Programming In Apple Dylan

Preliminary

Developer Press
Apple Computer, Inc.

Apple Computer, Inc.
1 Infinite Loop
Cupertino, CA 95014
408-996-1010

Contents

Preface

What to Read xvii
Conventions Used in This Book xviii
Special Fonts xviii
Types of Notes xviii
Numerical Formats xviii

Chapter 1 Introduction 1

Key Concepts 3
About Apple Dylan 4
Standard Dylan 5
Apple Dylan Language Extensions 7
Additional Topics 8
The Apple Dylan Programming Language 8
The Apple Dylan Development Environment 8
The Apple Dylan Application Framework 8

Chapter 2 Apple Dylan Programs 9

Key Concepts 11
About Apple Dylan Programs 12
Program Organization 12
Modules 13
Constituents 13
Program Syntax 14
Comments 15
Whitespace 15
Commas and Semicolons 17
Constituent Syntax 17
Compilation and Execution 20

Additional Topics 21

Chapter 3 Constituents 23

Key Concepts 25
About Constituents 26
Definitions 28
Variable Definitions 29
Class Definitions 30
Method Definitions 31
Local Declarations 31
Local Variable Declarations 32
Local Method Declarations 32
Expressions 33
Literal Constants 34
Variable References 35
Function Calls 36
Statements 37
Additional Topics 39

Chapter 4 Values 41

Key Concepts 43
About Values 44
Objects 46
Classes of Objects 47
Built-In Classes 48
Boolean Values 48
Characters 50
Symbols 51
Number Classes 52
Integers 53
Ratios 53
Floating-Point Numbers 54
Collection Classes 54
Strings 55

Vectors 55
Lists 56
Pairs 57
Additional Topics 59
Objects 59
Built-In Classes 61
User-Defined Classes 62

Chapter 5 Variables 63

Key Concepts 65
About Variables 65
Bindings 66
Scope and Extent 66
Variable Names 68
Variable Values 69
Variable Specializers 70
Creating Variables 70
Module Variable Definitions 70
Constant Module Variables 71
Other Module Variables 72
Local Variable Declarations 72
Using Variables 73
Referencing Variables 73
Modifying Variables 75
Additional Topics 77
Scope and Extent 77
Classes and Functions 78

Chapter 6 Functions 81

Key Concepts 83
About Functions 84
Methods 85
Generic Functions 85
Built-In Functions 86

Calling Functions 87
Function Calls 87
Arguments 88
Evaluation 90
Return Values 91
Creating Functions 92
Creating Methods 92
Formal Parameters 93
Method Bodies 94
Return Values 94
Additional Topics 94
Built-in Functions 95
Methods 95
Generic Functions 97
Calling Functions 98
Function Objects 100

Chapter 7 Statements 103

Key Concepts 105
About Statements 106
Statement Macros and Evaluations 106
Test Expressions 107
Bodies of Code 108
Begin-End Statements 108
Local Declarations 109
Statements as Expressions 110
Conditional Statements 110
Iterative Statements 112
Additional Topics 113
Macros 113
Statements 114

Chapter 8 Methods 115

Key Concepts 117

About Methods 118
Method Definitions 119
Parameter Lists 120
Parameter Type Specialization 123
Method Bodies 124
Local Variable Declarations 125
Return Values 126
Additional Topics 129
Methods and Generic Functions 130
Parameters 130

Chapter 9 Direct Methods 133

Key Concepts 135
About Direct Methods 136
Bare Methods 136
Local Methods 138
Local Recursion 139
Mutual Recursion 140
Method Closures 142
Anonymous Methods 145
Additional Topics 148

Chapter 10 Conditionals 149

Key Concepts 151
About Conditionals 152
If Statements 153
Unless Statements 155
Case Statements 156
Select Statements 158
Additional Topics 160
Statements 160
Predicates 161
Bodies of Code 161

Chapter 11 Iterators 163

Key Concepts 165
About Iterators 166
While Statements 166
Until Statements 168
For Statements 169
  Numeric Clauses 170
  Collection Clauses 172
  Explicit Clauses 173
  End Tests 174
  Result Bodies 175
Additional Topics 176
  Statements 176
  Collections 176

Chapter 12 Objects and Classes 179

Key Concepts 181
About Objects and Classes 182
  Classes 182
  Class Inheritance 184
  Creating and Initializing Objects 185
  Examining and Modifying Objects 186
Built-In Classes 188
  Simple Classes 189
  Number Classes 190
  Collection Classes 192
User-Defined Classes 193
  Class Definitions 193
  Slots 194
  Slot Names and Getter Functions 194
  Other Slot Specification Options 195
Additional Topics 196
  Object Creation and Manipulation 196
  Other Built-In Classes 196
  Class Inheritance 197

Method Dispatch 198
Types 198

Chapter 13 Numbers 201

Key Concepts 203
About Numbers 204
Real Numbers 205
Integers 205
Ratios 206
Floating-Point Numbers 207
Numeric Functions 207
Predicates 208
Arithmetic Functions 209
Rounding and Division 210
Conversion and Simplification 212
Logical Bit-Oriented Functions 213
Comparisons 214
Operators 215
Operator Calls 215
Unary Operators 217
Binary Operators 218
Special Operators 219
Operator Precedence and Association List 220
Additional Topics 221
Operator Applicability 221

Chapter 14 Collections 223

Key Concepts 225
About Collections 226
Kinds of Collections 226
Collection Functions 227
Kinds of Sequences 228
Sequence Functions 228
Strings 229

Byte Strings 230
Stretchy Byte Strings 231
String Functions 233
Vectors 234
Simple Object Vectors 235
Vector Functions 237
Lists 238
Pairs 239
List Functions 241
Additional Topics 242

Chapter 15 User-Defined Classes 245

Key Concepts 247
About User-Defined Classes 248
Class Definitions 249
Adjectives 250
Class Names 250
Superclasses 252
Slot Specifications 252
Initialization Argument Specifications 254
Slots 256
Slot Specifications 256
Slot Accessors 257
Slot Type Specialization 258
Slot Initialization 258
Slot Allocation 262
Constant Slots and Setter Functions 264
Additional Topics 266
Class Creation 266
Instantiation 266

Chapter 16 Inheritance 269

Key Concepts 271
About Inheritance 272

Direct Superclasses 272
Hierarchies 273
Instantiability 274
Dynamism 276
Multiple Inheritance 276
Multiple Direct Superclasses 277
Heterarchies 278
Primary and Free Classes 278
Slot Inheritance 280
Inherited Slot Specifications 280
Initialization-Argument Specifications 281
Slot Allocation 282
Additional Topics 283
Method Dispatch 283
Dynamism 284

Chapter 17 Generic Functions 285

Key Concepts 287
About Generic Functions 288
Implicit Generic Function Creation 288
Slot Accessors 291
Explicit Generic Function Creation 293
Method Dispatch 294
Parameter Type Specialization 294
Singletons 296
Method Specificity 297
Next Method 300
Additional Topics 302
Parameter Lists 302
Class Precedence and Method Specificity 302
Dynamism 303

Chapter 18 Modules 305

Key Concepts 307

About Modules 308
Variable Name Spaces 308
Exporting Variables 310
Importing Variables 311
Subclassing Across Modules 312
Adding Methods Across Modules 313
Libraries 315
Exporting Modules 316
Importing Modules 316
Adding Methods Across Libraries 319
Subclassing Across Libraries 320
Dynamism 322
Sealing Classes 323
Sealing Generic Functions 323
Sealing Branches of Generic Functions 324
Sealing Slot Accessors 325
Additional Topics 326
The Development Environment 326
Dynamism 327

Chapter 19 Macros 329

Key Concepts 331
About Macros 332
Definition Macros 333
Statement Macros 336
Function Macros 338
Defining and Using Macros 339
Conditional Evaluation 339
Free Variables 342
Nested Macro Calls 343
Recursive Macro Calls 344
Input/Output Arguments 345
Additional Topics 346

Chapter 20 Conditions 347

Key Concepts 349
About Conditions 350
Signaling Conditions 351
  Signaling Errors 351
  Signaling Warnings 353
  Signaling Your Own Condition Classes 354
Handling Conditions 355
  Establishing A Handler 356
  Handling Your Own Conditions 358
  Handling a Condition By Declining 359
  Handling a Condition By Returning 360
  Handling a Condition By Restarting 361
  Handling a Condition By Taking a Non-Local Exit 363
Additional Topics 364

Chapter 21 C-Compatible Libraries 365

Key Concepts 367
About C-Compatible Libraries 368
  Interface Definitions 368
  Access Paths 370
Importing C-Compatible Functions 371
  Importing a Macintosh Toolbox Function 371
  Importing a Shared Library Function 373
Importing Other C Entities 375
  Importing C Constants 378
  Importing C Structures 381
  Importing C Structure Pointer Types 383
  Importing C Function Pointer Types 384
Callouts and Callbacks 386
  Callouts: Calling C Functions Using Function Pointers 386
  Callbacks: Calling Dylan from C 387
Additional Topics 390

Chapter 22 Introducing the Framework 391

The Apple Dylan Framework 393
Basic Concepts 395
Windows and Views 395
The View Hierarchy 395
View Determiners 397
Other User Interface Elements 398
Documents and Data 399
Event Handling 399
The Target Chain 399
Mouse Event Handling 402
Changing the Target 402
Behaviors 404

Chapter 23 Framework Tutorial 405

The Skeleton Application 407
Defining a Library and Module 408
Defining Classes 409
Initializing the Application 410
Implementing a Menu Item 411
Creating Windows, Views, and Adorners 412
Drawing in a View 414
Handling an Event 415
Implementing Mouse Tracking 416
Implementing Open Scripting Architecture support 419

Figures and Listings

Chapter 9 Direct Methods 133

Listing 9-1 Mutual recursion 140
Listing 9-2 Creating a simple method closure 143

Chapter 22 Introducing the Framework 391

Figure 22-1 Views in a window 396
Figure 22-2 A window's view hierarchy 397
Figure 22-3 Views and adorners in a dialog box window 398
Figure 22-4 The view hierarchy for a dialog box 399
Figure 22-5 The target chain for a document and its window 400
Figure 22-6 Menu item event handling 401
Figure 22-7 Target chain for an active dialog 403

Chapter 23 Framework Tutorial 405

Listing 23-1 Library and module definitions 408
Listing 23-2 Constants 408
Listing 23-3 Class definitions for a behavior, document, and view 409
Listing 23-4 Initializing the application 410
Listing 23-5 Implementing a menu item 411
Listing 23-6 Setting up windows, views, and adorners 412
Figure 23-1 The initial screen of the skeleton application 414
Listing 23-7 Specifying how to draw the contents of a view 414
Figure 23-2 The screen after drawing data 415
Listing 23-8 Handling an event 416
Listing 23-9 Implementing mouse tracking 417
Figure 23-3 Providing feedback during mouse tracking 419
Figure 23-4 An Apple script 420
Listing 23-10 Converting between Dylan objects and Apple event descriptor records 420
Listing 23-11 Responding to an Apple event 421
Figure 23-5 The static text view after being changed by an Apple script 422

Preface

This book, *Programming in Apple Dylan,* introduces Apple Dylan—both as an implementation of the standard Dylan programming language and a set of language extensions added by Apple Dylan. This book also introduces the application framework provided with Apple Dylan.

You should supplement the material in this book with the *Dylan Reference Manual* and the *Apple Dylan Extensions and Framework Reference*.

What to Read

This book is intended for developers who are interested in programming using Apple Dylan. It also includes introductory information about developing applications without starting from scratch with the Dylan language. Although this book is not designed explicitly to teach programming, it does provide an introduction dynamic, object-oriented programming using the Dylan language.

This book is intended to be read sequentially; most chapters build on the material in earlier chapters. However, a few of the chapters provide introductory overviews of select topics. You may want to skim these chapters first for a broad overview of programming in Apple Dylan. These chapters include:

- Chapter 1, “Introduction”
- Chapter 2, “Apple Dylan Programs”
- Chapter 3, “Constituents”
- Chapter 6, “Functions”
- Chapter 7, “Statements”
- Chapter 12, “Objects and Classes”
- Chapter 21, “C -Compatible Libraries”
- Chapter 22, “Introducing the Framework”

Conventions Used in This Book

This book uses various conventions to present certain types of information.

Special Fonts

All code listings, reserved words, and the names of data structures, constants, fields, parameters, and functions are shown in a monospaced font (`this is monospaced`). When new terms are introduced, they are in **boldface.** These terms are also defined in the glossary.

This book also uses special conventions for describing program syntax. These conventions are introduced in Chapter 2, "Apple Dylan Programs."

Types of Notes

There are several types of notes used in this book.

Note
A note like this contains information that is interesting but possibly not essential to an understanding of the main text. ◆

IMPORTANT
A note like this contains information that is especially important. ▲

Numerical Formats

Hexadecimal numbers are shown in this format: #x0008.

The numerical values of constants are shown in decimal, unless the constants are flag or mask elements that can be summed, in which case they are shown in hexadecimal.

For More Information

APDA is Apple's worldwide source for hundreds of development tools, technical resources, training products, and information for anyone interested in developing applications on Apple platforms. Customers receive the *APDA Tools Catalog* featuring all current versions of Apple development tools and the most popular third-party development tools. APDA offers convenient payment and shipping options, including site licensing.

To order products or to request a complimentary copy of the *APDA Tools Catalog*, contact

APDA
Apple Computer, Inc.
P.O. Box 319
Buffalo, NY 14207-0319

Telephone	1-800-282-2732 (United States) 1-800-637-0029 (Canada) 716-871-6555 (International)
Fax	716-871-6511
AppleLink	APDA
America Online	APDAorder
CompuServe	76666,2405
Internet	APDA@applelink.apple.com

CHAPTER 1

Introduction

Contents

Key Concepts 3
About Apple Dylan 4
Standard Dylan 5
Apple Dylan Language Extensions 7
Additional Topics 8
  The Apple Dylan Programming Language 8
  The Apple Dylan Development Environment 8
  The Apple Dylan Application Framework 8

This chapter introduces the key elements of programming with Apple Dylan.

Key Concepts

This section, "Key Concepts," appears at the beginning of the chapters in this book. It gives a brief overview of the structure of the chapter, and introduces the most important terms and concepts discussed in the chapter.

About Apple Dylan

- **Apple Dylan** is a **programming language** and a **development environment** built around that programming language.

Standard Dylan

- **Standard Dylan** is a powerful new programming language that combines the best features of many programming paradigms into a simple, elegant design.
- Standard Dylan is **object-oriented,** providing **objects, classes**, **inheritance,** and **multiple inheritance.**
- Standard Dylan is also functional, allowing you to manipulate functions as data, apply functions, create **polymorphic functions,** and create special functions called **closures.**

Apple Dylan Language Extensions

- The **Apple Dylan language extensions** are a set of extensions added to the standard Dylan language by Apple Dylan.
- These language extensions include the ability for your Apple Dylan code to call **C-compatible library code,** and for C-compatible library code to call your Apple Dylan code.

Additional Topics

This type of note, "Additional Topics," appears throughout this book. It typically includes cross references to other books, or to additional information contained in the last section of the chapter. See "Additional Topics" on page 8. ◆

About Apple Dylan

Apple Dylan is a powerful programming environment designed around the Dylan programming language. In particular, Apple Dylan includes

- The **Apple Dylan programming language,** which is Apple Computer's implementation of the standard Dylan programming language.

  Dylan is a new programming language that combines the power of dynamic programming and object-oriented programming. Dylan, which stands for DYnamic LANguage, is an OODL, or an object-oriented dynamic language. Like other object-oriented languages, Dylan includes powerful data encapsulation and data abstraction. Like other dynamic languages, Dylan allows a variety of functional programming techniques and provides substantial runtime flexibility.

  The Apple Dylan programming language is a complete implementation of standard Dylan, plus a number of **language extensions**, which add even more flexibility to this powerful language. One of the most important language extensions is the ability for Apple Dylan programs to communicate with **C-compatible libraries.**

- The **Apple Dylan development environment,** which is a programming environment designed around the Apple Dylan programming language. In addition to a powerful **source code editor** and an **incremental compiler,** the development environment contains a number of other useful tools to aid you in your program development.

  For example, the development environment provides a number of different **browsers** that allow you to view different aspects of your program: class hierarchies, polymorphic function relationships, internal object inspection, and so on.

  Much of the power of the Apple Dylan programming paradigm comes from the relationship between the development environment and the **runtime environment.** The two environments are distinct, and can be separated completely when you are ready to ship your application. However, while you are developing your application, the two environments work together closely. For example, the development environment can incrementally compile new source code and download the compiled code into your runtime environment. Also, Apple Dylan provides a Listener window,

which allows you to communicate directly with your application's runtime environment—you can inspect variables, execute functions, evaluate expressions, and so on.

The rest of this chapter introduces the Apple Dylan programming language and the development environment in a little more detail.

Additional Topics

For more information about the Apple Dylan development environment, see "The Apple Dylan Development Environment" on page 8. ◆

Standard Dylan

Dylan is a new programming language designed to unify the best features from a variety of programming methodologies into a simple, elegant language. Dylan is designed to provide the maximum power with a minimum amount of runtime overhead.

Some of the most important aspects of the Dylan programming language are listed here:

- **Object-oriented programming.** Dylan provides a true object-oriented approach to programming. In Dylan, all information is represented during runtime as **objects** in memory—not only data such as numbers, strings, and so on, but also functions, function dispatchers, types, and object classes.

  Objects are categories into **classes,** and classes belong to **class hierarchies,** which determine how classes **inherit** structure and behavior. Dylan also provides **multiple inheritance,** which allows for full class heterarchies.

  As a result, Dylan benefits from the data abstraction and reusability inherent in the object-oriented programming paradigm.

- **Functional programming.** Dylan also provides true functional programming. As functions are objects themselves, they can be passed and manipulated as data during runtime. They can also be applied, combined, curried, and created on the fly.

  Dylan also provides for **polymorphism.** One kind of Dylan function, called a generic function, simply acts as a function dispatcher, selecting the most appropriate implementation given a set of arguments.

Additionally, Dylan provides **closures**—functions that maintain their own private, permanent storage. Closures are in some ways the inverse of objects: where an object is data that also encapsulates functionality, a closure is a function that also encapsulates data.

As a result, Dylan benefits from the flexibility and concision inherent in functional programming languages.

- **Dynamic programming.** Another powerful aspect of Dylan is its dynamic nature. Classes and functions can be created, overridden, subclassed, augmented, and otherwise modified during runtime.

  With many languages, the price of such flexibility is efficiency. However, Dylan provides a number of mechanisms by which you can explicitly control the flexibility/efficiency trade-off.

  With this control, you can effectively choose to give your program the maximum runtime flexibility, or limit its flexibility to the point where efficiency trade-offs are eliminated.

- **Data abstraction and hiding.** In many other object-oriented languages, the concept of data abstraction and data hiding is inextricably combined with the concepts of objects, classes, and inheritance. Dylan separates the object-oriented paradigm from the data hiding paradigm, allowing you a much simpler mechanism for securing data privacy.

- **Built-in utilities.** Dylan provides many built-in classes and functions that provide common data abstractions and functionality for you.

  For example, Dylan provides a complete set of **collection classes,** including arrays, strings, lists, deques, ranges, and hash tables. With these tools built-in, you can focus your development efforts on programming the unique aspects of your application.

- **Automatic memory management.** Dylan also solves one of the greatest challenges facing application developers—memory management—with an efficient memory management system that automatically allocates and reclaims memory for you. You never have to worry about explicitly allocating memory, testing for dangling pointers, or remembering to explicitly free memory.

Additional Topics

For more information about standard Dylan, see "Standard Dylan" on page 8. ◆

Apple Dylan Language Extensions

The Apple Dylan programming language is Apple Computer's implementation of the Dylan programming language. It includes a complete implementation of the standard Dylan programming language, plus a set of language extensions designed to make Apple Dylan an even more useful language.

The following list highlights the Apple Dylan language extensions:

- **C-compatible libraries.** Apple Dylan allows your Apple Dylan program to communicate with C-compatible libraries. That is, your Apple Dylan program can call C-compatible functions (such as those in the Macintosh Toolbox), and functions in C-compatible libraries can call your Apple Dylan code. Also, Apple Dylan provides tools to assist in the necessary type conversions inherent in cross-language function calls.
- **Garbage Collection.** Although Apple Dylan performs automatic memory management for you, which is both efficient and effortless, there are times when you might want to tweak the garbage-collecting behavior to ensure maximum efficiency for your application. Apple Dylan provides a set of tools that allow you some control over the garbage collection process.
- **Termination.** Apple Dylan performs memory deallocation for you, but sometimes you might want to specify behavior for an object when it is deallocated. Apple Dylan provides a set of tools that allow you this flexibility.

Additional Topics

For more information about the Apple Dylan language extensions, see "Apple Dylan Language Extensions" on page 8. ◆

Additional Topics

This section, "Additional Topics," appears at the end of the chapters in this book. It contains information that supplements the information in the chapter and cross references to other books where you can find more information.

The Apple Dylan Programming Language

- **Standard Dylan**

  The rest of Part I of this book introduces much of the standard Dylan programming language. However, for complete information, you should see the *Dylan Reference Manual*.

- **Apple Dylan Language Extensions**

  Some of the Apple Dylan language extensions are also introduced in this book, as, for example, in Chapter 21, "C-Compatible Libraries." However, for complete information, you should see the *Apple Dylan Extensions and Framework Reference*.

The Apple Dylan Development Environment

- **The Development Environment**

  This book mentions the development environment in many places; in particular, in Chapter 18, "Modules." For complete information about the development environment, you should see the *Using the Apple Dylan Development Environment*.

The Apple Dylan Application Framework

- **The Application Framework**

  Chapter 22 and Chapter 23 of this book introduce the application framework included with Apple Dylan. For complete information about this framework, see the *Apple Dylan Extensions and Framework Reference*.

CHAPTER 2

Apple Dylan Programs

Contents

Key Concepts 11
About Apple Dylan Programs 12
Program Organization 12
  Modules 13
  Constituents 13
Program Syntax 14
  Comments 15
  Whitespace 15
  Commas and Semicolons 17
  Constituent Syntax 17
Compilation and Execution 20
Additional Topics 21

This chapter introduces Apple Dylan programs: what they are, how they are organized, how the development environment organizes them, how they are compiled and executed, and so on.

Key Concepts

Apple Dylan Programs

- Apple Dylan programs are made up of **constituents**—individual, executable pieces of Apple Dylan source code.
- Inside an Apple Dylan program, constituents are organized into modules. A **module** is a group of constituents that share a namespace.

Program Organization

- The development environment represents Apple Dylan programs as **projects.**
- Projects have many levels of organization. At the highest level, they can include **libraries** as well as modules.
- Modules contain **source folders,** which contain **source records.**
- Source records contain the actual constituents, such as the **definitions** and the **expressions** that make up your program.

Program Syntax

- You can use two kinds of **comments** in your programs: single-line and double-line.
- You use **delimiters** (such as whitespace, commas, and semicolons) to separate the elements of your program.
- Every different type of constituent has its own format, called its **syntax.**
- This book uses **formal syntax descriptions**, with a variety of typographical conventions, to describe the syntaxes of the various constituents.

Compilation and Execution

- You can use the development environment to **incrementally compile** your program and download new definitions into the application nub.
- You can examine and interact with your program using the tools of the development environment, particularly the Listener window.
- Many examples in this book show how to use the Listener to execute source code and interact with your application's runtime environments.

About Apple Dylan Programs

An Apple Dylan program is a collection of source code written in the Apple Dylan programming language. The individual pieces of executable source code are called the program **constituents.** Examples of constituents include **definitions** (such as variable and function definitions) and **expressions** (such as mathematical operations, functions calls, and control statements).

In an Apple Dylan program, the constituents are organized into groups called **modules.**

You'll use the development environment to edit your source code and to organize your programs. The next two sections, "Program Organization" on page 12 and "Program Syntax" on page 14, describe the organization and contents of an Apple Dylan program.

You'll also use the development environment to compile your programs, download them into runtime environments, and interact with them during runtime. The section "Additional Topics" on page 21 introduces these aspects of the creating an Apple Dylan program.

Program Organization

The Apple Dylan development environment organizes all of the pieces of your Apple Dylan program into a **project**, which contains the various modules of your program. A project can also contain **libraries,** which are collections of modules.

Modules

The module is the basic unit of organization within an Apple Dylan program: A program is comprised of modules; modules are comprised of constituents (definitions and expression). In an Apple Dylan program, every constituent belongs to exactly one module.

Modules are not only used to organize your program; they also define your program's **namespaces**. Each module represents a single namespace: a definition in a module is accessible anywhere inside that module. To be accessible to other modules, definitions must be explicitly exported and imported.

Each module in your program corresponds to a runtime environment in your application. The module's definitions become accessible objects in that runtime environment.

The Apple Dylan development environment provides extra levels of organization to help you organize the source code in your program's modules. Inside each module, you create a collection of **source folders**, which themselves contain collections of **source records.**

The source records contain the individual source code constituents that make up your program.

Constituents

Apple Dylan programs are made up of **constituents**—individual, executable pieces of Apple Dylan source code. Examples of constituents include function definitions, function calls, variable definitions, variable references, if statements, assignment operations, literal constants, local variable declarations, and so on.

All constituents fall into one of three main categories:

- **Definitions** are the declarative parts of Apple Dylan programs. An example of a definition is a **variable definition**:

```
define variable *global* = 0;
```

- **Local declarations** are similar to definitions, except that their scope is to limited bodies of code, whereas a definition typically applies to an entire module. An example of a local declaration is a **local variable definition**:

```
let temp = 37;
```

- **Expressions** are the parts of Apple Dylan programs that represent values or that compute values. Examples of expressions are **literal constants, variable references, function calls,** and **statements:**

```
100                 // a literal constant

x                   // a variable reference

max (x, 100)        // a function call

x + 100             // an operator call (a kind of function call)

if (value < 0)      // a statement
    - value
else
    value
end
```

Constituents can contain other constituents. For example, a function definition might contain local variable declarations and a number of expressions.

These definitions, in turn, typically contain the other kinds of constituents—local variables and expressions.

In sum, an Apple Dylan program is represented by a project in the development environment. That project contains libraries and modules. The modules contain source folders, which in turn contain source records. The source records contain individual constituents, which are typically definitions. Those definitions often contain other constituents, such as local declarations and expressions.

Program Syntax

Basically, an Apple Dylan program consists of constituents organized into modules. As a result, most of the syntax rules concerning Apple Dylan programs are the syntax rules for the different types of constituents.

However, there are some syntax rules that apply to programs generally. The next few sections discuss some of these general rules and then introduce syntax rules for individual constituents.

Comments

Apple Dylan provides two kinds of **comments** you can include in your programs: single-line comments and a multiple-line comments.

A **single-line comment** begins with two slash characters (`//`) and ends at the end of the line:

```
// import the Random() function
```

These comments can occur on the same line as other source code:

```
let temp = 0;        // this is a local variable
```

A **multiple-line comment** begins with a slash-asterisk delimiter (`/*`) and continues until next occurrence of an asterisk-slash delimiter (`*/`):

```
/* An example of a comment
   that takes up two lines. */
```

Whitespace

Apple Dylan uses **whitespace** (which includes spaces, tabs, and end-of-line characters) as a **delimiter**—that is, whitespace separates the various elements of Apple Dylan source code.

You do not always have to use whitespace to separate elements of your source code, however. It is only required in situations where omitting it would change the meaning of the code. For example, whitespace is required in this subtraction operation:

```
value - 1
```

Without the whitespace, this expression would become `value-1`, which is not a subtraction operation—it's a variable name:

```
let value-1 = 37;
```

Some types of constituents use other types of delimiters—parentheses, commas, semicolons, and so on. Whitespace is not required when another delimiter is already being used. For example, consider the function call:

```
find-average ( low , high )
```

Since the parentheses and the comma serve as delimiters to separate the function and the arguments, this function call can also appear as any of the following:

```
find-average (low, high)
find-average( low, high )
find-average(low, high)
find-average(low,high)
```

Where whitespace is allowed, you can use as much as you like. For example, this function call is perfectly acceptable:

```
find-average  (   low  ,

                    high    )
```

However, there are places where you can't use whitespace—at least not without changing the meaning of your program. For example, you cannot use whitespace in the middle of a function name:

```
find - average                  // not a function name
```

This expression is not a function name; rather, it's a subtraction operation on variables named `find` and `average`.

Commas and Semicolons

As mentioned in the previous section, commas and semicolons are used as delimiters in some kinds of constituents. Typically, these delimiters separate items that appear in series.

Here are two examples of how these delimiters are used:

- **Commas** are used to separate items in a list, such as the arguments to a function. No comma follows the last argument in the list:

  ```
  calculate (1, X, X + 2)
  ```

  This function call has three arguments in its argument list: the number `1`, the variable `X`, and the expression `X + 2`.

- **Semicolons** are used to separate multiple constituents that appear in a body of code; for example:

  ```
  define method my-method ( )
      do-something ();
      do-something-else ();
      clean-up ()
  end method
  ```

  You may include an optional semicolon after the last constituent; for example

  ```
  if ( a < b )
      do-something ();
      do-something-else ();
      clean-up ();               // optional semicolon
  end if
  ```

Constituent Syntax

The format of a particular type of constituent is called its **syntax.** As an example, consider these local variable declarations:

```
let y = x

let temp = 0

let the-final-example = max (10, 11)
```

These three examples reveal the basic structure shared by all local variable declarations:

- the word `let`
- the local variable name
- an equals sign (=)
- an expression

These elements, in this order, are common to all local variable declarations; this is the **syntax** of a local variable declaration. It is this syntax, in fact, that determines whether or not a constituent is a local variable declaration. If you write a piece of Apple Dylan source code that does not share this basic structure, it is not a local variable declaration.

This book uses certain typographic conventions to denote the syntax structures for the various types of constituents. For example, the syntax for local variable declarations is denoted as

let *variable-name* = *initial-value-expression*

In this syntax description, the word `let` and the equals sign (=) are shown in `plain computer voice` to indicate that all local variable declarations share these elements exactly.

The elements *variable-name* and *initial-value-expression* are shown in *italicized text*. This notation indicates the parts of a local variable declaration that change from declaration to declaration.

As you continue through the chapters of this book, you'll see many of these **syntax descriptions**, detailing the format of the various types of constituents.

The list that follows introduces all of the typographic conventions this book uses in syntax descriptions. If you aren't familiar with formal syntax descriptions, this list may not be clear to you. Don't worry—you can read it now for a quick introduction to syntax descriptions and reference it when you start encountering the actual syntax descriptions in this book.

Every item in this list describes a different type of element found in syntax descriptions. For each type of syntax element, the typographic convention used to identify the element is shown, a few examples are given, and instructions on how you interpret that kind of element in a syntax description (that is, how that element corresponds to real Apple Dylan source code).

- **Copied elements** are shown in `plain computer voice`:

  ```
  begin
  define variable
  if
  ```

  You copy these elements exactly when creating an actual constituent.

- **Replaced elements** are shown *italicized text*:

  local-variable-declaraion
  expression

  You replace these elements with appropriate constituents.

- **Optional elements** are surrounded by bold brackets (**[** and **]**):

  [`unless` **]**

  [*expression* **]**

  You may choose to include or to omit optional elements.

- **Repeated elements** are surrounded by bold braces (**{** and **}**):

  { *argument* **}**

  { *expression* **}**

  You may include a repeated element as many times as you'd like—0, 1, or more times

  If the close brace is preceded by a **delimiter** (such as a comma or a semicolon), then you must separate the repeated elements with that delimiter:

  { *argument* **,}**

  If you may include an **optional delimiter** after the last repeated element, the syntax description uses this notation:

  { *expression* **;} [**;**]**

- **Element choices** are indicated by a bold vertical bar (**|**):

  local-declaration **|** *expression*

You may choose to include either option (but not both). More than two choices can be offered:

literal-constant | *variable-reference* | *function-call*

To avoid ambiguity, element choices must sometimes be surrounded by parentheses:

(*local-declaration* | *expression* **)**

When surrounded by parentheses, this notation indicates that you must include one of the choices. If the element choices are optional, they can be surrounded by brackets:

[*local-declaration* | *expression* **]**

In this case, you may choose to include either option, or neither.

Repeated element choices are surrounded by braces:

{ *local-declaration* | *expression* **}**

In this case, you may include any number of either option in any order.

Compilation and Execution

During development, you'll probably use the development environment to incrementally compile you source code, loading new definitions from your modules into the corresponding runtime environment of your Application Nub.

At this point, you'll usually examine the new definitions using the tools provided by the development environment. In particular, you can examine new objects and test new functions by using the Listener. (The Listener is also a wonderful tool as you are learning about Apple Dylan. This book includes many simple examples that you can execute with the Listener.)

Commonly, you'll define a start-up function in your program, and then call that function from the Listener when you want to test your program as a whole.

When your finished programming, the development environment allows you to make a stand-alone application based on your compiled code and the start-up function that you specify. When this application is executed, it creates the same runtime environments, just no longer tethered to the development environment.

Additional Topics

This chapter provided a brief introduction to a wide range of subjects related to programming with Apple Dylan. The following list describes where you can find additional information about these and related topics.

- **The development environment.** This chapter mentions only a few of the many features available to you as you program with Apple Dylan. *Using the Apple Dylan Development Environment* lists many more helpful features, it contains programming examples, and it provides complete reference material for the development environment.
- **Modules.** Modules allow you to organize your source code, create variable namespaces in your source code, and correspond to runtime environments in your application. You can find more information about modules in Chapter 18, "Modules." That chapter also introduces libraries and applications, but for complete information, see the *Dylan Reference Manual*.
- **Syntax.** Throughout this book, you'll see syntax descriptions for the different types of constituents you can use in your Apple Dylan program. You can find a complete formal grammar for the Apple Dylan programming language in the *Dylan Reference Manual*.
- **Constituents.** The next chapter, "Constituents," gives an overview of the various types of constituents you use when creating Apple Dylan programs. That chapter focuses on the most commonly-used types. More are described later in this book, such as in Chapter 19, "Macros," and Chapter 20, "Errors."

CHAPTER 2

Apple Dylan Programs

CHAPTER 3

Constituents

Contents

Key Concepts 25
About Constituents 26
Definitions 28
  Variable Definitions 29
  Class Definitions 30
  Method Definitions 31
Local Declarations 31
  Local Variable Declarations 32
  Local Method Declarations 32
Expressions 33
  Literal Constants 34
  Variable References 35
  Function Calls 36
  Statements 37
Additional Topics 39

In the previous chapter, you learned that constituents are the basic building blocks of Apple Dylan programs. This chapter introduces the different kinds of constituents and provides simple examples of the most common kinds.

This chapter is only a brief overview—later chapters of this book examine the different kinds constituents in much more detail.

Key Concepts

Constituents

- **Constituents** are individual, executable pieces of Apple Dylan source code.
- The three basic categories of constituents are **definitions, declarations**, and **expressions.**

Definitions

- Definitions apply to entire modules; they often create entities in the corresponding runtime environment.
- Definitions may not be contained inside a body of code, or inside any other constituent.
- There are nine kinds of **built-in definitions,** including **variable definitions, method definitions, class definitions,** and so on.

Local Declarations

- Local declarations apply only within a single body of code; their effects last only while that body of code is executing.
- Local declarations must be contained within a body of code.
- There are three kinds of **local declarations,** including **local variable declarations** and **local function declarations.**

Expressions

- Expressions represent or compute values. Executing an expression is called **evaluating** the expression.

- A single expression may return **multiple values.**
- The most common kinds of **expressions** are **literal constants, variable references, function calls,** and **statements**.

About Constituents

Apple Dylan programs are made up **constituents**—individual, executable pieces of Apple Dylan source code. Examples of constituents are function definitions, function calls, variable definitions, variable references, conditional statements, assignment operations, literal constants, local variable declarations, and so on. Despite the wide variety of constituents, they all fall into one of these three categories:

- **Definitions** are declarative constituents that apply to entire modules. You can use definitions to create constants, variables, functions, and macros. You can also use them to import and export entities from modules, libraries, and C-compatible libraries.

  As an example, here is a variable definition:

  ```
  define *global* = 0
  ```

  This definition declares the variable name `*global*` for the entire module containing this definition—this variable name can be referenced anywhere in that module.

  When executed, this definition creates a variable named `*global*` with a value of 0 in its corresponding runtime environment. When other references to `*global*` from the same module are executed, they access this variable in the runtime environment.

- **Local Declarations** are also declarative constituents, but they apply to a limited body of code. You can use them to create local variables, local functions, and local error-handlers.

  Here is an example that contains a local variable declaration:

  ```
  begin
      let temp = 5;
  end
  ```

This example shows a local declaration contained within a very simple body of code. This declaration declares the variable name `temp`, but only for the body of code that contains it. This variable name cannot be referenced by constituents not contained within the same body of code.

When this local declaration is executed, it creates a variable named `temp` with a value of 5. When the code body that contains it finishes executing (in this case, immediately after), that variable becomes inaccessible and any memory it used can be reused.

- **Expressions** are constituents that represent or compute values. You can use expressions to specify a value, to reference the value of a variable, to call a function, to perform mathematical operations, and to execute code conditionally or iteratively. Executing an expression, which is also called **evaluating** the expression, produces a value (or multiple values) as a result. This value (or values) is called the **value** of the expression. Here is an example of an expression that represents a value:

  ```
  100
  ```

  This expression is a literal constant. When it is evaluated, it returns the integer value 100.

  Here is an example of an expression that computes a value:

  ```
  max (10, 100)
  ```

  This expression is a function call. When it is evaluated, the `max` function operates on the arguments 10 and 100, computes that 100 has the maximum value of these arguments, and returns 100 as the function result. Therefore, the integer value 100 is the value of the expression.

Here is a key point to remember as you learn about constituents: *constituents are source code*. A constituent is simply a string of characters that you type into your project window, or into the Listener window. Constituents exist only in the development environment.

When you execute a constituent, the development environment compiles the source code and sends it to the runtime environment.

Some constituents, as a result, add information to the runtime environment. Others modify information. Still others leave the runtime environment unaffected, but instead produce a value (or multiple values).

The next few sections examine the three main categories of constituents in more detail. You'll see more examples of actual constituents and learn what happens when they are executed.

Definitions

Apple Dylan provides a number of different kinds of **definitions** for you to use. Each is slightly different, but in general they allow you to

- **declare** names that can be referenced anywhere with in a module
- **define** entities that exist in the corresponding runtime environment

It is the very nature of definitions to add to their runtime environments. In fact, definitions are the mechanisms by which you create runtime environments.

You cannot override this and limit a definition to a local body of code. As a result, you can not have a variable definition contained within a body of code. For example, consider this if statement:

```
if ( balance < 0 )
    define variable free-cash = 0;       // not allowed
end;
```

This example is not valid Apple Dylan code. In this example, a variable definition (which is meant to create a variable accessible to an entire module) appears within the body of an `if` statement, implying that the variable is only declared within that body of code and defined while the if statement is executing. To achieve this affect, you must use a local variable declaration.

The next few sections discuss the three kinds of definitions that you'll probably use the most:

- **Variable definitions** allow you to create named values.
- **Class definitions** allow you to create new classes of values.
- **Method definitions** allow you to create functions.

All three of these definitions declare a name that can be referenced anywhere within the definition's module. When executed, they also create entities that exist during runtime—named values, named classes, and named functions.

The six other built-in kinds of definitions are introduced later in this book.

- **Constant definitions** create named values whose value cannot be changed. See Chapter 5, "Variables," for more information.
- **Generic function definitions** allow you to create polymorphic functions (functions with different implementations for different types of arguments). See Chapter 6, "Functions," for an introduction to polymorphic functions and Chapter 17, "Generic Functions," for information about defining them.
- **Macro definitions** allow you to create macros (rules for translating source code, called macro calls, into executable constituents). See Chapter 19, "Macros," for more information.
- **Module definitions** and **library definitions** allow you to organize and share source code, and to control the accessibility of variables. See Chapter 18, "Modules," for more information.
- **Interface definitions** allow you to import information from C header files for use in cross-language communications with C-compatible libraries.See Chapter 21 for information about interface definitions.

Variable Definitions

A variable definition allows you to create a variable—that is, to associate a variable name with a value. The syntax of a variable definition is

`define variable` *variable-name* `=` *initial-value-expression*

Here is an example:

```
define variable *bakers-dozen* = 12 + 1
```

This variable definition

- begins with the required words `define variable`
- specifies `*bakers-dozen*` as the variable name
- includes the required `=` punctuation
- ends by specifying `12 + 1` as the initial-value expression

When this variable definition is executed, the expression `12 + 1` is evaluated, resulting in the integer value 13. In the runtime environment, the variable name `*bakers-dozen*` is associated with this value.

Subsequently, references to `*bakers-dozen*` in the same module access this value.

Apple Dylan has certain naming conventions for variable names. For example, modifiable module variables begin and end with an asterisk, as in `*bakers-dozen*`. In addition to these conventions, which are optional, there are definite rules you must follow to create valid variable names.

See Chapter 5, "Variables," for more information about variables, variable names, and variable definitions.

Class Definitions

A **class definition** allows you to create a new class. In Apple Dylan **classes** are similar to types in programming languages like C or Pascal. In particular, defining your own class in Apple Dylan is like defining your own structure in C or record in Pascal. (Of course, Apple Dylan classes are much more powerful!)

Here is an example:

```
define class <address> (<object>)
    slot number;
    slot street-name;
end class;
```

This definition creates a new class—more specifically, a new subclass of the built-in class `<object>`. This new class is named `<address>`. This class has two **slots**, which are like C structure members or Pascal record fields. These slots are named `number` and `street-name`.

You can create actual objects of this class using the built-in `make` function; for example:

```
define variable *my-address* = make(<address>);
```

When executed, this line of code evaluates the initial-value expression `make(<address>)`, which creates a new object in the runtime environment. This object is associated with the variable name `*my-address*`.

For more information about creating your own classes of objects, see Chapter 12, "Objects and Classes."

Method Definitions

A **method definition** allows you to create a function and specify its implementation. Here is an example:

Here is an example:

```
define method double (x)
    x * 2;
end method;
```

This definition creates a new function, named `double`. You can subsequently call this function (within the same module as the function definition). For example:

```
double(2)
```

If you were to type this function call into the Listener window, the result you'd get is

```
> double(2)
4
```

See Chapter 6, "Functions," and Chapter 8, "Methods," for more information.

Local Declarations

Local declarations also declare names and define entities, but the effects are temporary, limited to specific bodies of code. A local declaration has no effect outside of the body of code that contains it. Therefore, all local declarations must be contained within some body of code.

The next two sections discuss:

- **local variable declarations**, which allow you to create local variables
- **local method declarations**, which allow you to create local functions

There is a third kind of local declaration, the **local handler declaration,** which you can use for error handling, as described in Chapter 20, "Conditions."

Local Variable Declarations

Local variable declarations allow you to create temporary variables, variables that are only accessible locally within a specific body of code. As an example, consider this if statement

```
if ( *bakers-dozen* > 12 )
    let temp = 10;
    temp + temp;
end if;
```

When this if statement is executed, if the value of the variable `*bakers-dozen*` is greater than 12, then body of the if statement is executed.

In the statement body, the local variable declaration is executed, and a variable named `temp` with a value of 10 is created. This variable is very short-lived, however. No code outside the body of this if statement can access this `temp` variable or its value. When the if statement finishes executing (shortly thereafter), the variable becomes permanently inaccessible.

Chapter 5, "Variables," and Chapter 8, "Methods," explain local variables in more detail.

Local Method Declarations

Local method declarations allow you to create a temporary function that is accessible only locally from within the body of code that contains it. Here is an example:

```
define method quadruple (x)

    local method double (y)
```

```
            y * 2;
        end double;

    double(double(x));

end method;
```

This example defines a function named `quadruple`, which contains a local method named `double`, This local method can only be called within the body of the `quadruple` method.

See Chapter 9, “Direct Methods,” for more information about local methods.

Expressions

Expressions are the parts of Apple Dylan programs that represent values or that compute values. **Evaluating** an expression is the process of executing the expression to determine its resulting value. The **value** of an expression is the value returned when that expression is evaluated.

Actually, expressions are not restricted to a returning a single value —they can return any number of values.

The four most common types of expressions are

- **Literal constants,** which allow you to express certain types of values explicitly. Here are some examples:

```
100         // an integer
'a'         // a character
"hello"     // a string
```

- **Variable references,** which allow you to use a variable name to reference the value of the variable. Here are some examples:

```
x
range-size
*global*
```

- **Function calls,** which allow you to apply a function to a list of arguments. Here are some examples:

```
max ( 10, 20 )
negative ( x )
- x
3 - x
```

- **Statements,** which provide program control such as conditional or iterated execution. Here is an example:

```
if (value < 0)
    - value
else
    value
end
```

Literal constants and variable references are fairly simple expressions; they don't contain other expressions.

Function calls and statements, on the other hand, can be significantly more complex. Both function calls and statements have subexpressions. For a function call, these are called **argument expressions**. Part of evaluating a function call is evaluating its argument expressions.

How function calls and statements evaluate their subexpressions, and in what order, is of significant importance.

The next four sections of this chapter examine these four types of expressions in more detail.

Literal Constants

A **literal constant,** also called a **literal,** is an Apple Dylan expression that specifies an explicit value. There are different types of literal constants, each with its own syntax, to allow you to specify different types of values. Here are some examples:

```
#t        // the Boolean value true

'Q'       // the character Q
```

```
37        // the number 37

"ABCDE"   // a string containing the characters A B C D and E
```

Literal constants are constituents—that is, they are executable pieces of Apple Dylan source code. You can use the Listener window to evaluate a literal constant:

```
> 37
37
```

When you type the literal constant into the Listener, Apple Dylan compiles it, evaluates it, and prints the resulting value. In the simple example above, the literal constant `37` evaluates to the integer value 37. The development environment prints this result to the Listener window.

Chapter 4, "Values," introduces the many different types of values that are built-in to Apple Dylan and continues the discussion of literal constants.

Variable References

A **variable reference** is an expression that allows you to access the value of a variable.

The syntax of a variable reference is simple:

variable-name

A variable reference is simply the name of the variable. The value of a variable reference is the current value associated with that variable name in the current runtime environment.

For example, suppose you define a variable by typing this variable definition into the Listener window:

```
> define variable *global* = 100;
defined *global*
```

You can determine the value of the variable by evaluating a variable reference:

```
> *global*
100
```

Apple Dylan recognizes the variable name you enter into the Listener window as a variable reference and then sends a compiled version of that reference to the current runtime environment to be evaluated. The runtime environment evaluates the variable reference by determining the value currently associated with that variable name. It returns that value as the result of the variable reference, and it is printed to the Listener window.

Chapter 5, "Variables," examines all aspects of variables in more detail.

Function Calls

A **function call** is an expression that specifies a function and some number of **argument expressions**. When you evaluate a function call, Apple Dylan evaluates the argument expressions, applies the specified function to the resulting argument values, and returns any values returned by the function.

The standard syntax for function calls is

function-name (*argument-list*)

The *function-name* identifies the function. The *argument-list* is a comma-separated list of argument expressions, with this syntax:

{ *argument-expression* ,}

Here is an example of a function call using the standard syntax:

```
max ( *global*, 10 )
```

This function call contains two argument expressions—the first is a variable reference and the second is a literal constant.

When evaluating this function call, Apple Dylan evaluates the argument expressions first, passing the resulting values to the built-in `max` function. (This function compares the argument values and returns the one with the maximum value.)

Assuming the `*global*` variable still has the value 37, as assigned in the previous section, here is the result of evaluating the function call:

```
> max ( *global*, 10 )
37
```

There are other function call syntaxes that are used for different purposes. For example, the **operator call syntax** is more natural when performing mathematical calculations. Here is an example:

```
3 + 4
```

This expression is a special kind of function call, an **operator call.** Like a standard function call, an operator call specifies both a function and a set of argument expressions. In the above example, the function specified is the built-in addition function and the argument expressions are the literal constants `3` and `4`.

Operators are described in Chapter 6, “Functions,” and in Chapter 13, “Numbers.”

There are two other special types of function calls: array references, described in Chapter 14, and slot references, described in Chapter 15.

Statements

Statements are expressions that allow you to control the flow of execution in your program. You can use statements to create bodies of code, conditionally execute them, iteratively execute them, and even exit from them to perform error handling.

Like function calls, statements contain subexpressions. However, statements differ from function calls in a number of important ways:

- Statements can contain local declarations.
- Statements use their own syntaxes, which are not at all like function call syntax.
- Statements don’t always evaluate all of their subexpressions.

The simplest kind of statement is the **begin-end statement.** This statement allows you to gather several constituents (local declarations and expressions only) into a single body of code. Here is an example:

```
begin
    let temp = 13;
    temp - 1;
end
```

This begin-end statement creates a body of code with two constituents: a local variable declaration and a subtraction operation. These constituents are separated by a semicolon and an optional semicolon follows the last constituent.

When a begin-end statement is evaluated, Apple Dylan executes the body of code. The value returned by the begin-end statement is the value returned by the last constituent evaluated in the statement. For example:

```
> begin
    let temp = 13;
    temp - 1;
  end
12
```

You can use **block statements** to create bodies of code that can handle error situations. Chapter 20, "Errors," discusses error handling and block statements in detail.

All of the other built-in kinds of statements fall into two categories:

- **Conditional statements.** These statements allow you to create a body of code to be executed only under certain conditions. The if statement is an example:

  ```
  if ( *pay-check* < $minimum-wage )
      report-to-authorities ( *my-company* );
      *my-company* := find-new-job ();
  end if;
  ```

  This if statement creates a body of code with two constituents: a function call and an assignment operation. It also contains a test expression. When the if statement is executed, this test expression is evaluated. If the result is *false*, the if statement does not execute the body of code; instead, it immediately returns *false*. If the result is not *false*, the body of code is executed and the value returned by the last constituent executed is the value returned by the if statement.

Chapter 10, "Conditionals," gives more information about if statements and other conditional statements you can use.

- **Iterative statements** allows you to execute a body of code repeated times. Here is an example, using a while statement:

```
while ( *my-score* < $passing )
    take-test ();
    *my-score* := update (*my-score*);
end while;
```

In this while statement, the test expression is evaluated, and if it is not *false*, the body of code is executed and the whole process repeats.

Chapter 11, "Iterators," gives more information about while statements and other iterative statements you can use.

Additional Topics

This chapter highlighted common program constituents and gave some simple examples. Throughout the rest of this book, you can find more information about these constituents, as well as some not mentioned here.

In particular, here are a few of the more advanced topics related to the information presented in this chapter:

- **Function objects.** When you define a function, Apple Dylan creates an object in the runtime environment to represent that function. The function name is really a variable name whose value is the function object. See Chapter 6, "Functions," for further discussion and the *Dylan Reference Manual* for complete information.
- **Operator names.** Function names are simply variable names bound to a function object. You use the same function name to refer to the function object as you do when calling the function. Operator functions, however, can operate a little differently. The symbol you use when calling the operator may not be the variable name bound to the operator's function object. See Chapter 6 for more details.
- **Statement macro calls.** In this chapter, you saw a few examples of statements. A statement is actually a call to a special kind of macro—a statement macro. The statements you've seen so far are actually calls to

built-in statement macros. You can create your own kinds of statements by defining your own statement macros. See Chapter 19, “Macros,” for more details.

- **Function macro calls.** This chapter introduced four kinds of expressions. There is a fifth—the function macro call. These expressions look like function calls, but are actually calls to function macros. See Chapter 19, “Macros,” for more details.

CHAPTER 4

Values

Contents

Key Concepts 43
About Values 44
  Objects 46
  Classes of Objects 47
Built-In Classes 48
  Boolean Values 48
  Characters 50
  Symbols 51
Number Classes 52
  Integers 53
  Ratios 53
  Floating-Point Numbers 54
Collection Classes 54
  Strings 55
  Vectors 55
  Lists 56
  Pairs 57
Additional Topics 59
  Objects 59
  Built-In Classes 61
  User-Defined Classes 62

In Chapter 3, you learned that Apple Dylan programs manipulate individual units of information called **values**—literals represent values, variables have values, function calls have input values and output values, statements return values, and so on.

In this chapter, you'll learn what an Apple Dylan value really is. You'll also see some examples of the different types of values that are built-in to Apple Dylan, and you'll learn more about the literal constants you can use to specify a particular value.

Key Concepts

About Values

- A **value** is a unit of information—a unit of data— in Apple Dylan.
- Examples of values include: the value of a variable, the value of a literal constant, the values returned by a function call, and the values returned by a statement.
- Apple Dylan represents values as data **objects** that exist in memory—that is, they exist in runtime environments.
- Evaluating a literal constant or a variable references produces a value—a data object in memory.
- Evaluating a function call or a statement can result in **multiple values**—that is, functions and statements produce 0, 1, or more objects when evaluated.

Classes

- Objects are grouped into **classes.** Every object belongs to (is a direct instance of) exactly one class.
- An object's class defines the object's structure and its behavior.

Built-In Classes

- Apple Dylan provides **built-in classes** for common types of values. These classes are optimized for space and efficiency.

- Two simple examples are the **character class** and the **symbol class.**
- Many of the common built-in classes allow you to specify specific values using **literal constants.**

Number Classes

- Another set of built-in classes are the number classes, which include integers, ratios, reals, floating-point numbers, and so on.

Collection Classes

- An important and powerful set of built-in classes are the collection classes, which allow you to represent and manipulate groups of values.
- The collection classes include **strings, vectors, lists, pairs,** and so on.

Additional Topics

There are many classes of objects not even mentioned in this introductory chapter, including classes you define yourself. Be sure to see the "Additional Topics" section at the end of the chapter for a survey of related topics and pointers to additional information. ◆

About Values

Values are the units of information that Apple Dylan programs manipulate. Examples of values include the number 10, the character *Q*, the Boolean value true, the fraction 4/3, and the string *Hello*.

As you learned in the previous chapter, representing some kinds of values in your program is as easy as using a literal constants. For example:

```
10
'Q'
#t
"Hello"
```

You can name values—that is, you can assign a value to a particular variable name. For example,:

```
define variable *an-intger* = 10;
define variable *a-fraction* = 4/3;
define variable *a-letter* = 'Q';
define variable *a-truth-value* = #t;
define variable *a-string* = "Hello";
```

While variables allow you to name values, functions allow you to manipulate them. In general, a function takes some number of input values (called **arguments**) and produces some number of output values (called **return values** or **function results**). The number of argument values and return values depends on the function. For example, the `max` function can take any number of argument values, and it produces one return value:

```
> max (0, 5, 10, 15, 20)
20
```

As another example, the equality operator (`=`) is a function that takes two argument values and returns a single (Boolean) return value:

```
> 10 = 11
#f
```

In this example, the argument values to the `=` operator are the integers 10 and 11. The return value is the Boolean value *false* (which the Listener prints as `#f`).

Some functions return multiple values. For example:

```
> truncate ( 4/3 )
1
1/3
```

In this example, the `truncate` function takes one argument value—the fraction 4/3— and returns two return values—the integer part, (the value 1), and the remainder part, (the value 1/3).

Statements also examine input values and produce return values. For example, this if statement examines the Boolean input value *true*, and returns the string value *Hello*.

```
if (#t)
    "Hello";
end if
```

As you can see, Apple Dylan programs are constantly manipulating values—storing values in variables, sending values to functions, receiving values returned from functions, examining values in statements, executing bodies of code that return values, and so on.

Executing an Apple Dylan program is a constant process of creating values, passing values to functions, receiving values returned from functions, storing values in variables, performing operations on values, and so on. During runtime, these values are represented as data objects—objects that exist in memory in runtime environments.

As an Apple Dylan program is running, these values are represented by data objects.

Objects

In Apple Dylan, every value is an **object** in memory. For example, evaluating the literal constant

```
10
```

creates an object in the memory of the runtime environment. Similarly, if your program includes a variable definition

```
define variable x = 11;
```

then, during runtime, the variable name `x` is associated with an object representing the value 11—that is, the *value* of the variable `x` is an *object* representing the number 11.

As another example, when you evaluate the function call

```
concatenate ( "hot", "dog" );
```

Apple Dylan first creates an object that represents the string *hot* and another object that represents the string *dog.* These objects are passed to the `concatenate` function, which creates and returns a third object—one that represents the string *hotdog*.

Notice two important points from these examples:

- Objects may have a name (like the object that represented the number 11 had the name `x`), but they may exist in memory without a name (like the objects that represented the strings *hot* and *dog*). Values are objects in memory independent of whether they have a name.
- There are different categories of objects. For example, some objects represent numbers, like the object that represents the number 11, and some objects represent strings, like the object that represents the string *hot*.

The next chapter, "Variables," discusses the names of objects, while the rest of this chapter introduces some of the different categories of objects.

Classes of Objects

Apple Dylan groups objects into categories called **classes.** Every object has a class to which it belongs. An object's class defines the object's structure and the object's behavior.

For example, Apple Dylan provides a built-in class for objects that represent characters. This class is called `<character>`.

Note
By convention, Apple Dylan class names begin with an open angle bracket (`<`) and end with a close angle bracket (`>`). Although Apple Dylan does not require class names to follow this naming convention, it is recommend that you use this convention for consistency and clarity. ◆

You can create a character object using a literal constant. For example, the character literal `'Q'` in your source code translates into the runtime environment as a character object representing the letter *Q*. The literal `'?'` similarly translates into a character object representing the exclamation point character. Both of these objects are character objects—they both belong to the class called `<character>`. This class dictates how the objects are represented in memory—how many bytes of memory they use, how the bits in the object should be interpreted, and so on.

The `<character>` class also defines the behavior of these objects. For example, because these objects are character objects, you can apply the function `as-lowercase` to them:

```
> as-lowercase('Q')
'q'

> as-lowercase('!')
'!'
```

If these objects belonged to another class, such as the <integer> class, this function would no longer be applicable.

Apple Dylan provides the <character> class and the <integer> class, as well as a number of other **built-in classes** that you can use to create and manipulate common types of values. The next few sections in this chapter discuss some of these basic classes of objects

Additional Topics

There are many more built-in classes than described in this chapter. See "Built-In Classes" on page 61.

Apple Dylan also allows you to define your own classes of objects. See "User-Defined Classes" on page 62. ◆

Built-In Classes

Apple Dylan provides many **built-in classes** of data objects. In this section, we'll examine three of the simplest classes of objects: Booleans, characters, and symbols.

Two other important categories of built-in classes are the number classes, which are introduced beginning on page 52, and the collection classes, which are introduced beginning on page 54.

Boolean Values

Boolean values represent the values *true* and *false*. Apple Dylan provides the built-in <Boolean> class for objects that represent these values. Every object of the class <Boolean> either represents the value *true* or the value *false*.

Boolean literals allow you to specify the Boolean objects for *true* and *false*. These two literal constants are `#t` and `#f`:

- The literal `#t` evaluates to the object that represents *true*.
- The literal `#f` evaluates to the object that represents *false*.

You can also create these `<Boolean>` objects in other ways. For example, evaluating the function call

```
odd? (7)
```

returns the `<Boolean>` object for *true*, while evaluating the equality comparison

```
"hello" = "goodbye"
```

returns the `<Boolean>` object for *false*.

You can evaluate these expressions using the Listener:

```
> odd? (7)
#t

> "hello" = "goodbye"
#f
```

In each of these evaluations, the development environment compiles the expression and downloads it into the runtime environment. The runtime environment evaluates the expression, creates an object in memory (an object of the class `<Boolean>`), and returns that object to the development environment. The development environment then translates that object, which is in Apple Dylan internal format, into a printable form. In this case, the *true* object is translated to the printable form `#t` and the *false* object is translated to the printable form `#f`.

Additional Topics

Apple Dylan stores objects using blocks of memory called object references. Some values, like Boolean values, are small enough to fit completely within the object reference. These values are called **immediate values.** Other, larger, values use the object reference as a pointer to the actual memory that stores the actual value. See "Implementation" on page 60.

Also, some objects are **mutable**—that is, their contents can be changed—while others are **immutable**—that is, their contents cannot be altered. Boolean objects are immutable—you cannot change the contents of a Boolean object. See "Mutability" on page 60. ◆

Characters

The `<character>` class of objects are used to represent individual character values, such as a lowercase *a*, an uppercase *Z*, a question mark *?*, and so on.

The simplest way to create a character object is use a character literal constant. **Character literals** are single characters delimited by single quotes. Some examples are

```
'a'
'Z'
'?'
```

Each of these literals evaluates to an object. The literal `'a'`, for example, evaluates to an object of the class `<character>`. This object contains the internal representation Apple Dylan uses for the character *a*.

As another example, consider evaluating the following function:

```
> as-uppercase ('a')
'A'
```

This function call evaluates the character literal `'a'` and creates an object of class `<character>` containing the internal representation of the character *a*. The function creates a new <character> object representing the character *A*, and

returns that object as the return value of the function. The Listener converts this object from its internal format to the printable form: `'A'`.

Additional Topics

Like Boolean objects, character objects are immediate and immutable. See ◆

Symbols

A **symbol** is an object of the `<symbol>` class. Each symbol object contains a unique string of characters. For example, one symbol object might represent the five-character symbol *street*, while another might represent the seven-character symbol *address*.

Symbols can also include spaces and other characters; for example, a single symbol object can represent the twelve-character symbol *hello, world* (a symbol with a comma and a space).

An interesting aspect of symbols, and what makes them different than strings, is that each symbol object is unique—that is, no two symbol objects represent the same symbol. This is not necessarily so with other classes of objects. For example, it is possible to have two different string objects in memory at the same time that both represent the four-character string *date*. Since string objects are mutable, you could change one of these string objects without changing the other.

Symbol objects, on the other hand, are guaranteed to be unique. Apple Dylan maintains a table of all the current symbol objects. If you try to create a new symbol object with a value that none of the existing symbol objects have, Apple Dylan adds the new symbol object to its internal table. However, if you try to create a symbol object with the same value as an existing symbol object, Apple Dylan simply returns the existing symbol object.

You can specify symbol objects using **symbol literals.** These constants use the `#""` syntax to allow you to specify the value of the symbol. Some examples of symbol literals are

```
#"date"
#"time"
#"another-date-and-time"
#"a symbol with spaces"
#"Symbol No. X3476"
```

Apple Dylan provides another syntax—the **keyword syntax**—that you can also use to specify symbol values. To use this syntax, you follow the sequence of characters that identifies the symbol with a colon (:). Some examples are

```
a-symbol:
another-symbol:
a-keyword-symbol-cannot-have-spaces:
Symbol-Number-X3476:
```

The colon (:) indicates where the symbol literal ends. However, the keyword syntax doesn't provide any indication of where the symbol literal begins. Therefore, this syntax only works for a limited subset of symbols; it does not work, for example, for symbols that contain whitespace.

Notice that the two syntaxes allow you to specify keyword symbols in two different ways. For example:

```
#"a-symbol" = a-symbol:
```

Both of these literal constants refer to the same symbol object—the one that contains the characters *a-symbol*.

Additional Topics

Symbol objects are not immediate—they use their object reference as an index into the symbol table. However, they are immutable. See "Mutability" on page 60 and "Simple Objects" on page 61. ◆

Number Classes

Apple Dylan provides a wide variety of number classes you can use to create objects that represent numerical values. The next few sections introduce some

basic classes of number objects, but Chapter 13, "Numbers," contains more complete information.

Numbers come in a variety of classes. As a partial list, you can use number objects to represent integer values, fractions, and floating-point numbers.

Additional Topics

Apple Dylan number objects are immutable. See "Mutability" on page 60 and "Simple Objects" on page 61. ◆

Integers

Integer objects are instances of the class `<integer>`. You can use integer objects to represent integer values. Actually, Apple Dylan uses two different classes for integer objects: `<small-integer>` for integers less than 2^{28} and `<big-integer>` for larger integers.

Integer literals allow you to specify positive and negative integers, as well as integers in decimal, binary, octal, and hexadecimal notation. Some examples are

```
100        // a decimal integer
-100       // a negative integer
#b1100100  // a binary integer
#o144      // an octal integer
#x64       // a hexadecimal integer
1000000000 // a big integer
```

Each of these literal constants evaluates to an object representing an integer value.

Ratios

Although Apple Dylan provides floating-point numbers, introduced in the next section, the <ratio> classes allow for more accurate calculations with less error propagation. Like integers, ratios come in two flavors: `<big-ratio>` objects and `<small-ratio>` objects.

Ratio literals look like division operations:

```
1/2                 // a rational constant

1234/5678           // another rational constant

245850922/78256779  // pi (accurate to approximately 16 digits)
```

Since there is no whitespace around the / symbol, these expressions are ratio literal constants rather than function calls to the division operation.

Floating-Point Numbers

The `<float>` class of objects represent floating-point numbers. Apple Dylan includes the classes `<double-float>`, `<extended-float>`, and `<single-float>` for various levels of precision.

Floating-point literal constants include a decimal point; for example:

```
0.5         // a floating-point number
```

Additional Topics

There are other built-in number classes and many built-in functions for performing numerical calculations. See "Numbers" on page 61. ◆

Collection Classes

The **collection classes** allow you to create data objects that are collections of other data objects. Some examples are

- **strings,** which are ordered collections of characters
- **vectors,** which are ordered collections of data objects
- **pairs,** which are an ordered collection of two data objects
- **lists,** which are ordered collections of data objects

The next few sections discuss these classes of objects in more detail.

Additional Topics

There are many more collection classes than described here, and you can define your own. See "Collections" on page 61. ◆

Strings

String objects are ordered collections of characters. Apple Dylan provides many different classes of string objects, each slightly different, but the most common is the `<byte-string>` class, in which each character of the string fits in a byte of memory.

String literals are series of characters delimited by double quotes. For example:

```
"hello, world"
```

To include a double quote (") or a backslash (\) in a string literal, you must precede it with a backslash (\). For example:

```
"This string has a double quote \" and a backslash \\ in it."
```

These literal constants create string objects. Apple Dylan uses a special internal format for efficient string storage.

Additional Topics

String objects are not immediate—they use their object reference to reference the actual string data. See "Implementation" on page 60.

They are mutable, however—that is, once you've created a string object, you can make changes to that object. See "Mutability" on page 60. ◆

Vectors

Vector objects (of class `<vector>`) are ordered collections of other objects.

Vector literals use the `#[]` syntax to allow you to specify vectors of values. Commas separate the elements of the vector. Here is a simple example of a vector literal:

```
#[ 1, 2, 3 ]   // a vector with three integers as elements
```

Each elements of a vector can be an object of any class. For example, a single vector can contain a string object, an integer object, and a Boolean object:

```
#[ "hi", 4, #t ]   // a vector with a string, an integer, and a Boolean
```

You can also nest vectors within vectors:

```
#[ #[1, 2], #[3, 4] ]  // a vector with two vector elements
```

When you use a vector literal to specify a vector, the elements that you specify must be literal constants. If you want to create a vector using variable values or function results, you can use the built-in `vector` function, for example:

```
vector (*global*, max(3, 4), 10 + 10)
```

This function call creates and returns a vector with three elements: the first element is the value of the variable `*global*`, the second element is 4, and the third element is 20.

Vectors (like lists described in the next section) are ordered collections of data objects. However, vectors are implemented in a substantially different way than lists.

A vector is a random-access array of elements—that is, you can access any element in a vector as quickly as any other element in the vector. However, you to add or remove an element from a vector, you must copy the entire vector.

Lists

List objects (of type `<list>`) are also ordered collections of objects. Unlike vectors, however, list objects are implemented as linked lists. As a result, it takes more time to access elements later in the list than it does to access elements earlier in the list. However, you can add and remove elements from a list without copying the entire list.

List literals use the `#()` syntax to allow you to specify a linked list of values. Commas separate the elements of the list. Some examples of list literals are

```
#(100, 200, 300)    // a list with three numbers
#('A', 100)         // a list with a character and a number
#("hi")             // a list with one string
#(#(1, 2), #(3, 4)) // a list of lists
```

As with vector literals, list literals require you to specify elements as literal constants. If you want to create a list using variable values or function results, you can use the built-in `list` function, for example:

```
list (*global*, max(3, 4), 10 + 10)
```

This function call creates and returns a list with three elements: the first element is the value of the variable `*global*`, the second element is 4, and the third element is 20.

Apple Dylan represents lists internally as linked lists of objects. Every list object has a **head** and a **tail**. The head of a list is the first element in the list. The tail of the list is the rest of the list—that is, the list containing all but the first element of the original list.

Apple Dylan implements list objects efficiently using pair objects. as described in the next section.

An important list object is the empty list, which you can represent with this literal constant:

```
#()                 // the empty list
```

This corresponding object is a list object that has no elements.

Pairs

A pair object (of class `<pair>`) is a collection of two objects.

Pair literals use the `#()` syntax to allow you to specify a pair of values. A period, which must be surrounded by spaces, separates the elements of the pair. Some examples of pair literals are

```
#('A' . 'B')      // a pair with two characters
#(100 . 'B')      // a pair with an integer and a character
```

The most common use of pair objects is to create linked lists—the list objects described in the previous section.

Here is how linked lists are built from pair objects. The simplest list object is the empty list:

```
#()
```

Consider creating a pair that contains one object and the empty list:

```
#( 3 . #() )          // a pair: a 3 and an empty list
```

This pair includes the number 3 and an empty list. This pair is identical to (in fact, is the actual implementation of) this single-element list object:

```
#( 3 )                // a list with one element: a 3
```

Now, consider creating another pair object:

```
#( 2 . #( 3 . #() ))  // a pair: a 2 and the previous pair
```

This new pair object, which contains a pair object as its second element is equivalent to the list:

```
# ( 2, 3 )
```

One more time: let's create another pair:

```
#( 1 . #( 2 . #( 3 . #() )))  // many levels of pairs
```

This pair includes the number 1 as the first element and a pair as the second element. That pair includes the number 2 as the first element and a pair as its second element. That pair includes the number 3 as its first element and the empty list as its second element.

This pattern of objects and pairs is equivalent to the list

```
#( 1, 2, 3 )       // the same object in list notation
```

By building lists in this way, lists become binary trees; the left branch is always an object and the right branch is another pair, until you come to the end of the list, where the right branch is the empty list.

Apple Dylan provides a very efficient storage mechanism for pairs and lists; they are often useful in recursive programming styles such as those used in the LISP language.

Additional Topics

This chapter is a brief introduction to values, the objects that Apple Dylan uses to represent them, and the classes to which they belong. The following few sections provide more details for the more advanced reader, and provide pointers to more information.

Objects

Apple Dylan exhibits its true object-oriented nature by representing all information, from simple integers to compiled functions, as objects in a runtime environment.

- **Creating Objects**

  In this chapter, you saw how to create objects by evaluating literal constants, and by calling functions that returned one or more objects as return values. In the most general case, Apple Dylan provides the `make` function, which creates an object of a specified class. For example:

  ```
  make ( <vector> )
  ```

  This function call specifies a new vector object to be created. Since no initialization information is provided, the vector object returned by this function call is the empty vector `#[]`.

  Depending on the class of object you specify, the `make` function accepts various additional arguments that allow you to specify initial values for the new object.

  See Chapter 12, “Objects and Classes,” and Chapter 15, “Defining Classes,” for more information.

- **Disposing Objects**

  Apple Dylan provides **garbage collection,** an automatic memory management system that handles object disposal and memory reclamation for you. See the "Garbage Collection" chapter of the *Apple Dylan Extensions and FrameWork Reference* for more information.

- **Object Constructors**

  Apple Dylan provides special built-in functions that create and initialize certain built-in classes of objects. For example:

  ```
  vector (1, 2, 3);
  ```

  This function call creates and returns an object—a vector object with three elements (the integers 1, 2, and 3).

- **Variables**

  Objects can exist in runtime memory without having names. However, in your source code you must be able to create objects and refer those specific objects later. You do this by associating a name with an object. Named objects are called variables, and are discussed in the next chapter.

- **Mutability**

  An object is a value in memory. Some objects are **immutable**—once the object has been created, the value of that object cannot change. Boolean values are examples of immutable objects. If you specify the Boolean value `#f`, the object representing that value can never be changed. If you have a variable name bound to that object, the only way to change the value of the variable is to bind the variable name to another object.

  Characters and numbers are also immutable. Two variables may be bound to an object representing the number 100. You cannot change the value of that object so that both variable values change.

  Strings, vectors, and lists are mutable, however. If two variables are bound to the same string, changing the value of the string changes the value of both variables to the new string.

  See the next chapter for more details.

- **Implementation**

  Apple Dylan implements objects as blocks of memory called **object references.** Some small objects, such as characters and small integers are encoded entirely within their object reference. These types of objects are called **immediate objects**; they are always immutable.

Larger objects (such as strings and vectors) use their object reference to point to the memory location of the object. (This is not a machine pointer—Apple Dylan uses an internal representation for object references.)

Pair objects, interestingly enough, use a single object reference: the first half of the reference references the first part of the pair, and the last half references the second part of the pair. This optimizes the storage efficiency of linked lists.

Built-In Classes

A more thorough introduction to objects and classes is in Chapter 12, "Objects and Classes." For complete information, see the *Dylan Reference Manual*.

- **Simple Objects**

  The Boolean, character, and symbol classes are described in Chapter 12.

- **Numbers**

  The number classes, and the functions and operators you can use to manipulate them, are discussed in Chapter 13, "Numbers."

- **Collections**

  The discussion of collection classes is continued in Chapter 14, "Collection Classes." To create your own collection classes, see the *Dylan Reference Manual*.

- **Functions**

  Functions are also represented as objects in runtime memory—executable objects containing compiled code. Chapter 6, "Functions," introduces functions, but see the *Dylan Reference Manual* for complete information about the function-related classes and the operations you can perform on function objects.

- **Types**

  Classes themselves, such as `<integer>`. are also objects. For example, the `<integer>` class is an object that contains information about the structure and behavior of integer objects. As a result, you can manipulate classes just as you can manipulate other objects. In addition to classes, there are other kinds of types used by Apple Dylan. See Chapter 8, "Methods," for a discussion of the **singleton** type, and the *Dylan Reference Manual* for the complete discussion of types, classes, and type-related classes and functions.

User-Defined Classes

Even though Apple Dylan provides many useful classes built-in, creating your own classes is an important aspect of writing an Apple Dylan program.

- **Class Definitions**

  Chapter 12, "Objects and Classes," introduces class definitions, and Chapter 15, "Defining Classes," discusses how you can create your own classes.

- **Class Hierarchies**

  Reusing functionality—inheriting behavior from previously-written code—is one of the great strengths of object-oriented programming. See Chapter 16, "Inheritance" for an introduction to this powerful paradigm.

- **Creating Collection Classes**

  One example of reusing code is creating your own collection class, adding specialized behavior particular to your program's needs, but inheriting the majority of functionality from the classes provided by Apple Dylan. See the *Dylan Reference Manual* for details.

CHAPTER 5

Variables

Contents

Key Concepts 65
About Variables 65
  Bindings 66
  Scope and Extent 66
  Variable Names 68
  Variable Values 69
  Variable Specializers 70
Creating Variables 70
  Module Variable Definitions 70
  Constant Module Variables 71
  Other Module Variables 72
  Local Variable Declarations 72
Using Variables 73
  Referencing Variables 73
  Modifying Variables 75
Additional Topics 77
  Scope and Extent 77
  Classes and Functions 78

In the previous chapter, you saw some examples of constant values and how to specify literal constants in an Apple Dylan program.

Key Concepts

About Variables

- A **variable** represents a **binding** between a variable name and a data object
- Each variable name is accessible over a a particular has a **scope** and each variable's object exists at runtime for a particular **extent**
- There are rules for creating valid **variable names**.
- The values of variables are objects, which have varying degrees of mutability.
- **Specialized variables** may only have values of a specified type.

Creating Variables

- You can use **variable definitions** to create and initialize **module variables.**
- You can specialize module variables, or create **read-only variables** (also called **named constants**), which cannot be reassigned to a new object.
- You can use local variable declarations to create and initialize **local variables**

Using Variables

- You use **variable references** to determine the value of the variable.
- You can modify a variable using the **assignment** operator.

About Variables

In Apple Dylan, **variables** are values with names. The next few sections give an overview of Apple Dylan variables:

- how they are implemented

- when they are accessible
- what names are valid
- what values are valid

Bindings

Specifically, a variable is a **binding** between a value (an object at runtime) and a variable name (an expression in source code).

Unlike some dynamic languages, Apple Dylan does not store variable names at runtime. Variable names are inaccessible to the runtime environment. The development environment is responsible for compiling variable names into object references:

- Variable names exist in your source code; they are maintained by the development environment.
- When compiled, a variable name is replaced by a pointer to an object. (This pointer is not a machine pointer—Apple Dylan uses an internal representation.)
- The object pointed to contains the value of the variable.

In general, a variable name may be bound to any object; in fact, multiple variable names may be bound to the same object. Altering one such variable may or may not affect the others, depending on whether the object is mutable or immutable.

Similarly, the same variable name may be bound to two different objects—but only in different scopes, as explained in the next section.

Scope and Extent

The **scope** of a variable refers to the area of a program in which the variable can be referenced. For example, when you define a **module variable:**

```
define variable *global* = "always there";
```

you can reference the variable `*global*` anywhere within the module it is defined (unless shadowed by a local variable, as explained below). You can also

export the variable from its module and import it into other modules, as explained in Chapter 17, "Modules."

Another type of variable provided by Apple Dylan is the **local variable:**

```
begin
    let x = 1;
    let *global* = "shadowed";

    // block of code

end
```

Within the indicated block of code, the variable name `x` is bound to the integer object 1 and the variable name `*global*` is bound to the string object `"shadowed"`.

The `x` variable is local, the scope of this local variable is the begin-end body of code. References to the variable name `x` outside of this local scope are errors.

The variable name `*global*` is also declared as a local variable. When this name is referenced inside the body of code, it refers to the local value. Outside this body of code, references to this variable name refer to the module variable value again.

The **extent** of an object is how long the object exists in the runtime environment—which is as long as it may be accessible by some variable. Module variables contain objects with **indefinite extent**—their objects last until they are explicitly made inaccessible (for example, by redefining the variable to a new value). Local variables have a **definite extent**—they name objects when the local variable declaration is executed, but the name is no longer aid (that is, the object cannot be referenced by this name) when the surrounding body of code finishes.

A **runtime environment** is the complete set of bindings available when a constituent executes. Each module has its own basic set of module variables, and as code is executed, the environment changes as bindings are added, removed, shadowed, and modified.

Additional Topics

Apple Dylan also provides **closure variables,** which have a local scope and an indefinite extent. See "Closures" on page 78. ◆

Variable Names

Every variable has a **variable name.** A valid variable name is a string of characters that satisfies the following rules:

- A variable name may contain **alphabetic characters** (*a* – *z*). The capital alphabetic characters (*A* – *Z*) may also be used, but variable names are not case-sensitive.
- A variable name may contain **numeric characters** (*0* – *9*). However, if the variable name begins with a numeric character, it must contain two alphabetic characters in a row.
- A variable name may contain **graphic characters** (*! & * < = > | ^ $ % @ _*). However, if the variable name begins with a graphic character, it must contain at least one alphabetic character.
- A variable name may contain **special characters** (*- + ~ ? /*). However, a variable name may not begin with a special character.
- A variable name may not contain any other types of characters.
- A variable name may not match any word reserved by Apple Dylan.

Examples of valid variable names include

```
x
a-variable-name      // <= These two variable
A-Variable-Name      // <= names are the same
10%-&-up
who-knows-what-x=?
x=y?
!wow!
```

These are not valid variable names:

```
I-declare.     // Variable names may not include periods
100x200y       // Needs two alphabetic characters in a row
!@&2%-+=       // Needs at least one alphabetic character
?what?         // Starts with a special character
```

Although any object can have any valid variable name, Apple Dylan uses the following **naming conventions** to make programs easier to read:

- multiple words in a single name are separated by hyphens, such as the local variable name `range-size`
- constant names start with a `$`, as in `$max-random-value`
- module variables whose values are meant to change have names that are surrounded by asterisks, as in `*global*`
- class names are surrounded by angle brackets, as in `<integer>`
- function names that return a Boolean value end with a question mark, as in `positive?`
- function names that can destructively modify the values of their arguments end with an exclamation mark, as in `remove!`

Remember that these are simply naming conventions, and are not enforced by the Apple Dylan language.

Variable Values

The value of a variable is an Apple Dylan object—a variable name can be bound to an object of any class. Because Apple Dylan stores everything as an object during runtime, variables not only name typical data values:

- simple objects, like Booleans and characters
- numbers, like integers and fractions
- collections, like strings and lists

but the also name other classes of objects. For example:

- Classes are represented by objects. The variable name `<integer>` is bound to the built-in object that stores information about integers in general.
- Functions are also represented by objects. The variable name `max` is bound to the built-in object that contains the compiled code to find the greatest of its inputs.

See Chapter 12, "Objects and Classes," for more information, and the *Dylan Reference Manual* for the complete details.

Variable Specializers

You can use the double-colon (::) syntax to specify a **specialized** variable—variables that can only have values of a certain class. Here is the syntax:

variable-name :: *class-name*

and an example:

```
my-favorite-letter :: <character>
```

You specialize variables for a number of reasons. For example, the expression

```
define variable my-favorite-letter :: <character> = 'Q'
```

creates a specialized module variable named `my-favorite-letter`. The initial value for this variable is the object that represents the character *Q*. You may use the assignment operator, introduced in "Modifying Variables" on page 75, to change the value of this variable to a different character. However, you may not change its value to a different class of value, (such as an integer).

Additional Topics
In addition to type checking and clearer code, variable specializing can be used for compiler optimizations and, most importantly, to influence method dispatch for polymorphic functions. See "Specializers" on page 78. ◆

Creating Variables

This section examines some of the ways that you can create variables with Apple Dylan.

Module Variable Definitions

As you've seen a few times throughout the early chapters of this book, you create module variables using the `define variable` defining macro. The simple version of the syntax is

```
define variable variable-name = expression
```

The *variable-name* must be a valid variable name, and the *expression* is evaluated to provide the initial value for the module variable.

Here is an example:

```
define variable my-favorite-number = 37
```

When you execute this definition—by typing it into the Listener window, for example—it creates a module variable in the current runtime environment. That variable binds the variable name `my-favorite-number` to the integer data object with the value 37.

You can specialize a module variable as shown here:

```
define variable my-favorite-number :: <integer> = 37
```

You may subsequently change the value of this variable to another integer value, but attempts to change the value to a non-integer result in an error.

Additional Topics
Classes are organized into hierarchies; therefore you can specialize a variable to a related group of classes. See "Class Precedence" on page 79. ◆

Constant Module Variables

A specialized variable must always reference the same class of object; you can change the object, but not to a new class.

Apple Dylan also allows you to create module variables must always reference the same object. Called **named constants,** or **read-only variables,** these variables, once created, cannot be reassigned to a new object. You use the `define constant` defining form to create named constants.

As an example, the expression

```
define constant $my-favorite-letter-always = 'Q'
```

creates a named constant— a read-only module variable named `my-favorite-letter-always`. The value of this variable is the character object representing the character *Q*. This value may never be changed.

If the value of the named constant is mutable—that is, the object's value itself can be modified, then you can change the value of a named constant. For example:

```
define constant $some-vector = #[ 1, 2, 3 ]
```

Since vectors are mutable, you can change the value of this constant without changing the object it points to. See "Modifying Variables" on page 75.

Other Module Variables

Apple Dylan provides other kinds of definitions that allow you to create specific kinds of module variables. For example, a define class definition allows you to create a module variable: you supply the name and a class description. Apple Dylan creates a class object and binds it to that name.

As another example, a `define method` definition allows you to specify a variable name and an implementation (Apple Dylan source code). A module variable by that variable name is bound to an object that implements the function.

Additional Topics

Although they are simply module variables, classes and functions require that you specify extra information when you create them. See "Classes and Functions" on page 78. ◆

Local Variable Declarations

You create a local variables—always within a body of code—using a local variable declaration. For example:

```
let my-favorite-number-for-this-method = 11
```

You can also use **multiple values** to bind multiple local variables at once. Here is a simple example:

```
let (integer-part, remainder) = truncate (5/2)
```

The `truncate` function returns two values; this local variable declaration binds each value to the corresponding variable name. You can use the built-in values function to create multiple values when you need them:

```
let (first-choice, second-choice) = values ("Beef", "Chicken")
```

Additional Topics
Besides local variable declarations, perhaps the most common use of multiple values is returning multiple values from a function. See "Multiple Values" on page 78. ◆

Using Variables

The previous section showed how to create variables—module variables, named constants, local variables, and so on. This sections shows how you can examine and modify them after you've created them.

Referencing Variables

A variable reference is an expression that consists only of a variable name; evaluating a variable reference results in the value of the named variable.

Where a variable reference appears in your source code affects its value. In fact, a variable reference is a valid expression only if the reference occurs within the scope of a variable that has the specified variable name.

The **runtime environment** of a constituent contains the complete set of variables that are available to be referenced by the constituent.

If you attempt to use the Listener to reference a variable that has not been defined

```
> an-undefined-variable
```

Apple Dylan returns an error message.

The expression `an-undefined-variable` attempts to reference a variable. However, since there is no variable named `an-undefined-variable` available the runtime environment when this expression is evaluated, an error results.

If you've already defined a module variable in the current runtime environment, then evaluating a reference to the variable's name returns the value of the variable:

```
> my-favorite-number   // previously-defined variable
37

> my-favorite-letter   // also previously-defined
'Q'
```

Variable references frequently occur as parts of other expressions. For example, you can use a variable reference as an argument to a function call. Suppose you use the Listener to define these two variables:

```
> define variable x = 6
defined x

> define variable y = 9
defined y
```

As a result the module variables `x` and `y` are defined, initialized, and added to the current Listener environment. You can then use the Listener reference these variables:

```
> x
6

> y
9
```

You can also reference these variables in other expressions; for example:

```
> gcd (x, y)
3
```

When Apple Dylan evaluates this function call, it first evaluates the variable references (`x` and `y`), then it applies the `gcd` function to the resulting values (6 and 9), and finally it returns the resulting value (3).

You can also use variable references to initialize other variables:

```
> define variable twice-y = 2 * y
defined twice-y

> twice-y
18
```

In this example, once the variable `twice-y` is defined, it is in no way connected to the variable `y`—`y` is only evaluated when initializing `twice-y`. After that, they reference different objects.

Additional Topics
You can, however, define `twice-y` as a function whose value depends on `y`. See "Method Definitions" on page 78. ◆

Modifying Variables

In general, the **assignment operator** (`:=`) allows you

- to reassign an existing variable to a new object
- to modify an existing variable's object

If a variable's value is immutable, assigning that variable to a new value binds the variable name to a completely new object. (The reference to the old object is removed.)

For example:

```
> define variable *my-favorite-letter* = 'Q'
defined my-favorite-letter

> *my-favorite-letter*
'Q'
```

```
> *my-favorite-letter* := '?'
'?'

> *my-favorite-letter*
'?'
```

When this variable is assigned to the question mark character, the variable name `*my-favorite-letter*` is bound to a different object—no other variable that referenced the character object *Q* has suddenly started referencing a question mark!

Mutable objects, however, such as vectors, can be altered, and all references to the object (for example, all variables that reference the object) subsequently reference the altered object.

For example, imagine that you define the variables `x` and `y` to have vector values:

```
> define variable x = vector(1, 2, 3, 4)
defined x

> define variable y = x
defined y
```

As defined by Apple Dylan, the variables `x` and `y` not only have the same value—they reference the same vector in memory. If you change the value of this vector:

```
> x[0] := 100
100
```

you automatically change the value of both variables:

```
> x
#[100, 2, 3, 4]

> y
#[100, 2, 3, 4]
```

You can use the assignment operator to change the value of a variable from one class of value to another. For instance, the expression

```
x:= "A very different value, indeed."
```

not only changes the value of the variable `x`, it also changes the value from an vector to a string.

However, if `x` were specialized, as below, such an assignment would not be allowed:

```
> define variable x :: <vector> = #[1, 2, 3, 4]
#[1, 2, 3, 4]
```

If `x` were a constant module variable, no assignments would be allowed if its value was immutable:

```
> define constant x = 1
1
```

No assignment to `x` is allowed; the assignment operator cannot be used to change its value. However, if the constant's defined value is mutable:

```
> define constant x = #[1, 2, 3, 4]
#[1, 2, 3, 4]
```

You can still use the assignment operator to change the contents of the vector; you simply cannot reassign the variable `x` to another object altogether.

Additional Topics

In this chapter, you learned to bind variable names to objects, create variables with special restrictions, reference variable names to determine the values of variables, and reassign or modify a variable. More advanced and related information is described in the next few sections.

Scope and Extent

Scope and extent are of critical importance when creating and referencing variables.

- **Modules**

  The module is the basic namespace unit for Apple Dylan programs—it defines the scope of most definitions. You can use modules to hide and to share variables. See Chapter 18, “Modules,” for more information.

- **Local Declarations**

  Local variable declarations have a limited scope and extent. See Chapter 8, “Methods,” for more information.

- **Closures**

  Some Apple Dylan functions allow you to have local variables with local scope, but indefinite extent. These functions remember the values of their local variables from one call to the next. See Chapter 8, “Methods,” for details.

Classes and Functions

Apple Dylan provides many built-in functions; some are described in Chapter 13, “Numbers,” and Chapter 14, “Collection Classes.” Other groups include

- **Class Definitions**

  There are a number of ways to define classes, but the `define class` definition is the most common. See Chapter 12, “Objects and Classes,” and Chapter 15, “Defining Classes,” for details.

- **Method Definitions**

  Chapter 6, “Functions,” describes the various kinds of functions you can use with Apple Dylan. Chapter 8, “Methods,” discusses how to define specific function implementations.

- **Multiple Values**

  When creating a function that has multiple outputs, you can use the `values` function to return multiple return values. See Chapter 8, “Methods,” for details.

- **Specializers**

  One common use of variable specializers is to define methods that operate only arguments of specific classes. See Chapter 8, “Methods,” for details.

- **Method Dispatch**

  When a generic function with multiple methods is called, the generic function examines the arguments supplied, compares them to the specializers of each method 's parameters, and selects the method with the most appropriate specializers in its parameter list. See Chapter 17, "Generic Functions," for details.

- **Class Precedence**

  Classes in Apple Dylan are arranged in hierarchies; therefore, more than one method may be appropriate when a generic function is called. Apple Dylan uses class precedence to decide which method most specifically matches the arguments provided. See Chapters 16 and 17 for an introduction, but see the *Dylan Reference Manual* for the complete details.

CHAPTER 5

Variables

CHAPTER 6

Functions

Contents

Key Concepts 83
About Functions 84
  Methods 85
  Generic Functions 85
  Built-In Functions 86
Calling Functions 87
  Function Calls 87
  Arguments 88
  Evaluation 90
  Return Values 91
Creating Functions 92
  Creating Methods 92
  Formal Parameters 93
  Method Bodies 94
  Return Values 94
Additional Topics 94
  Built-in Functions 95
  Methods 95
  Generic Functions 97
  Calling Functions 98
  Function Objects 100

This chapter introduces Apple Dylan functions: what they are, which are built-in, how you call them, and how you define your own. This chapter provides a high-level overview of these subjects. Chapter 8, "Methods," and Chapter 17, "Generic Functions," provide much more detailed information.

Key Concepts

Functions

- **Functions** are units of compiled, executable code; typically, they accept **arguments** as input and produce **return values** as output.
- The two types of functions are **methods** and **generic functions.**
- A **method** is a basic function; it accepts parameters, provides a single implementation, and returns values.
- A **generic function** is a **polymorphic** function—that is, a function with different implementations for different types of inputs.
- Specifically, a generic function is a **method dispatcher.** Each generic function references a family of related methods; When invoked, the generic function examines the provided arguments and calls the most appropriate method.
- Apple Dylan provides a wide variety of **built-in functions** that provide a number of calculations for you. You can also create your own methods and generic functions.

Calling Functions

- A **function call** is an expression—a piece of Apple Dylan source code—that invokes the execution of a function.
- A function call typically includes the name of the function being called and a list of **argument expressions.**
- Evaluating a function call evaluates the argument expressions and sends the resulting values to the specified function.
- There are various syntaxes for different kinds of function calls: standard function calls, operator function calls, and special function calls.

Creating Functions

- There are numerous ways to create methods and generic functions.
- When defining a method, you specify a **parameter list,** which corresponds to the arguments sent to the method,
- You also specify any **local declarations,** and the **method body,** which determines the **return values** of the method.

About Functions

In Apple Dylan, **functions** are executable code that you can use to perform operations, compute values, create side effects in the runtime environment, and so on. Functions are compiled—they exist in runtime environments.

There are a number of **built-in functions** provided by Apple Dylan that provide a wide range of services for you.

A **function definition** is Apple Dylan source code you use when creating your own function. When a function definition is executed, Apple Dylan creates a corresponding function in the runtime.

To execute a function, you include a **function call** to that function in your source code. A **function call** is an expression—an executable piece of Apple Dylan source code. In a function call, you specify which function to execute, and you provide input values for the function. When a function call is executed, Apple Dylan sends the specified input values to the specified function, and returns the function return values, if there are any.

In general, there are two kinds of functions: methods and generic functions. A **method** is a simple function—it provides a single implementation. A generic function is polymorphic—it selects from different implementations depending on the input values.

Often when you are calling a function you cannot tell whether you are calling a method (a specific implementation) directly or whether you are calling a generic function (which selects an appropriate method for you).

Methods

Methods are analogous to the types of functions provided by most conventional programming languages. When you create a method, you typically specify

- information about the method's **parameters**—how many, what kind, and so on
- information about the values returned by the method
- the method **body**—the source code that specifies the implementation of the method

The section "Creating Functions" on page 92 discusses this process in more detail, and Chapter 8, "Methods," describes it at length.

Generic Functions

Generic functions are analogous to the polymorphic functions that are provided by some modern programming languages. A generic function is the mechanism Apple Dylan uses to implement **polymorphism**. A polymorphic function has multiple possible implementations; when the function is called, the most appropriate implementation is chosen based on the types of arguments provided in the function call.

Like methods, generic functions exist at runtime. However, a generic function does not contain its various implementations itself. Instead, a generic function contains references to a family of related methods. Each of these methods contains one possible implementation for the generic function.

The generic function, then, simply examines the arguments passed to it and selects the most appropriate method to apply to the arguments. This process of choosing the most appropriate method is called **method dispatch.**

Apple Dylan allows various amounts of **dynamism** with generic functions:

- You can extend some generic functions by adding new methods to them—giving them additional implementations that apply to specific kinds of arguments.
- Other generic functions cannot be extended—you can add no more methods to them.

- Some generic functions allow some dynamism, but restrict it in various ways.

Built-In Functions

Writing an Apple Dylan program, for the most part, involves defining your own functions. Many top-level constituents (such as the ones you store in source records) are function definitions that are compiled and downloaded into the runtime environment. "Creating Functions," beginning on page 92, introduces you to creating your own functions.

However, Apple Dylan includes a great number of built-in functions that you can use for a wide variety of purposes.

Some common groups of built-in functions are

- **numerical functions,** which include `max`, `min`, `abs`, `round`, and `truncate`.
- **numerical predicates,** which include `odd?`, `even?`, `positive?`, and `negative?`.
- **numerical operators,** which include `+`, `-`, `*`, `/`, and `^`.
- **comparison operators,** which include `=`, `==`, `~=`, `>`, and `<`.
- **collection constructors,** which include `list`, `pair`, `range`, and `vector`.
- **collection manipulators,** which include `reverse`, `sort`, `union`, and `add`.

Some built-in functions are not polymorphic. For example, the `head` function returns the first element of a list object. You cannot use this function with any other type of object.

Other built-in functions are polymorphic. For example, the `first` function also returns the first element of a list object. However, it also works for vectors, pairs, strings, and so on.

Similarly, some built-in functions are dynamic—you can add methods to them to augment or to specialize their behavior. Others are not—you may not add methods to them. While dynamic functions allow you more flexibility, closed functions allow for efficiency optimization.

The rest of this chapter provides more introductory material about functions. In particular, it discusses function calls and function definitions, and includes examples of built-in functions and functions you define yourself.

Additional Topics
There are many other kinds of built-in functions calls provided with Apple Dylan. See "Built-in Functions" on page 95 ◆

Calling Functions

Once a function exists (either a built-in function or one you create yourself), you execute its code by **calling** the function.

This process involves:

- the **function call,** in which you specify the function and the **arguments** to send to it as input.
- the execution of the function body
- the resulting **return values** returned by the function

The next few sections examine the process of executing functions in more detail.

Function Calls

In Chapter 3, "Constituent," you saw that a **function call** is a kind of expression—that is, a function call is Apple Dylan source code that is evaluated and produces return values.

In general, a function call specifies a function and any input values to send to the function. An example of a function call is

```
max (1, 2, 3, 4, 5)
```

This function call specifies the built-in `max` function and five values to send as input to the function. This function call uses **standard function call syntax:**

function `(` *argument-list* `)`

Any Apple Dylan function can be called using this syntax. However, other syntaxes are provided to make certain kinds of function calls more natural. An example is this function call:

```
60 + 40
```

In this example, the function is the addition (+) operator and the arguments are 60 and 40. This function call uses **operator call syntax:**

{ *argument* } *operator argument*

The first argument is optional for some operators. For example:

```
- 50
```

The function specified in this example is the built-in unary minus operator (-). The single argument is the number 50.

Additional Topics

There are two other kinds of function calls used for specific situations: element and slot references. See "Special Function Call Syntaxes" on page 99. ◆

Arguments

Arguments are the input values sent to a function when the function is called. In a function call, you typically specify these input values using **argument expressions.** An argument expression can be any Apple Dylan expression—the value of the argument expression, called the **actual argument,** is the value sent to the function.

To recap: argument expressions are Apple Dylan source code; they appear in function calls. Actual arguments are the values of the those expressions; they are objects sent to the function at runtime.

For example, consider this function call

```
max ( 10 , 5 + 6 )
```

The first argument expression is a literal constant: `10`. The second is an addition operation: `5 + 6`. When this function call is evaluated, the actual arguments are an integer object with the value 10 and an integer object with the value 11.

To allow maximum flexibility when using functions, Apple Dylan provides a number of different kinds of arguments:

- **Required arguments** must be supplied in the function call. As an example, the built-in function `negative` requires exactly one argument:

  ```
  negative(3)
  ```

  Required arguments are ordered—that is, you must supply them in the order the function expects. For example, the built-in pair function takes two arguments and uses them to create a pair:

  ```
  > pair ( 1 + 1, 'A' )
  #(2 . 'A')
  ```

  The order of the two required arguments is significant:

  ```
  > pair ( 'A', 1 + 1 )
  #('A', 2)
  ```

  In this example, changing the order of the arguments creates a different pair object.

- **Optional arguments** may be, but are not required to be, supplied in a function call. As an example, the built-in function `max` requires one argument, but can accept any number of optional arguments:

  ```
  max (1)

  max (1, 2)

  max (1, 2, abs(-3))
  ```

  The first argument in these three examples is required, but the subsequent arguments are optional. Optional arguments always follow required arguments, if there are any required arguments. Like required arguments, the order of optional arguments can be significant. As an example, consider the list function, which has no required arguments, but can accept any number of optional arguments:

  ```
  list (1, 2, 3, 4)

  list (3, 4, 2, 1)
  ```

  These two function calls have the same number of optional arguments that represent the same values, but the order of the arguments is significant and these two function calls create different list objects.

- **Keyword arguments** are optional, unordered arguments. In a function call, these arguments are specified by a keyword. For example, consider the built-in `sort` function, which you can use to sort the elements of list:

```
> sort ( #(4, 2, 3, 1, 5) )
#(1, 2, 3, 4, 5)
```

  This function normally sorts in ascending order. You can change that behavior by specifying a keyword argument:

```
> sort ( #(4, 2, 3, 1, 5), test: \> )
#(5, 4, 3, 2, 1)
```

  Here, the keyword `test:` indicates that an optional keyword argument is being provided. The argument expression is `\>`, which is the name of the built-in greater-than function (see the "Operator Names" information in "Additional Topics"). This optional argument specifies how the `sort` function should sort the list—in this case, in descending order.

Evaluation

Consider the standard function call syntax:

function `(` *argument-list* `)`

When evaluating a function call with this syntax,

1. Apple Dylan evaluates the *function* expression to determine the function being called. Typically, this expression is simply the function name.
2. Apple Dylan then evaluates the argument expressions in the *argument-list* in order from left to right to determine the actual argument values to pass to the function.
3. If the function begin called is a generic function, it examines the actual arguments, selects the most appropriate method, and sends the arguments to that method.
4. Apple Dylan executes the body of the resulting method.
5. Apple Dylan returns any resulting return values.

Additional Topics

The *function* expression does not have to be a function name—it can be any expression that evaluates to a function. See "Function Names" on page 100 and "Anonymous Methods" on page 96.

The order of argument evaluation is significant and can produce unexpected results. See "Argument Evaluation" on page 98.

Operator calls use a different syntax, and operator precedence becomes important during evaluation. See "Operator Precedence" on page 98. ◆

Return Values

Function calls, like other types of expressions, return values. In Apple Dylan, a function call may return any number of values—0, 1, 2, or more. This ability is called **multiple values.**

As an example, consider the built-in function `truncate`. This function requires one argument—any real number. It returns two values—the integer part of that real number and the remainder. Here is an example:

```
> truncate (5/2)
2
1/2
```

In this example, the input argument is the ratio 5/2; the function returns the value 2 (the integer part) and the value 1/2 (the remainder).

Multiple return values allow a single function to provide more than one result without using input/output parameters as some conventional languages do. However, it is most common for a function to return a single value.

Additional Topics

Multiple return values can be a powerful tool when using the functional style of programming. In Apple Dylan, you typically use them to bind multiple variables in parallel. See "Function Objects" on page 100. ◆

Creating Functions

A functions, as you've seen, is either a method or a generic function, which is basically a dispatcher for a family of methods. Either way, the method is basic unit of implementation for a function. To create functions, then, you must define their behavior by defining methods. The next few sections introduce the process of function creation by examining the various aspects of defining methods.

Additional Topics

Creating generic functions—functions that include multiple methods—involves different techniques. See "Generic Functions" on page 97.

All functions, whether methods or generic functions are represented during runtime as objects in the runtime environment. See "Function Objects" on page 100. ◆

Creating Methods

Not all method definitions require you to specify the same information, but some common elements are

- a function name
- a parameter-list specification
- a return value type specification
- a method body (an implementation)

As a simple example, consider this method definition:

```
define method increment ( original-number )
    original-number + 1;
end method
```

This method definition includes three important elements:

- The function name, which is `increment`.
- The parameter list, which consists of one variable name: `original-number`.
- The body, which consists of a single expression: `original-number + 1`.

The next few sections examine each aspect of a method definition in more detail.

Additional Topics

There are different kinds of method definitions, and although they all share certain attributes, they each have their own unique characteristics. See "Methods" on page 95.

As an example, functions are not required to have names. See "Anonymous Methods" on page 96. ◆

Formal Parameters

When you define a method you provide a **parameter-list specification,** which is a list of the formal parameters of the function. A **formal parameter** is a kind of local variable: it has a name and a value and can be referenced or modified only within the scope of the method.

What makes formal parameters different from other local variables is the manner in which they are assigned their initial value. When a function is called, the formal parameter variables are bound to the values of the corresponding actual arguments.

Remember:

- Formal parameters are a kind of variable; they are part of a function's definition; they have a name and a value; they exist in memory during the execution of a function.
- Argument expressions are expressions that are part of a function call; they are evaluated and the resulting values, or arguments, are passed to the function to be used as the initial values for the parameters.

Method Bodies

When you define a method, you provide a **method body,** which contains the source code that implements the method. Typically, method bodies contain:

- local variable declarations, which create variables that you can reference (and modify) from within the method body
- local method declarations, which create methods that you can call from within the method body
- expressions, which perform the operations and calculations that implement the method's functionality

When you call a method, the declarations and expressions in the method body are evaluated in order (subject to certain modifications, such as control statements and condition handling).

Return Values

The value of the last expression evaluated before a method finishes executing is the value returned as the result of the method call.

Methods are not required to return a result; also, they may return multiple values. When you define a method you can specifically limit the types of return values the method can return.

Additional Topics

The various aspects of method definitions—names, parameters, local declarations, bodies, and return values—are fully described in Chapter 8, "Methods." ◆

Additional Topics

This chapter provides a brief overview of the types and uses of functions in Apple Dylan. The following few sections provide some more details for the more experienced reader, and provide pointers to more information.

Built-in Functions

Apple Dylan provides many built-in functions; some are described in Chapter 13, "Numbers," and Chapter 14, "Collection Classes." Other groups include

- **Objects**

  Some functions allow you to create, initialize, and copy objects. See Chapter 12, "Objects and Classes," for details.

- **Comparing Objects**

  Apple Dylan includes operations that test objects for equality, as well as operations that compare them in other ways. Chapter 12, "Objects and Classes," and Chapter 13, "Numbers," discuss these operations.

- **Coercing Objects**

  The `as` function allows you to convert an object from one class to another. Chapter 12 introduces this function.

- **Manipulating Classes and Types**

  In Apple Dylan, classes (like `<integer>`) are represented by objects in runtime environments. As a result, there are built-in functions that allow you to manipulate classes themselves. Some of these functions are mentioned in Chapter 12, "Objects and Classes," but you'll find most of the relevant information in the *Dylan Reference Manual*.

- **Handling Conditions**

  For information about built-in condition-handling functions, see Chapter 20, "Conditions."

Methods

- **Generic Function Methods**

  Many methods belong to a generic function. These methods provide one possible implementation for the function, depending on the types of arguments sent to the generic function. You can create generic function methods using the `define method` defining form, or you can define a method in another way and add it to an existing generic function. See Chapter 17, "Generic Functions," for details.

- **Bare Methods**

A bare method is a method that you call directly, rather than through a generic function. One way to create a bare method is using the built-in `method` expression. For example:

```
define constant double = method (x)
                             2 * x
                         end method
```

This definition creates a method named double that multiplies its argument by 2. See Chapter 8, "Methods," for more information about bare methods.

- **Local Methods**

  A local method is a bare method that exists only within the scope of a limited body of code. For example:

```
if ( *keeping-score* )

     local method increment (x)
               x + 1;
          end increment;

     increment ( *my-score* );
     increment ( *your-score* );
end if
```

  This if statement defines a local method which increments a single argument. Within the body of the if statement, this local method is called twice. Outside the body of this if statement, this local method is not accessible.

  See Chapter 9, "Direct Methods," for more information about local methods.

- **Anonymous Methods**

  An anonymous method is a bare method that has no name. Typically, you define it and call it within the same expression. For example:

```
(method (x, y, z)  x + y + z  end) (1, 2, 3)
```

  This expression creates an anonymous method that adds three arguments and applies the unnamed method to the arguments 1, 2, and 3. The result is the value 6. Once this expression is evaluated, the anonymous method is no longer accessible, as it has no name.

  See Chapter 9, "Direct Methods," for more details.

- **Closures**

  A closure is a method that can maintain the values of its local variables—even between executions of the method. The local variables are not accessible outside the method body, but their values are stored until the method is called again. Chapter 9 also describes closures.

Generic Functions

- **Creating Generic Functions**

  There are a variety of ways to create a generic function. The `define method` defining form creates a method and adds it to a generic function. If no generic function of the specified name exists, one is automatically created.

  You can also create a generic function object using the `make` function on the class `<generic-function>`; you can subsequently add methods to it using the built-in function `add-method`.

  You can even create local generic functions by defining local methods of the same name within the same local method declaration.

  See Chapter 17, "Generic Functions," for details about creating generic functions.

- **Method Dispatch**

  Method dispatch is the process by which a generic function selects a method to execute based on the arguments provided in the function call.

  In general, method dispatch examines the classes of the arguments and chooses the most appropriate method to apply to those arguments.

  Which method is most appropriate depends on the parameter list specified in the method definition, which is discussed in Chapter 8, "Methods." It also depends on the class hierarchy, which is introduced in Chapter 16, "Inheritance."

  Some simple examples of method dispatch are provided in Chapter 17, "Generic Functions." The complete algorithm for method dispatch is complicated; you can find it in the *Dylan Reference Manual*.

- **Singletons**

  Apple Dylan allows generic functions to dispatch based on the classes of the argument provided. Another powerful feature allows a generic function to check for specific argument values and allow those specific values to determine which method is called.

For example, a generic function might have one method used when the first argument is an integer, but a different, more specific, method, if the first argument is the integer 0.

See Chapter 8, "Methods," and Chapter 17, "Generic Functions," for examples. See the *Dylan Reference Manual* for complete information.

- **Dynamism**

  When you create generic functions, you can control different aspects of their **dynamism**—how much the implementations can be manipulated. For example, you can create **sealed** generic functions, which do not allow more methods to be added, or **open** generic functions, which allow more methods to be added at runtime.

  You can also seal **branches** of a generic function, allowing certain methods to be overridden by more specialized methods, but disallowing other methods from being overridden. Chapter 17, "Generic Functions," discusses function dynamism. Again, the complete discussion is in the *Dylan Reference Manual*.

Calling Functions

- **Argument Evaluation**

  An important aspect of function call evaluation is that every argument expression is evaluated before the actual function is executed. (This is in contrast to macro calls, such as statements, which may follow different evaluation rules.)

  The order in which a function call's argument expressions are evaluated becomes particularly important if any of the arguments cause side effects that affect other arguments. For example, consider the function call:

```
max ( set-global-variable() , *global-variable* )
```

  If the `set-global-variable` function has a side effect of changing the value of the global variable named `*global-variable*`, the new value of that variable is the second actual argument to the `max` function.

- **Operator Precedence**

  Since operator calls use an infix notation, multiple operator calls can be strung together ambiguously:

```
3 + 4 * 5
```

This expression involves two operator calls. However, is the expression a call to the + operator with the arguments 3 and 4 * 5, or is it a call to the * operator with the arguments 3 + 4 and 5. In other words, does the addition or the multiplication happen first? The order in which operators are evaluated is called operator precedence.

Multiple calls to the same operator also have a precedence, called association. For example, consider this expression:

```
3 + 4 + 5
```

Addition associates left to right, so the 3 and the 4 are added first, and the result is added to 5.

Operator precedence and association are discussed in Chapter 13, "Numbers."

- **Special Function Call Syntaxes**

  In addition to the standard function call syntax and the operator call syntax, there are two other function call syntaxes intended for special kinds of function calls: the array reference and the slot reference. Here is an example of an array reference:

  ```
  my-array[1]
  ```

  This syntax is shorthand for the standard function call:

  ```
  element (my-array, 1)
  ```

  A slot reference looks like this:

  ```
  my-object.my-slot
  ```

  and is shorthand for the standard function call:

  ```
  my-slot(my-object)
  ```

  Chapter 14, "Collection Classes," describes array references and Chapter 15, "Defining Classes," describes slot references.

- **Return Values**

  As described in this chapter, a function can return multiple values. Typically, you use these values by binding them to local variables and then using the variables. For example:

  ```
  let (integer-part, remainder) = truncate (5/2)
  ```

This local variable declaration creates two local variables, one for each value returned by the `truncate` function. See Chapter 8, "Methods," for more information about local variables and multiple return values.

Function Objects

Apple Dylan represents functions as objects. Just as there are integer objects and string objects, there are function objects. In particular, there are method objects and there are generic function objects.

- **Method Objects**

  A method object contains compiled information about a method: its parameter list, its body including local declarations, and information about its return values.

  Method objects contain executable code; they can be called directly, or called indirectly through a generic function object.

  There are a number of ways to create method objects and to define their behavior. See Chapter 8, "Methods," for more information.

- **Generic Function Objects**

  A generic function object contains a set of references to method objects, and also contains the compiled code that examines arguments and dispatches them to the appropriate method. There are a number of ways to create and manipulate generic functions. Some of these functions are discussed in Chapter 17, "Generic Functions," but you can find complete information in the *Dylan Reference Manual*.

- **Function Names**

  Methods and generic functions are compiled into runtime objects. Like any objects, they can be bound to variable names.

  For example, the built-in `max` function is actually a variable: the variable name is `max` and the value of the variable is the method object that finds the maximum value of its arguments.

  A function name, then, is simply a variable name that evaluates to a function object. For example:

```
> max
#<the method max (<object>, #rest) at: #x2451B4A>
```

Evaluating the function name `max` returns the method object for the `max` function (which is actually compiled code, but the Listener prints it out the best it can).

Since functions are implemented as objects, you can manipulate them in a variety of ways. For example:

```
define constant biggest = max
```

This definition binds the variable name `biggest` to the `max` method object. Therefore, you can now use `biggest` in place of `max`:

```
> biggest (1, 2, 3, 4, 5)
5
```

When this function call is evaluated, the variable `biggest` is evaluated first, resulting in the built-in method object also called `max`. Then, the argument expressions are evaluated and sent to that method object, which returns the maximum value of the arguments.

- **Operator Names**

  Operators also represent function objects. For example the + operator is really a variable: the name of the variable is `\+` and the value of the variable is the generic function object that adds its two arguments. Therefore, the special operator call syntax:

  ```
  3 + 4
  ```

  is really shorthand for the function call:

  ```
  \+ (3, 4)
  ```

  When making an operator call, you use the `+` symbol. If you want to refer to the addition generic function object, you use its variable name, which is `\+` (the backslash is added to resolve potential ambiguities when parsing operator calls).

- **Reflexive Functions**

  Since functions are objects, they can be operated on like other values. In fact, Apple Dylan provides a number of built-in functions that operate on functions themselves. For example, the `apply` function takes a function as its first argument and a sequence as its second argument. It calls the function, sending as arguments the elements of the sequence. For example:

  ```
  > apply ( max, #(1, 2, 3) )
  3
  ```

This function call to the `apply` function sends the three values 1, 2, and 3 to the function named `max`, and returns the value returned by that function.

You can also use the `apply` function to apply operators. For example, the expression

```
apply (\+, #(1, 2))
```

applies the + operator (the function named `\+`) to the arguments `1` and `2`.

Another function that operates on functions is the `compose` function. This function creates a new function by composing the functionality of other functions. For example:

```
define constant negative-add = compose ( negative, \+ )
```

This definition creates a function named `negative-add` that combines the built-in `negative` function with the built-in addition operator. As a result:

```
> negative-add ( 10, 20 )
-30
```

See the *Dylan Reference Manual* for more information about these and similar functions.

CHAPTER 7

Statements

Contents

Key Concepts 105
About Statements 106
  Statement Macros and Evaluations 106
  Test Expressions 107
  Bodies of Code 108
  Begin-End Statements 108
  Local Declarations 109
  Statements as Expressions 110
Conditional Statements 110
Iterative Statements 112
Additional Topics 113
  Macros 113
  Statements 114

This chapter provides a brief overview of the fourth kind of expression: the statement. It provides some introduction to statements, bodies of code, and conditional and iterative execution. Later chapters 9 and 10 examine the individual statement types in more detail

Key Concepts

About Statements

- **Statements** are expressions which, when evaluated, provide control of the flow of your program.
- Statements are actually **macro calls** to **statement macros.** Apple Dylan provides built-in statement macros for you to use. You can create your own kinds of statements by defining statement macros.
- Statements, in general, include **test expressions** and **bodies of code.** The test expressions determine whether the bodies of code are executed.
- In general, test expressions are considered to be false if they evaluate to the Boolean value `#f`. *All other values* are considered to be true—including some surprising values like 0, -1, the empty string `""`, and the empty list `#()`.
- The **begin-end statement** creates and **explicit body** of code. Most statements use **implicit** bodies of code—that is, you do not need to include the `begin` and `end` keywords.

Conditional Statements

- **If statements** allow you to execute a body of code if a test expression is not *false*.
- **Unless statements** execute a body of code only if a test expression evaluates to *false*.
- **Case statements** contain a series of test expressions and corresponding bodies of code.
- **Select statements** contain a test expression, called a **target expression,** and a a series of different bodies of code to execute depending on the value of the target expression.

Iterative Statements

- **While statements** allow you to execute a body of code repeatedly while a test expression is not *false*.
- **Until statements** execute a body of code repeatedly until a test expression is not *false*.
- **For statements** iterate a body of code while maintaining one or more **iteration variables.**

About Statements

Statements are another kind of Apple Dylan expression. Typically, you use statements to

- group constituents into a body of code
- provide exception-handling for a body of code
- execute a body of code conditionally
- iteratively execute a body of code

In general, the built-in statements allow you to modify the flow of control of your Apple Dylan program.

Statement Macros and Evaluations

Statements are similar to function calls—they are expressions, they have input, they are evaluated, and they return values. In fact, a statement is actually to a call to a **statement macro** (whereas a function call is a call to a function object).

Macro calls differ from function calls in a few important ways:

- The development environment provides a preliminary translation of macro calls, typically into Apple Dylan source code, which is then compiled.
- Therefore, each type of statement macro can have its own syntax. Typically,. a call to one of the built-in statement macros contain one or more **test expressions** and one or more **bodies of code.** (As compared to the list of argument expressions found in a function call.)

- Statements do not necessarily evaluate all of their inputs—that is, they may contain expressions and bodies of code that are not evaluated even when the statement is executed. By contrast, function calls evaluate every argument expression.

The statements that this chapter covers are all examples of **macro calls** to **built-in statement macros.**

Additional Topics

There are other kinds of macros besides statement macros, including defining macros and function macros.

You can extend the Apple Dylan language by defining your own macros. See Chapter 19, "Macros." ◆

Test Expressions

Many of the built-in statement macros require you to provide an expression. When the statement is executed, your test expression is evaluated in the current runtime environment. Typically,:

- a body of code is executed if the return value is anything except *false* (`#f`)
- the body of code is not executed if the return value is *false*

Each kind of statement varies slightly—some have multiple test expressions and multiple bodies of code, some continue to test the test expression until it returns *false*, and so on.

NOTE

The built-in statements consider *any value* that is not *false* to be *true*. Only the object *false* is considered to be false. All other values—including `0`, `-1`, the empty list, the string `"false"`, the symbol `#"false"`, and so on—are all considered to be *true*. ◆

As a result, you don't need to use a predicate function (like odd? or the == operator) in your test expressions.

Bodies of Code

A **body** of code is a series of constituents, separated by semicolons. Usually, any number of constituents is allowed—including zero. Also, the final constituent may have a semicolon after it, but it is usually neither required nor forbidden.

Bodies of code in statements contain local declarations and expressions. (Definitions, which apply to entire modules, are usually not allowed to appear within a specific body of code).

Most of the built-in statement macros include **implicit bodies** of code, which is simply a body of code that is not required to be surrounded by the words `begin` and `end`.

Here is an example body of code:

```
let urgent = head ( *things-to-do* );
complete ( urgent );
write-report-about ( urgent );
tail ( *things-to-do* );
```

This example creates a local variable, initializes it from the first element in a global list, sends the value to two functions, and ends with an expression that determines the rest of the list.

The value returned by a body of code is the value of the last expression evaluated in the body., so the value returned by this body of code is the `tail` of the `*things-to-do*` list. (If a body is empty, it returns the value `#f`.)

Explicit bodies of code are actually statements themselves: simple calls to the begin-end statement macro, described in “Begin-End Statements” on page 108.

Begin-End Statements

The **begin-end** statement allows you to gather constituents (local declarations and expressions) into a single body of code. Here is an example:

```
begin
    let sample = #(1, 2, 3, 4, 5);
    tail (sample);
end
```

This begin-end statement creates a body of code with two constituents: a local variable declaration and a function call. These constituents are separated by a semicolon and an optional semicolon follows the last constituent.

When a begin-end statement is evaluated, Apple Dylan executes the body of code. The value returned by the begin-end statement is the value returned by the last constituent evaluated in the statement. For example:

```
> begin
    let sample = #(1, 2, 3, 4, 5);
    tail (sample);
  end
#(2, 3, 4, 5)
```

This explicit body of code returns the value of the last expression evaluated.

Additional Topics
Block statements also create bodies of code, but they allow you to provide for exception-handling and non-local exits.
See "Block Statements" on page 114. ◆

Local Declarations

Within a body of code you can include local declarations and expressions. The **scope** of a local declarations, such as a local variable or local method, begins when the local declaration and ends at the end of the smallest enclosing body of code. For example, consider:

```
begin
    let mess = #(3, 2, 4, 1);
    begin
         let tidy = sort ( mess );
    end;
    tidy;     // this is an error
end
```

In this example, the local variable `tidy` falls out of scope immediately—at the `end` of the smallest enclosing body of code. The variable reference can be moved back into the inner body of code:

```
begin
    let mess = #(3, 2, 4, 1);
    begin
         let tidy = sort ( mess );
         tidy;
    end;
end;
```

This body of code returns `#(1, 2, 3, 4)`: the variable reference `tidy`, which is in scope, is the last expression evaluated.

Statements as Expressions

Since statements are expressions, you can use them wherever you would use an expressions. For example, consider:

```
concatenate( begin
                let x = #(2, 3, 1);
                sort (x);
             end,
             begin
                let x = #(6, 5, 4);
                sort (x);
             end )
```

This function call to concatenate has two argument expressions: both are begin-end bodies of code. The return value is `#(1, 2, 3, 4, 5, 6)`.

Conditional Statements

Conditional statements allow you to execute a body of code, or multiple bodies of code, as determined by the value of a test expression. There are four types of conditional statements

An **if statement** allows you to execute a body of code if a test expression is not *false*. Here is an example:

```
if ( hours ~= 0 )
     let (mph, remainder) = truncate/ (miles, hours);
     mph;
end if;
```

This simple if statement has an implicit body that contains a local variable declaration that binds two variables and a variable reference. The body of code is only executed if the variable hours has a value which is not 0.

The **unless statement** performs the opposite function:

```
unless ( hours = 0 )
     let (mph, remainder) = truncate/ (miles, hours);
     mph;
end if;
```

A **case statement** evaluates a series of test expressions until one returns a value that is not *false*, and then executes a corresponding body of code.

Here is an example of a simple case statement:

```
case
    hours < 10 => "speed racer";
    hours > 10 => "sunday driver";
end case;
```

This case statement contains to simple test expressions and to simple bodies of code (each code body consists of one string literal). When this statement is executed, it returns `"speed racer"` if the value of the variable `hours` is less than 10, and `"sunday driver"` if it is more than 10.

A **select statement** is similar to a case statement, except that it selects a body of code to execute based on the value of a single expression.

For example:

```
select ( day )
    1 => "Monday";
    2 => "Tuesday";
```

```
    3 => "Wednesday";
    4 => "Thursday";
    5 => "Friday";
    6 => "Saturday";
    7 => "Sunday";
end select
```

This select statement evaluates the variable reference day and returns an appropriate string.

Additional Topics
All of these built-in statements provide numerous additional options. See "Conditional Statements" on page 114. ◆

Iterative Statements

Iterative statements allow you to execute a body of code multiple times until some condition is met. There are four types of iterative statements

An **while statement** executes a body of code repeatedly until a test expression is *false*. Here is an example:

```
while ( *savings* > 0 )
    *savings* := tiny-withdrawal(*savings*);
end while;
```

In this while statement, the test expression is evaluated, and if it is not *false*, the body of code is executed and process repeats until the test expression is *false*. A while statement always returns *false*.

An **until statement** executes a body of code repeatedly until a test expression is no longer *false*. Here is an example:

```
until ( *debt* >= 0 )
    *debt* := make-minimum-payment(*debt*);
end until;
```

In this until statement, the test expression is evaluated, and if it is *false*, the body of code is executed and process repeats until the test expression is not *false*. An until statement always returns *false*.

A **for statement** executes a body of code repeatedly while maintaining one or more iteration variables. Here is a simple example; the iteration variable is `count`:

```
for (count from 1 to *input*)
    *factorial* := *factorial* * count;
end for
```

The for statement is quite powerful; it has many options and is particularly useful with collection classes. Unlike the other iterative statements, you can explicitly specify that a for statement return a value. Otherwise, it, too, returns the value *false*.

Additional Topics

Many more options exist for these powerful iterative statement macros. See "Iterative Statements" on page 114 ◆

Additional Topics

This chapter provides most of the information you need to use conditional statements. A few more facts and pointers are included here.

Macros

- **Statement Macros**

  Statements are macro calls—macro calls to statement macros. You can find out more about macros, macro calls, statement macros, and defining your own macros in Chapter 19, "Macros."

Statements

- **Block Statements**

  A block statement is a body of code that allows you to handle error conditions. See Chapter 20, “Conditions,” for details.

- **Conditional Statements**

  Conditional statements allow you to execute bodies of code when certain conditions are met. See Chapter 10, “Conditionals.”

- **Iterative Statements**

  Iterative statements are similar to conditional statements, except that they allow you to execute bodies of code multiple times while certain conditions are met. See Chapter 11, “Iterators.

CHAPTER 8

Methods

Contents

Key Concepts 117
About Methods 118
Method Definitions 119
Parameter Lists 120
    Parameter Type Specialization 123
Method Bodies 124
  Local Variable Declarations 125
Return Values 126
Additional Topics 129
  Methods and Generic Functions 130
  Parameters 130

Chapter 6 introduced you to the Apple Dylan approach to functions. This chapter examines the basic functional unit—the method—more closely and provides the information you need to start defining your own methods.

Key Concepts

About Methods

- A **method** is represented in memory at runtime as a method object.
- You can invoke methods directly, or through a generic function dispatcher.

Method Definitions

- Apple Dylan provides multiple ways to create method objects.
- When defining a method, you typically specify a **function name,** a **parameter list,** a **method body.**

Parameter Lists

- In the formal parameter list, you specify the **required, rest,** and **keyword** parameters that correspond to arguments accepted by the method.
- A method can **specialize** its formal parameters—that is, specify which classes of objects it accepts as arguments.

Method Bodies

- A **method body** consists of **local declarations** and **expressions.**
- A local variable declaration creates a local variable; it can create multiple variables with multiple values.

Return Values

- A method body has no explicit return statement; the method returns the value returned by the last expression evaluated during the execution of the method.
- You can specify the number and classes of a method's return values.

About Methods

Methods are the basic units of compiled Apple Dylan functionality. A method, like a function in conventional programming languages, represents a specific implementation that may accept input values, perform calculations, cause side effects, and return result values.

There are a number of ways you can define a method in your Apple Dylan program; the result is a **method object**—an executable object that exists in the runtime environment.

Whereas a method is an individual implementation of a function suitable for specific types of arguments, a generic function defines a function protocol, and manages a group of methods to provide the actual implementation. When a generic function is called, it examines the actual arguments provided, determines the most suitable method, and applies that method to the arguments provided.

Polymorphism is a powerful aspect of Apple Dylan, allowing for many of its object-oriented features as well. As a result, many of the methods you create will belong to a particular generic function. You won't call the method directly, instead you'll call the generic function and it will dispatch the job appropriately.

However, you are not required to call methods through the generic function dispatching mechanism. In many cases, you might want to call a **direct method**. This chapter focuses on the information you'll need when defining direct methods or methods for generic functions.

Additional Topics

There are powerful programming techniques available in Apple Dylan; both in the functional approach of direct methods and the polymorphic/object-oriented approach of generic functions. See "Methods and Generic Functions" on page 130. ◆

Method Definitions

There are a number of different ways to define methods, just as there are a number of kinds of methods. For example, you can:

- use the `define method` defining form to create a method object and add it to a generic function (creating the generic function if necessary)
- use the `local` method declaration form to create local methods
- use the built-in `method` expression to create a bare or anonymous method

All of these method definitions require you to specify approximately the same information:

- a function name (if not anonymous)
- a parameter-list specification
- a return value type specification (if desired)
- a method body (an implementation)

For example, let's look at two method definitions. The first is

```
define method increment ( original-number )
    original-number + 1;
end method
```

And the second is

```
define constant increment = method ( original-number )
                                original-number + 1;
                            end;
```

Both of these definitions create a method. The first one uses the `define method` defining form. It creates a method object that takes a single argument, increments it, and returns the result.

The second definition creates the same method object using the built-in `method` expression.

The difference is that the first definition added the new method object to a generic function named `increment`. More implementations (maybe one for characters? or strings?) could be added to this generic function.

The second definition created a bare method. In this case, the method itself is bound to the variable name `increment`. This method would not be connected to a generic function, and you could not add any more implementations.

Despite the differences, these method definitions have much in common:

- A **function name,** which is `increment`. In the first example, this is the name of a generic function, in the second case, it's the name of the bare method.
- A **parameter list,** which consists of one variable name: `original-number`. This list allows you to specify the names, and optionally to specify the types, of the formal parameters to a method. When the method is called, the values of the actual arguments are bound to these parameter names, which you can then use as local variables in the method body.
- A **method body,** which consists of a single expression: `original-number + 1`. In general, a method body is a series of local declarations and expressions, which are executed when the method is called. The constituents of a method body are separated by semicolons, with an optional semicolon following the final constituent. The method body may be empty—that is, contain no constituents.

The next few sections introduce the aspects of the parameter list and the method body in more detail.

Parameter Lists

When you define a method you provide a **parameter-list specification,** which is a list of the formal parameters of the function. A **formal parameter** is a kind of local variable: it has a name and a value and can be referenced or modified only within the scope of the method.

What makes formal parameters different from other local variables is the manner in which they are assigned their initial value. When a function is called, the formal parameter variables are bound to the values of the corresponding actual arguments.

For example, consider this method definition:

```
define method add-2 (x, y)
    x + y;
end method;
```

This method definition includes a parameter list that specifies two variable names: `x` and `y`.

You can call this method using this function call:

```
add-2 (3, 4);
```

This function call specifies two arguments—3 and 4. During the execution of the method body, the values of these arguments are bound to the local variable names `x` and `y`: the name `x` is bound to the value 3 and the name `y` is bound to the value 4.

As a result, the function call returns the value 7.

Apple Dylan provides four basic types of parameters:

- **required parameters,** which correspond to arguments that must be supplied when a function is called
- **rest parameters,** which allow a function to accept an indefinite number of arguments when it is called
- **keyword parameters,** which correspond to optional arguments that may be specified in any order when the function is called
- **next-method parameters,** which are discussed in Chapter 16, "Generic Functions"

A **required parameter** is a parameter that corresponds to an argument that must be supplied when a function is called. Consider this method definition:

```
define method takes-two-arguments (first, second)
    do-something (first);
    do-something (second);
end method
```

This function has two required parameters, named `first` and `second`. When you call this function, you supply two arguments:

```
takes-two-arguments ("hello", "world");
```

The order of the parameters is significant: the first parameter in the formal parameter list is bound to the value of the first argument in the function call, the second parameter to the second argument, and so on.

A method may have no required parameters. If a method does have required parameters, however, you must specify them in the parameter list before specifying any other types of parameters.

Rest parameters allow you to create methods that take an indefinite number of arguments. When defining the method, you provide the name of the rest parameter; when you call the method, the required parameters are bound to their corresponding argument values, and the rest parameter is bound to a sequence containing all the rest of the arguments (in the order they appear in the argument list).

For example, consider this method definition:

```
define method sort-all-arguments (#rest all-arguments)
    sort (all-arguments)
end method;
```

This method has no required parameters; the word `#rest` indicates that `all-arguments` is a rest parameter. When you call the `sort-all-arguments` method, all of the arguments you provide are collected into a sequence, and the parameter name `all-arguments` is bound to that sequence. For example, consider this function call:

```
sort-all-arguments (1, 4, 2, 3, 5);
```

When this method call is evaluated, the `all-arguments` parameter is bound to a sequence containing the numbers 1, 4, 2, 3, and 5. The body of the method then applies the `sort` function to this sequence.

You may define a method that has both required parameters and a rest parameter.

Keyword parameters correspond to optional arguments that may be specified in any order when a function is called. As an example, consider the method definition:

```
define method make-fraction (#key the-numerator, the-denominator)
    the-numerator / the-denominator;
end method make-fraction;
```

The word `#key` indicates that the `the-numerator` and `the-denominator` parameters are keyword parameters. When you call a method with keyword parameters, you use the argument list to specify both the parameter name and the argument value:

```
make-fraction (the-numerator: 5, the-denominator: 7);
```

Since you specify both the name and the value, you can specify keyword parameters in any order. For example,

```
make-fraction (the-denominator: 7, the-numerator: 5);
```

Additional Topics
There are many combinations of required, rest, and keyword parameters that provide special functionality, but that require you to follow specific rules. See "Formal Parameter Lists" on page 130. ◆

Parameter Type Specialization

Consider the following method definitions:

```
define constant combine = method (a, b)
                             pair (a, b);
                          end method;
```

This bare method definition accepts any two objects as its arguments. If you wanted to include type checking—for example, to allow only pairs that created lists, you could use variable specialization, as introduced in Chapter 5, "Variables," to create a more specific method:

```
define constant combine = method (a, b :: <list>)
                             pair (a, b);
                          end method;
```

Now the method accepts any object as the first argument, but only objects of class `<list>` as the second. The result will always be another list.

Additional Topics

Parameter type specialization is used most frequently when adding methods to a generic function—by specifying parameter types, each method can provide an implementation for a certain set of argument types.

Parameter type specialization allows certain parameters to accept only argument values of a particular class.Another powerful feature of Apple Dylan, **singleton parameters,** allows you to specify that certain parameters accept only one particular argument value. This allows for very specialized implementations with generic functions. See "Singletons" on page 130 ◆

Method Bodies

When you define a method, you provide a **method body,** which contains the source code that implements the method. Typically, method bodies contain:

- local variable declarations, which create variables that you can reference (and modify) from within the method body
- local method declarations, which create methods that you can call from within the method body
- expressions, which perform the operations and calculations that implement the method's functionality

The declarations and expressions are separated by semicolons, with an optional semicolon following the final declaration or expression. The method body may be empty—that is, contain no declarations or expressions.

When you call a method, the declarations and expressions in the method body are evaluated in order (subject to certain modifications, such as statements and condition handling).

The next two sections, "Local Variable Declarations," and "Return Values,"

Local Variable Declarations

A **local declaration** is an executable piece of Apple Dylan source code that applies to a limited scope of a program. For example, a local declaration can create a binding between a name and a value within the body of a method or within the body of a statement.

There are three kinds of local declarations:

- **Local method declarations,** which allow you to bind a variable name to a method. Local methods are described in the next chapter.
- **Local handler declarations,** which allow you to specify a condition handler. Condition handlers, and error handling in general, are discussed in Chapter 20, "Conditions."
- **Local variable declarations,** which allow you to bind a variable name to a specified value.

You declare local variables using the `let` declaration form. In its simplest form, a local variable declaration has this syntax:

`let` *variable-name* `=` *initial-value*`;`

For example,

```
let x = 1;
let y = #(1, 2, 3);
let $pi = calculate-pi(significant-digits: 10);
```

You can also specify the class of a local variable:

```
let my-name :: <string> = "Jane";
```

In this example, the `my-name` local variable is restricted to string values; you cannot assign other types of values to this variable.

You can declare multiple local variables with a single declaration. For instance:

```
let (x, y, z) = values (1, 2, 3);
```

This declaration creates three local variables, named `x`, `y`, and `z`. The value of `x` is initialized to 1, the value of `y` to 2, and the value of `z` to 3.

As another example, consider this local declaration:

```
let (integer-part, decimal-part) = truncate (5/2);
```

The built-in `truncate` function returns two values: the integer part of its argument and the decimal part of its argument. In this example, the local variable `integer-part` is initialized to 2and the local variable `decimal-part` is initialized to 1/2.

You can also use a local variable declaration to bind an unspecified number of values. For example:

```
let (#rest both-parts) = truncate (5/2);
```

In this example, Apple Dylan collects the values returned by the `truncate` function into a sequence and binds the variable name `both-parts` to that sequence. Therefore, the value of `both-parts` after this declaration is the list `#(5, 1/2)`, or an equivalent sequence of a different type.

When you declare multiple local variables in a single variable declaration, the initial bindings are performed in parallel. For example, consider these three local declarations:

```
let x = 1;                  // declare and initialize x
let y = 2;                  // declare and initialize y
let (x, y) = values (y, x); // switch their values
```

In this example, the third declaration binds `x` to the previous value of `y` and binds `y` to the previous value of `x`. Since these bindings are performed in parallel, the values of the variables are effectively switched in a single line of code!

Return Values

The **return values** of a function are the values returned by a function after it is executed in response to a function call.

Apple Dylan functions return whatever values are returned by the last expression evaluated when the function is executed.

▲ **WARNING**
Apple Dylan does not have an explicit return statement—the values returned by a function are simply the last values calculated by that function. The lack of an explicit return statement is sometimes confusing to beginning Apple Dylan programmers. ◆

Here is an example:

```
define method greater-of-the-two (a, b)
    if (a > b)
        a
    else
        b
    end if
end method;
```

This method has two parameters, named `a` and `b`. The body of the method tests if the value of `a` is greater than the value of `b`. If so, it returns the value of `a`. If not, it returns the value of `b`.

Notice that no return statement is needed. If the `a` parameter has the greater value, then the variable reference to `a` is the last expression evaluated; its value is returned as the function result. If the `b` parameter has the greater value, then the variable reference to `b` is the last expression evaluated and its value is returned as the function result

If you want a function to return multiple values, you can use the built-in `values` function. For example:

```
define method sum-and-difference (x, y)
    values(x + y, x - y);
end method;
```

This method returns two values. For example, consider this method call:

```
sum-and-difference(100, 25);
```

The result of this call is two values: 125 and 75.

Apple Dylan allows you to explicitly specify the class of a function's return value, which is useful for type-checking your program and for compiler optimizations. It is also useful for documenting your code.

Here is an example of a return value type specification:

```
define method long-name? (name :: <string>)
    => result :: <Boolean>;

    size(name) > 20;

end method;
```

This method, `long-name?`, takes a single string as a parameter and returns a Boolean value, which indicates whether the string has more than 20 characters. The `=>` punctuation indicates a specified return value type; it always follows the parameter list.

Two important facts to remember about return value type specifications:

- *The name of the return value is ignored.* You provide it simply to make your code easier to understand. In the above example, the name of the return value is `result`. This name is arbitrary and cannot be manipulated in the body of the method (that is, you can neither reference it nor assign to it).
- You may provide a name for the return value without specifying the class of the return value. This feature is for program documentation only; the return value can be of any class.

▲ **WARNING**
A common mistake for beginning Dylan programmers is to leave out the return value name (since it is ignored), and only include the class specification. For example:

```
define method long-name? (name :: <string>)
    => <Boolean>;
```

This mistake leads to a surprising result. It does *not* specify that the *class* of the return value is `<Boolean>`. Instead, it specifies that the *name* of the return value is `<Boolean>`. Therefore, this specification effectively does nothing. To specify the class of the return value, you must use the `::` notation. ◆

If your method returns multiple values, you can specify their classes using this notation:

```
define method name-and-age (employee-number :: <integer>)
    => (name :: <string>, age :: <integer>)
```

This method returns a string and an integer. The names `name` and `age` are ignored; however, specifying these names is useful for remembering the order in which the values are returned.

If your method returns an unspecified number of values, you can use the `#rest` notation to specify their class:

```
define method ages (name-list :: <list>)
    => ( #rest ages :: <integer> )
```

This method returns an unspecified number of integers. The name `ages` is ignored.

Additional Topics

In this chapter, you learned the basic rules for defining methods. You'll find more information about defining and using direct methods and generic function methods in this book.

Methods and Generic Functions

You've seen some method definitions, but there are many ways of defining and using methods with Apple Dylan.

- **Direct Methods**

  Direct methods are methods that you call directly—with no dispatch through a generic function. See Chapter 9, "Direct Methods," for many examples of functional programming with these methods.

- **Generic Functions**

  Generic functions, on the other hand, provide polymorphism and support the object-oriented approach of Apple Dylan. See Chapter 17, "Generic Functions," for more information.

Parameters

Apple Dylan offers a wide variety of parameter types you can use when defining your functions.

- **Formal Parameter Lists**

  Formal parameter lists can become quite complicated. See the *Dylan Reference Manual* for more information.

- **Parameter Specialization**

  Specifying the types of values that a method accepts as arguments not only helps type checking and compiler optimization, it also forms the basis of polymorphism, which is implemented with method dispatching.

  Method dispatch is the process by which a generic function selects a method to execute based on the arguments provided in the function call.

  In general, method dispatch examines the classes of the arguments and chooses the most appropriate method to apply to those arguments.

  You can find examples of method dispatch in Chapter 17, "Generic Functions." The complete algorithm for method dispatch is complicated; you can find it in the *Dylan Reference Manual*.

- **Singletons**

  Singletons provide a mechanism for very specific parameter specialization: you can specialize a parameter so that it will accept only one

exactly-specified value. See Chapter 17, "Generic Functions," for examples, but read the *Dylan Reference Manual* for complete information about Apple Dylan's type, class, and singleton system.

CHAPTER 8

Methods

CHAPTER 9

Direct Methods

Contents

Key Concepts 135
About Direct Methods 136
Bare Methods 136
Local Methods 138
  Local Recursion 139
  Mutual Recursion 140
Method Closures 142
Anonymous Methods 145
Additional Topics 148

This chapter examines the specific tasks you can accomplish using direct methods—methods not dispatched through a generic function.

Key Concepts

About Direct Methods

- A **direct method** is a method you call directly.
- This chapter discusses four kinds of direct methods you can use with Apple Dylan.

Bare Methods

- A **bare method** is a direct method bound to a module variable.
- You create bare methods using the built-in `method` expression.

Local Methods

- A **local method** is a method with a local scope.
- You create local methods with **local method declarations.**
- You can use local methods to program with **mutual recursion.**

Method Closures

- A **method closure** is a method that has one or more closure variables.
- A **closure variable** is a local variable with indefinite extent—it holds its value between calls to the method closure.

Anonymous Methods

- The built-in method expression returns **anonymous method** objects—they are bound to no variable name.
- You can use anonymous methods for immediate calculations and for some interesting functional programming techniques.

About Direct Methods

Polymorphism is a powerful aspect of Apple Dylan, allowing for many of its object-oriented features as well. As a result, many of the methods you create will belong to a particular generic function. You won't call the method directly, instead you'll call the generic function and it will dispatch the job appropriately.

However, you are not required to call methods through the generic function dispatching mechanism. In many cases, you might want to call a specific method directly. For example

- You may create a **bare method**—a method without a generic function—because you have no need for method dispatch, or want to avoid the slight overhead involved in dispatching.
- You may create a **local method**—a method with a limited scope and extent—that doesn't require method dispatching.
- You may create an **anonymous method**—a bare method object bound to no variable name—that you define and use in the same expression and therefore do not need a module variable.
- You may create a **method closure**—a method whose local variables retain their values between method calls—that requires you to use a local method kind of definition.

Although generic function methods are perhaps the most common, the method-dispatching mechanism requires understanding of the more advanced object-oriented aspects of Apple Dylan. So, this chapter focuses primarily on the direct methods—those without a generic function.

Bare Methods

A **bare method** is a method that does not belong to a generic function. You can create bare methods using the built-in `method` expression. The methods returned by this expression are simply method objects—they are not named; however, you can explicitly bind a name to such a method.

Here is a simple example:

```
define variable double = method (x) 2 * x end;
```

This example uses a `method` expression to create a method that doubles the value of its argument. This example also creates a binding: it binds the name of the module variable, `double`, to the method object.

The `double` method is a bare method since it does not belong to a generic function. You can call this bare method using the variable name bound to it, which in this case is `double`:

```
double(4);
double(1/2);
double( 1 / 2 );
```

Since this method is a variable (a variable name bound to a method object), you can bind a new method object to the name `double` with an assignment:

```
double := method (x) 3 * x end;
```

If you know that you'll never want to change the binding (that is, bind a different method object to the name `double`) you can use a constant definition:

```
define constant double = method (x) 2 * x end;
```

This example creates a named constant—a module variable that you cannot assign to. The name of the constant is `double` and its value is the bare method object.

You can call this bare method anywhere within its module using its module variable name (`double`):

```
*my-salary* := double(*my-salary*);
```

Local Methods

When declaring a local method, you use the `local` declaration form, which allows you to specify the method name, the parameter list, and the body of the method:

```
local [ method ] method-name (parameter-list)
    method-body
end [ method ];
```

As an example, the following local method declaration creates a local method named `sort`, which takes two parameters. The method returns two values: the values of the two parameters in ascending order.

```
local method sort (a, b)
    if (a < b)
        values (a, b)
    else
        values (b, a)
    end if;
end;
```

As you can see, the syntax of local method declarations is similar to the syntax of `method` expressions.

You can declare a local method inside any body of code. For example, here is a local method declaration contained in the body of a method definition:

```
define method sum-of-squares (x, y)
    local method square (z)     // local method declaration
        z * z;
    end;

    square (x) + square (y);
end method;
```

This method definition defines a method named `sum-of-squares`. The body of this method contains a local method declaration that declares a local method

named `square`. You can call the `square` local method anywhere within the body of the `sum-of-squares` method. However, outside of the body of the `sum-of-squares` method, you cannot call the `square` local method defined in this example. The body of code in which you can call a local method is called the local method's scope.

You can use a single local method declaration to declare multiple local methods, separating the method specifications by commas. For example,

```
local method double (a :: <integer>)
          2 * a;
      end,
      method triple (a :: <integer>)
          3 * a;
      end;
```

This local declaration declares two local methods: a local method named `double` and a local method named `triple`.

Local Recursion

Local methods declared in the same local declaration can call each other.When you declare a local method using the `local` declaration form, the scope of the local method name includes the body of the local method itself (as well as its parameter list). Therefore, local methods can be recursive—that is, they may contain calls to themselves.

For example, consider this method definition:

```
define method sum-two-lists (list-1 :: <list>, list-2 :: <list>)

    local method recursive-sum (the-list :: <list>)
        if (empty? (the-list))
            0
        else
            head(the-list) + recursive-sum(tail(the-list));
        end if
    end;
```

```
    recursive-sum(list-1) + recursive-sum(list-2);

end method;
```

This method definition defines a method named `sum-list-elements`, which takes two list arguments. The body of this method consists of a local method declaration and two calls to that local method.

The local method, named `recursive-sum`, takes a single list parameter. This local method determines if the list is empty, and if so returns a value of 0. If the list is not empty, the local method determines the value of the first element of the list and adds that value to the sum of the values of the rest of the elements in the list. The local method calls itself to determine the sum of the remaining values. This recursive method call is permitted because the scope of the `recursive-sum` local method includes the body of the local method itself.

Mutual Recursion

Apple Dylan permits another form of recursion along the same lines. If you define a method that contains two local methods, these local methods can call each other, because the scope of each local method includes the body of the other one.

Listing 9-1 provides an example using two local methods that call one another. This examples counts the number of times that a list of numbers switches from being increasing to decreasing.

Listing 9-1 Mutual recursion

```
define method count-swings (numbers :: <list>)
    => (swings :: <integer>);

    local method up (numbers :: <list>, count-so-far :: <integer>)
              let first = numbers.head;
              let rest = numbers.tail;
              if (empty?(rest))
                  count-so-far
              elseif (first <= rest.head)
                  up(rest, count-so-far)
```

```
            else
                down(rest, count-so-far + 1)
            end if;
        end method,

        method down (numbers :: <list>, count-so-far :: <integer>)
            let first = numbers.head;
            let rest = numbers.tail;
            if (empty?(rest))
                count-so-far
            elseif (first >= rest.head)
                down(rest, count-so-far)
            else
                up(rest, count-so-far + 1)
            end if;
        end method;

    if (size(numbers) < 2) 0
    elseif (numbers[0] <= numbers[1]) up(numbers, 0)
    else down(numbers, 0)
    end if;
end method;
```

In this example, there are two local methods, `up` and `down`. The `up` local method is called when a list has previously been determined to be increasing. This method examines the first two elements of the list:

- If the first element in the list is less than the second, the list is still increasing, so the `up` method calls itself to examine the rest of the list.
- If not, the list has changed direction and started decreasing, so the `up` method calls the `down` method to examine the rest of the list. It also increments the `count-so-far` parameter, to reflect that the list has changed direction again.

The down local method works in the opposite way:

- If the first element in the list is greater than the second, the list is still decreasing, so the `down` method calls itself to examine the rest of the list.
- If not, the list has changed direction and started increasing, so the `down` method calls the `up` method to examine the rest of the list. It also increments

the `count-so-far` parameter, to reflect that the list has changed direction again.

So, each of these local methods calls itself and the other.

Here is an example of executing this example:

```
> count-swings( #(1, 2, 3, 4, 3, 2, 5, 7, 6) );
3
```

Method Closures

A method closure is a method that uses a kind of local variable called a **closure variable:**

- Like regular local variables, closure variables have a **local scope;** that is, they can only be referenced in the body of their method.
- Unlike regular local variables, closure variables have an **indefinite extent;** that is, they continue to exist in memory even when their method is not currently executing.

Apple Dylan maintains a set of closure variables for each method closure that you create. This set of variables exists whenever the method closure is executing. These closure variables also exist between executions of the method closure.

A method closure can use closure variables to store values from one execution of the method until the next time the method is called. Whenever the method closure is executing, it has access to the closure variables, which have persisted (and kept their values) since the previous execution.

A single closure variable can belong to two or more different method closures; that is, you can create a closure variable accessible to some methods, but not accessible to others.

Method closures provide one way to encapsulate data and functionality. A method closure is a capsule of functionality that includes its own set of persistent data.

Here is a simple example that shows how you can make a method closure:

Listing 9-2 Creating a simple method closure

```
define method make-an-incrementor ()

    let count = 0;

    method ()
        count := count + 1;
    end;

end;

define variable keep-score = make-an-incrementor();
```

This listing includes a method definition and a variable definition that calls the method. The method definition defines a function called `make-an-incrementor`. This function creates a method closure and returns the method closure as its result.

The body of the `make-an-incrementor` method includes a local variable, named `count`, which is initialized to a value of 0. This variable is simply a local variable; there is nothing special about it (yet).

The `make-an-incrementor` method then includes a `method` expression. This `method` expression creates a method object and returns it as the result of the expression. The method created by this expression is very simple: it increments a variable named `count` and (as a result of the assignment operation) returns the new value of the variable as the result of the method.

Notice that the `count` variable is a local variable of the `make-an-incrementor` method. Therefore, the scope of this variable is the entire body of the `make-an-incrementor` method. Since the `method` expression occurs inside the body of the `make-an-incrementor` method, references to the `count` variable contained in the `method` expression are perfectly valid.

Now, when you call the `make-an-incrementor` function, something interesting happens: it returns an anonymous method object that increments a variable named `count`. However, once the `make-an-incrementor` method finishes

executing, its local variable named `count` is no longer in scope and normally the local variable would be destroyed.

However, Apple Dylan recognizes that the anonymous method returned by the `make-an-incrementor` method could very likely reference that `count` variable sometime in the future. So, instead of destroying the `count` local variable, Apple Dylan converts it into a closure variable belonging to the anonymous method.

The example in Listing 9-2 ends with this variable definition, which creates and names a method closure:

```
define variable keep-score = make-an-incrementor();
```

This variable definition calls the `make-an-incrementor` method, which creates an anonymous method that has a closure variable. The variable name `keep-score` is then bound to this anonymous method closure.

Subsequently, you can use the `keep-score` variable name to call the method closure:

```
> keep-score();
1

> keep-score();
2

> keep-score();
3
```

As you can see, each call to the method closure increments its closure variable.

You can re-use the `make-an-incrementor` method to create more method closures. Each time you call the `make-an-incrementor` method, it returns a new method closure with its own unique closure variable. Every method closure created this way uses the same name for its closure variable (`count`), but the variables themselves are separate; Apple Dylan maintains a separate closure variable (named `count`) for each method closure.

As an example, suppose you create a second method closure:

```
define variable days-so-far = make-an-incrementor();
```

You can now call this new method closure to determine that it does, indeed, maintain its own separate closure variable:

```
> days-so-far();
1

> keep-score();
4

> days-so-far();
2
```

As another example, the following method creates two method closures:

```
define method make-two-closures ()

    let count = 0;

    let incrementor = method ()
                          count := count + 1;
                      end;

    let decrementor = method ()
                          count := count - 1;
                      end;

    values(incrementor, decrementor);
end;
```

Each time you call this function, it returns two method closures that share the same closure variable; one method closure increments the closure variable, the other decrements the closure variable.

Anonymous Methods

If you are new to object-oriented, dynamic programming languages, you may wonder why you would want to create an anonymous function. These functions are handy in a number of situations:

- You can use them to perform a local calculation—for example, a calculation that you need to perform only once.
- You can use them to create functions that you know will never require any method dispatching, and therefore will never need a generic function.
- You can use them to explicitly prevent method dispatching from occurring.
- You can use them to build up a generic function programmatically, rather than have Apple Dylan do it for you.
- You can use them as **functional arguments**—arguments that are themselves function objects. (The next section gives an example using the `apply` function.)
- You can use them to create method closures.

Since an anonymous method has no name, you might wonder how to call an anonymous method once you have defined one. Remember from Chapter 6 the standard function call syntax:

function-specifier `(` *argument-list* `);`

The function specifier is typically a variable name that evaluates to a generic function object. For example, consider this function call:

```
element ("fascinating", 3);
```

The function specifier is the variable name `element`. Evaluating this variable name results in the built-in generic function object named `element`. This generic function examines the arguments, selects the most appropriate method, applies that method to the arguments, and returns the method's result.

However, the function specifier does not have to be a variable name bound to a generic function object. In fact, the function specifier can be any expression that evaluates to a generic function object or to a method object.

Since a `method` expression evaluates to a method object, you can use a `method` expression as the function specifier in a function call.

Here is an example:

```
method (x) x * 2 end (4);
```

This odd-looking line of code is actually a function call.

- The function specifier of this function call is

  ```
  method (x) x * 2 end
  ```

  which evaluates to an anonymous method that takes one argument and returns double the value of the argument.

- The argument list of the function call is

  ```
  (4)
  ```

  which specifies a single argument with a value of 4.

When you evaluate the function call, Apple Dylan applies the anonymous method to the argument, which results in a return value of 8.

This syntax for a function call might seem strange if you do not have any experience with object-oriented dynamic languages. You can add parentheses to make the syntax more readable:

```
(method (x) x * 2 end) (4);
```

You can also format the `method` expression over several lines:

```
method (x)
    x * 2
end (4);
```

Another way to call an anonymous method is to use the built-in `apply` function to call an anonymous method. Here is an example:

```
>apply (method (x, y) x * y end, #[3, 7]);
21
```

This call to the `apply` function applies the anonymous method specified by its first argument to the values specified by its second argument. In this example, the `method` expression is being used as a **functional argument.**

Additional Topics

- **Generic Functions**

  Generic functions, on the other hand, provide polymorphism and support the object-oriented approach of Apple Dylan. See Chapter 17, "Generic Functions," for more information.

- **Functional Programming**

  Apple Dylan provides many functions that operate on methods as objects. Some of these you can find in Chapter 14, "Collection Classes," but see the *Dylan Reference Manual* for the complete set.

CHAPTER 10

Conditionals

Contents

Key Concepts 151
About Conditionals 152
If Statements 153
Unless Statements 155
Case Statements 156
Select Statements 158
Additional Topics 160
  Statements 160
  Predicates 161
  Bodies of Code 161

This chapter describes conditionals, which are statements you use to affect the flow of control of your program. Conditional statements allow you to execute code conditionally—that is, you can specify conditions which determine whether or not specific bodies of code are executed.

Key Concepts

About Conditionals

- **Conditionals** are on of the kinds of built-in statements. The others are begin-end statements, block statements, and iterative statements. You can create your own kinds of statements by defining statement macros.
- Conditional statements, in general, include one or more **test expressions** and one or more **bodies of code.** The test expressions determine whether the bodies of code are executed.
- In general, test expressions are considered to be false if they evaluate to the Boolean value `#f`. *All other values* are considered to be true. Once again: not only is the Boolean value `#t` considered to be true, but *every value* besides `#f` is considered to be true—including some surprising values like 0, -1, the empty string "", and the empty list `#()`.

If Statements

- **If statements** allow you to execute a body of code if a test expression is not false.
- If statements can include **elseif clauses,** which include additional test expressions and bodies of code. Each test expression is evaluated in order until one is not false; the corresponding body of code is then executed.
- If statements can also include an **else clause,** which contains a body of code to execute if the test expressions in all other clauses fail.

Unless Statements

- **Unless statements** are the opposite of if statements; they execute a body of code only if a test expression evaluates to false. This statements contain only a single clause.

Case Statements

- **Case statements** are similar to if statements. They contain a series of test expressions and corresponding bodies of code.
- Each test expressions is evaluated in order until one is not false. The corresponding body of code is executed.
- Case statements may have an **otherwise clause,** which contains a body of code that is executed when all test expressions have failed.

Select Statements

- **Select statements** are slightly different than the other conditionals. They contain a test expression, called a **target expression.** They include different bodies of code to execute depending on the value of the target expression.

About Conditionals

Conditionals, or **conditional statements,** are expressions that allow you to execute code depending on specified conditions. As expressions, they are source code, they are compiled, they are evaluated, and they return values.

Like all statements, a conditional statement is actually a **macro call**—in particular, a call to a statement macro. Conditional statements are all calls to **built-in statement macros.**

Unlike function calls, which always evaluate every one of their subexpressions (for example, every one of their argument expressions), statements may evaluate some of their subexpressions and not evaluate others. In particular, conditional statements typically evaluate their test expressions, one by one, until one is not false. Then the correspond body of code is executed. The remaining test expressions and the other bodies of code are not evaluated at all.

The bodies of code in conditional statements can include local declarations as well as expressions. The value returned by a body of code is the value returned by the last expression evaluated within that body.

There are four kinds of built-in conditional statements:

- if statements

- unless statements
- case statements
- select statements

The next four sections discuss these statements in detail.

If Statements

An **if statement** allows you to execute code conditionally. The simplest syntax for an if statement is

```
if ( test-expression )
    body
end [ if ]
```

An if statement always evaluates the *test-expression*.

- If the *test-expression* evaluates to any true value (that is, any value that is not the Boolean value `#f`), then the constituents in the *body* are evaluated and the value of the if statement is the value of the last expression evaluated in the *body*. If there are no constituents in the *body*, then the value returned by the if statement is the Boolean value `#f`.
- If the *test-expression* evaluates to the Boolean value `#f`, then the constituents in the body are not evaluated and the value returned by the if statement is `#f`.

Here is an example of a simple if statement:

```
if ( *pay-check* < $minimum-wage )
    report-to-authorities ( *my-company* );
    *my-company* := find-new-job ();
end if;
```

If statements can have an optional **else clause:**

```
if ( test-expression )
    body
[ else
    else-body ]
end [ if ]
```

If the *test-expression* evaluates to the Boolean value `#f`, then the constituents of the *body* are not evaluated; instead, the constituents of the *else-body* are evaluated and the value of the `if` statement is the value of the last expression evaluated in the *else-body*. If the *else-body* is empty, then the if statement returns the Boolean value `#f`.

Here is an example of an if statement with an `else` clause:

```
if ( *my-pay* < *your-pay* )
   switch-jobs ( *me*, *you* )
else
   buy-sympathy-card ( *you* )
end if
```

Finally, an if statement can have any number of `elseif` clauses:

```
if ( test-expression )
    body
{ elseif ( elseif-test-expression )
    elseif-body }
[ else
    else-body ]
end [ if ]
```

An if statement with `elseif` clauses first evaluates the *test-expression*. If it evaluates to the Boolean value `#f`, then the statement evaluates each *elseif-test-expression* in order until one returns any true value (that is, any value that is not the Boolean value `#f`). If any *elseif-test-expression* evaluates to a true value, the corresponding *elseif-body* is evaluated and the result of that body is the result of the if statement; no other bodies are evaluated.

Here is an example of an if statement with multiple `elseif` clauses:

```
if ( *current-day* = #"monday" )
    go-to ( $work );
    get-paid ( $paycheck-amount, *bank-balance* )
elseif ( *current-day* = #"friday")
    go-to ( $home )
elseif ( *current-day* = #"saturday")
    go-to ( $city );
    spend ( *bank-balance* )
else
    stay-put ()
end if;
```

Unless Statements

Unless statements are the approximately the opposite of if statements (except unless statements don't have the added complexity of `elseif` and `else` clauses). The syntax for an unless statement is

```
unless ( test-expression )
    body
end [ unless ]
```

An unless statement always evaluates the *test-expression*.

- If the *test-expression* evaluates to the value `#f`, then the constituents in the *body* are evaluated and the value of the last expression evaluated in the *body* is returned as the value of the unless statement. If there are no constituents in the *body,* then the value `#f` is returned as the value of the unless statement.
- If the *test-expression* evaluates to any other value than `#f`, then the constituents in the body are not evaluated, and `#f` is returned as the value of the unless statement.

Here is an example of an unless statement:

```
unless ( *today* = "friday" )
    let todays-message = "Is it friday yet?";
    communicate ( todays-message, *co-workers* );
    communicate ( todays-message, *friends* );
    communicate ( todays-message, *family* );
end unless;
```

This unless statement contains a body of code with a local variable—a string. It uses that local variable in three subsequent function calls. When the unless statement is finished executing, the local variable is no longer accessible.

Case Statements

Case statements are similar to if statements with `elseif` and `else` clauses. The syntax of a simple case statement is

```
case
    { test-expression => case-body ;} [;]
end [ case ]
```

The case statement can have any number of test clauses, separated by semicolons (with an optional semicolon after the last clause).

A case statement evaluates each *test-expression* in order until one returns a true value (that is, any value that is not the Boolean value `#f`). If any *test-expression* returns a true value, the corresponding *case-body* is evaluated and the case statement returns the value of the last expression evaluated in the *case-body*. (If the case-body is empty, the case statement returns the Boolean value `#f`). No further *test-expressions* are evaluated (and no other *case-body* is evaluated).

Here is an example of a simple case statement:

```
case
    x < y => "less than";
    x > y => "greater than";
    x = y => "equal to";
end case;
```

This case statement contains three simple test expressions and three simple case bodies. Assuming `x` and `y` are variables with numerical values, one of these case bodies should be evaluated and returned as the value of the case statement.

Case statements can contain an optional **otherwise clause**. Here is the syntax:

```
case
    { test-expression => case-body ; }
    [ otherwise => otherwise-body ] [;]
end [ case ]
```

If a case statement has an otherwise clause, the *otherwise-body* is evaluated only if every *test-expression* evaluates to the Boolean value `#f`.

Here is an example of such a case statement:

```
case
    x < y => "less than";
    x > y => "greater than";
    x = y => "equal to";
    otherwise => "trouble";
end case;
```

It is permissible for a case statement to have nothing but an otherwise clause:

```
case
     otherwise => let mixed-up-list = #(3, 2, 5, 4, 1, 7);
                  let sorted-list = sort (mixed-up-list);
                  mixed-up-list = sorted-list;
end
```

This case statement has a single clause—an otherwise clause. When this case statement is evaluated, the otherwise body is guaranteed to be executed. (In this case, the body of code creates two local variables and returns `#t` if their values are equal and `#f` otherwise.) Notice that the body is an implicit body—the word `begin` and `end` do not have to surround the body.

Select Statements

A **select statement** is similar to a case statement, except that it chooses a body of code to execute depending on the value of an expression. The simplest syntax of a select statement is

```
select ( target-expression )
    { match-list => body ;} [;]
end [ select ]
```

where *match-list* is a series of expressions separated by commas. In this simple version of a select statement, the *target-expression* is evaluated and then each expression in each *match-list* is evaluated in order until a match is found. When a match is found, the corresponding *body* is evaluated and the select statement returns the value of the last expression evaluated in the *body.* (If the *body* is empty, the select statement returns the Boolean value `#f`.)

Once a match is found, no more *match-list* expressions are evaluated. If no match is found, the select statement returns the Boolean value `#f`.

An example of a simple select statement is

```
select ( get-test-score () )
    9, 10 => *grade* := 'A';
             "Excellent";
    7, 8  => *grade* := 'B';
             "Fair";
end select
```

In this case statement, the *target-expression* is a function call. The value returned by that function call is compared to the value 9, then to the value 10, then to value 7, and then to the value 8 (but only until a match is found). If the value 9 or 10 is matched, then the first body of code is executed, assigning the value `'A'` to the global variable `*grade*` and returning the string `"Excellent"` as the return value.

If the value 7 or the value 8 is matched, then the second body of code is executed. If none of the specified values are matched, the value `#f` is returned.

You can also include an otherwise clause in select statements:

```
select ( target-expression )
    { match-list => body ; }
    [ otherwise => otherwise-body ]
end [ select ]
```

Here is an example:

```
select ( get-test-score () )
    9, 10     => *grade* := 'A';
                 "Excellent";
    7, 8      => *grade* := 'B';
                 "Fair";
    otherwise => *grade* := 'D';
                 "Poor";
end select
```

In the previous examples, the value of the *target-expression* is compared to the values of the expressions in each *match-list* using the equality operator (=).

You may specify a different test function for a select statement to use to determining a match. Here is the syntax:

```
select ( target-expression [ by test-function ] )
    { match-list => body ; }
    [ otherwise => otherwise-body ]
end [ select ]

Here is an example:

select ( get-test-score() by \>= )
    9         => *grade* := 'A';
                 "Excellent";
    7         => *grade* := 'B';
                 "Fair";
    otherwise => *grade* := 'D';
                 "Poor";
end select
```

In this example, the *test-function* is the greater-than-or-equals (>=) function, so fewer *match-list* values need to be specified.

Additional Topics

This chapter provides most of the information you need to use conditional statements. A few more facts and pointers are included here.

Statements

- **Statement Macros**

  Statements are macro calls—macro calls to statement macros. You can find out more about macros, macro calls, statement macros, and defining your own macros in Chapter 19, “Macros.”

- **Begin-End Statements**

  Although conditional statements use implicit bodies of code, in some situations you are required to make an **explicit body** of code by including the constituents in a begin-end statement. Chapter 3, “Constituents,” mentioned begin-end statements.

- **Block Statements**

  A block statement is a body of code that allows you to handle error conditions. See Chapter 20, “Conditions,” for details.

- **Iterative Statements**

  Iterative statements are similar to conditional statements, except that they allow you to execute bodies of code multiple times while certain conditions are met. Since these often make use of collection classes, they are discussed later in this book, in Chapter 11, “Iterators.”

Predicates

- **Boolean Values**

  The test expression used in conditional statements are evaluated to a Boolean false (`#f`), or to another value. The Boolean values are described in Chapter 4, "Values."

- **Predicate Functions**

  A predicate function is a function that returns a Boolean value. By convention, their names end with a question mark (`?`). Some built-in examples include `odd?`, `even?`, `zero?`, `positive?`, and `negative?`. See Chapter 13, "Numbers," for descriptions.

Bodies of Code

- **Code Bodies**

  A body of code is a series of constituents separated by semicolons, with an optional final semicolon. They can contain local declarations and expressions, but not definitions. See Chapter 3, "Constituents," for information about code bodies and expressions and Chapter 8, "Methods," for information about local declarations.

- **Implicit Bodies**

  Certain constituents in Apple Dylan, such as the conditional statements, accept implicit bodies—that is, bodies of code that are not contained within a begin-end statement.

CHAPTER 10

Conditionals

CHAPTER 11

Iterators

Contents

Key Concepts 165
About Iterators 166
While Statements 166
Until Statements 168
For Statements 169
  Numeric Clauses 170
  Collection Clauses 172
  Explicit Clauses 173
  End Tests 174
  Result Bodies 175
Additional Topics 176
  Statements 176
  Collections 176

This chapter describes iterators, which are statements you use, like conditionals, to alter the flow of control of your program. Iterator statements allow you to execute code iteratively—that is, you can repeat the same body of code multiple times, typically until some condition is met.

Key Concepts

About Iterators

- **Iterators** are one of the kinds of built-in statements. The others are begin-end statements, block statements, and conditional statements. You can create your own kinds of statements by defining statement macros.
- Iterator statements, in general, include one or more **test expressions** and one or more **bodies of code.** The test expressions determine whether and how many times the bodies of code are executed.
- In general, test expressions are considered to be false if they evaluate to the Boolean value `#f`. *All other values* are considered to be true. Once again: not only is the Boolean value `#t` considered to be true, but *every value* besides `#f` is considered to be true—including some surprising values like 0, -1, the empty string "", and the empty list `#()`.

While Statements

- **While statements** allow you to execute a body of code repeatedly if a test expression is not *false*.

Until Statements

- **Until statements** allow you to execute a body of code repeatedly while a test expression is *false*.

For Statements

- **For statements** allow you to execute a body of code repeatedly while managing one or more **iteration variable.**

About Iterators

Iterators, or **iterator statements,** are expressions that allow you to execute code multiple times depending on specified conditions. As expressions, they are source code, they are compiled, they are evaluated, and they return values.

Like all statements, an iterator statement is actually a **macro call** to a statement macro. Iterator statements are all calls to **built-in statement macros.**

Unlike function calls, which always evaluate every one of their subexpressions (for example, every one of their argument expressions), statements may evaluate some of their subexpressions and not evaluate others. In particular, iterator statements typically evaluate some test expression, and if conditions are met, the correspond body of code is executed, and then the process repeats. If the condition is not met, a body of code may not be executed at all.

The bodies of code in iterator statements can include local declarations as well as expressions. Typically, iterator statements return `#f` as their return value, although you can override that behavior with the for statement.

There are three kinds of built-in iterator statements:

- while statements
- until statements
- for statements

The next three sections discuss these statements in detail.

While Statements

A **while statement** allows you to execute a body of code repeatedly while a test expression evaluates to a true value. The syntax of a while statement is

```
while ( test-expression )
    body
end [ while ]
```

Here is the flow of control for a while statement:

1. The while state evaluates the *test-expression*.
2. If the test expression returns a true value (that is, any value that is not the Boolean value `#f`), the while statement evaluates the constituents of the *body*, and returns to step 1.
3. If the test expression returns the Boolean value `#f`, the while statement is finished executing and it returns the value `#f`.

A while statement always returns the Boolean value `#f` as its return value.

Here is an example of a while statement:

```
while ( *my-score* < $passing )
    take-test ();
    *my-score* := update (*my-score*);
end while;
```

The entire body of a while statement is executed repeatedly—including any local variable declarations. As a result, local variables are redeclared every iteration. For example:

```
while ( unfinished?() )
    let local-variable = 10;
    local-variable := local-variable + 1;
    calculate (local-variable);
end while;
```

This while statement calls the `calculate` function repeatedly (until the test expression `unfinished?()` returns `#f`). The argument sent to the calculate function is always 11—the local variable is redeclared each time the body of the while statement is executed.

The for statement, described in "For Statements" on page 169, allows you to keep local variables that are not reinitialized before each iteration.

Alternatively, you can wrap your while statement in another local body:

```
begin
    let local-variable = 10;
    while ( unfinished?() )
        local-variable := local-variable + 1;
        calculate (local-variable);
    end while;
end;
```

Here, the local variable is local to the begin-end statement, so it takes on the values 10, 11, 12, and so on during iterations of the while statement body.

Until Statements

An **until statement** allows you to execute a body of code repeatedly until a test expression evaluates to any non-*false* value. The syntax of an until statement is

```
until ( test-expression )
    body
end [ until ]
```

The until statement evaluates the *test-expression* and if it returns a Boolean value `#f`, it evaluates the constituents of the `body`. The *test-expression* is then evaluated again, and if it still returns a `#f` value, the body is evaluated again as well. This pattern repeats until the *test-expression* returns any non-*false* value (that is, any value other than a Boolean value `#f`).

An until statement always returns the Boolean value `#f` as its return value.

Here is an example of a until statement:

```
until ( *paycheck-amount* > 2000 * *age*)
    work-harder ();
    tell ($manager);
end until;
```

For Statements

For statements allow you to repeat bodies of code while maintaining **iteration variables** that control the number of iterations. The syntax of a simple for statement is

```
for ( clauses )
    body
end [ for ]
```

In this simple form of a for statement, the *clauses* control iteration variables that change each time the *body* is evaluated. An example of a simple for statement is

```
for (count from 1 to 5)
    *total* := *total* + count
end for
```

In this example, the for statement declares a local variable named `count` and initializes it to the value 1. The body of the for statement is executed, and then the `count` variable is incremented and the body is executed again. In this example, the body is executed 5 times, once each with `count` being equal to 1, 2, 3, 4, and 5.

For statements provide a number of different clauses you can use to control iteration variables. There are three types of clauses:

- **numeric clauses,** which create local variables with number values and automatically increment the values of these variables for you between iterations
- **collection clauses,** which create local variables that are initially bound to an element in a collection and that are assigned new values—subsequent elements in the same collection—between iterations
- **explicit clauses,** which create local variables for which you specify the initial value and how the variable should be modified between iterations

You can specify more than one clause in a for statement, and therefore maintain multiple iteration variables. If any iteration variable created by a numeric or a collection clause reaches its final value, the for statement finishes executing.

For statements also allow you to specify

- **end-test expressions,** which you can use to indicate when a for statement should quit iterating
- **result bodies,** which you can provide to be executed after the final iteration.

The next few sections examine the various clauses—numeric, collection, and explicit— as well as end-test expressions and result bodies.

Numeric Clauses

A numeric clause creates a **numeric iteration variable.** This is a variable local to the for statement that is assigned a different numeric value each time the body of the for statement is executed. You can specify the initial value, the bounding value, and the increment by which the value changes each iteration. When the final value is exceeded, the for statement quits iterating.

Numeric clauses have four parts:

- The variable name specification, which may include a type specifier. This part is required. Some examples are:

```
count
count :: <integer>
```

- The initial-value specification, which is also required. Some examples are:

```
from 1
from 10
from -20
```

- An optional bounding-value specification, which specifies when this iteration variable causes its for statement to stop iterating. Some examples include:

```
to 10
to 100
above 10
above -1
below 0
below 100
```

- An optional increment specification, which specifies how much to increment the value of the variable between iterations. Some examples are:

```
by 2
by 5
by -1
```

Putting these parts together, here are some example numeric clauses:

```
count from 0
```

This example starts `count` at 0 and increments `count` by 1 (the default) between iterations. This example has no bounding clause, so count will increment indefinitely, until some other clause or an end-test expression indicates that the for statement should stop iterating.

```
count from 1 to 5
```

This example starts `count` at 1 and increments `count` by 1 (the default) between iterations. When `count` is incremented and becomes greater than 5, this clause causes the for statement to stop iterating.

```
count from 1 to 5 by 2
```

This example starts `count` at 1 and increments it by 2 between iterations. When `count` is incremented and becomes greater than 5, this clause causes the for statement to stop iterating. As a result, `count` will be set to 1, 3, and 5 during the iterations. When it is set to 7, its for statement does not iterate again.

```
count from 4 to 1 by -2
```

This example starts `count` at 4 and decrements it by 2 between iterations. When `count` is decremented and becomes less than 1, this clause causes its for statement to stop iterating. As a result, `count` will be set to 4 and then 2 during the iterations. When it is set to 0 after the second iteration, its for statement does not iterate again.

```
count from 1 below 5
```

This example starts `count` at 1 and increments it by 1 (the default) between iterations. When `count` is incremented and becomes greater than *or equal to* 5,

this clause causes the for statement to stop iterating. As a result, `count` will be set to 1, 2, 3, and 4 during the iterations. When it is set to 5 after the fourth iteration, its for statement does not iterate again.

```
count from 5 above 1 by -1
```

This example starts `count` at 5 and decrements it by 1 between iterations. When `count` is decremented and becomes *less than or equal to* 1, this clause causes the for statement to stop iterating. As a result, `count` will be set to 5, 4, 3, and 2 during the iterations. When it is set to 1 after the fourth iteration, its for statement does not iterate again.

Collection Clauses

A collection clause creates a **collection iteration variable.** This is a variable local to the for statement that is assigned a different element from a collection between iterations of a for statement. You specify the collection; Apple Dylan uses collection iteration protocol to determine the initial value and each subsequent value.

Collection clauses have two parts:

- The variable name specification, which may include a type specifier. This part is required. Some examples are:

  ```
  elem
  elem :: <character>
  ```

- The collection specification, which is also required. Some examples are:

  ```
  in #(1, 2, 3)
  in #['a', 'b', 'd', 'z']
  in "testing every character"
  in *my-favorite-collection*
  in vector(max, min, lcd, gcm)
  ```

Putting these parts together, here are some example collection clauses:

```
letter :: <character> in "Apple Dylan"
```

This collection clause initializes the variable `letter` to the value `'A'`. Between each iteration, the `letter` variable is reassigned to the next character in the

string. When all the characters have been used, the for statement does not iterate again.

```
which-function in vector(max, min, lcd, gcm)
```

This collection clause initializes the variable `which-function` to the object representing the built-in `max` function. Between each iteration, the `which-function` variable is reassigned to the next function object in the vector. When all the elements of the vector have been used, the for statement does not iterate again.

Explicit Clauses

An explicit clause creates an **explicit iteration variable.** This is a variable local to the for statement that you explicitly assign values to between each execution.

Explicit clauses have two parts:

- The variable name specification, which may include a type specifier. This part is required. Some examples are:

  ```
  test
  test :: <Boolean>
  ```

- The initial-value specification, which is also required, can be any expression. Some examples are:

  ```
  = 1
  = *global*
  = gcd(10, *global*)
  ```

- The next-value specification, which is also required, can also be any expression. Some examples are:

  ```
  then test + 1
  then test + 2
  then increment(test)
  then *some-other-value*
  ```

Putting these parts together, here are some example explicit clauses:

```
test = 1 then test + 1
```

This explicit clause initializes the variable `test` to the value 1. Between each iteration, the `test` variable is reassigned to value of the expression `test + 1`. This clause alone never causes the for statement to stop executing.

```
test = #t then #f
```

This explicit clause initializes the variable `test` to the value `#t`. Before every other iteration, the `test` variable is reassigned to value `#f`. Again, this clause alone never causes its for statement to stop executing.

End Tests

You can also supply an optional **end-test expression** in a for statement:

```
for ( clauses [ ( until | while ) end-test-expression ] )
    body
end [ for ]
```

There are two types of end-test expressions:

- **while** end-test expressions, which terminate the iteration of the for statement if the end-test-expression evaluates to `#f`.
- **until** end-test expressions, which terminate the iteration of the for statement if the end-test-expression evaluates to any value except `#f`.

Here is a simple example of a for statement with an end-test expression:

```
define variable *abort* = #f;

for ( count from 1 to 10, until *abort* )
    if (count >= 5)
        *abort* := #t;
    end if;
end for;
```

This for statement iterates five times. On the fifth iteration, the value of the module variable `*abort*` is set to `#t`. Before the sixth iteration, the for statement increments `count` to 6, and then evaluates the end-test expression. Since it is an until end-test expression and the expression evaluates to `#t`, the for statement does not execute the body of code a sixth time.

Sometimes you need to have an end-test expression. For example:

```
for ( test = 1 then test + 1, until test = 100 )
    // body
end;
```

Since this example uses a single explicit clause, it has no indication of when to quit iterating without an explicit end-test expression.

Result Bodies

For statements normally return the value `#f` as their result. However, for statements allow you to specify a **result body** of code to execute after the final iteration—even if there were no iterations at all. The complete syntax of the for statement is

```
for ( clauses [ ( until | while ) end-test-expression ] )
    body
    [ finally result-body ]
end [ for ]
```

If you specify a result body, the for statement returns the value returned by the execution of that result body.

Here is a simple example of a for statement with a result body:

```
for ( count from 1 to 10 )
    // could do something here
    finally
        count;
end for;
```

This for statement returns the final value of the iteration variable `count`, which is 11. (Remember that `count` is incremented before the for statement compares it to its bounding value.) Therefore, the above for statement returns the value 11.

Additional Topics

This chapter provides most of the information you need to use iterator statements. A few more facts and pointers are included here.

Statements

- **Statement Macros**

  Statements are macro calls—macro calls to statement macros. You can find out more about macros, macro calls, statement macros, and defining your own macros in Chapter 19, "Macros."

- **Begin-End Statements**

  Although conditional statements use implicit bodies of code, in some situations you are required to make an **explicit body** of code by including the constituents in a begin-end statement. Chapter 3, "Constituents," mentioned begin-end statements.

- **Block Statements**

  A block statement is a body of code that allows you to handle error conditions. See Chapter 20, "Conditions," for details.

- **Conditional Statements**

  Conditional statements allow you to execute a body of code zero or one times, but not more. The previous chapter discussed conditionals.

Collections

- **Collections**

  Some clauses in the for statement iterate through the elements in a collection object. See Chapter 14, "Collections," for more examples.

- **Iteration Protocol**

  Some kinds of collections do not have an obvious order to their elements. See the *Dylan Reference Manual* for more information.

CHAPTER 11

Iterators

CHAPTER 11

Iterators

CHAPTER 12

Objects and Classes

Contents

Key Concepts 181
About Objects and Classes 182
  Classes 182
  Class Inheritance 184
  Creating and Initializing Objects 185
  Examining and Modifying Objects 186
Built-In Classes 188
  Simple Classes 189
  Number Classes 190
  Collection Classes 192
User-Defined Classes 193
  Class Definitions 193
  Slots 194
  Slot Names and Getter Functions 194
  Other Slot Specification Options 195
Additional Topics 196
  Object Creation and Manipulation 196
  Other Built-In Classes 196
  Class Inheritance 197
  Method Dispatch 198
  Types 198

Throughout this book, you've seen examples of different classes of objects: you've seen integer objects, ratio objects, list objects, vector objects, and so on.

This chapter, along with the next few, examine objects and classes of objects in more detail and discuss the relation between the two.

Key Concepts

About Objects and Classes

- **Objects** are units of information stored in runtime memory. Every object is an instance of a specific class.
- **Classes** define basic information about objects; they define object structure and they influence object behavior.
- Classes can **inherit** from other classes, called **superclasses.** A **hierarchy** of classes is the result.
- Generic functions use class information and the class hierarchy to determine the most appropriate method to call during method dispatch.

Built-In Classes

- **Built-in classes** are classes defined by Apple Dylan. Some are **abstract**—only meant to have **subclasses**. Some are **concrete**—you can use them to make objects.
- **Simple built-in classes** include Booleans, characters, and symbols.
- **Number classes** are included for your numerical information.
- **Collection classes** allow you to create powerful tools for organizing and manipulating groups of objects.

User-Defined Classes

- You can define your own classes using **class definitions.**
- User-defined classes have **slots.** Objects use these slots to store their private data.

About Objects and Classes

Objects are individual values stored in memory during runtime. These objects naturally fall into categories: some store numerical values, some store character values, some store string values, and so on. A category of similar objects is called a **class** of objects.

Every object belongs to a class. An object is sometimes called an **instance** of its class. For example, the object representing the character *Q* belongs to the class called `<character>`. The *Q* object is an instance of the `<character>` class.

Classes, then, describe the nature of a category of objects. In fact, an object's class determines a number of things about the object:

- Classes define the **structure** of their objects—that is, an object's class determines how that object is represented in memory.
- Classes influence the **creation** of their objects—that is, the way in which you create an object depends on that object's class.
- Classes influence the **initialization** of their objects—they influence how initial values are assigned to their newly-created instances.
- Classes influence the **behavior** of their instances—through type specification and method dispatch, the class of an object influences which implementation of a function is selected when a function is applied to the object.

Classes obviously exert substantial influence over their instances—objects of the same class share many qualities. However, individual objects of the same class are still distinct from one another. Two objects of the same class may have the same basic structure, yet still store two very different values.

Classes

By convention, the names of Apple Dylan classes begin and end with angle brackets. For example, `<integer>`, `<ratio>`, `<string>`, and `<list>` are all names of Apple Dylan classes. In fact, they are all names of built-in classes—classes that Apple Dylan provides for you. When creating and manipulating objects, you typically use the built-in classes for some objects and define your own classes for other objects:

- **Built-In Classes.** These classes are provided for you; they describe common kinds of objects, such as characters and numbers. Apple Dylan also provides classes for useful collections of objects, such as lists and hash tables. Built-in classes provide many benefits:
  - You can create and use objects without having to define classes yourself.
  - Apple Dylan can optimize the internal representations used for the most common kinds of objects.
  - Apple Dylan can also optimize the built-in functions that apply to these common kinds of objects.
  - Apple Dylan can also provide specialized functions for creating, initializing, specifying, examining, and modifying these common kinds of objects.

  "Built-In Classes" on page 188 introduces these classes of objects in more detail.

- **Classes You Define.** Defining your own classes involves more work than simply using the built-in classes. Also, Apple Dylan can't provide your classes with the same level of optimization as it can for the built-in ones. However, defining your own classes provides you with a powerful tool that allows you to organize information in the manner that best suits your program's needs. It also allows you to express powerful functionality with concise source code.

  Unlike the built-in classes of objects, which have internal structures and accessing mechanisms that are optimized (and therefore not standardized), the classes of objects you define all have similar structures and accessing mechanisms:
  - These objects contain **slots**—a unit of storage within an object.
  - You define a class of objects by describing the slots contained by the objects of the class.
  - Every object of the same class has the same basic structure—that is, they have the same slots.
  - However, every object of the same class can store different information—that is, they each can have their own values for those slots.

  Therefore, the class determines the set of slots that its objects have, but each object determines the values it stores in those slots.

  "User-Defined Classes" on page 193 contains more detail about defining your own classes and specifying their slots.

Additional Topics

Classes themselves (like the `<integer>` class) are objects during runtime. Their names (like `<integer>`) are simply variable names that happen to be bound to class objects. As a result, classes can be passed to functions and otherwise manipulated as data. See "Classes As Objects" on page 198. ◆

Class Inheritance

Apple Dylan classes are not a set of completely distinct, separate categories of objects. Instead, the classes are related to one another through the paradigm of **inheritance.** Inheritance allows for data abstraction and code reusability.

Every Apple Dylan class has one or more other classes that it inherits from—these classes are called its **superclasses.** It is a **subclass** of each of its superclasses.

The only exception is the class `<object>`, which is the class that has no superclasses. All other classes are either **direct subclasses** of `<object>`, or **general subclasses,** which includes subclasses of subclasses, and so on.

In general, subclasses are more specific, or specialized, versions of their superclasses. They inherit some attributes from their superclasses and specialize other attributes.

You've seen how a generic function uses method dispatch to select the appropriate implementation to invoke in response to a function call. The generic function matches the classes of the arguments specified in the function call to the type specializers of the methods to select the appropriate method.

Due to inheritance, a generic function might have more than one applicable method for a set of arguments. In this case, method dispatch chooses the most specific method—the applicable method with the most specific type specifiers (that is, methods specialized to subclasses are chosen over methods specialized to superclasses).

Additional Topics

Method inheritance is powerful but complex. As a result, the subject is not fully introduced until Chapter 16, "Inheritance," but see "Class Inheritance" on page 197.

Since classes can have more than one superclass, the set of all Apple Dylan classes forms a **heterarchy** (a directed acyclic graph) rather than a hierarchy. See "Heterarchies" on page 197.

Class heterarchies create some ambiguous method-dispatching situations. See "Method Dispatch" on page 198.

Class dynamism refers to the ability to control whether classes can be subclassed in other modules—modules that do not contain the class definition itself. See "Class Dynamism" on page 197.

Creating and Initializing Objects

You've seen a number of ways to create objects, and to initialize their values, throughout this book. Two examples of object creation and initialization include literal constant evaluation and function call evaluation:

```
define variable *new-string-object* = "this is a new object";

define variable *new-list* = concatenate (#(1, 2), #(3, 4));
```

The first example creates (and initializes) a string object using a string literal. The second creates a list object using a function call, which uses literal constants as its argument expressions.

These object-creation mechanisms are limited, however. For example, you cannot use literal constants to create objects of some of the built-in classes, nor can you use literals to create objects of classes you define yourself.

Apple Dylan provides the `make` function to allow you create an object of a specified class. Here is a simple example:

```
define variable *new-list* = make(<list>);
```

This call to `make(<list>)` returns a list object—in this case, the empty list:

```
> *newlist*
#()
```

In this example, the empty list is returned, because that is the default initial value for a `<list>` object. Depending on the class of object you're creating, the `make` function allows you to specify keyword arguments, called **init-keywords,** to specify an initial value different from the default. For example:

```
> make(<list>, size: 3, fill: "ha");
#("ha", "ha", "ha")
```

This call to the `make` function specifies two init-keywords, which indicate the size and elements of the initial value of the newly-created `<list>` object.

Additional Topics

The `make` function calls the `initialize` function to initialize the new object. These functions are not defined for all classes.

When you define your own classes, you may need to provide specialized implementations (methods) for these functions. Frequently, however, you provide enough information in the method definition for the built-in `make` and `initialize` functions to work without specialized method definitions. See Chapter 15, "Defining Classes," for more information. ◆

Examining and Modifying Objects

In Apple Dylan, you examine the contents of an object (such as the value of an) its elements or its slots) using a function, rather than examining memory locations directly, as in some languages.

For example, the built-in `element` function allows you to determine the value of an element of a collection object:

```
> element ( "abcd", 1 );
'b'
```

This function call returns element number 1 (starting at 0) of the string `"abcd"`—which is the character `'b'`.

Apple Dylan provides a special function call syntax intended for use with the `element` function:

```
> "abcd"[1];
'b'
```

If that notation seems odd, consider this similar version:

```
> define constant $sample-string = "abcd";
defined $sample-string

> $sample-string[1];
'b'
```

If an object is mutable, you can modify its contents. The built-in `element-setter` function allows you to change the value of an object element:

```
> element-setter( '!', $sample-string, 1 );
'!'

> $sample-string
"a!cd"
```

Apple Dylan also provides a special function call syntax for the `element-setter` function. This syntax combines the `[]` notation with the assignment operation:

```
> $sample-string[0] := '?';
'?'

> $sample-string
"?!cd"
```

Additional Topics

The objects with elements are collection objects—for more information, see "Collection Classes" on page 192 and Chapter 14, "Collections."

In user-defined classes, you examine and modify slots, rather than elements. See "User-Defined Classes" on page 193, and Chapter 16, "Defining Classes."

Whereas mutability describes whether you can change an object's value at all, accessibility denotes how much access you have over examining and modifying an object from outside its own module. See "Mutability" on page 196. ◆

Built-In Classes

There are a number of built-in classes that Apple Dylan provides. Three of the most common categories of built-in classes are

- **Simple classes.** These classes include <Boolean>, <character>, and <symbol>. They are all direct subclasses of <object>, and they have no subclasses themselves. The next section, "Simple Classes," discusses these classes.
- **Number classes.** These classes represent numerical values; they include real numbers, floating-point numbers, rational numbers, integers, and so on. "Number Classes" on page 190 introduces these classes.
- **Collection classes.** These classes represent collections—objects that have other objects as **elements.** "Collection Classes" on page 192 introduces these classes.

Additional Topics

Functions, classes, and conditions are also represented by objects during runtime in Apple Dylan. Therefore, there are also built-in classes that are function-related, that are class-related, and that are condition-related. See "Other Built-In Classes" on page 196. ◆

Simple Classes

The simple classes are <Boolean>, <character>, and <symbol>:

- The <Boolean> class has exactly two instances—the built-in object *true* and the built-in object *false*. You can represent these objects using the literals #t and #f. These objects are immediate and immutable.

  Boolean values are important particularly in relation to the conditional and iterative statements, explained in Chapters 10 and 11, and in relation to the logical operators, particularly the special built-in operators & and |, which are described with the other operators in the next chapter.

- The <character> class represents the alphanumeric characters (8-bit? ASCII?). These objects are also immediate and immutable. You can specify objects of this class using literal constants delimited by single quotes.

  Apple Dylan provides the as-uppercase and as-lowercase functions to allow you to specify the case of alphabetic characters. Here are some examples:

```
> as-uppercase('a');
'A'

> as-uppercase('A');
'A'

> as-uppercase('?');
'?'
```

  You can also convert between characters and integers using the built-in as function:

```
> as( <integer>, 'a' );
97

> as( <character>, 97 );
'a'
```

- The <symbol> class represents symbol objects—unique, immutable objects associated with non-case-sensitive strings. Apple Dylan maintains a dictionary of symbol objects—this dictionary is guaranteed to contain only one object for any particular non-case-sensitive string.

You can specify symbol objects using symbol literals; for example:

```
#"Hello, world!"
#"hello, world!"
#"HELLO, WORLD!"
```

These three literals refer to the same object—a unique symbol object in Apple Dylan's symbol dictionary that corresponds to the characters *h e l l o , w o r l d !*.

You can use the keyword syntax to specify symbols if their strings follow the rules for standard variable names (have no whitespace, and so on):

```
key-word-symbol:
KEY-world-SYMbol:
```

Again, these two literals refer to the same symbol object—the one corresponding to the characters *k e y - w o r d - s y m b o l*.

You can convert between symbols and strings using the built-in `as` function:

```
> as( <string>, #"Hello" );
"hello"

> as( <symbol>, "Hello" );
#"hello"
```

Symbols provide you with a mechanism for efficient non-case-sensitive string storage and fast string comparison.

(Note that the keyword syntax is used in certain Apple Dylan expressions, such as function calls with keyword arguments.)

Number Classes

The number classes define objects that represent different kinds of numerical values. Apple Dylan provides an entire hierarchy of number classes.

Some of number classes, such as `<big-integer>` and `<small-integer>`, can have **direct instances.**

As an example, you can use the built-in function `object-class` on this object to find out what class it is a direct instance of:

```
> object-class(13)
#<the class <small-integer>>
```

Similarly, you can use the exponentiation operator to create an object that is a direct instance of a different class:

```
> object-class(13 ^ 13)
#<the class <big-integer>>
```

There are other number classes, such as `<integer>`, that are **abstract classes—** they have no direct instances, but they define commonalities inherited by their subclasses. In this case, the class `<integer>` has only two subclasses: `<big-integer>` and `<small-integer>`.

Some functions, such as the exponentiation operator (`^`) are defined for integers. The default version applies to any integer object—big or small. Other functions, such as the `even?` predicate are defined individually for each of the two subclasses.

The instantiable number classes include:

- `<small-integer>`
- `<big-integer>`
- `<small-ratio>`
- `<big-ratio>`
- `<single-float>`
- `<double-float>`
- `<extended-float>`

These classes are all **sealed**—you import the classes from the Dylan library, and you cannot define subclasses that inherit from them. This limitation allows Apple Dylan to optimize storage and implementation of objects of these classes.

Direct instances (objects) of the classes listed above are immutable—you cannot change the value of a number object. (You can, however, reassign a variable name to a different number object. This does not change the value of the original number object.)

Apple Dylan provides many functions and operators that manipulate number objects (some of which are defined for other classes of objects as well). The next chapter, “Numbers,” examines the instantiable number classes in more detail and discusses some of the built-in functions and operators you can apply to them.

Collection Classes

Collection classes are classes of objects that contain **elements,** which are themselves objects. When it comes to *examining* the elements of a collection, collection classes fall into two categories:

- **Sequences,** which use successive integers (starting from 0) to identify their elements. Examples are vectors and lists. For example, by calling

  ```
  element(*long-list*, 99)
  ```

  you can examine the 100th element (the indexes start at 0) of the list object bound to the variable `*long-list*`.

- **Keyed collections,** which use **explicit keys** to identify elements in the collection. An explicit key may be any object. An example of a keyed collection is a **hash table.** You specify any value as a key and the hash table returns the element associated with that key. For example, if you have a hash table bound to the variable `*my-table*`, then calling

  ```
  element(*my-table*, #"zebra")
  ```

  returns the element in the table associated with the key `#"zebra"`.

The above two categories divide collections according to how you examine their elements. The following categories indicate the extent to which you can *modify* their elements:

- **Mutable collections** allow you to modify their existing elements.
- **Stretchy collections** allow you to add and/or remove elements.

Any particular collection object can be either, both, or neither of these.

Apple Dylan provides many built-in collection classes for you, and a great many more functions and operators that manipulate collection objects and their elements. Chapter 14, "Collections," examines the common instantiable collection classes and discusses the built-in functions and operators you can apply to them. That chapter also describes how to apply functions iteratively over the elements in a collection object.

Additional Topics
Apple Dylan allows you to create your own collection classes, but there are specific guidelines you must follow to satisfy the built-in protocols. See the *Dylan Reference Manual* for details.

User-Defined Classes

Although Apple Dylan provides many powerful built-in classes, there are times when you need to organize information in a manner specially suited to your program.

For this purpose, Apple Dylan allows you to define your own classes of objects.

Class Definitions

The basic mechanism for defining your own class is the class definition. Here is a simple example:

```
define class <phone-message> ( <object> )
    slot who;
    slot when;
    slot message;
    slot urgent?;
end class;
```

This simple class definition defines a class named `<phone-message>`. It specifies that this new class inherits from the basic class `<object>`—and no other superclass. The class defines four slots—named `who`, `when`, `message`, and `urgent?`.

Additional Topics
There are other ways to create new classes. See "Classes As Objects" on page 198. ◆

Slots

Conceptually, **slots** correspond to the fields or instance variables of other object-oriented programming languages. Objects of the same class have the same set of slots, but each object can store its own values in its own slots.

For example, if you create two `<phone-message>` objects, they both contain four slots: `who`, `when`, `message`, and `urgent?`. However, they can each store their own values in these slots. One object might use its `who` slot to store the value `"Your IRS Agent"`, while the other might use its `who` slot to store the value `"Ed McMahon"`.

Typically, a slot is implemented as an object—the value of the slot is simply an object in memory.

Slot Names and Getter Functions

As mentioned earlier, all access to values inside of objects, including slot values, is implemented through function calls. In fact, when you specify a slot name in a class definition, you are actually specifying the name for the function that retrieves the value of the corresponding slot.

IMPORTANT

Slot names are function names. They are bound to function objects that take a single argument—an object—and return the value of the corresponding slot for that object. Since they get the value of a slot, these functions are called **getter functions.** ◆

For example, imagine you have a `<phone-message>` object named `*recent-message*` and you want to determine who called—that is, the value of the `who` slot. You can call the getter function for this slot (which is named `who`), sending it the object whose slot you want to examine:

```
> who(*recent-message*);
"Your sister"
```

This example shows that `who` is a function; it examines its argument, the object `*recent-message*`, and returns the value of its `who` slot.

Apple Dylan provides another special syntax—the **slot reference syntax**—for this type of function call. Here is an example:

```
*recent-message*.who
```

This notation is simply shorthand for

```
who(*recent-message*);
```

Other Slot Specification Options

Class definitions allow you to specify many attributes of each slot. You've already seen one example—the name of the getter function for the slot. Here is a partial list of the other slot options you can specify when defining a class; in particular, you can specify

- whether a corresponding **setter function** should be created to allow the slot value to be modified
- whether the getter and setter functions can be specialized (overridden) by definitions in other modules
- whether the value of the slot should be limited to objects of a particular class
- whether the slot is actually stored as an object or is simply calculated and returned by the setter function
- what the initial value for the slot should be when new instances of the class are created
- whether the `make` function should accept an extra argument that allows you to specify the initial value for the slot when you create a new instance of the class

Additional Topics

All of these options, and more, are discussed in Chapter 15, "Defining Classes." However, Apple Dylan provides many types of slots and slot options not covered in this book. See the *Dylan Reference Manual* for complete details. ◆

Additional Topics

This chapter provides a brief overview of objects and their classes. The following few sections provide some more details as well as pointers to more information.

Object Creation and Manipulation

Each Apple Dylan object is a direct instance of exactly one class.

- **Instantiation**

  You instantiate a class using the `make` function, or one of the built-in object-creation functions. See Chapter 14, "Collections," and Chapter 15, "Defining Classes," for examples.

- **Mutability**

  Some functions allow you to modify their contents. Some don't. See Chapter 13, "Numbers," for examples of immutable objects and Chapter 14, "Collections," for examples of mutable objects.

- **Variables and Accessibility**

  In Chapter 5, "Variables," you learned that you could associate a variable name with an object within the scope of a module. Chapter 18, "Modules," discusses how you can share variables, and therefore objects, between modules. In particular, that chapter discusses **accessibility—**the amount that an object can be examined and/or modified in modules other than the one in which it was defined.

Other Built-In Classes

Chapter 10, "Numbers," and Chapter 12, "Collection Classes," discuss some of the built-in classes. You can also find information on these classes:

- **Condition Classes**

  Chapter 20, "Conditions," for details about classes used for exception-handling.

- **Function Classes**

  Functions are objects, too, so they have classes, just like other objects. See Chapter 9, "Direct Methods," for some examples of function objects, but see the Dylan Reference Manual for complete information.

- **Type Classes**

  See "Types" on page 198 for additional information, and pointers to additional information, about type classes.

Class Inheritance

- **Inheritance**

  Apple Dylan classes are allowed to inherit properties from other classes. Typically, they inherit slots as well as functionality through the method dispatch system. See Chapter 16, "Inheritance," for an introduction to inheritance.

- **Hierarchies**

  Inheritance causes classes to be arranged in a hierarchy. Some classes are **superclasses** of other classes; those classes are called their **subclasses.** See Chapter 16 for more information.

- **Heterarchies**

  Apple Dylan provides **multiple inheritance**—classes can inherit from more than one class. As a result, the class hierarchy is actually a **heterarchy**—a directed acyclic graph of class inheritance.

  Classes can be **primary** or **free;** these attributes affect how the class can be used for multiple inheritance. See Chapters 15 and 16 for some introductory material, but the *Dylan Reference Manual* contains the definitive information.

- **Class Dynamism**

  Class dynamism refers to the ability to control how classes are subclassed outside of their own module. In particular, a class can be **sealed,** which indicates that no subclasses of this class can be defined outside of the library in which the class is defined. Or a class can be **open,** which indicates that subclasses can be defined in other libraries

  These features of classes are introduced in Chapter 18, "Modules."

Method Dispatch

- **Method Dispatch**

  Generic functions compare the classes of actual arguments with the type specifiers in method argument lists to determine which method to call.

- **Specific Method Dispatch**

  Since some classes are subclasses of other classes, method dispatch takes into account the specificity of a type specification; subclasses are more specific than their superclasses.

- **Class Precedence**

  Heterarchies and complex argument lists cause intricate, and sometimes ambiguous, situations during method dispatch. The class precedence system helps to resolve these difficulties. See the *Dylan Reference Manual* for details. ◆

Types

- **Classes As Objects**

  Classes, like the class <integer>, or the class <character>, are represented during runtime as objects—just like other objects. As a result, you can operate on classes during runtime, sending them to functions, binding them to variable names, and so on.

- **The <class> Class**

  Since classes are objects, they each have to have a specific class. For example, the class <integer> is an object at runtime. That object is an instance of some class: it is an instance of the class named `<class>`. (Not a direct instance, though; there is an entire hierarchy of class-related classes!).

- **Types**

  Classes are one kind of **type.** There are other kinds as well. The most common is the singleton, but there are also union and limited types.

- **Singletons**

  Whereas a class is a type that identifies an entire category of objects, a **singleton** is type that specifies a specific object. Singletons are most frequently used in type specifiers in function argument lists. See Chapter 15, "Defining Classes," for an introduction.

- **Union and Limited Types**

  Other kinds of types allow you more control over method dispatch, but the discussion of these advanced types is left to the *Dylan Reference Manual*.

CHAPTER 12

Objects and Classes

CHAPTER 13

Numbers

Contents

Key Concepts 203
About Numbers 204
Real Numbers 205
  Integers 205
  Ratios 206
  Floating-Point Numbers 207
Numeric Functions 207
  Predicates 208
  Arithmetic Functions 209
  Rounding and Division 210
  Conversion and Simplification 212
  Logical Bit-Oriented Functions 213
  Comparisons 214
Operators 215
  Operator Calls 215
  Unary Operators 217
  Binary Operators 218
  Special Operators 219
  Operator Precedence and Association List 220
Additional Topics 221
  Operator Applicability 221

This chapter introduces the most common built-in classes for manipulating numerical information. It also discusses the built-in functions that apply to numbers, and discusses operators in some detail.

Key Concepts

About Numbers

- Apple Dylan provides many **number classes** that you can use to create objects representing numerical values.
- This chapter focuses on classes of **real numbers.**

Real Numbers

- **Real numbers** include **rational numbers** and **floating-point numbers.**
- Rational numbers are further divided into **integers** and **ratios.**
- Apple Dylan provides **concrete classes** that you can use to create objects that represent integer values, ratio values (fractions), and floating-point values.

Numeric Functions

- Apple Dylan provides many **built-in functions** for manipulating number objects.
- This chapter gives an overview of most of these functions, including arithmetic functions, rounding functions, logical functions, and so on.

Operators

- **Operators** are special functions that you can call using the special **operator syntax.**
- Since the operator syntax is infix, rather than prefix, there is a **precedence** and **associativity** of operators.
- There are three **special operators** which are not functions; the **assignment operator** is one of them.

About Numbers

Apple Dylan represents numerical values as instances of the built-in class `<number>`—not direct instances, however, as the class `<number>` is an abstract superclass. It has many subclasses specialized for the different kinds of numerical values your program may need to manipulate.

The only direct subclass of `<number>` is `<complex>`, which is also an abstract superclass. This class represents the common attributes of all complex numbers; it has one subclass: `<real>`.

This chapter focuses on the different classes of real numbers and the functions you can use with them. See the *Dylan Reference Manual* for information about other complex numbers.

The class `<real>` represents real numbers, which Apple Dylan divides into these subcategories:

- **Rational numbers,** represented by the class `<rational>`, which include
  - **Integers,** represented by the class <integer>
  - **Ratios,** or fractions, represented by the class <ratio>
- **Floating-point numbers,** represented by the class `<float>`

The next section of this chapter, "Real Numbers" on page 205, examines these classes of numbers in more detail. In particular, it discusses instantiable subclasses of `<integer>`, `<ratio>`, and `<float>` that you can use to create real number objects.

The other two main sections of this chapter concern the built-in functions Apple Dylan provides for you to manipulate number objects.

"Numeric Functions" on page 207 gives examples of each kind of built-in numeric function.

"Operators" on page 215 discusses the built-in operators, the special operator syntax, operator precedence and association, and briefly discusses how many of these operators apply to other classes of objects (non-numerical classes) as well.

Real Numbers

Real numbers are objects that represent integers, fractions, or floating-point values. The next few sections discuss these classes of values.

Integers

The class `<integer>`, which inherits from `<rational>`, is abstract and sealed. You cannot instantiate it, and you cannot create your own subclasses of it. To use this class, and create integer objects, you must use one of the two built-in subclasses of `<integer>`: `<small-integer>` and `<big-integer>`.

Integers are divided into these two subclasses for optimization purposes: small integers are stored as immediate values and can be manipulated very quickly, while big integers are not necessarily immediate values, and so can require more processing time.

Apple Dylan automatically creates the appropriate class of object for you when you specify an integer literal or create an integer by calling a function. For example:

```
> define variable *changes-class* = 20;
defined *changes-class

> object-class(*changes-class*);
#<the class <small-integer>>

> *changes-class* := *changes-class* ^ *changes-class*
104857600000000000000000000

> object-class(*changes-class*);
#<the class <big-integer>>
```

You can also specify integer literals in binary, octal, or hexadecimal:

```
#b10001000
#o210
#x88
```

You cannot use the function `make` to create small integer objects or big integer objects; you must use literal constants or other expressions that return integer values.

Like all numbers, integers are immutable. Apple Dylan provides no mechanism for you to change the value of a number object. (You can use the assignment operation to assign a variable name to a new number object, but you cannot change the value of a number.)

Ratios

Ratios are fractions of two integers. Like the class `<integer>`, the class `<ratio>` inherits from `<rational>` and is abstract and sealed. You cannot instantiate it, and you cannot create your own subclasses of it. To use this class, and create ratio objects, you must use one of the two built-in subclasses of `<ratio>`: `<small-ratio>` and `<big-ratio>`.

Ratios are divided into these two subclasses for the same reason that integers are: optimization. Small ratios—numerator and denominator—are stored as a single immediate value and can therefore be manipulated quickly, while big ratios are not necessarily stored as immediate values, and so can require more processing time.

Apple Dylan automatically creates the appropriate class of object for you when you specify an ratio literal or create a ratio by calling a function. In fact, Apple Dylan automatically simplifies when the ratio you specify is an integer. For example:

```
> object-class( 10000/1001 );
#<the class <big-ratio>>

> object-class( 1000/1001 );
#<the class <small-ratio>>

> object-class( 10000/1000 );
#<the class <small-integer>>
```

Remember that ratio literals do not include any whitespace around the division (`/`) punctuation. If you include whitespace, the ratio literal becomes a function call.

Floating-Point Numbers

Although the rational numbers allow for extended calculations with minimal—virtually no—error propagation, the floating-point numbers can allow for faster calculations, though rounding errors can compound.

Apple Dylan offers three subclasses of the sealed, abstract class <float>:

- `<single-float>`, which represents single precision floating-point numbers
- `<double-float>`, which represents double precision floating-point numbers
- `<extended-float>`, which represents extended precision floating-point numbers

Here are some examples of floating-point literal constants:

```
1.5
-1.5
1.05
-1.05
.5
-.5
1.5E20
-1.E-20
```

The number after the `E` notation indicates the power of 10 by which to multiply the number before the `E`.

Numeric Functions

Apple Dylan provides a number of functions exclusively for the number classes, as well as polymorphic functions that apply to other classes of objects as well. The next few sections survey the kinds of functions provided for your use.

Predicates

The first set of functions are predicate functions—they test a number for a certain property and return a Boolean value indicating whether the number passed the test.

First, there are the `odd?` and `even?` functions which apply only to integers and test whether the integer is evenly divisible by 2.

Note
These are functions, not open generic functions—that is, you cannot specialize or augment these functions by adding new methods for different argument types. ◆

Here are some examples:

```
> odd? (13)
#t

> even? (0)
#t

> even? (-1)
#f
```

The next two functions are the `positive?` and `negative?` functions, which apply to any real number. These are open generic functions—you can add additional methods to them to apply to other classes of arguments. The built-in versions of these functions determine if argument value is greater than or less than the value 0. Here are some examples:

```
> positive? (13)
#t

> negative? (0)
#f

> positive? (-10/11)
#f
```

The final two predicate functions are zero? and integral? They apply to any number objects, even complex numbers. They are open generic functions, so you can extend their functionality to your favorite classes. Here are some examples:

```
> zero? (13)
#f

> integral? (1/2)
#f

> integral? (+1.0E1)
#t
```

Arithmetic Functions

The next set of numeric functions are the common arithmetic functions. The first five of these are addition (`\+`), subtraction (`\-`), multiplication (`\*`), division (`\/`), and exponentiation (`\^`). Although these function names may look strange, they operate like any normal function:

```
> \+ (3, 4);
7

> \- ( \*(3, 4), \/(4, 2) );
10
```

Of course, you'll typically use these functions with the more familiar operator syntax:

```
3 + 4
3 * 4 - 4 / 2
```

So these function will be discussed in more detail in "Binary Operators" on page 218. All five of these are predefined to work on any number object, but they are open generic functions, so you can extend them to other classes as you like.

The functions `negative` and `abs` allow you to manipulate a number's sign. Again, they are defined for numbers, but you may add implementations. Here are some examples:

```
> negative ( 4/3 );
-4/3

> abs ( -3.1415 );
3.1415
```

The final two functions are `gcd` and `lcm`. These functions apply to integer arguments only, and they each return an integer value as a result. They are not extensible. The first finds the greatest common divisor of two integers, and the second finds the least common multiple:

```
> gcd ( 10, 5 );
5

> lcm ( 4, 6 );
12
```

Rounding and Division

Apple Dylan provides four functions that separate a real number into an integer part and a remainder part. These (sealed) functions are `floor`, `ceiling`, `round`, and `truncate`. All of these functions return two values.

The `floor` function returns the nearest integer that is strictly less than its input. The remainder returned is the difference between the two—a nonnegative number less than 1.

The `ceiling` function returns the nearest integer that is strictly greater than its input. The remainder returned is the difference between the two—a nonpositive number greater than -1.

The `round` function returns the integer nearest its input. If the input lies exactly halfway between two integers, the even integer is returned. The remainder returned is the difference between the two—a number between -1/2 and +1/2.

The `truncate` function returns the nearest integer that is closer to 0. The remainder returned is the difference between the two—a number between -1 and +1.

In all four cases, the integer value returned added to the remainder value returned equals the original number.

Here are some examples:

```
> floor ( -5/2 );
-3
1/2

> ceiling ( -5/2 );
-2
-1/2

> round ( -7/2 );
-4
1/2

> truncate ( -7/2 );
-3
-1/2
```

Apple Dylan also provides versions of these four functions that divide two real arguments; they each round the result of the division as described above and also return a remainder. In these cases, the remainder is the difference between the first argument and the integer returned times the second argument.

In other words: *argument-1* - (*argument-2* * *integer-part*) = *remainder*

Here are some examples:

```
> floor/ ( -5, 2 );
-3
1

> ceiling/ ( -5, 2 );
-2
-1
```

```
> round/ ( -7, 2 );
-4
1

> truncate/ ( -7, 2 );
-3
-1
```

The final two function in this section are based on earlier functions: the `modulo` function returns the remainder as determined by the `floor/` function, and the `remainder` function returns the remainder as determined by the `truncate/` function:

```
> modulo ( -5, 2 );
1

> remainder ( -7, 2 );
-1
```

Conversion and Simplification

The `rationalize` function converts a floating-point number to a rational number.

The `numerator` and `denominator` functions simplify a rational number and return the simplified numerator and denominator, respectively. Here are some examples:

```
> numerator ( 20/15 );
4
> denominator ( 20/15 );
3
```

All of these functions are open generic functions—you may add your own methods.

Logical Bit-Oriented Functions

The functions `logior`, `logxor`, and `logand`, take any number of integer arguments and perform bitwise logical operations on them, returning the integer that results from the bitwise operations. The first function, `logior`, returns an integer whose binary value has any bit set that was set in any of the arguments. The second, logxor, performs a bitwise exclusive-or and returns the resulting integer. The third, performs a bitwise and operation and returns the resulting integer. Here are some examples, shown in binary for clarity:

```
> logior( #b1110, #b1100, #b1000 );
14      /* #b1110  */

> logxor( #b1110, #b1100, #b1000 );
10      /* #b1010  */

> logand( #b1110, #b1100, #b1000 );
8       /* #b1000  */
```

The `lognot` function inverts every bit in its input, ensuring the result has as many bits as the input. The result is equal to *`-input - 1`*. For example:

```
> lognot( 10 );
-11

> lognot( -10 );
9
```

The `logbit?` function tests if a specified bit in an integer is set. The first argument to this function specifies the bit (starting with 0 and going from right to left):

```
> logbit? ( 0, #b1110 );
#f

> logbit? ( 1, #b1110 );
#t

> logbit? ( 2, #b1110 );
#t
```

```
> logbit? ( 3, #b1110 );
#t
```

Finally, the `ash` function performs a bitwise shift. You specify an integer and the number of bits to shift:

```
> ash ( #b11110000, -3 );
30          /* #b1110  */
```

The result is the `floor(`*argument-1* `* 2^`*argument-2*`)`.

All of these bitwise functions are sealed.

Comparisons

The next set of numeric functions are the comparison operations, which strictly speaking, apply to many more classes than the number classes. These functions are not (`\~`), identical (`\==`), not-identical (`\~==`), equal (`\=`), not-equal (`\~=`), less-than (`\<`), greater-than (`\>`), less-than-or-equal-to (`\<=`), and greater-than-or-equal-to (`\>=`). Although these function names may look strange, they operate like any normal functions:

```
> \= (3, 4);
#f

> \>= ( 3 + 3, 1 );
#t
```

You'll typically use these functions with the more familiar operator syntax:

```
3 = 4
3 + 3 >= 1
```

As a result, these function are discussed in more detail in "Binary Operators" on page 218.

The following of the comparison operators are open generic functions; that is, you can specialize them:

- `\=`
- `\<`

All of the others are either implemented by Apple Dylan (\~ and \==), or they are implemented as some combination of these four. Therefore, if you specialize \= and \< for one of your classes, you get the others for free.

There are two more functions that compare their arguments: the min function and the max function. These sealed functions call the \< function to find the minimum, or maximum, value of their arguments, respectively. They each require one argument, but can take any number greater than that:

```
> min (3, 4, 5, -10);
-10

> max(5);
5
```

Operators

Apple Dylan also provides a special syntax for operator functions to make arithmetic expressions look more natural. This infix notation is called the **operator syntax.** It is only allowed for certain functions, called operators, and for a few special operators that are not functions.

Operator Calls

The **operator call syntax** is

{ *expression* } *operator expression*

Here are some simple examples:

```
> 3 + 3
6

> 1/2 * 1/2
1/4
```

```
> max(3,4) / min(3,2)
2
```

Notice the first expression is optional (for some operators):

```
> - 1/2
-1/2
```

Note
Whitespace is significant with operators: operators and their arguments must be separated by whitespace or parentheses. As an example, the expression `- 3` is a unary operator call, while the expression `-3` is a literal constant. ◆

Since operator calls are expressions, you can use operator calls as inputs to other operator calls. In Apple Dylan, you can always surround an operator call with parentheses to explicitly control the order of execution. For example:

```
> ( (4 - 3) * ( - 1) ) + (6 / 3)
1
```

Here the parentheses specify which operator calls to evaluate first, which next, and so on. However, you can create ambiguous stretches of operator calls:

```
> 4 + 5 * 3 ^ 4 - 3 + - 2 * 5
396
```

Where did that number come from? Apple Dylan has a default precedence for each operator if you don't specifically override it with parentheses. In the above example, the `- 2` is evaluated first, then the `3 ^ 4`, leaving:

```
4 + 5 * 81 - 3 + -2 * 5
```

then the `5 * 81` and the `-2 * 5` are evaluated, leaving:

```
4 + 405 - 3 + -10
```

then, these additions and subtractions are evaluated from left to right:

```
409 - 3 + -10
```

```
406 + -10

396
```

Precedence indicates the order in which different operators are evaluated. **Associativity** indicates the direction in which operators of the same precedence are evaluated.

For example, multiplication has a higher precedence than subtraction, so Apple Dylan begins to evaluate this expression

```
100 - 5 * 5 - 10
```

by evaluating the multiplication first, leaving:

```
100 - 25 - 10
```

Now all the operators are subtractions! Subtraction has the same precedence as subtraction, so which subtraction is evaluated first? It makes a difference. If the right subtraction is evaluated first, you have

```
100 - 15
85
```

But if the left subtraction is evaluated first, you have

```
75 - 10
65
```

Well, subtraction associates from left to right, so the result is 65.

See “Operator Precedence and Association List” on page 220 for more information.

Unary Operators

The **unary operators** are

- the negative operator (-)
- the logical-not operator (~)

The negative operator finds the additive inverse of its argument. The logical-not operator returns `#t` if its argument is `#f`. Otherwise, it always returns `#f`.

Here are some examples of unary operator calls:

```
> - 3
-3

> ~ 3
#f

> ~ #f
#t
```

Binary Operators

The **binary operators** are

- the **arithmetic operators:**
  - exponentiation (`^`)
  - multiplication (`*`) and division (`/`),
  - addition (`+`) and subtraction (`-`)
- the **comparison operators:** identity (`==`), equality (`=`), not-identity (`~==`), inequality (`~=`), less-than (`<`), less-than-or-equals (`<=`), greater-than (`>`), and greater-than-or-equals (`>=`).

Here are some examples of binary operator calls

```
> 3 ^ 2
9

> -2 * 1/3
-2/3

> 5 == 5
#t

> 1/2 >= 5
#f
```

Special Operators

The unary operators and the binary operators are functions—when called, they always evaluate all of their argument expressions first, and then execute on the resulting argument values.

There are three special operators that are not functions—they have their own special rules for evaluation.

The **special operators** are

- logical-and (`&`) and logical-or (`|`)
- assignment (`:=`)

Logical-and and logical-or evaluate their first argument, and depending on the result, they evaluate their second argument. For example

```
> 5 = 5 | report-problem()
#t
```

The `|` operator evaluated its first argument, `5 = 5`, determined that it was `#t`, and so it returned `#t` immediately. The second argument, the call to `report-problem`, was never executed.

When multiple logical-or operators are lined up, evaluation progresses until the first non-*false* result:

```
> #f | 5 = 6 | "hello" | shut-down()
"hello"
```

The first non-false argument value is `"hello"`, so that value is returned and the last expression never executed.

The logical-and (`&`) operator is similar, except it stops at the first `#f` value.

Assignment is a very special operator. It interprets the argument on its left-hand-side argument as a variable name (or a slot or an element inside an object), and it sets the value of the variable (or slot or element) to the value of the expression on its right-hand side:

variable-name `:=` *expression*

Here, *variable-name* is not evaluated normally (which would result in the value of the variable), but it is interpreted as a variable name. However, *expression* is evaluated normally.

There are a few special syntaxes that involve the assignment operator. For example:

object[*index*] := *expression*

is interpreted as

element-setter(*expresssion*, *object*, *index*)

and the special syntax:

object.*slot* := *expression*

is interpreted as

slot-setter(*object*, *expression*)

Additional Topics
The assignment operator is used in many import syntaxes and you can find examples all throughout this book. See "Assignment Operator" on page 221. ◆

Operator Precedence and Association List

Operators are evaluated in the order listed in the previous sections; that is:

1. unary operators are evaluated first
2. followed by the exponentiation operator (which associates from right to left)
3. followed by multiplication and division operators (left to right)
4. followed by the addition and subtraction operators (left to right)
5. followed by the comparison operators (left to right)
6. followed by the special logical operators (left to right)
7. followed by the assignment operator (right to left)

Notice that the exponentiation and the assignment operators associate from right to left. As an example, consider the expression:

```
c := b := a
```

This expression is evaluated from right to left: first `b` is assigned to the value of `a`, and then `c` is assigned to the value of `b`.

Additional Topics

This section provides some additional information and also some pointers to additional information in other chapters and books.

Operator Applicability

Many of the operators discussed in this chapter apply to other classes of objects. In particular:

- **Collection Classes**

  Many collection classes rely on the comparison operators. See Chapter 14, "Collections," for an introduction, but see the *Dylan Reference Manual* for details.

- **Assignment Operator**

  This operator is important and used in many different situations in Apple Dylan programs—variable reassignment, slot modification, element modification, even resizing of collection objects. Examples exist throughout this book, but see the *Dylan Reference Manual* for complete details.

CHAPTER 13

Numbers

CHAPTER 14

Collections

Contents

Key Concepts 225
About Collections 226
  Kinds of Collections 226
  Collection Functions 227
  Kinds of Sequences 228
  Sequence Functions 228
Strings 229
  Byte Strings 230
  Stretchy Byte Strings 231
  String Functions 233
Vectors 234
  Simple Object Vectors 235
  Vector Functions 237
Lists 238
  Pairs 239
  List Functions 241
Additional Topics 242

CHAPTER 14

This chapter introduces the three most common kinds of collections and how you can use the collection functions to manipulate them.

Key Concepts

About Collections

- A **collection** is an object that conceptually represents a collection of other objects, called **elements.**
- **Collection functions** allow you to determine information about a collection and its elements, and to manipulate them.
- A **sequence** is a kind of collection whose elements are indexed by consecutive nonnegative integers, starting with 0.
- **Sequence functions** complement the collection functions to allow you even more control over sequence objects.
- The three most common kinds of sequences are strings, vectors, and lists.

Strings

- A **string** is a collection of characters. String objects are **mutable**—you can change the characters in them. Some kinds of string objects are **stretchy**—you can change their size by adding more characters.

Vectors

- A **vector** is a linear array of objects of any class. Like strings, they can be mutable and/or stretchy.

Lists

- A **list** is a linked list of objects.
- Lists are implemented using **pair** objects as the nodes of the linked lists.

About Collections

A **collection,** or a collection object, is an object that you use to organize and manipulate groups of objects.

A collection object contains **elements**, which are themselves objects. Apple Dylan provides many different kinds of collections; each organizes and accesses its elements in a slightly different way. Apple Dylan also provides a number of built-in functions you can use to manipulate collections and their elements. For example, you can iterate over the elements in a collection, you can map a function across the elements of a collection, you can randomly access and modify collection elements, you can sort them, you can search through them, you can filter them, and so on.

The next section introduces the basic kinds of collections available to you, and the section immediately after that introduces collection functions.

Kinds of Collections

Collections contain elements. Some collections, called **sequences,** use sequential nonnegative integers as indexes to their elements. A vector, for instance, is a kind of sequence; its elements are numbered 0, 1, 2, 3, and so on. An index uniquely identifies the element in the sequence; you can use the index to randomly access sequence elements.

Some collections, called **keyed collections,** allow you to associate any values—called **keys**—with their elements. A hash table is an example of a keyed collection. You specify elements in the table by specifying a key which uniquely identifies the element.

This chapter focuses on sequences, because they are simpler and more common than keyed collections.

Additional Topics

Although they are not covered in this chapter, keyed collections, and hash tables in particular, are powerful collection objects available to you. See “Hash Tables” on page 243. ◆

There are a two important properties that apply to the different classes of collections:

- If a collection is **mutable,** you can modify its existing elements.
- If a collection is **stretchy,** you can add or remove elements.

In this chapter, you'll learn about sequences in general, and specifics about the three most important kinds of sequences and how you can use them.

Collection Functions

Collection functions allow you to operate on any kind of collection object. You can use them to determine information about a collection or its elements, to iteratively execute some procedure over its elements, to search through its elements, and so on. Here are some of these important and useful built-in functions:

- The `element` function returns the value of a specified element.
- The `element-setter` function, which works on mutable collections, sets the value of a specified element.
- The `empty?` function determines if a collection contains any elements.
- The `size` function returns the number of elements in a collection.
- The `member?` function determines whether a collection contains a particular element.
- The `do`, `map`, `map-as`, `map-into`, `reduce`, and `reduce1` functions iteratively apply functions to the elements of one or more collections.
- The `any?` and `every?` functions iteratively perform tests over the elements of one or more collections.

Additional Topics

Some of these functions are shown in examples in this chapter, but for complete information on collection functions, see the *Dylan Reference Manual*. ◆

Kinds of Sequences

Sequences are collections that index their elements using sequential nonnegative integers. All sequences share this property.

There are different kinds of sequences, however, that have different attributes, can be used in different ways, and have been optimized for different purposes.

In this chapter, we'll focus on three kinds of sequences:

- **strings,** which are collections of characters
- **vectors,** which are linear arrays of objects
- **lists,** which are linked lists of objects

Additional Topics
Other kinds of sequences are available, including multi-dimensional arrays, deques, and ranges. See "Additional Topics" on page 242. ◆

Sequence Functions

Sequence functions allow you to operate on any kind of sequence object—in particular, you can use them on the strings, vectors, and lists you'll see later in this chapter.

You can use sequence functions to determine information about a sequence or its elements, to manipulate a sequence or its elements, or to modify a sequence. Some examples are:

- The `size-setter` function allows you to change the size of stretchy sequences. This function can be invoked using the special syntax:

  `size(`*collection*`) :=` *new-size*

- The `add`, `add-new`, `remove`, and `remove-duplicates` functions allow you to create new sequences that are variations of existing sequences.
- The `add!`, `add-new!`, `remove!`, and `remove-duplicates!` functions allow you to alter existing stretchy sequences.
- The `choose` and `choose-by` functions allow you to create new sequences that contain elements from existing sequences that satisfy specified criteria.

- The `intersection` and `union` functions allow you to create new sequences that combine elements from existing sequences.
- The `reverse` and `sort` functions allow you to create new sequences which are reordered versions of existing sequences.
- The `reverse!` and `sort!` functions allow you to reorder items in a mutable sequence.
- The `first`, `second`, `third`, and `last` functions allow you to access particular elements of a sequence.

Additional Topics
Some of these functions are shown in examples in this chapter, but there are many more functions available. See the *Dylan Reference Manual* for complete information. ◆

Strings

Strings are sequences of characters—the elements of string objects are character objects. (Don't worry about extra object overhead, however,—Apple Dylan does optimize the internal representation used for string objects.)

The abstract class `<string>` is the superclass of all string classes. All of the built-in string classes define mutable sequences—that is, you can modify existing elements in a string object.

For optimization purposes, not all strings are stretchy, however. Some classes of string objects do not allow you to resize them once they are initialized, while others do.

The `<string>` class is both abstract and instantiable, which means that when you attempt to instantiate it using the `make` function, the object returned is actually not a direct instance of `<string>`, but rather it is a direct instance of one of the concrete subclasses of `<string>`.

The most common concrete subclasses of `<string>` is `<byte-string>`. The rest of this section on string objects examines byte strings in more detail.

Additional Topics

Standard Dylan also defines the `<unicode-string>` classes for unicode strings. See the *Dylan Reference Manual* for details.

Apple Dylan extends the Standard Dylan language by adding still more string classes: `<C-string>` and `<Pascal-string>`. See Chapter 21, "C-Compatible Libraries," and the *Apple Dylan Extensions and Framework Reference*. ◆

Byte Strings

A byte string is a sequence of characters that each require one byte (8 bits) of storage. When you instantiate the <string> class, Apple Dylan returns a byte string:

```
> define constant $status = make(<string>);
defined $status

> object-class($status);
#<the class <byte-string>>

> $status
""
```

Since the class `<byte-string>` is not stretchy, there is no way to modify the string bound to the `$status` constant—it has no elements to modify, and none can be added.

You can specify an initial size when instantiating strings:

```
> define constant $message = make( <string>, size: 10 );
defined $message

> object-class($message);
#<the class <byte-string>>

> $message
"          "
```

Since byte strings are mutable, you can now edit the elements of this string:

```
> $message[0] := $message[9] := '!';
'!'
```

Remember, assignment associates from right to left and returns the value of the right-hand expressions. Therefore, the above expression:

1. assigns the value `'!'` to the last element of the string (element number 9)
2. returns the value `'!'` as the result of that assignment
3. assigns that value to the first element of the string (element number 0)

As a result, the string object has been modified:

```
> $message
"!        !'
```

Apple Dylan also creates byte strings when you specify string literal constants. For example:

```
> define constant $message = "we are all going down";
defined $message

> object-class($message);
#<the class <byte-string>>
```

This class of objects, as you have seen, is mutable; but it is not stretchy. You can modify characters in a byte string, but you cannot add characters to it.

Stretchy Byte Strings

However, Apple Dylan does provide the `<stretchy-byte-string>` class to allow you to create stretchy strings. For example,

```
> define variable new-message = as(<stretchy-byte-string>, $message);
defined new-message
```

The `add!` function allows you to add a single new element to the end of a stretchy string:

```
> add!(new-message, ' ');
#<<strectchy-byte-string> "we are all going down ">
```

You could use a `for` statement to iteratively add more elements, one at a time:

```
> for (char in "to the store")  add!($new-message, char);  end for;
#f

> new-message
#<<strectchy-byte-string> "we are all going down to the store">
```

You can also add new elements all at once:

```
> size(new-message) := size($new-message) + 5;
39
```

And then modify the new elements all at once:

```
> new-message := replace-subsequence!(new-message, " soon",
                                      start: 34, end: 38);
#<<stretchy-byte-string> "we are all going down to the store soon">
```

Note, however, that you do not have to destructively modify the same string object to manipulate strings. If you define a variable instead of a constant, you can achieve the same results more easily. More objects are created and garbage collected in the process, but Apple Dylan handles that for you. Here is an example:

```
> define variable *message* = "we are all going down";
defined *message*

> object-class(*message*);
#<the class <byte-string>>
```

Although the object bound to the variable name `*message*` is *not* stretchy, the variable is reassignable, so:

```
> *message* := concatenate (*message*, " to the store soon");
"we are all going down to the store soon"
```

```
> *message*
"we are all going down to the store soon"
```

The variable `*message*` is thus bound to a new byte string object. This object is larger than the previous value of message, but it is still not stretchy—`*message*` changed it's binding; it did not stretch its original object.

String Functions

The previous two sections showed you some of the functions you can use on string objects, including:

- string creation and initialization with the `make` function
- string class conversion with the `as` function
- element access with the `element` function (using the `[]` notation)
- destructive editing with the `size`, `add!`, and `replace-subsequence` functions
- non-destructive editing with the `concatenate` function

All of these functions apply to objects, collections, or sequences in general; they are not specific to strings. There are other general-purpose functions that you might find useful when operating on strings. For example:

```
> define variable *message* = "we are all going down";
defined *message*

> subsequence-position ( *message*, "all" );
7

> subsequence-position ( *message*, "few" );
#f
```

If you simply wanted to know whether or not a string contained a particular substring (as opposed to determining the location of the substring), you could define a predicate function:

```
define method contains-string? ( big-string :: <string>,
                                 search-for :: <string> )

    subsequence-position (big-string, search-for) ~= #f;
end;
```

This predicate function returns either #t or #f, depending on whether subsequence-position found the subsequence or not.

Note
The body of this predicate function could also be implemented as

```
~(~subsequence-position (big-string, search-for));
```

or

```
as (<Boolean>, subsequence-position (big-string, search-for));
```

Either of these expressions guarantee a `#t` or `#f` value returned. ◆

You can also use any other collection, sequence, or vector function on string objects.

There are a few functions defined specifically for string objects. In particular:

- The `as-lowercase` function returns a new string, identical to the original, but with all of the uppercase letters now lowercase.
- The `as-uppercase` function also returns a new string, but it uppercases the original lowercase characters.
- The `as-lowercase!` and `as-uppercase!` functions modify the original string, rather than create a new one.

Vectors

Vectors are similar to strings; they are ordered sequences of elements. Vectors, however, can contain elements of any class—not just characters.

The abstract class <vector> is the superclass of all vector classes. You've already seen some subclasses of <vector>—all of the instantiable string classes are subclasses of <vector>. As with strings, all vector classes are mutable and some vector classes are stretchy.

Like the <string> class, the <vector> class is both abstract and instantiable, which means that when you attempt to instantiate it using the make function, the object returned is actually not a direct instance of <vector>, but rather it is a direct instance of a concrete subclass of <vector>.

Additional Topics
Vectors are actually one-dimensional versions of another class of sequences: multi-dimensional arrays. See "Additional Topics" on page 242. ◆

Simple Object Vectors

The class <simple-object-vector> is the instantiable subclass of vector. When you instantiate the <vector> class, Apple Dylan returns a simple object vector:

```
> make(<vector>);
#[]

> make(<vector>, size: 2);
#[#f, #f]

> make(<vector>, size: 2, fill: 0);
#[0, 0]

> define constant $lotto-guesses = make(<vector>, size: 6, fill: 0);
#[0, 0, 0, 0, 0, 0]

> object-class($lotto-guesses);
#<the class <simple-object-vector>>
```

This class is mutable, but not stretchy:

```
> $lotto-guesses[3] := 37;
37
```

```
> $lotto-guesses
#[0, 0, 0, 37, 0, 0]
```

You can also use vector literal constants to create simple object vectors:

```
> define constant $lotto-guesses = #[0, 10, 20, 30, 40, 50];
defined $lotto-guesses

> object-class($lotto-guesses);
#<the class <simple-object-vector>>
```

To grow or shrink a vector, you can use a variable, rather than a constant:

```
> define variable *lotto-guesses* = #[0, 10, 20];
defined *lotto-guesses*

> *lotto-guesses* := concatenate(*lotto-guesses*, #[30, 40, 50]);
#[0, 10, 20, 30, 40, 50]
```

or you could create a stretchy vector:

```
> define constant $lotto-guesses = make(<stretchy-object-vector>);
defined $lotto-guesses

> $lotto-guesses
#<<stretchy-object-vector> #[]>

> for (count from 0 to 5)
     $lotto-guesses[count] := count;
  end for;
#f

> $lotto-guesses
#<<stretchy-object-vector> #[0, 1, 2, 3, 4, 5]>
```

Notice that the for statement added elements simply by assigning them values. Here's another example:

```
> $lotto-guesses[8] := 8;
8
```

```
> $lotto-guesses
#<<stretchy-object-vector> #[0, 1, 2, 3, 4, 5, #f, #f, 8]>
```

The vector object was automatically stretched to accommodate the element assignment.

So far, you've seen vectors that contain integers and boolean values, but vector elements can be any object:

```
> $lotto-guesses[7] := "hello";
"hello"

> $lotto-guesses
#<<stretchy-object-vector> #[0, 1, 2, 3, 4, 5, "hello", #f, 8]>
```

You can even have a vector of function objects:

```
> define variable *favorite-functions* = make (<vector>, size: 2);
defined *favorite-functions*

> *favorite-functions*[0] := max;
#<the method max (<object>, #rest) at: #xD57B72>

> *favorite-functions*[1] := min;
#<the method min (<object>, #rest) at: #xD57AEA>
```

Vector Functions

This section shows you some of the powerful ways you can use collection functions on vectors.

First, we'll look at the `map` function.

Remember from Chapter 9, "Direct Methods," that you can use the `apply` function to apply a function to a sequence of arguments. For example:

```
> apply (max, #[1, 2, 3, 4]);
4
```

As a result, you can use the `map` function to apply a sequence of functions to a sequence of argument sequences:

```
> map (apply, *favorite-functions*, #[#[10, 20], #[5, 50]]) ;
#[20, 5]
```

The above example called the `apply` function twice, and returned the 2 results in a vector:

```
apply(max, #[10, 20]) // first element of *favorite-functions* => 20
apply(min, #[5, 50])  // second element of *favorite-functions* => 5
```

If you wanted to apply each function in the vector of functions to the same arguments, you could use this function call:

```
> map ( method (f) apply(f, #[1, 2, 3, 4, 5]) end,
        *favorite-functions* );
#[5, 1]
```

This function call applied each function in the `*favorite-functions*` vector to the argument list `#[1, 2, 3, 4, 5]`. The results were returned in a vector.

Some other collection and sequence functions that you might find useful with vector objects are `reduce`, `member?`, `intersection`, `union`, `reverse`, and `sort`. These are just a few examples. See the Dylan Reference Manual for the complete list.

Lists

A list is a similar to a vector in that it contains an ordered sequence of elements. Each element can be an object of any class; and you can reference the elements using indexes starting with 0.

However, vectors and lists differ substantially in their implementations.

Vectors are optimized for random element access—that is, they are designed to minimize the amount of time it takes to access any element. This efficiency, however, comes with a price: either the vector is not stretchy and you cannot add elements, or the vector is stretchy but adding elements requires inefficient copying of the entire vector.

Lists provide the opposite set of trade-offs. A list is implemented as a linked list of nodes. Each node references one element of the list; each node also

references the next node in the list. Therefore, each node contains two references: one to an element of the list, and one to the next node in the list.

With this implementation, lists are not optimized for random element access; accessing elements further down the list requires time to travel down the list of references. However, this implementation optimizes lists for being traversed, such as during iteration or recursion. Lists are also optimized for adding and subtracting elements, since only a few object references must be changed.

Lists are not explicitly stretchy; they are naturally stretchy due to their implementation.

List literals use the `#()` syntax to allow you to specify a list. Commas separate the elements of the list. Some examples of list literals are

```
#(100, 200, 300)    // a list with three numbers
#('A', 100)         // a list with a character and a number
#("hi")             // a list with one string
#(#(1, 2), #(3, 4)) // a list of lists
```

The `<list>` class is the abstract class of all lists. It has two subclasses:

- the `<empty-list>` class, which has one instance—the list with no elements

  ```
  #()                 // the empty list
  ```

- the `<pair>` class, which represents a node in the linked list. The first element of a pair references a list element and the second eliminating of a pair references the next pair (the next node in the list). The final node in a list references the last list element and, instead of reference the next node, references the empty list.

Every list object, then is either the empty list, or a `<pair>` object.

Before discussing the functions you can apply to list objects, the next section discusses the `<pair>` class in more detail.

Pairs

As you've seen, pair objects are used to implement the nodes of a linked list. However, pairs are interesting collection objects in their own right. They

simply allow you to create a collection with two elements: a `head` element and a `tail` element.

Pair literals use the `#( . )` syntax to allow you to specify a pair of values. A period, which must be surrounded by spaces, separates the elements of the pair. An example of a pair literal is

```
#('A' . 'B')      // a pair with two characters
```

You can also create pair objects using the `pair` function:

```
> define variable *name* = pair("Apple", "Dylan");
defined variable *name*
```

And you can access and modify the elements of the pair using the `head` and `tail` functions:

```
> head(*name*);
"Apple"

> tail(*name*);
"Dylan"

> tail(*name*) := "Orchard";
"Orchard"

> *name*
#("Apple" . "Orchard")
```

A pair object is called a **proper list** if its tail element references:

- another pair object that is a proper list, or
- the empty list

For a proper list, every pair in the list must use its head element to reference an element of the list, and its tail element to reference another pair, and so on, until the last pair, which points to the last element in the list and to the empty list.

A simple example of a proper list is a single pair object:

```
#( 4 . #())
```

The head of this pair object references the number 4 and the tail references the empty list. Therefore, this pair object is the list `#(4)`.

To add another element, you can create another pair object:

```
#( 3 . #( 4 . #() ) )
```

This pair is the list `#(3, 4)`.

Continuing in this manner, we have:

```
#( 1 . #( 2 . #( 3 . #( 4 . #() ) ) ) )
```

which is four pair objects, each a node of a proper list. The head of each node references a list element, and the tail of each node references the next node. This linked list of pair objects is the list:

```
#(1, 2, 3, 4)
```

List Functions

Just as you can create a pair with the `pair` function, you can create a proper list with the `list` function:

```
> define variable *frequency* = list('e', 't', 'o', 'a', 'n');
defined variable *frequency*

> *frequency*
#('e', 't', 'o', 'a', 'n')
```

But notice that

```
> object-class(*frequency*)
#<the class <pair>>
```

You can access elements of lists using the `element` function, or the `[]` notation, just as you would for vectors:

```
> *frequency*[2]
'o'
```

```
> *frequency*[4] := 'r'
'r'

> *frequency*
#('e', 't', 'o', 'a', 'r')
```

However, you tend to access list elements by traversing through the list, rather than accessing individual elements. For example, consider the collection function `reduce1`:

```
> reduce1 (\+, #(1, 2, 3, 4, 5) )
15
```

This function applies its first argument to each argument in the indicated sequence, one at a time.

You could implement a similar function with this definition:

```
define method add-em-up ( numbers :: <list> )
    if (empty?(numbers))
        0
    else
        head(numbers) + add-em-up(tail(numbers))
    end if;
end
```

This function takes a list of numbers. If this list is empty, it returns 0. Otherwise, it adds the first number in the list to the total of the rest of the numbers in the list. Of course, it calculates that total recursively, by calling the same function on the tail of the list.

Additional Topics

This chapter introduced the three most common kinds of collections: strings, vectors, and lists. Apple Dylan provides more built-in kinds of collections, and you can define your own.

- **Deques**

  Deques are stretchy sequences that provide functions that allow you to add or remove elements from either end.

- **Ranges**

  Ranges are sequences, but they are not mutable or stretchy. They represent ranges of values.

- **Hash Tables**

  Hash tables are not sequences, but keyed collections. Elements in a hash table are identified by keys, rather than a simple integer index.

For more information about all of these subjects, as well as information about defining your own collection classes, see the *Dylan Reference Manual*.

CHAPTER 14

Collections

CHAPTER 15

User-Defined Classes

Contents

Key Concepts 247
About User-Defined Classes 248
Class Definitions 249
  Adjectives 250
  Class Names 250
  Superclasses 252
  Slot Specifications 252
  Initialization Argument Specifications 254
Slots 256
  Slot Specifications 256
  Slot Accessors 257
  Slot Type Specialization 258
  Slot Initialization 258
  Slot Allocation 262
  Constant Slots and Setter Functions 264
Additional Topics 266
  Class Creation 266
  Instantiation 266

The last three chapters have discussed the built-in classes you can use to create and manipulate objects.

This chapter shows how you can create your own classes of objects.

Key Concepts

About User-Defined Classes

- Classes you define are called **user-defined classes.**
- User-defined classes, like all classes, have certain attributes like **instantiability, inheritance,** and **dynamism.**
- User-defined classes define **slots,** which allow objects to store information, and allow you to access that information.

Class Definitions

- One way to create your own classes is to use the `define class` form of **class definition.**
- A class definition allows you to specify the attributes of the class, information about the class's slots, and information about how objects of the class are initialized.

Slots

- A **slot specification** is the part of a class definition where you specify information about a slot.
- You can specify the type of a slot, and initialization information for a slot.
- **Slot access** is through function calls; slots have **getter functions** and **setter functions.**

About User-Defined Classes

As you learned in Chapter 12, objects are grouped by class—an object's class defines its structure (how it is represented in memory) and its behavior (how functions operate on it).

The built-in classes implement common and useful kinds of objects for you. You can extend the built-in classes by defining your own classes. Classes you define are called **user-defined classes.** They share these attributes:

- **Instantiability.** Classes can be **concrete** or **abstract**. A concrete class is one that is meant to have instances (objects of that class). An abstract class is a class that is not meant to have instances; abstract classes are used for inheritance, as described in the next chapter.

  Classes can also be **instantiable** or **uninstantiable.** An instantiable class is one that you can specify as the first argument to the built-in `make` function. An uninstantiable class is one that you cannot send to `make`.

  Typically, concrete classes are instantiable and abstract classes are uninstantiable. However, there are exceptions:

  - **Uninstantiable concrete classes** (like `<small-integer>`) can have instances; however, you cannot use the `make` function to create these instances.
  - **Instantiable abstract classes** (like `<string>`) cannot have direct instances, but you can send them to the `make` function, which typically returns an instance of one of the class's **concrete subclasses** (like `<byte-string>`).

- **Inheritance.** Inheritance is the ability for a class to inherit structure and behavior from one or more other classes, called **superclasses.** Inheritance is discussed in the next chapter.

- **Dynamism.** Classes can be **sealed** or **open.** In general, an open class can be used as a superclass, while a sealed class cannot. These restrictions apply only across libraries, however. See Chapter 18, "Modules," for more information.

- **Slots.** Instances of user-defined classes store their information in slots. The class defines how many slots, and what kind of slots, its objects have. Each instance of a class has the same slots, but each instance can store its own information in its own slots.

Class Definitions

The most common way to create a new class is to execute a **class definition,** using the built-in `define class` defining form. Here is the syntax for a class definition:

`define` **[** *adjectives* **]** `class` *class-name* `(` *superclasses* `)`

{ *slot-specifier* **|** *initialization-argument-specifier* **}**

`end` **[** `class` **]** **[** *class-name* **]**

As you can see, a class definition has many parts, including *adjectives* which specify different attributes of the class, the *class-name* of the class, and the *superclasses* of the class. The body of the class definition contains a series of *slot-specifiers* and/or *initialization-argument-specifiers*.

Here is a simple example:

```
define class <1D-point> (<object>)
   slot x;
end class;
```

This class definition uses no *adjectives*, the *class-name* is `<1D-point>`, there is one *superclass*: `<object>`, and there is one *slot-specifier*:

```
slot x;
```

There are no *initialization-argument-specifiers* in this simple example.

The next few sections discuss each aspect of class definitions in more detail.

Additional Topics
Class definitions are not the only way to create new classes. For information about other ways to define classes, see "Defining Classes" on page 266. ◆

Adjectives

There are six adjectives you can specify in a class definition; they fall into these three categories:

- **Dynamism.** You can specify a class to be either `sealed` or `open`. The default is `sealed`. These adjectives control whether a class can be subclassed outside of the library in which the class is defined. See Chapter 18, "Modules," for more information.
- **Multiple Inheritance.** You can specify a class to be either `primary` or `free`. The default is `free`. These adjectives control how other classes can use this class as a superclass. See Chapter 16, "Inheritance," for more information.
- **Instantiability.** You can specify a class to be either `abstract` or `concrete`. The default is `concrete`. These adjectives allow you to specify whether or not a class is intended to have direct instances. Chapter 16, "Inheritance," discusses these adjectives.

You may combine adjectives from the different categories. For example, this class definition:

```
define class <1D-point> (<object>)
   slot x;
end class;
```

which uses no adjectives, is equivalent to this class definition:

```
define sealed free concrete class <1D-point> (<object>)
   slot x;
end class;
```

which uses three adjectives (the three default adjectives).

Class Names

In Apple Dylan, classes are represented during runtime by objects in memory.

IMPORTANT

Classes exist during runtime as objects. For example, there is an object representing the `<integer>` class—just as there is an objects representing the value 1. There is an object representing the <string> class, just as there is an object representing the string `"hello"`. The fact that classes are represented as objects is a powerful feature of Apple Dylan. ▲

A **class name** is simply a variable name that is bound to an object representing a class. For example, the variable name `<integer>` is bound to the runtime object that represents the built-in class for integers.

When you define a class, you are creating an object to represent that class. You must supply a variable name to be bound to that class object. This variable name must adhere to the same restrictions as any variable name. (These are described in Chapter 5).

By convention, Apple Dylan class names begin and end with angle brackets:

```
<integer>

<list>

<my-class>
```

This convention is not enforced, however; you can use any valid variable name for your classes. However, it is recommended that you use this convention for consistency and clarity. This convention also helps to avoid naming conflicts, because class names exist in the same namespace as other variable names.

Additional Topics

Since classes are objects, they can exist without being bound to variable names. To create an **anonymous class,** you must use an alternative way of defining the class. See "Defining Classes" on page 266. ◆

Superclasses

In a class definition, you can specify one or more superclasses that your class inherits from. Chapter 16 discusses superclasses, inheritance, and multiple inheritance.

In this chapter, the examples are all **simple classes:** concrete, instantiable classes whose only superclass is `<object>`.

Slot Specifications

The body of a class definition is a series of slot specifications (and/or initialization-argument specifications) separated by semicolons.

Each slot specification specifies a slot for the class. At the very least, a slot specification specifies a name for the slot. For example, consider this class definition:

```
define class <1D-point> (<object>)
    slot x;
end class;
```

This class definition creates a class (that is, a class object) and binds it to the name `<1D-point>`.

This class definition contains one slot specification:

```
slot x;
```

As a result of this slot specification, two more objects are created:

- A method object that takes a `<1D-point>` object as its argument and returns the value of its `x` slot. This method object is added to the generic function named `x`. This is called the **getter function** for the slot.
- A second method object that takes two arguments: a value and a `<1D-point>` object. This method sets the value of the specified object' `x` slot to the specified value. This method object is added to the generic function named `x-setter`. This is called the **setter function** for the slot.

When you execute this simple class definition, then, not only is a class object created, but two method objects are also created. It's as if you executed these two method definitions in addition to your class definition:

```
define method x (input :: <1D-point>)
    // get value stored in x slot of input object
end method;

define method x-setter (new-value, input :: <1D-point>)
    // set value of x slot of input object to new-value
end method;
```

Consider the following example. First, you create a `<1D-point>` object using the `make` function:

```
> define variable *my-1D-point* = make(<1D-point>);
defined *my-1D-point*
```

You can then set the value of the new object's `x` slot using the setter function:

```
> x-setter (10, *my-1D-point*);
10
```

You can determine the value of that slot using the getter function:

```
> x(*my-1D-point*)
10
```

To make slot access more natural, Apple Dylan provides the **slot reference syntax** for calling getter functions:

```
> *my-1D-point*.x
10
```

Apple Dylan also extends the assignment operator so that you can use the following syntax to call the setter function:

```
> *my-1D-point*.x := 20
20
```

So, the simplest of slot specifications:

```
slot x;
```

indicates that your new class contains a slot, and creates the two methods that allow you to reference and to modify the value of that slot in instances of your class.

You can also use slot specifications to specify other information relating to the slot. For example, you can use a slot specification to specify an initialization keyword for the slot. For example:

```
define class <2D-point> (<object>)
   slot x, init-keyword: x:;
   slot y, init-keyword: y:;
end class
```

An **initialization keyword** is a keyword argument that you can specify when creating new objects using the `make` function. For example:

```
define constant $origin = make (<2D-point>, x: 0, y: 0);
```

This call to the `make` function creates a new `<2D-point>` object and initializes its `x` slot to the value 0 and its `y` slot to the value 0.

See “Slots,” beginning on page 256, for many more examples.

Initialization Argument Specifications

Class definitions can also include **initialization-argument specifications,** which specify additional information about initialization keywords. For

example, this class definition contains two slot specifications and two initialization-argument specifications:

```
define class <2D-point> (<object>)
   slot x, init-keyword: x:;
   slot y, init-keyword: y:;

   required keyword x:;
   keyword y:, type: <integer>;
end class
```

The slot specifications specify two initialization keywords (`x:` and `y:`). As you saw in the previous section, when you use the `make` function to create a `<2D-point>` object, you can use the keyword `x:` to provide an initial value for the `x` slot and you can use the keyword `y:` to provide an initial value for the `y` slot.

The initialization-argument specifications provide additional information—not about the slots themselves, but about the corresponding keyword arguments to the `make` function. For example, the initialization-argument specification

```
required keyword x:;
```

specifies that you *must* provide a value for the `x:` keyword when calling the `make` function to create a `<2D-point>` object.

The second initialization-argument specification

```
keyword y:, type: <integer>;
```

specifies that if you use the `y:` keyword when calling the `make` function to create a `<2D-point>` object, the value you provide must be an integer.

IMPORTANT

Note that the `y` slot may contain values that are not integers; type specializations in initialization-argument specifications apply only to the initialization arguments expected by the `make` function—not to the slots themselves. ▲

The section “Slot Initialization” on page 258 provides more examples of initialization-argument specifications.

Slots

Slots are units of storage within objects. The slots that an object has are defined by the object's class. In a class definition, you use slot specifications to define the various slots for instances (objects) of that class.

Slot Specifications

The simplest slot specification specifies only the slot name:

```
slot x;
```

This specification specifies a slot, and a getter and setter function for the slot. You can also use slot specifications to specify the acceptable classes of values that may be stored in a slot. For example, the slot specification

```
slot x :: <integer>;
```

indicates that only integer values may be stored in the x slot. The `x-setter` setter function only allows you to specify integer values when setting the slot. It's as if you defined this method:

```
define method x-setter (new-value :: <integer>, input :: <class-name>)
    // set the value of the x slot of the input object to the new-value
end method;
```

You can also use a slot specification to specify initialization information about a slot; for example:

```
slot x :: <integer>, init-value: 1;
```

This slot specification specifies an initial value of 1 for the `x` slot. When you create new instances of this class, this slot is given the initial value 1 by default.

You can also specify various adjectives that control properties of the slot; for example. The full syntax of a slot specification is shown here:

[*adjectives* **]** **[** *allocation* **]** `slot` *getter-name* **[** `::` *type* **]** **{** `,` *keyword* **}**

The next few sections examine the various aspects of slot specifications, and of slots themselves.

Slot Accessors

In Apple Dylan, all slot access is performed by function calls. (This is in contrast to some other languages where slots are accessed through variable references.)

As you saw in "Slot Specifications" on page 252, the method that returns the value of a slot is called the **getter method**, and the method that sets the value of a slot is called the **setter method**.

(These methods are added to generic functions. When defining a class, you specify slots by specifying the generic functions to which the getter and setter methods should be added. Chapter 17, "Generic Functions," provides more information.)

Normally, the name of the getter method is the name of the slot and the name of the setter method is the name of the slot concatenated with `"-setter"`. For example, consider this class definition for `<2D-point>`:

```
define class <2D-point> (<object>)
  slot x;
  slot y;
end class;
```

You can create an object of class `<2D-point>` using the `make` function:

```
define variable *my-point* = make (<2D-point>);
```

You can set the value of the `x` slot using any of these three notations:

```
x-setter(10, *my-point*);
x(*my-point*) := 10;
*my-point*.x := 10;
```

You can access the value of the x slot using either of these notations:

```
x(*my-point*)
*my-point*.x
```

Slot Type Specialization

Slots may be specialized by declaring the type of the slot in the class definition. For example:

```
define class <2D-point> (<object>)
  slot x :: <integer>;
  slot y :: <ratio>;
end class;
```

Specializing a slot has the following effects on the getter and setter methods of the slot:

- The automatically defined slot getter method has a return value type declaration indicating that it returns a value of the specified class.
- The automatically defined slot setter method has two arguments: the new-value argument is specialized on the type specified for the slot. Therefore, when you use this setter method, you can only set the value of the slot to values of the appropriate type.

The next section shows some examples.

Slot Initialization

Apple Dylan provides a number of options you can specify in a slot specification that control how the slot is initialized.

Consider the <2D-point> class definition:

```
define class <2D-point> (<object>)
  slot x;
  slot y;
end class;
```

When you create an instance of this class using the `make` function, the slots have no initial values. They are **uninitialized slots.**

You can provided an initial value for a slot, as shown in this example:

```
define class <2D-point> (<object>)
  slot x, init-value: 0;
  slot y, init-value: 0;
end class;
```

You can use any expression with the `init-value:` keyword. This expression is evaluated *when the class is defined*. For example:

```
define variable *global* = 0;

define class <3D-point> (<object>)
  slot x, init-value: 0;
  slot y, init-value: *global*;
  slot z, init-value: max(0, *global*);
end class;
```

This class definition defines a class with three slots. All three are initialized to the value 0 when you create a new `<3D-point>` object:

```
define variable *origin* = make (<3D-point>);
```

Since the initial-value expressions are evaluated when the class is defined, the initial value for these three slots will always be 0.

Consider this example:

```
define variable *global* = 100;

define variable *new-point* = make (<3D-point>);
```

In this example, even though the value of `*global*` has changed, the new `<3D-point>` object is initialized with the values 0, 0, and 0.

If you want the `make` function to initialize objects differently at different times, one option you have is the `init-function:` keyword. For example, consider this function definition:

```
define method get-global ()
    *global*
end;
```

Now consider the following class definition:

```
define class <2D-point> (<object>)
  slot x, init-value: *global*;
  slot y, init-function: get-global;
end class;
```

The `init-function:` keyword allows you to specify a function (of no arguments) that is called when new objects are created.

As an example, whenever you use the `make` function to create new `<2D-point>` objects, the `x` slot is initialized to whatever value `*global*` had when you defined the class. The `y` slot, however, is initialized to the *current* value of the `*global*` variable, because the *get-global* function is called *when the object is created*, not when the class is defined.

You can also initialize slots by specifying their initial values when you call the `make` function. For example, consider this class definition:

```
define class <4D-point> (<object>)
  slot x, init-keyword: x:;
  slot y, required-init-keyword: y:;
  slot z, init-keyword: z:, init-value: 0;
  slot t, init-keyword: t:, init-function: get-current-time;
end class;
```

This definition indicates:

- the `x` slot *can* be initialized (during calls to `make`) with the keyword `x:`
- the `y` slot *must* be initialized (during calls to `make`) with the keyword `y:`
- the `z` slot *can* be initialized (during calls to `make`) with the keyword `z:`, but if no `z:` value is specified, a default value of 0 is used
- the `t` slot *can* be initialized (during calls to `make`) with the keyword `t:`, but if no `t:` value is specified, `make` initializes the slot by calling the function `get-current-time`

For example, consider this object creation:

```
define variable *time-space* = make (<4D-point>, y: 10, z: 20);
```

This call to `make` creates a `<4D-point>` object. Immediately after `make` returns this object:

- The `x` slot is uninitialized.
- The `y` slot has a value of 10.
- The `z` slot has a specified value of 20 (not the default value of 0).
- The `t` slot has the whatever value was returned by the function call `get-current-time()`.

Initial values, whether you provide them in the class definition or in the call to `make`, must be consistent with any type specializers for the slot. For instance:

```
define class <2D-point> (<object>)
  slot x :: <integer>, init-value: 0;
  slot y :: <integer>, init-keyword: y:;
end class;
```

The initial value for x (which in this case is 0) must be an integer. When you call `make` to create a `<2D-point>` object, the value you specify for the `y` slot must also be an integer:

```
define variable *point* = make(<2D-point>, y: 1);
```

Class definitions can also include **initialization-argument specifications,** which allow you to specify more information about an init-keyword without specifying information about the slot itself. For example:

```
define class <4D-point> (<object>)
   slot x, init-keyword: x:;
   slot y, init-keyword: y:;
   slot z, init-keyword: z:;
   slot t, init-keyword: t:;

   required keyword x:;
   required keyword y:, type: <integer>;
   keyword z:, type: <integer>, init-value: 0;
   keyword t:, init-function: get-global;
end class;
```

This class defines four slots, each with an initialization keyword. It also specifies four initialization-argument specifications, which provide additional information about the initialization keywords for the slots:

- The `x:` keyword is required.
- The `y:` keyword is required and the specified value must be an integer.
- The `z:` keyword, if specified, must specify an integer value; if not specified, the default value is 0.
- The `t:` keyword, if not specified, is given the value returned by calling `get-global()`.

Additional Topics
Initialization-argument specifications are particularly useful when subclassing. See Chapter 16, "Inheritance," and Chapter 18, "Modules," for examples of creating subclasses. ◆

Slot Allocation

Most slots are **instance slots**—the *name* of the slot is the same for all instances of the class, but the *value* stored in that slot can be different for each instance. Each instance gets to store its own private information in each of its slots; as a result, you can change the value of one object's slots without affecting other objects' slots.

You can also create **class-allocated slots**—slots whose *value* is shared by all the instances of a class.

For example:

```
define class <example> (<object>)
    class slot shared, init-value: "common string";
    instance slot private, init-keyword: private:;
end class;

define variable *sample1* = make(<example>, private: "hello");

define variable *sample2* = make(<example>, private: "goodbye");
```

Each object has its own value for the `private` slot:

```
> *sample1*.private
"hello"

> *sample2*.private
"goodbye"
```

However, the two objects share the value of the `shared` slot. Changing its value for one of the objects:

```
> *sample1*.shared := "uncommon string"
"uncommon string"
```

affects all objects of the class:

```
> *sample2*.shared
"uncommon string"
```

A third option is the **virtual slot**—a slot for which no storage is allocated at all. For example, consider the class definition:

```
define class <two-values> (<object>)
    slot value-1, init-value: 10;
    slot value-2, init-value: 20;
    virtual slot sum;
end class;
```

The `sum` slot has no memory allocated to it. In addition, you must supply the getter and setter functions for the `sum` slot—they are not automatically created for you. Here is an example getter function you could define:

```
define method sum (object :: <two-values>)
    object.value-1 + object.value-2;
end method;
```

Now you can use sum as if it were an actual slot:

```
> define variable *some-values* = make(<two-values>)
defined *some-values*
```

```
> *some-values*.sum
30
```

You can also supply a setter function for a virtual slot:

```
define method sum-setter (value :: <integer>, object :: <two-values>)
    object.value-1 := truncate/ (value, 2);
    object.value-2 := value - object.value-1;
end method;
```

This setter function works by altering the value of the object's real slots. Here is an example that uses this setter function:

```
> *some-values*.sum := 100
100

> *some-values*.value-1
50

> *some-values*.value-2
50
```

Additional Topics
You can also specify that each subclass, rather than each instance, of a class allocate its own storage for a slot. See Chapter 16, "Inheritance," for details. ◆

Constant Slots and Setter Functions

A **constant slot** is a slot that has no setter function. Here are how you can create a constant slot:

```
define class <2D-point> (<object>)
  slot x, setter: #f, init-value: 10;
  slot y, setter: #f, required-init-keyword: y:;
end class;
```

These slots use the `setter:` keyword with a value of `#f` to indicate that no setter function should be created for this slot. The `x` slot uses an init-value to provide

the initial value and the `y` slot uses an init-keyword (`y:`) to provide the initial value. Once these slots are set, they cannot be reassigned:

```
define variable *my-point* = make(<2D-point>, y: 20);
```

You can also use the `setter:` keyword to rename the setter function for a slot. For example:

```
define class <2D-point> (<object>)
   slot x, setter: set-x;
   slot y, setter: set-y;
end class;
```

In this example, the setter functions are not `x-setter` and `y-setter`, as they would be by default, but `set-x` and `set-y`.

If you rename the setter functions, you can no longer use the extended form of the assignment operator to call the setter functions.

To set these slots, you must call their setter functions using normal function call syntax:

```
> define variable *my-point* = make(<2D-point>)
defined *my-point*

> set-x(10, *my-point*)
10

> set-y(20, *my-point*)
20
```

Of course, you can still use the slot reference syntax for getter function calls:

```
> *my-point*.x
10

> *my-point*.y
20
```

Additional Topics

You can also specify a slot to be `sealed` or `open`. These adjectives control the dynamism of the getter function and setter function for the slot. See Chapter 18, "Modules," for more information about dynamism. ◆

Additional Topics

This chapter shows you how to define your own classes and create instances of them. This section includes some related information and contains references to other chapters and to other documents where you can find supplementary information.

Class Creation

- **Defining Classes**

  Class definitions are not the only way to create class objects. You can also use the `make` function to create class objects at runtime. See the *Dylan Reference Manual* for details.

- **Inheritance**

  When you define a class, you must specify at least one superclass for the class to inherit from. The next chapter discusses inheritance, superclasses, subclasses, hierarchies, and heterarchies.

- **Dynamism**

  Some classes (that you import from other libraries) allow you to create subclasses of them, and some don't. Chapter 18, "Modules," discusses libraries and creating subclasses across library boundaries.

Instantiation

- **The Built-In `make` Function**

  This built-in function creates instances of classes. Sometimes you need to override it (for example, if you want to create an abstract instantiable class). See the next three chapters for more information.

- **The Built-In `initialize` Function**

  The `make` function calls the built-in `initialize` function to initialize the slots of an object after creating it. You can learn about this function in the *Dylan Reference Manual*.

CHAPTER 15

User-Defined Classes

CHAPTER 16

Inheritance

Contents

Key Concepts 271
About Inheritance 272
  Direct Superclasses 272
  Hierarchies 273
  Instantiability 274
  Dynamism 276
Multiple Inheritance 276
  Multiple Direct Superclasses 277
  Heterarchies 278
  Primary and Free Classes 278
Slot Inheritance 280
  Inherited Slot Specifications 280
  Initialization-Argument Specifications 281
  Slot Allocation 282
Additional Topics 283
  Method Dispatch 283
  Dynamism 284

This chapter continues the discussion of classes by introducing inheritance, class hierarchies, multiple inheritance, and class heterarchies.

Key Concepts

About Inheritance

- Every class must have at least one **direct superclass** from which it **inherits** structure and behavior.
- A class is called a **direct subclass** of its direct superclasses.
- Subclasses not only inherit from their superclasses, they can also be **specializations** of them.

Multiple Inheritance

- Classes can have more than one direct superclass.
- Classes can have only one **primary** direct superclass, but they can have any number of **free** direct superclasses.
- All of the classes in an Apple Dylan program form a **class heterarchy**—a class hierarchy with multiple inheritance.

Slot Inheritance

- Classes inherit the slots of their superclasses.
- Classes can specialize certain characteristics of inherited slots, such as initialization-argument characteristics.
- A fourth kind of slot allocation, the **each-subclass** allocation, is like the class allocation, except each subclass of the class maintains its own private value for the slot.

About Inheritance

Inheritance is the mechanism by which a class is able to use the structure and functionality of other classes, called **superclasses.** Through inheritance, class hierarchies and heterarchies are created.

Direct Superclasses

Every class, except the class `<object>`, has at least one **direct superclass** from which it inherits structure and behavior.

When you define a class, you must specify at least one direct superclass. This direct superclass determines certain information about the class you are defining. In particular, your class **inherits** certain characteristics from its direct superclass:

- Your class inherits structure from its direct superclass by inheriting the superclass's slots.
- Your class inherits behavior from its direct superclass due to the method-dispatching mechanism discussed in the next chapter.

When you define a class, it is said to be a **direct subclass** of its direct superclass. In general, a direct subclass represents some specialization of its direct superclass. That is, it inherits certain default structure and behavior from its direct superclass, but it also provides additional structure and behavior specifications.

For example, consider these two class definitions:

```
define class <2D-point> (<object>)
    slot x;
    slot y;
end class;

define class <3D-point> (<2D-point>)
    slot z;
end class;
```

The class <2D-point> represents values that have an x-coordinate and a y-coordinate. Its direct superclass is the built-in class <object>.

The <3D-point> class inherits from the <2D-point> class. In particular, it inherits the slots for containing an x-coordinate and a y-coordinate. It also adds a slot for containing a z-coordinate.

The class <2D-point> is the direct superclass of <3D-point>. The class <3D-point> is the direct subclass of <2D-point>; it *inherits* from <2D-point>, but it is also a *specialization* of <2D-point>.

Hierarchies

In the example in the previous section, notice that <object> is a direct superclass of <2D-point>, which is a direct superclass of <3D-point>. The class <3D-point> has one direct superclass (<2D-point>) and two **general superclasses:** <2D-point> and <object>. In Apple Dylan, the class <object> is a general superclass of every other class; it is also a direct superclass of some of them.

You can find all the general superclasses of a class using the built-in all-superclasses function; for example:

```
> all-superclasses(<3D-point>)
#(#<the class <3D-point>>, #<the class <2D-point>>, #<the class <object>>)
```

Notice that this list, called the **class precedence list**, includes the class itself, its direct superclass, its direct superclass's direct superclass (and so on; in general, this process repeats up to the class <object>).

A class may have more than one direct subclass. For example, consider this class definition:

```
define class <offset-2D-point> (<2D-point>)
    slot x-offset;
    slot y-offset;
end;
```

This class is a direct subclass of <2D-point>. Now, the class <2D-point> has two direct subclasses: <3D-point>, which adds a z slot, and <offset-2D-point>, which adds an x-offset slot and an y-offset slot.

These four classes (<object>, <2D-point>, <3D-point>, and <offset-2D-point>) form a small **class hierarchy.** The class <object> is the root of the hierarchy; it has one direct subclass, <2D-point>, which has two direct subclasses, <3D-point> and <offset-2D-point>. The three classes <2D-point>, <3D-point>, and <offset-2D-point>, are all **general subclasses** of <object>.

With class hierarchies comes the following distinction:

- An object is a **direct instance** of exactly one class.
- That object is a **general instance** of that class *and* all of that class's general superclasses.

As an example,

```
define variable *my-offset-point* = make (<offset-2D-point>);
```

This call to the make function creates an object which is a **direct instance** of the class <offset-2D-point>; this object is a **general instance** of <offset-2D-point>, <2D-point>, and <object>.

Instantiability

When you define a new class, you can specify it to be an abstract class or a concrete class. Abstract classes are not meant to have direct instances; concrete classes are.

Instantiability is separate from, but related to, abstractness or concreteness. A class is **instantiable** if you can send it as the first argument to the make function. Concrete classes, in general, are instantiable. Also, in general, when you send a concrete class to the make function, the make function creates and returns a direct instance of the concrete class.

For example, if you create a <2D-point> object using this function call

```
make(<2D-point>)
```

the object returned is a direct instance of the <2D-point> class, as this is a concrete class (by default).

Although it seems contradictory, some abstract classes are instantiable, and some concrete classes are not instantiable:

- An abstract class is instantiable if you can send it as the first argument to the `make` function. In this case, the `make` function typically returns a direct instance of one of the concrete subclasses of the specified abstract class. An example is the `<string>` class. Instantiating this abstract class returns an object of class `<byte-string>`:

  ```
  > object-class (make (<string>));
  #<the class <byte-string>>
  ```

- A concrete class is uninstantiable if you cannot create instances of it using the `make` function. The class `<small-integer>` is a concrete class; you can create direct instances of it (by specifying a literal constant, for instance). However, it is not instantiable. The function call

  ```
  > make(<small-integer>);
  ```

  results in the error:

  ```
  Error: #<the class <small-integer>> is not instantiable
  ```

You can create an abstract class by specifying the adjective `abstract` in your class definition:

```
define abstract class <all-points> (<object>)
end class;

define concrete class <1D-point> (<all-points>)
    slot x;
end class;
```

You can make this abstract class instantiable by defining a `make` method for it that returns a direct instance of the class `<1D-point>`:

```
define method make (the-class == <all-points>, #key)
    make(<1D-point>);
end method;
```

Additional Topics

See the *Dylan Reference Manual* for important information about specializing the `make` function. ◆

Dynamism

Whenever you create a new class, you specify at least one direct superclass. Therefore, defining a new class is sometimes called **subclassing** the specified direct superclass.

As described in Chapter 18, "Modules," you may subclass any class defined in the same library.

Whether you may subclass a class defined in another library, however, depends on whether that class is a sealed class or an open class. (It also depends, more fundamentally, on whether that class is exported from that library.)

When you define a class, you can specify whether it should be sealed or open:

```
define open class <all-points> (<object>)
end class;

define sealed class <1D-point> (<all-points>)
    slot x;
end class;
```

These definitions allow the `<all-points>` class to be subclassed in other libraries, but the `<1D-point>` class may not be subclassed in other libraries.

Additional Topics

Dynamism includes class dynamism and function dynamism, which applies to slot accessors. See "Dynamism" on page 284. ◆

Multiple Inheritance

Multiple inheritance refers to the ability for a class to inherit from more than one direct superclass.

Multiple Direct Superclasses

When defining a class, you may specify multiple direct superclasses. For example, consider the class definition:

```
define class <offset-3D-point> (<3D-point>, <offset-2D-point>)
    slot z-offset;
end class;
```

This class inherits the slots `x`, `y`, and `z` from the class `<3D-point>`, and the slots `x-offset` and `y-offset` from the class `<offset-2D-point>`. It defines one additional slot, `z-offset`. Therefore, objects of class `<offset-3D-point>` have six slots:

- `x`
- `y`
- `z`
- `x-offset`
- `y-offset`
- `z-offset`

The list of superclasses becomes more complicated when multiple inheritance is involved. For instance:

```
> all-superclasses(<offset-3D-point>)
#(#<the class <offset-3D-point>>, #<the class <3D-point>>, #<the class
<offset-2D-point>>, #<the class <2D-point>>, #<the class <object>>)
```

This list of superclasses is **class precedence list** for the class `<offset-3D-point>`.

Additional Topics

Method dispatching relies on the class precedence list to determine which classes are more specific than others, and uses that information to determine which methods are more specific than others. See "Method Dispatch" on page 283. ◆

Heterarchies

Since Apple Dylan allows a class to have multiple direct superclasses, the set of all classes is organized into a **heterarchy,** rather than a hierarchy. A heterarchy, also known as a **directed acyclic graph,** is a hierarchy where each class can have multiple direct subclasses and multiple direct superclasses. It is **directed** in that there is a difference between the direction of the superclass links and the subclass links. It is **acyclic** in that no class can be its own direct superclass, or any of its own general superclasses. In other words, there are no cycles in a heterarchy.

The number classes are an example of a hierarchy. The collection classes are an example of a heterarchy.

Additional Topics

Multiple inheritance and class heterarchies provide some powerful, if sometimes confusing, tools. See the *Dylan Reference Manual* for more information about these topics. ◆

Primary and Free Classes

When you define a class, Apple Dylan allows you to specify that the class is a **primary class** or a **free class.** The default is free.

Every class can have only one primary class in its list of direct superclasses. This is called the **primary superclass** of the class. Each class can have any number of free classes in its list of direct superclasses.

Here is an example. Imagine you define this class:

```
define abstract primary class <gui-element> (<object>)
end class;
```

and this class has two direct subclasses:

```
define abstract primary class <window> (<gui-element>)
end class;
```

```
define abstract primary class <button> (<gui-element>)
end class;
```

Now, suppose you want to create movable windows and stationary windows, as well as movable buttons and stationary buttons. You could define two free classes to encapsulate the notions of *movable* and *stationary*:

```
define abstract free class <movable-element> (<gui-element>)
end class;

define free class <stationary-element> (<gui-element>)
end class;
```

Now, you can mix the free classes in with the primary classes to create your concrete classes:

```
define class <movable-window> (<window>, <movable-element>)
    // slot descriptions
end class;

define class <stationary-window> (<window>, <stationary-element>)
    // slot descriptions
end class;

define class <movable-button> (<button>, <movable-element>)
    // slot descriptions
end class;

define class <stationary-button> (<button>, <stationary-element>)
    // slot descriptions
end class;
```

The class precedence list of one of these concrete classes looks like this:

```
> all-superclasses(<stationary-button>)
#(#<the class <stationary-button>>, #<the class <button>>, #<the class
<stationary-element>>, #<the class <gui-element>>, #<the class <object>>)
```

Slot Inheritance

As you've seen in the previous sections of this chapter, a subclass automatically inherits the slots of its superclasses. It also automatically inherits the behavior of its superclasses, as described in the next chapter, "Generic Functions." An example of inherited behavior is the slot getter and setter functions for inherited slots.

For example, consider this code:

```
> define variable *my-offset-point* = make(<offset-3D-point>)
defined *my-offset-point*

> x-setter (10, *my-offset-point*)
10

> x (*my-offset-point*)
10
```

As this example shows, the `x` getter function and the `x-setter` setter function, both defined for the class `<2D-point>`, also work for objects that are direct instances of `<offset-3D-point>`.

Of course, the reason to create subclasses is not merely to inherit information, but also to specialize it. Therefore, Apple Dylan allows you a number of ways to specialize slot information when you create a subclass. You can also specialize other behavior using generic functions, as described in the next chapter.

Inherited Slot Specifications

An **inherited slot specification** allows you to override certain initialization information for a slot. For example:

```
define class <window> (<object>)
    slot h-position, init-value: 0;
    slot v-position, init-value: 0;
end class;

define class <dialog> (<window>)
    inherited slot h-position, init-value: 50;
    inherited slot v-position, init-value: 50;
end class;
```

You can also specify an init-function in an inherited slot specification. If you have an inherited slot specification that specifies neither an init-value nor an init-function, its sole purpose is to require that some superclass of the class have a slot by the same name.

Initialization-Argument Specifications

You can also use initialization-argument specifications to specialize initialization behavior for subclasses. For example:

```
define class <window> (<object>)
    slot h-position, init-keyword: h:;
    slot v-position, init-keyword: v:;
end class;

define class <dialog> (<window>)
    required keyword h:;
    keyword v:, init-value: 0;
end class;
```

In this example, the slots are inherited, but the keyword arguments are slightly modified. For the `<window>` class, there are two (optional) init-keywords (`h:` and `v:`). Here is an example of creating a `<window>` object:

```
define variable *my-window* = make(<window>, v: 10);
```

This call to the `make` function creates a `<window>` object with the `h-position` slot uninitialized and the `v-position` slot initialized to 0.

To create a `<dialog>` object, however, you might use this code:

```
define variable *my-dialog* = make(<dialog>, h: 10);
```

The `h:` keyword is required. It specifies an initial value of 10 for the `h-position` slot. The `v:` keyword is not provided, so the `v-position` slot is initialized to the specified init-value of 0.

Slot Allocation

In the previous chapter, you saw instance-allocated slots, class-allocated slots, and virtual slots. Apple Dylan also provides the `each-subclass` allocation. This allocation differs slightly from the `class` allocation, as illustrated in the following example:

```
define class <example> (<object>)
    class slot x, init-value: "original x string";
    each-subclass slot y, init-value: "oringinal y string";
end class;

define class <subexample> (<example>)
end class;
```

The slot `x` is shared amongst all instances of the `<example>` class, direct or otherwise:

```
> define variable *e1* = make(<example>)
defined *e1*

> define variable *s1* = make(<subexample>)
defined *s1*

> define variable *s2* = make(<subexample>)
defined *s2*

> *e1*.x := "everyone changes"
"everyone changes"

> *s1*.x
"everyone changes"
```

```
> *s2*.x
"everyone changes"
```

However, the slot `y` is shared only amongst direct instances of a particular class. Therefore, all direct instances of `<example>` share a single `y` slot, and all direct instances of `<subexample>` share a (different) single `y` slot:

```
> *s1*.y := "only some of us change"
"only some of us change"

> *s2*.y
"only some of us change"

> *e1*.y
"original y string"
```

Additional Topics
You can also create an interesting type of slot, called a **filtered slot,** using inheritance and virtual slot allocation. See the *Dylan Reference Manual* for details. ◆

Additional Topics

This chapter provides only a brief introduction to the concepts and power behind inheritance and multiple inheritance. The following few sections provide some additional information and also some pointers to supplementary material.

Method Dispatch

Generic functions, which are described in the next chapter, use class heterarchies to implement method dispatch.

- **Generic Functions**

  A generic function is a function that selects an implementation depending on the arguments sent to the function.

- **Methods and Argument Type Specialization**

  A method is a single function implementation. Methods can be specialized for different classes of objects.

- **More Specific Methods**

  Using class precedence lists, generic functions can examine the argument type specializations for different methods to determine the most specific method that applies to a particular set of arguments.

Dynamism

Dynamism, which is discussed in Chapter 18, "Modules," includes both class dynamism and function dynamism.

- **Class Dynamism**

  Class dynamism refers to the ability to create subclasses of a class in a different library than the one in which the class is defined.

- **Getter and Setter Function Dynamism**

  Function dynamism is the ability to add methods to generic functions in a different library than the one in which the function is defined. Function dynamism applies to all generic functions, including slot accessors.

CHAPTER 17

Generic Functions

Contents

Key Concepts 287
About Generic Functions 288
  Implicit Generic Function Creation 288
  Slot Accessors 291
  Explicit Generic Function Creation 293
Method Dispatch 294
  Parameter Type Specialization 294
  Singletons 296
  Method Specificity 297
  Next Method 300
Additional Topics 302
  Parameter Lists 302
  Class Precedence and Method Specificity 302
  Dynamism 303

In Chapters 8 and 9, you learned about methods—specific function implementations. In this chapter, you'll learn how generic functions implement polymorphism by using the class heterarchy to dispatch function calls to the appropriate methods.

Key Concepts

About Generic Functions

- A **generic function** is a function object that implements **polymorphism.**
- Each generic function maintains a set of references to method objects that provide the actual implementations of the function.
- Generic functions can be created **implicitly** by `define method` method definitions and by slot descriptors in `define class` class definitions.
- You can also create generic functions **explicitly** using the `define generic` form of definition. You can subsequently add methods to the generic function

Method Dispatch

- **Method dispatch** is the process of selecting the most appropriate method for a given set of argument values.
- Different methods of a generic function include **type specializations** in their parameter lists to indicate which types of arguments they accept.
- Methods can use **singletons** to specialize an argument to a specific value, rather than an entire class of values.
- Generic functions use type specializations and class precedence to determine **applicable** methods, and then to sort the applicable methods by **specificity.**
- You can use the **next-method** mechanism to call the next most specific method, which allows a class to inherit functionality from its superclasses.

About Generic Functions

Generic functions are the mechanism by which Apple Dylan implements **polymorphism**—the ability for a single function to have multiple implementations depending on the types of arguments sent to it.

A generic function is an object. In fact, it's an instance of the class `<generic-function>`, which is a subclass of `<function>`. Methods are instances of the class `<method>`, which is the other subclass of `<function>`.

Method objects contain the compiled code that actually implements functions.

Although a generic function does not contain implementation code itself, it does have a few other responsibilities:

- It maintains references to some number of method objects, which provide the actual implementations for the function.
- It defines the basic acceptable parameter list for the function. All the methods of the generic function must have compatible parameter lists, called **congruent parameter lists.**
- When called, it examines the classes (and sometimes the values) of the arguments, and makes a list of **applicable methods,** sorted from the **most specific method** to the **least specific method.** It then calls the most specific applicable method, sending it the argument values. This process is called **method dispatch.**

The next few sections discuss the creation of generic functions. The section "Method Dispatch," beginning on page 294, discusses the method-dispatching process in more detail.

Implicit Generic Function Creation

Implicit generic function creation is the creation of a generic function object while defining another type of object. There are two common ways to implicitly define a generic function:

- the `define method` form of method definition
- the `define class` form of class definition

As an example, consider this definition:

```
define constant double = method (input)
                             input + input;
                         end method;
```

This definition creates a bare method—a method not connected to any generic function—that you can call directly:

```
> double(3)
6
```

No method dispatch happens during this call; the method object bound to the name `double` is called directly.

However, if you define `double` using the `define method` form of definition:

```
define method double (input)
    input + input;
end method;
```

This method definition creates a method object—in fact, it creates an equivalent method object to the one that was created by the earlier method definition. In this case, however, the method object is *not* bound to the name `double`. Instead:

- If there is a generic function named `double`, then this method is added to that generic function.
- If there is no generic function named `double`, then one is created; that is, a generic function object is created and bound to the name `double`. Then, the method object being defined is added to the new generic function. This is called **implicit generic function definition.**

Once you've implicitly created a `double` generic function, you can use the `define method` form of method definition to add more methods to it. For example:

```
define method double (input :: <string>)
    concatenate (input, input);
end;
```

Now, the generic function double has two methods in its arsenal: one is invoked for the general case, and one works specifically on strings. Therefore:

```
> double (3)
6

> double ("tu")
"tutu"
```

The `double` generic function examines the classes of the actual argument values and selects the appropriate implementation (method) for you.

As you add more methods to the generic function, you must ensure that each new method has a congruent parameter list. For example, the generic function `double` implicitly created by the earlier `define method` definition expects methods with a single required parameter.

Suppose you try to add a method with an incongruent parameter list:

```
define method double (input-1 :: <integer>, input-2 :: <integer>)
    2 * (input-1 + input-2);
end method;
```

This method takes two required arguments. Since this is incongruent with the previous generic function, this `define method` definition *eliminates* the previous generic function, implicitly defines a *new* generic function, and binds the new function to the name `double`. The old methods are now inaccessible!

```
> double(3)
Error: Wrong number of arguments to the #<the method double (<integer>,
<integer>) at: #xD99282>: #(3).
```

Additional Topics

Parameter list congruency is a complicated subject; there are many rules that depend on the number of required, rest, and keyword parameters, and also on the type specializations for those parameters. See "Parameter Lists" on page 302. ◆

Slot Accessors

Another form of implicit generic function creation is the creation of slot accessors during class definitions. For example:

```
define class <2D-point> (<object>)
    slot x;
    slot y;
end;
```

This definition defines four methods:

- a method that accesses the slot `x`
- a method that sets the slot `x`
- a method that accesses the slot `y`
- a method that sets the slot `y`

These four methods are added, respectively, to the generic functions named `x`, `x-setter`, `y`, and `y-setter`. If any of these generic functions do not already exist, they are created, and then the corresponding method is added to them.

Note that these four generic functions can interfere with existing generic functions that happen to have the same name, but incongruent parameter lists.

As an example, the previous class definition creates a class and four methods. The slot descriptors in that class definition are equivalent to calling these four method definitions:

```
define method x (object :: <2D-point>)
    // return value of x slot of input object
end;

define method x-setter (new-value, object :: <2D-point>)
    // set value of x slot of input object to new-value
end;

define method y (input :: <2D-point>)
    // return value of y slot of input object
end;
```

```
define method y-setter (new-value, input :: <2D-point>)
    // set value of y slot of input object to new-value
end;
```

If you already have one of these generic functions (`x`, for example) that has an incompatible parameter list (for example, more than one required parameter), then this class definition eliminates the earlier version of the generic function `x` and replaces it with the new version (which has a single required parameter).

Conversely, if you now define a method named `x` with an incompatible parameter list:

```
define method x (input-1 :: <integer>, input-2 :: <integer>)
    // . . .
end method
```

then this method definition creates a new generic function `x`, eliminating the `x` generic function that gets the `x` slot of `<2D-point>` objects.

At this point, you can no longer access the value of the x slot of `<2D-point>` objects!

```
> define variable *2D* = make (<2D-point>);
defined *2d*

> *2D*.x := 10;   // This assignment uses the x-setter function.
10

> *2D*.x          // This function call attempts to call the x function.
Error: Wrong number of arguments to #<the method x (<integer>,
<integer>) at: #xD992A2>: #(#<<2D-point> id: 44>).
```

For this reason, slot names are often prefixed with their class name. For example:

```
define class <2D-point> (<object>)
    slot 2D-point-x;
    slot 2D-point-y;
end;
```

This naming conventions makes generic function conflicts unlikely, although slot references are somewhat more unwieldy:

```
> define variable *2D* = make (<2D-point>);
defined *2d*

> *2D*.2D-point-x := 10;
10

> *2D*.2D-point-x
10
```

Explicit Generic Function Creation

You can also create a generic function explicitly, using the `define generic` form of definition. Here is an example:

```
define generic triple (input :: <number>);
```

This definition creates a generic function object, with no methods, that expects a single required argument of class `<number>`. You can now add methods to this generic function using the `define method` form of definition:

```
define method triple (input :: <ratio>)
    (numerator(input) * 3) / denominator(input);
end method
```

(It's a little inefficient, but it does triple the input ratio.)

Since the parameter list is congruent to the existing generic function named `triple`, this method is added to that generic function. (If the parameter lists were incongruent, this method definition would create a new generic function named `double`, effectively eliminating the existing one.)

Additional Topics

You can also create anonymous generic functions, by applying the make function to the class `<generic-function>`. See the *Dylan Reference Manual* for complete information.

There are additional built-in functions that allow you to manipulate generic functions. See the *Dylan Reference Manual*.

The `define generic` form of definition also allows you to specify the dynamism (`sealed` or `open`) of the generic function. In addition, you can seal specific branches of generic functions, while leaving other branches open. See "Dynamism" on page 303. ◆

Method Dispatch

Method dispatch is the process by which a generic function selects the most appropriate implementation for a given set of argument values.

Method dispatch works in separate stages:

1. First, the generic function examines the classes of all the arguments (and sometimes the values of the arguments as well).
2. Then, it determines which of its methods are applicable to the arguments—that is, which of its methods have **type specializers** that are compatible with the types of the actual arguments.
3. Then, it sorts the **applicable arguments** from most specific to least specific.
4. Then, it calls the **most specific method**.

The next few sections examine the different aspects of this process in more detail.

Parameter Type Specialization

A generic function chooses which method to call based on the specified type specializations in its methods' parameter lists.

For example,

```
define method example (input-1 :: <integer>, input-2 :: <string>)
    // . . .
end method
```

This method specifies a type specialization for both of its parameters. When the `example` function is called, this method is **applicable** if the first argument is a general instance of the class `<integer>` and the second argument is a general instance of the class `<string>`. That is, the above method is applicable to these function calls:

```
example (0, "");
example (3 + 7, "this is a test");
example (max(5, 4, 3, 2, 1), concatenate("hello, ", "world"));
```

The above method is not applicable to these function calls:

```
example ( 1.1E10, 'a');
example ("test", 10);
example (#(1), #("test"));
```

If a method definition includes a parameter with no type specializer, the specializer is assumed to be `:: <object>`—that is, for that parameter, the method matches any value. For example,

```
define method example (input-1, input-2 :: <integer>)
    // . . .
end method
```

This method does not include a type specializer for the `input-1` parameter. Therefore, this parameter matches any argument value. As a result, this method is applicable to these function calls:

```
example ("", 0);
example (#(1, 2, 3), max (3, 4));
example (#["hi", "hello"], max(5, 4, 3, 2, 1));
```

Notice that the first implementation of example and the second implementation are **disjoint**—that is, there is no function call for which both methods are applicable.

Now, consider this method definition:

```
define method example (input-1 :: <integer>, input-2 :: <integer>)
    // . . .
end method
```

This method is applicable to these function calls:

```
example (1, 0);
example (first(#(1, 2, 3)), max (3, 4));
example (#[1, 2, 3][1], max(5, 4, 3, 2, 1));
```

However, the previous `example` method, which had the parameter list

```
(input-1, input-2 :: <integer>)
```

is also applicable to these three function calls.

The section "Method Specificity" on page 297 explains how Apple Dylan handles function calls for which there are multiple applicable methods.

Singletons

In the previous section, you saw type specializers which allow you to specify the class of value that a method accepts for a particular argument. **Singletons** allow you to be even more specific—they allow you to define methods that accept only one specified value for a particular argument.

For example:

```
define method example (input-1 == 0, input-2 :: <integer>)
    // . . .
end method;
```

This method is applicable to function calls *only if* the first argument is *the actual value* 0 (and the second argument is an integer). For example, this method is applicable to these function calls:

```
example (0, 0)
example (0, 10)
example (0, max(5, 10))
```

but not to these functions calls:

```
example (1, 0)
example ('a', 10)
example (#f, max(5, 10))
```

Of course, if this method is applicable to a function call, the two previous methods (defined in the previous section) are also applicable to that function call. That's where method specificity comes in.

Method Specificity

When you call a generic function, it examines the classes of the arguments. If any of the generic function's methods use singleton specializers, the generic function also examines the actual values of the corresponding arguments. The generic function uses the classes (and values) of the actual arguments and the type specializers of its methods to determine the set of applicable methods.

If there is only one applicable method, the generic function calls that method.

However, if there is more than one applicable method, the generic function needs to select one. To select the most appropriate method, the generic function orders the applicable methods in order of specificity.

The two most basic rules for determining method specificity are:

1. A subclass is more specific than any of its superclasses.
2. A singleton is more specific than any class.

For example, imagine you have these four method definitions:

```
define method double (input)
    pair (input, input);
end;

define method double (input :: <number>)
    2 * input;
end;

define method double (input :: <integer>)
    input + input;
end;

define method double (input == 0)
    input;
end;
```

All four methods are applicable to this function call:

```
double(0);
```

So, the generic function `double` has to sort the methods in order of specificity. Using the two basic rules described earlier, the most specific method is the one that specializes on the singleton value 0. The second most specific is the one that specializes on the class `<integer>`. Third is the one that specializes on `<number>`. And fourth is the one with no specializer, which is equivalent to specializing on `<object>`.

So, in this case, the generic function calls the method that specializes on the singleton value 0, and the value returned is 0.

Suppose you make the function call:

```
double(2/3);
```

Only two of the methods are applicable, and the most specific method is the one that specializes on `<number>`.

As another example, consider these two method definitions:

```
define method add-em-up (input :: <list>)
    head(input) + add-em-up(tail(input));
end method;

define method add-em-up (input == #())
    0;
end method;
```

These two methods create a generic function named `add-em-up`, which takes a single list as an argument. If the list is the empty list, the generic function calls the more specific method and returns 0. If the list is not an empty list, the generic function calls the less specific method (which is the most specific *applicable* method). This method adds the first element of the list to the sum of the rest of the list. In this way, you can create recursive functions and have Apple Dylan test for the base case for you; you do not have to use any control statements at all.

Here are some examples of calling this generic function:

```
> add-em-up (#())
0

> add-em-up (#(1, 2, 3, 4, 5))
15
```

Additional Topics

These examples are made unusually simple because they specify only one argument, and because there is a clear linear relationship among the various type specializers. When you have multiple arguments, or multiple inheritance, sorting methods by specificity becomes substantially more complex. Sometimes it is impossible and the ordering is ambiguous. See "Class Precedence and Method Specificity" on page 302. ◆

Next Method

As you learned in the previous chapter, when you create a class, you **inherit** certain characteristics from the direct superclass, or direct superclasses, of that class:

- You automatically inherit the slots of the superclass.
- You also automatically inherit the behavior of the superclass, as a result of the method-dispatching algorithm.

For example, consider the following two class definitions:

```
define class <2D-point> (<object>)
    slot x :: <real>, init-keyword: x:;
    slot y :: <real>, init-keyword: y:;
end class;

define class <3D-point> (<2D-point>)
    slot z :: <real>, init-keyword: z:;
end class;
```

The class `<3D-point>` automatically inherits the slots `x` and `y` from its superclass, `<2Dpoint>`. Now, suppose you define this method:

```
define method sum-coordinates (input :: <2D-point>)
    input.x + input.y;
end method;
```

As a result of the method-dispatching algorithm, the class `<3D-point>` automatically inherits this functionality, even though it is defined for the superclass `<2D-point>`. For example:

```
> define variable *2D* = make (<2D-point>, x: 10, y: 20);
defined *2D*

> define variable *3D* = make (<3D-point>, x: 30, y: 40, z: 50);
defined *3D*

> sum-coordinates(*2D*)
30
```

```
> sum-coordinates(*3D*)
70
```

As you can see, the `sum-coordinates` function applies equally well to objects of class `<3D-point>` as it does to objects of class `<2D-point>`. This is how classes inherit behavior from their superclasses.

Of course, sometimes a class wants to override the behavior of its superclass entirely. You can achieve this simply by adding a more specific method to the generic function in question:

```
define method sum-coordinates (input :: <3D-point>)
    input.x + input.y + input.z;
end method;
```

This method, being more specific, completely overrides the previous method when called on objects of class `<3D-point>`:

```
> sum-coordinates(*3D*)
120
```

Sometimes, however, you don't want to override a superclass's behavior completely; instead, you just want to specialize its behavior. To this end, Apple Dylan provides then **next-method** mechanism. A call to `next-method` is a call to the next most specific method. Consider this method definition:

```
define method sum-coordinates (input :: <3D-point>)
    next-method() + input.z;
end method;
```

This method definition also sums the three coordinates of a `<3D-point>`. It does so by calling `next-method`, the next most specific method, which sums the `x` slot and the `y` slot. (The `input` argument is passed to `next-method` by default.) Then, this method adds the value of the `z` slot.

In this way, the class `<3D-point>` both inherits behavior (the call to `next-method` inherits behavior from `<2D-point>`) and specializes that behavior (performs the extra addition operation).

Additional Topics

The `next-method` function always calls to the next most specific method. When you have multiple arguments and multiple inheritance, this mechanism can be more powerful than the inherited function calls available in some object-oriented programming languages. See the *Dylan Reference Manual* for more information the `next-method` function. ◆

Additional Topics

Parameter Lists

Scope and extent are of critical importance when creating and referencing variables.

- **Type Specialization**

  As you saw in this chapter, each parameter of a method can be specialized to a specific type. **Types** include classes and singletons, and also union types and limited types. See the *Dylan Reference Manual* for more information about types.

- **Congruency**

  The rules for parameter list congruency are complex, and depend on the number and kinds of parameters (required, optional, and keyword), and also on the type specializers for the parameters. The complete set of rules for parameter list congruency is in the *Dylan Reference Manual*.

Class Precedence and Method Specificity

The examples in this chapter created methods using simple class hierarchies and unambiguous ordering. The *Dylan Reference Manual* discusses more

complex class precedence rules and Apple Dylan's response to method ambiguity.

- **Class Precedence Lists**

  Chapter 16, "Inheritance," introduced class precedence lists. Generic functions use these lists for both checking parameter list congruency and for method specificity.

- **Ambiguous Method Specificity**

  As a result of the various kinds of parameters, as well as multiple inheritance, it is not possible to order all of the applicable methods by specificity. Some applicable methods may be just as specific, but no more specific, than other methods. This create method ambiguity, and method dispatch may become impossible.

Dynamism

Dynamism includes class dynamism and function dynamism. The next chapter, "Modules," discusses dynamism in detail.

- **Class Dynamism**

  Class dynamism refers to the ability to create subclasses of a class in a different library than the one in which the class is defined.

- **Getter and Setter Function Dynamism**

  Function dynamism is the ability to add methods to generic functions in a different library than the one in which the function is defined. Function dynamism applies to all generic functions, including slot accessors.

CHAPTER 17

Generic Functions

CHAPTER 18

Modules

Contents

Key Concepts 307
About Modules 308
  Variable Name Spaces 308
  Exporting Variables 310
  Importing Variables 311
  Subclassing Across Modules 312
  Adding Methods Across Modules 313
Libraries 315
  Exporting Modules 316
  Importing Modules 316
  Adding Methods Across Libraries 319
  Subclassing Across Libraries 320
Dynamism 322
  Sealing Classes 323
  Sealing Generic Functions 323
  Sealing Branches of Generic Functions 324
  Sealing Slot Accessors 325
Additional Topics 326
  The Development Environment 326
  Dynamism 327

This chapter discusses modules in more detail, and describes how you group them into libraries. It also discusses the effects of modules and libraries on subclassing and adding methods to generic functions.

Key Concepts

About Modules

- A **module** is a **namespace** for variables.
- Modules in your source code create different **runtime environments** in your application nub.
- A module can **export** variables defined it for other modules to use.
- A module can **import** variables exported by other modules.
- You can create subclasses in a module of classes imported from another module.
- You can also add methods in a module to generic functions imported from another module.

Libraries

- **Libraries** are collections of modules; they are separate units of compilation and optimization.
- The Apple Dylan development environment uses **projects** to represent libraries.
- Libraries can **import** and **export** modules (that is, the variables defined in those modules).

Dynamism

- **Class dynamism** allows you to specify whether a class can be subclassed outside of its library.
- **Function dynamism** allows you to specify whether methods can be added to a generic function outside of its library.

About Modules

In Apple Dylan, **modules** are mechanisms for organizing your source code, for creating large-scale namespaces, and for creating different runtime environments while your program is running.

Variable Name Spaces

A module, first and foremost, is a **namespace** for variables. When you define a variable in a module, the development environment recognizes that name anywhere else in the same module.

Modules of source code in the development environment correspond to different runtime environments in the application nub. A **runtime environment** is the group of variables accessible to a particular constituent. You set up different runtime environments by creating modules.

The Listener window allows you to execute source code under different runtime environments. Up to this point in this book, all of the examples have assumed that you were using the default `Dylan-user` module that is automatically provided for you when you create a new project.

In general, you want to use that Dylan-user module to define other modules, and keep your actual source code in those other modules.

For example, you could include this **module definition** in your Dylan-user module:

```
define module module-1
    use Apple-Dylan;
end module;
```

If you update your project to compile and execute this module definition, Apple Dylan automatically creates a new module named `module-1` in your project. You'll see it in the upper-left pane in the browser window for you project. If you select this module, you can create source folders and source records within it. Suppose you create five source records, each with a single definition:

```
define constant $initial-value = 1;

define variable *current-value* = $initial-value;

define class <1D-point> (<object>)
    slot x, init-keyword: x:, init-value: $initial-value;
end class;

define method double (input)
    pair(input, input);
end method;

define method quadruple (input)
    double ( double (input) );
end method;
```

After entering these definitions, you can update your project to compile them and load them into the runtime environment.

If you switch to the Listener window now, you can use the pop-up menu in the bottom-left corner to choose the `module-1` module from the `User-Library` library. (This library is automatically created for you by Apple Dylan; you'll find more information about libraries later in this chapter.) Once you choose this module, you can use the Listener to examine and modify the runtime environment created by your module:

```
> *current-value*
1

> *current-value* := 2;
2

> *current-value*
2
```

This module, named `module-1`, provides a single **namespace**—every variable defined in this module is accessible from anywhere within the same module. (Exception: module variables can be temporarily **shadowed**— made inaccessible—by local variables.)

To create other namespaces, you must create other modules. One module can **use** another module to get access to the variables defined in the other module.

As an example, the `module-1` module defined earlier uses the `Apple-Dylan` module. In this way, the `module-1` module gains access to Apple Dylan variables. For example, the `module-1` module contains a reference to the variable name `pair`, which is bound to the built-in `pair` function. This variable name can be referenced in the `module-1` module because the module **uses** the `Apple-Dylan` module.

The `module-1` module does not have access to every variable defined in the `Apple-Dylan` module, however. It has access only to those variables that are **explicitly exported** by Apple Dylan. The next section discusses the exporting of variables.

Exporting Variables

When you create a module and you want other modules to be able to use the variables defined in it, you must explicitly export the variables you want to be accessible. Here is an example, which changes the module definition of `module-1` so that it explicitly exports some of its variables:

```
define module module-1
    use Apple-Dylan;

    export
        $initial-value,
        *current-value*,
        <1D-point>,
        x, x-setter,
        double;

end module;
```

This module definition exports every variable defined in the `module-1` module, except for the variable named `quadruple`, which is bound to a generic function object. Since this module does not explicitly export that variable, other modules cannot get access to that generic function.

Notice that although the class `<1D-point>` is exported, the slot-accessing functions `x` and `x-setter` are exported separately. In Apple Dylan, slot accessors

are normal generic functions; they are not attached to the class and they must be exported separately if you want other modules to have access to them.

Importing Variables

Back in your `Dylan-user` module, you can add another source record to define a second module:

```
define module module-2
    use Apple-Dylan;
    use module-1;
end module;
```

This module definition uses Apple Dylan, and it also uses the `module-1` module. As it is written, it imports every variable that the `module-1` module exports. Apple Dylan does provide options you can specify that allow you to import some of the available variables, but exclude others; you can also rename the variables upon importing. For example:

```
define module module-2
    use Apple-Dylan;
    use module-1,
        import: { *current-value* => *external-value*,
                  <1D-point>,
                  x => point-x,
                  double => times-two }
end module;
```

This module definition specifies that `module-2` module imports four variables from the `module-1` module. Three of these variables are renamed; that is, the name used to specify the object in `module-2` is different than the name used to specify the object in `module-1`—but they still specify the same object.

For example, update your project, and, in your Listener window, choose the `module-2` module. You can then execute this code:

```
> *external-value* := 11;
11
```

Using the Listener window to switch back to the `module-1` module, you can examine the corresponding variable:

```
> *current-value*
11
```

So, the name `*external-value*` in `module-2` is bound to the same value that the name `*current-value*` is bound to in `module-1`.

Subclassing Across Modules

Since the module `module-1` exports the class `<1D-point>` and the module `module-2` imports it, you can create subclasses of `<1D-point>` in `module-2`.

For example, you can add this class definition in a source record of your `module-2` module:

```
define class <2D-point> (<1D-point>)
    slot point-y, init-keyword: y:, init-value: 1;
end class;
```

This class inherits the `x` slot from the class `<1D-point>`. (However, the getter function for this slot is renamed `point-x` when it is imported. See the previous section.) This class adds the slot `point-y`.

In the Listener window, you can choose the `module-2` module and create a `<2D-point>` object with the following definition:

```
> define variable *point* = make(<2D-point>, x: 10, y: 20);
defined *point*
```

You can examine the value of the inherited `x` slot of this object using the imported slot accessor function, which is renamed `point-x`:

```
> *point*.point-x;
10
```

You can also examine the value of the point-y slot using the point-y getter function:

```
> *point*.point-y;
20
```

Notice, however, that the slot accessor function x-setter is not imported into module-1 (in the previous section). Therefore, code in the module-2 module cannot change the value of this inherited slot! Inside module-2, objects of class <2D-point> have read-only x slots.

However, you *can* pass a <2D-point> object to a function defined in module-1 (as long as that function is imported into module-2).

For example, imagine you defined this function in module-1:

```
define method double-x-slot (input :: <1D-point>)
    double (input.x);
end method;
```

If you exported this function and imported it into module-2, you could make this function call from module-2:

```
> double-x-slot(*point*);
#(10 . 10)
```

This function, defined in module-1, has access to the x slot of the object (but not to the point-y slot, as the point-y getter and setter functions are defined in module-2, not module-1).

Adding Methods Across Modules

Circular references are not allowed amongst modules. For example, since module-2 uses module-1, it is an error for module-1 to use module-2.

However, module-1 can still call code written in module-2. Here's how:

- The module-1 module defines a generic function.
- The module-2 module adds a method to that generic function.
- The module-1 module calls the generic function.

- Depending on the arguments sent to the function, method dispatch may result in a call to the method that `module-2` added to the generic function.

As an example, you can add this method definition (in a source record) to your `module-2` module:

```
define method times-two (input :: <integer>)
    2 * input;
end method;
```

Now, in the Listener window, choose the `module-2` module, and execute the following function call:

```
> times-two (3)
6
```

The `module-2` module obviously has access to the method you defined in it. Now, execute this function call:

```
> times-two ("hi")
#("hi" . "hi")
```

As this result shows, the `module-2` module also has access to the method defined in `module-1` (under the name `double`). Here is what is happening:

- The `module-1` module creates a generic function named `double`, and adds a method to it.
- The `module-2` module imports that generic function, but under the name `times-two`. Important: the *same generic function object* is referenced by the name `times-two` in `module-2` and by the name `double` in `module-1`.
- The `module-2` module adds a method, specialized for integers, to that generic function (using the name `times-two`).
- Subsequent calls in `module-2` to the `times-two` function are actually calls to the generic function defined in `module-1`. Based on the arguments, this generic function chooses the more appropriate of its two methods.

It is important to note that `module-2` adds the new method directly to the generic function—therefore, any module that can call the generic function has access to the new method.

For example, if you set your Listener window to `module-1`, you can execute the following code:

```
> double ("hi")
#("hi" . "hi")
```

This is no surprise, as this implementation of `double` is defined in `module-1`. However, you can also make this function call in `module-1`:

```
> double (3)
6
```

As this result shows, the `module-1` module can execute the method defined in the `module-2` module. In `module-1`, the name `double` is bound to the generic function object; when `module-2` adds a method to that generic function, the new method becomes accessible to `module-1`.

Libraries

In Apple Dylan, a **library** is a unit of compilation and of optimization. In particular, a library

- is represented in the development environment by a **project**
- contains modules
- is a unit of source code that can be compiled separately
- is the mechanism that controls and limits **dynamism**

When you create a new project in the development environment, you are automatically creating a new library. By default, Apple Dylan names this library `User-Library`, as you saw earlier in this chapter.

You can provide a name for your library, and specify other information about it, using a **library definition.** Here is an example that you can add to your `Dylan-user` module, along with your module definitions:

```
define library my-library
    use Dylan;
end library;
```

This library definition specifies:

- The name of the library is `my-library`.
- The `my-library` library uses the `Dylan` library, which is the library that contains the `Apple-Dylan` module. If the `my-library` library did not use the `Dylan` library, then the modules `module-1` and `module-2` would not be able to use the `Apple-Dylan` module.

Exporting Modules

You can also use a library definition to specify which modules in the library are exported—that is, which modules can be imported into other libraries. For example:

```
define library my-library
    use Dylan;
    export module-1;
    export module-2;
end library;
```

This library definition specifies that the modules `module-1` and `module-2` are exported for use by other libraries.

The next section explains how you can create a new library that uses `my-library` and imports variables exported by `module-1` and `module-2`.

Importing Modules

As mentioned earlier, libraries are separately compilable units of Apple Dylan programs. You can use the development environment to compile the `my-library` library, and save the corresponding project to a project file.

To use the `my-library` library in another library, you must first create another library to use it in. To do this:

1. You create a new project for the new library.
2. You choose the "Add To Project" menu item and add your existing project file (the one that contains the `my-library` library). It is added as a **subproject** of your new project.

3. You add a source record to the `Dylan-user` module of your new project to define your new library. For example:

```
define library new-library
    use Dylan;
    use my-library;
end library;
```

Now you can add a module to your new library, by including a module definition in your Dylan-user module:

```
define module new-module
    use Apple-Dylan;
    use module-1;
    use module-2;
end module;
```

Compiling this module definition creates a new module, named `new-module`, in your new library, named `new-library`, represented by your new project. This module

- uses the `Apple-Dylan` module, which it can use because `new-library` uses the `Dylan` library
- uses the `module-1` and `module-2` modules, which it can use because `new-library` uses the `my-library` library

This module definition, by default, imports every exported variable from the `Apple-Dylan`, `module-1`, and `module-2` modules. At this point, the `module-2` module doesn't export any variables, so you might want to go back and change its method definition.

(You don't have to leave your new project to do this—you can change it directly in the subproject.)

Here is an example:

```
define module module-2
    use Apple-Dylan;

    use module-1,
        import: { *current-value* => *external-value*,
                  <1D-point>,
                  x => point-x,
                  double => times-two },
        export: all;

    export
        <2D-point>,
        point-y, point-y-setter;

end module;
```

In this example, the **option** `export: all` has been added to the `use module-1` clause. This option indicates that `module-2` should export all of the variables that it imports from `module-1`.

This example also adds the **export clause:**

```
export
    <2D-point>,
    point-y, point-y-setter;
```

This clause exports variables actually defined in `module-2`.

The `new-module` imports every exported variable from `module-1` and from `module-2`. As a result, this new module imports the variable name `*current-value*` (which is exported from `module-1`) and the variable name `*external-value*` (which is exported from `module-2`). These two variable names still refer to the same value, but you can use either name in `new-module`:

```
> *current-value*
1

> *external-value*
1
```

```
> *current-value* := 10
10

> *external-value*
10
```

Additional Topics
The Apple Dylan development environment provides many more tools for organizing your source code and examining your runtime environments. See "The Development Environment" on page 326. ◆

Adding Methods Across Libraries

You can add methods to generic functions defined in other libraries, using the same technique shown in "Adding Methods Across Modules" on page 313.

For example, you can add the following definitions to `new-module`:

```
define method double (input :: <string>)
    concatenate (input, input);
end method;

define method times-two (input :: <character>)
    make(<string>, size: 2, fill: input);
end method
```

Both of these method definitions add a method to the same generic function—because `new-module` can use either the name `double` or the name `times-two` to refer to the generic function created in `module-1` (under the name `double`).

You can test the new generic function in any of the three modules. In `module-1`, you must refer to it as `double`; in `module-2`, you must refer to it as `times-two`; in `new-module`, you may refer to it by either name. Nevertheless, all three modules have access to the entire generic function—which now has four possible methods to choose from.

Here are some examples of calling this function from `new-module`:

```
> double(#"test")
#(#"test" . "test")
```

```
> double(3)
6

> double("bye")
"byebye"

> double('b')
"bb"

> times-two(#"test")
#(#"test" . "test")

> times-two(3)
6

> times-two("bye")
"byebye"

> times-two('b')
"bb"
```

As you can see, all four methods are accessible to this module, under either name.

You can control various aspects of dynamism between libraries. For example, when you define a function, you can control whether modules in other libraries can add methods to it or not. See the section "Dynamism," which begins on page 322, for more information.

Subclassing Across Libraries

Just as you can subclass across modules contained in the same library, you can subclass across modules contained in different libraries. For example, you could add this class definition to your `new-module` module:

```
define class <3D-point> (<2D-point>)
    slot point-z, init-keyword: z:, init-value: 0;
end class;
```

You can now create and use <3D-point> objects in new-module:

```
> define variable *point* = make (<3D-point>, x: 10, y: 20, z: 30);
defined *point*

> *point*.point-x;
10

> *point*.point-y;
20

> *point*.point-z;
30
```

The first two of these references work because the variable names point-x and point-y are imported from module-2, and they are bound to the getter functions for the x slot and the point-y slot.

The point-x function is a renaming of the x function from module-1. Since new-module imports everything exported from module-1 as well as module-2, you could reference the x function by its module-1 name:

```
> *point*.x;
10
```

Remember that module-2 does not import the x-setter function from module-1, so there is no point-x-setter function. However, new-module imports everything exported by module-1. Therefore, to modify the slots of a <3D-object> in new-module, you need to refer to the x slot by its module-1 name, the point-y slot by its module-2 name, and the point-z slot by its new-module name:

```
> *point*.x := 100;         // point-x would not work here
100

> *point*.point-y := 200;
200

> *point*.point-z := 300;
300
```

To even out this implementation, you might modify the `new-module` module definition:

```
define module new-module
    use Apple-Dylan;

    use module-1,
        import: all,
        rename: { x-setter => point-x-setter };

    use module-2;
end module;
```

Now you can modify the `x` slot using the `point-x` name:

```
> *point*.point-x := 101;
101
```

As you can with generic functions, you can control the dynamism of a class—that is, whether a class can be subclassed across libraries. The next section, "Dynamism," discusses this subject.

Additional Topics

There are many options available when defining modules and libraries. See the *Dylan Language Manual* and *Using the Apple Dylan Development Environment* for more information.

In particular, you can use the development environment to create stand-alone applications from compiled libraries; you can also use libraries to call Toolbox functions or other C-compatible code. See "The Development Environment" on page 326. ◆

Dynamism

Dynamism is the amount of specialization that a class or a generic function allows, outside of its own library.

Class dynamism determines whether subclasses can be added to a class imported from another library.

Function dynamism determines whether methods can be added to a generic function imported from another library.

Dynamic classes and functions allow you great flexibility, but at the cost of some potential optimization. Sealing classes and functions from dynamism costs flexibility, but allows some potential optimization and allows you more control of how your classes and functions are used.

Sealing Classes

Sealing a class indicates that you do not want direct subclasses to be defined in other libraries. You seal a class by including the adjective `sealed` when you define the class. For example, if `module-2` included this modified definition of the `<2D-point>` class:

```
define sealed class <2D-point> (<1D-point>)
    slot point-y, init-keyword: y:, init-value: 0;
end class;
```

then you would not be able to create the `<3D-point>` subclass in `new-module`. Many Apple Dylan classes are **sealed,** to allow optimizations that would not be possible if they were left **open** (that is, if they could be subclassed across libraries). All of the concrete number classes, for example, are sealed—you cannot create subclasses of `<small-integer>`, `<small-ratio>`, and so on.

Sealing Generic Functions

Generic functions can also be sealed. In fact, Apple Dylan allows you two options when sealing generic functions:

- You can seal the entire generic function; no methods can be added outside the generic function's library.
- You can seal **branches** of the generic function—that is, you can add a method to a generic function and specify that methods that are more specific cannot be added to the function outside of its library.

To seal an entire generic function, you can explicitly define the generic function using the `define generic` form of definition, specifying the `sealed` adjective:

```
define sealed generic double (input);
```

This definition creates a generic function object (with no methods) that does not allow methods to be added outside of its library. Only methods explicitly added in the same library are allowed. Therefore, in the same library, this definition would be valid:

```
define method double (input :: <ratio>)
   (2 * numerator(input)) / denominator(input);
end method;
```

In a different library, you may be able to import and *call* the `double` generic function, but you won't be able to *add* any more methods to it.

Sealing Branches of Generic Functions

Sometimes you don't want to seal an entire generic function, but rather just a **branch** of the generic function. For example, consider these definitions:

```
define method double (input)
    pair (input, input);
end method;

define sealed method double (input :: <number>)
    2 * input;
end method;
```

The first method definition creates a generic function named `double` and adds a method to it. The second method definition adds another method to the generic function, but this method is defined to be **sealed**. As a result, other libraries that import this generic function can add methods to it—but they cannot add any methods that would be more specific than the sealed method.

In this case, the following method definition would be acceptable in another library:

```
define method double (input :: <string>)
    concatenate(input, input)
end method;
```

But this method would not be acceptable in another library, because it is more specific than the sealed method:

```
define method double (input :: <integer>)
    input + input;
end method;
```

Sealing Slot Accessors

A special case of sealing generic function branches is sealing slot accessors. For example, Apple Dylan allows you to specify sealed slots using this syntax:

```
define class <2D-point> (<object>)
    slot x;
    sealed slot y;
end class;
```

The slot specifications in this class definition are equivalent to these method definitions:

```
define method x (input :: <2D-point>)
    // return x slot of input
end method;

define method x-setter (new-value, input :: <2D-point>)
    // set x slot of input to new-value
end method;

define sealed method y (input :: <2D-point>)
    // return y slot of input
end method;
```

```
define sealed method y-setter (new-value, input :: <2D-point>)
    // set y slot of input to new-value
end method;
```

Thus, the generic functions `x` and `x-setter` are completely open—other libraries that import these functions can add methods to them, including methods that are more specific than the two above.

The generic functions `y` and `y-setter` are partially open—other libraries that import these functions can add methods to them, but not methods that are more specific than the two defined above. For example, the function

```
define method y-setter(new-value :: <integer>, input :: <2D-point>)
   // do something
end method;
```

would not be allowed, because it is more specific than the sealed `y-setter` method.

Additional Topics

In this chapter, you learned to bind variable names to objects, create variables with special restrictions, reference variable names to determine the values of variables, and reassign or modify a variable. More advanced and related information is described in the next few sections.

The Development Environment

The Apple Dylan development environment helps you to organize your modules and libraries. However, module definitions and library definitions themselves are standard Dylan source code.

- **Modules**

  Module definitions have various clauses and options not introduced in this book. See the *Dylan Reference Manual* for complete information.

- **Libraries**

  Library definitions also have various clauses and options not introduced in this book; see the *Dylan Reference Manual*. You might also want to read Chapter 21, "C-Compatible Libraries," for related information.

- **Applications**

  The Apple Dylan development environment uses libraries (actually, projects) to create stand-alone applications. See *Using the Apple Dylan Development Environment* for details.

Dynamism

The *Dylan Reference Manual* contains more information about dynamism. In particular, there is one kind of dynamism not introduced in this chapter:

- **Runtime Dynamism**

  Since classes and functions are objects, you can use the `make` function to created new classes and functions at runtime. Classes and functions created at runtime are also subject to certain dynamism restrictions.

CHAPTER 18

Modules

CHAPTER 19

Macros

Contents

Key Concepts 331
About Macros 332
  Definition Macros 333
  Statement Macros 336
  Function Macros 338
Defining and Using Macros 339
  Conditional Evaluation 339
  Free Variables 342
  Nested Macro Calls 343
  Recursive Macro Calls 344
  Input/Output Arguments 345
Additional Topics 346

Throughout this book, you've seen examples of built-in macros, such as the built-in statement macros. This chapter introduces the concept of macros in more detail and shows how you can extend the Apple Dylan language by defining your own macros.

Key Concepts

About Macros

- A **macro** is a set of rules for translating a constituent (called a **macro call**) into an equivalent set of constituents (called the **expansion** of the macro call).
- A **definition macro** is a macro that allows you to create bindings in the runtime environment or define certain kinds of information about your program for the development environment. Built-in definition macros include `define variable`, `define class`, and `define module`.
- A **statement macro** is a macro that allows you to create bodies of code and execute those bodies conditionally or iteratively. Built-in statement macros include `begin`-`end` statements, `if` statements, and `for` statements.
- A **function macro** is a special kind of macro. Calls to a function macro look like function calls, but actually result in macro expansion like other kinds of macros.

Defining and Using Macros

- When you define a macro, you specify **rules.** Each rule specifies a **pattern,** which Apple Dylan pattern matches against calls to the macro, and a **template,** which Apple Dylan uses to expand the macro call.
- You can use macros to perform **conditional evaluation** on their arguments.
- You can include **free variables** in a macro template. These variables reference values in the scope of the macro definition, rather than the macro call.
- You can include **nested** and **recursive** macro calls.
- You can use macros to change bindings of input arguments, effectively creating **input/output arguments.**

About Macros

A **macro** is set of rules for translating a constituent (called a **macro call**) into a different, equivalent set of constituents (called the **expansion** of the macro call).

For example, consider the following `unless` statement:

```
unless ( *global* = 0 )  1 / *global*  end;
```

The `unless` macro could be defined as a set of rules that would expand the above statement (a macro call to the unless macro) into the following, equivalent statement:

```
if ( ~ ( *global* = 0 ) )  1 / *global*  end;
```

Of course, the unless macro and the if macro are **built-in macros;** this example is simply for illustrative purposes. However, later in this chapter, you'll see how to define your own macros which expand into calls to the built-in macros.

In your own **macro definitions,** you'll define **patterns,** which Apple Dylan uses to pattern match macro calls, and **templates,** which Apple Dylan uses when expanding calls to your macro.

Because macros perform pattern matching and template expansion, you can use them to create effects you cannot create with functions alone. In particular, you can use macros to conditionally evaluate arguments, and to create new statement and definition syntaxes.

The next three sections introduce the three kinds of macros:

- Defining Macros
- Statement Macros
- Function Macros

and show how you can create a simple macro of each.

Definition Macros

Definition macros allow you to

- create new objects and bindings in a runtime environment
- specify information about your program to the development environment

For example, the built-in `define variable` definition macro allows you to create a new binding in a runtime environment; the `define module` macro allows you to specify importing and exporting information to the development environment. Built-in definition macros include `define constant`, `define variable`, `define class`, `define module`, and `define library`.

Here is an example of a macro call to a built-in definition macro:

```
define variable *global* = 37
```

Let's examine the parts of this macro call:

- The first word of this macro call is the word `define`; this word indicates that a definition macro is being called.
- The second word is `variable`. This word is a **defining word;** it indicates what kind of definition this macro call is. Defining words are reserved words—you cannot use them anywhere else or for any other purpose (such as a variable name) in your Apple Dylan program. (This is a deviation by Apple Dylan from the standard Dylan language.)
- The third word is `*global*`. This word is a **macro argument.**
- The fourth word is the `=` punctuation, which the `define` variable macro uses to separate the variable-name argument and the initial-value argument.
- The fifth word is the literal constant `37`. It is also an argument to the macro.

Macro calls to the built-in `define variable` macro have this syntax:

`define variable` *variable-name* `=` *initial-value-expression*

In the language of macro **patterns,** this syntax is expressed:

```
define variable ?name:variable = ?initial-value:expression
```

This macro pattern matches syntactically valid `define variable` macro calls. That is, this pattern matches any constituent that starts with the words `define variable`, then contains a variable name, then contains an = sign, and finally contains an expression.

You can extend the set of definitions available to your program by creating your own definition macros. For example, consider the following macro definition:

```
define macro boolean-variable
{ define boolean-variable ?name:name = ?value:expression }
  =>
    { define variable ?name :: <Boolean> = (?value ~= #f) }
end macro;
```

(Note that this syntax is a deviation from standard Dylan as described in the *Dylan Reference Manual*.)

This macro definition defines a new kind of definition macro: the `boolean-variable` definition macro. This macro definition consists of four parts:

- The first line

  ```
  define macro boolean-variable
  ```

  indicates that a macro is being defined, and specifies the name of the macro.

- The second line

  ```
  { define boolean-variable ?name:name = ?value:expression }
  ```

  specifies the pattern that Apple Dylan recognizes as a macro call to this macro. In particular, it specifies that the syntax of boolean-variable macro call is

  `define boolean-variable` *variable-name* = *value-expression*

  In addition, this pattern associates the **pattern variable** `?name` with the actual name provided in a call to this macro; it also the associates the pattern variable `?value` with the actual value expression specified in the macro call.

- The third line

  ```
  =>
  ```

  is punctuation that separates the pattern from its corresponding template.

- The fourth line

  ```
  { define variable ?name :: <Boolean> = (?value ~= #f) }
  ```

  is the template that specifies how macro calls are expanded.

As an example of using this macro, consider the following macro call:

```
define boolean-variable *boolean-global* = *global*
```

This macro call matches the pattern specified in the second line of the macro definition. In particular, this particular pattern matching results in the pattern variable `?name` being associated with the variable name `*boolean-global*` and the pattern variable `?value` being associated with the expression `*global*`.

Using the expansion rule defined by the `boolean-variable` macro, this macro call expands to

```
define variable *boolean-global* :: <Boolean> = (*global* ~= #f)
```

This expansion is the result of taking the pattern

```
define variable ?name :: <Boolean> = (?value ~= #f)
```

and substituting `*boolean-global*` for `?name` and `*global*` for `?value`.

In sum, the **macro call**

```
define boolean-variable *boolean-global* = *global*
```

matches the **pattern**

```
define boolean-variable ?name:variable = ?value:expression
```

which expands using the **template**

```
define variable ?name :: <Boolean> = (?value ~= #f)
```

into the **constituent**

```
define variable *boolean-global* :: <Boolean> = (*global* ~= #f)
```

This constituent is then executed in place of the original macro call. As a result, a new binding is created in the runtime, with the name `*boolean-global*` and either the value `#t` or the value `#f`, depending on the value of the variable `*global*` at the time the macro call is executed.

Notice that the `define boolean-variable` macro is defined in terms of the built-in `define variable` macro. It is common for the macros you define to expand their macro calls into similar macro calls using built-in macros.

Statement Macros

Statement macros allow you to

- organize bodies of code
- execute bodies of code conditionally
- execute bodies of code iteratively

For example, the built-in `if` statement macro allows you to execute a body of code if a certain condition is met. The built-in `until` statement allows you to execute a body of code repeatedly until a specified condition is met.

Here is an example of a macro call to the built-in `if` statement macro:

```
if ( *global* )
    perform-action();
    calculate-result()
end
```

This macro call examines the value of the variable `*global*`. If it is not `#f`, then the two function calls in the body are evaluated. If it is `#f`, these function calls are not evaluated at all.

Here is an example macro definition that defines a statement macro:

```
define macro if-true
    { if-true (?e:expression)     // pattern
          ?b:body
      end }
      =>
        { if ( ?e = #t )          // template
```

```
            ?b
        end;  }
end;
```

The **pattern** in this macro definition is

```
if-true (?e:expression)
    ?b:body
end
```

Therefore, this macro matches macro calls that have the syntax

```
if-true (expression)
    body
end
```

During a macro call to this macro, the pattern variable `?e` is associated with the specified expression and the pattern variable `?b` is associated with the specified body.

The **template** in this macro definition is

```
if ( ?e = #t )
    ?b
end;
```

which uses the pattern variables `?e` and `?b` to translate an `if-true` macro call into a call to the built-in `if` macro. For example, consider this macro call:

```
if-true ( *global* )
    perform-action();
    calculate-result()
end
```

This macro call matches the pattern specified for the `if-true` macro. The pattern variable `?e` is associated with the expression `*global*` and the pattern variable `?b` is associated with the body of this macro call—the two function calls.

After macro expansion, this macro call becomes

```
if ( *global* = #t )
    perform-action();
    calculate-result()
end
```

which is a call to the built-in `if` macro. This `if` statement is then evaluated in place of the `if-true` macro call. By expanding into an `if` statement, the `if-true` macro can execute a body of code conditionally.

Function Macros

Function macros are the third kind of macro. Function macros are different from definition macros or statement macros in that calls to a function macro look just like function calls.

However, a function call always evaluates its arguments (and passes them to a function). A function macro call does not necessarily evaluate its arguments. Instead, it uses them in the expansion of the macro call. For example, consider the following function macro definition:

```
define macro both-true?
    { both-true? (?e1:expression, ?e2:expression) }  // pattern
      =>
        { ?e1 = #t  &  ?e2 = #t }                    // template
end;
```

This macro definition defines a function macro named `both-true?`. This function macro takes two arguments, both of which can be any expression. The first expression is associated with the pattern variable `?e1`, and the second expression is associated with the pattern variable `?e2`. Here is an example of a macro call to this function macro:

```
both-true? ( *done*, *total* > 3 )
```

The pattern variable `?e1` is associated with the expression `*done*` and the pattern variable `?e2` is associated with the expression `*total* > 3`. When this macro call is expanded, the result is

```
*done* = #t  &  (*total* > 3) = #t
```

This expression is then evaluated in place of the `both-true?` macro call.

Notice that parentheses are automatically added around the expression `*total* > 3`. In this case, the parentheses don't make any difference, but if the expression contained an operator of lower precedence than the equals operator, the parentheses would be necessary to retain the meaning of the original template.

Defining and Using Macros

The rest of this chapter provides examples of macro definitions, patterns, templates, macro expansion, and macro calls. For simplicity, these examples all use function macros. The principles introduced here apply to definition and statement macros as well; you should see the *Dylan Reference Manual* for more information about all three kinds of macros.

Conditional Evaluation

A primary reason for using a function macro rather than a function is that functions always evaluate their arguments, whereas function macros have no such restriction.

For example, imagine that you implemented `both-true?` as a function, rather than a function macro:

```
define method both-true? (e1, e2)
    (e1 = #t) & (e2 = #t);
end;
```

Calls to `both-true?` always evaluate both arguments. For example, the function call

```
both-true? ( *done*, *finished* )
```

evaluates the variable reference `*done*` and the variable reference `*finished*`, and sends the resulting values to the function.

Imagine that the arguments to the function are not variable references, however, but slot references:

```
both-true? ( my-object.realslot, my-object.virtualslot )
```

In this case, the arguments are evaluated before the function is called. To evaluate the first argument, Apple Dylan calls the `realslot` getter function on the object `my-object`. For the second argument, Apple Dylan calls the `virtualslot` getter function on that object. As a result of this call to the `both-true?` function, both of these getter functions are executed. If either of them produces any side effects, then those side effects will always result from this call to the `both-true?` function.

Now, recall the macro definition for `both-true?`:

```
define macro both-true?
    { both-true? (?e1:expression, ?e2:expression) }  // pattern
      =>
        { ?e1 = #t  &  ?e2 = #t }                    // template
end;
```

This macro expands macro calls into calls to the `&` operator. For example, the macro call

```
both-true? ( my-object.realslot, my-object.virtualslot )
```

expands into the equivalent expression

```
my-object.realslot = #t  &  my-object.virtualslot = #t
```

which is evaluated in place of the macro call. Remember from Chapter 13 that the `&` operator evaluates its first argument, and *only evaluates its second argument* if the first argument is not `#f`. Therefore, the macro call

```
both-true? ( my-object.realslot, my-object.virtualslot )
```

calls the `virtualslot` function on `my-object` if the value returned by calling `realslot` is equal to `#t`. As a result, you can make this macro call to `both-true?` without necessarily evaluating both arguments. If `both-true?` were a function, this would not be possible.

Note that this conditional evaluation of arguments is possible because the macro can expand macro calls into built-in expressions, like calls to the `&` operator, that conditionally evaluate their arguments.

A second, equivalent, definition of the `both-true?` macro follows. This definition expands calls to `both-true?` into `if` statements.

```
define macro both-true?
    { both-true? (?e1:expression, ?e2:expression) }    // pattern
      =>
        { if ( ?e1 = #t )                              // template
              ?e2 = #t
           end;  }
end;
```

As a result, this macro call

```
both-true? ( *global* > 0, (1 / *global*) < 1 )
```

expands to this `if` statement:

```
if ( (*global* > 0) = #t )
    ( (1 / *global*) < 1) = #t )
end if
```

Notice that Apple Dylan surrounds the input expressions with parentheses to avoid operator-call ambiguity. Also notice that, since the macro call is expanded to an `if` statement, the second expression

```
(1 / *global*) < 1
```

is evaluated only if the first expression

```
*global* > 0
```

evaluates to `#t`.

Free Variables

A **free variable** is a variable that appears in a macro template that is not specifically bound by that macro. For example, consider this macro definition:

```
define macro both-equal-global?
    { both-equal-global? (?e1:expression, ?e2:expression) }
      =>
        { (?e1 = *global*  &  ?e2 = *global*) }
end;
```

The template references the pattern variables `?e1` and `?e2`, which are associated with expressions when a call to this macro is being pattern matched. The template also includes a reference to the free variable `*global*`, which is not given a value anywhere within the macro.

When this macro is called, the free variable references the current value of the variable `*global*` in the scope where the macro is *defined* (rather than the scope where the macro is *called*). For example, consider this macro call:

```
both-equal-global? ( *done*, *finished* )
```

It is expanded to the following expression:

```
( *done* = *global*  &  *finished* = *global* )
```

In this expansion, the variable names `*done*` and `*finished*` reference variables in the scope of the macro *call*, while the variable name `*global*` references a variable in the scope of the macro *definition*. Imagine that the earlier macro call occurred in a body of code that shadowed the original `*global*` variable:

```
define variable *global* = 10;

// . . . macro defined here . . .

begin
    let *global* = 100;
    let *done* = 100;
    let *finished* = 100;
    both-equal-global?(*done*, *finished*);
end;
```

This call to the `both-equal-global?` macro returns `#f`, because the variable reference to `*global*` in the macro call expansion refers to the current value of the variable named `*global*` in the scope of the macro definition (which is 10) rather than the value of the `*global*` variable in the scope of the macro call (which is 100).

Nested Macro Calls

The expansion of a macro call can contain other macro calls. For example, consider this implementation for the `both-equal-global?` macro:

```
define macro both-equal-global?
    { both-equal-global? (?e1:expression, ?e2:expression) }
      =>
        { both-true? (e1 = *global*, ?e2 = *global* ) }
end;
```

In this example, the expansion template contains a call to the `both-true?` macro. As a result, this macro call

```
both-equal-global? ( *done*, *finished* )
```

expands to this macro call

```
both-true? ( *done* = *global*, *finished* = *global*)
```

which, in turn, expands to this expression

```
*done* = *global*  &  *finished* = *global*
```

Recursive Macro Calls

In addition to calling other macros, a macro expansion can contain calls to its own macro. For example, consider this macro definition:

```
define macro all-true?
    { all-true? ( ) }                              // first rule
      =>
        { #t }

    { all-true? (?e1:expression, ?more) }          // second rule
      =>
        { ( ?e1 = #t  &  all-true?(?more)) }
end;
```

This macro definition creates a macro named `all-true?` that takes a variable number of arguments. This macro definition contains two rules—that is, two *pattern* => *template* productions. The first rule pattern matches a call to `all-true?` with an empty list of arguments and expands such a call into the literal constant `#t`.

The second rule tests if the first argument is equal to `#t`, and if so, it tests if the second argument is equal to `#t`, and so on. It does this using two pattern variables. The first pattern variable, `?e1`, is associated with the first argument expression in a macro call, and the second pattern variable, `?more`, is associated with the comma-separated series of remaining argument expressions.

As an example, consider this macro call:

```
all-true? (#t, 3 = 3)
```

This macro call, using the first rule, expands to the following expression:

```
( #t = #t  &  all-true? (3 = 3) }
```

This expression contains another call to the `all-true?` macro. Using the first rule again, the expression expands into

```
( #t = #t  &  ( 3 = 3  &  all-true? ( ) ) )
```

This expression also contains a call to the `all-true?` macro. Using the second rule, this macro call expands to `#t`, so the resulting expression is

```
( #t = #t  &  ( 3 = 3  &  #t ) )
```

This expression, then, is evaluated in place of the original macro call.

Input/Output Arguments

In addition to conditional evaluation of arguments, macros also allow you to change the bindings of input arguments, effectively creating input/output arguments. For example, consider this macro definition:

```
define macro switch!
    { switch! (?e1:expression, ?e2:expression) }
      =>
        { let (_e1, _e2) = values(?e1, ?e2);
          ?e1 := _e2;
          ?e2 := _e1;}
end;
```

This macro definition takes two expressions as its arguments, and switches their values.

For example, the macro call

```
switch! (*global*, my-object.slot-x)
```

expands to

```
let (_e1, _e2) = values(*global*, my-object.slot-x);
*global* := _e2;
my-object.slot-x := _e1;
```

(Apple Dylan automatically surrounds this body of code with `begin` and `end`, as necessary.)

The macro call, then, results in the value of the `*global*` variable and the value of the `slot-x` slot of `my-object` being switched. Both values are permanently

changed; after the macro call returns, accessing `*global*` or `my-object.slot-x` results in their new values.

Input/output arguments like the ones shown here are not possible with function definitions in Apple Dylan; you must use function macros for this effect.

Additional Topics

This chapter included many simple examples of defining your own macros. However, there are many additional aspects to macro definition, such as macro hygiene, macro scope and extent, auxiliary rule sets, importing and exporting macros, and so on. See the *Dylan Reference Manual* for more information and additional examples.

CHAPTER 20

Conditions

Contents

Key Concepts 349
About Conditions 350
Signaling Conditions 351
  Signaling Errors 351
  Signaling Warnings 353
  Signaling Your Own Condition Classes 354
Handling Conditions 355
  Establishing A Handler 356
  Handling Your Own Conditions 358
  Handling a Condition By Declining 359
  Handling a Condition By Returning 360
  Handling a Condition By Restarting 361
  Handling a Condition By Taking a Non-Local Exit 363
Additional Topics 364

This chapter introduces the exception-handling mechanism of Apple Dylan and shows how you can signal and handle exceptional conditions.

Key Concepts

About Conditions

- A **condition** is an object that represents an exceptional situation.
- When an exceptional situation arises, Apple Dylan, or your program, creates a condition object and **signals** that object.
- A condition **handler** is a function that responds when a condition is signaled.
- The three main kinds of conditions are **errors, warnings,** and **restarts.**

Signaling Conditions

- You signal a condition by creating a condition object and passing that object to the built-in `signal` function.
- You can signal conditions of the built-in simple-error and simple-warning classes to provide diagnostic messages to the Listener window.

Handling Conditions

- All condition classes have a built-in **default handler.**
- You can establish **dynamic handlers** that shadow the default handler for specific extents during your program's execution.
- Your handlers can respond to a condition in one of four ways: **declining** to handle the condition, **recovering** from the condition by returning to the signaling code, **terminating** from the condition by taking a non-local exit out of the signaling code, and **restarting** from the condition by signaling a restart condition.

About Conditions

In Apple Dylan, a **condition** is an object that represents an exceptional situation, such as a divide-by-zero error. When such a condition arises, Apple Dylan, or your program, creates a condition object and **signals** that object.

For every class of condition object, there is a **default handler,** which handles the condition in some default way (such as printing a message to Listener window or simply ignoring the condition altogether).

You can also establish **dynamic handlers** for the different classes of condition objects. When a condition object is signaled, Apple Dylan calls the most recently established dynamic handler for that condition object (if there is one). That handler has the option of handling the condition or declining to handle the condition. If it declines to handle the condition, it passes the condition object on to the next most recently established dynamic handler. This pattern can repeat until the default handler for the condition is reached.

Handlers can respond to a condition by handling the condition and returning to the signaling code (called **recovering** from the condition), or by taking a non-local exit from the signaling code to an earlier state of computation (called **terminating** from the condition).

There are three main categories of condition classes:

- **Errors,** which correspond to exceptional conditions that cannot be safely ignored.
- **Warnings,** which correspond to exceptional conditions that can be safely ignored or easily corrected, but may be of some interest to the user.
- **Restarts**, which are a special kind of condition object that provide a formal interface for restarting computation after another condition has been signaled.

Each condition class has its own **recovery protocol**—the manner in which handlers for the condition class are expected to behave.

Each condition class also has a built-in default handler.

Signaling Conditions

Signaling Errors

The built-in default handler for errors takes a non-local exit, and then:

- If the application is connected to the development environment, the handler prints a message to the Listener window.
- If the application is not connected to the development environment (that is, it is a stand-alone application), the handler displays a dialog box with a *Quit* button, which quits the application.

and prints a message to the Listener window. As an example, when your code contains an attempt to divide by zero, Apple Dylan signals an error. If you haven't installed any of your own handlers, this error is handled by the default handler, as shown here:

```
> 1 / 0
Error: Divide by zero: 1 / 0
```

In this case, Apple Dylan detects the error and signals it for you. You can signal errors yourself by calling the built-in function `signal`. This function has one required argument—an instance of one of the condition classes. Here is an example:

```
> signal (make (<error>) )
Error: #<<error> id: 25>
```

In this example, the `make` function is used to instantiate the class `<error>`, and the resulting error object is sent to the `signal` function. The default handler for errors is invoked, and it prints an error message to the Listener window.

Apple Dylan provides the subclass `<simple-error>` to provide more informative error messages. When you instantiate `<simple-error>`, you can provide a string for the default error handler to print as the error message. Here is an example:

```
> signal (make (<simple-error>, format-string: "To err is human.") )
Error: To err is human.
```

The `format-string:` keyword allows you to specify a **format string,** which is similar to the kind of string accepted by the `printf` function of C. The following example shows how to use the format string to include multiple pieces of information in the error message. In this example, the format string contains the symbol `%d` to indicate that a decimal number is to be inserted in the error message, and the symbol `%s` to indicate that a string is to be inserted in the error message. The actual number and string are provided in a list as the `format-arguments:` keyword argument.

```
define constant signal-simple-error =
    method (count, message)
        signal (make (<simple-error>,
                      format-string: "(error #%d)  %s",
                      format-arguments: list(count, message) ) );
        "A string to return";
    end method;
```

Notice the last line of this method provides a value for the `signal-simple-error` method to return. However, an error is signaled before that line, and the default error handler does not return control to the signaling method.

Therefore, the final line of code is never evaluated, and the method returns no value:

```
> signal-simple-error(237, "All is not well.")
Error: (error #237) All is not well.
```

The number 237 and the string `"All is not well."` are inserted into the error message, but no value is returned from the method call.

Additional Topics

Other built-in functions, such as `error` and `cerror`, are available for you to use to signal errors. Also, format strings allow you to insert a variety of data types not mentioned in this section. See the *Dylan Reference Manual* for complete information. ◆

Signaling Warnings

You signal warnings in the same way that you signal errors. The main difference is that the default handler for warnings *does* return control to the signaling method.

For example, suppose you rewrote the method from the previous section to create and signal a `<simple-warning>` object:

```
define constant signal-simple-warning =
    method (count, message)
        signal (make (<simple-warning>,
                      format-string: “(warning #%d)  %s”,
                      format-arguments: list(count, message) ) );
        "A string to return";
    end method;
```

Calling this method results in a warning message, as the previous example resulted in an error message.

However, as the following example shows, control is returned to the signaling method, and the final line of code is evaluated and returned as the method result:

```
> signal-simple-warning(12, "Something awry is afoot.")
Warning: (warning #12)  Something awry is afoot.
"A string to return"
```

Apple Dylan provides a shorthand for creating and signaling a `<simple-warning>` object. Instead of this code:

```
signal (make (<simple-warning>,
              format-string: “(warning #%d)  %s”,
              format-arguments: list(count, message) ) );
```

you can simply provide the `signal` function with a format string and a series of format arguments, as shown here:

```
> signal ( "(warning #%d)  %s", 13, "Never mind." );
Warning: (warning #13)  Never mind.
```

Apple Dylan automatically creates a <simple-warning> object for you when you use this version of the signal method. This mechanism provides a simple way for you to print diagnostic messages to the Listener window while debugging your program, without interrupting your program's flow of control.

Since a simple warning is signaled, the default handler for simple warnings is invoked, and control returns to the signaling method, as shown in the following example:

```
define constant signal-simple-warning =
    method (count, message)
        signal ( "(warning #%d)  %s", count, message) ) );
        "A string to return";
    end method;

> signal-simple-warning(12, "Something awry is afoot.")
Warning: (warning #12)  Something awry is afoot.
"A string to return"
```

Signaling Your Own Condition Classes

In the previous two sections, you saw how to signal some built-in conditions: errors, warnings, and their subclasses simple errors and simple warnings. You can also create your own subclasses of <error> and <warning>, for example:

```
define class <my-warning> (<warning>)
end class;
```

When you signal an object of this class, the default warning handler is invoked, which prints a warning message and returns control to the signaler:

```
> signal (make (<my-warning>) );
Warning: #<<my-warning> id: 25>
```

The rest of this chapter shows how you can override the default handlers, by creating and establishing handlers of your own.

Handling Conditions

Whenever a condition is signaled, Apple Dylan passes control to a **handler** which responds to the condition. Signaling a condition always results in either **recovery** or **termination.** In particular, handlers can respond to conditions in one of these ways:

- The handler can perform some action(s) and return directly to the signaling method. In addition, the handler can return values, which the signaling method sees as the function results returned by the call to `signal`. This process is described in "Handling a Condition By Returning" on page 360.
- The handler can choose not to return to the signaler, by taking a non-local exit. This process typically uses the `block` statement, and is described in "Handling a Condition By Taking a Non-Local Exit" on page 363.
- The handler can handle the condition by creating a restart object and signaling that object. The handler for that restart object then decides whether to recover (and return to the signaler) or to terminate (by taking a non-local exit). This process is described in "Handling a Condition By Restarting" on page 361.
- The handler can decline to handle the condition, by passing the condition to the next handler. The next handler then decides whether to handle the condition by recovering or terminating. This process is described in "Handling a Condition By Declining" on page 359.

Apple Dylan provides two basic mechanisms for establishing a handler:

- You can use a `let handler` local declaration. This declaration establishes a condition handler for the dynamic extent of the enclosing body of code.
- You can the exception clause of a block. This clause also establishes a condition handler for the dynamic extent of the block. However, these handlers cannot be used for recovery; they can only be used for making non-local exits.

Establishing A Handler

When you establish a handler, you specify the class of condition that the handler responds to, and the function that handles the condition. This function takes two arguments:

- the condition object that was passed to `signal`
- the next most recently established handler

Here is an example of a simple handler:

```
define constant simple-handler =
    method (condition, next-handler)
        // do nothing but return
    end;
```

This handler accepts two arguments—the signaling condition and the next handler—but it simply ignores them and immediately returns. Here is an example that establishes this simple handler and then signals a condition that is handled by it:

```
define constant handle-a-warning =
    method ()
        let handler <simple-warning> = simple-handler;

        signal("This is a simple warning.");

        "A string to return.";
    end method;
```

In this example, the local declaration

```
let handler <simple-warning> = simple-handler;
```

establishes the `simple-handler` handler as the handler for conditions of class `<simple-warning>`. This handler is established for the dynamic extent of the body of `handle-a-warning` method. When a simple warning is signaled during the execution of the body of this method, Apple Dylan calls the `simple-handler` method, passing it the signaling `<simple-warning>` object (as the first argument) and the next most recently established handler for simple warnings (as the

second argument). If you haven't established any other handlers for simple warnings, the default handler for simple warnings is passed as the next-handler argument.

Calling the `handle-a-warning` method shows that the `simple-handler` method ignores the simple warning completely and returns control to the signaling line of code. The `handle-a-warning` method then evaluates its last line of code, and returns its value normally:

```
> handle-a-warning();
"A string to return."
```

Notice that no warning message is printed to the Listener window, because your dynamically-established handler shadows the default simple-warning handler for the extent of the `handle-a-warning` method.

As another example, consider the handler:

```
define constant decline =
    method (condition, next-handler)
        next-handler();
    end;
```

This handler declines to handle a condition by simply calling `next-handler` function, which is passed in as the second argument. Notice that you do not need to pass any arguments to this function—it automatically knows how to call the next handler with the appropriate arguments.

The next handler is either the next most recently established dynamic handler, or the default handler if no other dynamic handlers have been established. Here is an example that uses `decline`:

```
define method handle-an-error()
    let handler <error> = decline;

    signal(make(<error>));

    "A string to return.";
end method;
```

This `method` establishes `decline` as the handler for all errors for the extent of the method. Then, it signals an error. In response, Apple Dylan calls the decline

method, passing it the <error> object and the next handler, which in this case is the default <error> handler.

```
> handle-an-error();
Error: #<<error> id: 25>
```

Notice that the default error handler, which prints a message to the Listener window and does not return control to the signaler, is called in response to the error signaled in handle-an-error.

Handling Your Own Conditions

In the previous section, you saw how to establish handlers for built-in condition classes like <simple-warning> and <error>. In the remaining sections of this chapter, you'll learn how to create your own class of conditions and you'll learn four different ways to handle them.

Suppose you want to provide condition-handling for the condition of unbound slots. You can define an <unbound-slot> condition class using this code:

```
define class <unbound-slot> (<warning>)
    slot object, required-init-keyword: object:;
    slot setter, required-init-keyword: setter:;
end;
```

In this example, the class <unbound-slot> is a subclass of <warning>. This new condition class has two slots:

- The object slot is intended to contain a reference to the object that has the unbound slot.
- The setter slot is intended to contain a reference to the setter function that sets the unbound slot of the object.

The next four sections show you four ways to create, signal, and handle an unbound-slot condition.

Handling a Condition By Declining

You can decline to handle your own condition classes just as you decline to handle the built-in condition classes—by calling the next handler. For example:

```
define method decline (condition, next-handler)
    next-handler();
end method;

define method get-x-coordinate(point :: <2D-point>)
    let handler <unbound-slot> = decline;

    if (~slot-initialized(x, point))
        signal(make(<unbound-slot>, object: point, setter: x-setter));
    else
        point.x;
    end if;
end method;
```

In this example, the `get-x-coordinate` function takes a `<2D-point>` object and returns the value of the `x` slot. Before calling the `x` getter function, however, it checks to be sure that the `x` slot is initialized. If it is not, the function signals an `<unbound-slot>` condition, initializing the slots of the `<unbound-slot>` condition with the `<2D-point>` object and the `x-setter` setter function.

In response to this call to `signal`, Apple Dylan calls the most recently established handler for `<unbound-slot>` conditions. In this case, the most recently established handler is the method `decline`, which is established as a handler for unbound slots at the beginning of the `get-x-coordinate` method. This handler simply calls the next handler, which in this simple case is the default handler for warnings, which simply prints a warning to the Listener window and returns to the signaling function.

Notice that his handler does nothing at all; you would not need to define such a handler in your application. However, you might want to define a handler that performs some tests, and then decides whether to decline or to handle a condition.

Handling a Condition By Returning

Instead of declining to handle a condition, your handler can handle a condition by performing any necessary actions and returning directly to the signaling method. For example, in the case of unbound slots, your handler can use the information in the `<unbound-slot>` object to set the unbound slot, and return the value as the result of the handler:

```
define method always-handle-unbound-slots (condition, next-handler)
    condition.setter(0, condition.object);
end method;
```

This handler ignores the `next-handler` argument. Instead of declining to handle the condition, it uses the information in the `setter` and `object` slots of the condition to set the unbound slot to `0`, and this value is returned as the result of the handler. Here is an implementation of `get-x-coordinate` using this handler:

```
define method get-x-coordinate(point :: <2D-point>)
    let handler <unbound-slot> = always-handle-unbound-slots;
    if (~slot-initialized(x, point))
        signal(make(<unbound-slot>, object: point, setter: x-setter));
    else
        point.x;
    end if;
end method;
```

If the `get-x-coordinate` method is passed a `<2D-point>` object with an unbound `x` slot, it signals an `<unbound-slot>` condition, and initializes the slots of that condition with a reference to the `<2D-point>` object and a reference to the `x-setter` setter function.

In response to this call to `signal`, Apple Dylan calls the most recently established handler for `<unbound-slot>` conditions, which is the `always-handle-unbound-slots` function. This function handles the condition by setting the indicated slot of the indicated object to the value 0. The handler returns the value 0 to the signaling function—which is `get-x-coordinate`. The `get-x-coordinate` function, in turn, returns the value returned by `signal` as its function result. In this way, the `get-x-coordinate` always returns the value of the `x` slot of its input point, even if it has to initialize that slot itself.

Handling a Condition By Restarting

In the previous sections, you've seen how to handle a condition by declining or returning. Apple Dylan provides a more formal condition-handling mechanism called restarting. To restart, your handler does not handle a condition itself. Instead, it creates a restart object and signals that object. The most recently established handler for that class of restart object, then, actually handles the condition.

As an example, consider the following definition subclassing `<restart>`:

```
define class <new-value> (<restart>)
    slot caller :: <condition>, required-init-keyword: caller:;
    slot value, required-init-keyword: value:;
    slot permanent :: <Boolean>, required-init-keyword: permanent;
end;
```

The `<new-value>` class is a condition class; in particular it is a restart class. It has three slots:

- The `caller` slot is intended to contain a reference to the condition object that signaled the restart.
- The `value` slot is intended to contain the suggested initial value for an unbound slot.
- The `permanent` slot is intended to contain a Boolean value indicating whether the suggested value is meant to be simply returned, or whether the unbound slot should be initialized with the value.

Here is a sample handler for `<new-value>` restart conditions:

```
define method handle-new-value (restart, next-handler)
    let object = restart.caller.object;
    let setter = restart.caller.setter;
    if applicable-method? (setter, restart.value, object) then
        if (restart.permanent)
            setter(restart.value, object);
        else
            restart.value;
        end if;
    else
```

```
        next-handler();
    end if;
end method;
```

This handler determines if the suggested initial value for the unbound slot is an acceptable value for the slot. If not, the handler declines to handle the condition and the next handler is called. If the suggested value is acceptable, then the handler determines if the suggested value is meant to be permanently stored in the unbound slot. If so, it stores the value in the unbound slot and returns the value. Otherwise, it simply returns the value without storing it in the unbound slot.

Here are sample methods that use the `<new-value>` restart condition:

```
define method try-to-handle-unbound-slots (condition, next-handler)
    signal(make(<new-value>, caller: condition,
                             value: 0,
                             permanent:#t);
end method;

define method get-x-coordinate(point :: <2D-point>)
    let handler <new-value> = handler-new-value;
    let handler <unbound-slot> = try-to-handle-unbound-slots;

    if (~slot-initialized(x, point))
        signal(make(<unbound-slot>, object: point, setter: x-setter));
    else
        point.x;
    end if;
end method;
```

In this example, the `get-x-coordinate` function determines if the `x` slot of the input `<2D-point>` argument is initialized and, if it is not, the function signals an `<unbound-slot>` condition, as in previous examples.

In this example, however, the handler for `<unbound-slot>` conditions does not handle the condition itself. Instead, it creates a `<new-value>` restart condition, initializes the slots of that restart condition, and then signals the restart. The handler for the `<new-value>` restart condition, then, decides how to handle the condition.

Restarts are typically useful when you create them further up the stack, in a different function than the error handler.

Handling a Condition By Taking a Non-Local Exit

The fourth and final way to handle a condition is to make a non-local exit—that is, to handle the condition but not return to the signaling code, but by returning to some earlier point in your program's computation. Apple Dylan provides the `block` statement for this purpose.

A block statement allows you to define a body of code and an **exit function.** Anywhere within the dynamic extent of the body of the block, you can call the exit function to make a non-local exit. Whenever the exit function is called, the remainder of the body of code is not executed, and control is transferred to the end of the block statement.

Block statements may also contain **exception clauses,** which allow you to establish condition handlers for the extent of the block. They can also contain other types of clauses that define protected bodies of code—bodies whose contents are executed even if a non-local exit occurs.

The following code shows a simple example using a `block` statement. The exit function is named `return`, and the block establishes a single condition handler, which simply calls the exit function, effectively handling the condition by taking a non-local exit.

```
define class <unbound-slot-error> (<error>)
end class;

define method use-x-coordinate(point :: <2D-point>)

    block (return)
        if (~slot-initialized(point, x))
            signal(make(<unbound-slot-error>));
        end if;

        calculate(point.x);

    exception (<unbound-slot-error>)
        // respond to error;
```

```
    end block;

end method;
```

In this `block` statement, an `<unbound-slot-error>` condition is signaled if the x slot of the `<2D-point>` input argument is uninitialized. The `block` statement has an exception clause that establishes a handler for `<unbound-slot-error>` conditions. This handler responds to the error, and then passes control to the end of the block.

Additional Topics

There are other clauses you can use with block statements, and many repercussions of using nested block statements. See the *Dylan Reference Manual* for complete information. ◆

Additional Topics

This chapter included many simple examples of signaling and handling conditions. See the *Dylan Reference Manual* for more information and additional examples.

CHAPTER 21

C-Compatible Libraries

Contents

Key Concepts 367
About C-Compatible Libraries 368
  Interface Definitions 368
  Access Paths 370
Importing C-Compatible Functions 371
  Importing a Macintosh Toolbox Function 371
  Importing a Shared Library Function 373
Importing Other C Entities 375
  Importing C Constants 378
  Importing C Structures 381
  Importing C Structure Pointer Types 383
  Importing C Function Pointer Types 384
Callouts and Callbacks 386
  Callouts: Calling C Functions Using Function Pointers 386
  Callbacks: Calling Dylan from C 387
Additional Topics 390

This chapter provides an introduction to the Apple Dylan language extensions that allow you to use Macintosh Toolbox functions and C-compatible library code from your Apple Dylan programs.

Key Concepts

About C-Compatible Libraries

- **Interface definitions** allow you to create named Apple Dylan objects that correspond to entities defined in C header files.
- **Access paths** allow your Apple Dylan program to locate, and therefore call, C functions at runtime.

Importing C-Compatible Functions

- You can import and call Macintosh Toolbox functions from your Apple Dylan program.
- You can also call other C-compatible functions from Apple Dylan.

Importing Other C-Compatible Entities

- You can import and use other entities defined in C header files, including constants, structure types, pointer-to-structure types, and pointer-to-function types.
- You can **import, exclude,** and **rename** these C entities when importing them.

Callouts and Callbacks

- A **callout function** is an Apple Dylan function that allows you to call a C function given a C pointer to the function.
- A **callback function** is an Apple Dylan function for which you can create a C-compatible pointer; you can pass this pointer to C functions which can use it to call your Apple Dylan function.

About C-Compatible Libraries

To use C-compatible library code from your Apple Dylan program, you

- create Apple Dylan objects that correspond to entities defined in C header files
- ensure that your Apple Dylan program can locate the appropriate C entities in memory during runtime

You use **interface definitions** to achieve the first of these goals. Interface definitions allow you to create named Apple Dylan objects that provide an interface to constants, types, and functions defined in a C header file.

You use **access paths** to achieve the second goal. Access paths allow your Apple Dylan program to locate C functions in memory so that you can call the C functions from your Apple Dylan program.

Note
The library code that you access from your Apple Dylan program does not have to be written in C. It merely has to be sufficiently C-compatible that its interface can be described in a C header file. ◆

Interface Definitions

Interface definitions use the `define interface` built-in defining macro to allow you to create named Apple Dylan objects that correspond to entities defined in a C header file.

C entities that you can import into Apple Dylan programs include:

- functions (which include C macros with parameters)
- constants (which include `enum` constants and C macros with no parameters)
- types (which include structure types, structure pointer types, function pointer types, and so on)

Each kind of C entity is represented by a corresponding Apple Dylan entity, as reflected in this chart:

this C entity:	is represented by this kind of Apple Dylan object:
function	function
`#define` function	function
`#define` constant	named constant
`const` constant	named constant
`enum` constant	named constant
struct	class
struct member	getter (and optional setter) function
union	class
union member	getter (and optional setter) function
type	class

Interface definitions, like module and library definitions, allow you a great deal of control over the importing process. Using interface definitions, you can control

- which C entities are imported
- which are excluded
- how the names of the C entities should map to names of Dylan objects
- how types are converted between Apple Dylan code and C code
- whether function parameters are used for input, output, or both

and so on. You can also use interface definitions to create

- callout functions, which allow you to call C functions using C function pointers
- callback functions, which allow you to create C-compatible pointers to Apple Dylan functions, which you can pass to C functions to allow them to call Apple Dylan functions.

This chapter provides numerous examples of interface definitions and their **clauses** and **options,** which you use to control the importing process.

Access Paths

There are five access paths that your Apple Dylan program can use to locate C entities and execute compiled C code. Which access path you use depends on the C-compatible library you want to use.

- The **Inline Machine Code Access Path.** Apple Dylan uses this access path automatically for Macintosh Toolbox functions, or any function specified as inline machine code in a C header file. This access path is the simplest to use. (This access path is 68k-only.)
- The **Apple Shared Library Manager Access Path.** You use this access path if your C-compatible library is packaged as an ASLM shared library. This access path uses dynamic linking; therefore it is convenient to use. (This access path is 68k-only.)
- The **Code Fragment Manager Access Path.** You use this access path if your C-compatible library is packaged as a CFM shared library. This access path also uses dynamic linking and is convenient to use.
- The **External Module Access Path.** You use this access path if your C-compatible library is in the form of a compiled C file (a `.o` file, for example). This access path is easy to use, but it does require you to use MPW to create a customized version of the Application Nub. (This access path is 68k-only.)
- The **Direct Pointer Access Path.** You use this access path if your C-compatible library is in the form of a code resource, such as a HyperCard XCMD. This access path requires that you use MPW to add information to the code resource, and that you use Apple Dylan to load and unload the code resource and call the functions in the resource by calculating direct pointers to them. This is the trickiest access path.

If you have access to the C source code for your C-compatible library (for instance, if you've written the C code yourself), you can compile and package your library in any form you'd like. Therefore, you could use any of the access paths. To guide your selection, you might have to consider the machines that your application is going to run on. For example, to use the Apple Shared Library Manager or the Code Fragment Manager access paths, the runtime machines must have the correct extensions installed.

Additional Topics
A few of the examples in this chapter discuss access paths; however, you should see the "Access Paths" chapter of the *Apple Dylan Extensions and Framework Reference* for more complete information. ◆

Importing C-Compatible Functions

Importing a Macintosh Toolbox Function

Here's an example of an interface definition that imports a C function from a C header file:

```
define interface
    #include "QuickDraw.h",
        CFM-library: "InterfaceLib",
        import: { "Random" };
end interface;
```

This interface definition instructs Apple Dylan to

- parse the header file named `QuickDraw.h`
- find the C entity named `Random`, which turns out to be a function
- import that C entity into your Apple Dylan program

(The CFM-library option is specified for PowerPC; it is ignored on 68k systems.)

The `Random` function declared in `QuickDraw.h` takes no arguments and returns a value of type `short`. This function is *declared* in the header file; it also happens to be *defined* inline as a call to a Macintosh Toolbox routine.

When Apple Dylan **imports** a C entity, like the `Random` function, it creates an Apple Dylan version of the entity. For example, to import the `Random` function, Apple Dylan creates an Apple Dylan function—that is, it creates a variable named `Random` and binds it to a function object. The function object contains all the information that Apple Dylan needs to call the appropriate Macintosh Toolbox function.

You can call the Apple Dylan `Random` function from your Apple Dylan program. When you do, Apple Dylan executes the function object, which calls the Macintosh Toolbox function, which returns a value of the C type `short`. Apple Dylan converts this value to an object of the Dylan class `<integer>`, and returns this object as the function result. For example:

```
Dylan> Random()
15138

Dylan> Random()
-3517
```

To recap, the **interface definition** imports the `Random` function from a C header file into Apple Dylan. The resulting Apple Dylan function is also named `Random`; it contains all of the information necessary to call the `Random` function described by the C header file.

When you call the Apple Dylan function `Random`, Apple Dylan executes the function object, which uses the inline machine code **access path** (on the 68k) to call the Macintosh Toolbox random-number-generating function. (This example uses the CFM access path on the PowerPC.) Apple Dylan also performs **type conversion** on the value returned by this function; the returned value is converted to a value of the Dylan `<integer>` class.

Apple Dylan provides **function clauses** to allow you to specify more information about how specific functions should be imported. For example, you can use a function clause to specify whether arguments are used for input, output, or both.

For example:

```
define interface
    #include "OSUtils.h",
        CFM-library: "InterfaceLib",
        import: { "Delay" };

    function "Delay",
        output-argument: finalTicks;
end interface;
```

This interface definition imports the `Delay` function from the `OSUtils.h` header file. The interface definition also uses a function clause to specify that the

parameter named `finalTicks` in the header file represents an output argument. When you specify this option for a function, you do not have to provide any value for this parameter when you call the function. For example, the C definition of Delay specifies two parameters:

```
void Delay(long numTicks, long *finalTicks);
```

However, when you call this function from Apple Dylan, you only need specify the value for the `numTicks` parameter:

```
Dylan> Delay(60);
10543758
```

The Apple Dylan function `Delay` returns as its function result the value the Macintosh Toolbox function returns using its second parameter (`finalTicks`).

In this example, only one value is returned by Apple Dylan because the Toolbox function has only one output parameter (and no function result). The Apple Dylan function would return multiple values if the Toolbox function also returned a function result, or if you specified more than one parameter to be an output parameter.

Importing a Shared Library Function

The example in the previous section used the inline machine code access path (on 68k systems) to call a function from the Macintosh Toolbox. The other four access paths allow you to use functions from other C-compatible libraries.

For example, imagine that you implement your own C function for generating random numbers and you package the function in a Code Fragment Manager (CFM) shared library, as discussed in the *Apple Dylan Extensions and Framework Reference*.

For the purposes of this example, assume that the name of your CFM shared library is `sample_CFM_library`, and that the interface for your function is declared in a header file named `SampleHeader.h`, as shown here:

```
/*** SampleHeader.h ***/

short Random (void);
```

This header file declares your C function. Like the Macintosh Toolbox function of the same name, your `Random` function takes no arguments and returns a value of type `short`.

You can import this `Random` function into your Apple Dylan program using the following interface definition:

```
define interface
    #include "SampleHeader.h",
        CFM-library: "sample_CFM_library";
end interface;
```

This interface definition imports everything from the `SampleHeader.h` header file—which in this case is just the `Random` function. The interface definition also specifies where Apple Dylan can find the compiled implementation of your `Random` function—namely, in the CFM shared library named `sample_CFM_library`.

If you were using an Apple Shared Library Manager (ASLM) shared library, your interface definition would look slightly different:

```
define interface
    #include "SampleHeader.exp";
end interface;
```

ASLM shared libraries use `.exp` files as well as `.h` files to define the interface to the library. When you specify an `.exp` file in an interface definition, Apple Dylan automatically knows you are using an ASLM shared library and examines the `.exp` file to determine the name of the shared library.

Since each type of access path requires a slightly different interface definition, the rest of the examples in this chapter are written as if you are using the inline machine code access path, which is the simplest. When using other access paths, you should make the appropriate adjustments to your interface definitions.

Additional Topics
The static linking and code resource access paths require you to provide more complicated Apple Dylan source code. See the "Access Paths" chapter of the *Apple Dylan Extensions and Framework Reference* for more information. ◆

Importing Other C Entities

As the last few sections have described, interface definitions allow you to import entities defined in C header files into your Apple Dylan program. So far, the examples in this chapter have been very simple—they have imported a single function. As these examples have shown, importing a C function into Apple Dylan produces an Apple Dylan function that performs the cross-language function call and the type conversions for you.

Interface definitions allow you to import more than a single function, however. For example, imagine this header file:

```
/*** ThreeFunctions.h ***/

long calculate (long a, long b);

long combine (long a, long b);

long conflate (long a, long b);
```

This header file declares three C functions. Each of these functions expects two arguments of type `long` and each of the functions returns a value of type `long`. You can import the entire header file using this interface definition:

```
define interface
    #include "ThreeFunctions.h";
end interface;
```

This interface definition imports all three of the functions declared in the header file `ThreeFunctions.h`. (Remember, your actual interface definition will change slightly depending on which access path you use.)

This interface definition creates three Apple Dylan functions, named `calculate`, `combine`, and `conflate`. Each of these Apple Dylan functions expects two arguments of class `<integer>`. When you call any of these functions, Apple Dylan:

1. converts the arguments to values of type `long`
2. passes the resulting values to the corresponding C function
3. converts the result of the C function from the C type `long` to the Dylan class `<integer>`

Apple Dylan does the type conversion between the C type `long` and the Dylan class `<integer>` automatically, because `long` is a built-in C type.

Apple Dylan will also convert between other C types and Dylan classes:

this C type:	is converted to this Apple Dylan type:
`short`	`<integer>`
`int`	`<integer>`
`long`	`<integer>`
`Boolean`	`<Boolean>`
`Str255`	`<Pascal-string>`

(The types `Boolean` and `Str255` are defined in the `Types.h` header file.)

Additionally, Apple Dylan converts between user-defined C types, such as structure types, and Apple Dylan types. For example, consider the function `switch` as declared in this header file:

```
/*** StructureAndFunction.h ***/

typedef struct {
    long x;
    long y;
} position;
```

```
position* switch (position* original);
```

The `switch` function takes a single argument—a pointer to a structure of type `position`. The `switch` function also returns a pointer to a structure of type `position`. When you import this header file, Apple Dylan imports both the `position` structure type and the `switch` function.

You use this interface definition to import the structure type and the function:

```
define interface
    #include "StructureAndFunction.h";
end interface;
```

Apple Dylan imports the `position` structure type as an Apple Dylan class named `<position>`, and the `switch` function as an Apple Dylan function named `switch`. After importing this header file you can use the `<position>` class to create a new object:

```
Dylan> define variable my-position = make (<position>);
defined my-position
```

The `<position>` class corresponds to the `position` C structure type; similarly, the slots of the `<position>` class correspond to the members of the `position` structure type. However, by default, Apple Dylan prepends the structure name onto the slot name. For example:

```
Dylan> my-position.position$x := 10;
10

Dylan> my-position.position$y := 20;
20
```

The <position> class has two slots: one is named `position$x`, the other `position$y`.

When you call the `switch` function from Apple Dylan, you supply an argument of the class `<position>`. Apple Dylan

- converts this object to a C pointer to a structure of type `position`
- sends this pointer to the C function `switch`

- converts the value returned by the C function from a C pointer to an Apple Dylan object that points to the same structure in memory.

As an example, consider this variable definition, which calls the `switch` function to create a new object of class `<position>`:

```
define variable new-position = switch (my-position);
```

Assuming your C implementation of the `switch` function switches the values of the structure members `x` and `y`, examining the slots of the object named `new-position` produces these values:

```
Dylan> my-position.position$x;
20

Dylan> my-position.position$y;
10
```

Now you've seen how to import a C structure type and a C function from a C header file and how to use them from within an Apple Dylan program. The rest of the sections in this chapter highlight other C entities you can import into an Apple Dylan program, and introduce some of the options you have that control how these entities are imported into Apple Dylan.

Importing C Constants

You can use interface definitions to import three different kinds of C constants:

- constants created with `const`
- individual enumerated constants created with `enum`
- macros (with no arguments) created with `#define`

For example, consider the header file shown here:

```
/*** Constants.h ***/

const kConstant = 5;
```

```
enum { kFirstEnumerator = 1,
       kSecondEnumerator = 10 };

#define kMacro kConstant + kConstant + kConstant
```

You can import this C header file using this interface definition:

```
define interface
    #include "Constants.h";
end interface;
```

This interface definition creates four Apple Dylan constants:

- a constant named `$kConstant`, which has the value 5
- a constant named `$kFirstEnumerator`, which has the value 1
- a constant named `$kSecondEnumerator`, which has the value 10
- a constant named `$kMacro`, which has the value 15

Notice that Apple Dylan automatically renames the constants, so that they all begin with a *$* character, which is the naming convention for constants in Apple Dylan.

You can suppress this automatic name mapping using the **name-mapper option**. For example:

```
define interface
    #include "Constants.h",
        name-mapper: identity-name-mapping;
end interface;
```

The name-mapper option allows you to specify one of several built-in functions that map the names of C entities to names more appropriate for Apple Dylan objects. The `identity-name-mapping` function performs no mapping on the names. In this example, specifying the `identity-name-mapping` function as the name mapper results in the Apple Dylan constants having the names `kConstant`, `kFirstEnumerator`, `kSecondEnumerator`, and `kMacro`—unchanged from their C names.

There are other built-in functions you can specify as the name mapper. The `minimal-name-mapping-with-structure-prefix` function is the default. This function

- adds a *$* character to the beginning of the constant name.
- adds the structure name and a *$* character to the beginnings of slot names for imported structure members
- adds angle brackets around C types that become classes

The `minimal-name-mapper` function behaves similarly, except it does not add the structure names and a `$` character to the front of slot names.

You can also use the **rename option** to rename C entities explicitly. Sometimes, you must use this option, in order to avoid name conflicts when importing C entities.

For example, consider the C header file:

```
/*** Conflicts.h ***/

const kmyconstant = 1;

const kMyConstant = 2;

const kMYCONSTANT = 3;
```

Since C is case sensitive, these three constants have distinct names in C. However, Apple Dylan is not case sensitive; therefore, these three constants present a naming conflict when you import them into Apple Dylan.

To resolve this naming conflict, you can use the rename option to provide explicit Apple Dylan names for these constants:

```
define interface
    #include "Conflicts.h",
        rename: { "kmyconstant" => $my-first-constant,
                  "kMyConstant" => $my-second-constant,
                  "kMYCONSTANT" => $my-final-constant };
end interface;
```

Another feature of interface definitions is selective importing: you can use interface definitions to import selected C entities, or to exclude selected entities. For example, the interface definition

```
define interface
    #include "Conflicts.h",
        import: { "kmyconstant" };
end interface;
```

imports only the C constant named `kmyconstant` (which, by the default name-mapping conventions, becomes the Apple Dylan constant named `$kmyconstant`).

The following interface definition imports the same constant, but this interface definition does so by excluding the other two constants:

```
define interface
    #include "Conflicts.h",
        exclude: { "kMyConstant", "kMYCONSTANT" };
end interface;
```

Importing C Structures

Earlier in this chapter, you saw how to import a C structure type from a C header file into an Apple Dylan program. Apple Dylan creates classes to represent C structure types; the slots of the Apple Dylan class correspond to members of the C structure. For example, the C structure type

```
typedef struct {
    long x;
    long y;
} position;
```

becomes the Apple Dylan class `<position>`, which has a slot `x` and a slot `y`.

The fact that Apple Dylan implements slots as setter and getter functions leads to some interesting repercussions. When you import the `position` structure type, Apple Dylan creates not only creates a `<position>` class—it also creates four functions:

- the function `position$x`, which takes one argument (an object of class `<position>`) and returns the value of that object's `x` slot
- the function `position$y`, which takes one argument (an object of class `<position>`) and returns the value of that object's `y` slot
- the function `position$x-setter`, which takes two arguments (an integer value and an object of class `<position>`) and sets the value of the object's `x` slot to the specified integer value
- the function `position$y-setter`, which takes two arguments (an integer value and an object of class `<position>`) and sets the value of the object's `y` slot to the specified integer value

For example, if you import the `position` structure type and you create a `<position>` object using this call to the make function:

```
define variable my-position = make (<position>);
```

you can then set the value of the object's slots using assignment operations:

```
my-position.position$x := 10;
my-position.position$y := 20;
```

These assignment operations are Apple Dylan shorthand for the following function calls:

```
position$x-setter (10, my-position);
position$y-setter (20, my-position);
```

Similarly, you can examine the value of the object's slots by evaluating these expressions:

```
my-position.position$x;
my-position.position$y;
```

These expressions are Apple Dylan shorthand for the following function calls:

```
position$x (my-position);       // returns value of x slot
position$y (my-position);       // returns value of y slot
```

To recap: when you import a C structure type, you create getter and setter functions as well as an Apple Dylan class. These functions can lead to name conflicts.

The `minimal-name-mapping-with-structure-prefix` function, which is the default name mapper, helps to avoid naming conflicts by prepending the structure name and a *$* character: it is unlikely that you would already have module variables named `position$x` and `position$y`.

The `C-to-Dylan` name mapper function also helps to resolve this kind of conflict. This name mapper adds the prefix `get-` to the slot getter name. If you specify this name mapper when importing the `position` structure type, you would use this code to access the slots of the `my-position` object:

```
my-position.get-x;
my-position.get-y;
```

which is equivalent to:

```
get-x (my-position);
get-y (my-position);
```

Importing C Structure Pointer Types

Importing a pointer to a structure type results in an Apple Dylan class, just as importing the structure type itself does. In fact, there is no difference between importing a structure type and importing a pointer to the structure. Both are represented in Apple Dylan as objects that point to a structure in memory.

When you import a C structure pointer type explicitly, you can use the resulting class just as you would use the class created to represent a structure type. For example, consider the header file shown here:

```
/*** StructurePointer.h ***/

typedef struct {
    long x;
    long y;
} position;

typedef position *posptr;
```

This example defines two C entities:

- a `position` structure type
- a pointer to that structure type named `posptr`

You can import the `posptr` type (and exclude the `position` type) using this interface definition:

```
define interface
    #include "StructurePointer.h",
      import: { "posptr" };
end interface;
```

This interface definition creates a class `<posptr>`, which works identically to the `<position>` class created in the previous section.

IMPORTANT
Although you can import pointers to structures, the standard Apple Dylan style is to import the structure, rather than a pointer to the structure. ▲

Importing C Function Pointer Types

You can also use interface definitions to import pointer-to-function types declared using `typedef`. For example, consider this header file:

```
/*** FunctionPointer.h ***/

typedef long (*combiner) (long, long);

combiner getACombiner (void);

long applyCombiner (combiner, long a, long b);
```

This header file defines three entities:

- A pointer-to-function type, named `combiner`. This pointer type points to functions that require two `long` arguments and provide one `long` result.

- A function named `getACombiner`. This function returns a pointer to a function as its function result.
- A function named `applyCombiner`. This function takes a pointer to a function and two long values as its arguments. It returns a single value of type `long`.

You can import this header file using this interface definition:

```
define interface
    #include "FunctionPointer.h",
        name-mapper: C-to-Dylan;
 end interface;
```

This interface definition creates three Apple Dylan objects:

- a class named `<combiner>`, which allows you to store pointers to C functions in your Apple Dylan environment
- a function named `get-a-combiner`, which calls the C function `getACombiner`, which returns a pointer to a function
- a function named `apply-combiner`, which takes a function pointer as its first argument

(Notice how the C-to-Dylan name mapper adds angle brackets to the class name, and inserts hyphens into the names in place of inner caps.)

Here is how you might use these Apple Dylan objects:

```
define variable my-combiner = get-a-combiner();

define variable result = apply-combiner(my-combiner, 1, 2);
```

The first variable definition in this example calls the `get-a-combiner` function, which in turn calls the C function `getACombiner`. The C function returns a function pointer, which Apple Dylan converts to an object of the `<combiner>` class. This object is bound to the name `my-combiner`.

The second variable definition in this example calls the `apply-combiner` function, passing it the arguments `my-combiner`, 1, and 2. Apple Dylan converts the `my-combiner` object to a C function pointer, and calls the C function `applyCombiner`, passing that function the function pointer and the values 1 and 2. The `applyCombiner` function returns a result of type `long`, which Apple Dylan convert to an object of class `<integer>` and binds to the name `result`.

In this example, the <combiner> class allows you to represent function pointers as Apple Dylan objects; therefore you can pass them as arguments to C functions and receive them as return values from C functions.

Callouts and Callbacks

Callouts: Calling C Functions Using Function Pointers

The previous section showed how you can pass function pointers between C functions. However, sometimes you have a function pointer and you want to call the function directly, rather than having to pass the function pointer to a C function to do the calling for you.

Apple Dylan provides the **callout** mechanism for calling a C function given a pointer to the function. As an example, consider this header file:

```
/*** Callouts.h ***/

typedef long (*combiner) (long, long);

combiner getACombiner (void);
```

This header file has two of the elements from the header file in the previous section: a function pointer type and a function that returns a pointer to a function.

Now, consider this interface definition:

```
define interface
    #include "Callouts.h",
        name-mapper: C-to-Dylan;

    callout "combiner" => call-combiner;
end interface;
```

This interface definition creates three Apple Dylan objects:

- A class named `<combiner>`, which is used to represent function pointers.
- A function named `get-a-combiner`, which returns an object of type `<combiner>`.
- A **callout function** named `call-combiner`. This function takes three arguments: an object of class `<combiner>`, and two objects of class `<integer>`. It applies the function represented by the `<combiner>` object to the two integer arguments, and returns a value of type `<integer>`.

Here is how you might use this callout function:

```
define variable my-combiner = get-a-combiner;

define variable result = call-combiner (my-combiner, 1, 2);
```

The first variable definition calls the `get-a-combiner` function, which calls the C function `getACombiner`, which returns a function pointer. Apple Dylan converts that function pointer to an object of type `<combiner>` and names the object `my-combiner`.

The second variable definition calls the `call-combiner` callout function. This function converts its first argument into a C function pointer. Then, it calls the C function pointed to by that pointer, passing it the arguments 1 and 2. The C function returns a value of type `long`, which Apple Dylan converts to an object of class `<integer>`. This object is returned and bound to the variable name `result`.

The `call-combiner` callout function allows you to call a C function when you have a pointer to the function. Notice that the arguments expected by a callout function, and the value returned by a callout function, depend on the arguments and return value specified by the function pointer type.

Callbacks: Calling Dylan from C

Apple Dylan also allows you to create **callback functions.** You can think of a callback function as a kind of pointer to an Apple Dylan function. When you call a C function from Apple Dylan, you can pass a callback function as a argument to the C function. The C function can then use the callback function

as it would use any function pointer—effectively allowing the C function to call Apple Dylan code.

Here is a sample header file:

```
/*** Callbacks.h ***/

typedef long (*combiner) (long, long);

long applyCombiner (combiner f, long a, long b);
```

This header file defines two C entities:

- a function pointer type, named `combiner`, that points to functions with two `long` arguments and an `long` return value
- a function named `applyCombiner` that takes three arguments—a function pointer and two values of type `long`—and returns a value of type `long`.

Imagine that you've implemented the `applyCombiner` function using the following C code:

```
long applyCombiner (combiner f, long a, long b) {

    return ((*f) (a, b));
}
```

This simple function takes a pointer to a function and two arguments of type `long`, applies the function to the `long` arguments, and returns the result.

You can import the C header shown above using this interface definition:

```
define interface
    #include "Callbacks.h",
         name-mapper: C-to-Dylan;

    callback "combiner" => make-Dylan-combiner;
end interface;
```

This interface definition creates three Apple Dylan entities:

- A class named `<combiner>`, which is used to represent function pointers.
- A function named `apply-combiner`, which applies a function to two integers and returns a single integer.
- A **callback macro** named `make-Dylan-combiner`. This macro allows you to create an object of class `<combiner>`—effectively a C pointer to an Apple Dylan function.

Here is how you might use the callback macro to create a **callback function**—in this case, an object of class `<combiner>`:

```
define variable my-callback =
    make-Dylan-combiner (a, b)
        (a * a) + (b * b);
    end;
```

Note that the above example is shown in standard Dylan syntax. The Apple Dylan implementation uses the following syntax:

```
define variable my-callback =
    make-Dylan-combiner (a(b),
    begin
        (a * a) + (b * b);
    end);
```

This variable definition calls the `make-Dylan-combiner` callback macro. This macro creates a callback function—a special kind of Apple Dylan function. In this example, the callback function created is an object of class `<combiner>`. The callback function, which is implemented in Apple Dylan code, takes two integer arguments and calculates the sum of their squares.

Now, here is how you use this callback function:

```
define variable result = apply-combiner(my-callback, 1, 2);
```

Executing this variable definition results in these steps:

1. The `my-callback` object is converted to a C function pointer of type `combiner`.
2. The integer objects representing 1 and 2 are converted to values of type `long`.
3. The function pointer and the `long` values are passed to the C function `applyCombiner`.

4. The C function `applyCombiner` applies the function pointed to by its first argument to the values provided in its second and third arguments.
5. Since the function pointed to by the first argument is the callback function, control is passed back to Apple Dylan, which converts the `long` values to objects of type `<integer>`, and passes them to the `my-callback` callback function.
6. The `my-callback` callback function evaluates the `(1 * 1) + (2 * 2)` expression and returns the resulting value of 5.
7. Apple Dylan converts the integer object 5 into a value of type `long` and passes it back to the C function `applyCombiner`.
8. The C function `applyCombiner` finishes executing and returns the value 5 to Apple Dylan, which converts it to an integer object and binds that object to the name `result`.

Additional Topics

This chapter included many simple examples of importing C entities from C header files and using C entities from C-compatible libraries. See the *Apple Dylan Extensions and Framework Reference* for more information and additional examples.

CHAPTER 22

Introducing the Framework

Contents

The Apple Dylan Framework 393
Basic Concepts 395
  Windows and Views 395
    The View Hierarchy 395
    View Determiners 397
    Other User Interface Elements 398
  Documents and Data 399
  Event Handling 399
    The Target Chain 399
    Mouse Event Handling 402
    Changing the Target 402
    Behaviors 404

This chapter introduces the features provided by the Apple Dylan Framework. It then describes basic concepts and the conceptual model for the Framework and identifies how the Framework classes fit into this model.

The Apple Dylan Framework

The Apple Dylan Framework (called the Framework in this manual) is a body of Dylan code that implements the infrastructure, or **framework**, for a Macintosh application. You can think of the Framework as a generic, or content-free Macintosh application that implements the common set of features found in an application. These features include the Macintosh Graphical User Interface (GUI), document handling, and so on.

The Framework implements these features as an object-oriented class library. If you are familiar with class libraries for developing applications, such as the Microsoft Foundation Class Library (MFC) or MacApp, you should find similarities between classes in the Apple Dylan Framework and those found in other class libraries.

If you are familiar with MacApp, you will notice a lot of similarity, because both frameworks are used to implement Macintosh applications. However, the Apple Dylan Framework is more than simply MacApp written in the Dylan language. Although the Framework borrows ideas and techniques that have been proven over the years in MacApp, it has been designed from top-to-bottom with extensibility in mind. The Apple Dylan Framework also takes advantage of features built into the Dylan language, which should make programming with the Framework even easier than programming with MacApp.

The Framework includes support for the following features found in most Macintosh applications:

- an event loop and event handling, including support for Apple events
- windows, menus, and other user interface elements, such as dialog boxes and controls
- text editing
- clipboard
- documents and data manipulation

- resource management
- error reporting and exception handling

The Framework also provides implementations of techniques and mechanisms that are generally useful to developers. They include:

- views, which provide a way to subdivide a window for drawing and respond to events within windows
- grid views, which subdivide a window into cells like those found in a spreadsheet or a tool palette
- text views, which manage typing in a view
- adorners, which are reusable specifications for drawing in a view
- scrolling and tracking
- behaviors, which are reusable specifications for how to handle various kinds of events
- extensible graphics, including nonbuffered graphics for tight memory situations and graphics based on Macintosh graphics worlds (GWorlds).
- object model support for scripting, including recordability
- drag and drop as provided by the Drag Manager
- scheduling and execution of tasks to be performed during the event loop's idle time.
- resource management
- streams, which allow data to be operated on sequentially; for example, to move code and resources to and from memory.
- dependency tracking and notification
- debugging support

Basic Concepts

This section presents a brief overview of the major concepts you should become familiar with before you start to program with the Dylan Framework. The section does not attempt to introduce all concepts nor even all important ones, just the ones you need to get started.

Conceptually, you must have an understanding of windows, views, and event handling to get started. Because many applications are document-based, an understanding of documents and the data associated with them is also important. Thus, the following sections introduce basic concepts related to:

- windows and views
- documents and data
- event handling

Windows and Views

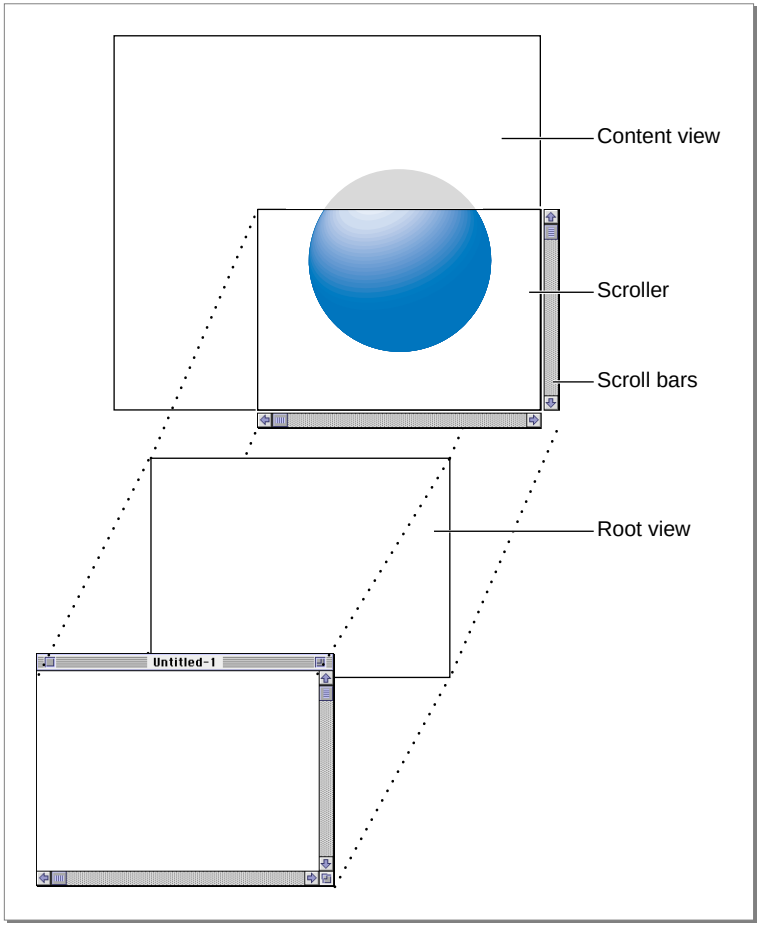
This section introduces the concepts necessary to understand how windows and views are implemented. For information about creating views and resources with the Dylan Interface Builder, see the *Apple Dylan User-Interface Builder* manual.

A window as supported by the Framework is exactly the same as a window in the Window Manager. You can create the same kinds of windows with the Framework as you can with the Window Manager. A view, however, does not have a direct counterpart with a Macintosh manager. In the Framework, views represent the different parts of a window, such as the content part of the window, scroll bars, Control Manager controls, and so on.

The View Hierarchy

The Framework provides a view that represents the entire area of the window, called the **root view**. The root view is at the top of the window's view hierarchy. The **view hierarchy** establishes the relationship between views. As an example, consider the scrolling window in Figure 22-1, which contains the minimum number of views to support horizontal and vertical scrolling.

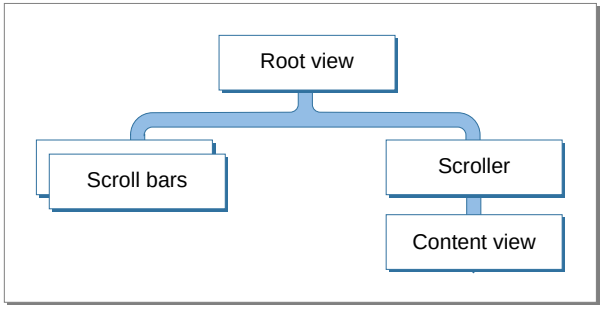
Figure 22-1 Views in a window



Five views implement a scrolling window—the root view, a scroller view that knows how to scroll, two scroll bar controls that are also views, and the content view that represents the data you want to display in the window. (The Framework also maintains a grow-icon view as part of the window; you need not be concerned with this view.)

Figure 22-2 shows the view hierarchy.

Figure 22-2 A window's view hierarchy



Note
A window is not part of a view hierarchy. ◆

The scroller and the scroll bars are all **subviews** of the root view, thus the root view is the **superview** of the others. The content view is a subview of the root view and of the scroller. The content view's superview is the scroller. The location and size of each view is defines its **extent**.

Although view hierarchies are not restricted in depth, many windows have a relatively flat view hierarchy associated with them. In general, a view hierarchy can be as deep as it needs to be but should be kept as flat as possible for better performance.

There are two major reasons why the Framework organizes views into a hierarchy. One reason is to allow views to handle events, which is described in the section "Event Handling" on page 399. The other reason is so you can specify rules for changing the extent of a subview if the extent of the superview changes, as described in the next section.

View Determiners

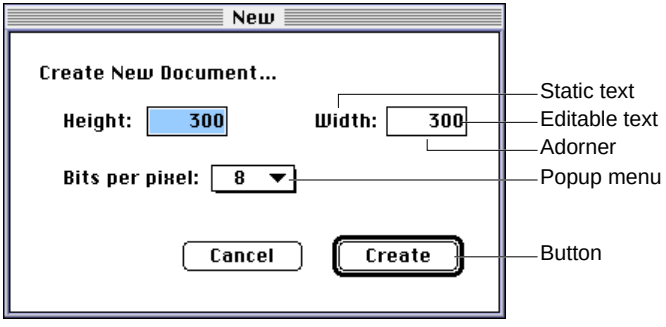
The rules that determine how a subview changes in response to a change in the extent of its superview are called **view determiners**. For example, if the user resizes the window, the Framework automatically resizes the root view to match the window's size. The subviews of the root view (and subviews of subviews, and so on) are notified of the change to their superview so that the visible contents of the scroller can be redrawn and scroll bars can be resized as well.

View determiners for each view are different because each view must change in a different way due to the change in the superview. In the previous example, the horizontal scroll bar sticks to the bottom of the scroller view, the vertical scroll bar sticks to the right side of the scroller view. The scroller changes proportionally to the size of the root view. The content view does not need to change when the window is resized, thus it does not have a view determiner.

Other User Interface Elements

Figure 22-3 shows a moveable modal dialog box that contains additional kinds of views—static text, editable text fields, and push button controls. The editable text fields are constrained to be digits, thus they are called **number text** fields.

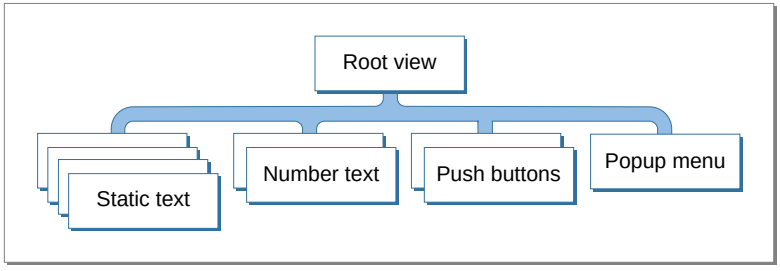
Figure 22-3 Views and adorners in a dialog box window



The window also contains **adorners**, which are modifications to views that are drawn along with the contents of the view. Adorners are reusable because the same adorner can be attached to several views. Figure 22-3 shows two kinds of adorners, the frames around the number text fields and the border around the default push button. In Figure 22-3, a single frame adorner is reused twice, once for each number text field.

Figure 22-4 shows the view hierarchy for the dialog box.

Figure 22-4 The view hierarchy for a dialog box



The view hierarchy in Figure 22-4 is similar to the one in Figure 22-2. Both hierarchies descend from the root view, which the Framework provides when you create a window. In Figure 22-4, each of the non-root views is a subview of the root view, as in Figure 22-2. Note that adorners are not in the view hierarchy.

Documents and Data

A **document** is an object that allows you to associate data with a window and a file. Typically, a document also refers to the data in memory. This data may be in a collection class object provided by Dylan, in a region or graphics buffer provided by the Framework, or in some other data structure that you provide.

Event Handling

This section introduces the concepts necessary to understand how event handling is implemented. Windows, views, and documents are event handlers. An **event handler** is an object that is capable of responding to events, such as keystrokes, mouse clicks, or menu item selections.

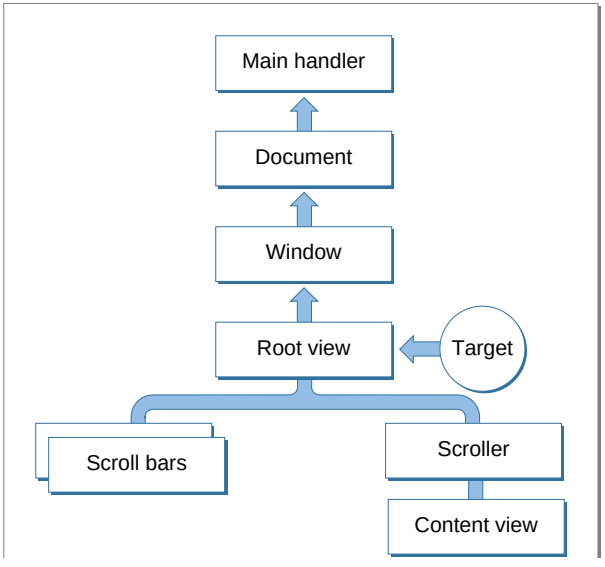
The Target Chain

Event handlers are organized into a chain that specifies the order in which they can respond to events. The chain is called a **target chain**, because an event is first passed to the target event handler, which may handle the event by taking

some action. If the event is not handled by the target event handler, the next handler in the chain is allowed to respond, and so on.

Figure 22-5 shows the target chain for the window in Figure 22-1 on page 396, which includes a view (the root view) a window, and a document.

Figure 22-5 The target chain for a document and its window



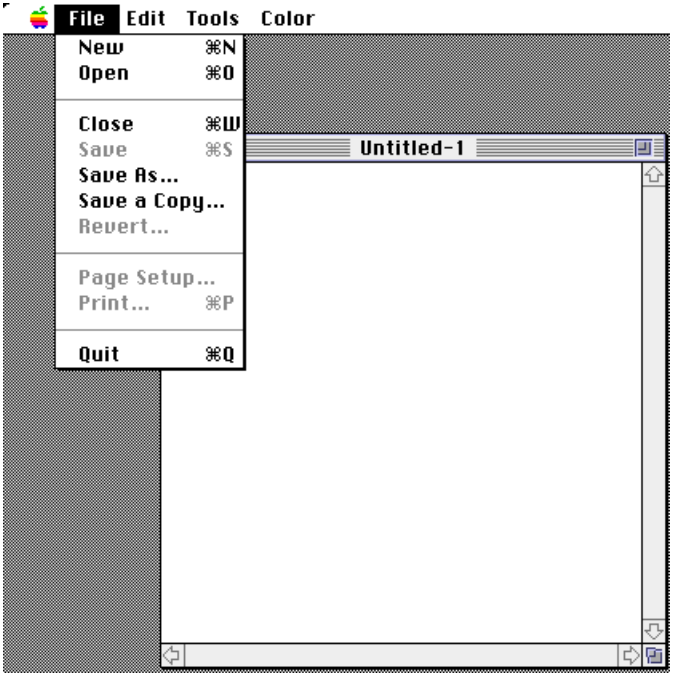
At the top of the chain is the **main handler**, which the Framework provides. It is the event handler of last resort, meaning that if no other event handler handles an event, the main handler can respond. The target chain includes all event handlers between the target handler and the main handler, inclusive.

Before any windows are opened or documents created, the main handler is the target handler and the target chain consists of only the main handler. When you open a document or window, you specify where the document or window is linked into the target chain by specifying its **next handler** in the chain. Typically, you specify that the next handler for a document as the main handler and that the next handler for the window that displays the document's data as the document itself.

By default, when a window is added to the target chain, its root view becomes the target event handler. Thus in Figure 22-5, the root view is the target handler, meaning that an event can be handled by the root view, the window, the document, and the main handler. You often want to specify a target that is different from the root view—you can do so by specifying the target when you create the window or by specifying the target dynamically.

Events, excepting mouse events, are sent up the target chain. Consider the case of menu events related to the menu items in Figure 22-6.

Figure 22-6 Menu item event handling



The events corresponding to the New and Open menu items are often handled by the main handler because they are not related to a specific document—the document has not net been opened or created. The Close, Save, and Save As items typically are handled by the document. The Print item could be handled

by either the window or the document, depending on the interpretation of what is to be printed; typically, however, printing relates to the contents of the document and not just to the visible content in the window. The Quit item would be handled by the main handler because the event relates to the application overall and not to a document or window.

Note
In many cases, menu handling by the main handler is implemented as a behavior. For more about behaviors, see "Behaviors" on page 404. ◆

Mouse Event Handling

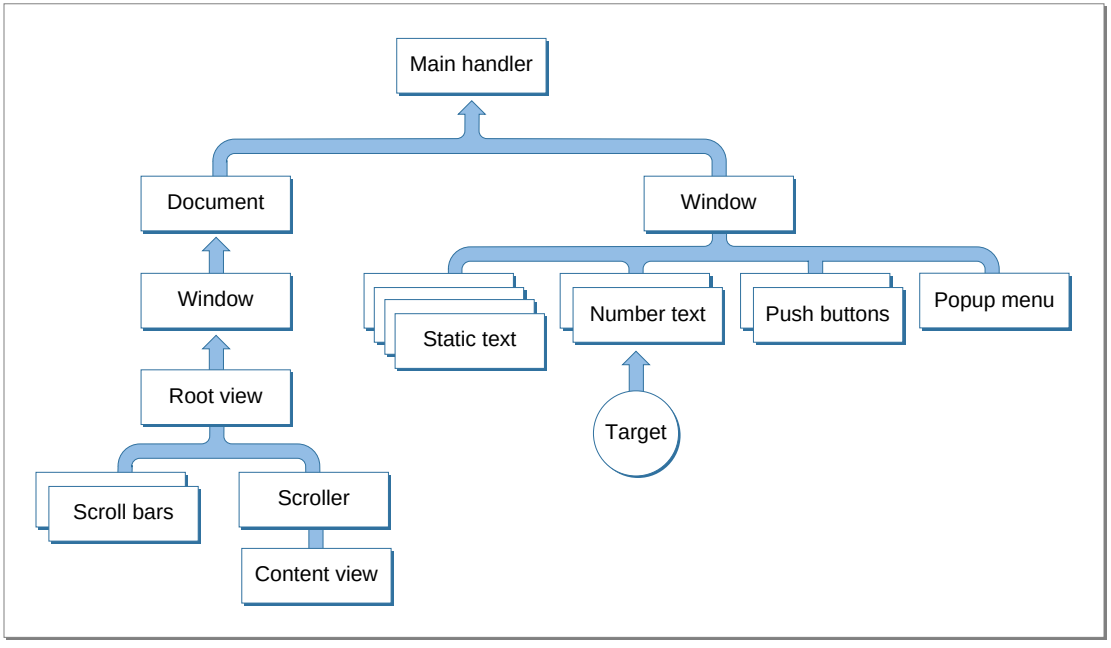
Mouse events, such as a mouse-down or mouse-up event, are directed to a window. In the case where the window is active, the deepest subview in the view hierarchy under the location where the mouse event occurred is given the chance to respond. In Figure 22-5 on page 400, either a scroll bar, the scroller, or the content view will be given the chance to respond. If the mouse click is within the scroller, the content view will be given the chance to respond, otherwise one of the scroll bars will have the chance.

If the mouse event is in the scroller but the content view does not handle it, the scroller would have the chance next, followed by the root view, the window, and finally the main handler. Note that mouse events are sent from the window to the main handler directly—event handlers outside of windows do not need to handle mouse events; mouse events on the desktop are handled by the main handler.

Changing the Target

Consider a slightly more complicated scenario, in which the dialog window in Figure 22-3 on page 398 appears in response to the menu event for the New item while a document is active. Figure 22-7 shows the target chain after the dialog window is activated.

Figure 22-7 Target chain for an active dialog



A window event occurs that deactivates the document's window and activates the dialog window. The Framework handles this event by dispatching a resign-target event up the target chain, notifying each event handler that the target has changed and that the old target is no longer the root view for the document's window. This allows event handlers to get ready to relinquish the target; for example, the root view can instruct its subviews to change their highlighting.

As soon as the window is activated, the Framework dispatches a become-target event up the target chain, which allows event handlers to get ready; for example, by highlighting views such as the target view, and so on.

In the case of the dialog box shown in Figure 22-3 on page 398, the target handler was set to the number text field associated with the Height when the window was created. Thus key-down events are dispatched to this event handler and are handled by it unless the user changes the target handler by

clicking on a different view, by using the tab key to move to a different field, or by confirming or canceling the dialog box.

Behaviors

Behaviors specify additional actions for an event handler. They serve two major purposes:

- They allow you to customize the normal actions associated with an event handler. You can define the actions in the behavior and attach it to the event handler without creating a subclass for the event handler.
- They are reusable so that once the actions have been defined once, you need not duplicate them. You can simply add the behavior to the event handler and remove them when no longer necessary.

The Framework defines methods for behaviors that serve the same purpose as those for event handlers. For example, the Framework provides `do-event` methods in which you specify actions for event handlers to take in response to an event. The Framework also defines a `behavior-event` methods for behaviors to specify actions.

Event handlers can have one or more behaviors attached to them. The behaviors for an event handler can respond to events immediately before the event handler itself.

CHAPTER 23

Framework Tutorial

Contents

The Skeleton Application 407
- Defining a Library and Module 408
- Defining Classes 409
- Initializing the Application 410
- Implementing a Menu Item 411
- Creating Windows, Views, and Adorners 412
- Drawing in a View 414
- Handling an Event 415
- Implementing Mouse Tracking 416
- Implementing Open Scripting Architecture support 419

This chapter describes a skeleton application, which is an application that shows the steps you typically take to get started programming with the Framework.

Note
The skeleton application described in this chapter works with the Alpha 2 version of the Apple Dylan Framework. ◆

The Skeleton Application

This section describes a very basic application, callled the skeleton application, that introduces you to some of the tasks you will probably perform when you create an application with the Framework. Fortunately, these tasks are not difficult compared to building an application without the Framework because the Framework does much of the work to complete each task for you.

The skeleton application handles a document whose window displays the document's data. The document's data consists of rectangles which the user creates by pressing the mouse and moving it to a new location. The skeleton application also implements a static text field that can be set externally with an Apple script.

In building the skeleton application, the following tasks are performed:

- define a module
- define classes for a behavior, document and view
- handle initialization for the application
- implement a menu item
- create windows and views
- draw in a view
- handle an event
- implement mouse tracking
- provide Open Scripting Architecture support

The following sections describe each of these tasks.

Defining a Library and Module

When you first create your application, you must define a library and module for your application in the Dylan User module. Your library specifies all the objects that conceivably could be used in the application. Your module is the name space that contains your source code. Listing 23-1 shows the library and module definitions.

Listing 23-1 Library and module definitions

```
define library skeleton-library
    use Dylan;
    use mac-toolbox;
    use dylan-framework;

    export skeleton-module;
end library;

define module skeleton-module
    use Apple-Dylan;
    use mac-toolbox;
    use dylan-framework;
end module;
```

The library `skeleton-library` uses the Dylan language, the Macintosh toolbox, and the Framework. It exports the module `skeleton-module`. The module also uses the language, toolbox, and Framework. It exports the `start` method from the Framework and the `run-skeleton` method from the application.

Listing 23-2 shows the constants used by the library.

Listing 23-2 Constants

```
define-framework-library("skeleton-library");

define constant $skeleton-string-id = 1000;

define constant $skeleton-window-title-id = 1001;
```

```
define constant $file-menu-id = 1000;

define constant $edit-menu-id = 1001;
```

Note
These constants are not specified in the library or module definition. They are stored along with the application code.
◆

Defining Classes

The skeleton application requires three classes before much else can be defined:

- a behavior class that customizes the application
- a document class that represents the data in a window
- a view class that is responsible for rendering the data

Listing 23-3 shows these class definitions and their `initialize` functions. In this application, the view can access the document's data from a slot in the view definition. (The `initialize` function is not strictly necessary in this example. An easier way would be to specify the document using `init-keyword:` with the document slot of the `<my-view>` class.)

Listing 23-3 Class definitions for a behavior, document, and view

```
define class <app-behavior> (<behavior>)
end class;

define class <my-document> (<document>)
    slot data :: <list>,
        init-value: #();
end class;

define class <my-view> (<view>)
    slot the-document :: <my-document>;
end class;
```

```
define method initialize (view :: <my-view>,
                          #key document: doc) => ()
    next-method();
        if (doc)
            view.the-document := doc;
        end if;
    end method;
```

Initializing the Application

The application provides an initialization method and uses it as an argument to the `set-library-init-function` macro to register the initialization method's name. The application also defines a constant that is set to `start` so that when the Listener is set to the `skeleton-module` module, the constant can be used to start the application. Listing 23-4 shows the `init-skeleton` initialization method, the `set-library-init-function` macro, and the `run-skeleton` constant.

Listing 23-4 Initializing the application

```
define method init-skeleton () => ()
    install-menu (get-resource-menu($file-menu-id));
    install-menu (get-resource-menu($edit-menu-id));
    make-debug-menu();

    add-behavior(*main-handler*, make(<app-behavior>));
    add-document-type
        (*main-handler*, ostype("PICT"), <my-document>);
    open(make(<my-document>));
end method;

set-library-init-function(init-skeleton);

define constant run-skeleton = start;
```

The `init-skeleton` method initializes the application by performing the following actions:

- installs the File and Edit menus

- creates a Debug menu
- adds the application behavior to the main handler
- sets the kind of document supported by the application
- creates a document and opens it

Note
The `make-debug-menu` method is a "goodie," which is an object provided by the Framework as a template for developing similar objects or simply provided as a convenience. Check the `goodie` source container for useful ideas as you develop your application. ◆

Implementing a Menu Item

The skeleton application only implements one menu item, New. The menu item is already been defined in the applications resource fork and is loaded by the `init-skeleton` method described in the previous section. Enabling the menu item and responding when the user chooses New is implemented in the `<app-behavior>` behavior associated with the main handler.

Listing 23-5 shows the `behavior-setup-menus` method that enables the New menu item and the `behavior-event` method that responds by creating a document object and opening it when New is selected.

Listing 23-5 Implementing a menu item

```
define method behavior-setup-menus
                            (behavior :: <app-behavior>,
                            next :: <list>,
                            main-handler :: <main-handler>) => ()
    next-method();

    enable-item(#"new");
end method;

define method behavior-event (behavior :: <app-behavior>,
                                    next :: <list>,
```

```
                                    main-handler :: <main-handler>,
                                    event :: <menu-event>,
                                    id == #"new") => ()
    ignore(behavior, next, main-handler, event, id);
    open(make(<my-document>));
end method;
```

Note
The call to the `next-method` in `behavior-setup-menus` allows other event handlers in the target chain to install their menus also. ◆

Creating Windows, Views, and Adorners

The Framework's `open` method whose parameter is specialized on a document object calls the `make-windows` method. This is where the document's window should be created. It is also a convenient place to create the window's views and the views' adorners. As you create views, you can also add behaviors to them.

Listing 23-6 shows the `make-windows` method whose parameter is specialized on `<my-document>`.

Listing 23-6 Setting up windows, views, and adorners

```
define method make-windows (document :: <my-document>) => ()
    let view = make(<my-view>,
                     location: point(0, 0),
                    extent:   point(1000, 1000),
                    document: document);

    let static-text = make(<static-text>,
                    location: point(0, 0),
                    extent: point(200, 50),
                    identifier: #"static-text",
                    text: get-string($skeleton-string-id));
    add-adorner(static-text, make(<frame-adorner>));
```

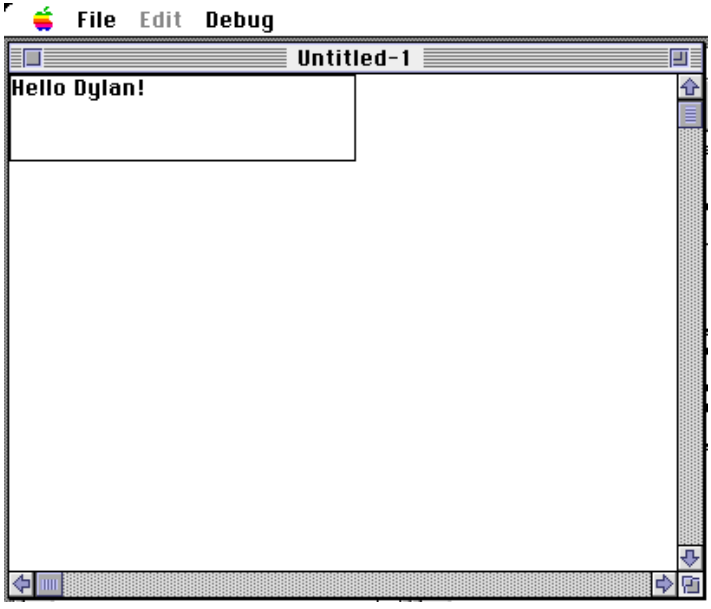
```
    let window = make-scrolling-window(list(view),
                                       title: document.title,
                                       next-handler: document,
                                       main-window: #t,
                                       location: point(100, 100),
                                       extent: point(400, 300),
                                       closable: #t,
                                       zoomable: #t, resizable: #t,
                                       target-view: static-text);

    add-sub-view(view, static-text);
    add-behavior(window, make(<simple-scripting-behavior>));
    open(window);
end method;
```

The `make-windows` method performs the following actions:

- creates a view object to render the document's data.
- creates a static text view whose text is in the `skeleton-library` library; it is specified by the `$skeleton-string-id` constant.
- adds a frame as an adorner around the static text view. The Framework defines the `<frame-adorner>` class for you.
- makes a scrolling window with the `<my-view>` object, `view`, as a subview of the root view. Note that the `make-scrolling-window` method is another goodie.
- adds the static text view as a subview of the view.
- adds a behavior that allows the text to be retrieved and set by an Apple script, as described in "Converting between Dylan objects and Apple event descriptor records" on page 420.
- opens the window.

Figure 23-1 shows the initial window after it has been opened.

Figure 23-1 The initial screen of the skeleton application



Drawing in a View

The document's data is stored in a list within the document object. The view that renders the data can reference the document through its `the-document` slot. When the view is drawn, the Framework calls the view's `draw` method, which iterates over the data and displays each rectangle stored in the list. Listing 23-7 shows the `draw` method.

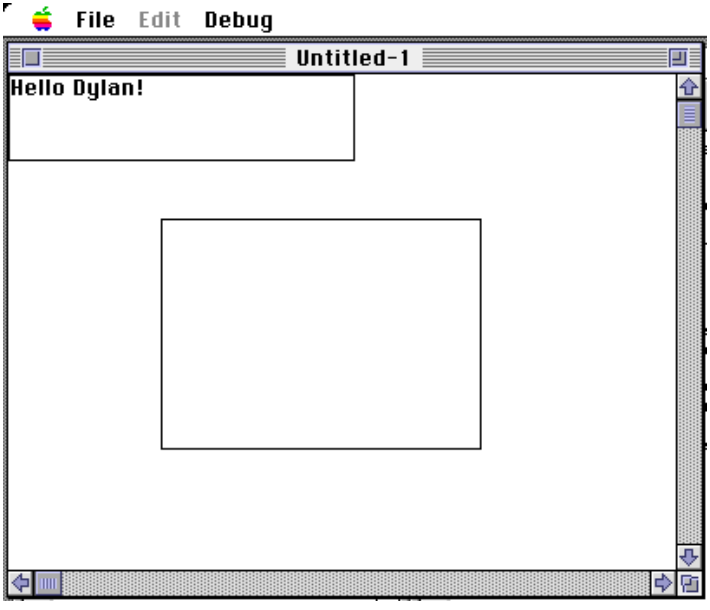
Listing 23-7 Specifying how to draw the contents of a view

```
define method draw (view :: <my-view>,
                    draw-region :: <region>) => ()
  ignore(draw-region);
  for (a_rect in view.the-document.data)
```

```
    frame-region(a_rect);
  end for;
end method;
```

Figure 23-2 shows the initial window after one rectangle has been drawn.

Figure 23-2 The screen after drawing data



Handling an Event

The `do-event` method specifies the actions to take when the specified event is passed up the target chain and the specified event handler has the chance to respond. Listing 23-8 shows the `do-event` method that specifies the actions to perform when a mouse-down event is handled by the `<my-view>` event handler.

Note
The `<my-view>` view is given the chance to handle the event because it is a subview of the root view. The `<my-view>` view is not actually in the target chain, which is from the `<static-text>` view, through the root view, and up. ◆

Listing 23-8 Handling an event

```
define method do-event (view :: <my-view>,
                        event :: <mouse-down-event>,
                        id :: <object>) => ()
    ignore(event, id);
    let final-point = track-mouse(make(<my-tracker>),
                                  view, event.local-mouse);
    let draw-rect = make(<rect>);
    set-rect-from-points(draw-rect, event.local-mouse, final-point);
    view.the-document.data :=
        insert(view.the-document.data, draw-rect, last: #t);
    invalidate(view, draw-rect);
end method;
```

The `do-event` method in Listing 23-8 performs the following actions:

- tracks the mouse. The point where tracking ends is returned by the `track-mouse` method. See the following section for implementation information.
- creates a rectangle and sets its coordinates from the point of the mouse down event to the point where the mouse is released.
- inserts the rectangle as the last one in the document's list of data.
- invalidates the area represented by the rectangle so that the Framework will redraw the area when it receives the next update event.

Implementing Mouse Tracking

Mouse tracking is implemented with four methods:

- the `track-begin` method sets up mouse tracking

- the `track-end` method cleans up mouse tracking
- the `track-constrain` method performs actions to keep the feedback within the specified bounds even if the mouse moves outside of them
- the `track-feedback` method provides visual feedback during mouse tracking

The Framework calls these methods at the appropriate time after you call `track-mouse`, as shown in the previous section.

Listing 23-9 shows the `<my-tracker>` class, on which parameters to the four methods are specialized, and the methods themselves. The `track-begin` method sets up the constraint rectangle to match the extent of the view. The `track-end` method turns off feedback. The `track-constrain` method limits the current mouse position to be within the rectangle, The `track-feedback` method draws an outline of the rectangle as the mouse moves.

Listing 23-9 Implementing mouse tracking

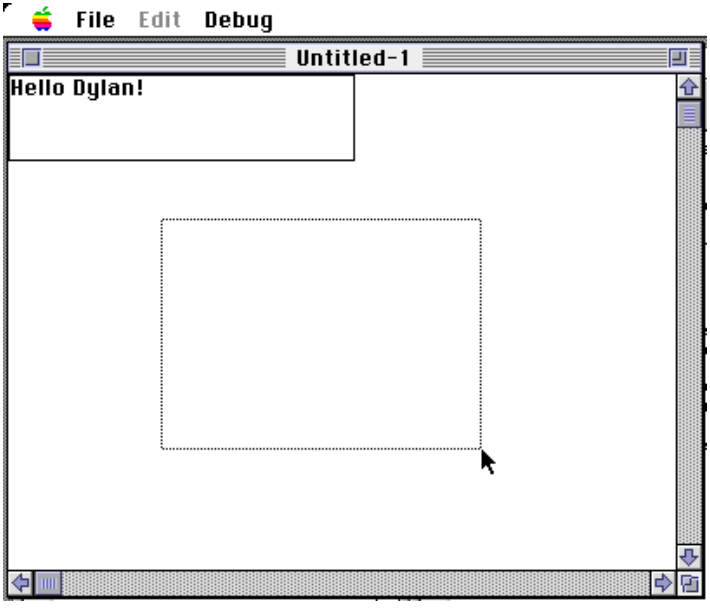
```
define class <my-tracker> (<tracker>)
    slot constraint :: <rect>,
        init-function:
            method()
                rect(0,0,0,0);
            end method;
end class;


define method track-begin (tracker :: <my-tracker>) => ()
    ignore(tracker);
    tracker.constraint.top := tracker.tracker-view.location.v;
    tracker.constraint.left := tracker.tracker-view.location.h;
    tracker.constraint.bottom := tracker.tracker-view.location.v
                               + tracker.tracker-view.extent.v;
    tracker.constraint.right := tracker.tracker-view.location.h
                               + tracker.tracker-view.extent.h;
end method;


define method track-end (tracker :: <my-tracker>,
                         final-point :: <point>) => ()
```

```
    ignore(final-point);

    track-feedback(tracker, final-point, final-point, #"off");
end method;

define method track-constrain (tracker :: <my-tracker>,
                               last-point :: <point>,
                               current-point :: <point>)
                               => constrained-point :: <point>;
    ignore(last-point);
    let new-v = current-point.v;
    let new-h = current-point.h;

    if (current-point.v < tracker.constraint.top)
        new-v := tracker.constraint.top;
    end if;
    if (current-point.v > tracker.constraint.bottom)
        new-v := tracker.constraint.bottom;
    end if;
    if (current-point.h < tracker.constraint.left)
        new-h := tracker.constraint.left;
    end if;
    if (current-point.h > tracker.constraint.right)
        new-h := tracker.constraint.right;
    end if;

    point(new-h, new-v);
end method;


define method track-feedback (tracker :: <my-tracker>,
                               last-point :: <point>,
                               current-point :: <point>,
                               mode :: <symbol>) => ()
    ignore(last-point, mode);

    let draw-rect = make(<rect>);
    set-rect-from-points(draw-rect,
    tracker.start-point, current-point);
```

```
    PenMode($srcXor);
    PenPat($gray-pattern);
    frame-region(draw-rect);
    PenNormal();
end method;
```

Figure 23-3 shows the track feedback for the rectangle in Figure 23-2 on page 419.

Figure 23-3 Providing feedback during mouse tracking



Implementing Open Scripting Architecture support

The Framework supports the Open Scripting Architecture by allowing you to send and receive Apple events. The skeleton application shows you how to convert a Dylan object to and from an Apple event descriptor, and shows how to respond to the get and set data Apple events.

Figure 23-4 shows an Apple script that communicates with the skeleton application. It retrieves the contents of the front window of the application and sets its contents.

Figure 23-4 An Apple script

```
tell application "Application Nub"
    set message to the contents of the front window
    set dialogResult to display dialog "Set hello message to:" default answer message
    if button returned of dialogResult = "OK" then
        set the contents of the front window to the text returned of dialogResult
    end if
end tell
```

The Framework dispatches Apple events to the target chain; they are handled in the same way as other events. The Framework provides a `<property>` class that represents conversion between the Apple event descriptor record and the Dylan object. You must define a subclass of the `<property>` class that specifies how to perform the conversion.

Listing 23-10 shows the `<static-text-message-property>` subclass and the conversion methods for text in the static text view of the skeleton application.

Listing 23-10 Converting between Dylan objects and Apple event descriptor records

```
define class <static-text-message-property> (<property>)
end class;

define method get-property-value
                    (property :: <static-text-message-property>,
                     property-type == $pContents)
                    => value :: <ae-desc>;
    ignore(property-type);
    let text-view
        = find-sub-view(property.container, #"static-text");
    make-descriptor(text-view.text);
end method;

define method set-property-value
```

```
                    (property :: <static-text-message-property>,
                     property-type == $pContents,
                     data :: <ae-desc>) => ()
    ignore(property-type);
    let text-view
        = find-sub-view(property.container, #"static-text");
    text-view.text := as-string(data);
    invalidate-all(text-view);
end method;
```

The `get-property-value` method specifies how convert a Dylan object to an Apple event descriptor record. The actual conversion is performed by the `make-descriptor` method.

The `set-property-value` method specifies how convert an Apple event descriptor record to a Dylan object and sets the property. The conversion is performed by the `as-data` method. The property's container is also invalidated so that the contents will be redrawn.

Listing 23-11 shows the behavior that implements the response to the get and set data Apple events. The behavior is attached to the window that contains the static text view, as shown in Listing 23-6 on page 412.

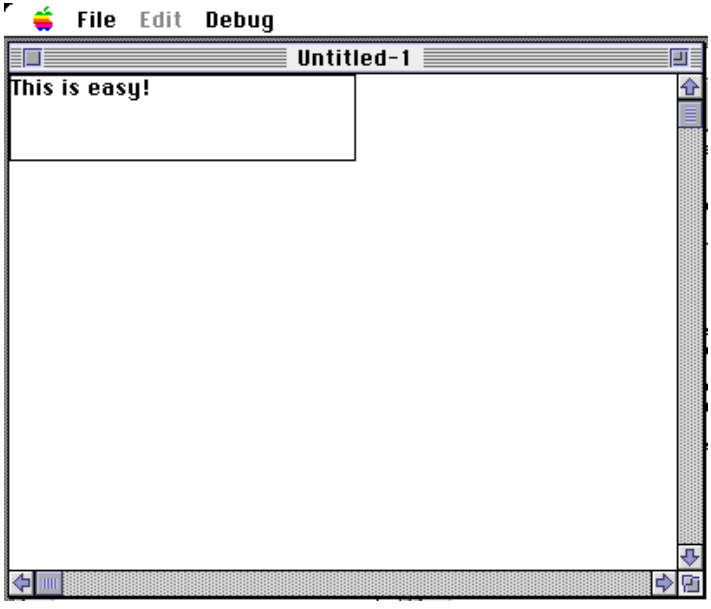
Listing 23-11 Responding to an Apple event

```
define class <simple-scripting-behavior> (<behavior>)
end class;

define method behavior-get-property
                    (behavior :: <simple-scripting-behavior>,
                     next :: <list>,
                     window :: <window>,
                     property-type == $pContents)
                    => property :: <property>;
    ignore(behavior, next);

    make(<static-text-message-property>,
        property-type: property-type,
        container: window);
end method;
```

The `behavior-get-property` method is executed when a get or set Apple event occurs and view to which the behavior is attached is allowed to respond. This method creates the property object that specifies how the Apple event descriptor record is converted and handled.

Figure 23-5 shows the initial window after it has been changed by the Apple script in Figure 23-4.

Figure 23-5 The static text view after being changed by an Apple script



Index

A

access paths 368
actual arguments 88
adorners 398, 412
anonymous methods 145
applicable arguments 294
applicable methods 288
argument expressions 88
arguments 45, 88
arithmetic functions 209
assignment operator 75, 219
associativity 217, 220
automatic memory management 6

B

bare methods 136
begin-end statements 108
binary operators 218
bindings 66
bit-oriented functions 213
blocks of code 363
bodies of code 108
Boolean literals 49
Boolean values 48, 189
browsers 4
built-in classes 48, 188
built-in functions 84
built-in macros 332
byte strings 230

C

callbacks 387
callouts 386
case statements 156
C-compatible libraries 368
changing the target 402
character literals 50
characters 50
class-allocated slots 262
class definitions 30, 193, 249, 409
class dynamism 185, 276, 320
classes 47, 182
class names 251
class precedence list 273
closures 6
closure variables 142
collection classes 6, 54, 192, 226
collection clauses 169
collection iteration variables 172
collections 226
commas 17
comments 15
comparisons 214
conditional evaluation 339
conditional statements 110, 152
conditions 350
congruent parameter lists 288
constant slots 264
constituents 13, 26
conversion 212
cross-language calls 386

D

declining conditions 359
`define class` defining form 249
`define constant` defining macro 71
`define method` defining form 119

`define variable` defining macro 70
defining classes 409
definite extent 67
definitions 13, 26, 28
delimiters 15, 19
development environment 4
direct instances 274
direct methods 118, 136
direct subclasses 272
direct superclasses 272
division 210
documents 399
drawing in a view 414
dynamism 85, 248, 276, 315, 322

E

editable text 398
elements 226
else clause 154
end-test expressions 174
equality 214
errors 350, 351
evaluation 27, 90
event handler 399
exception clauses 363
exit functions 363
explicit clauses 169
explicit generic function creation 293
explicit iteration variables 173
exporting modules 316
exporting variables 310
expressions 14, 27, 33
extent 67, 142
extents 397

F

floating-point literals 54
floating-point numbers 54, 204, 207
formal parameters 93, 120
for statements 169
framework 393
framework tutorial 407
free classes 278
free variables 342
functional arguments 146
function calls 14, 34, 36, 84, 87
function call syntax 87
function definitions 84
function macros 338
function results 45, 91, 94, 126
functions 84

G

general instances 274
general superclasses 273
generic functions 85, 288
getter functions 194, 252, 257
goodie source container 411

H

handlers 350, 355
handling an event 415
handling conditions 355
hash tables 226
heterarchies 185, 278
hierarchies 273

I

if statements 153
immediate values 50
immutable objects 50
implicit bodies 108
implicit generic function creation 288
importing C entities 378
importing modules 316

importing variables 311
indefinite extent 67
inheritance 184, 248, 272
inherited slots 280
initialization keywords 254
initialize functions 409
initializing the application 410
instances 182
instance slots 262
instantiability 248, 274
integer literals 53
integers 53, 205
interface definitions 368
iteration variables 169
iterative statements 112
iterators 165
iterator statements 166

K

keyed collections 226
keys 226
keyword arguments 90
keyword parameters 122
keyword syntax 52

L

language extensions 4
`let` declaration form 125
libraries 12, 315
library 408
library definitions 315
list literals 57
lists 56, 238
literal constants 14, 33
local declarations 14, 26, 31, 109, 125
local method declarations 32
local methods 138
local variable declarations 32, 125
local variables 67, 72
logical functions 213

M

Macintosh Toolbox functions 371
macro calls 166, 332
macro definitions 332
macro expansions 332
macros 332
main handler 400
`max` function 45
memory 6
memory management 6
menu events 401
menu items 411
method Bodies 94
method bodies 124
method closures 142
method definitions 31, 119
method dispatch 85, 288, 294
method objects 118
methods 84, 118
method specificity 288, 294, 297
module 408
module definition 308
modules 13, 308
module variables 66
mouse events 402
mouse tracking 416
multiple inheritance 276
multiple-line comments 15
multiple values 72, 91
mutable collections 227
mutable objects 50
mutual recursion 140

N

named constants 71
name mapping 379
namespace 308

naming conventions 68
next handler 400
next method 300
non-local exits 363
number classes 52, 190
numbers 53, 204
number text 398
numeric clauses 169
numeric functions 207
numeric iteration variables 170

O

objects 46, 182
Open Scripting Architecture support 419
operator call syntax 88, 215
operator precedence 220
operators 215
optional arguments 89
otherwise clause 157

P

pair literals 57
pairs 57, 239
parameter list congruency 288
parameter lists 120
parameter-list specifications 93, 120
parameter type specialization 123
patterns 332, 334
pattern variables 334
polymorphism 5, 85, 288
precedence 217, 220
predicates 208
primary classes 278
programs 12
projects 12, 315

R

ratio literals 53
ratios 53, 204, 206
read-only variables 71
real numbers 204, 205
recovery 355
recovery protocol 350
recursion 139, 344
referencing variables 73
required arguments 89
required parameters 121
restarts 350, 361
rest parameters 122
result bodies 175
return values 45, 91, 94, 126
root view 395
rounding 210
runtime environment 4, 67, 308

S

scope 66, 109, 142, 308
scrolling window 396
sealing classes 323
sealing generic function branches 324
sealing generic functions 323
sealing slot accessors 325
select statements 158
semicolons 17
sequences 226, 228
setter functions 195, 252, 257, 264
shared libraries 373
signaling warnings 353
simple object vectors 235
simplification 212
singaling errors 351
single-line comments 15
singleton parameters 124
singletons 296
skeleton application 407
slot accessors 257, 291

slot allocation 262, 282
slot inheritance 280
slot reference syntax 195
slots 30, 194, 248, 252, 256
slot specifications 252
source folders 13
source records 13
specializers 70
special operators 219
Standard Dylan 4
standard function call syntax 87
statement macros 106, 152, 166, 336
statements 14, 34, 37, 106
static text 398
stretchy byte strings 231
stretchy collections 227
string literals 55
strings 55, 229
subclasses 184
subprojects 316
subviews 397
superclasses 184, 252, 272
superviews 397
symbol literals 51
symbols 51, 189
syntax 17
syntax descriptions 18

T

target chain 399
templates 332, 335
termination 7, 355
test expressions 107, 153
text 398
`truncate` function 45
type conversion 372
type specializers 294

U

unary operators 217
unless statements 155
until statements 168
user-defined classes 193, 248

V

values 44, 69
variable definitions 29
variable names 68
variable references 14, 33, 35, 73
variables 65
vector literals 56
vectors 55, 234
view determiners 397
view hierarchy 395
views 395, 412

W

warnings 350, 353
while statements 166
whitespace 15
windows 395, 412

This Apple manual was written, edited, and composed on a desktop publishing system using Apple Macintosh computers and FrameMaker software. Proof pages were created on an Apple LaserWriter Pro printer. Final pages were created on a Docutek. Line art was created using Adobe Illustrator™ and Adobe Photoshop™. PostScript™, the page-description language for the LaserWriter, was developed by Adobe Systems Incorporated.

Text type is Palatino® and display type is Helvetica®. Bullets are ITC Zapf Dingbats®. Some elements, such as program listings, are set in Apple Courier.

PRODUCTION MANAGER
Trish Eastman

LEAD WRITER
Linda Kyrnitszke

WRITERS
Marq Laube, Gary McCue

ILLUSTRATOR
Sandee Karr

PRODUCTION EDITORS
Lorraine Findlay

SPECIAL THANKS TO
Kim Barrett, Bob Cassels, Phil Kania, Ross Knights, Mike Lockwood, David Moon, Andrew Shalit, David Sotkowitz, Steve Strassman