger is a set of classes and routines that let you examine existing objects while your program is running. Since the debugger is a part of your application, you can use the MacApp debugger in a built application.

---

Here’s a summary of what each project is for:

<b>Project name</b>	<b>Used for...</b>
MacApp.π	running in the THINK Pascal environment without the MacApp debugger
MacApp.Debug.π	running in the THINK Pascal environment with the MacApp debugger
MacApp.Build.π	building a stand-alone application without the MacApp debugger
MacApp.Build.Debug.π	building a stand-alone application with the MacApp debugger

The Build, Debug, and π suffixes are naming conventions that you may want to adopt. You don’t have to name your project this way.

**Build the projects**

To start building the seed projects, double-click on MacApp.π to open it. Next, choose the **Build** command from the **Run** menu. THINK Pascal starts building the project, loading libraries and compiling source files. This build will take about 45 minutes on a Macintosh Plus and about 15 minutes on a Macintosh IIcx.

---

**Note**

If you’re reading this at your machine, you might want to start building now, and keep reading while you wait.

---

If THINK Pascal can’t find a file as it’s building the MacApp.π project, and you can’t find it manually, it probably means that something went wrong during the MacApp conversion. Make sure that you started with a complete set of the original MacApp source, and try the conversion again.

## A MacApp and THINK Pascal

### Warning

Never try to run the .Build.π project under the environment. It is only for building stand-alone applications.

### The compiler variables

This table tells you which options are on for the different seed projects. A dash (-) means that the compile-time variable is set to 0, and a bullet (•) means that the compile-time variable is set to 1.

Compile-time variable	MacApp.π	MacApp. Build.π	MacApp. Debug.π	MacApp. Debug.Build.π
qDebug	-	-	•	•
qWriteLnWin	-	-	•	•
qTrace	-	-	•	•
qRangecheck	-	-	-	-
qNames	•	-	•	•
qTemplateViews	•	•	•	•
qWWNeeded	-	-	•	•
qProceduralViews	•	•	•	•
qWriteTemplates	•	•	•	•
qInspector	-	-	•	•
qUnInit	-	-	-	-
qDebugTheDebugger	-	-	-	-
qNeedsHierarchialMenus	-	-	-	-
qNeedsStyleTextEdit	-	-	-	-
qNeedsWaitNextEvent	-	-	-	-
qNeedsMC68020	-	-	-	-
qNeedsMC68030	-	-	-	-
qNeedsFPU	-	-	-	-
qNeedsScriptManager	-	-	-	-
qNeedsROM128K	-	-	-	-
qNeedsColorQD	-	-	-	-
qMacApp	•	•	•	•
qPerform	-	-	-	-
qBusyCursor	-	•	-	•

**Table A-1** MacApp compiler variables

You may want to change some of these compiler variables in your project to suit the kind of application you're writing. Your MacApp documentation explains what these compiler variables are for.

### Segmentation

MacApp sources sometimes make assumptions about how MacApp is segmented. You should not rearrange the segments in the seed projects unless you know what you're doing. Be sure to follow Apple's recommendations for segmenting your own MacApp-based program.

Particularly make sure that you do not add any additional entries to the «%\_MethTables» segment. You should also make sure that your main program is in the «Main» segment and that it is the first segment in the project.

---

#### Note

To learn how to segment your program in THINK Pascal, see Chapter 7 of your *THINK Pascal User Manual*.

---

## Using the Seed Projects

Whenever you start a new MacApp-based program, copy the MacApp Seeds folder. Rename the folder and the projects inside the folder to the name of your program. You may or may not want to change the names of the resource files.

---

#### Note

If you change the names of the resource files, be sure to change the resource file setting in the **Run Options...** dialog box.

---

To use the seed projects, use the **Add File...** command to add your program files to the project. Since all of the MacApp files are already built, THINK Pascal will have to compile only the new files that you just added.

To see how the seed projects work, make a duplicate of the four projects and drag them into one of the folders inside the •Samples From MacApp• folder. These folders contain the converted source code for the example programs that came with your original MacApp package. All you need to do is add the converted source files to the project and use the **Run Options...** command to set the resource file.

### Note

These folders already have resource files in them, so you don't need to copy the `MacApp.rsrc` or `MacApp.Debug.rsrc` resource files.

---

## Creating Resource Files for MacApp

MacApp applications rely on several resources being present when you run them. When you use MacApp with MPW Pascal, you use the MPW tools Rez, DeRez, and PostRez to create these resource files from resource scripts called description files. Your THINK Pascal package comes with stand-alone versions of these applications called SAREz, SADeRez, and SAPostRez. You can find these applications in the THINK Pascal Utilities folder. They're described in detail in the *THINK Pascal User Manual*.

### Using SAREz

MacApp uses different resources depending on whether you're using the MacApp debugger. To have SAREz create the correct resources for the MacApp debugger, you need to set certain symbols. Your THINK Pascal package includes two files that define the necessary symbols for both debugger and non-debugger versions of the resource files. The file `Settings.R` defines the symbols so Rez doesn't produce these debugging resources. The file `Debug.Settings.R` sets the symbols so Rez produces the resources that the MacApp debugger needs.

When you build your resource file in MPW, you fold in your application's CODE resources with the other resources. In THINK Pascal, you use the **Run Options...** command to tell THINK Pascal which file contains the resources that project needs. If you're working with an existing Rez resource description file, you should remove any lines of the form:

```
include $$Shell("ObjApp") "MyApplication" 'CODE';
```

THINK Pascal takes care of merging your application's CODE resources with the other resources.

Here are step-by-step instructions for creating a resource file for MacApp:

1. Double-click on the SAREz application to launch it.
2. Click on the Resource Output file pop-up menu and choose the command that lets you create a new file.
3. Give your resource file a name. You should make sure that the resulting resource file is in the same folder as your project.
4. Set the new file's type to 'rsrc' and its creator to 'RSED'.
5. Check the Redefined types OK check box.
6. Click on the Description files button. This button lets you tell SAREz where your resource description files are.
7. If you're not using the MacApp debugger, choose the file Settings.R from the THINK MacApp folder. If you are using the MacApp debugger, choose the file Debug.Settings.R from the THINK MacApp folder.
8. Choose the description files for your programs, then click on the Done button.
9. Click on the #include Paths button. This button lets you specify where SAREz should look for #include files.
10. Choose the RIncludes folder from the THINK Pascal Folder, the RIncludes folder from the THINK MacApp folder, and any addition directories that contain your own type description files, then click Done.
11. Click on the Include Paths button. This button lets you specify where SAREz should look for include files.
12. If you're not using the MacApp debugger, choose the Resources folder in the THINK MacApp folder. Otherwise, choose the Debug Resources folder. If you have any precompiled resource files to merge in to the compiled resource file, then choose the directories that contain those files.
13. If you want to save the settings, choose the **Save...** command from the **File** menu. This command lets you save SAREz settings in a file that you can open later.
14. When everything is done, click the SAREz button to start the compilation process.

If the resource compilation is successful, launch SAPostRez tool to convert the cmnu resources into MENU and mntb resources. Double-click on SAPostRez, and choose the resource file you just created.

### Using the prebuilt resource files

The MacApp Seeds folder contains two resource files: MacApp.rsrc and MacApp.Debug.Rsrc. These resource files contain the standard resources that you need to run programs with and without the MacApp debugger.

## ◆ A *MacApp and THINK Pascal*

---

If you like, you can use these two resource files as the basis for your resource files. Both of these files were built by merging the resources in either the Resources or Debug Resources folders in the THINK MacApp folder.

### Environment Differences

When you run your application in the THINK Pascal environment, you're not running in exactly the same kind of environment that your final application will run in. Though THINK Pascal does a good job of simulating an application environment, there are some things that you need to be aware of.

Some of the differences arise because THINK Pascal needs to take some time away from your application so its debugging tools can monitor what's going on. Others have to do with some assumptions that MacApp makes about the environment it's running in. And some differences just have to do with THINK Pascal being a different compiler than MPW Pascal.

#### Toolbox Initialization

Since MacApp handles all of the Toolbox initialization for you, be sure that you turn off THINK Pascal's automatic initialization with the { \$I- } directive. If you don't, your application will crash.

#### Busy cursor

When you're running under the environment, be sure that the `qBusyCursor` compiler variable is set to 0. Setting the compiler variable this way disables the cursor changing code. When your program is running under the THINK Pascal environment, the debugging code in THINK Pascal doesn't return quickly enough to give the VBL task that manages the cursor changing enough time to run.

---

---

#### Warning

If you try to run your application in the THINK Pascal environment with `qBusyCursor` set to 1, your application will crash.

---

---

The projects in the MacApp Seeds folder have `qBusyCursor` set correctly.

#### Segmentation in MacApp

MacApp makes several assumptions about code resources. That's why you should be sure to follow Apple's recommendations on segmenting MacApp-based programs. Here are some things to watch out for:



- The segment named Main must be the first segment in your project. When you look at your project by segment, this segment should be the one at the top of the list.
- MacApp assumes that code segments are in the application's resource file. When you run in the THINK Pascal environment, your code segments are in the project document, not the application's resource file. The UMemory unit in the THINK MacApp folder is modified to look for CODE resources in the right place.
- Typically, you can't have more than 32K in a segment of your application. While you're running under the THINK Pascal environment, you can have up to 64K in a code segment. When you build your final application, though, no segment can have more than 32K.

### **UObject and instantiation by name**

The UObject unit that THINK Pascal uses is not the same as the UObject unit that comes with the MacApp package. The THINK Pascal UObject unit knows how THINK Pascal implements objects.

One service that UObject provides is object instantiation by name. Sometimes, the smart linker may remove references to a class that you may later want to instantiate by name. To make sure that THINK Pascal doesn't remove references to classes in this case, you can use the member function like this:

```
procedure ForceReferences;  
begin  
    if member(nil, ClassName) then  
        { null clause } ;  
end;
```

## ◆ **A    *MacApp and THINK Pascal***

---

# Index

Entries in **bold face** refer to menu commands. Entries in typewriter face refer to functions, methods, variables, keywords, or files.

## Numerics

16-bit coordinates 79, 80–81  
32-bit coordinates 79, 80–81

## A

**AbortInQueue** 461  
**AboutToPrint**  
    **CDocument** 266  
    **CEditText** 276  
    **CPane** 329  
    **CPanorama** 350  
abstract classes 67  
**AbTx** resource 87  
**Activate** 72  
    **CControl** 234  
    **CDesktop** 244  
    **CDirector** 252  
    **CEditText** 272  
    **CFWDesktop** 290  
    **CScrollBar** 385  
    **CSizeBox** 402  
    **CView** 440  
    **CWindow** 455  
activate events 72  
**ActivateDirector**  
    **CDirector** 252  
    **CDirectorOwner** 256  
**ActivateWind**  
    **CDirector** 253  
**Add**  
    **CCluster** 218  
**AddDependent**  
    **CCollaborator** 228  
**AddDirector**  
    **CDirectorOwner** 256  
**AddLongPt** 463  
**AddMenu**  
    **CBartender** 173  
**AddProvider**  
    **CCollaborator** 228  
**AddSubview**  
    **CView** 445  
**AddWind**  
    **CDesktop** 247  
    **CFWDesktop** 291  
**AdjustBounds**  
    **CEditText** 275  
**AdjustCursor** 87  
    **CAbstractText** 130  
    **CDesktop** 246  
    **CFWDesktop** 291  
    **CView** 442  
**AdjustHoriz**  
    **CPane** 326  
**AdjustScrollMax**  
    **CScrollPane** 390  
**AdjustToEnclosure**  
    **CPane** 326  
**AdjustVert**  
    **CPane** 326  
alert resources 96  
ALRT resources 96  
ancestor, *See* superclass  
**Append**  
    **CList** 298  
**Apple** menu 89, 98  
**AppleEventIdle**  
    **CSwitchboard** 408  
applications  
    commands, handling 75  
    subclass, writing your 74

writing 72–77  
 Art Class Demo folder 12  
 ASSERT macro 95  
 Assert procedure 96  
 AssignIdleChore  
   CApplication 158  
 AssignUrgentChore  
   CApplication 158  
 auto key events 72  
 AutoScroll  
   CPanorama 349

## B

bartender 67, 71  
 base class, *See* superclass  
 BecomeGopher  
   CAbstractText 124  
   CBureaucrat 191  
 BeginDrawing  
   CBitmap 182  
 BeginTracking  
   CMouseTask 311  
 bounds rectangle 84  
 BringBehind 460  
 BringFront  
   CList 298  
 BroadcastChange  
   CBureaucrat 191  
   CCollaborator 227  
 Browser 55–60  
 bureaucrats 70

## C

CAbstractText 117  
   resources, reading from 87  
 CalcAperture  
   CPane 331  
 CalcBorderRect  
   CPaneBorder 340  
 CalcFrame  
   CPane 330  
 CalcTERTects  
   CEditText 275  
 CalcTopFloat  
   CFWDesktop 292  
 Calibrate  
   CScrollPane 390  
 CanBeGopher  
   CView 440  
 CancelDependency  
   CCollaborator 227  
 CancelIdleChore

CApplication 158  
 CancelTyping  
   CTextEditTask 424  
 canFail 93  
 CanStillType  
   CTextEditTask 423  
 CAppleEvent 131  
 CApplication 137  
   *See also* applications  
 CArray 159  
 catch handler 103  
   THINK C 106  
   THINK Pascal 109  
 CATCH macro 106  
 CatchFailure 109, 110  
 CBarOwner 8  
 CBartender 165  
 CBitmap 179  
 CBitmapPane 183  
 CBorder 8  
 CBureaucrat 187  
 CButton 193  
 CCharGrid 197  
 CCheckBox 201  
 CChore 205  
 CClipboard 209  
 CCluster 8, 217  
 CCollaborator 223  
   *See also* collaborators  
 CCollection 229  
 CColorWindow 8  
 CControl 231  
 CDataFile 8, 237  
 CDecorator 241  
 CDesktop 243  
 CDirector 249  
   *See also* directors  
 CDirectorOwner 255  
 CDocument 259  
   *See also* documents  
 CEditText 269  
   resources, reading from 87  
 CenterWindow  
   CDecorator 242  
 CenterWithinEnclosure  
   CPane 327  
 CEnvironment 279  
 CError 281  
 CFile 8, 285  
 CFWDesktop 289  
 CGridSelector 293  
 chain of command 70

---

ChangeName  
     CFile 288  
 ChangeSelection  
     CSelector 395  
 ChangeSize  
     CControl 234  
     CPane 326  
     CScrollPane 390  
     CWindow 456  
 CheckAllocation 283  
 checking menu items 90  
 CheckInsertion  
     CEditText 273  
 CheckMarkCmd  
     CBartender 175  
 CheckOSError  
     CError 282  
 CheckResource 283  
 ChooseFile  
     CApplication 157  
 ChooseItem  
     CMenuDefProc 307  
     CSelectorMDEF 400  
 Class Browser 55–60  
 class variables  
     naming conventions 66  
 classes 20, 27–29  
     abstract 67  
     declaring 27  
     finding declaration 55–56  
     forward references 28  
     naming conventions 66  
     organizing 35  
     root class 28  
     testing for membership in 30  
     type compatibility 29  
     which to define 23  
 Cleanup  
     CDesktop 248  
     CFWDesktop 292  
 clicks 71  
     drawing, and 81  
     receiving in pane 77  
 CList 8, 297  
 Clone  
     CObject 314  
 Close  
     CClipboard 212  
     CDataFile 239  
     CDirector 253  
     CDirectorOwner 257  
     CDocument 264  
     CFile 287  
     CResFile 378  
     CWindow 452  
 ClosePrintMgr  
     CPrinter 366  
 CloseWind  
     CClipboard 212  
     CDirector 253  
     CDocument 264  
     CTearOffMenu 418  
 CMBBarChore 301  
 cmdNull 88  
 CMenuDefProc 303  
 CMouseTask 309  
 CNTL resources 97  
 CObject 313  
 collaborators 71  
 ColorQDIsPresent 462  
 Command keys, handling 72, 88  
 command numbers 88  
     menu resources, in 89  
     negative 90  
 command, chain of 70  
 commands 71  
     application, handling 75  
     document, handling 75  
     menu, handling 88  
     *See also* events  
 Compatibility classes 8  
 Compatibility classes folder 11  
 ConcatPStrings 461  
 ConfirmClose  
     CDocument 264  
 constants  
     naming conventions 66  
 Contains  
     CDesktop 246  
     CPane 324  
     CView 439  
     CWindow 454  
 Control classes folder 11  
 control resources 97  
 control, flow of 71  
 conventions, naming 66  
 ConvertGlobal  
     CClipboard 215  
 converting coordinates 80–81  
 ConvertPrivate  
     CClipboard 215  
 coordinate systems 79  
     panes, and 80–81  
     panoramas, and 84

---

Copy  
     CObject 314  
 CopyFrom  
     CBitMap 182  
 CopyFromTemporary  
     CArray 164  
 CopyPString 462  
 CopyTextRange  
     CAbstractText 126  
 CopyTo  
     CBitMap 182  
 CopyToTemporary  
     CArray 164  
 core classes 67  
 Core classes folder 11  
 CPane 315  
     *See also* panes  
 CPaneBorder 335  
 CPaneMDEF 341  
 CPanorama 345  
     *See also* panoramas  
 CPatternGrid 351  
 CPictFile 355  
 CPicture 357  
     resources, reading from 87  
 CPNTGFile 361  
 CPrinter 363  
 CRadioButton 8  
 CRadioControl 371  
 CRadioGroup 8  
 CRadioGroupPane 373  
 CreateDocument 74  
     CApplication 157  
 CreateNew  
     CFile 287  
     CResFile 378  
 CResFile 377  
 criticalBalance 92  
 CRunArray 379  
 CScrollBar 383  
 CScrollPane 387  
 CScrollPane  
     *See also* scroll panes  
 CSelector 393  
 CSelectorMDEF 399  
 CSizeBox 401  
 CStack 403  
 CStaticText 8  
 CSwitchboard 405  
 CTask 411  
     *See also* tasks  
 CTearChore 415  
 CTearOffMenu 417  
 CTextEditTask 421  
 CTextEnvirons 431  
 CTextStyleTask 427  
 cursor tracking 87  
     *See also* mouse tracking  
 CView 435  
     *See also* views  
 CWindow 449  
**D**  
 data member, *See* instance variable  
 DataSize  
     CClipboard 214  
 Dawdle  
     CBureaucrat 190  
     CEditText 276  
 Deactivate 72  
     CControl 234  
     CDesktop 244  
     CDirector 252  
     CEditText 272  
     CFW/Desktop 290  
     CScrollBar 385  
     CSizeBox 402  
     CView 441  
     CWindow 455  
 DeactivateDirector  
     CDirector 252  
     CDirectorOwner 256  
 DeactivateWind  
     CDirector 253  
 Debug.Settings.R 494  
 DebugExceptions 96  
 debugging 95–96  
     THINK C aids 95  
     THINK Pascal aids 96  
 DeleteAll  
     CRunArray 381  
 DeleteFromBar  
     CBartender 173  
 DeleteItem  
     CArray 162  
 DeleteRange  
     CTextEditTask 424  
 DeleteRun  
     CRunArray 382  
 DeleteValue  
     CRunArray 381  
 dependents 71  
 DependUpon  
     CCollaborator 227

---

derived class, *See* subclass  
 desk accessories, and THINK Class Library 72  
 desktop 69  
 Dialog classes folder 11  
 dialog resources 96  
 dimALL 91  
 dimming menu items 90  
 dimNONE 91  
 dimSOME 91  
 direct commands 67  
 directors 71  
 DisableCmd  
     CBartender 174  
 DisableMenu  
     CBartender 174  
 DisableMenuBar  
     CBartender 174  
 Dispatch  
     CSwitchboard 409  
 DispatchClick 71  
     CDesktop 245  
     CView 441  
     CWindow 457  
 DispatchCursor  
     CDesktop 246  
     CView 442  
     CWindow 457  
 Dispose  
     CAbstractText 120  
     CArray 161  
     CBitMap 181  
     CBitMapPane 185  
     CBureaucrat 188, 358  
     CCharGrid 200  
     CCluster 218  
     CCollaborator 226  
     CControl 232  
     CDesktop 244  
     CDirector 251  
     CDirectorOwner 256  
     CDocument 262  
     CEditText 271  
     CFile 286  
     CFWDesktop 290  
     CObject 314  
     CPane 323  
     CPatternGrid 354  
     CPictFile 356  
     CPNTGFile 362  
     CPrinter 366  
     CSizeBox 402  
     CTextEditTask 423  
     CView 438  
     CWindow 451  
     document, for your 75  
     pane, for your 77  
 dispose procedure 30, 38  
 DisposeAll  
     CCluster 218  
 DisposeItems  
     CCluster 218  
 DITL resources 96  
 Do  
     CTask 413  
     CTextEditTask 423  
     CTextStyleTask 428  
 DoActivate  
     CSwitchboard 407  
 DoAppleEvent  
     CApplication 151  
     CBureaucrat 190  
 DoAutoKey 72  
     CAbstractText 123  
     CApplication 148  
     CBureaucrat 189  
 DoBackspace  
     CTextEditTask 424  
 DoClick 71  
     CControl 235  
     CEditText 271  
     CScrollBar 385  
     CSelector 395  
     CView 441  
     pane, for your 77, 81  
 DoCommand  
     application, for your 75  
     CAbstractText 122  
     CApplication 149  
     CBureaucrat 189  
     CDirector 251  
     CDocument 263  
     document, for your 75  
     menus, for your 88, 90  
 documents 74, 75–76  
     commands, handling 75  
     subclass, writing your 75  
     *See also* files  
 DoDeactivate  
     CSwitchboard 407  
 DoDiskEvent  
     CSwitchboard 406  
 DoDoubleClick  
     CSelector 397

---

DoForEach	DoSaveAs
CCluster 220	CDocument 266
DoForEach1	DoSaveFileAs
CCluster 221	CDocument 267
DoFwdDelete	DoScroll
CTextEditTask 424	CScrollPane 391
DoGoodClick	DoSuspend
CButton 195	CSwitchboard 407
CCheckBox 203	DoTask
CControl 236	CTask 413
CRadioControl 372	CTextEditTask 423
DoHighLevelEvent	CTextStyleTask 428
CSwitchboard 408	DoThumbDrag
DoHorizScroll	CScrollPane 391
CScrollPane 390	DoThumbDragged
DoIdle	CControl 235
CSwitchboard 407	CScrollBar 385
DoKeyDown 72	DoTyping
CAbstractText 123	CTextEditTask 423
CApplication 148	DoUpdate
CBureaucrat 189	CSwitchboard 407
CPanorama 350	DoVertScroll
DoKeyEvent	CScrollPane 391
CSwitchboard 406	Down Arrow key 59
DoKeyUp	Drag
CApplication 149	CWindow 455
CBureaucrat 189	DragWind
DoMouseDown	CDesktop 247
CSwitchboard 406	CFWDesktop 292
DoMouseUp	Draw 80
CDesktop 246	CBitMapPane 185
CSwitchboard 406	CControl 234
CView 441	CEditText 271
DonePrinting	CGridSelector 294
CDocument 266	CPane 328
CEditText 276	CPicture 359
CPane 329	CScrollBar 385
CPanorama 350	CSizeBox 402
DoNormalChar	DrawAll
CTextEditTask 424	CControl 234
DoOtherEvent	CPane 328
CSwitchboard 407	DrawBorder
DoPageSetup	CPaneBorder 339
CPrinter 368	DrawGrid
DoPrint	CGridSelector 295
CPrinter 368	drawing, in pane 80
DoResume	DrawItem
CSwitchboard 407	CCharGrid 200
DoRevert 76	CGridSelector 295
CDocument 266	CPatternGrid 354
DoSave 76	DrawMenu
CDocument 266	CMenuDefProc 307



CPaneMDEF 342  
DrawSICN 459

## E

**Edit** menu 89, 98  
EmptyGlobalScrap  
    CClipboard 215  
EmptyLongRect 465  
EmptyScrap  
    CClipboard 216  
EnableCmd  
    CBartender 174  
EnableMenu  
    CBartender 174  
EnableMenuBar  
    CBartender 174  
encapsulation 19  
enclosures 69, 77  
EnclosureScrolled  
    CPane 327  
EnclToFrame  
    CPane 332  
EnclToFrameR  
    CPane 332  
EndDrawing  
    CBitMap 182  
EndTracking 94  
    CMouseTask 311  
ENDTRY macro 106  
Enter key 59  
EqualLongPt 463  
EqualLongRect 465  
EqualMem 468  
error handling, *See* exception handling  
error messages  
    displaying your own 106, 115  
    exception handling, and 105  
    strings, for THINK Class Library 97  
error parameter 110  
ErrorAlert 466  
Estr resources 97  
events  
    passing to objects 71  
    *See also* commands  
exception handling 103–116  
    error messages, displaying 105, 115  
    Exit, and 113  
    memory allocation 105  
    objects, creating 105, 107, 111  
    propagating errors 109, 113  
    retrying try handler 109, 113  
    returning values 108

THINK C 106–109  
THINK Pascal 109–113

Exceptions.h 106  
ExistsOnDisk  
    CFile 287  
Exit  
    Application 156  
Exit, and exception handling 113  
ExitApp  
    Application 156  
ExtractFromDescList  
    CAppleEvent 134

## F

FailInfo 110  
FailMemError 115  
FailNIL 105, 115  
FailNILRes 115  
FailOSErr 115  
FailResError 115  
Failure 114  
fi variable 110  
File classes folder 11  
File menu 89, 98  
files  
    documents, and 71  
    *See also* documents  
FindCmdNumber 88, 90  
    CBartender 175  
FindDlgPosition 460  
FindIndex  
    CList 299  
FindItem  
    CCluster 219  
    CGridSelector 295  
    CSelector 396  
FindItem1  
    CCluster 219  
FindItemBox  
    CGridSelector 296  
FindItemText  
    CBartender 176  
FindLine  
    CAbstractText 130  
    CEditText 276  
FindMenuIndex  
    CBartender 176  
FindMenuItem  
    CBartender 176  
FindRun  
    CRunArray 382  
FindSubview

- CView 446
- FindSum
  - CRunArray 381
- FindViewByID
  - CDirector 251
  - CView 446
- FirstItem
  - CList 299
- FirstSuccess
  - CList 299
- FirstSuccess1
  - CList 299
- FitToEnclFrame
  - CPane 327
- FitToEnclosure
  - CPane 327
- flags
  - naming conventions 66
- flow of control 71
- Font menu 89, 98
- ForceClassReferences
  - CApplication 147
- ForceNextPrepare
  - CView 447
- ForgetHandle 468
- ForgetObject 468
- ForgetPtr 468
- ForgetResource 468
- frame coordinates 79
- frames 80
- FrameToBounds
  - CPicture 360
- FrameToEncl
  - CPane 332
- FrameToEnclR
  - CPane 332
- FrameToGlobalR
  - CPane 333
  - CView 447
  - CWindow 457
- FrameToQD 80
  - CPane 333
- FrameToQDR 80
  - CPane 333
- FrameToWind
  - CPane 332
- FrameToWindR
  - CPane 332
- Free
  - CAbstractText 120
  - CArray 161
  - CBitMap 181
  - CBitMapPane 185
  - CBureaucrat 188, 358
  - CCharGrid 200
  - CCluster 218
  - CCollaborator 226
  - CControl 232
  - CDesktop 244
  - CDirector 251
  - CDirectorOwner 256
  - CDocument 262
  - CEditText 271
  - CFile 286
  - CFWDesktop 290
  - CObject 314
  - CPane 323
  - CPatternGrid 354
  - CPictFile 356
  - CPNTGFile 362
  - CPrinter 366
  - CSizeBox 402
  - CTextEditTask 423
  - CView 438
  - CWindow 451
    - document, for your 75
    - pane, for your 77
- function member, *See* method
- FW/Tearoffs folder 11

## G

- gApplication 473
- gAskFailure 95
- gBartender 473
- gBreakFailure 95
- gClicks 475
- gClipboard 474
- gDecorator 474
- gDesktop 473
- GenericMDEF 308
- gError 474
- Get1Height
  - CAbstractText 128
- GetAEEEvent
  - CAppleEvent 133
- GetAERefCon
  - CAppleEvent 134
- GetAEReply
  - CAppleEvent 134
- GetAlignCmd
  - CAbstractText 128
  - CEditText 274
- GetAnEvent
  - CSwitchboard 409

---

GetAperture	GetFontNumber 461
CDesktop 248	GetFrame
CPane 324	CPane 323
CView 439	CView 439
CWindow 453	CWindow 452
GetBalloonInfo	GetFramePosition
CView 445	CPanorama 347
GetBitMap	GetFrameSpan
CBitMapPane 185	CPanorama 347
GetBorder	GetFSSpec
CPane 325	CFile 287
GetBorderFlags	GetGlobalScrap
CPaneBorder 338	CClipboard 213
GetBounds	GetHeight
CBitMap 181	CAbstractText 128
CDesktop 248	CEditText 274
CPanorama 348	GetHelpResID
GetBoundsOrigin	CPane 325
CBitMap 181	CView 445
GetCharAfter	CWindow 454
CAbstractText 126	GetHomePosition
GetCharBefore	CPanorama 349
CAbstractText 126	GetID
GetCharOffset	CView 440
CAbstractText 128	GetInterior
CEditText 274	CScrollPane 390
GetCharPoint	CView 439
CAbstractText 129	CWindow 452
CEditText 275	GetItem
GetCharStyle	CArray 162
CAbstractText 129	GetLength
CEditText 275	CAbstractText 130
GetClassName	CDataFile 238
CObject 314	CEditText 276
GetClickCmd	CPane 324
CButton 195	GetMacPicture
GetCmdText	CPicture 359
CBartender 175	GetMacPort
GetCommandBase	CView 439
CSelector 396	GetMacWindow
GetData	CTearOffMenu 419
CClipboard 215	GetMargin
GetDescList	CPaneBorder 339
CAppleEvent 134	GetMargins
GetErrorResult	CTearOffMenu 419
CAppleEvent 135	GetMark
GetEventClass	CDataFile 239
CAppleEvent 133	GetMaxValue
GetEventID	CControl 232
CAppleEvent 133	GetMinValue
GetExtent	CControl 232
CPanorama 347	GetName

---

CDocument 267	GetSteps
CFile 287	CScrollPane 390
GetNameIndex	GetStripCount
CTask 412	CPrinter 367
GetNumItems	GetSupervisor
CCollection 229	CBureaucrat 188
CRunArray 380	GetTEFontInfo
GetNumLines	CEditText 274
CAbstractText 130	GetTextHandle
CEditText 276	CAbstractText 125
GetOrigin	CEditText 272
CPane 324	GetTextInfo
CView 439	CTextEnvirons 433
GetPageArea	GetTextRange
CPrinter 368	CEditText 273
GetPageInfo	GetTextStyle
CPrinter 368	CAbstractText 129
GetPageStart	CEditText 275
CPrinter 368	GetTitle
GetPattern	CControl 233
CPaneBorder 338	CWindow 453
CPatternGrid 354	GetTopWindow
GetPenSize	CDesktop 248
CPaneBorder 338	GetValue
GetPhase	CControl 232
CApplication 148	CRunArray 381
GetPixelExtent	GetWantsClicks
CPane 324	CView 439
CPanorama 349	GetWCount
GetPosition	CDecorator 242
CPanorama 348	GetWholeLines
GetPrintRecord	CAbstractText 129
CPrinter 368	GetWindow
GetRounding	CDirector 251
CPaneBorder 339	CPane 325
GetScaled	GetXferMode
CPicture 359	CBitMap 181
GetScales	gGopher 474
CPanorama 348	gIBeamCursor 475
GetSelection	gInBackground 477
CAbstractText 125	gLastError 477
CEditText 272	gLastMessage 477
CSelector 396	gLastMouseDown 474
GetShadow	gLastMouseUp 474
CPaneBorder 339	gLastViewHit 474
GetSpacingCmd	global coordinates 79
CAbstractText 128	global variables
CEditText 274	naming conventions 66
GetSpecification	gopher 70, 71
CAbstractText 121	GotRequiredParams
GetStationID	CAppleEvent 134
CRadioGroupPane 375	grow icon 98

GrowMemory 92  
     CApplication 153  
 GrowZoneFunc 282  
 gSignature 477  
 gSleepTime 476  
 gSystem 475  
 gUtilRgn 477  
 gWatchCursor 475

## H

HandleFailure 110  
 HasResFork  
     CResFile 378  
 HavePagination  
     CPrinter 366  
 Hide  
     CControl 233  
     CDesktop 244  
     CFWDesktop 290  
     CPane 326  
     CView 440  
     CWindow 454  
 HideSuspend  
     CWindow 455  
 HideWind  
     CDesktop 247  
     CFWDesktop 291  
 HiliteItem  
     CGridSelector 295  
     CPatternGrid 354  
     CSelector 396  
 HitSamePart  
     CDesktop 247  
     CSelector 395  
     CView 441  
 horizontal sizing characteristics, of pane  
     82

## I

IAbstractText  
     CAbstractText 119  
 IAppleEvent  
     CAppleEvent 133  
 IApplication 74, 92  
     CApplication 142  
 IArray  
     CArray 161  
 IBartender  
     CBartender 173  
 IBitMap  
     CBitMap 181  
 IBitMapPane

    CBitMapPane 184  
 IBureaucrat  
     CBureaucrat 188  
 IButton  
     CButton 194  
 ICharGrid  
     CCharGrid 199  
 ICheckBox  
     CCheckBox 202  
 IClipboard  
     CClipboard 211  
 ICluster  
     CCluster 218  
 ICollaborator  
     CCollaborator 226  
 ICollection  
     CCollection 229  
 icon, small, resource 98  
 IDataFile  
     CDataFile 238  
 IDecorator  
     CDecorator 242  
 IDesktop  
     CDesktop 244  
 IDirector  
     CDirector 250  
 IDirectorOwner  
     CDirectorOwner 256  
 Idle  
     CApplication 156  
 IDocument 75  
     CDocument 262  
 IEditText  
     CEditText 270  
 IFile  
     CFile 286  
 IFWDesktop  
     CFWDesktop 290  
 IGridSelector  
     CGridSelector 294  
 IList  
     CList 297  
 IMenuDefProc  
     CMenuDefProc 307  
 IMouseTask  
     CMouseTask 311  
 Includes  
     CCluster 219  
 inCriticalSection 92  
 INewButton  
     CButton 194  
 INewCheckBox

---

CCheckBox 202  
 INewRadioControl  
     CRadioControl 372  
 INewWindow  
     CWindow 451  
 inherited keyword 32  
 InitMemory  
     CApplication 143  
 InitToolbox  
     CApplication 143  
 InsertAfter  
     CList 298  
 InsertAt  
     CList 298  
 InsertAtIndex  
     CArray 162  
 InsertHierMenu  
     CBartender 174  
 InsertInBar  
     CBartender 173  
 InsertMenuCmd  
     CBartender 175  
 InsertRun  
     CRunArray 382  
 InsertTextHandle  
     CAbstractText 126  
 InsertTextPtr  
     CAbstractText 126  
     CEditText 273  
 InsertValue  
     CRunArray 380  
 InsetLongRect 464  
 InspectSystem  
     CApplication 146  
 InstallPanorama 86  
     CScrollPane 389  
 InstallPatches  
     CApplication 147  
 instance 20  
 instance variables 20  
     accessing 30  
     declaring 28  
     naming conventions 66  
     referring to 30  
     taking address of 36  
     using in Toolbox calls 36  
 instances  
     *See also* objects  
 instantiation by name 497  
 instantiation, *See* creating  
 integer type 66  
 IPane 76  
     CPane 321  
 IPaneBorder  
     CPaneBorder 338  
 IPaneMDEF  
     CPaneMDEF 342  
 IPanorama  
     CPanorama 346  
 IPatternGrid  
     CPatternGrid 353  
 IPictFile  
     CPictFile 355  
 IPicture  
     CPicture 358  
 IPNTGFile  
     CPNTGFile 361  
 IPrinter  
     CPrinter 365  
 IRadioControl  
     CRadioControl 372  
 IRadioGroupPane  
     CRadioGroupPane 374  
 IResFile  
     CResFile 377  
 IResPaneBorder  
     CPaneBorder 338  
 IRunArray  
     CRunArray 380  
 IsActive  
     CDirector 254  
     CView 438  
 IsCancelEvent 461  
 IsChecked  
     CCheckBox 203  
 IsColor  
     CWindow 453  
 IScrollBar  
     CScrollBar 384  
 IScrollPane  
     CScrollPane 388  
 IsDialogWindow 460  
 ISelector  
     CSelector 394  
 ISelectorMDEF  
     CSelectorMDEF 399  
 IsEmpty  
     CCollection 230  
 IsFloating  
     CWindow 453  
 ISizeBox  
     CSizeBox 401  
 IsModal  
     CWindow 453

- IsMyWindow 460
- IsOpen
  - CResFile 378
- IsSystemWindow 460
- IStack
  - CStack 403
- IsUndone
  - CTask 412
- IsVisible
  - CView 438
- ISwitchboard
  - CSwitchboard 406
- ITask
  - CTask 412
- ITearChore
  - CTearChore 415
- ITearOffMenu
  - CTearOffMenu 418
- item number 90
- ItemOffset
  - CArray 164
- ITextEditTask
  - CTextEditTask 422
  - CTextStyleTask 428
- ITextEnvirons
  - CTextEnvirons 433
- IView
  - CView 437
- IViewRes 87
  - CAbstractText 120
  - CEditText 270
  - CPane 323
  - CPanorama 347
  - CPicture 358
  - CScrollPane 389
  - CView 438
- IViewTemp
  - CPane 323
  - CPanorama 347
  - CPicture 358
  - CScrollPane 389
  - CView 438
- IWindow
  - CWindow 450

## J

- JumpToEventLoop
  - CApplication 157

## K

- KeepTracking 94
  - CMouseDownTask 311

- key down events 72
- KeyIsDown 461
- keywords 37
- kSilentError 106

## L

- LastItem
  - CList 299
- LastSuccess
  - CList 300
- LastSuccess1
  - CList 300
- LClipRect 471
- LCopyBits 471
- Left Arrow key 59
- LEraseOval 469
- LEraseRect 469
- LEraseRoundRect 470
- LFillOval 470
- LFillRect 469
- LFillRoundRect 471
- LFrameOval 470
- LFrameRect 469
- LFrameRoundRect 470
- LIInvertOval 470
- LIInvertRect 469
- LIInvertRoundRect 470
- LLineTo 471
- LMoveTo 471
- local coordinates 79
- location, of pane 77
- Lock
  - CObject 314
- long coordinates 79, 80–81
- long int type 66
- longint type 66
- LongPt 80
- LongRect 80
- LongToQDPt 463
- LongToQDRect 464
- LPaintOval 470
- LPaintRect 469
- LPaintRoundRect 470
- LRectRgn 471

## M

- MacApp
  - busy cursor 496
  - compiler variables 492
  - converting 486
  - environment differences 496
  - minimum requirements 484

- resource files 494
- seed projects 490
- segmentation 496
- Toolbox initialization 496
- MacApp Seeds folder 490
- MacApp.π 490
- MacApp.Build.π 490
- MacApp.Build.Debug.π 490
- MacApp.Debug.π 490
- MacApp.Debug.Rsrc 495
- MacApp.rsrc 495
- MacApp.Script 486
- MakeBartender
  - CAApplication 146
- MakeClipboard
  - CAApplication 144
- MakeClipView
  - CClipboard 216
- MakeCurrent
  - CResFile 378
- MakeDecorator
  - CAApplication 145
- MakeDesktop
  - CAApplication 144
- MakeEditTask
  - CAbstractText 124
- MakeError
  - CAApplication 144
- MakeMacTE
  - CEditText 271
- MakeMacWindow
  - CWindow 452
- MakePrinter
  - CDocument 265
- MakeStyleTask
  - CAbstractText 124
- MakeSwitchboard
  - CAApplication 144
- MatchView
  - CView 446
- MBAR resource 98
- member 20
- member function 30, 38
- member function, *See* method
- members
  - See also* instance variables, methods
- memory
  - allocation and exception handling 105, 115
  - memory allocating document, for your 75
  - memory, allocating 92
  - pane, for 77
- MemoryReplenished
  - CAApplication 154
- MemoryShortage 92
  - CAApplication 153
- menu bar resource 98
- menu ID 90
- MENU resources 89, 97
- MENUapple 89
- MENUedit 89
- MENUfile 89
- MENUfont 89
- menus 88–92
  - adding items 90
  - checking items 90
  - clicking in 90
  - creating in code 90
  - dimming items 90
  - resources, reading from 89
- MenuSelect 91
- MENUsize 89
- message parameter 110
- messages 19
  - sending 32
  - See also* methods
- methods 20, 31–32
  - calling 32
  - calling inherited 32
  - declaring 28
  - defining 31
  - finding definition 57–59
  - monomorphic 32
  - overriding 28
  - referring to current object 31
- «%\_MethTables» 95
- «%\_MethTables» 35
- MissingResources
  - CError 282
- monomorphic methods 32
- More classes folder 11
- MoreSlots
  - CArray 164
- mouse clicks, *See* clicks
- mouse tracking 94
  - See also* cursor tracking
- Move
  - CWindow 456
- MoveDown
  - CList 298
- MoveItemToIndex
  - CArray 163
- MoveOffScreen



CWindow 456  
MoveToCorner  
    CTearOffMenu 419  
MoveToIndex  
    CList 299  
MoveUp  
    CList 298  
MultiFinder, running under 72  
**N**  
naming conventions 66  
new operator  
    exception handling, and 105  
new procedure 30, 37  
**New**, handling 74, 76  
NewClassDemo Folder 12  
NewFile 76  
    CDocument 264  
NewHandleCanFail 466  
NIL 66  
NO\_PROPAGATE macro 109  
Notify 93, 94  
    CApplication 148  
    CBureaucrat 189  
    CDocument 262  
NthItem  
    CList 299  
NULL 66  
**O**  
object reference 29  
objects 19, 29–30  
    as handles 29, 35  
    creating 30  
    creating and exception handling  
        105, 107, 111  
    defining 29  
    deleting 30  
    in desk accessories and code re-  
        sources 27  
    segmentation 35  
    self 31  
    size of 37  
    type compatibility 29  
OCluster 8  
ODataFile 8  
Offset  
    CCluster 219  
    CControl 234  
    CPane 326  
OffsetLongRect 464  
OFile 8

OList 8  
Open  
    CDataFile 239  
    CFile 287  
    CResFile 378  
**Open...**, handling 74, 76  
OpenDocument 74  
    Application 157  
OpenFile 76  
    CDocument 265  
OpenPrintMgr  
    CPrinter 366  
origin 84  
OutOfMemory  
    Application 154  
override keyword 28  
overriding methods 21  
OwnsWindow  
    CDirector 251  
**P**  
PackageAppleEvent  
    CApplication 150  
PageCount  
    CDocument 265  
PageNumToStrips  
    CPrinter 368  
Paginate  
    CAbstractText 125  
    CDocument 265  
    CPane 329  
    CPanorama 350  
Pane resource 87  
panes 77–87  
    clicks, receiving 77  
    coordinates 79, 80–81  
    drawing 80  
    initializing 76  
    location, setting 77, 81–83  
    memory, allocating 77  
    moving 81  
    resizing 81  
    resources, reading from 87  
    scrolling 83–86  
    subclass, writing your 76  
    supervisor, and 77  
    windows, and 78  
    *See also* windows  
Pano resource 87  
panoramas 83–86  
    resources, reading from 87  
parameters

- naming conventions 66
- PctP resource 87
- Perform
  - CChore 206
  - CBarChore 301
  - CTearChore 415
- PerformEditCommand
  - CAbstractText 123
  - CEditText 271
- PickFileName
  - CDocument 267
- PinInRect 460
- PixelIsBlack
  - CBitMap 181
- Place
  - CPane 326
- PlaceNewWindow
  - CDecorator 242
- PlacePopUp
  - CMenuDefProc 307
- polymorphism 22
- Pop
  - CStack 403
- position 84
- PositionDialog 461
- PostAlert
  - CError 282
- Preload
  - CApplication 155
- Prepare 79
  - CControl 234
  - CDesktop 248
  - CPane 330
  - CView 447
  - CWindow 456
- PrepareToPrint
  - CPane 330
- Prepend
  - CList 298
- PrintPage
  - CEditText 276
  - CPane 329
  - CPanorama 350
- PrintPageOfDoc
  - CDocument 266
- PrintPageRange
  - CPrinter 369
- PrivateChanged
  - CClipboard 216
- ProcessEvent
  - CApplication 155
- ProcessEvent

- CSwitchboard 408
- ProviderChanged
  - CBureaucrat 191
  - CCollaborator 227
  - CDirector 254
  - CRadioGroupPane 375

- providers 71
- Pt2LongRect 465
- PtInLongRect 465
- PtInQDSpace 464
- PtInTearRgn
  - CPaneMDEF 344
- Push
  - CStack 403
- PushTryHandler 114
- PutData
  - CClipboard 214
- PutGlobalScrap
  - CClipboard 213

## Q

- qBusyCursor 496
- QDToFrame
  - CPane 333
- QDToFrameR
  - CPane 333
- QDToLongPt 463
- QDToLongRect 464
- QuickDraw coordinates 79, 80–81
- Quit
  - CApplication 156
  - CDirectorOwner 257

**Quit**, handling 75

## R

- rainy day fund 92
- raising exceptions, *See* exception handling
- ReadAll
  - CDataFile 239
  - CPictFile 356
- ReadNewBitMap
  - CPNTGFile 362
- ReadSome
  - CDataFile 240
- ReallyVisible
  - CDesktop 245
  - CPane 324
  - CView 438
- RectInQDSpace 465
- Redo
  - CTask 413

---

CTextEditTask 423  
 Refresh 80  
     CPane 328  
 RefreshBorder  
     CPane 329  
 RefreshLongRect  
     CPane 329  
 RefreshRect  
     CPane 328  
 Remove  
     CCluster 219  
 RemoveDependent  
     CCollaborator 228  
 RemoveDirector  
     CDirectorOwner 256  
 RemoveMenu  
     CBartender 173  
 RemoveMenuCmd  
     CBartender 175  
 RemovePatches  
     CApplication 147  
 RemoveProvider  
     CCollaborator 228  
 RemoveSubview  
     CView 446  
 RemoveWind  
     CDesktop 247  
     CFWDesktop 291  
 RequestInteraction  
     CAppleEvent 134  
 RequestMemory  
     CApplication 151  
 ResEdit 13  
 ResetPagination  
     CPrinter 366  
 Resize  
     CArray 164  
     CWindow 456  
 ResizeFrame  
     CAbstractText 130  
     CEditText 275  
     CPane 330  
     CPanorama 349  
     CPicture 359  
 ResizeHandleCanFail 466  
 ResolveFileAlias  
     CFile 287  
 resource files  
     for MacApp 494  
     menus, reading from 89  
     THINK Class Library, for 72, 96–99  
     views, reading from 87  
 Restore  
     CEnvironment 279  
     CTextEnvirons 433  
 RestoreEnvironment  
     CPane 330  
 RestoreQuickDraw  
     CPaneMDEF 344  
 RestoreRange  
     CTextEditTask 425  
 RestoreStyle  
     CTextStyleTask 429  
 Resume  
     CApplication 155  
     CClipboard 212  
     CDirector 253  
     CDirectorOwner 257  
     CTearOffMenu 418  
 resume events 72  
 ResumeAfterError 96  
 Retrieve  
     CArray 163  
 RETRY macro 109  
 RetryException 114  
 retrying try handler 109, 113  
 Return key 59  
 returning values and exception handling  
     108  
**Revert**, handling 76  
 Right Arrow key 59  
 root class 21, 28  
 .rsrc file, *See* resource file  
 Run  
     CApplication 154  
**S**  
     •Samples From MacApp 493  
**Save**, handling 76  
 SaveRange  
     CTextEditTask 424  
 SaveStyle  
     CTextStyleTask 429  
 SBarActionProc 391  
 SBarThumbFunc 391  
 scale 84  
 ScPn resource 87  
 ScrapConverted  
     CClipboard 214  
 Scroll  
     CEditText 272  
     CPanorama 349  
 scroll panes 86  
 scrolling 83–86

---

ScrollTo	SetCanBeGopher
CPanorama 349	CView 440
ScrollToSelection	SetClickCmd
CAbstractText 124	CButton 195
CPanorama 349	SetCmdText
Search	CBartender 175
CArray 163	SetCommandBase
SectAperture	CSelector 396
CPane 333	SetCriticalOperation 93, 467
SectLongRect 464	CApplication 152
seed projects 490	SetCursor 87
segmentation 35, 94	SetDefault
MacApp 496	CButton 195
Select	SetDimOption 91
CWindow 455	CBartender 176
SelectAll	SetErrorResult
CAbstractText 125	CAppleEvent 135
SelectionChanged	SetFailInfo 116
CAbstractText 124	SetFontName
CTextEditTask 424	CAbstractText 127
SelectWind	SetFontNumber
CDesktop 247	CAbstractText 127
CFWDesktop 291	CEditText 273
self (current object) 31	SetFontSize
«%_SelProcs» 95	CAbstractText 127
«%_SelProcs» 35	CEditText 273
SendBack	SetFontStyle
CList 298	CAbstractText 127
SetActClick	CEditText 273
CWindow 453	SetFrameOrigin
SetActionProc	CPane 323
CControl 233	SetGridOn
SetAlignCmd	CGridSelector 296
CAbstractText 127	SetHelpResID
CEditText 274	CWindow 454
SetAlignment	SetHorizPageBreak
CEditText 274	CPrinter 367
SetAllocation 93, 466	SetID
SetAllStripHeights	CView 440
CPrinter 367	SetItem
SetAllStripWidths	CArray 162
CPrinter 367	SetLength
SetBitMap	CDataFile 238
CBitMapPane 185	SetLockChanges
SetBlockSize	CArray 161
CArray 161	SetLongPt 463
SetBorder	SetLongRect 464
CPane 325	SetMacPicture
SetBorderFlags	CPicture 359
CPaneBorder 338	SetMargin
SetBounds	CPaneBorder 339
CPanorama 347	SetMargins

---

CTearOffMenu 419  
 SetMark  
     CDataFile 238  
 SetMaxValue  
     CControl 232  
 SetMinimumStack 468  
 SetMinValue  
     CControl 232  
 SetModal  
     CWindow 453  
 SetOverlaps  
     CScrollPane 390  
 SetPattern  
     CPaneBorder 338  
 SetPenSize  
     CPaneBorder 338  
 SetPosition  
     CPanorama 348  
 SetPrintClip  
     CPane 325  
 SetPrintDir  
     CPrinter 366  
 SetResBorder  
     CPane 325  
 SetRounding  
     CPaneBorder 339  
 SetScaled  
     CPicture 359  
 SetScales  
     CPanorama 348  
 SetScrollPane  
     CPanorama 348  
 SetSelection  
     CAbstractText 124  
     CEditText 272  
 SetShadow  
     CPaneBorder 339  
 SetSizeRect  
     CWindow 454  
 SetSpacingCmd  
     CAbstractText 128  
     CEditText 274  
 SetStationID  
     CRadioGroupPane 375  
 SetStdState  
     CWindow 454  
 SetSteps  
     CScrollPane 389  
 SetStripHeight  
     CPrinter 367  
 SetStrips  
     CPrinter 366  
  
 SetStripWidth  
     CPrinter 367  
 SetTextHandle  
     CAbstractText 125  
 SetTextInfo  
     CTextEnvirons 433  
 SetTextMode  
     CAbstractText 127  
     CEditText 273  
 SetTextPtr  
     CAbstractText 125  
     CEditText 272  
 SetTextString  
     CAbstractText 125  
 SetThumbFunc  
     CScrollBar 384  
 Settings.R 494  
 SetTitle  
     CControl 233  
     CWindow 453  
 SetUnchecking 91  
     CBartender 177  
 SetUpFileParameters 74  
     Application 145  
 SetUpMenus 89  
     Application 146  
 SetupQuickDraw  
     CPaneMDEF 344  
 SetValue  
     CControl 232  
     CRunArray 381  
 SetVertPageBreak  
     CPrinter 367  
 SetWantsClicks  
     CView 439  
 SetWholeLines  
     CAbstractText 129  
 SetXferMode  
     CBitMap 181  
 SevereMacError  
     CError 282  
 SFSpecify  
     CFile 286  
 short coordinates 79, 80–81  
 short int type 66  
 Show  
     CControl 234  
     CDesktop 244  
     CFWDesktop 290  
     CPane 325  
     CView 440  
     CWindow 454

- ShowOrHide
    - CWindow 455
  - ShowResume
    - CWindow 455
  - ShowWind
    - CDesktop 247
    - CFWDesktop 291
  - SICN resource 98
  - SimulateClick
    - CButton 195
  - 16-bit coordinates 79, 80–81
  - Size menu 89, 98
  - sizELASTIC 82
  - SizeMenu
    - CMenuDefProc 307
    - CPaneMDEF 343
  - sizFIXEDBOTTOM 82
  - sizFIXEDLEFT 82
  - sizFIXEDRIGHT 82
  - sizFIXEDSTICKY 82
  - sizFIXEDTOP 82
  - sizing characteristics, of pane 81–83
  - small icon resource 98
  - Specify
    - CAbstractText 120
    - CFile 286
  - SpecifyFSSpec
    - CFile 286
  - SpecifyHFS
    - CFile 286
  - SpecifyMsg 115
  - StaggerWindow
    - CDecorator 242
  - Starter Folder 12
  - Starter project
    - building 12
    - writing program with 72–77
  - StartupAction
    - CApplication 155
  - static data member, *See* class variable
  - static member function, *See* class method
  - Status
    - CClipboard 214
  - Store
    - CArray 162
  - StoreToClip
    - CTextEditTask 425
  - STR resources 98
  - STR# resources 99
  - string resources 98
  - strings, error messages 97
  - subclass 21
    - when to create 23
    - See also* class
  - SubclassResponsibility
    - CObject 314
  - SubLongPt 463
  - SubpaneLocation
    - CView 446
  - Success 110, 113
  - SumRange
    - CRunArray 381
  - superclass 21
  - supervisors 70
    - panes, and 77
  - Suspend 72
    - CApplication 155
    - CClipboard 212
    - CDirector 252
    - CDirectorOwner 256
    - CTearOffMenu 418
  - suspend events 72
  - Swap
    - CArray 163
  - switchboard 67, 71
  - SwitchFromDA
    - CApplication 156
  - SwitchToDA
    - CApplication 156
- ## T
- Tab key 59
  - Table classes folder 11
  - tasks 93
    - TCL 1.1 doc folder 12
    - TCL 1.1 Pascal Demos folder 12
    - TCL 1.1 Pascal Demos.sea 12
    - \_\_TCL\_DEBUG\_\_ 95
  - TCL Libraries folder 11
  - TCL Resources 72
  - TCL Resources folder 12
  - TCL TEMPLs 87
    - installing 13
  - TCL TEMPLs folder 12
  - TCL\_DEBUG 96
  - TCLRuntime.lib
    - exception handling, and 105
  - TearOffMenu
    - CPaneMDEF 343
  - TempMemCallsAvailable 462
  - Text classes folder 11
  - THINK Class Library 65–100
    - chain of command 70
    - class diagram 67

- distributing 100
- flow of control 71
- modifying 99
- naming conventions 66
- resources 96–99
- seed projects 12
- visual hierarchy 69
- THINK Class Library 1.1 folder 11
- THINK Class Library 1.1.sea 11
- 32-bit coordinates 79, 80–81
- ThrowOut
  - CFile 288
- TinyEdit Folder folder 12
- TMPL resources 13
- TObject 33–35
- Toggle
  - CClipboard 212
- Toolbox routines
  - exception handling, and 115
  - pane coordinates for 79
- toolboxBalance 92
- TornOff
  - CTearOffMenu 419
- tracking, cursor 87
- tracking, mouse 94
- TrackMouse
  - CView 331
- TrapAvailable 462
- try handler 103
  - retrying 109, 113
  - THINK C 106
  - THINK Pascal 109
- TRY macro 106
- Type
  - CEditText 271
- type
  - C and Pascal 65
- TypeChar
  - CAbstractText 123
- typing, handling 72

## U

- UMemory 497
- Undo 93, 94
  - CTask 413
  - CTextEditTask 423
  - CTextStyleTask 428
- undo 93
- UnionLongRect 465
- units, panorama 84
- UObject 497
- Up Arrow key 59

- Update 72
  - CResFile 378
  - CWindow 456
- update events 72, 80
- UpdateAllMenus
  - CBartender 177
- UpdateDisplay
  - CClipboard 213
- UpdateMenuBar
  - CBartender 177
- UpdateMenus 90
  - CAbstractText 122
  - CApplication 150
  - CBureaucrat 190
  - CDirector 252
  - CDocument 263
- UpdateUndo
  - CDocument 267
- UpdateWindows
  - CDesktop 247
- UseLongCoordinates
  - CView 440
- UsePICT
  - CPicture 359

## V

- variables
  - naming conventions 66
- vertical sizing characters, of pane 82
- View resource 87
- views 69
  - resources, reading from 87
- visual hierarchy 69
- visual messages 67

## W

- WantsActClick
  - CWindow 453
- WIND resources 99
- window coordinates 79
- window resources 99
- windows
  - documents, and 71
  - events, handling 71, 72
  - panes, and 78
  - See also* panes
- WindToFrame
  - CPane 331
- WindToFrameR
  - CPane 332
- WNEIsImplemented 462
- WriteAll



---

    CDataFile 240  
    CPictFile 356  
WriteBitMap  
    CPNTGFile 362  
WriteSome  
    CDataFile 240

## **Z**

Zoom  
    CWindow 456



# A New OOP Book Especially for THINK Pascal™ Programmers!

## Object-Oriented Programming Power for THINK Pascal™ Programmers

*Code Disk Included!*

If you program the Apple® Macintosh® in Pascal and you're intrigued by object-oriented programming and application framework programming, this book is for you. It delivers what you need to know—in your native language. Your knowledge of Pascal is all the preparation you need. The book features:

**A Thorough Introduction to OOP.** Learn OOP from the ground up, revisiting the central concepts at higher and higher levels.

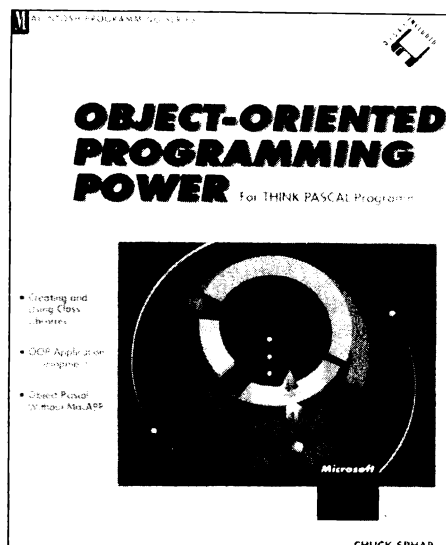
**Clear Instruction on Application Frameworks.** Discover the fundamentals of using application frameworks such as Symantec's THINK Class Library (TCL) and Apple's MacApp®. Look under the hood of the book's own tiny framework, PicoApp.

**Writing Fast, Reliable, Reusable Software Components.** Explore the author's original object innovations as he develops a robust, reusable list data structure.



**Microsoft Press**

**One Microsoft Way, Redmond, WA 98052-6399**



**Dozens of Projects and Examples.** Throughout, the author tests your progress with projects that build on the book's examples. The accompanying code disk contains full versions of all the book's examples (written in THINK Pascal but usable with both MPW® and TML Pascal).

**Package contains:** full-length book and one 800-KB code disk

**System requirements:** Apple Macintosh with 1-MB RAM and hard disk; THINK Pascal 2.0, MPW Pascal 2.0, TML Pascal 2.0, or later versions

**Only \$39.95 (\$54.95 in Canada)**

**To place your credit card order, call**

**1-800-MSPRESS**

(8AM to 5PM Central Time)

**Refer to campaign BTP.**

U.S. orders only. To order in Canada, call Macmillan Canada, (416) 293-8141

THINK Pascal is a trademark of Symantec Corp. Microsoft Press and not Symantec Corporation is responsible for fulfilling this order. Symantec makes no representation or warranty as to the quality or suitability of the products offered hereby.





**SYMANTEC.**<sup>TM</sup>

Symantec Corporation  
10201 Torre Avenue  
Cupertino, CA 95014-2132  
408/253-9600