Programming Languages and Techniques

Lecture Notes for CIS 120

Steve Zdancewic
Benjamin C. Pierce
Stephanie Weirich

University of Pennsylvania

September 8, 2013

Contents

1 Overview and Program Design **9**
1.1 Introduction and Prerequisites 9
1.2 Course Philosophy 9
1.3 How the different parts of CIS 120 fit together 13
1.4 Course History 16
1.5 Program Design 16

2 Introductory OCaml **23**
2.1 OCaml in CIS 120 23
2.2 Primitive Types and Expressions 23
2.3 Value-oriented programming 25
2.4 `let` declarations 27
2.5 Local `let` declarations 30
2.6 Function Declarations 32
2.7 Types 35
2.8 Failwith 37
2.9 Commands 38
2.10 A complete example 41
2.11 Notes 43

3 Lists and Recursion **45**
3.1 Lists 45
3.2 Recursion 50

4 Tuples and Nested Patterns **61**
4.1 Tuples 61
4.2 Nested patterns 63
4.3 Exhaustiveness 65

4.4 Wildcard (underscore) patterns 65
4.5 Examples 66

5 User-defined Datatypes 67
5.1 Atomic datatypes: enumerations 67
5.2 Datatypes that carry more information 70
5.3 Type abbreviations 72
5.4 Recursive types: lists 73

6 Binary Trees 75

7 Binary Search Trees 79
7.1 Creating Binary Search Trees 82

8 Generic Functions and Datatypes 87
8.1 User-defined generic datatypes 89
8.2 Why use generics? 91

9 Modularity and Abstraction 93
9.1 A motivating example: finite mathematical sets 93
9.2 Abstract types and modularity 94
9.3 Another example: Finite Maps 102

10 First-class Functions 105
10.1 Partial Application and Anonymous Functions 106
10.2 List transformation 109
10.3 List fold 111

11 Partial Functions: option types 115

12 Unit and Sequencing Commands 119
12.1 The use of ';' 121

13 Records of Named Fields 123
13.1 Immutable Records 123

14 Mutable State and Aliasing 125
14.1 Mutable Records 126
14.2 Aliasing: The Blessing and Curse of Mutable State 129

15 The Abstract Stack Machine **133**
15.1 Parts of the ASM 134
15.2 Values and References to the Heap 135
15.3 Simplification in the ASM 139

16 Linked Structures: Queues **149**
16.1 Representing Queues 150
16.2 The Queue Invariants 154
16.3 Implementing the basic Queue operations 155
16.4 Iteration and Tail Calls 157
16.5 Loop-the-loop: Examples of Iteration 160
16.6 Infinite Loops 163

17 Local State **167**
17.1 Closures 168
17.2 Objects 170
17.3 The generic `'a ref` type 172
17.4 Reference (==) vs. Structural Equality (=) 173

18 Wrapping up OCaml: Designing a GUI Library **175**
18.1 Taking Stock 175
18.2 The Paint Application 176
18.3 OCaml's Graphics Library 178
18.4 Design of the GUI Library 179
18.5 Localizing Coordinates 181
18.6 Simple Widgets & Layout 184
18.7 The widget hierarchy and the `run` function 187
18.8 The Event Loop 189
18.9 GUI Events 191
18.10Event Handlers 192
18.11Stateful Widgets 195
18.12Listeners and Notifiers 196
18.13Buttons (at last!) 200
18.14Building a GUI App: Lightswitch 201

19 Transition to Java **205**
19.1 Farewell to OCaml 205
19.2 Objects and Classes 209

19.3 Mutability and `null` 212
19.4 Core Java 214

20 Interfaces and Subtypes **219**
20.1 Interfaces 220
20.2 Subtyping 222
20.3 Multiple Interfaces 223

21 Static vs. Dynamic **227**
21.1 Static vs. Dynamic Methods 227
21.2 Static Types vs. Dynamic Classes 230

22 Java Design Exercise: Resizable Arrays **233**
22.1 Arrays 233
22.2 Resizable Arrays 240

23 The Java ASM and encapsulation **249**
23.1 Differences between OCaml and Java Abstract Stack Machines 249

24 Extension and Inheritance **257**
24.1 Interface Extension 257
24.2 Inheritance 259
24.3 `Object` 263

25 The Java ASM and dynamic methods **265**
25.1 Refinements to the Abstract Stack Machine 267
25.2 The revised ASM in action 268

26 Encapsulation and Queues **285**
26.1 Queues in ML 286
26.2 Queues in Java 287
26.3 Implementing Java Queues 288

27 Generics, Collections, and Iteration **293**
27.1 Polymorphism and Generics 294
27.2 Subtyping and Generics 296
27.3 The Java Collections Framework 297
27.4 Iterating over Collections 300

28 Exceptions **305**
28.1 Ways to handle failure 306
28.2 Exceptions in Java 307
28.3 Exceptions and the abstract stack machine 309
28.4 Catching multiple exceptions 310
28.5 Finally 311
28.6 The Exception Class Hierarchy 312
28.7 Checked exceptions 313
28.8 Undeclared exceptions 314
28.9 Good style for exceptions 315

29 IO **319**
29.1 Working with (Binary) Files 320
29.2 PrintStream 322
29.3 Reading text 324
29.4 Writing text 324
29.5 Histogram demo 326

30 Swing: GUI programming in Java **335**
30.1 Drawing with Swing 337
30.2 User Interaction 341
30.3 Action Listeners 345
30.4 Timer 346

31 Swing: Layout and Drawing **349**
31.1 Layout 349
31.2 An extended example 352

32 Swing: Interaction and Paint Demo **371**
32.1 Version A: Basic structure 371
32.2 Version B: Drawing Modes 375
32.3 Version C: Basic Mouse Interaction 376
32.4 Version D: Drag and Drop 377
32.5 Version E: Keyboard Interaction 379
32.6 Interlude: Datatypes and enums vs. objects 381
32.7 Version F: OO-based Refactoring 383

Chapter 1

Overview and Program Design

1.1 Introduction and Prerequisites

CIS 120 is an introductory computer science course taught at the University of Pennsylvania.

Entering students should have some previous exposure to programming and the ability to write small programs (10-100 lines) in some imperative or object-oriented language. Because of CIS 110 and AP Computer Science, the majority of entering students are familiar with Java. However, students with experience in languages such as Python, C, C++, Matlab, or Scheme have taken this course and done well.

In particular, the skills that we look for in entering CIS 120 students are familiarity with the basic *tools* of programming, including editing, compiling and running code, and familiarity with the basic *concepts* of programming languages, such as variables, assignment, conditionals, objects, methods, arrays, and various types of loops.

1.2 Course Philosophy

The core of CIS 120 is *programming*. The skill of writing computer programs is fun, useful and rewarding in its own right. By the end of the semester, students should be able to design and implement—from scratch—sophisticated applications involving graphical user interfaces, nontrivial data structures, mutable state, and complex control. In fact, the

final homework assignment for the course is a playable game, chosen and designed by the student.

More importantly, programming is a conceptual foundation for the study of computation. This course is a prerequisite for almost every other course in the computer science program at Penn, both those that themselves have major programming components and those of a more theoretical nature. The science of computing can be thought of as a modern day version of logic and critical thinking. However, this is a more concrete, more potent form of logic: logic grounded in computation.

Like any other skill, learning to program takes plenty of practice to master. The tools involved—languages, compilers, IDEs, libraries, and frameworks—are large and complex. Furthermore, many of these tools are tuned for the demands of rigorous software engineering, including extensibility, efficiency and security.

The general philosophy for introductory computer science at Penn is to develop programming skills in stages. We start with basic skills of "algorithmic thinking" in our intro CIS 110 course, though students enter Penn already with this ability through exposure to AP Computer Science classes in high school, through summer camps and courses on programming, or independent study. At this stage, students can write short programs, but may have less fluency with putting them together to form larger applications. The first part of CIS 120 continues this process by developing design and analysis skills in the context of larger and more challenging problems. In particular, we teach a systematic process for program design, a rigorous model for thinking about computation, and a rich vocabulary of computational structures. The last stage (the second part of CIS 120 and beyond) is to translate those design skills to the context of industrial-strength tools and design processes.

This philosophy influences our choice of tools. To facilitate practice, we prefer mature platforms that have demonstrated their utility and stability. For the first part of CIS 120, where the goal is to develop design and analysis skills, we use the OCaml programming language. In the second half of the semester, we switch to the Java language. This dual language approach allows us to teach program design in a relatively simple environment, make comparisons between different programming paradigms, and motivate sophisticated features such as objects and classes.

OCaml is the most-widely used dialect of the ML family of languages. Such languages are not new—the first version of ML, was designed by

Robin Milner in the 1970s; the first version of OCaml was released in 1996. The OCaml implementation is a free, open source project developed and maintained by researchers at INRIA, the French national laboratory for computing research. Although OCaml has its origins as a research language, it has also attracted significant attention in industry. For example, Microsoft's F# language is strongly inspired by OCaml and other ML variants. Scala and Haskell, two other strongly typed functional programming languages, also share many common traits with OCaml.

Java is currently one of the most popularly used languages in the software industry and representative of software object-oriented development. It was originally developed by James Gosling and others at Sun Microsystems in the early nineties and first released in 1995. Like OCaml, Java was released as free, open source software and all of the core code is still available for free. Popular languages related to Java include C# and, to a lesser extent, C++.

Goals

There are four main, interdependent goals for CIS 120.

Increased independence in programming While we expect some familiarity with programming, we don't expect entering students to be full-blown programmers. The first goal of 120 is to extend their programming skills, going from the ability to write program that are 10s of lines long to programs that are 1000s of lines long. Furthermore, as the semester progresses, the assignments become less constrained, starting from the application of simple recipes, to independent problem decomposition.

Fluency in program design The ability to write longer programs is founded on the process of program design. We teach necessary skills, such as test-driven development, interface specification, modular decomposition, and multiple programming idioms that extend individual problem solving skills to system development.

Firm grasp of fundamental principles CIS 120 is not just an introductory programming course; it is primarily an introductory computer *science* course. It covers fundamental principles of computing, such as recursion,

lists and trees, interfaces, semantic models, mutable data structures, references, invariants, objects, and types.

Fluency in core Java We aim to provide CIS 120 students with sufficient core skills in a popular programming language to enable further development in a variety of contexts, including: advanced CIS core courses and electives, summer internships, start-up companies, contributions to open-source projects and individual exploration. The Java development environment, including the availability of libraries, tools, communities, and job opportunities, satisfies this requirement. CIS 120 includes enough details about the core Java languages and common libraries for this purpose, though is not an exhaustive overview to Java or object-oriented software engineering. There are many details about the Java language that CIS 120 does not cover; the goal is to provide enough information for future self study.

Why OCaml?

The first half of CIS 120 is taught in the context of the OCaml programming language. In the second half of the semester, we switch to Java for lectures and assignments. If the goal is fluency in core Java, why use OCaml at all?

We use OCaml in CIS 120 for several reasons.

"[The OCaml part of the class] was very essential to getting fundamental ideas of comp sci across. Without the second language it is easy to fall into routine and syntax lock where you don't really understand the bigger picture." —CIS 120 Student

It's not Java. By far, the majority of students entering CIS 120 know only the Java programming language. Different programming languages foster different programming paradigms. Only knowing one language leads to a myopic view of programming, being unable to separate "that is how it is done" from "that is how it is done in Java". By switching languages, we provide perspective about language-independent concepts, introducing other ways of decomposing and expressing computation that are not as well supported by Java.

Furthermore, not all students have this background in Java, nor do they have the same degree of experience. We start with OCaml to level the playing field and give students with alternative backgrounds a chance to develop sophistication in programming and review the syntax of Java on

their own. (For those that need more assistance with the basics, we offer CIS 110.)

Finally, learning a new programming language to competence in six weeks builds confidence. A student's first language is difficult, because it involves learning the concepts of programming at the same time. The second comes much easier, once the basic concepts are in place, and the second is understood more deeply because the first provides a point of reference.

Rich, orthogonal vocabulary.

We specifically choose OCaml as an alternative language because of several properties of the language itself. The OCaml language provides native features for many of the topics that we would like to study: including lists and tree-like data structures, interfaces, and first-class computation. These features can be can be studied in isolation as there are few intricate interactions with the rest of the language.

"[OCaml] made me better understand features of Java that seemed innate to programming, which were merely abstractions and assumptions that Java made. It made me a better Java programmer." —CIS 120 Student

Functional programming. OCaml is a *functional programming* language, which encourages the use of "persistent" (immutable) data structures. In contrast, every data structure in Java is mutable by default. Although CIS 120 is not a functional programming course, we find that the study of persistent data structures is a useful introduction to programming. In particular, persistence leads to a simpler, more mathematical programming model. Programs can be thought of in terms of "transformations" of data instead of "modifications," leading to simpler, more explicit interfaces.

1.3 How the different parts of CIS 120 fit together

Homework assignments provide the core learning experience for CIS 120. Programming is a skill, one that is developed through practice. The homework assignments themselves include both "etudes", which cover basic concepts directly and "applications" which develop those concepts in the context of a larger purpose. The homework assignments take time.

Lectures Lectures serve many roles: they *introduce*, *motivate* and *contextualize* concepts, they *demonstrate* code development, they *personalize* learning, and *moderate* the speed of information acquisition.

To augment your learning, we make lecture slides, demonstration code and lecture notes available on the course website. Given the availability of these resources, why come to class at all?

- To see code development in action. Many lectures include live demonstrations of code design, and only the resulting code is posted on the website. That means the thought process of code development is lost: your instructors will show some typical wrong turns, discuss various design trade-offs and demonstrate debugging strategies. Things will not always go according to plan, and observing what to do when this happens is also a valuable lesson.

- To interact with the new material as it is presented, by asking questions. Sometimes asking a question right at the beginning can save a lot of later confusion. Also to hear the questions that your classmates have about the new material. Sometimes it is difficult to realize that you don't fully understand something until someone else raises a subtle point.

- To regulate the timing of information flow for difficult concepts. Sure, you can read the lecture notes in less than fifty minutes. However, sometimes slowing down, working through examples, and thinking deeply is required to internalize a topic. You may have more patience for this during lecture than while reading lecture notes on your own.

- For completeness. We cannot promise to include everything in the lectures in the lecture notes. Instructors are only human, with limited time to prepare lecture notes, particularly at the end of the semester.

Labs The labs (or "recitations") provide a small-group setting for coding practice and individual attention from the course staff.

The lab grade comes primarily from lab participation. The reason is that the lab is there to get you to write code in a low-stress environment. In this environment, we use pair programming, teaming you up with your classmates to find the solutions to the lab problems together. The purpose

of the teams is not to divide the work—in fact, we expect that in many cases it will take longer to complete the lab work using a team than by doing it yourself! The benefit of team work lies in discussing the entire exercise with your partner—often you will find that you and your partner have different ideas about how to solve the problem and find different aspects difficult.

Furthermore, labs are another avenue for coding practice, adding to the quantity of code that you write during the semester. The folllowing parable illustrates the value of this:

> The ceramics teacher announced he was dividing his class into two groups. All those on the left side of the studio would be graded solely on the quantity of work they produced, all those on the right graded solely on its quality.
>
> His procedure was simple: on the final day of class he would weigh the work of the quantity group: 50 pounds of pots rated an A, 40 pounds a B, and so on. Those being graded on quality, however, needed to produce only one pot — albeit a perfect one — to get an A.
>
> Well, come grading time and a curious fact emerged: the works of highest quality were all produced by the group being graded for quantity!
>
> It seems that while the quantity group was busily churning out piles of work — and learning from their mistakes — the quality group had sat theorizing about perfection, and in the end had little more to show for their efforts than grandiose theories and a pile of dead clay.

David Bayles and Ted Orland, from "Art and Fear" [1].

Exams The exams provide the fundamental assessment of course concepts, including both knowledge and application. Some students have difficulty seeing how pencil-and-paper problems relate to writing computer programs. Indeed, when completing homework assignments, powerful tools such as IDEs, type checkers, compilers, top-levels, and online documentation are available. None of these may be used in exams. Rather, the purpose of exams is to assess both your understanding of these tools and, more importantly, the way you *think* about programming problems.

1.4 Course History

The CIS 120 course material, including slides, lectures, exams, examples, homeworks and labs were all developed by faculty members at the University of Pennsylvania, including Steve Zdancewic, Benjamin Pierce, Stephanie Weirich, Fernando Pereira, and Mitch Marcus. From Fall 2002 to Fall 2010, the course was taught primarily using the Java programming language, with excursions into Python. In Fall 2010, the course instructors radically revised the material in an experimental section, introducing the OCaml language as a precursor to Java. Since Spring 2011, the dual-language course has been used for all CIS 120 students. In some ways, the use of OCaml as an introductory language can be seen as CIS 120 "returning to its roots." In the 1990s the course was taught only in functional languages: a mix of OCaml and Scheme. The last major revision of the course replaced these wholesale with Java. Joel Spolsky lamented this switch in a 2005 blog post entitled "The Perils of JavaSchools", available at `http://www.joelonsoftware.com/articles/ThePerilsofJavaSchools.html`.

1.5 Program Design

Program design is the *process* of translating informal specifications ("word problems") into running code. Learning how to design programs involves decomposing problems into a set of simpler steps.[1]

The program design recipe comprises four steps:

1. **Understand the problem.** What are the relevant concepts in the informal description? How do the concepts relate to each other?

2. **Formalize the interface.** How should the program interact with its environment? What types of input does it require? Why type of output should it produce? What additional information does it need? What properties about its input may it assume? What is true about the result?

[1] The material in this section is adapted from the excellent introductory textbook, *How to Design Programs* [3].

3. **Write test cases.** How does the program behave on typical inputs? On unusual inputs? On erroneous ones?

4. **Implement the required behavior.** Only after the first three steps have been addressed is it time to write code. Often, the implementation will require decomposing the problem into simpler ones and applying the design recipe again.

We will demonstrate this process by considering the following design problem:

> Imagine an owner of a movie theater who wants to know how much he should charge for tickets. The more he charges, the fewer people can afford tickets. After some experiments, the owner has determined a precise relationship between the price of a ticket and average attendance. At a price of $5.00 per ticket, 120 people attend a performance. Decreasing the price by a dime ($.10) increases attendance by 15. However, the increased attendance also comes at an increased cost: Each attendee costs another four cents ($0.04), on top of the fixed per-performance cost of $180. The owner would like to be able to calculate, for any given ticket price, exactly how much profit he will make.

We will develop a solution to this problem by following the design recipe in the context of the OCaml programming language. In the process, we'll introduce OCaml by example. In the next chapter, we give a more systematic overview of its syntax and semantics.

Step 1: Understand the problem. In the scenario above, there are five relevant concepts: the ticket *price*, (number of) *attendees*, *revenue*, *cost* and *profit*. Among these entities, we can define several relationships.

From basic economics we know the basic relationships between profit, cost and revenue. In other words, we have

$$profit = revenue - cost$$

and

$$revenue = price \times attendees$$

Also, the scenario tells us how to compute the cost of a performance

$$cost = \$180 + attendees \times \$0.04$$

but does not directly specify how the ticket price determines the number of attendees. However, because revenue and cost (and profit, indirectly) depend on the number of attendees, they are also determined by the ticket price.

Our goal is to determine the precise relationship between ticket price and profit. In programming terms, we would like to define a *function* that, given the ticket price, calculates the expected profit.

Step 2: Formalize the Interface. Most of the relevant concepts—cost, ticket price, revenue and profit—are dollar amounts. That raises a design choice about how to represent *money*. Like most programming languages, OCaml can calculate with integer and floating point values, and both are attractive for this problem, as we need to represent fractional dollars. However, the binary representation of floating point values makes it a poor choice for money, since some numeric values—such as 0.1—cannot be represented exactly, leading to rounding errors. (This "feature" is not unique to OCaml—try calculating 0.1 + 0.1 + 0.1 in your favorite programming language.) So let's represent money in cents and use integer arithmetic for calculations.

Our goal is to define a function that computes the profit given the ticket price, so let us begin by writing down a skeleton for this function—let's call it `profit`—and noting that it takes a single input, called `price`. Note that the first line of this definition uses type annotations to enforce that the input and and output of the function are both integers.[2]

```
let profit (price:int) : int =
  ...
```

Step 3: Write test cases. The next step is to write test cases. Writing test cases *before* writing any of the interesting code—the fundamental rule of

[2]OCaml will let you omit these type annotations, but including them is mandatory for CIS120. Using type annotations is good documentation; they also improve the error messages you get from the compiler. When you get a type error message, the first thing you should do is check that your type annotations correct.

test-driven program development—has several benefits. First, it ensures that you understand the problem—if you cannot determine the answer for one specific case, you will find it difficult to solve the more general problem. Thinking about tests also influences the code that you will write later. In particular, thinking about the behavior of the program on a range of inputs will help ensure that the implementation does the right thing for each of them. Finally having test cases around is a way of "futureproofing" your code. It allows you to make changes to the code later and automatically check that they have not broken the existing functionality.

In the situation at hand, the informal specification suggests a couple of specific test cases: when the ticket price is either $5.00 or $4.90 (and the number of attendees is accordingly either 120 or 135). We can use OCaml itself to help compute what the expected values of these test cases should be.

The OCaml `let`-expression gives a name to values that we compute, and we can use these values to compute others with the `let`-`in` expression form.

```
let profit_500 : int =
  let price     = 500 in
  let attendees = 120 in
  let revenue   = price * attendees in
  let cost      = 18000 + 4 * attendees in
  revenue - cost

let profit_490 : int =
  let price     = 490 in
  let attendees = 135 in
  let revenue   = price * attendees in
  let cost      = 18000 + 4 * attendees in
  revenue - cost
```

Using these, we can write the test cases themselves:

```
let test () : bool =
    (profit 500) = profit_500
;; run_test "profit at $5.00" test

let test () : bool =
    (profit 490) = profit_490
;; run_test "profit at $4.90" test
```

The predefined function `test` (provided by the `Assert` module) takes

no input and returns the boolean value `true` only when the test succeeds.[3] We then invoke the command `run_test` to execute each test case and give it a name, which is used to report failures in the printed output; if the test succeeds, nothing is printed. After invoking the first test case, we can define the `test` function to check the `profit` function's behavior at a different price point.

Step 4: Implement the behavior. The last step is to complete the implementation. First, the profit that will be earned for a given ticket price is the revenue minus the number of attendees. Since both revenue and attendees vary with to ticket price, we can define these as functions too.

```
let revenue (price:int) : int =
  ...

let cost (price:int) : int =
  ...

let profit (price:int) : int =
  (revenue price) - (cost price)
```

Next, we can fill these in, in terms of another function `attendees` that we will write in a minute.

```
let attendees (price:int) : int =
  ...

let revenue (price:int) : int =
  price * (attendees price)

let cost (price:int) : int =
  18000 + 4 * (attendees price)
```

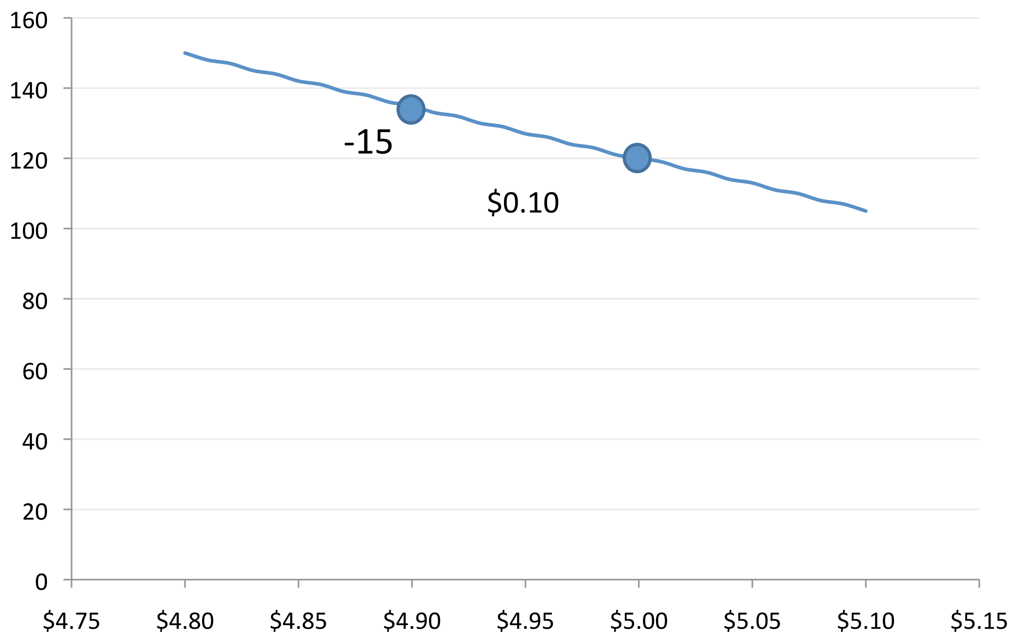
For `attendees`, we can apply the design recipe again. We have the same concepts as before, and the interface for `attendees` is determined by the code above. Furthermore, we can define the test cases for `attendees` from the problem statement.

[3] Note that single `=`, compares two values for equality in OCaml.

```
let test () : bool =
         (attendees 500) = 120
;; run_test "atts. at $5.00" test

let test () : bool =
    (attendees 490) = 135
;; run_test "atts. at $4.90'' test
```

To finish implementing `attendees`, we make the assumption that there is a linear relationship between the ticket price and the number of attendees. We can graph this relationship by drawing a line given the two points specified by the test cases in the problem statement.



We can determine what the function should be with a little high-school algebra. The equation for a line

$$y = mx + b$$

says that the number of attendees y, is equal to the slope of the line m, times the ticket price y, plus some constant value b. Furthermore, we can determine the slope of a line given two points:

$$m = \frac{\textit{difference in attendance}}{\textit{difference in price}} = \frac{-15}{10}$$

Once we know the slope, we can determine the constant b by solving the equation for the line for b and plugging in the numbers from either test case. Therefore $b = 120 - (-15/10) \times 500 = 870$.

Putting these values together gives us a mathematical formula specifying attendees in terms of the ticket price.

$$attendees = (-15/10) \times price + 870$$

Translating that math into OCaml nearly completes the program design.

```
let attendees (price : int) : int =
  (-15 / 10) * price + 870
```

Unfortunately, this code is not quite correct (as suggested by the pink background). Fortunately, however, our tests detect the issue, failing when we try to run the program and giving us a chance to think a little more carefully.

The problem turns out to be our choice of integer arithmetic. Dividing the integer -15 by the integer 10 produces the integer 1, rather than the exact answer 1.5. If, instead, we multiply by the price before dividing we retain the precision needed for the problem.

```
let attendees (price : int) : int =
  (-15 * price) / 10 + 870
```

Finally, note that there are other ways to implement `profit` that return the correct answer and pass all of the tests, but that are inferior to the one we wrote. For example, we could have written:

```
let profit (price:int) : int =
    price * (-15 * price / 10 + 870) -
    (18000 + 4 * (-15 * price / 10 + 870))
```

However, this program hides the structure and concepts of the problem. It duplicates sub-computations that could be shared, and it does not record the thought process behind the calculation.

Of course, for such a simple problem, this four-step design methodology may seem like overkill. During the course, we'll use it to attack much larger and more complex design problems, where its benefits will be clearer.

Chapter 2

Introductory OCaml

2.1 OCaml in CIS 120

OCaml is a rich and expressive programming language, but, for the purposes of CIS 120, we need only a very minimal subset of its features. Moreover, we use OCaml in a very stylized way that is designed to make using it as simple as possible and reduce the kinds of errors that students might encounter while writing their first programs.

These notes summarize the OCaml features that are needed in the first few homework assignments for CIS 120. As the course progresses, we will explain more concepts and the associated syntax, *etc.*, as needed.

Note: *Our intention is that these notes should be self contained: It should not be necessary to use any OCaml features not described here when completing the CIS 120 homework projects. If we've missed something, or if you find any of the explanations confusing, please post a question to the class discussion web site.*

2.2 Primitive Types and Expressions

In programming, a *type* describes the structure of some form of data and specifies a collection of operations for manipulating and computing with data of this form.

OCaml, like any programming language, supports various *primitive data types* like integers, booleans, and strings, all of which are built into the language. Each of these primitive data types supports some specific

Integers:	`int`	`...-2,-1,0,1,2,...`	constants
		`1 + 2`	addition
		`1 - 2`	subtraction
		`2 * 3`	multiplication
		`10 / 3`	integer division
		`10 mod 3`	modulus (remainder)
		`string_of_int 3`	convert int `3` to string `"3"`
Booleans:	`bool`	`true, false`	constants
		`not true`	logical negation
		`true && false`	and (conjunction)
		`true || false`	or (disjunction)
Strings:	`string`	`"hello","CIS 120",...`	constants
		`"\n"`	newline
		`"hello"^"world\n"`	concatenation (carat)
Generic comparisons (producing a `bool`):			
		`=`	equality
		`<>`	inequality
		`<`	less-than
		`<=`	less-than-or-equals
		`>`	greater-than
		`>=`	greater-than-or-equals

Figure 2.1: OCaml primitive data types and some operations on them.

operations; there are also a few generic operations that work on data of any type. Figure 2.1 gives a summary of a few basic OCaml types, some example constants of those types, and some of their operations.

Given these operations and constants, we can build more complex *expressions*, which are the basic unit of OCaml programs. For example, here are some simple expressions:

```
3 + 5

2 * (5 + 10)

"hello" ^ "world\n"

false && (true || false)

not ((string_of_int 42) = "forty two")
```

Parentheses are used to group expressions according to the usual laws of precedence. Note that different groupings may have different meanings (just as in math):

```
3 + (2 * 7)   <>   (3 + 2) * 7
```

2.3 Value-oriented programming

The way we run an OCaml program is to calculate it to a *value*, which is just the result of a computation. All of the constants of the primitive data types like `3`, `"hello"`, `true`, *etc.*, are values.

We write '$\langle exp \rangle \Longrightarrow \langle val \rangle$' to mean that the expression $\langle exp \rangle$ computes to the value $\langle val \rangle$.

Note: *The symbol '$\Longrightarrow$' is not part of OCaml's syntax; it is notation that we use to talk* about *the OCaml language.*

For example:

$$
\begin{array}{rcl}
\texttt{17} & \Longrightarrow & \texttt{17} \\
\texttt{2 + 3} & \Longrightarrow & \texttt{5} \\
\texttt{2 + (5 * 3)} & \Longrightarrow & \texttt{17} \\
\texttt{"answer is "\^{}(string_of_int 42)} & \Longrightarrow & \texttt{"answer is 42"} \\
\texttt{true || (false \&\& true)} & \Longrightarrow & \texttt{true} \\
\texttt{17 + "hello"} & \not\Longrightarrow & \textit{this doesn't compute!}
\end{array}
$$

The value of an expression is computed by calculating. For values, there is no more calculation left to do—they are done. For expressions built out of operations, we first calculate the values of the subexpressions and then apply the operation to those results.

The operations on the primitive types `int`, `string`, and `bool` do the "expected thing" in one step of calculation, which we indicate by using the notation '$\longmapsto$'. We can then calculate via a series of steps like so:

$$
\begin{array}{lll}
 & \texttt{(2 + 3) * (5 - 2)} & \\
\longmapsto & \texttt{5 * (5 - 2)} & \textit{because } \texttt{2 + 3} \longmapsto \texttt{5} \\
\longmapsto & \texttt{5 * 3} & \textit{because } \texttt{5 - 2} \longmapsto \texttt{3} \\
\longmapsto & \texttt{15} &
\end{array}
$$

We write `(2 + 3) * (5 - 2)` $\Longrightarrow$ `15` when the expression calculates to a value in an arbitrary number of steps.

Here is another example, this time using boolean values:

$$
\begin{array}{lll}
 & \texttt{true || (false \&\& true)} & \\
\longmapsto & \texttt{true || false} & \textit{because } \texttt{false \&\& true} \longmapsto \texttt{false} \\
\longmapsto & \texttt{true} & \textit{because } \texttt{true || false} \longmapsto \texttt{true}
\end{array}
$$

`if-then-else`

OCaml's `if-then-else` construct provides a useful way to choose between two different result values based on a boolean condition. Importantly, OCaml's `if` is itself an expression, which means that parentheses can be used to group them just like any other expression. Here are some simple examples of conditional expressions:

```
if true then 3 else 4
(if 3 > 4 then 100 else 0) + 17
```

Because OCaml is value oriented, the way to think about `if` is that it chooses one of two expression to evaluate. We run an `if` expression by first running the boolean test expression (which must evaluate to either `true` or `false`). The value computed by the whole `if-then-else` is then either the

value of the `then` branch or the value of the `else` branch, depending on whether the test was `true` or `false`.

For example, we have:

```
     (if (3 > 4) then 5 + 1 else 6 + 2) + 17
⟼   (if false then 5 + 1 else 6 + 2) + 17
⟼   (6 + 2) + 17          because the test was false
⟼   8 + 17
⟼   25
```

Note: *Unlike C and Java, in which* `if` *is a* statement *form rather than an expression form, it does not make sense in OCaml to leave off the* `else` *clause of an* `if` *expression. (What would such an expression evaluate to in the case that the test was* `false`*?) Moreover, since either of the expressions in the branches of the* `if` *might be selected, they must both be of the same type.*

Here are some erroneous uses of `if`:

```
if true then 3                    (* BAD: no else clause *)
if false then 3 else "hello"      (* BAD: branch types differ *)
```

Note that because `if-then-else` is an expression form, multiple `if-then-else`s can be nested together to create a "cascading" conditional:

```
if (3 > 4) then "abc"
else if (5 > 4) then "def"
else if (7 > 6) then "ghi"
else "klm"
```

This expression evaluates to `"def"`.

2.4 `let` declarations

A complete OCaml program consists of a sequence of top-level *declarations* and *commands*. Declarations define constants and functions; commands (described in §2.9) are used to generate output and test the behavior of those functions.

Constant declarations use the `let` keyword to *name* the result of an expression.

```
let x = 3
```

This declaration *binds* the identifier `x` to the number `3`. The name `x` is then available for use in the rest of the program. Informally, we sometimes call these identifiers "variables," but this usage is a bit confusing because, in languages like Java and C, a variable is something that can be modified over the course of a program. In OCaml, like in mathematics, once a variables value is determined, it can never be modified. . . As a reminder of this difference, for the purposes of OCaml we'll try to use the word "identifier" when talking about the name bound by a `let`.

A slightly more complex example is:

```
let y = 3 + 5 * 2
```

When this program is run, the identifier `y` will be bound to the result of evaluating the expression `3 + 5 * 2`. Since `3 + 5 * 2` $\Longrightarrow$ `13` we have:

`let y = 3 + 5 * 2` $\Longrightarrow$ `let y = 13`

The general form of a `let` declaration is:

`let` $\langle id \rangle$ = $\langle exp \rangle$

If desired, `let` declarations can also be annotated with the type of the expression. This type annotation provides documentation about the identifier, but can be inferred by the compiler if it is absent.

`let` $\langle id \rangle$: $\langle type \rangle$ = $\langle exp \rangle$

To run a sequence of `let` declarations, you evaluate the first expression and then *substitute* the resulting value for the identifier in the subsequent declarations. Substitution just means to replace the occurrences of the identifier with the value. For example, consider:

```
let x  = 3 + 2
let y  = x + x
let z  = x + y + 4
```

We first calculate `3 + 2` $\Longrightarrow$ `5` and then substitute `5` for `x` in the rest of the program:

```
let x  = 5
let y  = 5 + 5
let z  = 5 + y + 4
```

We then proceed with evaluating the next declaration (`5 + 5` $\Longrightarrow$ `10`) so we have:

```
let x  = 5
let y  = 10
let z  = 5 + 10 + 4
```

And finally, `5 + 10 + 4` $\Longrightarrow$ `19`, which leads to the final "fully simplified" program:

```
let x  = 5
let y  = 10
let z  = 19
```

Shadowing

What happens if we have two declarations binding the same identifier, as in the following example?

```
let x  = 5
let x  = 17
```

The right way to think about this is that there are unrelated declarations for *two* identifiers that both happened to be called `x`. In particular, we don't substitute the value `5` for the second `x` above—the `let` declaration needs an *identifier* and `5` is not an identifier. We say that `x` is *bound* by a `let`, and we do not substitute for such binding occurrences.

Now consider how to run this example:

```
let x  = 5
let y  = x + 1
let x  = 17
let z  = x + 1
```

The occurrence of `x` in the definition of `z` refers to the one bound to `17`. In OCaml identifiers refer to the *nearest enclosing declaration*. In these examples the second occurrence of the identifier `x` is said to *shadow* the first occurrence—subsequent uses of `x` will find the second binding, not the first. So the values computed by running this sequence of declarations is:

```
let x   = 5
let y   = 6        (* used the x on the line above *)
let x   = 17
let z   = 18       (* used the x on the line above *)
```

Since identifiers are just names for expressions, we can make the fact that the two `x`'s are independent by consistently renaming the latter one, for example:

```
let x   = 5
let y   = x + 1
let x2   = 17
let z   = x2 + 1
```

Then there is no possibility of ambiguity.

Note: *Identifiers in OCaml are* not *like variables in C or Java. In particular, once a value has been bound to a an identifier using* `let`*, the association between the name and the value never changes. There is no possibility of changing a variable once it is bound, although, as we saw above, one can introduce a new identifier whose definition shadows the old one. Identifiers are more like variables in math—they are placeholders for which values can be "plugged in."*

2.5 Local `let` declarations

So far we have seen that `let` can be used to introduce a top-level binding for an identifier. In OCaml, we can use `let` as an expression by following it with the `in` keyword. Consider this simple example:

```
let x   = 3 in   x + 1
```

Unlike the top-level `let` declarations we saw above, this is a *local* declaration. To run it, we first see that `3` $\Longrightarrow$ `3` and then we substitute `3` for `x` in the expression following the `in` keyword. Since this is a local use of `x`, we don't keep around the `let`. Thus we have:

`let x = 3 in x + 1` $\Longrightarrow$ `4`

More generally, the syntax for local lets is:

`let` $\langle id \rangle$ `:` $\langle type \rangle$ `=` $\langle exp \rangle$ `in` $\langle exp \rangle$

Here, the identifier bound by the `let` is in available only in the expression after the `in` keyword—when we run the whole thing, we substitute for the identifier only there (and not elsewhere in the program). The identifier `x` is said to be in *scope* in the expression after `in`.

Since the `let-in` form is itself an expression, it can appear anywhere an expression is expected. In particular, we can write sequences of `let-in` expressions, where what follows the `in` of one `let` is itself another `let`:

```
let price   = 500 in
let attendees  = 120 in
let revenue  = price * attendees in
let cost   = 18000 + 4 * attendees in
revenue - cost
```

Also, the expression on the right-hand side of a top-level `let` declaration can itself be a `let-in`:

```
(* this is a top level let - it has no 'in' *)
let profit   =
  let price   = 500 in            (* these lets are local *)
  let attendees  = 120 in
  let revenue  = price * attendees in
  let cost   = 18000 + 4 * attendees in
  revenue - cost
```

We follow the usual rules for computing the answer for this program: evaluate the right-hand-side of each `let` binding to a value and substitute it in its scope. This yields the following sequence of steps:

```
let profit   =
  let attendees  = 120 in
  let revenue  = 500 * attendees in  (* substituted price *)
  let cost   = 18000 + 4 * attendees in
  revenue - cost
```

$\longmapsto$

```
let profit   =
  let revenue  = 500 * 120 in     (* substituted attendees *)
  let cost   = 18000 + 4 * 120 in (* substituted attendees *)
  revenue - cost
```

$\longmapsto$

```
let profit   =
  let cost   = 18000 + 4 * 120 in
  60000 - cost                       (* substituted revenue *)
```

$\longmapsto$

```
let profit  =
  60000 - 18480                          (* substituted cost*)
```

$\longmapsto$

```
let profit  = 41520                      (* done! *)
```

Non-trivial scopes

It may seem like there is not much difference between redeclaring a variable via shadowing and the kind of assignment that updates the contents of a variable in a language like Java or C. The real difference becomes apparent when you use local `let` declarations to create non-trivial scoping for the shadowed identifiers.

Here's a simple example:

```
(* inner let shadows the outer one *)
let x  = 1 in
  x + (let x  = 20 in x + x) + x
```

Because identifiers refer the their *nearest enclosing* `let` *binding*, this program will calculate to a value like this:

```
    let x = 1 in x + (let x = 20 in x + x) + x
⟼  1 + (let x = 20 in x + x) + 1
⟼  1 + (20 + 20) + 1
⟼  1 + 40 + 1
⟼  41 + 1
⟼  42
```

Note that in the first step, we substitute `1` for the outer `x` in its scope, which doesn't include those occurrences of `x` that are shadowed by the inner `let`.

Local `let` definitions are particularly useful when used in combination with functions, as we'll see next.

2.6 Function Declarations

So far we have seen how to use `let` to name the results of intermediate computations. The `let` declarations we've seen so far introduce an identifier that stands for one particular expression. To do more, we need to be

able to define *functions*, which you can think of as *parameterized expressions*. Functions are useful for capturing similarity between related calculations.

In mathematical notation, you might write a function using notation like:

$$f(x) = x + 1$$

In OCaml, top-level function declarations are introduced using `let` notation, just as we saw above. The difference is that function identifiers take *parameters*. The function above is written in OCaml like this:

```
let f (x:int) : int =
  x + 1
```

Here, `f` is an identifier that names the function and `x` is the identifier for the function's input parameter. The notation `(x:int)` indicates that this input is supposed to be an integer. The subsequent type annotation indicates that `f` produces an integer value as its result.

Functions can have more than one parameter, for example here is a function that adds its two inputs:

```
let sum (x:int) (y:int) :int =
  x + y
```

Note: *Unlike Java or C, each argument to a function in OCaml is written in its own set of parentheses. In OCaml you write this*

```
let f (x:int) (y:int) : int = ...
```

rather than this:

```
let f (x:int, y:int) : int = ...   (* BAD *)
```

In general, the form of a top level function declaration is:

`let` ⟨*id*⟩ (⟨*id*⟩:⟨*type*⟩) ... (⟨*id*⟩:⟨*type*⟩) : ⟨*type*⟩ = ⟨*exp*⟩

Calling functions

To call a function in OCaml, you simply write it next to its arguments—this is called *function application*. For example, given the function `f` declared above, we can write the following expressions:

```
f 3
(f 4) + 17
(f 5) + (f 6)
f (5 + 6)
f (f (5 + 6))
```

As before, we use parentheses to group together a function with its arguments in case there is ambiguity about what order to do the calculations.

To run an expression given by a function call, you first run each of its arguments to obtain values and then calculate the value of the function's body after substituting the arguments for the function parameters. For example:

```
     f (5 + 6)
⟼   f 11
⟼   11 + 1        substitute 11 for x in x + 1 (the body of f)
⟼   12
```

Similarly we have:

```
     sum (f 3) (5 + 2)
⟼   sum (3 + 1) (5 + 2)    substitute 3 for x in x+1
⟼   sum 4 (5 + 2)
⟼   sum 4 7
⟼   4 + 7                  susbstitute 4 for x and 7 for y in x + y
⟼   11
```

Functions and scoping

Function bodies can of course refer to any identifiers in scope, including constants...

```
let one = 1

let increment (x:int) : int =
  x + one
```

... and other functions that have been defined earlier:

```
let double (x:int) : int =
  x + x

let quadruple (x:int) : int =
  double (double x)                (* uses double *)
```

Given the above declarations, we can calculate as follows:

```
      quadruple 2
⟼   double (double 2)     substitute 2 for x in the body of quadruple
⟼   double (2 + 2)        substitute 2 for x in the body of double
⟼   double 4
⟼   4 + 4                 substitute 4 for x in the body of double
⟼   8
```

Local `let` declarations are particularly useful inside of function bodies, allowing us to name the intermediate steps of a computation. For example:

```
let sum_of_squares (a:int) (b:int) (c:int) : int =
  let a2 = a * a in
  let b2 = b * b in
  let c2 = c * c in
  a2 + b2 + c2
```

2.7 Types

OCaml (like Java but unlike, for example C) is a *strongly typed* programming language. This means that the world of OCaml expressions is divided up into different types with strict rules about how they can be combined. We have already seen the primitive types `int`, `string`, and `bool`, and we will see many other types throughout the course.

Clearly, an integer like `3` has type `int`. We express this fact by writing "`3 : int`", or, more generally $\langle exp \rangle : \langle type \rangle$. We have already seen this notation in use in `let` declarations and in function parameters, where `(x : int)` means that `x` is a parameter of type `int`.

In a well-formed OCaml program, every expression has a type. Moreover, this type can be determined "compositionally" from the operations used in the expression. An expression is *well-typed* if it has at least one type. Here are some examples of well-typed expressions and their types:

```
                                   3  :  int
                              3 + 17  :  int
                             "hello"  :  string
                     string_of_int 3  :  string
if 3 > 4 then "hello"else "goodbye"  :  string
```

Before you run your OCaml program, the compiler will *typecheck* your program for you. This process makes sure that your use of expressions

in the program is consistent, which rules out many common errors that programmers make as they write their programs.

For example, the expression `3 + "hello"` is *ill typed* and will cause the OCaml compiler to complain that `"hello"` has type `string` but was expected to have type `int`. Such error checking is good, because there is no sensible way for this program to compute to a value—OCaml's type checking rules out mistakes of this form. It is better to reject this program while it is still under development than to allow it to be distributed to end users, where such mistakes can cause serious reliability problems.

All of the built in operators like `+`, `*`, `&&`, *etc.*. have types as you would expect. The arithmetic operators take and produce `int`'s; the logical operators take and produce `bool`'s. The comparison operators `=` and `<>` take two arguments of any type (they must both be of the *same* type) and return a `bool`. This means that, `3 = "hello"` will produce an error message, for example, while `3 = 4` and `"hello"<> "goodbye"` are both well typed.

The type annotations on user-defined functions tell you about the types of their inputs and result. For example, the function `quadruple` declared above expects an `int` input and produces an `int` output. We abbreviate this using the notation `int -> int`, the type of "functions that take one `int` as input and produce an `int` as output."

```
quadruple : int -> int
```

Because `double` also expects an `int` and produces an `int`, it also has the type `int -> int`.

On the other hand, the function `sum_of_squares` takes three arguments (each of type `int`) and produces an `int`. Its type is written as follows:

```
sum_of_squares : int -> int -> int -> int
```

Each of the functions in OCaml's libraries has a type, and these types can often give a strong hint about the behavior of the function. One example that we have seen so far is `string_of_int`, which has the type `int -> string`.

Common typechecking errors

OCaml's type checker uses these function types to make sure that your program is consistent. For example, the expression `double "hello"` is ill-

typed because `double` expects an `int` but the argument `"hello"` has type `string`. If you try to compile a program that contains such an ill-typed expression, the compiler will give you an error and point you to the offending expression of the program.

Another common error in OCaml programming is to omit one or more of the arguments to a function. Suppose you meant to increment a sum of squares $2^2 + 3^2 + 4^2$ by 1, and you wrote the expression below, accidentally leaving off the third argument:

```
(sum_of_squares 2 3) + 1     (* missing argument *)
```

In this case, OCaml will complain that the expression on the left of the `+` should be of type `int` but instead has type `int -> int`. That is because you have supplied two out of the three arguments to `sum_of_squares`; the resulting expression is still a function of the third parameter, that is, if you give the missing argument `4`, the resulting expression will have the expected type `int`. The OCaml error messages take some getting used to, but they can be very informative.

Conversely, another frequent mistake is to provide too many arguments to a function. For example, suppose you meant to perform the calculation described above, but accidentally left out the `+` operator. In that case, you have:

```
(sum_of_squares 2 3 4) 1     (* extra argument, missing + *)
```

OCaml interprets this as trying to pass an extra argument, but since the expression in parentheses isn't a function, the compiler will complain that you have tried to apply an expression that isn't of function type.

2.8 Failwith

OCaml provides a special expression, written `failwith "error string"`, that, when run, instead of calculating a value, immediately terminates the program execution and prints the associated error message. Such `failwith` expressions can appear *anywhere* that an expression can (but don't forget the error message string!).

Why is this useful? When developing a program it is often helpful to "stub out" unimplemented parts of the program that have yet to be filled

in. For example, suppose you know that you want to write two functions f and g and that g calls f. You might want to develop g *first*, and then go back to work on f later. failwith can be used as a placeholder for the unimplemented (parts of) f, as shown here:

```
let f (x:int) : int =
  if (x < 0)
    then (failwith "case x= 0 unimplemented")
    else x * 17 - 3

let g (y:int) : int =
  f (y * 10)
```

In CIS 120, the homework assignments use failwith to indicate which parts of the program should be completed by you. Don't forget to remove the failwith's from the program as you complete the assignment.

Note that using failwith at the top level is allowed, but will cause your program to terminate early, potentially without executing later parts of the code:

```
let x : int = failwith "some error message"

let y : int = x + x      (* This code is never reached *)
```

2.9 Commands

So far, we have seen how to use top-level declarations to write programs and functions that can perform calculations, but we haven't yet seen how to generate output or otherwise interact with the world outside of the OCaml program. OCaml programs can, of course, do I/O, graphics, write to files, and communicate over the network, but for the time being we will stick with three simple ways of interacting with the external world: printing to the screen, importing other OCaml modules and libraries, and running program test cases.

We call these actions *commands*. Commands differ from expressions or top-level declarations in that they don't calculate any useful value. Instead, they have some *side effect* on the state of the world. For example, the print_string command causes a string to be displayed on the terminal, but doesn't yield a value. Note that, because commands don't calculate

to any useful value, it doesn't make sense for them to appear within an expression of the program.

Commands (for now) may appear only at the top level of your OCaml programs. Syntactically, commands look like function call expressions, but we prefix them with `;;` to distinguish them from other function calls that yield values. This notation emphasizes the fact that commands don't do anything except interact with the external world.

Displaying Output

The simplest command is `print_string`, which causes its argument (which should have type `string`) to be printed on the terminal:

```
;; print_string "Hello, world!"
```

If you want your printed output to look nice, you probably want to include an "end-of-line" character, written `"\n"`, so that subsequent output begins on the next line.

```
;; print_string "Hello, world!\n"
```

OCaml also provides a command called `print_endline` that is just like `print_string` except it automatically supplies the `"\n"` at the end:

```
;; print_endline "Hello, world!"
```

There is also a command called `print_int`, which prints integers to the terminal:

```
;; print_int (3 + 17)          (* prints "20" *)
```

The arguments to commands can be expressions whose values depend on other identifiers that are in scope, which lets you print out the results of calculations:

```
let volume (x:int) (y:int) (z:int) : int =
  x * y * z

let side : int = 17

;; print_int (volume side side side)    (* prints "4913" *)
```

The function `string_of_int` and the string concatenation operator `^` are often useful when constructing messages to print. We might rewrite the example above to be more informative:

```
let volume (x:int) (y:int) (z:int) : int =
  x * y * z

let side : int = 17

let message : string = "The volume is: " ^
        (string_of_int (volume side side side))

;; print_endline message
```

Importing Modules: the `Assert` library

OCaml provides several libraries, and programmers can develop their own libraries of code. The command `open` takes a (capitalized) module name as an argument and imports the functions of that module so that they can be used in the subsequent program.

For the time being, the only external module we need to worry about is the one that provides the testing infrastructure for CIS 120 projects. This module is called `Assert`, and one of the commands it provides is the `run_test` command described next.

All of our homework projects and labs already include the following line, which ensures that `run_test` is available:

```
(* Import the CIS 120 testing infrastructure library *)
;; open Assert
```

The `run_test` Command

The last command needed for the homework assignments (at least for now), is `run_test`, which, as described above is provided by the `Assert` library.

The `run_test` command takes two arguments: (1) a `string` that serves as a label for the test and is used when printing the results of the test, and (2) a function identifier that names the test to be run.

Here is an example use of `run_test`:

```
let test () : bool =
  (1 + 2 + 3) = 7
;; run_test "1 + 2 + 3" test
```

The `test` function declared just above `run_test` takes *no* arguments, as indicated by the `()` parameter, and returns a `bool` result. A test succeeds if it returns the value **`true`** and fails otherwise (note that a test might fail by calling `failwith`, in which case no answer is actually returned).

The effect of the `run_test` command is to execute the `test` function and, if the test fails, print an error message to the terminal that indicates which test failed. In the example above, it is easy to see that this test fails, since `1 + 2 + 3` $\Longrightarrow$ `6`, which is not equal to `7`.

To run multiple tests in one program, we simply use shadowing so that we can re-use the identifier `test` for all the test functions. Here's an example:

```
let f (x:int) : int = (x + 1) / 2

(* This test succeeds *)
let test () : bool =
  (f 7) = 4
;; run_test "f 7" test


(* This test function shadows the one above; it fails *)
let test () : bool =
  (f 0) = 1
;; run_test "f 0" test
```

2.10 A complete example

Putting it all together, we can implement a program that solves the simple design problem posed in lecture.

```
(* Import the CIS 120 testing framework *)
;; open Assert


(* Using OCaml as a simple calculator *)
let profit_five_dollars : int =
  let price = 500 in
  let attendees = 120 in
  let revenue = price * attendees in
  let cost = 18000 + 4 * attendees in
  revenue - cost

(* Generate output useful for debugging, understanding *)
;; print_endline ("profit $5.00 = " ^
                    (string_of_int profit_five_dollars))

(* Assumes a linear relationship between attendees and price *)
let attendees (price:int) : int =
        (-15 * price) / 10 + 870

(* Tests for attendees, generated from the problem description *)
let test () : bool =
        (attendees 500) = 120
;; run_test "attendees @ $5.00" test

let test () : bool =
        (attendees 490) = 135
;; run_test "attendees @ $4.90" test


let revenue (price:int) : int =
        price * (attendees price)

let cost (price:int) : int =
        18000 + 4 * (attendees price)

let profit (price : int) : int =
        (revenue price) - (cost price)


(* A test case generated from the problem description *)
let test () : bool =
        (profit 500) = profit_five_dollars
;; run_test "profit $5.00" test

let test () : bool =
        (profit 490) = 47610
;; run_test "profit $4.90" test
```

2.11 Notes

Parts of this Chapter were adapted from lecture notes by Dan Licata written for CMU's course 15-150 Fall 2011, Lecture 2.

Chapter 3

Lists and Recursion

So far we have seen how to create OCaml programs that compute with *atomic* or *primitive* values—integers, strings, and booleans. A few real world problems can be solved with this just these abstractions, but most problems require computing with collections of data—sets, lists, tables, or databases.

In this chapter, we'll study one of the simplest forms of collections: *lists*, which are just ordered sequences of data values. Many of you will have seen list data structures from your previous experience with programming. OCaml's approach to list processing emphasizes that computing with lists closely follows the *structure* of the datatype. This will be a recurring theme throughout the course: the structure of the data tells you how you should compute with it.

3.1 Lists

What is a list? It's just a sequence of zero or more values. That is, a list is either

`[]`	the *empty* list, sometimes called *nil*, or
`v::tail`	a value `v` followed by `tail`, a list of the remaining elements.

There is no other way of creating a list: all lists have one of these two forms. The double-colon operator '`::`' *constructs* a non-empty list—it takes as inputs the head of the new list, `v`, and a (previously constructed) list, `tail`. This operator is sometimes pronounced "cons" (short for "construc-

tor"). Note that `::` is just a binary operator like `+` or `^`, but, unlike those, it happens to take arguments of two different types—its left argument is an element and its right argument is a *list* of elements.

Here are some examples of lists built from `[]` and `::`:

```
[]
3::[]
1::2::3::[]
true::false::true::[]
"do"::"re"::"mi"::"fa"::"so"::"la"::"ti"::[]
```

The examples above can also be written more explicitly using parentheses to show how `::` associates. For example, we can equivalently write `1::(2::(3::[]))` instead of `1::2::3::[]`. We usually leave out these parentheses because they aren't needed: the OCaml compiler fills them in automatically. Note that if you were to write `(1::2)::3::[]` you will get a type error—`2` by itself isn't a list.

You can, however have lists of lists—i.e., the elements of the outer list can themselves be lists:

```
(1::2::[])::(3::4::[])::[]
```

Writing out long lists using `::` over and over can get a little tedious, so OCaml provides some syntactic sugar to make things shorter. You can write a list of values by enclosing the elements in `[` and `]` brackets and separating them with semicolons. Rewriting some of the examples we have seen so far using this more convenient notation, we have:

```
                   3::[]  =  [3]
             1::2::3::[]  =  [1;2;3]
   true::false::true::[]  =  [true;false;true]
(1::2::[])::(3::4::[])::[]  =  [[1;2];[3;4]]
```

Calculating With Lists

As with the primitive operators discussed in §2, the arguments to `::` can themselves be complex expressions. To simplify such an expression, we calculate the head element down to a value and then calculate the tail list down to a value. A list is a value just when all of its elements are values.

For example, we might calculate like this:

```
      (1 + 2)::((2 + 3)::[])
⟼     3::(2+3)::[]            because 1+2⟼ 3
⟼     3::5::[]                because 2+3⟼ 5
```

And then we're done because the entire list consists of values. Equivalently, we could have used the more compact notation:

```
      [(1 + 2);((2 + 3)])
⟼     [3;(2+3)]            because 1+2⟼ 3
⟼     [3;5]                because 2+3⟼ 5
```

All of the rules for calculation that we've already seen carry through here as well. We can use `let` to bind list expressions and name intermediate computations, and we use substitution just as before:

```
let square (x:int) : int = x * x
let l : int list = [square 1; square 2; square 3]
let k : int list = 0::l
```

⟹

```
let square (x:int) : int = x * x
let l : int list = [1;4;9]
let k : int list = [0;1;4;9]
```

List types

The example just above uses the type annotation `l : int list` to indicate that `l` is a list of `int` values. Similarly, we can have a list of strings whose type is `string list` or a list of `bool list`s, whose type is `(bool list) list`.

All of the elements of a list must be of the *same* type: OCaml supports only "homogeneous" lists. If you try to create a list of mixed element types, such as `3::"hello"::[]`, you will get a type error. While this may seem limiting, it is actually quite useful; knowing that you have a list of `int`s, for example, means that you can perform arithmetic operations on all of those elements without worrying about what might happen if some non-integer element is encountered.

Simple pattern matching on lists

We have seen how to construct lists by using `::`. How do we take a list value apart? Well, there are only two cases to consider: either a list is

empty, or it has some head element followed by some tail list. In OCaml we can express this kind of case analysis using *pattern matching*. Here is a simple example that uses case analysis to determine whether the list `l` is empty:

```
let l : int list = [1;2;3]

let is_l_empty : bool =
   begin match l with
     | []     -> true
     | x::tl -> false
   end
```

The `begin match` ⟨*exp*⟩ `with ... end` expression compares the shape of the value computed by ⟨*exp*⟩ against a series of pattern cases. In this example, the first case is `| [] -> true`, which says that, if the list being analyzed is empty, then the result of the `match` should be `true`.

The second case is `| x::tl -> false`, which specifies what happens if the the list being analyzed matches the pattern `x::tl`. If that happens, then the expression to the right of the `->` in this branch will be the result of the whole `match`.

Note that the *patterns* `[]` and `x::tl` look exactly like the two possible ways of constructing a list! This is no accident: since all lists are built from these constructors, these are exactly the cases that we have to handle—no more, and no less. The difference between the pattern `x::tl` and a list value is that in the pattern, `x` and `tl` are *identifiers* (i.e. place holders), similar to the identifiers bound by `let`. When the `match` is evaluated, they are *bound* to the corresponding components of the list, so that they are available for use in the expression following the pattern's `->`.

To see how this works, let's look at some examples. First, let's see how to evaluate the example above, which doesn't actually use the identifiers bound by the pattern:

```
let l : int list = [1;2;3]
let is_l_empty : bool =
   begin match l with
     | []     -> true
     | x::tl -> false
   end
```

$\longmapsto$ *(by substituting* `[1;2;3]` *for* `l`*)*

```
let l : int list = [1;2;3]
let is_l_empty : bool =
   begin match [1;2;3] with
     | []     -> true
     | x::tl -> false
   end
```

At this point, we match the list `[1;2;3]` against the first pattern, which is `[]`. Since they don't agree—`[]` doesn't have the same shape as `[1;2;3]`—we continue to the next case of the pattern. Now we try to match the pattern `x::tl` against `[1;2;3]`. Remember that `[1;2;3]` is just syntactic sugar for `1::2::3::[]`, which, if we parenthesize explicitly, is just `1::(2::(3::[]))`. This value *does* match the pattern `x::tl`, letting `x` be `1` and `tl` be `2::3::[]`. Therefore, the result of this pattern match is the right-hand side of the `->`, which in this example is just **`false`**. The entire program thus steps to:

```
let l : int list = [1;2;3]
let is_l_empty : bool = false
```

In more interesting examples, the identifiers like `x` and `tl` that are used in a cons-pattern will also appear in the right-hand side of the match case. For example, here is a function that returns the square of the first element of an `int list`, or `-1` if the list is empty:

```
let square_head (l:int list) : int =
  begin match l with
    | []      -> -1
    | x::tl   -> x * x
  end
```

Let's see how a call to this function evaluates:

```
square_head [3;2;1]
```

⟼ *(by substituting* `[3;2;1]` *for* `l` *in the body of* `square_head`*)*

```
  begin match [3;2;1] with
    | []     -> -1
    | x::tl -> x * x
  end
```

Now we test the branches in order. The first branch's pattern (`[]`) doesn't match, but the second branch's pattern does, with `x` bound to `3` and `tl` bound to `[2;1]`, so we substitute `3` for `x` and `[2;1]` for `tl` in the

right-hand side of that branch, i.e.:

```
begin match [3;2;1] with
  | []    -> -1
  | x::tl -> x * x
end
```

⟼ *(substitute* `3` *for* `x` *and* `[2;1]` *for* `tl`*) in the second branch*

```
3 * 3
```

⟼

```
9
```

3.2 Recursion

The definition of lists that we gave at the start of this chapter said that a list is either

`[]`	the *empty* list, sometimes called *nil*, or
`v::tail`	a value `v` followed by `tail`, a list of the remaining elements.

Note that this description of lists is *self referential*: we're defining what lists are, but the second clause uses the word "list"! How can such a "definition" make sense? The trick is that the above definition tells us how to build a list of length 0 (i.e. the empty list `[]`), or, given a list of length n, how to construct a list of length $n + 1$. The phrase "list of the remaining elements" is really only talking about strictly *shorter* lists.

This is the hallmark of an *inductive data type*: such data types tell you how to build "atomic" instances (here, `[]`) and, given some smaller values, how to construct a bigger value out of them. Here, `::` tells us how to construct a bigger list out of a new head element and a shorter tail.

This pattern in the structure of the data also suggests how we should compute over it: using *structural recursion*. Structural recursion is a fundamental concept in computer science, and we will see many instances of it throughout the rest of the course.

The basic idea is simple: to create a function that works for *all* lists, it is sufficient to say what that function should compute for the empty list, and, *assuming that you already have the value of the function for the tail of the list*, how to compute the function given the value at the head. Like the definition of the list data type, functions that compute via structural

recursion are themselves self referential—the function is defined in terms of itself, but only on smaller inputs!

Calculating the length of a list

Here is an example of how to use structural recursion to compute the length of a list of integers:

```
let rec length (l:int list) : int =
  begin match l with
    | [] -> 0
    | x::tl -> 1 + (length tl)   (* Nb: length is used here *)
  end
```

Note that we use the keyword `rec` to explicitly mark this function as being recursive. If we omit the '`rec`', OCaml will complain if we try to use the function identifier inside its own body. Also, note that this function does case analysis on the structure of `l` and that the body of the second branch calls `length` recursively on the tail—this is an example of structural recursion, since the tail is always smaller than the original list.

The first branch tells us how to compute the length of the empty list, which is just 0. The second branch tells us how to compute the length of a list whose head is `x` and whose tail is `tl`. Assuming that we can calculate the length of `tl` (which is the result of `length tl` obtained by the recursive call), we can compute the length of the whole list by adding one.

Let us see how these calculations play out when we run the program:

```
length [1;2;3]
```

$\longmapsto$ *(substitute* `[1;2;3]` *for* `l` *in the body of* `length`*)*

```
begin match [1;2;3] with
  | [] -> 0
  | x::tl -> 1 + (length tl)
end
```

$\longmapsto$ *(the second branch matches with* `tl=[2;3]`*)*

```
1 + (length [2;3])
```

$\longmapsto$ *(substitute* `[2;3]` *for* `l` *in the body of* `length`*)*

```
1 + (begin match [2;3] with
       | [] -> 0
       | x::tl -> 1 + (length tl)
     end)
```

⟼ *(the second branch matches with* `tl=[3]`*)*

```
1 + (1 + (length [3]))
```

⟼ *(substitute* `[3]` *for* `l` *in the body of* `length`*)*

```
1 + (1 + (begin match [3] with
            | [] -> 0
            | x::tl -> 1 + (length tl)
          end))
```

⟼ *(the second branch matches with* `tl=[]`*)*

```
1 + (1 + (1 + length []))
```

⟼ *(substitute* `[]` *for* `l` *in the body of* `length`*)*

```
1 + (1 + (1 + (begin match [] with
                 | [] -> 0
                 | x::tl -> 1 + (length tl)
               end))
```

⟼ *(the first branch matches)*

```
1 + (1 + (1 + (0)))
```

⟹

```
3
```

More `int list` examples

By slightly modifying the `length` function from above, we can obtain a function that sums all of the integers in a list.

Again we apply the design pattern for recursive functions. The sum of all elements of the empty list is just `0`, since there are no elements. If `sum tail` is the sum of all the elements in the tail of a list whose first element is `n`, then we obtain the total sum by simply calculating `n + (sum tail)`.

Here's the code, along with a couple test cases:

```
let rec sum (l:int list) : int =
  begin match l with
    | [] -> 0
    | n::tail -> n + (sum tail)
  end

let test () : bool =
   (sum []) = 0
;; run_test "sum []" test

let test () : bool =
   (sum [1;2;3]) = 6
;; run_test "sum [1;2;3]" test
```

Now let's look at a slightly more complex example. Suppose we want to write a function that takes a list of integers and filters out all the even integers—that is, our function should return a list that contains just the even elements of the original list.

Filtering the even numbers out of the empty list yields the empty list. And if `filter_out_evens tail` is the result of filtering the tail of the original list, we just need to check whether the head of the list is even to know whether it belongs in the output:

```
let rec filter_out_evens (l:int list) : int list =
  begin match l with
    | [] -> []
    | x::tail ->
        let filtered_tail = filter_out_evens tail in
        if (x mod 2) = 0
        then filtered_tail     (* x not included, it's even *)
        else x::filtered_tail (* x is included, it's odd *)
  end

let test () : bool =
   (filter_out_evens []) = []
;; run_test "filter_out_evens []" test

let test () : bool =
   (filter_out_evens [1;2;3]) = [1;3]
;; run_test "filter_out_evens [1;2;3]" test
```

For a third example, suppose we want to write a function that determines whether a list *contains* a given integer. The solution again uses structural recursion over the list parameter, but also takes as input the `int` to

search for:

```
let rec contains (l:int list) (x:int) : bool =
  begin match l with
    | [] -> false
    | y::tail -> x = y || contains tail x
  end

let test () : bool =
   (contains [1;2;3] 1) = true
;; run_test "contains [1;2;3] 1" test

let test () : bool =
   (contains [1;2;3] 4) = false
;; run_test "contains [1;2;3] 4" test
```

Finally, suppose we want to compute the list of *all suffixes* of a given list, that is:

`suffixes [1;2;3]` $\Longrightarrow$ `[[1;2;3]; [2;3]; [3]; []]`

We can easily compute this by observing that at every call to `suffixes` we simply need to cons the list itself onto the list of its own suffixes. Note that this observation holds true even for the empty list!

```
let rec suffixes (l:int list) : int list list =
  begin match l with
    | [] -> [[]]                          (* [[]] = []::[] *)
    | x::tl -> l :: (suffixes tl)
  end

let test () : bool =
   (suffixes [1;2;3]) = [[1;2;3]; [2;3]; [3]; []]
;; run_test "suffixes [1;2;3]" test
```

The `list` structural recursion pattern

All of the example list functions we have seen so far follow the same pattern. To define a function `f` over a list, we to case analysis to determine whether the list is empty or not. If it is empty, we can calculate the result of `f []` directly, without any recursive calls to `f`. If the list is *not* empty, we compute the answer for a list whose head is `hd` using the result of the recursive call `(f rest)`. In code:

```
let rec f (l : ... list) ... : ... =
  begin match l with
    | [] -> ...
    | (hd :: rest) -> ... hd ... (f rest) ...
  end
```

This pattern is extremely general: many, many useful functions can be written easily by following this recipe. Here is a (very partial) list of such functions, with brief descriptions. It is a good exercise to figure out how to implement all these functions using structural recursion.

- `all_even` — determines whether all elements of an `int list` are even
- `append` — takes two lists and "glues them together", for example `append [1;2;3] [4;5]` $\Longrightarrow$ `[1;2;3;4;5]`
- `intersection` — takes two lists and returns the list of all those elements that appear in both lists; for example: `intersection [1;2;2;3;4;4;3] [2;3]` $\Longrightarrow$ `[2;2;3;3]`
- `every_other` — computes the list obtained by dropping every second element from the input list, for example: `every_other [1;2;3;4;5]` $\Longrightarrow$ `[1;3;5]`

When designing tests for list processing functions, you should *always* consider at least two cases: a test case for the empty list, and a case for non-empty lists. Other kinds of test cases to consider might have to do with whether the list is even or odd, or whether the function has special behavior when the list has just one element.

Note: *For those of you who have taken or are taking CIS 160, it is also worth noticing that writing a program by structural induction over lists follows* the same pattern *as proving a property by* induction *on the list. There is a* base case *(i.e. the case for* `[]`*) and an* inductive case *(i.e. the recursive case). It is no accident that lists are called* inductive *data structures.*

Using helper functions

Sometimes the list function we are trying to implement isn't trivial to compute, even given the value of the recursive call. An example of such a

function is `prefixes`, which is like the `suffixes` example above, but instead calculates the list of prefixes of the input list. For example, we want to have:

`prefixes [1;2;3]` $\Longrightarrow$ `[[]; [1]; [1;2]; [1;2;3]]`

Why is this not so direct to implement as `suffixes` was? Suppose we're trying to calculate `prefixes [1;2;3]`. Following the structural recursion pattern, we would expect to call `prefixes` on the tail of this list, which is:

`prefixes [2;3]` $\Longrightarrow$ `[[]; [2]; [2;3]]`

Given this result for the recursive call to prefixes, and the head `1`, we need to compute `prefixes` for the entire list `[1;2;3]`. Looking for a pattern, it seems that we need to put `1` at the head of *each* of the prefixes of `[2;3]`. Since this functionality isn't given to us directly by prefixes, we need to define an auxiliary function to help out. Let's call it `prepend`. As usual, we document our examples as test cases.

```
(* put x on the front of each list in l *)
let rec prepend (l:int list list) (x:int) : int list list =
  begin match l with
    | [] -> []
    | ll:rest -> (x::ll)::(prepend rest x)
  end

let test () : bool =
   (prepend [[]; [2]; [2;3]] 1) = [[1]; [1;2]; [1;2;3]]
;; run_test "prepend "[[]; [2]; [2;3]]" test
```

Now, using `prepend`, it becomes much simpler to define `prefixes`:

```
let rec prefixes (l:int list) : int list list =
  begin match l with
    | [] -> [[]]
    | h::tl -> []::(prepend (prefixes tl) h)
  end

let test () : bool =
   (prefixes [1;2;3]) = [[]; [1]; [1;2]; [1;2;3]]
;; run_test "prefixes [1;2;3]" test
```

For another example where a helper function is needed, consider trying to compute a function `rotations` that, given a list of integers, computes the list of all "circular rotations" of that list. For example:

`rotations [1;2;3;4]` $\Longrightarrow$ `[[1;2;3;4]; [2;3;4;1]; [3;4;1;2]; [4;1;2;3]]`.

Here again, thinking about how `rotations` would work on the result of

a recursive call illustrates why there might be a need for a helper function. When we use `rotations` on the tail of the list `[1;2;3;4]`, we get:

`rotations [2;3;4]` ⟹ `[2;3;4]; [3;4;2]; [4;2;3]`

The relationship between `rotations [2;3;4]` and `rotations [1;2;3;4]` seems very nontrivial—it seems like we have to intersperse the head `1` at various points (in the last spot, the second-to-last spot, the third-to-last spot, *etc.*) in the prefixes of the tail. We could perhaps write an auxiliary function that does this, but that still seems complicated.

Here's another approach that simplifies the solution. Consider what information from the original list that we have at each recursive call to some function `f`

```
f [1;2;3;4]                   (* top-level call *)
f [2;3;4]                     (* first recursive call *)
f [3;4]                       (* second recursive call *)
f [4]                         (* third recursive call *)
f []                          (* last call *)
```

If each such recursive call to `f` were to calculate one of the rotations of the original list, what information would `f` be missing? Each call is missing a prefix of the original list!

```
                   (*missing prefix*)
f [1;2;3;4]        []
f [2;3;4]          [1]
f [3;4]            [1;2]
f [4]              [1;2;3]
f []               [1;2;3;4]
```

If we append the two lists in each row above, we obtain one of the desired permutations of the *original* list. (One of the permutations, `[1;2;3;4]`, appears twice—we'll have to be careful to leave one of them out of the result.)

This suggests that we can solve the rotations problem by creating a helper function that takes an extra parameter—namely, the missing prefix needed to compute one permutation of the original list.

If we were to accumulate these lists together into a list of lists, we would be done. That is, we want a function `f` that works like:

`f [1;2;3;4] []` ⟹ `[[1;2;3;4]; [2;3;4;1]; [3;4;1;2]; [4;1;2;3]]`

What should `f` do on a tail of the original, assuming we provide the missing prefix?:

`f [2;3;4] [1]` ⟹ `[[2;3;4;1]; [3;4;1;2]; [4;1;2;3]]`

Do you see the pattern? Each call to the helper function `f` computes some (but not all) of the rotations of the original list. Which ones? The ones beginning with each of the elements of the first input to `f`. Observe that at each call `f suffix prefix` we have that `prefix @ suffix` is the *original* list and `suffix @ prefix` is the next rotation to generate.

Function `f` should therefore simply recurse down the suffix list; but at each call it needs to rotate its own head element to the end of the `prefix` list, to maintain the relationship described above.

We already know how to put an element at the *front* of a list—we use `::`. How do we put one at the tail? Well, we simply write a second helper function that does the job. We call it `snoc` (because it is the opposite of `cons`):

Putting it all together, we can write rotations like so:

```
(* Add x to the end of the list l *)
let rec snoc (l:int list) (x:int) : int list =
  begin match l with
    | [] -> [x]
    | h::tl -> h::(snoc tl x)
  end

(* Compute some rotations of the list suffix @ prefix,
   where prefix @ suffix is the "original" list *)
let rec f (suffix:int list) (prefix:int list) : int list list
  begin match suffix with
    | [] -> []
    | x::xs -> (suffix@prefix)::(f xs (snoc prefix x))
  end

(* The top-level rotations function simply calls f with the
   empty prefix *)
let rotations (l:int list) = f l []
```

Infinite loops: Non-structural recursion

All of the examples list processing functions described above are guaranteed to terminate. Why? Because every list is contains only finitely many elements and each of the recursive calls in a structurally recursive function is invoked on a *strictly shorter* list (the tail). This means that at some point the chain of recursion will bottom out at the empty list.

OCaml does not strictly enforce that all list functions be structurally recursive. Consider the following example, in which the `loop` function calls itself "recursively" on the *entire* list, rather than just on the tail:

```
let loop (l:int list) : int =
  begin match l with
    | [] -> 0
    | h::tl ->  1 + (loop l)
  end
```

Watch what happens when we run this program on the list `[1]`:

```
loop [1]
```

⟼ *(substitute* `[1]` *for l in the body of loop)*

```
  begin match [1] with
    | [] -> 0
    | h::tl ->  1 + (loop l)
  end
```

⟼ *(the second case matches)*

```
1 + (loop l)      (* uh oh! *)
```

⟼ ...⟼

```
1 + (1 + (1 + (1 + (loop l))))
```

⟼ *(the program keeps running forever)*

If you find one of your programs mysteriously looping, a non-structural recursive call may be the culprit.

Chapter 4

Tuples and Nested Patterns

4.1 Tuples

Lists are good data structures for storing arbitrarily long sequences of homogeneous data, but sometimes it is convenient to bundle together two or more values of *different* types. OCaml provides *tuples* for this purpose. A tuple is a fixed-length sequence of values, enclosed in parentheses and separated by commas. Here are some examples:

```
(1, "uno")              (* a pair of an int and a string *)
(true, false, true)     (* a triple of bools *)
("a", 13, "b", false)   (* a quadruple *)
```

Note that, unlike lists, the elements of a tuple need not have the same types.

We write tuple types using infix `*` notation, so for the three examples above, we have:

```
(1, "uno")              : int * string
(true, false, true)     : bool * bool * bool
("a", 13, "b", false)   : string * int * string * bool
```

Of course, the elements of a tuple expression can themselves be complex expressions (including lists). OCaml evaluates a tuple expression by evaluating each of the components of the tuple. For example, `(1+2+3, true || false)` $\Longrightarrow$ `(6, true)` because `1+2+3` $\Longrightarrow$ `6` and `true || false` $\Longrightarrow$ `true`.

Lists and tuples can easily be combined to create more interesting data

structures. Here are a couple of examples and their types:

```
[(1, "uno"); (2, "dos"); (3, "tres")]
     : (int * string) list
([1; 2; 3], ["uno"; "dos"; "tres"])
     : (int list) * (string list)
```

Pattern matching on tuples

Like lists, tuples can be "inspected" by pattern matching. Also as for lists, the patterns themselves mimic the form of tuple values. For example:

```
let first (x:int * string) : int =
  begin match x with
    | (left, right) -> left
  end

let second (x:int * string) : string =
  begin match x with
    | (left, right) -> right
  end
```

Note that, since there's only one way to construct a tuple (using parens and commas), we only need *one* case when pattern matching against a tuple.

A more lightweight way to give names to the components of a tuple is to use a generalization of `let` binding that we have already seen. The idea is that, since there is only ever one case when matching against a tuple, we can abbreviate that case using the syntax `let (x,y) = ...`. For example, we can rewrite `first` and `second` above using `let`-tuple notation like this:

```
let first (x:int * string) : int =
  let (left, right) = x in left

let second (x:int*string) : string =
 let (left, right) = x in right
```

We omit the type annotation from the let binding because OCaml can infer it from the type of the tuple to the right of the `=` (e.g. by looking at the type of `x` above, it is easy to figure out that `left` should have type `int` and `right` should have type `string`).

The "empty" tuple, unit:

There's one special case of tuples to explain—the tuple of *no* elements. This is written in OCaml as `()`, and its type is called `unit`. There is only one value, namely `()`, of type `unit`, so not much information is conveyed by a unit value.

We have already seen `()` mentioned in our `test` functions: since they take no meaningful inputs, they have the type `unit -> bool`. We could write our test functions like

```
let test (x:unit) : bool =
  ...
```

(where we give an explicit name, `x`, to the `unit` parameter), but there is not much point in doing this, since the parameter is not used in the function's body. So we write `()` instead of `(x:unit)`, emphasizes that the parameter is trivial.

The other place in OCaml where `unit` is often used is as the return type of *commands* such as `print_string`. For example, `print_string` takes a `string` input and produces a `unit` output: its type is `string -> unit`. (Here, `unit` functioning like a `void` return type in C or Java.) Since there is only one value of type unit, such functions cannot return meaningful results: they are executed purely for their side effects on the machine's state (such as printing to the terminal). Whenever you see an OCaml function that return `unit`, you should take it as a strong hint that there will be some I/O or some kind of other effect when you call that function.

See §12 for a more in-depth discussion about the uses of the `unit` type.

4.2 Nested patterns

We saw in Chapter 3 that lists can be inspected by pattern matching. Lists values have two constructors, `[]`, and `::`, and list patterns analogously have two possible forms. Tuples, as we saw just above, are matched using `(..., ..., ...)` patterns. Furthermore, just as we can build complicated list and tuple *expressions* by nesting constructors, we can build complex list *patterns* by nesting too. Here are some examples of nested patterns:

```
[]                (* match the empty list *)
x::tail           (* match a non-empty list with head x *)
x::[]             (* match a list of exactly length 1 *)
x::y::tail        (* match a list of at least length 2 *)
[x;y]             (* match a list of exactly length 2 *)
x::y::[]          (* another way of writing the same *)
x::y::z::tail     (* match a list of at least length 3 *)
[x]::tail         (* match a list of lists, whose first element
                     is a singleton list *)
(x::[])::tail     (* another way of writing the same *)
(l,r)::tl         (* match a list of pairs whose head is a tuple
                     with two components l and r *)
(x::xs, y::ys)    (* match a pair of non-empty lists *)
([], [])          (* match a pair of empty lists *)
```

As an example of how one might use nested patterns, the following `match` expression has three cases, one for when the list is empty, one for singleton lists, and one for when a list has two or more elements:

```
begin match l with
  | [] -> ...       (* case for empty *)
  | x::[] -> ...    (* case for singleton *)
  | x::tail -> ...  (* case for two or more *)
end
```

When OCaml processes a match expression, it checks the value being matched against each pattern *in the order they appear*. In the example above, the case for singletons must appear *before* the case for two or more, since the pattern `x::tail` can match a singleton list like `1::[]` by binding `tail` to `[]`. This means that if we were to reverse the order of the cases, then the third branch would never be reached:

```
begin match l with
  | [] -> ...
  | x::tail -> ...
  | x::[] -> ...    (* this case is never reached! *)
end
```

If your cases are written so that one of them can never be executed, OCaml will warn you that the match case is unused. This is a good sign that there is a bug in your program logic!

4.3 Exhaustiveness

OCaml can determine whether you have covered all the possible cases when pattern matching against a datatype. For example, suppose you wrote the following:

```
begin match l with
  | [] -> ...      (* case for empty *)
  | x::[] -> ... (* case for singleton *)
end
```

This match expression only handles lists of length 0 and 1. If `l` happens to evaluate to a longer list, then at run-time the program will terminate and OCaml will report a `Match_failure` error. To prevent such surprises at run time, OCaml will give you a helpful warning that you have missed a possible case ("this pattern-matching is not exhaustive") when you compile the program. Besides the warning, the compiler will provide an example that isn't handled by your branches.

4.4 Wildcard (underscore) patterns

Sometimes the particular value in a pattern doesn't matter. For example, recall that we wrote a `length` function like this:

```
let rec length (l:int list) : int =
  begin match l with
    | [] -> 0
    | x::tl -> 1 + (length tl)    (* x not used *)
  end
```

Note that the second branch doesn't use `x` anywhere. We can emphasize that fact by using the special '_' pattern, which matches *any* value but doesn't give a name for it. So, for example, we could write:

```
let rec length (l:int list) : int =
  begin match l with
    | [] -> 0
    | _::tl -> 1 + (length tl)
  end
```

A wildcard pattern can appear anywhere in a pattern. Here are some examples:

```
_        (* matches anything *)
x::_     (* matches any non-empty list; names the head *)
_::_     (* matches a non-empty list without naming
            its parts *)
```

4.5 Examples

Here's an example function that uses lists, tuples, and nested pattern matching. The function is called `zip`; it takes as inputs two lists, and it returns a list of pairs, drawn "in lockstep" from the two lists. For example, we have:

```
zip [1;2;3] ["uno"; "dos", "tres"]
```

$\Longrightarrow$ `[(1, "uno"); (2, "dos"); (3, "tres")]`

```
let rec zip (l1:int list) (l2:string list)
           : (int * string) list =
  begin match (l1, l2) with   (* note the tuple here! *)
    | ([], []) -> []
    | (x::xs, y::ys) -> (x,y)::(zip xs ys)
    | _ -> failwith "zip called on unequal-length lists"
  end
```

(The last case is needed for exhaustive pattern matching.)

It is instructive to see how the same function would be written without the use of nested patterns or wildcards. It is considerably more verbose!

```
(* zip without using nested patterns *)
let rec zipX (l1:int list) (l2:string list)
            : (int * string) list =
  begin match l1 with
    | [] -> begin match l2 with
              | [] -> []
              | y::ys ->
                  failwith "zipX: unequal length lists"
            end
    | x::xs -> begin match l2 with
                 | [] -> failwith "zipX: unequal length lists"
                 | y::ys -> (x,y)::(zipX xs ys)
               end
  end
```

Chapter 5

User-defined Datatypes

5.1 Atomic datatypes: enumerations

We have seen that OCaml provides several means of creating and manipulating structured data. Our toolbox includes primitive datatypes `int`, `string`, and `bool`, immutable lists, and tuples, as well as the ability to write (potentially recursive) functions over such data.

Despite these many options, programs often need application-specific datatypes. For example, suppose you were building a calendar application, you might want to have a datatype for representing the days of the week. Or, if your program involved computing over DNA sequences, you might need a representation for the nucleotides (adenine, guanine, thymine, and cytosine) that make up a DNA strand.

One possibility would be to use the existing datatypes available to represent your program-specific data. For example, you might choose to use `int` values to represent days of the week, with some mapping like:

```
 Sunday  =  0
 Monday  =  1
Tuesday  =  2
         ...
```

This could work[1], but it's not a very good idea for a few reasons. First, `int` values and days of the week support different operations—it makes

[1] It's essentially what you do in a weakly-typed language like C

sense to subtract `1 - 2` to obtain the answer `-1`, but what would the subtraction `Tuesday - Wednesday` mean? There is no day of the week that is represented by `-1`. This may seem relatively innocuous, but it's a recipe for disaster. Consider writing a function that converts days of the week (represented as `int`s) into strings:

```
let string_of_weekday (wd:int) : string =
  if wd = 0 then "Sunday"
  else if wd = 1 then "Monday"
  else if wd = 2 then "Tuesday"
  ...
  else if wd = 6 then "Saturday"
  else failwith "not a valid weekday"
```

Everywhere you use such an encoding, you have to check to make sure (as in the last `else` clause above) to handle the case of an *invalid* encoding. Forgetting to check for such cases will lead to bugs and other unexpected failures in your code.

Another reason why such encodings are a bad idea is that it's also possible to accidentally confuse types—if you encode both weekdays and nucleotides as `int` values, it would be possible to pass an integer representing a nucleotide to the `string_of_weekday` function, which will likely yield nonsensical behavior.

For all of the reasons above, modern type-safe programming languages, including OCaml, Java, C#, Scala, Haskell, and others allow programmers to create *user-defined datatypes*, which effectively extend the programming language with new abstractions that can be manipulated just like any other values.

In OCaml, such datatypes are declared using the `type` keyword. For example, here is how the datatype for days-of-the-week can be defined:

```
type day =
  | Sunday
  | Monday
  | Tuesday
  | Wednesday
  | Thursday
  | Friday
  | Saturday
```

This declaration defines a new type, named `day`, and seven *constructors* for that type, `Sunday`, `Monday`, *etc.* These constructors *are* the values

of the type `day`—they enumerate all the possible ways of creating a `day`. Moreover, there are *no other ways* to create a value of type `day`.

Note: *In OCaml, user-defined types must be lowercase, and the constructors for such types must be uppercase identifiers.*

After this top-level declaration, you can program with days of the week as a new type of value, for example, we could build a list of some days of the week like this:

```
let weekend_days : day list = [Saturday; Sunday]
```

So, we can put values of type `day` into data structures; how else can we compute with them?

If you think about it, the only fundamental operation needed on a datatype like `day` is the ability to distinguish one value (like `Monday`) from another value (like `Tuesday`). Given that operation, we can build up more complicated operations simply by programming new functions.

Just as OCaml uses pattern matching to allow functions to examine the structure of lists and tuples, we also use pattern matching to do case analysis on user-defined datatypes. The patterns *are* the constructors of the datatype. So, we can write a function that computes with `day`s like this:

```
let is_weekend_day (d:day) : bool =
  begin match d with
    | Sunday -> true
    | Saturday -> true
    | _ -> false
  end
```

Here's another function that, given a day of the week, returns the day after it:

```
let day_after (d:day) : day =
  begin match d with
    | Sunday -> Monday
    | Monday -> Tuesday
    | Tuesday -> Wednesday
    | Wednesday -> Thursday
    | Thursday -> Friday
    | Friday -> Saturday
    | Saturday -> Sunday
  end
```

The fact that the seven constructors `Sunday`, `Monday`, *etc.*, are the *only* ways of creating a value of type `day` means that OCaml can automatically determine when you have covered all of the cases. For example, if you had left out the wildcard case _ in `is_weekday` or if you had accidentally left out one of the days in `day_after`, the OCaml compiler will give you a warning that your pattern matching is not exhaustive.

5.2 Datatypes that carry more information

Although the simple *atomic* data values, like those used for the `day` type, are useful in many situations, it is often useful to associate extra information with a data constructor.

For example, suppose you are collaborating with a biologist who researches DNA and are writing a program to process DNA sequences in various ways. (See Homework 2.)

Recall that there are four basic nucleotides, and we can easily represent them using the atomic datatypes we have already seen:

```
type nucleotide =
  | A     (* adenine *)
  | C     (* cytosine *)
  | G     (* guanine *)
  | T     (* thymine *)
```

Now consider the problem of how to represent data being produced by some potentially faulty sensor that is able to detect counts of nucleotides or triples of nucleotides that are bundled together into a codon. That is, the experiment can produce one of three possible measurements: the measurement can be *missing* (indicating a fault), the measurement can be a *nucleotide count* which consists of a particular nucleotide value along with an integer representing the count detected, or the measurement can be a *codon count* which consists of a triple of nucleotides along with a count.

In OCaml, we can use the **`type`** keyword to create a datatype whose values contain the information described above:

```
type experiment =
  | Missing
  | NucCount of nucleotide * int
  | CodonCount of (nucleotide * nucleotide * nucleotide) * int
```

This type declaration introduces the type `experiment` and three possible constructors. The `Missing` constructor is an atomic value—it doesn't depend on any more information. The other two constructors, `NucCount` and `CodonCount` each have some data associated with them. This is indicated syntactically by the `of` keyword followed by a type.

To build a `NucCount` value, for example, we simply provide the `NucCount` constructor with a pair consisting of a `nucleotide` and an `int`. Similarly, we can build a `CodonCount` by providing a triple of `nucleotide`s and an `int`. Here are some examples values, all of which have type `experiment`:

```
Missing
NucCount(A, 17)
NucCount(G, 124)
CodonCount((A,C,T), 0)
CodonCount((G,A,C), 2512)
```

Just as we use pattern matching to do case analysis on the atomic values in the `day` type, we can also use pattern matching on these more complex data structures. The only difference is that since these constructors carry extra data, we can use nested patterns (see §4.2) to bind identifiers to the subcomponents of a value. Here, for example, is a function that extracts the numeric count information from each experiment:

```
let get_count (m:measurement) : int =
  begin match m with
    | Missing -> 0
    | NucCount(_,n) -> n
    | CodonCount(_,n) -> n
  end
```

Note that since the `NucCount` constructor takes a pair, we use a nested tuple pattern to bind its components in the branches. The `CodonCount` branch is similar. If we had instead been looking for a particular occurrence of a codon triple, we might have used a more explicit pattern instead. For example, suppose we are looking for the counts of experiments for which a codon whose first and last nucleotide is `G`. We could write such a function like this:

```
let get_G_G_count (m:measurement) : int =
  begin match m with
    | CodonCount((G,_,G), n) -> n    (* codon with first = last = G *)
    | _ -> 0                         (* all others are irrelevant *)
  end
```

5.3 Type abbreviations

The examples above show the utility of defining *new* datatypes. The type structure defines both how we can create values of the type and how we inspect a value to determine its constituent parts. Sometimes, however, it is convenient to be able to name *existing* types, both to emphasize the abstraction and to make it simpler to work with.

Returning to the DNA example, we might find it useful to define a type for codons, which consist of a triple of nucleotides, and a type for helices, which are just lists of nucleotides. We do so in OCaml using *type abbreviations*, like this:

```
type codon = nucleotide * nucleotide * nucleotide
type helix = nucleotide list
```

These top-level declarations introduce new *names* for existing types. Note that, unlike the datatype definitions we saw above, these `type` declarations don't use a list of `|`-separated constructor names. OCaml can tell the difference because constructor names (`A`,`C`,`Missing`, `NucCount`) are always capitalized but type identifiers (`day`, `nucleotide`, `codon`) are always lowercase.

After having made these type abbreviations, we can use each name interchangeably with its definition throughout the remainder of the program, including subsequent type definitions. For example, if we have made these type abbreviations, we could shorten the type definition for `experiment` to this:

```
type experiment =
  | Missing
  | NucCount of nucleotide * int
  | CodonCount of codon * int     (* codon instead of a triple *)
```

Since a type abbreviation is just a new name for an existing type, we can use whatever functions or pattern matching operations were available for the existing type to process data of the abbreviated type.

5.4 Recursive types: lists

There is one final wrinkle in understanding user-defined datatypes, and that is the idea that when creating a new type definition, we can define the type *recursively*. That is, the data associated with one of the type's constructors can mention the type being defined.

For example, we are already familiar with the type `string list`, which contains lists of strings and has two constructors: `[]` (nil) and `::` (cons); for a refresher, see §3. We can define our own version of string lists by using a recursive type, like this:

```
type my_string_list =
  | Nil
  | Cons of string * my_string_list
```

Here, the constructor `Nil` plays the role of `[]`, and a value like `Cons(v,tl)` plays the role of `v::tl`. We can program recursive functions over these lists just as we would the built-in ones. For example, here is the length function for the `my_string_list` type:

```
let rec my_length (l:my_string_list) : int =
  begin match l with
    | Nil -> 0
    | Cons(_, tail) -> 1 + (my_length tail)
  end
```

We could even write functions that convert between `my_string_list` and the built-in `string list` type:

```
let rec list_of_my_string_list (l:my_string_list) : string list =
  begin match l with
    | Nil -> []
    | Cons(v,tl) -> v::(list_of_my_string_list tl)
  end

let rec my_string_list_of_list (l:string list) : my_string_list =
  begin match l with
    | [] -> Nil
    | v::tl -> Cons(v, my_string_list_of_list tl)
  end
```

Chapter 6

Binary Trees

OCaml includes built-in syntax for lists, and we have already seen how we can create a user-defined datatype that has the same structure as those built-in lists. Lists are a very common pattern in programming because sequences of data occur naturally in many settings—the list of choices in a menu, the list of students taking a course, the list of results generated by a search engine, *etc.*.

In this chapter, we introduce another frequently occurring data structure—the *binary tree*. Sometimes, such trees arise naturally from the problem domain: for example the evolutionary tree that arises when a species evolves into two new species. Other times, computer scientists use trees because they let us exploit an ordering on data to efficiently search for the presence of an element in a large set of possible values, as we will see below.

For the purposes of this Chapter, we'll consider a simple example of binary trees. What is a binary tree? A binary tree is either `Empty` (meaning that it has no elements in it), or it is a `Node` consisting of a left subtree, an integer *label*, and a right subtree.

In OCaml, we can represent this datatype like so:

```
type tree =
  | Empty
  | Node of tree * int * tree
```

Figure 6.1 shows a picture of a typical binary tree. The *root* node is at the top of the tree. A *leaf* is a node both of who's children are `Empty`—they appear at the bottom of the tree in this picture.[1] Any node that is not a leaf

[1]For some reason, trees in computer science have their roots at the top and their leaves

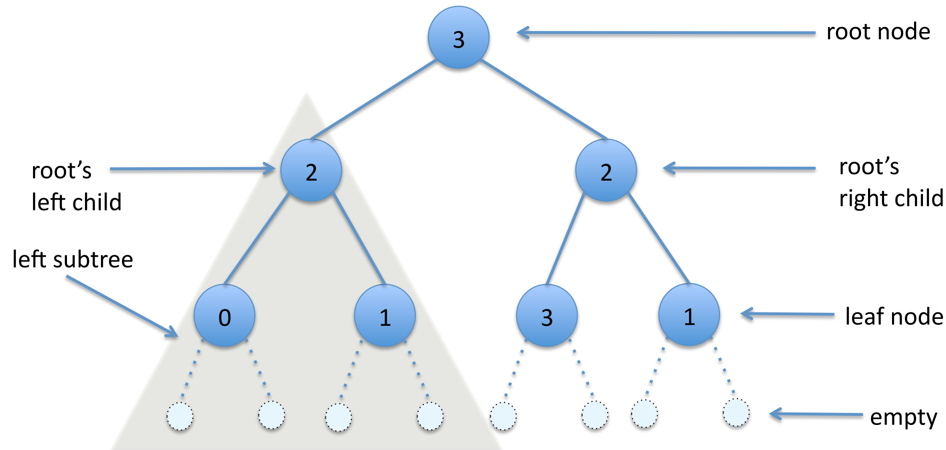


Figure 6.1: The parts of a binary tree.

is sometimes called an *internal* node of the tree.

The tree in Figure 6.1 is represented in OCaml by simply building up a value from the tree constructors. The serial nature of text isn't very good at representing the 2D picture, but you can still see how the picture corresponds to the OCaml value:

```
let tree_fig_51 : tree =
  Node(Node(Node(Empty, 0, Empty),
            2,
            Node(Empty, 1, Empty)),
       3,
       Node(Node(Empty, 3, Empty),
            2,
            Node(Empty, 1, Empty)))
```

We can simplify the presentation a little bit be creating a helper function to make a leaf when given the `int` for its label:

at the bottom... unlike real trees.

```
let leaf (i:int) : tree =
  Node(Empty, i, Empty)

let tree_fig_51 =
  Node(Node(leaf 0,
            2,
            leaf 1),
       3,
       Node(leaf 3,
            2,
            leaf 1))
```

Note that the definition of the type `tree` is recursive, but, unlike the list datatype we saw previously, there are *two* occurrences of `tree` within a `Node`. This means that when we write a recursive function that computes with trees, it will in general have *two* recursive calls.

Here are some simple examples that compute basic properties about a tree:

```
(* counts the number of nodes in the tree *)
let rec size (t:tree) : int =
  begin match t with
    | Empty -> 0
    | Node(l,_,r) -> 1 + (size l) + (size r)
  end

(* counts the longest path from the root to a leaf *)
let rec height (t:tree) : int =
  begin match t with
    | Empty -> 0
    | Node(l,_,r) -> 1 + max (height l) (height r)
  end
```

The *size* of a tree is the total number of nodes in the tree (ignoring the labels of those nodes). A tree's *height* is the length of the longest path from the root to any leaf.

Different ways of processing a tree use can traverse the elements in different orders, depending on the desired goal. Three canonical ways of traversing the tree are *in order* (first the left child, then the node, then the right child), *pre order* (first the node, then the left child, then the right), and *post order* (first the left child, then the right child, then the node). We can write functions that enumerate the elements of a tree in these orders by serializing the tree into a list:

```
(* returns the in order traversal of the tree *)
let rec inorder (t:tree) : int list =
  begin match t with
    | Empty -> []
    | Node(l,n,r) -> (inorder l)@(n::(inorder r))
  end

(* returns the preorder traversal of the tree *)
let rec preorder (t:tree) : int list =
  begin match t with
    | Empty -> []
    | Node(l,n,r) -> n::(preorder l)@(preorder r)
  end

(* returns the post-order traversal of the tree *)
let rec postorder (t:tree) : int list =
  begin match t with
    | Empty -> []
    | Node(l,n,r) -> (postorder l)@(postorder r)@[n]
  end
```

For the example tree in Figure 6.1, we have:

`inorder tree_fig_51` ⟹ `[0;2;1;3;3;2;1]`

`preorder tree_fig_51` ⟹ `[3;2;0;1;2;3;1]`

`postorder tree_fig_51` ⟹ `[0;1;2;3;1;2;3]`

One use for such tree traversals is to *search* for a particular element. For example, we can use the following function to determine whether a tree has a node labeled by the given integer:

```
(* an pre-order search through an arbitrary tree.
   returns true if and only if the tree contains n  *)
let rec contains (t:tree) (n:int) : bool =
  begin match t with
    | Empty -> false
    | Node(lt, x, rt) ->
        x = n || (contains lt n) || (contains rt n)
  end
```

This function first checks the root node's value, `x`, to see if it is `n`. If so, the answer is **`true`** (recall that OCaml's boolean or operator `||` is short circuiting—if its left argument evaluates to **`true`** then the right-hand argument is never evaluated). In the case that `x <> n`, the search continues recursively into first the left sub tree and then the right subtree.

Chapter 7

Binary Search Trees

The function `contains` might, in the worst case, have to search through all the nodes of the tree. In particular, when looking for an element `n` that *isn't* in the tree, this search will continue until every node is examined. For a small tree, that isn't such a problem, but in the case that the tree contains tens of thousands or millions of elements, such a search can take a long time.

Can we do better? It depends on what the data in the tree is representing. If it is the *structure* of the tree that matters or if there are patterns in the data of the nodes of the tree, then we may not be able to improve the search process—for example, it might be important that a certain element appears many times at various points in the tree.

However, there is another common case: the nodes of the tree are intended to be distinct labels, and what matters is whether a given label is present or absent in the tree. In this case, we can think of the tree of nodes as representing the *set* of node labels and the `contains` function as determining whether a given label is in the set.

If we further assume that the labels can be *linearly ordered* (i.e. arranged on the number line), it is possible to *dramatically* improve the performance of searching the tree by exploiting that ordering. The basic idea is the same one underlying the telephone book or a dictionary—arranging the data in a known order, for example by alphabetical order, allows someone who is searching through the data to skip over large irrelevant parts.

For example, when looking up the telephone number for a taxi cab, I might flip open the phone book to the "L" section (probably in the "lawyers") section. Since I know that "taxi" comes after "lawyer" alpha-

betically, I don't even have to bother to look earlier in the phone book. Instead, I flip much later, perhaps to the "R" part, where I see "restaurant" listings. Since "taxi" is still later, I know I don't have to look at the intervening pages. Flipping once more to the "V" section of "veterinarians", I now know to flip just a bit earlier, where I finally hit the "T" for "taxi" part of the phone book.

The basic idea of a binary search tree is to arrange the data in the tree such that by looking at the label of the node being visited the algorithm knows whether to proceed into the left child or the right child, and, moreover, it knows that the unvisited child contains irrelevant nodes.

This leads us to the *binary search tree invariant*.

Definition 7.1 (Binary Search Tree Invariant)**.**

- `Empty` *is a binary search tree.*
- *A tree* `Node(lt,x,rt)` *is a binary search tree if* `lt` *and* `rt` *are both binary search trees, and every label of* `lt` *is less than* `x` *and every label of* `rt` *is greater than* `x`*.*

Note that this definition, like the structure of a binary tree, is itself *recursive*—we define what it means to be a binary search tree in terms of binary search trees. Figure 7.1 shows an example binary search tree. The edges are marked with $<$ and $>$ to indicate the relationship between a node's label and its children.

What is the advantage of this invariant? Well, unlike the `contains` function defined above, which potentially has to look at every node of the tree to see if it contains a given element, we can write a `lookup` function that only searches the relevant parts of the tree.

```
(* ASSUMES: t is a binary search tree
   Determines whether t contains n *)
let rec lookup (t:tree) (n:int) : bool =
  begin match t with
    | Empty -> false
    | Node(lt, x, rt) ->
        if x = n then true
        else if n < x then lookup lt n
        else lookup rt n
  end
```

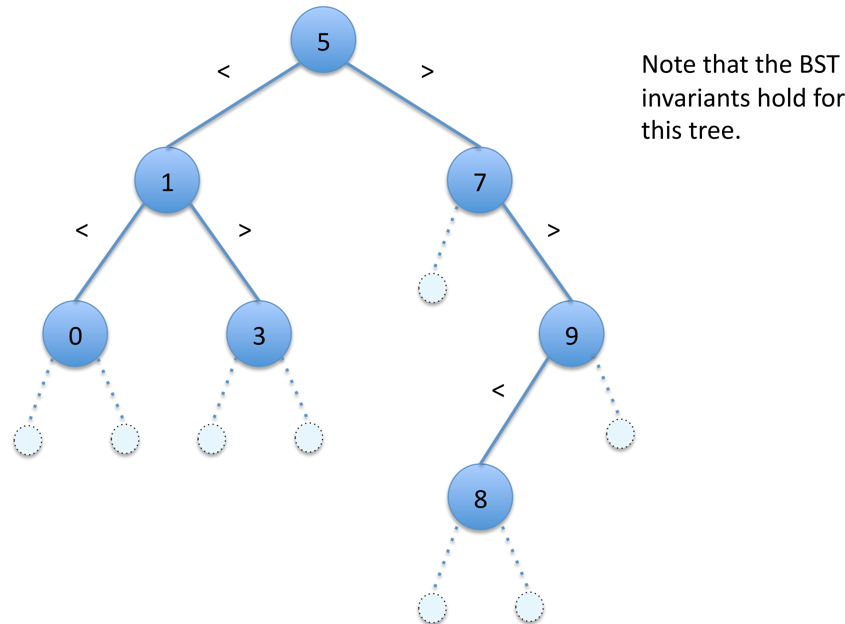


Figure 7.1: A binary search tree.

Note the difference: `lookup` will only ever take either the left branch (if `n` is less than the node's label) or the right branch (if `n` is greater than the node's label) when searching the tree.

How much difference could this make in practice? Consider the case of a binary search tree containing a million distinctly labeled nodes. Then the `contains` function will have to look at a million nodes to determine that an element is *not* in the tree. In contrast, if the `lookup` function takes time proportional to the *height* of the tree. In the good case when the tree is very full (i.e. almost all of the nodes have two children), then the height is roughly logarithmic in the size of the tree[1]. In the case of a million nodes, this works out to approximately 20. Thus, doing a `lookup` in a binary search tree will be about 50,000 times faster than using `contains`.[2]

[1] A complete, balanced binary tree of height h has $2^h - 1$ nodes, or, conversely if there are n nodes, then the tree height is $\log_2 n$.

[2] The assumption that the tree is nearly full and balanced is a big one—to ensure that this is the case more sophisticated techniques (e.g. red–black trees) are needed. See CIS

7.1 Creating Binary Search Trees

We now know how to do efficient lookup in a binary search tree by exploiting the ordering of the labels and the invariants of the tree. How do we go about obtaining such a tree?

One possibility would be to simply check whether a given tree satisfies the binary search tree invariant. We can code up the invariant as a boolean valued-function that determines whether a given tree is a binary search tree or not.

To do so, we first create two helper functions that can determine whether all of the nodes of a tree are less than (or greater than) a particular `int` value:

```
(* helper functions for writing is_bst *)
(* (tree_less t n) is true when all nodes of t are
   strictly less than n
*)
let rec tree_less (t:tree) (n:int) : bool =
  begin match t with
    | Empty -> true
    | Node(lt, x, rt) ->
       x < n && (tree_less lt n) && (tree_less rt n)
  end

(* (tree_gtr t n) is true when all nodes of t are
   strictly greater than n
*)
let rec tree_gtr (t:tree) (n:int) : bool =
  begin match t with
    | Empty -> true
    | Node(lt, x, rt) ->
       x > n && (tree_gtr lt n) && (tree_gtr rt n)
  end
```

The `is_bst` function uses these two helpers to directly encode the binary search tree invariant as a program:

121 if you're interested in such datastructures.

```
(* determines whether t satisfies the bst invariant *)
let rec is_bst (t:tree) : bool =
  begin match t with
    | Empty -> true
    | Node(lt, x, rt) ->
        is_bst lt && is_bst rt &&
        (tree_less lt x) && (tree_gtr rt x)
  end
```

This solution isn't very satisfactory, though. It is very unlikely that some tree we happen to obtain from somewhere actually satisfies the binary search tree invariant. Moreover, checking that the tree satisfies the invariant is pretty expensive.

A better way to construct a binary search tree, is to start with a simple binary search tree like `Empty`, which trivially satisfies the invariant, and then `insert` or `delete` nodes as desired to obtain a new binary search tree.

Each operation must *preserve* the binary search tree invariant: given a binary search tree `t` as input, `insert t n` should produce a new binary search tree that contains the same set of elements as `t` but additionally contains `n`. If `t` happens to already contain `n`, then the resulting tree is just `t` itself.

How can we implement such an insertion function? The key idea is that inserting a new element is just like searching for it—if we happen to find the element we're trying to insert, then the result is just the input tree. On the other hand, if the input tree does not already contain the element we're trying to insert, then the search will find a `Empty` tree, and we just need to replace that empty tree with a new leaf node containing the inserted element. In code:

```
(* ASSUMES: t is a binary search tree.
   Inserts n into the binary search tree t,
   yielding a new binary search tree *)
let rec insert (t:tree) (n:int) : tree =
  begin match t with
    | Empty ->
       (* element not found, create a new leaf *)
        Node(Empty, n, Empty)
    | Node(lt, x, rt) ->
        if x = n then t
        else if n < x then Node (insert lt n, x, rt)
        else Node(lt, x, insert rt n)
  end
```

The code for `insert` exactly mirrors that of `lookup`, except that it returns a tree at each stage rather than simply searching for the element. We have to check that the resulting tree maintains the binary search tree invariant, but this is easy to see, since we only ever insert a node `n` to the left of a node `x` when `n` is strictly less than `x`. (And similarly for insertion into the right subtree.)

Deletion is more complex, because there are several cases to consider. If the node we are trying to delete is not already in the tree, then `delete` can simply return the original tree. On the other hand, if the node is in the tree, there are three possibilities. First: the node to be deleted is a leaf. In that case, we simply remove the leaf node by replacing it with `Empty`. Second, the node to be deleted has exactly one child. This case too is easy to handle: we just replace the deleted node with its child tree. The last case is when the node to be deleted has two non-empty subtrees. The question is how delete the node while still maintaining the binary search tree invariant.

This requires a bit of cleverness. Observe that the left subtree must be non-empty, so it by definition contains a *maximal* element, call it `m`, that is still strictly less than `n`, the node to be deleted. Note also that `m` is *strictly less* than all of the nodes in the right subtree of `n`. Both of these properties follow from the binary search tree invariant. We can therefore *promote* `m` to replace `n` in the resulting tree, but we have to also remove `m` (which is guaranteed to be a leaf) at the same time.

Putting all of these observations together gives us the following code:

```
(* returns the maximum integer in a NONEMPTY bst t  *)
let rec tree_max (t:tree) : int =
  begin match t with
    | Empty -> failwith "tree_max called on empty tree"
    | Node(_,x,Empty) -> x
    | Node(_,_,rt) -> tree_max rt
  end

(* returns a binary search tree that has the same set of
   nodes as t except with n removed (if it's there) *)
let rec delete (n:'a) (t:'a tree) : 'a tree =
  begin match t with
    | Empty -> Empty
    | Node(lt,x,rt) ->
        if x = n then
          begin match (lt,rt) with
            | (Empty, Empty) -> Empty
            | (Node _, Empty) -> lt
            | (Empty, Node _) -> rt
            | _ -> let m = tree_max lt in
                Node(delete m lt, m, rt)
          end
        else
          if n < x then Node(delete n lt, x, rt)
          else Node(lt, x, delete n rt)
  end
```

Chapter 8

Generic Functions and Datatypes

Consider these two functions that compute the lengths of either an `int list` (`length1`) or a `string list` (`length2`):

```
let rec length1 (l:int list) : int =
  begin match l with
  | [] -> 0
  | _::tl -> 1 + (length1 tl)
  end

let rec length2 (l:string list) : int =
  begin match l with
  | [] -> 0
  | _::tl -> 1 + (length2 tl)
  end
```

Other than the type annotation on the argument `l`, both functions are *identical*—they follow exactly the same algorithm, independently of the kind of elements stored in the lists.

Computing a list length is an example of a *generic function*. In this case, the function is generic with respect to the type of list elements. Modern programming languages like OCaml (and also including Java and C#) provide support for writing such generic functions so that the same algorithm can be applied to many different input types.

For example, to write one `length` function that will work for *any* list, we can write:

```
(* a generic version of length *)
let rec length (l:'a list) : int =
  begin match l with
  | [] -> 0
  | _::tl -> 1 + (length tl)
  end
```

The only difference between this generic version and the two above, is that the type of the argument `l` is `'a list`. Here the `'a` is a *type variable*; a place holder for types. The type of `length` says that it works for an input of type `'a list` where `'a` can be instantiated to *any* type.

For example, given the definition above, we can pass `length` a list of integers or a list of strings:

```
length [1;2;3;4]                (* 'a instantiated to int *)
length ["uno", "dos", "tres"]   (* 'a instantiated to string *)
```

OCaml uses the type of the list passed in to `length` to figure out what the `'a` should be. In the first case, the type `'a` is instantiated to `int`, in the second, `'a` is instantiated to `string`.

The `length` function doesn't need to do anything with the elements of the list, but there are generic functions that can manipulate the list elements. For example, here is how we can write a generic `append` function that will take two lists of the same element type and compute the result of appending them:

```
(* generic append *)
let rec append (l1:'a list) (l2:'a list) : 'a list =
  begin match l1 with
  | [] -> l2
  | h::tl -> h::(append tl l2)
  end
```

Here there are a couple of observations to make. First, the type variable `'a` appears in the types of two different inputs (`l1` and `l2`); this means that whenever OCaml figures out what type `'a` stands for, it must agree with *both* list arguments—it is not possible to call `append` with a `int list` as the first argument and a `string list` as the second argument. Second, the result type of the function also mentions `'a`, which means that the element type of the resulting list is the *same* as the element types of both the input lists. Finally, note that we can still use pattern matching to manipulate generic data: since `l1` has type `'a list` we know that inside the case for

cons `h` must be of type `'a` and `tl` itself has type `'a list`.

Functions may be generic with respect to more than one type of value. For example, below is a generic version of the `zip` function that we saw in §4.5 (the version there worked only with inputs of type `int list` and `string list`):

```
let rec zip (l1:'a list) (l2:'b list) : ('a*'b) list =
  begin match (l1,l2) with
  | ([], []) -> []
  | (h1::tl1, h2::tl2) -> (h1,h2)::(zip tl1 tl2)
  | _ -> failwith "zip called on unequal length lists"
  end
```

Some examples of using `zip` show how it behaves "the same" no matter which types the `'a` and `'b` variables are instantiated to:

- `'a = int` and `'b = string`:

  `zip [1;2;3] ["uno", "dos", "tres"]` $\Longrightarrow$ `[(1,"uno");(2,"dos");(3,"tres")]`

- `'a = int` and `'b = int`:

  `zip [1;2;3] [4;5;6]` $\Longrightarrow$ `[(1,4);(2,5);(3,6)]`

- `'a = bool` and `'b = int`:

  `zip [true;false] [1;2]` $\Longrightarrow$ `[(true,1); (false,2)]`

8.1 User-defined generic datatypes

We saw in §5 how programmers can define their own datatypes in OCaml, but we haven't yet seen how to define a generic datatype like OCaml's built in `list`. The idea is straightforward: we create a generic datatype by *parameterizing* the type by *type variables* (`'a`, `'b`, *etc.*) just like the ones used to write down the types in a generic function.

Recall the definition of `int`-labeled binary trees that we worked with in §6:

```
(* non-generic binary trees with int labels *)
type tree =
  | Empty
  | Node of tree * int * tree
```

We can make this into a generic binary tree type by adding a type parameter like so:

```
(* generic binary trees labeled by 'a values *)
type 'a tree =
  | Empty
  | Node of ('a tree) * 'a * ('a tree)
```

Note the differences: we have generic type `'a tree` that represents binary trees all of whose nodes are contain values of type `'a`. Different concrete instances of such trees may instantiate the `'a` variable differently. The type variable `'a` is a type, so it can be used as part of a tuple, as shown in the case for the `Node` constructor. The recursive occurrences of `tree` must also be parameterized by the same `'a`—this ensures that all of the subtrees of an `'a tree` have nodes consistently labeled by `'a` values.

Here are some examples:

`Node(Empty, 3, Empty)`	:	`int tree`
`Node(Empty, "abc", Empty)`	:	`string tree`
`Node(Node(Empty, (true, 3), Empty), (false, 4), Empty)`	:	`(bool * int) tree`
`Node(Node(Empty, 3, Empty), "abc", Empty)`		*Error! ill-typed*

Such generic datatypes can be computed with by pattern matching, just as we saw earlier. In particular, the constructors of the datatype form the patterns, and those patterns bind identifiers of the appropriate types within the branches. For example, we can write a generic function that "mirrors" (i.e. recursively swaps left and right subtrees) like this:

```
let rec mirror (t:'a tree) : 'a tree =
  begin match t with
  | Empty -> Empty
  | Node(lt, x, rt) -> Node(mirror rt, x, mirror lt)
  end
```

In the branch for the `Node` constructor, the identifiers `lt` and `rt` have type `'a tree` and identifier `x` has type `'a`. Since this function doesn't depend on any particular properties of `'a` it is truly generic.

8.2 Why use generics?

Why are generic functions and datatypes valuable? They allow programmers to re-use algorithms in many contexts. For example, we can define lots of different list functions generically and then re-use them for any particular kind of list we happen to need. In particular, the designers of the generic list functions don't have to be aware of what particular kind of list elements some future program might happen to use. A programmer may find herself needing a `widget list` in a graphics program, but if she needs to know its length, then the generic list `length` will do the trick. Importantly, generic functions work even for types not yet defined when the generic function or datatype was created.

This flexible re-use of code has another benefit: it means less work debugging lots of specialized versions of the same thing. If we had to write a list `length` function for every type of list element, then we would have to have many copies of essentially the same program. Such code duplication becomes a nightmare to maintain in larger-scale software systems. Imagine needing to keep twenty "almost identical but not quite" versions of the same function in sync—if you find a bug in one instance of the code, you have to patch it the same way in all nineteen other instances.

Chapter 9

Modularity and Abstraction

In this chapter we consider another mechanism for re-using code in different contexts: *abstract types* and *modules*. The key idea of an abstract type is to bundle together the *name* of a fresh type together with operations for working with that new type. This *interface* specifies all of the ways that values of the new type can be created and used. The type is considered to be *abstract* because the interface does not reveal details about how the type is implemented "behind the scenes". Instead, various code modules can each provide different implementations of the same interface, perhaps with different performance characteristics.

9.1 A motivating example: finite mathematical sets

Recall from your mathematics courses the notion of a *set*. A set is an un-ordered collection of distinct elements. In mathematical notation, sets are usually written by writing down elements inside of { and } brackets, though sometimes the empty set is written $\emptyset$. Here are some examples:

$$\{1,2,3,4\}$$
$$\{a,b,c\}$$
$$\{(1,2),(3,4),(5,6)\}$$

Even though these sets are written using a list-like notation, the order of the elements doesn't matter. That is, according to mathematics:

$$\{1, 2, 3\} = \{3, 2, 1\} = \{2, 1, 3\}$$

Given a set S, we use the mathematical notation $x \in S$ to indicate the proposition that x is an element of the set S. Therefore, we have, for example:

$$1 \in \{1, 2, 3\}$$

In math, we have various operations that operate on sets. For example, we can combine two sets by taking their *union*, written $S_1 \cup S_2$, which is the set containing exactly the elements found in either S_1 or S_2:

$$\{1, 2, 3\} \cup \{2, 3, 4, 5\} = \{1, 2, 3, 4, 5\}$$

Similarly, set *intersection* $S_1 \cap S_2$ is the set containing exactly the elements found in both S_1 and S_2:

$$\{1, 2, 3\} \cap \{2, 3, 4, 5\} = \{2, 3\}$$

Just as the list abstraction occurs naturally in many problem domains, so too does the notion of *set*: the set of students in a class, the set of coordinates that make up a picture, the set of answers to a survey, the set of data samples from an experiment, *etc.*. That's why the idea of a "set" occurs so frequently in math and computer science—it's a really fundamental concept that appears just about everywhere.

The difference for programming is that OCaml already provides a built-in notion of lists, but the programmer has to implement the set abstraction herself.[1]

9.2 Abstract types and modularity

Suppose we want to implement a type of sets in OCaml. How do we go about that process? The first step of design, as always, is to understand the problem—what concepts are involved and how do they relate to one another. Clearly, sets contain elements, but, just as we can have a list of

[1] Actually OCaml's libraries do provide an implementation of sets in the `Set` module; here we'll see how one could write this library oneself.

integers and a list of strings, the element type can vary from set to set. Thus, we expect the `set` type to be generic over its element type:

```
type 'a set = ...        (* 'a is the type of elements *)
```

How do we create a set and how do we manipulate the sets once we have them? Here there are many possible design alternatives: we are looking for a simple list of operations that will allow us to create and use sets flexibly. We certainly need a way of creating an empty set, and it seems reasonable to be able to add an element to or remove an element from an existing set. Taking a cue from mathematics, we might also consider adding a union operation. It is also worth thinking about how sets relate to other datatypes like lists: for example, we might want to be able to create a set out of a list of elements. Putting all of these considerations together, we arrive at the following operations for creating sets:

```
let empty : 'a set = ...
let add (x:'a) (s:'a set) : 'a set = ...
let remove (x:'a) (s:'a set) : 'a set = ...
let union (s1:'a set)(s2:'a set) : 'a set = ...
let list_to_set (l:'a list) : 'a set = ...
```

We also need a way of examining the contents of a set, determining whether a given set is empty, or whether it is equal to another set. It might also be useful to be able to enumerate the elements of a set as a list. This yields these further operations:

```
let is_empty (x:'a set) : bool = ...
let member (x:'a) (s:'a set) : bool = ...
let equal (s1:'a set) (s2:'a set) : bool = ...
let elements (s:'a set) : 'a list = ...
```

Interfaces: `.mli` files and signatures

Having understood the concepts related to the set datatype and identified the operations that connect them, we can now proceed to the second step of the design process: formalizing the interface. To do so, we need to understand a bit about how programming languages package together code as re-usable components.

In programming languages jargon, a *module* is an independently-compilable collection of code that (typically) provides a few data types and

their associated operations. Modules provide a way of decomposing large software projects into several different pieces, each of which can be developed in isolation. Modules that are intended to provide common and widely-used implementations of datastructures, algorithms, or other operations, are often called *libraries*.

A key feature of modules is that they provide boundaries between different parts of a program. Modules separate code at *interfaces*, which specify the ways in which external code (outside the module) can legally interact with the module's *implementation*, which is the code inside the module that determines the behavior of the module's operations.

The point of the interface is that code outside the module can be written only with reference to the interface—the external code can't (and shouldn't) depend on the particular details of how the operations supported by the interface are implemented.

Different programming languages provide different mechanisms for specifying module interfaces—C uses "header" files and Java uses the `interface` keyword, for example. As we shall see, OCaml uses either a `.mli` file or a "module type signature". The commonality among these approaches is the ability to write down a specification using the types of the operations in the module.

For the `'a set` example above, if we look at only the types of each of the operations, we are left with this type signature:

```
type 'a set     (* 'a is the element type *)

val empty  : 'a set
val add    : 'a -> 'a set -> 'a set
val union  : 'a set -> 'a set -> 'a set
val remove : 'a -> 'a set -> 'a set
val list_to_set : 'a list -> 'a set

val is_empty : 'a set -> bool
val member   : 'a -> 'a set -> bool
val equal    : 'a set -> 'a set -> bool
val elements : 'a set -> 'a list
```

Note that we are using the "arrow" notation to specify function types (see §2.7), so we can read the type of `add`, for example, as a function that takes a value of type `'a` and and `'a set` and returns an `'a set`. Also note that, unlike an implementation, we use the keyword **`val`** (instead of **`let`**) to say that any module that satisfies this interface will provide a named

value of the appropriate type. So any implementation of the `set` module must provide an `add` function with the type mentioned above.

OCaml provides two ways of defining module interfaces. The first way is to use OCaml modules corresponding to file names. In this style, we put the above interface code in a file ending with the `.mli` extension, for example `ListSet.mli`, and the implementation in a similarly-named `.ml` file, for example `ListSet.ml`. The "i" part of the file extension stands for "interface", and each OCaml `.ml` file is, by default, associated with the correspondingly named `.mli` file.[2]

The second way to define an interface is to use an explicitly named module type signature. For example, we might put the following type signature in a file called `MySet.ml`:

```
module type Set = sig  (* A named interface called Set *)
  type 'a set     (* 'a is the element type *)

  val empty  : 'a set
  val add    : 'a -> 'a set -> 'a set
  val union  : 'a set -> 'a set -> 'a set
  val remove : 'a -> 'a set -> 'a set
  val list_to_set : 'a list -> 'a set

  val is_empty : 'a set -> bool
  val member   : 'a -> 'a set -> bool
  val equal    : 'a set -> 'a set -> bool
  val elements : 'a set -> 'a list
end
```

This program gives the name `Set` to the interface (a.k.a. module type) defined by the signature between the **`sig`** and **`end`** keywords. Other code can appear before or after this declaration, but it won't be considered part of the `Set` signature.

We can also create an explicitly-named module with a given interface using a similar notation. Rather than put the code implementing each set operation in a `.ml` file, we do this:

[2]In fact, if you don't create a `.mli` file for a given `.ml` file, the OCaml compiler will create one for you by figuring out the most liberal interface it can for the given implementation. You may have noticed these files appearing when you work with OCaml projects in Eclipse.

```
module LSet : Set = struct
  type 'a set = ...

  let empty : 'a set = ...
  let add (x:'a) (s:'a set) : 'a set = ...
  let remove (x:'a) (s:'a set) : 'a set = ...
  let union (s1:'a set)(s2:'a set) : 'a set = ...
  let list_to_set (l:'a list) : 'a set = ...

  let is_empty (x:'a set) : bool = ...
  let member (x:'a) (s:'a set) : bool = ...
  let equal (s1:'a set) (s2:'a set) : bool = ...
  let elements (s:'a set) : 'a list = ...

end
```

Here the keywords **struct** and **end** delineate the code that is considered to be part of the LSet module. The advantage of having a named interface is that we can re-use it in other contexts. For example, we might create a second, more efficient, implementation of sets in a new module:

```
module BSet : Set = struct
  ... (* a different implementation of sets *)
end
```

Regardless of whether we choose to use .mli files or explicitly-named interfaces, OCaml will check to make sure that the implementation actually complies with the interface. This means that every operation, type, or value declared in the interface must have an identically-named, but fully realized, implementation in the module. Moreover, OCaml will check that the implementation and the interface agree with respect to the types they use. The implementation may include more than necessary to meet the interface—it can contain extra types, helper functions, or auxiliary values that aren't revealed by the interface.

Because each file of an OCaml program corresponds to a module, if we want to access components of one file from another file, we have to either use the module "dot" notation or **open** the module to expose the identifiers it defines. For example, if we have defined the set module in ListSet.ml (whose interface is given by ListSet.mli), and we want to use those operations in a different module found in Foo.ml, we write ListSet.<identifier> inside of Foo.ml. For example, we might write:

```
let add_to_set (s:int ListSet.set) : int ListSet.set =
  ListSet.add 1 (ListSet.add 2 s)
```

This can become burdensome, so OCaml also provides the `open` command, which reveals all of the operations defined in a module's interface without the need to use the `ListSet.` prefix. The following is equivalent to the above:

```
;; open ListSet

let add_to_set (s:int set) : int set =
  add 1 (add 2 s)
```

There is one gotcha—if we use explicitly named modules, we still need to either explicitly use dot notation for the implicitly defined module corresponding to the filename. For example, if the `Set` interface and the two modules `LSet` and `BSet` were all defined in the `MySet.ml` file, we could write:

```
;; open MySet      (* reveal the LSet and BSet modules *)

(* work with LSet values *)
let add_to_lset (s:int LSet.set) : int LSet =
  LSet.add 1 (LSet.add 2 s)

(* work with BSet values *)
let add_to_bset (s:int BSet.set) : int BSet =
  BSet.add 1 (BSet.add 2 s)
```

Implementations and Invariants

The power of abstract ypes as embodied by modules and interfaces is that the interface can *hide* representation details from clients of the module. In particular, the interface can *omit* the definition of how a particular type is implemented internally to the module.

Consider the `Set` module, for example. There are many possible ways we could imagine implementing a datastructure for sets. Since sets are similar to lists, except that sets contain no duplicates and are considered to be unordered, we might choose to *represent* a set by a list subject to an *invariant* that is enforced by the implementing module.

As specific examples, here are just a few of the many possible represen-

tations for the abstract type `'a set`, along with an invariant that might be useful for implementing the set:

Representation Type	Invariant
`'a list`	no duplicate elements
`'a list`	no duplicates, in sorted order
`'a tree`	no duplicate elements
`'a tree`	binary-search-tree invariants

Inside a module implementing the `Set` interface, we are free to choose any suitable type and invariants for concretely representing the abstract type `'a set`. Inside the module, we can then implement the set operations in terms of that representation type. Crucially, if we are careful to ensure that any value of the abstract type produced by the module satisfies the representation invariants, we can assume that any such values passed in to the functions of the module will already satisfy the invariants—external code cannot violate the representation invariants.

Let us see how this is helpful by example. Suppose that we choose to implement the `Set` interface using `'a list` as the representation type and "no duplicates" as the invariant. We can proceed to implement the functions of the `Set` interface like so:

```
module LSet : Set = struct

  (* inside the module we represent sets as lists *)
  (* INVARIANT: the list contains no duplicates *)
  type 'a set = 'a list

  (* the empty set is just the empty list *)
  let empty : 'a set =
    []

  (* tests whether x is contained in the set s *)
  let rec member (x:'a) (s:'a set) : bool =
    begin match s with      (* we use the fact that s is a list *)
    | [] -> false
    | y::rest -> x = y || member x rest
    end

  (* add the element x to the set s *)
  (* NOTE: add produces a set, so it must maintain
   * the no duplicates invariant *)
  let add (x:'a) (s:'a set) : 'a set =
    if (member x s) then s     (* x is already in the set *)
    else x::s      (* add x to the set *)

  (* remove the element x from the set s *)
  let rec remove (x:'a) (s:'a set) : 'a set =
    begin match s with
    | [] -> []
    | y::rest ->
        if x = y then rest     (* x can't occur in rest
                                  because of the invariant *)
        else y::(remove x rest)
    end

   ...  (* implement the rest of the operations *)

end
```

The choice of which invariants to maintain can impact the implementation of various operations. For example, to implement the `equals` operation on sets when representing them as lists with the "no duplicates" invariant, we must check that each element of the first list is a member of the second, and vice-versa. This expensive equality check is necessary because the invariant doesn't say anything about the ordering of elements

in the list, and sets are supposed to be unordered.

On the other hand, if we had chosen a stronger invariant, such as representing a set as a sorted list of elements (with no duplicates), then testing for equality simply amounts to checking that each list contains the same elements in order. The tradeoff is that with this stronger invariant, the `add` function becomes more expensive—we have to insert the newly added element at the appropriate location to maintain the sorting invariant.

9.3 Another example: Finite Maps

As another example of an abstract type, consider the problem of implementing a *finite map*, which is a data structure that keeps track of a finite set of *keys*, each of which is associated with a particular *value*. These kinds of datastructures are useful for representing things like dictionaries—here the keys are words and the values are their definitions. We could also use a finite map to capture the relationship between a collection of students and their majors in college.

Using an informal notation, we might write down a finite map from students to their majors like this:

$$\begin{array}{rcl} \text{Alice} & \mapsto & \text{CSCI} \\ \text{Bob} & \mapsto & \text{ESE} \\ \text{Chuck} & \mapsto & \text{FNCE} \\ \text{Don} & \mapsto & \text{BIOL} \\ & \ldots & \end{array}$$

As with the sets, we can then think about which operations are needed to work with finite maps. Clearly we need ways of creating finite maps, adding key–value bindings to an existing map, looking up the value that corresponds to a key, *etc.*.

After some thought, we might arrive at an interface for finite maps that looks like this:

```
module type Map = sig
  (* a finite map from keys of type 'k to values of type 'v *)
  type ('k,'v) map

  (* ways to create and manipulate finite maps *)
  val empty    : ('k,'v) map
  val add      : 'k -> 'v -> ('k,'v) map -> ('k,'v) map
  val remove : 'k -> ('k,'v) map -> ('k,'v) map

  (* ways to ask about the contents of a finite map *)
  val is_empty : ('k,'v) map -> bool
  val mem      : 'k -> ('k,'v) map -> bool
  val find     : 'k -> ('k,'v) map -> 'v
  val keys     : ('k,'v) map -> 'k list
  val values : ('k,'v) map -> 'v list
  val bindings : ('k,'v) map -> ('k*'v) list
end
```

Also as with sets, we can imagine many ways of concretely implementing such a finite map interface. For example, we can represent the type `('k,'v) map` as a list of `'k*'v` pairs, perhaps with an invariant that requires each key appearing in the list to be unique. We could also choose to implement finite maps using binary search trees, where we index the nodes by the key component and also store a value with each key.

If we follow the first approach, we might end up with this implementation of the interface above:

```
module ListMap : Map = struct

  (* INVARIANT: each key occurs at most once in the list *)
  type ('k,'v) map = ('k * 'v) list

  let empty : ('k,'v) map = []

  let is_empty (m:('k,'v) map) =
    m = []

  let rec mem (k:'k) (m:('k,'v) map) : bool =
    begin match m with
      | [] -> false
      | (k1,_)::rest -> k1=k || mem k rest
    end
```

```
  let rec find (k:'k) (m:('k,'v) map) : 'v =
    begin match m with
      | [] -> failwith "Not found"
      | (k1,v)::rest -> if k1=k then v else find k rest
    end

  let rec remove (k:'k) (m:('k,'v) map) : ('k,'v) map =
    begin match m with
      | [] -> []
      | (k1,v1)::rest ->
        if k1=k then rest
        else (k1,v1)::(remove k rest)
    end

  let add (k:'k) (v:'v) (m:('k,'v) map) : ('k,'v) map =
     (k,v)::(remove k m)

  let rec keys (m:('k,'v) map) : 'k list =
    begin match m with
      | [] -> []
      | (k,v)::rest -> k::(keys rest)
    end

  let rec values (m:('k,'v) map) : 'v list =
    begin match m with
      | [] -> []
      | (k,v)::rest -> v::(values rest)
    end

  let bindings (m:('k,'v) map) : ('k*'v) list =
    m

end
```

Chapter 10

First-class Functions

In this chapter, conclude our tour of value-oriented (or "declarative") programming by studying the ramifications of a beautiful and amazing fact: *functions are values!*

What do I mean by that? Well, just as the number `3` is an `int` that can be used as an argument to a function or as a value computed by a function, functions themselves can be used both as arguments to other functions and as the result of a computation.

For example, consider the following function called `twice`:

```
let twice (f:int -> int) (x:int) : int =
  f (f x)
```

`twice` itself takes an input `f` of function type! In this case, `f` should be an `int -> int` function. What can `twice` do with `f`? It can only *call* the function or pass it to some other function. Here, `twice` calls `f` on the result of calling `f` on the argument `x`.

How do we use such a function? We can call `twice` by passing it an argument of type `int -> int`. For example, suppose we define an `add_one` function:

```
let add_one (z:int) : int = z + 1
```

Then we can write the expression `twice add_one 3`, which will evaluate to the value `5`. To see why, we just follow the familiar rules of substitution:

```
       twice add_one 3
⟼     add_one (add_one 3)     substitute add_one for f and 3 for x in twice
⟼     add_one (3 + 1)         substitute 3 for z in add_one
⟼     add_one 4               because 3+1 ⟹ 4
⟼     4 + 1                   substitute 4 for z in add_one
⟼     5                       because 4+1 ⟹ 5
```

Similarly, if we have the function `square`:

```
let square (z:int) : int = z * z
```

Then we have `twice square 3` $\Longrightarrow$ `81` because calling `square` twice computes the `z` to the 4th power.

10.1 Partial Application and Anonymous Functions

How do we return a function as the result of another function? Consider this example:

```
let make_incrementor (n:int) : int -> int =
  let helper (x:int) = n + x in
  helper
```

This function takes an `int` as input and *returns* a function of type `int -> int`. What does that function do? When it is called on some value `x`, it will compute the result `n + x`.

How does this function evaluate? If we apply `make_incrementor` to `3`, we can compute as follows:

```
       make_incrementor 3
⟼     let helper (x:int) = 3 + x in helper    substitute 3 for n
```

At this point, we seem to get stuck: what value is computed for `helper`?

More puzzling is how to think about a function that takes more than one argument. Suppose we apply the function to only one input—what happens then? Here's an example:

```
let sum (x:int) (y:int) : int = x + y     (* has two arguments *)

let sum_applied (x:int) : int -> int =
  sum x
```

The function `sum` has type `int -> int -> int`. If we *partially apply it*—give it only *some* of its inputs—then we can treat that partial application as a function! In this case, since we partially apply `sum` to an integer `x`, we are left with a function that expects only one input, namely `y`.

To explain how to compute with such partially applied functions and functions as results, we need to introduce one new concept: the *anonymous* function. An anonymous function is exactly what the term implies—it is a function without a name. Using OCaml syntax, we can write an anonymous function like this:

```
fun (x:int) -> x + 1
```

Here, the keyword `fun` indicates that we are creating a value of function type. In this case, the function takes one input called `(x:int)`, and the body of the function is the expression `x+1`.

We can write an anonymous version of the `sum` function above like this:

```
fun (x:int) (y:int) -> x + y
```

These anonymous functions *are* values. If we want to give such a function a name, we can do so using the regular `let` notation. For example, the following definition is equivalent to the "named" version of `sum` given above:

```
let sum : int -> int -> int = fun (x:int) (y:int) -> x + y
```

Note: *The syntax for anonymous functions, unlike named functions, does not have a place to write the return type. This is just an oddity of OCaml syntax; in practice OCaml can always figure out what the return type should be anyway.*

We can apply an anonymous function just like any other function, by writing it next to its inputs and using parentheses to ensure proper grouping. We evaluate anonymous function applications by substituting argument value for the parameter name in the body of the function. In the case that there is more than one parameter, we simply keep around the `fun ... -> ...` parts until the function as been fully applied (i.e. it has been applied to enough parameters.

For example, let's see how the anonymous version of `sum` evaluates when applied to just a single input:

```
      (fun (x:int) (y:int) -> x + y) 3
⟼    (fun (y:int) -> 3 + y)              substitute 3 for x
```

The resulting anonymous function *is* the answer of such a computation. Having around anonymous functions means we can name such intermediate computations that result in functions. The fact that "named" definitions are just shorthand for the `let`-named anonymous functions, means we can now see all of the steps in a computation as simple substitution and primitive operations:

```
let sum (x:int) (y:int) : int = x + y
let add_three = sum 3
let answer = add_three 39
```

⟼ *(by equivalence of named an let-bound anonymous forms)*

```
let sum = fun (x:int) (y:int) -> x + y
let add_three = sum 3
let answer = add_three 39
```

⟼ *(substituting the definition of* `sum`*)*

```
let sum = fun (x:int) (y:int) -> x + y
let add_three = (fun (x:int) (y:int) -> x + y) 3
let answer = add_three 39
```

⟼ *(substituting* `3` *for* `x`*)*

```
let sum = fun (x:int) (y:int) -> x + y
let add_three = (fun (y:int) -> 3 + y)
let answer = add_three 39
```

⟼ *(substituting the definition of* `add_three`*)*

```
let sum = fun (x:int) (y:int) -> x + y
let add_three = (fun (y:int) -> 3 + y)
let answer = (fun (y:int) -> 3 + y) 39
```

⟼ *(substituting* `39` *for* `y`*)*

```
let sum = fun (x:int) (y:int) -> x + y
let add_three = (fun (y:int) -> 3 + y)
let answer = 3 + 39
```

⟼ *(because* `3+39` ⟹ `42`*)*

```
let sum = fun (x:int) (y:int) -> x + y
let add_three = (fun (y:int) -> 3 + y)
let answer = 42
```

10.2 List transformation

What can we do with first-class functions? They provide a powerful way to share the common features of many algorithms. As a simple example, consider these two list operations that we saw in the list implementation of finite maps (§9.3):

```
let rec keys (m:('k*'v) list) : 'k list =
  begin match m with
    | [] -> []
    | (k,v)::rest -> k::(keys rest)
  end

let rec values (m:('k*'v) list) : 'v list =
  begin match m with
    | [] -> []
    | (k,v)::rest -> v::(values rest)
  end
```

These functions are nearly identical—each one processes the elements of the input list of key–value pairs in turn, projecting out either the key component or the value component as appropriate.

We can reorganize this program to exploit that common structure by creating a helper function parameterized by a function that says what to do with the key–value pair:

```
let rec helper (f:('k*'v) -> 'b) (m:('k*'v) list) : 'b list =
  begin match m with
    | [] -> []
    | (k,v)::rest -> (f (k,v))::(helper f rest)
  end
```

After factoring out this common algorithm, we can then express `keys` and `values` in terms of this `helper` by remember that the `fst` function returns the first element of a pair and the `snd` function returns the second element.

```
let keys (m:('k*'v) list) : 'k list =
  helper fst m                (* recall: fst (x,y) = x *)

let values (m:('k*'v) list) : 'v list =
  helper snd m                (* recall: snd (x,y) = y *)
```

Now observe that the helper function doesn't really depend on the fact that it is processing key-value pairs. We can further generalize by observing that the helper can be made to work over *all* lists, regardless of their element types, by simply giving `f` the right type. This leads to to the following function, called `transform`:[1]

```
let rec transform (f:'a -> 'b) (l:'a list) : 'b list =
  begin match l with
    | [] -> []
    | h::tl -> (f h)::(transform f tl)
  end
```

This list transformer applies the function `f` to each element of the input list and returns the resulting list—it transforms a list of `'a` values into a list of `'b` values. Such transformations are extremely fundamental to any list-processing programs, so this operation is very useful in practice. It also combines well with anonymous functions, since you can pass an anonymous function as the argument `f`.

Let's look at some examples:

```
transform String.uppercase ["abc"; "dog"; "cat"]
⟹ ["ABC"; "DOG"; "CAT"]

transform (fun (x:int) -> x+1) [1;2;3;4]
⟹ [2;3;4;5]

transform (fun (x:int) -> x * x) [1;2;3;4]
⟹ [1;4;9;16]

transform string_of_int [1;2;3]
⟹ ["1"; "2"; "3"]

transform (fun (x:(int*int)) -> (fst x) + (snd x)) [(1,2); (3,4); (5,6)]
⟹ [3; 7; 11]
```

[1]In an unfortunate accident of fate, the `transform` function is also often called `map`—the intuition is that the function *maps* a given function across each element of the list. This use of the word "map" is not to be confused with the abstract type of finite maps we saw in §9.3.

10.3 List fold

The list transformation function captures one common idiom of list processing, but it is possible to generalize even further. Consider these three functions that are defined using the standard list recursion pattern:

```
let rec length (l:'a list) : int =
  begin match l with
  | [] -> 0       (* base case *)
  | x::tl -> 1 + (length tl)   (* combine x and (length tl) *)
  end

let rec exists (l:bool list) : bool =
  begin match l with
  | [] -> false    (* base case *)
  | x::tl -> x || (exists tl)   (* combine x and (exists tl) *)
  end

let rec reverse (l:'a list) : 'a list =
  begin match l with
  | [] -> []    (* base case *)
  | x::tl -> (reverse tl) @ [x] (* combine x and (reverse tl) *)
  end
```

The comments in the code above indicate the common features of these functions. For *any* function defined by structural recursion over lists, there are two things to consider: First, what should the function return in the case that the list is empty? This is called the *base* case of the recursion because it is where the chain of recursive calls "bottoms out." Second, assuming that you know the value computed by the recursive call on the tail of the list, how do you *combine* that result with the head of the list to compute the answer for the whole list?

In the case of the the list `length` function, for example, the base case says that the empty list has length `0`. The recursive case combines the value of the recursive call `length tl` with the head of the list (which happens to be ignored) to compute `1 + (length tl)`.

For the `exists` function, which determines whether a list of `bool` values contains `true`, the base case indicates that the empty list does not contain `true` (i.e. the result is `false`). The recursive case computes the answer for the whole list with head `x` and tail `tl` by simply returning `true` either when `x` is `true` or when `exists tl` evaluates to `true`—combining the head with the recursive call is just taking their logical "or".

For `reverse`, the base case says that reversing the empty list is just the empty list, and the recursive case says that we can combine the reversal of the tail with x to obtain a completely reversed list by just appending `[x]` at then end of the reversed tail.

So what? All three functions follow the same recursive pattern. We can expose this common structure by creating a *single* function that is parameterized by the `base` value and the `combine` operation—those are the only places where our three examples differ.

Let's call this function `fold`—it "folds up" a list into an answer by following structural recursion. To make `fold` as generic as possible, let's consider implementing it for an arbitrary list of type `'a list`. The result type of a structurally recursive function varies from application to application, so we expect the result type of `fold` to be generic, and, as shown by the `length` and `exists` examples, it could be different from the element type of the list we're folding over. So, the return type of `fold` should be some type `'b`. Those choices dictate the type of `base` and `combine`: `base` must have type `'b` since it is the "answer" for the empty list. Similarly, `combine` takes the head element of the list, which has type `'a`, and the answer obtained from the recursive call on the tail of the list, which has type `'b`, and produces an answer, which must also be of type `'b`.

These considerations lead us to this definition of `fold`:

```
let rec fold (combine:'a -> 'b -> 'b) (base:'b) (l:'a list) : 'b =
  begin match l with
    | [] -> base
    | x::tl -> combine x (fold combine base tl)
  end
```

This recursive function is embodies the *essence* of structural recursion over lists—it is parameterized exactly where there can be some choice about what to do. Note that the first argument passed to `combine` is `x` and that the second argument is the result of recursively folding (with the same `combine` and `base`) over the `tl`.

What is `fold` useful for? Well, we can easily re-implement the three examples above like this:

```
let length2 (l:'a list) : int =
  fold (fun (x:'a) (length_tl:int) -> 1 + length_tl) 0 l

let exists2 (l:bool list) : bool =
  fold (fun (x:bool) (exists_tl:bool) -> x || exists_tl) false l

let reverse2 (l:'a list) : 'a list =
  fold (fun (x:'a) (reverse_tl:'a list) -> reverse_tl @ [x]) [] l
```

I've named the second parameter of the anonymous functions that get passed as `fold`'s `combine` operation to remind us how that parameter is related to the recursive call—there is no other particular significance to the choice of `length_tl`, for example. We could equally well have written `length2` like this:

```
let length2 (l:'a list) : int =
  fold (fun (x:'a) (y:int) -> 1 + y) 0 l
```

Any function that you can write by by structural recursion can be expressed using `fold`. Here, for example, is how to reimplement the `transform` function:

```
let transform (f:'a -> 'b) (l:'a list) : 'b list =
  fold (fun (x:'a) (trans_tl:'b list) -> (f x)::trans_tl)  [] l
```

Chapter 11

Partial Functions: option types

Consider the problem of computing the maximum integer from a list of integers. At first sight, such a problem seems simple—we simply look at each element of the list and retain the maximum. We can start to program this functionality very straightforwardly by using the by-now-familiar list recursion pattern:

```
let rec list_max (l:int list) : int =
  begin match l with
    | [] ->    (* what to do here? *)
    | x::tl -> max x (list_max tl)
  end
```

Unfortunately, this program doesn't have a good answer in the case that the list is empty—there is no "maximum" element of the empty list.

What can we do? One possibility is to simply have the `list_max` function fail whenever it is called on an empty list. This solution requires that we handle three separate cases, as shown below:

```
let rec list_max (l:int list) : int =
  begin match l with
    | [] -> failwith "list_max called on []"
    | x::[] -> x
    | x::tl -> max x (list_max tl)
  end
```

The cases cover the empty list, the singleton list, and a list of two-or more elements. We have to separate out the singleton list as a special case because it is the first length for which the `list_max` function is well defined.

This solution is OK, and can be very appropriate if we happen to know by external reasoning that `list_max` will never be called on an empty lists. We saw such use of failure in the `tree_max` function used to find the maximal element of a *nonempty* binary search tree in §6. However, what if we can't guarantee that the `list_max` function won't be called on an empty list? How else could we handle this possibility without failing, and thereby aborting the program[1]

The problem is that `list_max` is an example of a *partial* function—it isn't well-defined for all possible inputs. Other examples of partial functions that you have encountered are the integer division operation, which isn't defined when dividing by `0`, and the `Map.find` function, which can't return a good value in the case that a key isn't in its set of key–value bindings.

It turns out that we can do better than failing, and, moreover, we already have all the tools needed. The idea is to create a datatype that explicitly represents the absence of a value. We call this datatype the `'a option` datatype, and it is defined like this:

```
type 'a option =
  | None
  | Some of 'a
```

There are only two cases: either the value is missing, represented by the constructor `None`, or the value is present, represented by `Some v`. As with any other datatype, we use pattern matching to determine which case occurs.

Option types are very useful for representing the lack of a particular value. For a partial function like `list_max`, we can alter the return type to specify that the result is optional. Doing so leads to an implementation like this:

[1] OCaml, like Java and other modern languages, supports *exceptions* that can be caught and handled to prevent the program from aborting in the case of a failure. Exceptions are another way of dealing with partiality; we will cover them later in the course.

```
(* NOTE: this version has a different return type! *)
let rec list_max (l:int list) : int option =
  begin match l with
    | [] -> None      (* indicates partiality *)
    | x::tl -> begin match (list_max tl) with
                  | None -> Some x
                  | Some m -> Some max x m
                 end
  end
```

As you can see, this implementation handles the same three cases as in the one that uses `failwith`; the difference is that after the recursive call we must explicitly check (by pattern matching against the result) whether `list_max tl` is well defined.

At first blush, this seems like a rather awkward programming style, and the need to explicitly check for `None` versus `Some v` onerous. However, it is often possible to write the program in such a way such checks can be avoided. For example, here is a cleaner way to write the same `list_max` function by using `fold`:

```
let rec list_max (l:int list) : int option =
  begin match l with
  | [] -> None
  | x::tl -> Some (fold max x tl)
  end
```

The expression `fold max x tl` takes the maximum element from among those in `tl` and `x`, which is always well-defined.

It is also worth pointing out that because the type `'a option` is distinct from the type `'a`, it is never possible to introduce a bug by confusing them—OCaml will force the programmer to do a pattern match before being able to get at the `'a` value in an `'a option`.

For example, suppose we wanted to find the sum of the maximum values of each of two lists. We can write this program as:

```
let sum_two_maxes (l1:int list) (l2:int list) : int option =
  begin match (list_max l1, list_max l2) with
  | (None, None) -> None
  | (Some m1, None) -> Some m1
  | (None, Some m2) -> Some m2
  | (Some m1, Some m2) -> Some (m1 + m2)
  end
```

Here we are forced to explicitly think about what to do in the case that both lists are empty—here we make the choice to make `sum_two_maxes` itself return an `int option`. The option types prevent us from mistakenly trying to naively do `(list_max l1) + (list_max l2)`, which would result in a program crash (or worse) if permitted.

In languages like C, C++, and Java that have a `null` value which can be given *any* type, it is an extremely common mistake to conflate `null`, which should mean "the lack of a value," with "an empty value" of some type. For example, one might try to represent the empty list as `null`. However, such conflation very often leads to so-called "null pointer exceptions" that arise because some part of the program treats a `null` as though it has type `'a`, when it is really meant to be the `None` of an `'a option`.

There is a big difference between "the absence of a list" and "an empty list"—it makes sense to insert an element into the empty list, for example, but it never makes sense to insert an element into "the absence of a list."

Sir Tony Hoare, Turing-award winner and a researcher scientist at Microsoft, invented the idea of `null` in 1965 for a language called ALGOL W. He calls it his "billion-dollar mistake" because of the amount of money the software industry has spent fixing bugs that arise due to unfortunate confusion of the `'a` and `'a option` types. Option datatypes provide a simple solution to this problem.

We will see that option types also play a crucial role when we study linked, mutable datastructures in §16.

Chapter 12

Unit and Sequencing Commands

This Chapter studies a very uninteresting datatype: `unit`. This datatype is uninteresting because it contains exactly one value, called the unit value and written `()`. However, although `unit` itself is uninteresting, it is still *useful*. Here we will see why.

We have already seen `unit` in action in a couple of places. First, in OCaml, every function takes exactly one argument—we can use the *unit* type to indicate that the argument is uninteresting:

```
let f (x:unit) : int = 3
```

Since there is only one value of type `unit`, we can omit the `x:unit` in the definition above, to obtain the equivalent:

```
let f () : int = 3
```

This function takes the unit argument and produces the value `3`; it has type `unit -> int`. We call it, as usual, by function application to the (only!) value of type `unit`, like this: `f ()`.

The `unit` value is first class, and we can use it in `let` bindings and pattern matching like this:

```
let x : unit = ()
let y : string =
        begin match x with
        () -> "only one branch"
        end
```

As with tuples (and other datatypes that require only one branch when

pattern matching), we can also pattern match in `let` and `fun` bindings:

```
let () = print_string "hello"
let g : unit -> int = fun () -> 3
```

We have been using functions that take `unit` inputs to write the `test` predicates of the homework assignments, typically something like:

```
let test () : bool =
  length [1;2;3] = 3
```

Here, `test : unit -> bool` is a function.

We have also seen functions that *return* `unit` values: these are the *commands*. Since commands don't return any interesting data, the only reason to run them is for their *side effects* on the state of the computer. Here are some commands that we have seen, and their types:

```
 print_string  :  string -> unit
print_endline  :  string -> unit
    print_int  :  int -> unit
     run_test  :  string -> (unit -> bool) -> unit
```

So far, we have seen how to run these commands at the program's top level, using the `;;` notation:

```
;; print_string "this prints a string"
```

We can also embed commands within expressions using a binary operator called '`;`'. The idea is that `e1; e2` first runs `e1`, which must be an expression of type `unit`, which may have side effects. The resulting `()` value is discarded and then `e2` is evaluated. Thus, the `;` operator lets us sequence commands.

```
let ans : int =
  print_string "printing is a side effect";
  17
```

This program prints the string as a side effect and then binds the identifier `ans` to the value `17`. Note that `;` is an infix operator—you will get error messages if you write a `;` after the second expression.

```
let ans : int =
  print_string "printing is a side effect";
  17;    (* <-- don't put a semicolon after the second expression! *)
```

As usual, we can nest expressions, and use them inside of local `let`s.

This is very useful for printing out information inside a function body, for example:

```
let f (x:int) : int =
  print_string "x is ";
  print_int x;
  print_string "\n";
  x + x
```

12.1 The use of ';'

We have now seen several places where the symbol ';' appears in OCaml programs (and we'll see one more in the next section).

Unfortunately, `;` in OCaml means different things depending on how it is used. Also unfortunately, none of those usages corresponds exactly to how `;` is used in "usual" imperative programming as in C or Java.

In OCaml programs (but not the Toplevel loop), `;` is always a *separator*, not a *terminator*. The list below collects together all the syntax combinations that use `;`

`;; open Assert`	open a module at the top-level of a program
`;; print_int 3`	run a command at the top-level of a program
`[1; 2; 3]`	separate the elements of a list
`e1; e2`	sequence a command `e1` before the expression `e2`
`{x:int; y:int}`	separate the fields of a record type (see §13)
`{x=3; y=4}`	separate the fields of a record value

We have also seen that, when executing OCaml expressions in the top-level interactive loop, we use `;;` as a terminator to let OCaml know when to run a given expression.

Chapter 13

Records of Named Fields

13.1 Immutable Records

Tuples are a light-weight way to collect together a small number of data values into a coherent package. Sometimes, though, it is nice to give *names* to the different components of such a package so that we can easily remember what the different parts are, and access them.

OCaml, like most other languages, provides a datatype of *records* that are designed specifically for this purpose. As we shall see in the upcoming course material, the humble record plays an important role in both imperative and object-oriented programming. Here we study the basics.

Record types are like tuples with named fields. The type is written as a list of $\langle id \rangle$: $\langle type \rangle$ pairs between `{ }` brackets. For example, suppose we wanted to create a program for manipulating color data as part of an image-manipulation package. The color data comes as red, green, and blue components. We could define a suitable type like this:

```
type rgb = {r:int; g:int; b:int}   (* a type for colors *)
```

The values of the `rgb` type are records written using similar syntax:

```
(* Some rgb values *)
let red   : rgb = {r=255; g=0;   b=0;}
let blue  : rgb = {r=0;   g=0;   b=255;}
let green : rgb = {r=0;   g=255; b=0;}
let black : rgb = {r=0;   g=0;   b=0;}
let white : rgb = {r=255; g=255; b=255;}
```

Given one of these `rgb` values, we can access each field of the record using "dot" notation: `value.field`. For example, to write a function that averages each component of a record, we would write:

```
(* using 'dot' notation to project out components *)
(* calculate the average of two colors *)
let average_rgb (c1:rgb) (c2:rgb) : rgb =
  {r = (c1.r + c2.r) / 2;
   g = (c1.g + c2.g) / 2;
   b = (c1.b + c2.b) / 2;}
```

For example, we can calculate that:

`average_rgb red blue` $\Longrightarrow$ `{r=127; g=0; b=127}`

Because records often contain many fields, it is useful to be able to create a copy of the record that differs from the original in only a few places. The **`with`** notation for records does exactly that. It is used as follows:

```
(* using 'with' notation to copy a
   record but change one (or more) fields *)
let cyan = {blue with g=255}
let magenta = {red with b=255}
let yellow = {green with r=255}
```

For example, we have `cyan` $\Longrightarrow$ `{r=0; g=255; b=255}`, namely a copy of the `blue` value where the `g` field has been replaced by `255`. Note that the **`with`** notation encloses a record expression (like `blue`) inside curly-braces. We can create a copy with more than one field replaced by listing each changed field: `{blue with g=17; r=17}`.

In this color example, each field has the same type, but that doesn't have to be the case. For example, we might create a record of employee data by doing something like:

```
type employee = {
  name : string;
  age : int;
  salary : int;
  division : string
}
```

Chapter 14

Mutable State and Aliasing

Up to this point, we have studied programming in a style that is mostly *pure*—we have worked with tree structured data that can be defined by recursive datatypes, and we have seen how recursive functions that follow that structure can be used to compute new values from old ones. This style of programming is called *pure* because we model computation as simply producing new values from old, proceeding by *substituting* values for the identifiers that name the results of intermediate computations. We can think of substitution as simply *copying* data (though efficient implementations won't do that, of course).

Pure programs work with *immutable* values—once a value has been named, the association between the identifier and its value is never altered thereafter.[1] Pure programs are easy to reason about, since all computation can be explained by local reasoning with simple computation steps. This use of *persistent* data structures (ones that give don't change) can simplify many programming tasks, since you never have to worry about possibly destroying an old value when computing a new one. Moreover, since pure programs only *read* the data from existing values and never *modify* that data, it is easier to perform tasks in parallel—two computations running simultaneously on immutable data structures will never interfere with each other.

In contrast, most programming languages support a more *imperative* programming style, in which the program state is *mutable*, meaning that it

[1] Of course we can *shadow* an existing occurrence of a name with a new binding, but that doesn't change the old value.

can be modified *in place*. Imperative programming is extremely useful in many situations, and it can simplify code that would otherwise be difficult to implement in a pure way.

Mutable state lets us write programs that exhibit "action at a distance"—in which two remote parts of the program interact with one another by modifying a shared piece of state. Such sharing can also be used to create non-tree-like data structures that have cycles or explicitly shared subcomponents. Mutable state also allows the possibility of efficient re-use of the computer's memory, since modifying a value in place doesn't require any copying or extra space. These features combine to make it possible to implement algorithms that have strictly better space requirements or performance characteristics than their pure, immutable counterparts.

However, although mutable state is powerful, it also requires us to radically modify the model of computation that we use to reason about our programs' behaviors. Since mutable state requires us to "update a value in place" we have to explain where the "place" is that is being updated. This seemingly simple change necessitates a much more complex view of the computer's memory, and we can no longer use the simple substitution model to understand how our programs evaluate.

The new computation model, called the *abstract stack machine*, accounts for all of the new behaviors introduced by adding mutable state. These include *aliasing*, which lies at the heart of shared memory, and the non-local memory effects, which make it harder to reason about programs.

14.1 Mutable Records

To see how the "action at a distance" provided by mutable state can simplify some programming problems, let's consider a simple task. Suppose we wanted to do a performance analysis of the `delete` operation for the binary search trees we saw in §7. In particular, suppose that we want to count the number of times that the helper function `tree_max` is called, either by `delete` or via recursion.

First, let's recall the definition of these functions:

```
(* returns the maximum integer in a NONEMPTY BST t  *)
let rec tree_max (t:tree) : int =
  begin match t with
    | Empty -> failwith "tree_max called on empty tree"
    | Node(_,x,Empty) -> x
    | Node(_,_,rt) -> tree_max rt
  end

(* returns a binary search tree that has the same set of
   nodes as t except with n removed (if it's there)      *)
let rec delete (n:'a) (t:'a tree) : 'a tree =
  begin match t with
    | Empty -> Empty
    | Node(lt,x,rt) ->
        if x = n then
          begin match (lt,rt) with
            | (Empty, Empty) -> Empty
            | (Node _, Empty) -> lt
            | (Empty, Node _) -> rt
            | _ -> let m = tree_max lt in
                Node(delete m lt, m, rt)
          end
        else
          if n < x then Node(delete n lt, x, rt)
          else Node(lt, x, delete n rt)
  end
```

It isn't too hard to modify the `tree_max` function to return both the maximum value in the tree and a count of the number of times that it is called (including by itself, recursively):

```
let rec tree_max2 (t:'a tree) : 'a * int =
  begin match t with
    | Empty -> failwith "tree_max called on empty tree"
    | Node(_,x,Empty) -> (x, 1)
    | Node(_,_,rt) -> let (m, cnt) = tree_max2 rt in (m, cnt+1)
  end
```

Now to modify the `delete` function itself, we have to "thread through" this extra information about the count:

```
let rec delete2 (n:'a) (t:'a tree) : 'a tree * int =
  begin match t with
    | Empty -> (Empty, 0)
    | Node(lt,x,rt) ->
        if x = n then
          begin match (lt,rt) with
            | (Empty, Empty) -> (Empty, 0)
            | (Node _, Empty) -> (lt, 0)
            | (Empty, Node _) -> (rt, 0)
            | _ -> let (m, cnt1) = tree_max2 lt in
                     let (lt2, cnt2) = delete2 m lt in
                       (Node(lt2, m, rt), cnt1 + cnt2)
          end
        else
          if n < x then
            let (lt2, cnt) = delete2 n lt in
            (Node(lt2, x, rt), cnt)
          else
            let (rt2, cnt) = delete2 n rt in
            (Node(lt, x, rt2), cnt)
  end
```

This is a bit clunky, but it gets even worse if we consider that code that uses the `delete2` method will have to be modified to keep track of the count too. For example, before these modifications, we could have written a function that removes *all* of the elements in a list from a tree very elegantly by using `fold` like this:

```
let delete_all (l: 'a list) (t: 'a tree) : 'a tree =
  fold delete t l
```

After modifying `delete` to count the calls to `tree_max`, we now have the much more verbose:

```
let delete_all2 (l: 'a list) (t: 'a tree) : 'a tree * int =
  let combine (n:'a) (x:'a tree*int) : 'a tree * int =
    let (delete_all_tl, cnt1) = x in
    let (ans_t, cnt2) = delete2 n delete_all_tl in
      (ans_t, cnt1+cnt2)
  in
  fold combine (t,0) l
```

Ugh!

Mutable state lets us sidestep having to change `delete` and the all of the code that uses it. Instead, we can declare a global *mutable* counter,

which only needs to be incremented whenever `tree_max` is invoked. Let's see how to do this in OCaml:

```
type state = {mutable count : int}

let global : state = {count = 0}

let rec tree_max3 (t:'a tree) : 'a =
  globals.count <- globals.count + 1;     (* update the count *)
  begin match t with
    | Empty -> failwith "tree_max called on empty tree"
    | Node(_,x,Empty) -> x
    | Node(_,_,rt) -> tree_max3 rt
  end
```

Here, the type `state` is a record containing a single field called `count`. This field is marked with the keyword **`mutable`**, which indicates that the value of this field may be updated in place. We then create an instance of this record type—I have chosen to call it `global` as a reminder that this state is available to be modified or read anywhere in the remainder of the program. (We'll see how to avoid such use of global state below, see §17.)

The only change to the program we need to make is to update the `global.count` at the beginning of the `tree_max` function. OCaml uses the notation `record.field <- value` to mean that the (mutable) `field` component of the given `record` should be updated to contain `value`. Such expressions are *commands*—they return a `unit` value, so the sequencing operator '`;`' (see §12) is useful when working with imperative updates.

Neither the `delete` nor the `delete_all` functions need to be modified. At any point later in the program, we can find out how many times the `tree_max` function has been called by by simply doing `global.count`. We can reset the count at any time by simply doing `global.count <- 0`.

14.2 Aliasing: The Blessing and Curse of Mutable State

As illustrated by the example above, mutable state can drastically simplify certain programming tasks by allowing one part of the program to interact with a remove part of the program. However, this power is a double-edged sword: mutable state makes it potentially much more diffi-

cult to reason about the behavior of a program, and requires a much more sophisticated model of the computer's state to properly explain.

To illustrate the fundamental issue, consider the following example. Suppose we wanted to implement a type for tracking the coordinates of points in a 2D space. We might create a mutable datatype of points, and couple useful operations on them like this:

```
type point = {mutable x:int; mutable y:int}

(* shift a points coordinates *)
let shift (p:point) (dx:int) (dy:int) : unit =
  p.x <- p.x + dx;
  p.y <- p.y + dy

let string_of_point (p:point) : string =
  "{x=" ^ (string_of_int p.x) ^
  "; y=" ^ (string_of_int p.y) ^ "}"
```

We can now easily create some points and move them around:

```
let p1 = {x=0;y=0}
let p2 = {x=17;y=17}

;; shift p1 12 13
;; shift p2 2 4
;; print_string (string_of_point p1)   (* prints {x=12; y=13} *)
;; print_string (string_of_point p2)   (* prints {x=19; y=21} *)
```

So far, so good.

Now consider this function, which simply sets the `x` coordinates of two points and then returns the new coordinate of the first point:

```
let f (p1:point) (p2:point) : int =
  p1.x <- 17;
  p2.x <- 42;
  p1.x
```

What will this function return? The "obvious" answer is that since `p1.x` was set to `17`, the result of this function will always be `17`. But that's *wrong*! Sometimes this function can return `42`. To see why, consider this example:

```
let p = {x=0;y=0} in
  f p p    (* f called with the same point for both arguments! *)
```

Calling `f` on the same point twice causes the identifiers `p1` and `p2` men-

tioned in the body of `f` to be *aliases*—these two identifiers are *different names* for the *same* mutable record.

A more explicit way of showing the same thing is to consider the difference between these two tests:

```
(* p1 and p2 are not aliases *)
let p1 = {x=0;y=0}
let p2 = {x=0;y=0}
;; shift p2 3 4

(* this test will PASS *)
let test () : bool =
  p1.x = 0 && p1.y == 0
;; run_test "p1's coordinates haven't changed" test

(* p1 and p2 are aliases *)
let p1 = {x=0;y=0}
let p2 = p1
;; shift p2 3 4

(* this test will FAIL *)
let test () : bool =
  p1.x = 0 && p1.y == 0
;; run_test "p1's coordinates haven't changed" test
```

Aliasing like that illustrated above shows how programs with mutable state can be subtle to reason about—in general the programmer has to know something about which identifiers might be aliases in order to understand the behavior of the program. For small examples like those above, this isn't too difficult, but the problem becomes much harder as the size of the program grows. The two points passed to a function like `f` might themselves have been obtained by some complicated computation, the outcome of which might determine whether or not aliases are provided as inputs.

Such examples also motivate the need for a different model of computation, one that takes into account the "places" affected by mutable updates. If we blindly follow the substitution model that has served us so well thus far, we obtain the wrong answer! Here is an example:

```
let p1 = {x=0;y=0}
let p2 = p1       (* create an alias! *)
let ans = p2.x <- 17; p1.x
```

$\longmapsto$ *(by substituting the value for p1)*

```
let p1 = {x=0;y=0}
let p2 = {x=0;y=0}     (* alias information is lost *)
let ans = p2.x <- 17; {x=0;y=0}.x
```

$\longmapsto$ *(by substituting the value for p2)*

```
let p1 = {x=0;y=0}
let p2 = {x=0;y=0}
let ans = {x=0;y=0}.x <- 17; {x=0;y=0}.x
```

$\longmapsto$ *update the x field "in place", but we need to discard the result*

```
let p1 = {x=0;y=0}
let p2 = {x=0;y=0}
let ans = ignore({x=17;y=0}); {x=0;y=0}.x
```

$\longmapsto$

```
let p1 = {x=0;y=0}
let p2 = {x=0;y=0}
let ans =(); {x=0;y=0}.x
```

$\longmapsto$ *throw away the unit answer*

```
let p1 = {x=0;y=0}
let p2 = {x=0;y=0}
let ans = {x=0;y=0}.x
```

$\longmapsto$ *project the x field*

```
let p1 = {x=0;y=0}
let p2 = {x=0;y=0}
let ans = 0               (* WRONG *)
```

The next chapter develops a computation model, called the *abstract stack machine*, suitable for explaining the correct behavior of this and all other examples.

Chapter 15

The Abstract Stack Machine

We saw in the last chapter that the simple substitution model of computation breaks down in the presence of mutable state. This chapter presents a more detailed model of computation, called the *abstract stack machine* that lets us faithfully model programs even in the presence of mutable state. While we develop this model in the context of explaining OCaml programs, small variants of it can be used to understand the behaviors of programs written in almost any other modern programming language, including Java, C, C++, or C#. The abstract stack machine (abbreviated ASM) is therefore an important tool for thinking about software behavior.

The crucial distinction between the ASM and the simple substitution model presented earlier is that the ASM properly accounts for the *locations* of data in the computer's memory. Modeling such spatial locality is essential, precisely because mutable update modifies some part of the computer's memory in place—the notion of "where" an update occurs is therefore necessary. As we shall see, the ASM gives an accurate picture of how OCaml (and other type-safe, garbage-collected languages like Java and C#) represents data structures internally, which helps us predict how much space a program will need and how fast it will run.

Despite this added realism, the ASM is still *abstract*—it hides many of the details of the computer's actual memory structure and representation of data. The level of detail present in the ASM is chosen so that we can understand the behavior of programs in a way that doesn't depend on the underlying computer hardware, operating system, or other low-level details about memory. Such details might be needed to understand C or C++ programs for example, which aren't type-safe and require programmers to

manually manage memory allocation.

15.1 Parts of the ASM

Recall that the substitution model of computation had two basic notions:

- *values*, which are "finished" results such as integers (*e.g.* `0`,`1`, ...), tuples of values (*e.g.* `(1,(2,3))`), records (*e.g.* `{x=0; y=1}`), functions (*e.g.* `(fun (x:int) -> x + 1)`), or constructors applied to values (*e.g.* `Cons(3,Nil)`).
- *expressions*, which are "computations in progress", like `1+2*3`, `(f x)`, `p.x`, `begin match ... with ... end`, *etc.*.

The substitution model computes by simplifying an expression—that is, repeatedly substituting values for identifiers, and performing simple calculations—until no more simplification can be done.

For programs that don't use mutable state, the abstract stack machine will achieve the same net results. That is, for *pure* programs, the ASM can be thought of as a (complicated) way of implementing substitution and simplification. We didn't previously specify precisely what was meant by "substitution"; instead we relied on your intuitions about what it means to replace an identifier with a value, and just left it at that. The ASM gives an explicit algorithm for implementing substitution using a *stack*, and further refines the notion of value and computation model to keep track of where in memory the data structures reside.

There are three basic parts of the abstract stack machine model:

- The *workspace* keeps track of the expression or command that the computer is currently simplifying. As the program evaluates, the contents of the workspace change to reflect the progress being made by the computation.
- The *stack* keeps track of a sequence of *bindings* that map identifiers to their values. New bindings are added to the stack when the `let` expression is simplified. Later, when an identifier is encountered during simplification, its associated value can be found by looking in the stack. The stack also keeps track of partially simplified expressions that are waiting for the results of function calls to be computed.

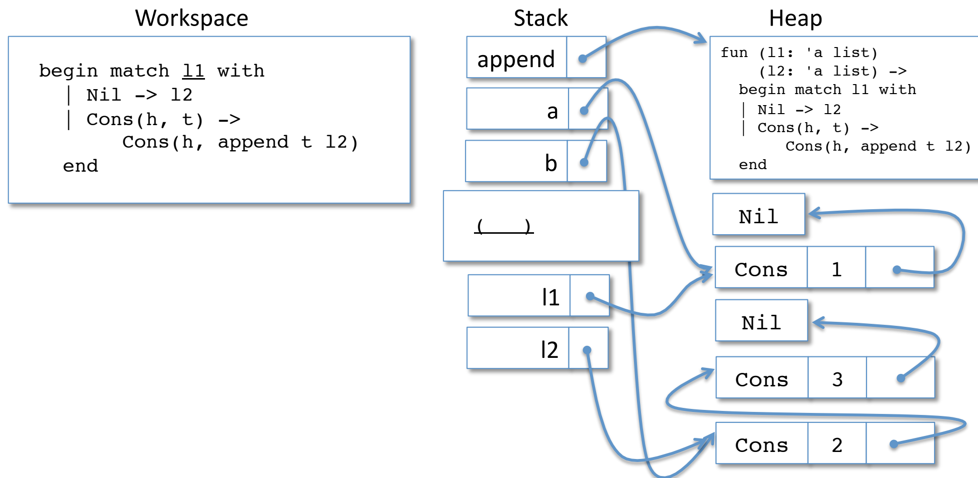


Figure 15.1: A picture of an Abstract Stack Machine in mid evaluation of a list operation `append a b`. The code in the workspace is preparing to look up the value of the local identifier `l1` in the stack (as indicated by the underline) before proceeding on to do a pattern match. The stack contains bindings for all of the identifiers introduced to this point, including `append` and the two lists `a` and `b`. It also has a saved workspace, which is there because `append` is recursive, and bindings for `l1` and `l2`. The stack values are themselves references into the heap, which stores both code (for the body of the `append` function itself), and the list structures built from `Cons` and `Nil` cells. The arrows are *references*, as explained in §15.2.

- The *heap* models the computer's memory, which is used for storage of non-primitive data values. It specifies (abstractly) where data structures reside, and shows how they reference one another.

Figure 15.1 shows these three parts of an ASM in action. The sections below explain each of these pieces in more detail and explains how they work together. First, however, we need to understand how the ASM represents its values.

15.2 Values and References to the Heap

The ASM makes a distinction between two kinds of values. *Primitive values* are integers, booleans, characters, and other "small" pieces of data

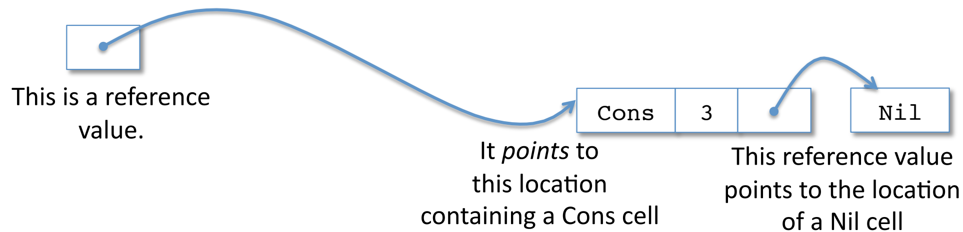


Figure 15.2: A pictorial representation of reference values.

for which the computer provides basic operations such as addition or the boolean logic operations.

All other values are references to structured data stored in the heap. A *reference* is the *address* (or *location*) of a piece of data in the heap. Pictorially, we draw a reference as an "arrow"—the start of the arrow is the reference itself (*i.e.* the address). The arrow "points" to a piece of data in the heap, which is located at the reference address. Figure 15.2 shows how we visualize reference values in the ASM.

The heap itself contains three different kinds of data:

- A *cell* is labeled by a datatype constructor (such as `Cons` or `Nil`) and contains a sequence of constructor arguments. Figure 15.2 shows two different heap cells. One is labeled with constructor name `Cons`, which has two arguments, namely `3` and a reference to the second heap cell, which is labeled `Nil` (and has no arguments). The arguments to a constructor are themselves just values (either primitive values or references.) Note that, conceptually, the constructor names like `Cons` take up some amount of space in the heap.

- A *record* contains a value for each of its fields. Unlike constructor cells, the field names of a record don't actually take up space in the heap, but we mark them anyway to help us do book keeping. We mark mutable record fields using a "doubled" box, as shown, for example, by the record of type `point` in Figure 15.3. Such a mutable field is the "place" where an "in-place" update occurs, as we shall see below.

- A *function*, which is just an anonymous function value of the form `(fun (x1:t1) ... (xn:tn) -> e)`. Figure 15.1 shows that the stack

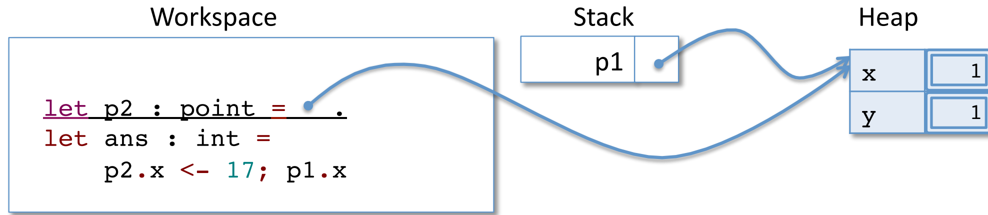


Figure 15.3: The state of the ASM just before it creates the stack binding for `p2`. Note that `p2` is an alias of `p1`—they both point to the *same* record of the heap.

binding for `append` is a reference to the code for the `append` function itself.[1]

References as Abstract Locations

What, exactly is a reference value, like the one shown in Figure 15.2? A reference is an abstract representation of a location in the heap—references are abstract because it doesn't matter exactly "where" the location being pointed to is. For the purposes of understanding aliasing and the other aspects of sharing among data structures, it is enough to know whether two references point to the same location.

We can peel back the abstraction just a bit to see how references work in the internals of the computer. Figure 15.4 shows two different views of the same situation. The right side of the figure depicts a reference value that points to a `Cons` cell containing the value `3` and a reference to `Nil`, exactly the same configuration as shown in Figure 15.1. The left-hand side gives a lower-level explanation in terms of the computer's memory.

On a 32-bit machine, we can think of the computer's memory as being a giant array of 32-bit words, where the array is indexed by numbers in the range 0 to $2^{32} - 1$ (which happens to be 4,294,967,295). In this low-level view, an address is simply a number, and we have to encode heap cells and other data structures by choosing some particular representation by using patterns of of 32-bit words. Such decisions are made during compilation. For example, the compiler might decide that the `Cons` tag of a memory cell should be represented using the number 120120120 and that `Nil` should

[1] We will have to refine this notion of heap-allocated functions slightly to account for *local* functions. See 17.1 for details.

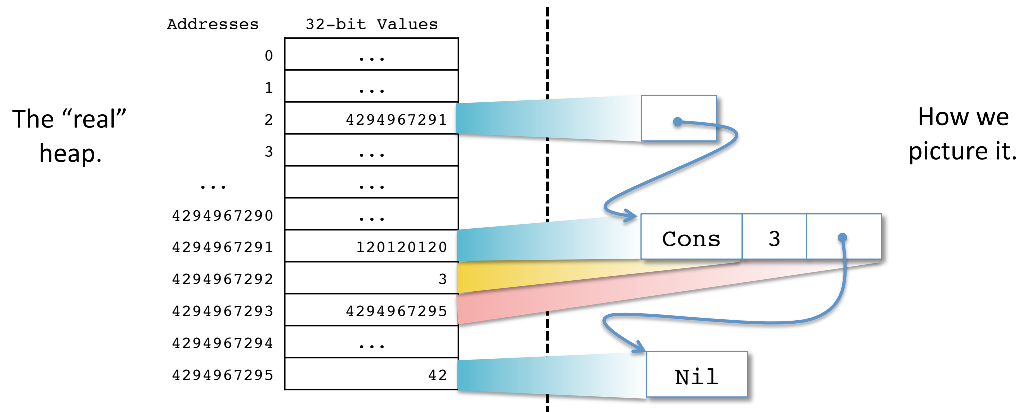


Figure 15.4: The "real" computer memory (left) is just an array of 32-bit words, each of which has an address given by the index into the array. The ASM provides an abstract view of this memory (right) in which the exact address of a piece of heap-allocated data is hidden. Constructor tags are just arbitrary 32-bit integers chosen by the compiler—in this example, the tag for `Cons` is 120120120 and the tag for `Nil` is 42. Constructor data is laid out contiguously in memory, and a reference is just the address of some other word of memory. Again, the ASM hides these representation details.

be represented by the tag 42, as depicted in Figure 15.4. The ASM hides these insignificant details, so we don't have to worry about them as we try to explain the behavior of our programs.[2]

Some languages, like C or C++, use the low-level, numeric view of memory references, in which case they are usually called *pointers*. The distinction is that references are *abstract*—the only operations supported by references are reference creation, dereferencing (that is, getting the value referred to by the reference), and determining whether two references point to the same heap location (reference equality). In contrast, pointers correspond directly to machine addresses, which are just numbers (as shown in Figure 15.4). Therefore, in languages like C or C++ you can do "pointer arithmetic" to, for example, calculate an offset from a pointer. This can be very useful for low-level programming, but can also lead to a lot of serious bugs if you have errors in your arithmetic calculations. In the general computer science vernacular, the terms "reference" and "pointer"

[2]Even this low-level explanation sweeps a lot of details under the rug—the operating system and hardware collaborate to provide the illusion that there are $2^{32} - 1$ words of memory available to a given process, though, strictly speaking, not all of it is immediately available. Also portions of the memory space are reserved for OS-related tasks, I/O, *etc.*

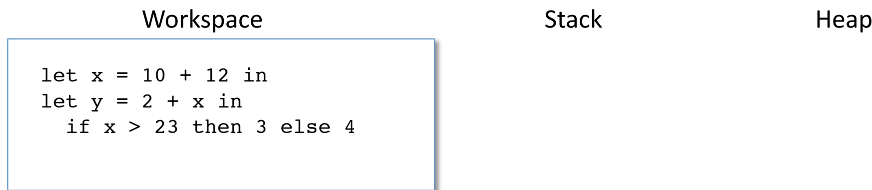


Figure 15.5: The initial state of the ASM starts with the workspace containing the entire program, and the stack and heap both empty.

are often used interchangeably, with some potential for confusion.

15.3 Simplification in the ASM

The abstract state machine processes a program by repeatedly simplifying the contents of the workspace until the workspace contains only a value, which is the answer computed by the program. During this process, the ASM creates new bindings in the stack, allocates data structures in the heap, and, in the case of mutable update, modifies the contents of mutable record fields.

In the start configuration of the ASM, the workspace contains the entire program to be executed, and the stack and heap are both empty.

At each step of simplification, the ASM finds the first (left-most) "ready subexpression" and simplifies it according to computation rules determined by the program expression. The intuition is that the "ready" expression is the one that will be simplified next. In our pictures of the ASM, we underline the ready expression.

As an example, consider simulating the ASM on this program:

```
let x = 10 + 12 in
let y = 2 + x in
  if x > 23 then 3 else 4
```

Figure 15.5 shows the initial configuration of the ASM for this example.

The basic rules for simplification are:

- An expression involving a primitive operator (like +) is ready if all of its arguments are values. Primitive operations are simplified by

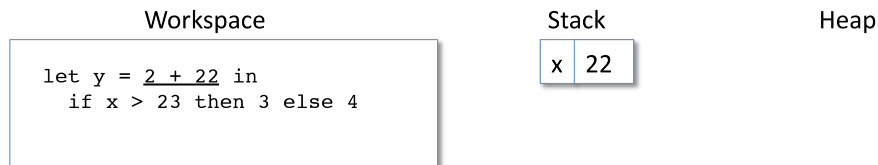


Figure 15.6: The example from Figure 15.5 after several steps of simplification. There is a binding of `x` to `22` in the stack, and the expression `2 + 22` is ready to simplify next.

replacing the expression with its result. This is exactly as we have seen previously. For example, the expression `10 + 12` ⟹ `22`, so we replace `10 + 12` by `22`.

- A `let` expression `let x : t = e in body` is ready if `e` is a value. It is simplified by adding (pushing) a new binding for the identifier `x` to `e` at the *end* of the stack, and leaving `body` in the workspace.
- A variable is always ready. It is simplified by looking up the binding associated with the variable in the stack and replacing the variable by the corresponding value. This lookup process proceeds by searching from the most recent bindings to the least recent—this algorithm guarantees that we find the newest definition for a given variable.
- A conditional expression `if e then e1 else e2` is ready if `e` is either `true` or `false`. It is simplified by replacing the workspace with either `e1` or `e2` as appropriate.

Figure 15.6 shows the example program after several steps of simplification according to these rules. Slides 15–31 from Lecture 12 show the full sequence of simplification steps for this example, which results in computing the value `4`.

Shadowing and the Stack

The ASM stack properly accounts for shadowing variables (recall §2.4), as can be seen by using it to evaluate this program:

```
let x = 22 in
let x = 2 + x in
  if x > 23 then 3 else 4
```

Figure 15.7: An example of how proper shadowing is implemented in the ASM. The value of the ready variable `x` will be found by searching the stack from most recent (`x` maps to 24) to least recent (`x` maps to 22).

Figure 15.7 shows the state of the ASM when the variable `x` of the conditional expression is ready to be looked up in the stack. There are two bindings for `x`, but the most recent one (*i.e.* the one closest to the bottom) will be used, which is consistent with shadowing.

The sequence of bindings is called a *stack* because we only ever *push* elements on to the stack and then *pop* them off in a last-in-first-out (LIFO) manner. This is just like a stack of dinner plates—it's easy to put more on top or take away the last one put there, but it's hard to remove one in the middle. (For some reason, computer scientist's stacks are often drawn with their "top" toward the bottom of the page as we show in the ASM diagrams; this is bizarre, but at least consistent with trees having their leaves at the bottom and roots at the top.)

Treating **`let`**-bound identifiers using a stack discipline ensures that the most recently defined value for a given identifier name will be used during the computation. Below we will see how the ASM pops bindings as part of returning a value computed by a function call.

Simplifying Datatype Constructors and `match`

The simplification rules mentioned above don't yet involve the heap—they just implement substitution by explicitly using a stack of identifier bindings. Data constructors, like `Cons` and `Nil`, on the other hand, do interact with the heap—simplifying a constructor allocates some space on the heap and creates a reference to the newly allocated space. The simplification rule is therefore quite simple:

- A constructor (like `Cons` or `Nil`) is ready if all of its arguments are values. A constructor expression is simplified by allocating a heap

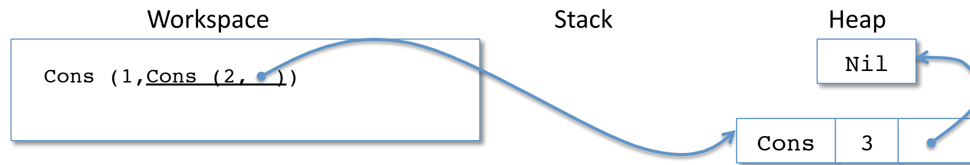


Figure 15.8: The ASM in the process of evaluating the expression `1::2::3::[]`, which we could rewrite equivalently as `Cons(1, Cons(2, Cons(3, Nil)))`. Here, the `Nil` and `Cons(3,_)` cells have already been allocated in the heap, and the ASM is ready to allocate the `Cons(2,_)` cell.

cell with the constructor tag and its arguments as data and then replacing the constructor expression with this newly created reference.

Figure 15.8 shows the ASM part way through evaluating the list expression `1::2::3::[]`. Each use of the `::` operator causes the ASM to allocate a new `Cons` cell in the heap. Slides 51–60 of Lecture 12 show the full animation of the ASM for this example.

Simplifying a **`match`** expression of the following shape is pretty straight forward:

```
begin match e with
  | pat1 -> branch1
  | ...
  | patN -> branchN
end
```

- Such a **`match`** expression is ready to simplify if `e` is a value (which will typically be a reference to a cell in the heap). The **`match`** is simplified by finding the first pattern starting from `pat1` and working toward `patN` that structurally matches the the heap cell referred to by `e`. Once such a matching pattern, `patX` is found, the ASM adds new stack bindings for each identifier in the pattern—the values to which those identifiers are bound is determined by the shape of the heap structure. The workspace is then modified to contain `branchX`, the branch body corresponding to `patX`. If no such pattern matches, the ASM aborts the computation with a `Match_failure` error.

The `append` example from the Lecture 12 slides shows in detail the use of pattern match simplification.

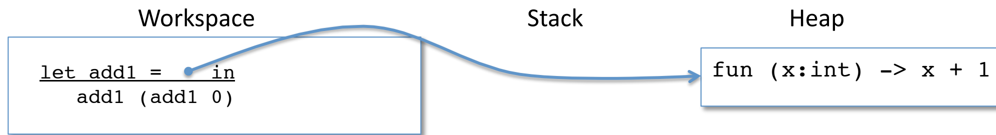


Figure 15.9: The ASM after moving a function value to the heap. It is ready to create a binding, named `add1` to the reference.

Simplifying functions

There are three parts to simplifying functions: moving function values to the heap, *calling* a function, and *returning* a value computed by a function to some enclosing workspace.

The first of these steps is very straight forward. Recall from §10.1 that top-level function declarations are just short hand for naming an anonymous function. For example, the following are equivalent ways of defining `add1`:

```
let add1 (x:int) : int = x + 1 in
  add1 (add1 0)
```

and

```
let add1 = fun (x:int) : int -> x + 1 in
  add1 (add1 0)
```

The ASM therefore simplifies the first expression to the second before proceeding. Since anonymous functions like `(fun (x:int) : int -> x + 1)` are one of the three kinds of heap data (see §15.2), the ASM just moves the `fun` value to the heap and replaces the `fun` expression with the newly created reference. The ASM then follows the usual rules for `let` simplification to create a binding for the function name on the stack. Figure 15.9 shows how the example program above looks after following these simplification rules.

Simplifying function call expressions is more difficult. The issue is that, in general, the function call may itself be nested within a larger expression that will require further simplification after the function returns its computed value. To model this situation faithfully, the ASM must therefore keep track of where in the surrounding expression the value computed by a function call should be returned to once the function is done.

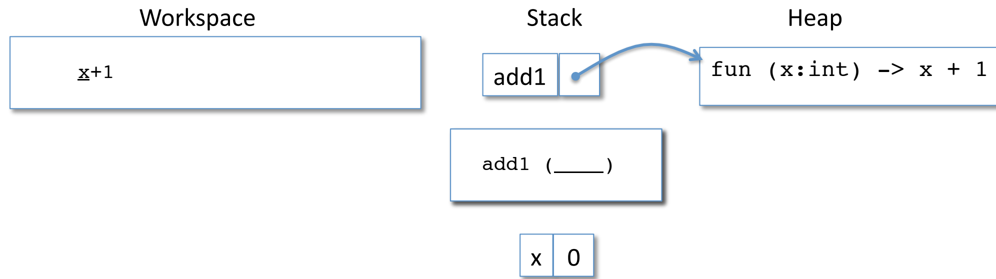


Figure 15.10: The ASM just after the call to the inner `add1` has been simplified. The stack contains a saved workspace whose hole marks where the answer of this function call should be returned. It also contains a binding for the `add1` function's argument `x`.

In the example program above, after creating a binding for `add1` on the stack, we reach a situation in which the worskspace contains the expression `add1 (add1 0)`. We can simplify the innermost `add1` by looking up the reference in the stack as usual. Then we're ready to do the function call—in general, a function call is ready to simplify if the function is a reference and all of its arguments are values. Suppose that we magically knew that the answer computed by `add1 0` was going to be a value `ANSWER`. Then the we should eventually simplify workspace by replacing inner function call, `(add1 0)`, with `ANSWER` to obtain a new workspace `add1 ANSWER`.

To achieve that goal, the ASM simplifies a function call like this:

First it saves the current workspace to the stack, marking the spot where the `ANSWER` should go with a "hole". In our example, since the original workspace was `ans1 (ans1 0)`, the saved workspace when doing the inner call will be `ans1 (____)`, where the ____ marks the "hole" to which the answer will be returned.

Second, the ASM adds new stack bindings for each of the called function's parameters. Suppose that the function being called has the shape **`fun`** `(x1:t1) ... (xN:tN) -> body` in the heap and it is being called with the arguments `v1` ... `vK`.[3] Then there will be stack bindings added for `x1` ↦ `v1` ... `xJ` ↦ `vJ`. In our running example, the `add1` function takes only one argument called `x`, so only one binding will be added to the stack.

Third, the workspace is replaced by the body of the function.[4]

---

[3]In general there can be *fewer* arguments than the function requires, which corresponds to the case of partial application.

[4]In the case of partial application, the workspace is replaced by a **`fun`** expression of the

Figure 15.10 shows the state of the ASM just after the inner call to `add1` has been simplified.

Once the function call has been initiated and the function body is in the workspace, simplification continues as usual. This process may involve adding more bindings to the stack, doing yet more function calls, or allocating new data structures in the heap.

Assuming that the code of the function body eventually terminates with some value, the ASM is in a situation in which the result of the function should be returned as the `ANSWER` to the corresponding workspace that was saved on the stack at the time that the function was called. For example, after a few steps of simplification, the workspace in Figure 15.10 will contain only the value `1`, which is the answer of the inner call to `add1`.

When this happens—that is, when the workspace contains a value and there is at least on saved workspace on the stack—the ASM returns the value to the old workspace. It does so by *popping* (*i.e.* removing) all of the stack bindings that have been introduced since the last workspace was saved on the stack. It then replaces the current workspace (which just contains some value `v`) with the last saved workspace (and also popping it from the stack), replacing the answer "hole" with the value `v`.

In our running example, after the workspace in Figure 15.10 simplifies to the value `1`, the ASM will pop the binding for `x` from the stack, restore the workspace to `add1 1` (where the hole has been replaced by `1`), and then pop the saved workspace. Simplification then proceeds as usual.

Lecture 12 contains an extended animation of the ASM simplification for a more complex example, which shows how to run the following program:

```
let rec append (l1: 'a list) (l2: 'a list) : 'a list =
  begin match l1 with
  | Nil -> l2
  | Cons(h, t) -> Cons(h, append t l2)
  end in

let a = Cons(1, Nil) in
let b = Cons(2, Cons(3, Nil)) in

append a b
```

form `fun (xL:tL) .. (xN:tN) -> body`, where `L` is `J+1`. To properly continue simplification will, in this case, require the use of a closure. See §17.1.

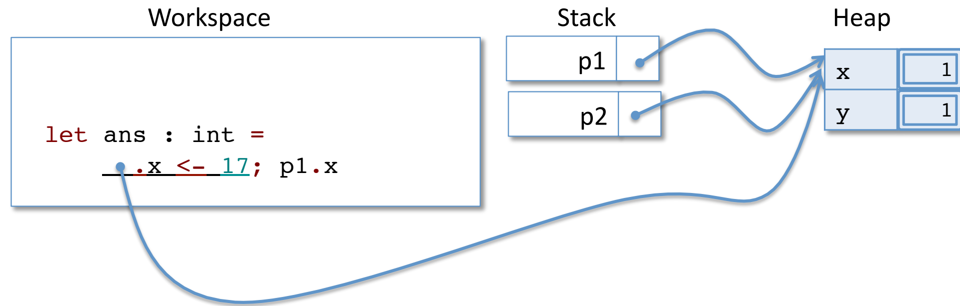


Figure 15.11: The state of the ASM just before doing the imperative update to the `p2.x` field. Note that `p1` and `p2` are aliases—they point to the same record in the heap.

Simplifying Mutable Record Operations

The whole purpose of the ASM is to allow us to explain the behavior of programs that use mutable state. We are finally to the point where we can make sense of "in place" updates to mutable record fields, and we can use the ASM to explain the behavior of programs that exhibit aliasing.

The rules for simplifying records, field projections, and mutable field updates are simple:

- A record expression is ready if each of its fields is a value. We simplify a record by allocating a record structure in the heap and replacing the record expression with a reference to that structure. Recall that we mark the mutable fields of a record using a double-lined box; this is simply to help us visualize which parts of the heap might be modified during a program's execution.

- A record field projection expression, like `p.x` is ready of `p` is a value. It is simplified by replacing the expression by the value stored in the `x` field of the record in the heap.

- A mutable field update expression, like `p.x <- e` is ready if both `p` and `e` are values. The expression is simplified by changing the `x` field of the record pointed to by `p` to contain the value `e` instead of whatever it contained before. The entire update expression is replaced by the `unit` value, `()`.

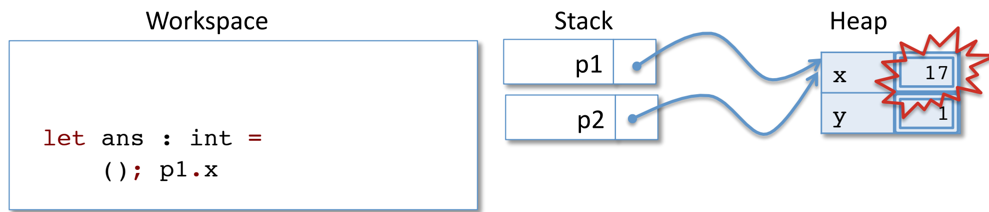


Figure 15.12: The ASM after doing the imperative update shown in Figure 15.11. It is clear that running `p1.x` from this state will yield `17`.

We can now see how the ASM correct the deficiencies of the substitution model. Recall the following example, which computed to the incorrect answer `0` when we used the substitution model (see §14.2)

```
let p1 = {x=0;y=0}
let p2 = p1        (* create an alias! *)
let ans = p2.x <- 17; p1.x
```

After creating the stack bindings for `p1` and `p2`, the resulting ASM configuration will be that shown in Figure 15.11. It is clear that `p1` and `p2` are *aliases*—they point to the same record in the heap. Modifying the `x` field via the `p2` reference, therefore also modifies the `x` field of the `p1` reference—they are the *same* field. Figure 15.12 shows the effect of doing the update; it is clear that the ASM will eventually compute `17` as the (correct) answer for this program.

Chapter 16

Linked Structures: Queues

In this chapter, we consider how to use mutable state to build "imperative linked" data structures in the heap. Why would we want to do that? The pure datatypes, like lists and trees, that we have studied so far all have a simple recursive structure: we build "bigger" structures out of already existing "smaller" structures. A consequence of that way of structuring data is that we can't easily modify "distant" parts of the data structure. For example, when we implement the `snoc` function, which adds an element to the end of a list, we were forced to do this:

```
let rec snoc (x:'a) (l:'a list) : 'a list =
  begin match l with
    | [] -> [x]
    | h::tl -> h::(snoc x tl)
  end
```

If we examine the behavior of the function call `snoc v l` using the ASM, we can see that `snoc` *copies* the entire list `l` (because it uses the `::` operator) after creating a new list to hold the value `v`. This means that adding a single element to the tail of the list costs time proportional to the length of the list, and, since `l` is duplicated, twice as much heap space will be used.

Sometimes this persistent nature of pure lists is useful—for example, if we needed to use both `l` and the result of `snoc v l`, we might have to create the copy anyway (which always takes time proportional to the length of `l`). However, in many situations it would be more efficient and simpler to just be able to add an element to the tail of the list directly. We can do that by creating a data structure that is similar to a list, but that uses mu-

table references to link together nodes. By keeping *two* references, one to the head and one to the tail, we can then efficiently update the structure at either end.

The resulting structure is called a *queue* because one common mode of use is to *enqueue* (add) elements to the tail of the queue and *dequeue* (remove) them from the head. (Think of people waiting in line for concert tickets.) Queues are used in many situations where lists can be used, but they also serve as a key component in "work list" algorithms (where they keep track of tasks remaining to be done), networking applications (where they buffer requests to be handled on a first-come, first-served basis), and search algorithms (where they keep track of what part of some space to explore next).

The design criteria above suggest that we use the following interface for a queue module:

```
module type QUEUE = sig
  (* type of the data structure *)
  type 'a queue

  (* Make a new, empty queue *)
  val create : unit -> 'a queue

  (* Determine if the queue is empty *)
  val is_empty : 'a queue -> bool

  (* Add a value to the tail of the queue *)
  val enq : 'a -> 'a queue -> unit

  (* Remove the head value and return it (if any) *)
  val deq : 'a queue -> 'a
end
```

16.1 Representing Queues

How do we implement mutable queues? We need two data types—one to store the data of the "internal" nodes that form the list-like structure, and one containing the references to the head and tail of the queue. The nodes of the queue will be linked together via references, but since the last element of the queue isn't followed by a next element, those references should be optional. Similarly, in an empty queue, there are no nodes for

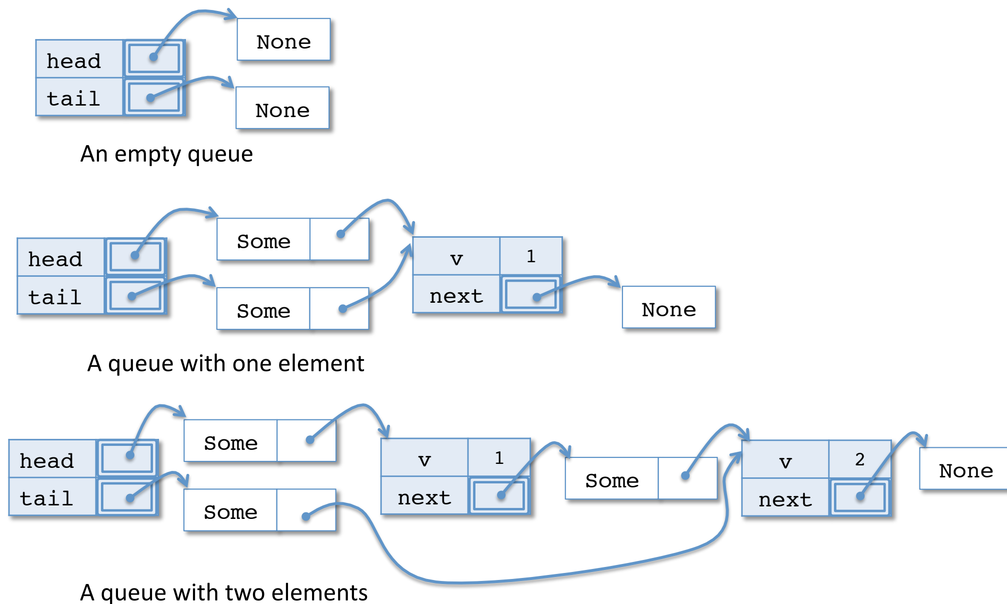


Figure 16.1: Several example queues as they would appear in the heap. Note that the frequent use of options motivates the need for some "visual" shorthand. See Figure 16.2 for a more compact way of drawing such linked structures.

the head and tail to refer to, so they must also be optional. These considerations lead us to define the queue representation types like this:

```
module Q : QUEUE = struct

  type 'a qnode = {
     v : 'a;
     mutable next : 'a qnode option;
  }

  type 'a queue = {
     mutable head : 'a qnode option;
     mutable tail : 'a qnode option;
  }

  ...
end
```

Figure 16.1 shows several examples of queue values as they would appear in the heap of the ASM. As shown in these examples, the proliferation

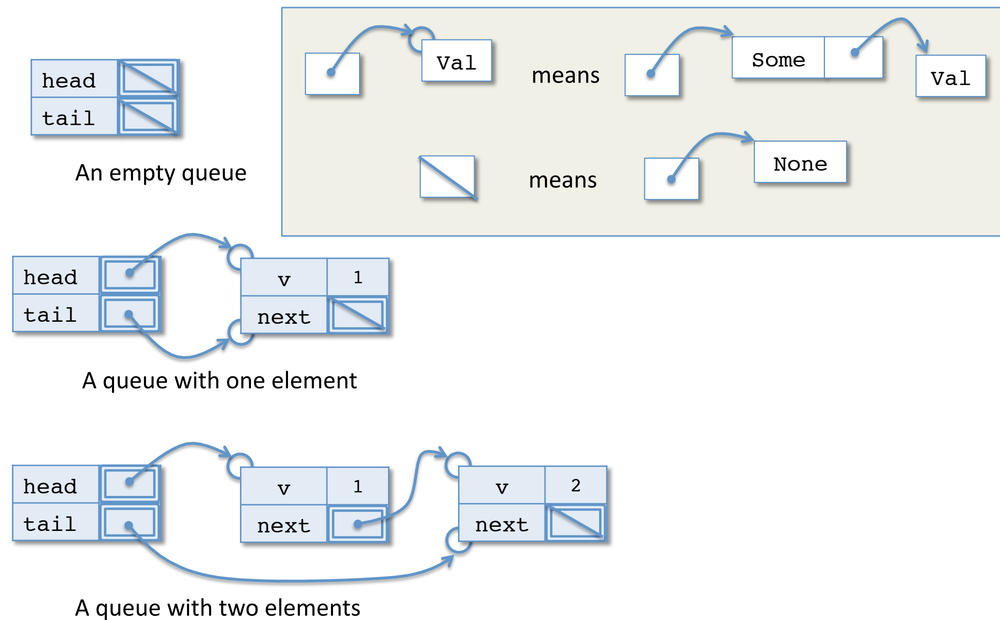


Figure 16.2: The queue structures from Figure 16.1 drawn using a visual short-hand in which references to `None` are represented by a slash and references to `Some v` are drawn as an arrow to a "`Some` bubble", which gives a "handle" to the underlying value `v`.

of `Some` and `None` constructors in the heap creates a lot of visual clutter. Although it is necessary to acknowledge their existence (especially since the *type* of a reference to `Some v` is different from that of a reference to just `v` itself), it is useful to present these drawings in a more compact way. Figure 16.2 shows the same three queue structures represented using "visual shorthand" for `None` and `Some` constructors in the heap.

As you can see, the basic structure of a queue is a linear sequence of `qnodes`, each of which has a `next` pointer to its successor. The tail element of the queue has its `next` field set to `None`. For the empty queue, both the head and tail are `None`, for a singleton queue, the head and tail both point to the same `qnode`, and for a queue with more than one element, the head and tail point to the first and last elements, respectively.

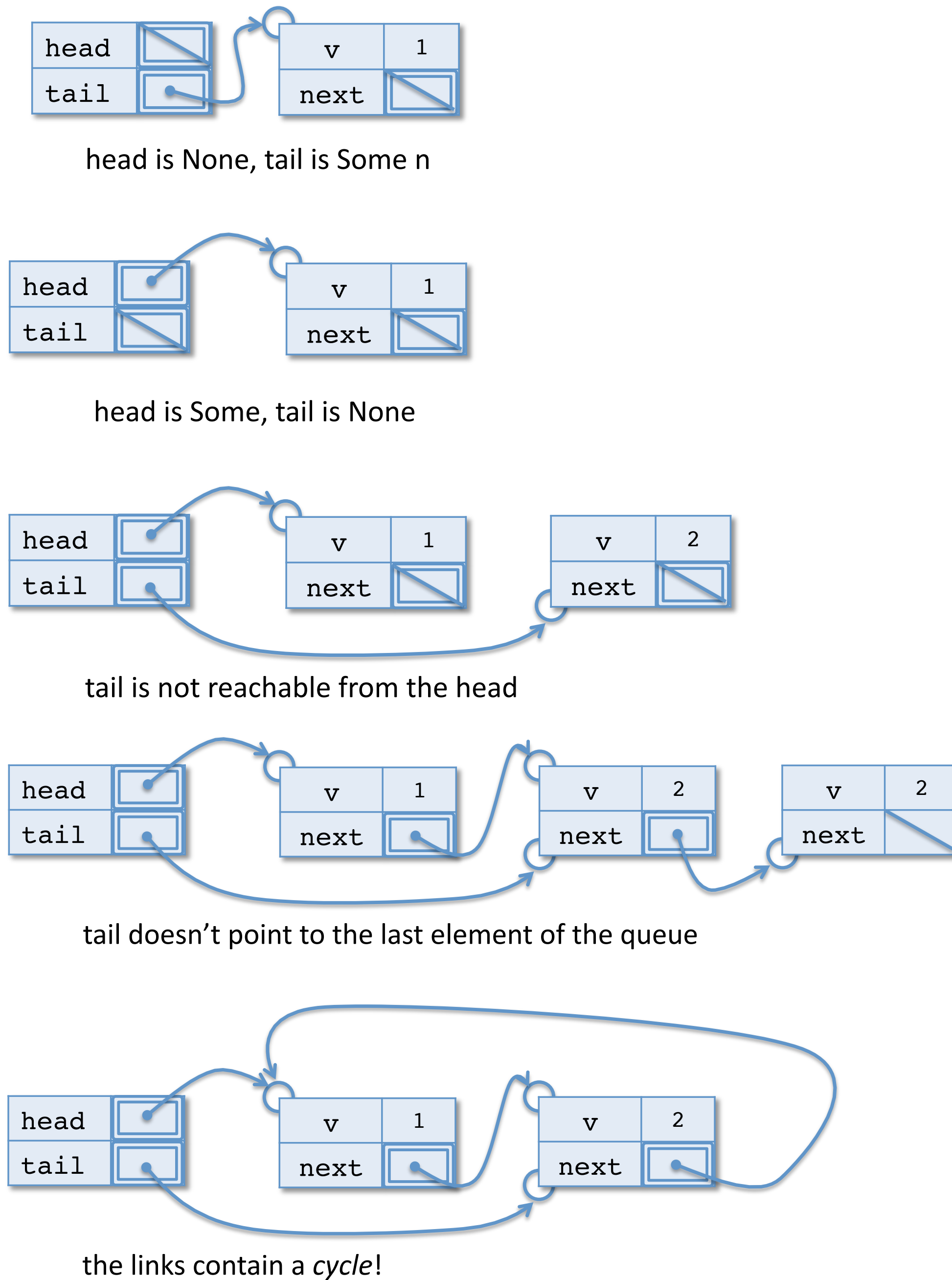


Figure 16.3: Several examples of bogus heap structures that conform to the queue datatypes. The queue invariants rule out all of these examples (and many more).

16.2 The Queue Invariants

Although the two types defined above provide enough structure to let us create heap values with the desired shapes, they are *too permissive*—there are many values that conform to these types that don't meet our expectations of what a proper queue should be. Figure 16.3 shows several examples of such bogus queue structures as they might appear in the heap.

This situation is similar to the one we encountered when studying binary search trees. There, the type of `'a trees` was rich enough to contain trees with arbitrary nodes, but we found that by imposing additional structure in the form of the binary search tree invariant (see Definition 7.1), we could constrain the shape of trees in a way so that the natural ordering of its nodes' data can be exploited to drastically improve search.

For queues, we would therefore like to impose some restrictions that rule out the bogus values but preserve all of the "real" queues.

Definition 16.1 (Queue Invariant)**.**
A data structure of type `'a queue` *satisfies the queue invariants if (and only if), either*

1. *both* `head` *and* `tail` *are* `None`*, or,*
2. `head` *is* `Some n1` *and* `tail` *is* `Some n2`*, and*
   - `n2` *is reachable by following* `next` *pointers from* `n1`
   - `n2.next` *is* `None`

The first part of the invariant ensures that there is a unique way of representing the empty queue. The second part applies if the queue is non-empty. It says that `tail` actually points to the tail element, and that this `n2` is reachable from the head. Together these latter invariants imply that there are no cycles in the link structure and that the queue itself is "connected" in the sense that all nodes are reachable from the `head`.

It is easy to verify that each of the bogus queue examples from Figure 16.3 can be ruled out by these invariants as ill-formed. Therefore, as long as we are careful that our queue manipulation functions establish these invariants when creating queues and preserve them when modifying an existing queue, we may also assume that any queues passed in to the `Q` module already conform to the invariants. As always, this reasoning

is justified because the type `'a queue` is *abstract*—the type definition is not exported in the module interface (see §9).

16.3 Implementing the basic Queue operations

Now that we understand the `'a queue` type and its invariants, it is not too difficult to implement the operations required by the `QUEUE` interface. Creating a fresh empty queue is easy, as is determining whether a given queue is empty:

```
(* create an empty queue *)
let create () : 'a queue =
  { head = None;
    tail = None }

(* determine whether a queue is empty *)
let is_empty (q:'a queue) : bool =
  q.head = None
```

Note that, due to the queue invariants, we could have equally well chosen to check whether `q.tail = None` in `is_empty`, and we never have to check both. By assumption, either both `q.head` and `q.tail` are `None` or neither is.

How do we add an element at the tail of the queue? If the queue is empty, we simple create a new internal queue node and then update both the `head` and `tail` pointers to refer to it. If the queue is non-empty, then we need to create a new queue node. By virtue of the fact that it will be the new tail, we know that its `next` pointer should be `None`. To maintain the queue invariants, we must also modify the old tail node's `next` field to point to the newly created node:

```
(* add an element to the tail of a queue *)
let enq (x: 'a) (q: 'a queue) : unit =
  let newnode = {v=x; next=None} in
  begin match q.tail with
     | None ->
         (* Note that the invariant tells us that q.head
            is also None *)
         q.head <- Some newnode;
         q.tail <- Some newnode
     | Some n ->
         n.next <- Some newnode;
         q.tail <- Some newnode
  end
```

Note that `enq` returns `unit` as its result—this function is a command that imperatively (destructively) modifies an existing queue. After the update, the old queue is no longer available.

Removing an element from the front of the queue is almost as easy. If the queue is empty, the function simply fails. Otherwise, head is adjusted to point to the next node in the sequence. One might therefore think that the correct implementation of `deq` is:

```
(* BROKEN attempt to remove an element from the head
   of the queue *)
let deq (q: 'a queue) : 'a =
  begin match q.head with
    | None ->
        failwith "deq called on empty queue"
    | Some n ->
        q.head <- n.next;
        n.v
  end
```

However, this implementation fails to re-establish the queue invariants: in the case that there is exactly one element in the queue, which gets removed, the `tail` pointer should be set to `None`—otherwise `head` will be `None` and the `tail` will still be `Some`. The correct implementation must therefore check for that possibility and adjust the `tail` accordingly:

```
(* remove an element from the head of the queue *)
let deq (q: 'a queue) : 'a =
  begin match q.head with
    | None ->
        failwith "deq called on empty queue"
    | Some n ->
        q.head <- n.next;
        if n.next = None then q.tail <- None;
        n.v
  end
```

In general, when manipulating linked, heap-allocated data structures, it is important to keep in mind the invariants that make sense of the structure. Understanding the invariants can help you structure your programs in a way that helps you get them right.

16.4 Iteration and Tail Calls

Suppose that we want to extend the queue interface to include some operations that work with the queue as a whole, rather than just with one node at a time. A simple example of such a function is the `length` operation, which counts the number of elements in the queue:

```
module type QUEUE =
sig
  (* type of the data structure *)
  type 'a queue

  |

  (* Get the length of the queue *)
  val length : 'a queue -> int
end
```

One simple way to implement this function is to directly translate the recursive definition of `length` that we are familiar with for immutable lists to the datatype of queues. Since the "linked" structure of the queues is given by the sequence of `qnode` values and the queue itself consists of just the `head` and `tail` pointers, we need to decompose the length operation into two pieces: one part that uses recursion as usual to process the `qnode`s and one part that deals with the actual `queue` type itself. (Note that this is

yet another place where the structure of the types guides the code that we write.)

We can therefore write `length` like this:

```
(* Calculate the length of the list using recursion *)
let length (q:'a queue) : int =
  let rec loop (no: 'a qnode option) : int =
    begin match no with
      | None -> 0
      | Some n -> 1 + (loop n.next)
    end
  in
  loop q.head
```

In this code, the helper function `loop` recursively follows the sequence of `qnode` values along their `next` pointers until the last node is found. Note that this function implicitly assumes that the queue invariants hold, since a cycle in the `next` pointers would cause the program to go into an infinite loop.

Although this program will compute the right answer, it is still somehow unsatisfactory: if we observe the behavior of a call to this version of `length` by using the ASM, we can see that the height of the stack portion of the ASM state is proportional to the length of the queue—each recursive call to `loop` will save a copy of the workspace consisting of `1 + (____)` and create a new binding for the copy of `no`. The slides for Lecture 15 give a complete animation of the ASM for such an example.[1]

It seems awfully wasteful to use up so much space just to count the number of elements in the queue—why not just talk down the queue, keeping a running total of the number of nodes we've seen so far? This version of `length` requires a slight modification to the `loop` helper function to allow it to keep track of the count:

[1]To be fair, this *same* criticism applies to the recursive version of `length` for immutable lists that we studied earlier.

```
(* Calculate the length of the list using iteration *)
let length (q:'a queue) : int =
  let rec loop (no:'a qnode option) (len:int) : int =
    begin match no with
      | None -> len
      | Some n -> loop n.next (1+len)
    end
  in
  loop q.head 0
```

Why is this better? At first glance, it seems that we are stuck with the same problems as with the recursive version above. However, note that in this version of the code, the workspace pushed on to the stack at the recursive call to `loop` will always be of the form `(____)`. That is, after the call to `loop` in the `Some n` branch of the match, there is no more work left to be done (in contrast, the first version always pushed workspaces of the form `1 + (____)`, which has a pending addition operation). If we watch the behavior of this program in the ASM, we see that the ASM will always perform a long series of stack pops, restoring these empty workspaces when `loop` finally returns the `len` value in the `None` branch.

The observation that we will always immediately pop the stack after restoring an empty workspace suggests a way to optimize the ASM behavior: there is no need to push an empty workspace when we do such a function call since it will be immediately popped anyway. Moreover, at the time when we would have pushed an empty workspace, we can also eagerly *pop* any stack bindings created since the last non-empty workspace was pushed. Why? Since the workspace is empty, we know that none of those stack bindings will be needed again.

We say that a function call that would result in pushing an empty workspace is in a *tail call*[2] position. The ASM optimizes such functions as described above—it doesn't push the (empty) workspace, and it eagerly pops off stack bindings. This process is called *tail call optimization*, and it is frequently used in functional programming.

Why is tail call optimization such a big deal? It effectively turns *re-*

[2]The word "tail" in this definition is not to be confused with the `tail` of a queue. The "tail" in tail call means that the function call occurs at the "end" of the function body. Note, however, that very often when writing loops over queues you will want a tail call whose argument is towards the `tail` of the queue, relative to the "current" node being visited.

cursion into *iteration*. Imperative programming languages include `for` and `while` loops that are used to iterate a fragment of code multiple times. The essence of such iteration is that only a constant amount of stack space is used and that parts of the program state are updated each time around the loop, both to accumulate an answer and to determine when the loop should finish.

In the iterative version of the queue `length` function shown above, the extra `len` argument is incremented at each call. If we see how the ASM stack behaves when using tail call optimizations, each such call to `loop` essentially *modifies* the stack bindings in place. That is, at every call to `loop`, there will be one stack binding for `no` and one stack binding for `len`[3]. Since, under tail call optimizations, those two old bindings will be popped and the two new bindings for `no` and `len` will be pushed each time `loop` is called, the net effect is the *same* as imperatively updating the stack. The Lecture 15 slides give a detailed animation of this behavior in the ASM.

The upshot is that writing a `loop` using tail calls is *exactly the same* as writing a `while` loop in a language like Java, C, or C++. Indeed, tail calls subsume even the `break` and `continue` keywords that are used to "jump out" of `while` loops—the `break` keyword corresponds to just returning an answer from the `loop`, and the `continue` keyword corresponds to calling `loop` in a tail-call position.

16.5 Loop-the-loop: Examples of Iteration

Let's see how we can use tail recursion to write several different functions that iterate over the queue structures.

First, let's just create a simple `print` operation that outputs the queue values on the terminal in a nicely formatted way:

[3]And, technically, one for the `loop` function itself, which is saved in the *closure* for the `loop` code when it is moved to the heap.

```
(* output the queue elements in order from head to
   tail to the terminal *)
let print (q:'a queue) (string_of_element:'a -> string) : unit =
  let rec loop (no: 'a qnode option) : unit =
    begin match no with
      | None -> ()
      | Some n -> print_endline (string_of_element n.v);
                       loop n.next
    end
  in
    print_endline "--- queue contents ---";
    loop q.head;
    print_endline "--- end of queue -----"
```

In this example, the `loop` doesn't produce any interesting output—it simply walks down the queue nodes, converts each value to a string, and prints out that string. Note that the call to `loop` in the `Some n` branch is in a tail-call position: the `print_endline` will be be done before the `loop`. The `print` function enters the `loop` by calling `loop q.head`—this means that the `loop` will start traversing the `queue` from the `head`.

Next, let's try writing a function that converts a queue to a list. Here the `loop` function will return an `'a list`, which is built up as the `loop` traverses over the sequence of nodes. Therefore the `loop` function itself needs an extra argument in which to "accumulate" this answer.

Our first instinct might be to do this:

```
(* Retrieve the list of values stored in the queue,
   ordered from head to tail. *)
let to_list (q: 'a queue) : 'a list =
  let rec loop (no: 'a qnode option) (l:'a list) : 'a list =
      begin match no with
      | None -> l
      | Some n -> loop n.next (l @ [n.v])
      end
  in loop q.head []
```

Since we are building up the accumulator list `l` from the head of the queue to the tail, we need to add each node's value to the *end* of `l` (as shown by the use of `@` in the recursive call). When `loop` has reached the end of the sequence of nodes (the `None` case), it simply returns the resulting list. Note that we provide the empty list `[]` as the second argument when we start the `loop`—this says that the "initial value" of the accumulator list

is `[]`.

However, this implementation isn't that great because, as we saw before the `append` operation takes time proportional to the length of `l`. Since we use it on `l`, which increases in length each time around the loop, this algorithm will end up taking roughly n^2 time, where n is the length of the queue.

A better way to structure this program is to build up the accumulator list in reverse order during the `loop`, and then reverse the entire thing only once at the end. This leads us to this variant, which will take time proportional to the length of the queue:

```
(* Retrieve the list of values stored in the queue,
   ordered from head to tail. *)
let to_list (q: 'a queue) : 'a list =
  let rec loop (no: 'a qnode option) (l:'a list) : 'a list =
      begin match no with
      | None -> List.rev l
      | Some n -> loop n.next (n.v::l)
      end
  in loop q.head []
```

Here we simply cons the node value to the front of the accumulator list `l`, but then use the library call `List.rev` to reverse the list when the loop is done (in the `None` branch).

We can rewrite many of the list-processing functions that used recursion to take advantage of tail-recursive loops. For example, we can sum the elements of an `int queue` using this function:

```
(* Sum the elements of the queue *)
let sum (q: int queue) : int =
  let rec loop (no: int qnode option) (sum:int) : int =
    begin match no with
      | None -> sum
      | Some n -> loop n.next (sum + n.v)
    end
  in loop q.head 0
```

Again the accumulator that changes at each iteration of the loop is the running total, here called `sum`.[4] When the `loop` terminates, we simply return the `sum`, at each iteration we increase the `sum` parameter by `n.v`. As

[4] In fact the term "accumulator" comes from thinking of the extra argument of the `loop` function as a "running total" begin "accumulated".

before, we have to initialize the value of `sum` to `0` by calling the `loop` on `q.head` and `0`.

It is instructive to compare these iterative functions to the recursive ones we are already familiar with. For example, here is the recursive version of `sum_list` that we have seen previously, where I have used suggestive names for the head and tail of the list:

```
let rec sum_list (l:int list) : int =
  begin match l with
    | [] -> 0
    | v::next -> v + (sum_list next)
  end
```

We can see that in the recursive solution the "base case" computes the length of the empty list, which is `0`. In contrast, the iterative version initializes the "accumulator" to `0` in the call to `loop`—the base case is analogous to the initial value of the accumulator. The recursive version does the addition *after* recursively computing the sum of the "next" part of the list, while the iterative version does the addition *before* it jumps back to the start of the loop.

16.6 Infinite Loops

When using iteration, it is possible to accidentally cause your program to go into an infinite loop. Unlike the case for recursion, however, an infinite loop may just "diverge silently"—since the loop doesn't consume any stack space, a loop might not exhaust all of the available memory. Infinite recursion usually causes OCaml to produce the error message:

```
Stack overflow during evaluation (looping recursion?)
```

This is possible because the operating system can detect that the program has used up all of its stack space and abort the program.

When using tail recursion, one could accidentally create an infinite loop by writing a program like this:

```
(* Accidentally go into an infinite loop *)
let accidental_infinite_loop (q:'a queue) : int =
    let rec loop (qn:'a qnode option) (len:int) : int =
      begin match qn with
        | None -> len
        | Some n -> loop qn (len + 1)
      end
    in loop q.head 0
```

This program mistakenly calls `loop` with the *same* `qn` each time in the body of the `Some` branch. When we run it on a non-empty queue, the program will just hang, producing no output and no error message. This kind of error can be frustrating to recognize—make sure that your programs actually terminate when you run them.

A second way in which an iterative program can go into an infinite loop is if the link structure being traversed contains a cycle. This could occur, for example, if a value of type `'a queue` that doesn't satisfy the queue invariants is passed to a function that expects the queue invariants to hold.

In some circumstances, we can plan for the possibility of cyclic data structures and check to see whether the function has detected a cycle. For example, suppose that you wanted to write a function that, given a possibly invalid queue (*i.e.* one that doesn't necessarily satisfy the queue invariant) returns the last node that can be reached by following `next` pointers from the `head`. (You might want to implement such a function to *check* whether a given queue satisfies the invariants—the last element reachable from the head should be pointed to by the `tail`.)

Here is a function that accomplishes this task:

```
(* get the tail (if any) from a possibly invalid queue *)
let rec get_tail (q: 'a queue) : 'a qnode option =
    let rec loop (qn: 'a qnode) (seen: 'a qnode list)
         : 'a qnode option =
      begin match qn.next with
        | None -> Some qn
        | Some n ->
           if contains_alias n seen then None
           else loop n (qn::seen)
      end
    in loop q.head []
```

This function relies on a helper function, called `contains_alias`[5], which, given a value and a list determines whether the list contains any aliases of the value. The loop traverses nodes starting from the head and adds each one it passes to the `seen` accumulator—if it ever encounters the same node twice the queue structure must have contained a cycle, so the result is `None`. Otherwise, the traversal will eventually find a node whose `next` field is `None`, which is the last element reachable from the `head`.

If we tried to write this function in the naive way shown below, calling it with a invalid cyclic queue would cause the program to loop silently:

```
(* BROKEN: this version of get_tail could loop *)
let rec get_tail (q: 'a queue) : 'a qnode option =
    let rec loop (qn: 'a qnode) : 'a qnode option =
      begin match qn.next with
        | None -> Some qn
        | Some n -> loop n
      end
    in loop q.head
```

[5]See HW05.

Chapter 17

Local State

This Chapter explores some ways of packaging state with functions that operate on that state. Such encapsulation, or *local* state, provides a way to restrict some parts of the program from tampering with mutable values. By *sharing* a piece of local state among several functions, we can profitably exploit the "action at a distance" that mutable references provide.

Recall the motivating example from §14 in which a global identifier with the mutable field `count` let us neatly keep track of how many times the `tree_max` function had been called. There are several drawbacks of using a single, global reference. First, if we want to have multiple different counters, each of which is used to track the usage of a different function, we have to name each of the counters at the top level of the program. What's worse, each function that uses such a counter would need to have to be modified in a slightly different way—we might have to write `global1.count <- global1.count + 1` in one function and the nearly identical `global2.count <- global2.count +1` in another function. Supposing that we might also sometimes want to decrement the counters, or reset them to `0` at various points throughout the program, keeping track of which `global` identifier to use can quickly become quite a hassle.

The key idea in solving this problem is to use first-class functions combined with *local* mutable state. As a first step, lets first isolate the main functionality of a counter. The code below shows an `incr` function, which (for now) increases the count of the global counter by `1` and then returns the new count:

```
type state = {mutable count : int}

let global = {count = 0}

let incr () : unit =
  global.count <- global.count + 1;
  global.count
```

How do first-class functions and local state apply here? The problem is that to have more than one counter, we need to generate a new mutable `state` record for each counter instance. We can do that by making a function that creates the state and then *returns* the `incr` function that updates the state:

```
type state = {mutable count : int}

let mk_incr () : unit -> int =
  let ctr = {count = 0} in
  let incr () =
    ctr.count <- ctr.count + 1;
    ctr.count
  in
  incr
```

Each time we call the `mk_incr` function, the result will be to create a new counter record `ctr` and then return a function for incrementing that counter. For example, if were to run the following program, we would get two `incr` functions, each with its own local counter:

```
let incr1 : unit -> int = mk_incr ()
let _ = incr1 ()
let incr2 : unit -> int = mk_incr ()
```

17.1 Closures

To understand how a call to the `mk_incr` function evaluates, we need to make one slight refinement to the ASM model of §15—the reason has to deal with returning a function that refers to locally-defined identifiers. In our particular example, the `incr` function returned by `mk_incr` refers to `ctr`, which is *local* to `mk_incr`. Therefore, the `ctr` binding will be created on the stack during the evaluation of a call to `mk_incr`, but that binding

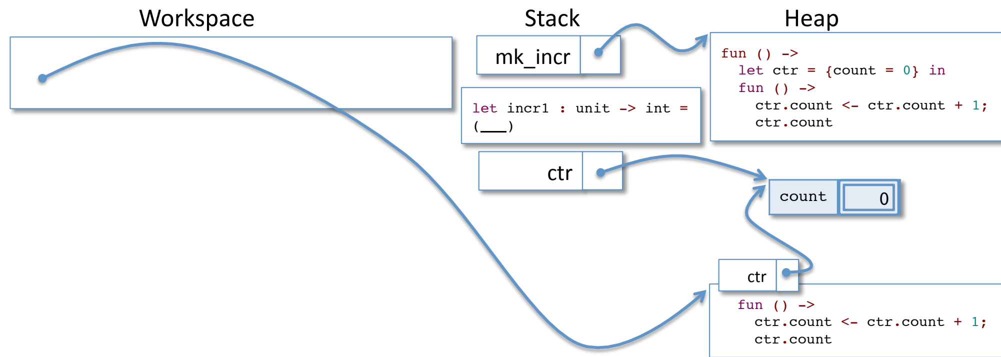


Figure 17.1: This ASM shows how the local function declared in `mk_incr` stores its own local copy of the stack bindings needed to evaluate its body. Here, since the `incr` function uses `ctr`, its *closure* contains a copy of that stack binding.

will be popped off at the point where `mk_incr` returns!

To remedy this problem, when the ASM stores a function value to the heap, it also stores any of the local stack bindings that might be needed during the evaluation of a call to that function. In this example, since the body of `incr` refers to `ctr`, the ASM stores a local copy of the stack binding for `ctr` with the function data itself. Figure 17.1 shows the state of the ASM at the point just after `incr` has been stored to the heap but before `mk_incr` returns. When `ctr` is popped off the stack, the function `incr` will still be able to access its value via the locally saved binding. This combination of a function with some local bindings is called a *closure*. As we shall see, closures are intimately related to *objects* of an object-oriented language like Java.

When do the bindings associated with a closure get used? They are needed to evaluate the body of the function, so, whenever a function invocation occurs, any local bindings stored in the function closure are copied back to the stack before the function's argument bindings are created.

Therefore, when we the program above calls `incr1 ()`, the `ctr` value will be copied back on to the stack before the body of the function is executed. This ensure that when the body mentions `ctr`, there is always the appropriate binding on the stack.

Moreover, since each copy of the function has its *own* closure bindings, multiple calls to `mk_incr` will result in distinct closures, each with different, local copies of their own counter records. We can see this by looking at the

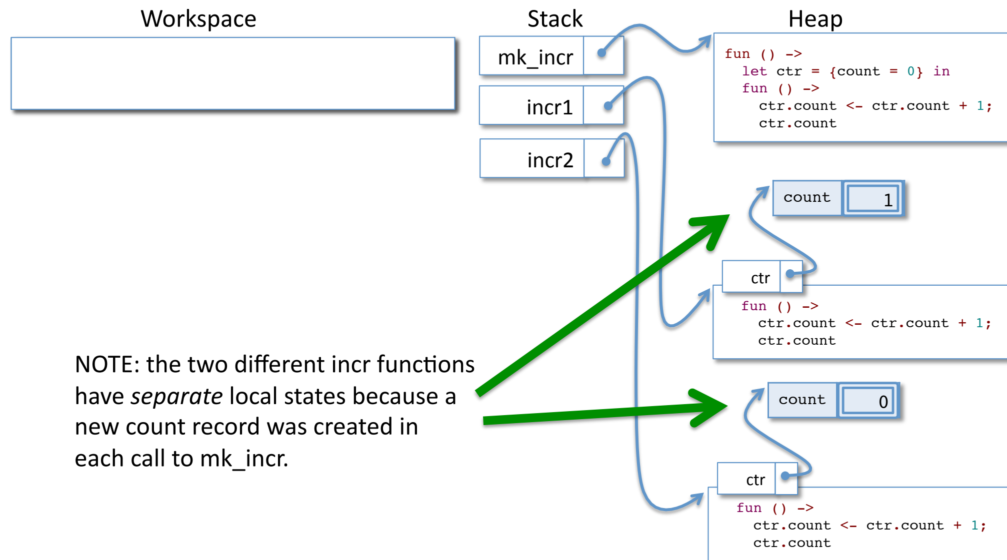


Figure 17.2: The resulting ASM configuration after two calls to `mk_incr`. Each closure has its own local copy of `ctr`. Note also that the *only* way to modify the state of the `ctr` record is to invoke the function—the state is effectively isolated from the rest of the program. This property is called *encapsulation* of state.

state of the ASM after running the second call to `mk_incr`. As Figure 17.2 shows, there are two closures, each with its own copy of `ctr`.

Another important aspect of using local state in this way is that the *only* way to access a `ctr` record is by invoking the corresponding `incr` function. This means that no other part of the program can accidentally tamper with the value stored in the counter. This kind of isolation of one part of the computer's state from other parts of the program is called *encapsulation*. If the state inside the counter object was more complex and required certain invariants to be maintained, restricting access to only a small number of functions means that we only have to ensure that they preserve the invariants.

17.2 Objects

A second step toward solving the problem of reusable counters is to define a better interface for counters. For example, we might want to decouple

incrementing the counter from reading its current value, or we might want to add the ability to decrement and reset the counter. This leads us to define `incr`, `decr`, `get`, and `reset` operations. It is straightforward to share a global reference among several functions in this way, as shown here:

```
type state = {mutable count : int}

let global = {count = 0}

let incr () : unit =
  global.count <- global.count + 1

let decr () : unit =
  global.count <- global.count - 1

let get () : int =
  global.count

let reset () : unit =
  global.count <- 0
```

This is a better interface—we now have a suite of operations that are suitable for manipulating one counter, but it still doesn't allow us to conveniently work with more than one such counter.

The solution to is to package these operations together in a record and, instead of using `mk_incr` to create one function, use a function called `mk_counter` that generates all of the functions in one go:

```
type counter = {
  incr : unit -> unit;
  decr : unit -> unit;
  get : unit -> int;
  reset : unit -> unit;
}

let mk_counter () : counter =
  let ctr : state = {count = 0} in
  {incr = (fun () -> ctr.state <- ctr.state + 1);
   decr = (fun () -> ctr.state <- ctr.state - 1);
   get = (fun () -> ctr.state);
   reset =  (fun () -> ctr.state <- 0);
  }

let ctr1 : counter = mk_counter () in
let ctr2 : counter = mk_counter () in
;; ctr1.incr ();
;; ctr2.incr ();
;; ctr1.incr ();
let ans : int = ctr1.get ()
```

17.3 The generic `'a ref` type

Situations like the counter example above, in which only a single mutable field is needed, arise often in programming. For example, you might need a mutable `bool` flag that indicates whether some feature of your program should be enabled, or perhaps you need a mutable `string` to keep track of some data being entered by the user into a text box.

While we could define a new record type specifically for each of these situations (just as we defined the `state`) type for the counters example), that would quickly become tedious and difficult to work with.

Instead, we can simply define a generic type of "single field" mutable records like this:

```
type 'a ref = {mutable contents : 'a}
```

This type is pronounced "ref" (as in *reference*), and it comes built in to OCaml. Working with the `'a ref` type is so common that OCaml also provides syntactic sugar for creating, updating, and getting the value stored

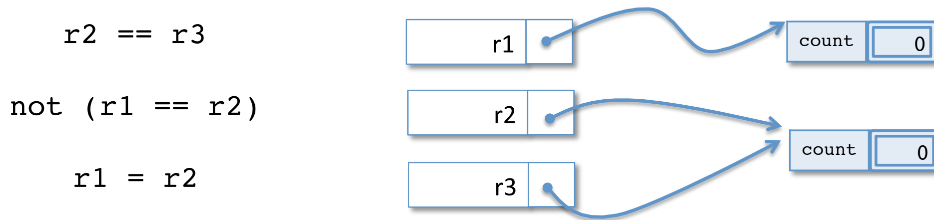


Figure 17.3: The difference between *reference* (`==`) and *structural* `=` equality. Reference equality simply checks whether two references point to the same heap location. Structural equality (recursively) compares the two values to see whether all of their contents are the same.

in the `contents` field of a `'a ref` value. These syntactic abbreviations are:

`ref e`	*means*	`{contents = e}`
`!e`	*means*	`e.contents`
`e := v`	*means*	`e.contents <- v`

As an example of using these syntactic abbreviations, we could have written the `mk_counter` function from above as the following, with equivalent results:

```
let mk_counter () : counter =
  let ctr : int ref = ref 0 in
  {incr = (fun () -> ctr := (!ctr) + 1);
   decr = (fun () -> ctr := (!ctr) - 1);
   get = (fun () -> !ctr);
   reset =  (fun () -> ctr := 0);
  }
```

17.4 Reference (==) vs. Structural Equality (=)

We have seen the importance of understanding aliasing when reasoning about heap-allocated mutable data structures. One natural problem is thus to determine whether two reference values are in fact aliases. OCaml, like all modern programming languages, provides an operation, written `v1 == v2` that yields `true` when `v1` and `v2` are aliases and `false` if they are not. This kind of equality, for obvious reasons, is called *reference equality*.

In contrast, the expression `v1 = v2` checks whether `v1` and `v2` are *structurally equal*. This means that `=` will, in general, traverse the two structures `v1` and `v2` comparing whether they agree on the values of their primitive

datatype consituent pieces *and* whether the contents of any references are (recursively) structurally equal. Figure 17.3 shows, pictorially, the difference between these two types of equality.

These two types of equality are useful in different circumstances. To summarize:

Structural equality:

- Recursively traverses the structure of the data, comparing the two values' components to make sure they contain the same data.
- May go into an infinite loop if the data structure contains cycles.
- Never considers one function value to be equal to another (even to *iself*).
- Is generally the right kind of equality to use for immutable data.

Reference equality:

- Determines whether two reference values are aliases, or whether primitive values are identical, and never traverses link structure.
- Will say that a function reference is equal only to itself.
- Otherwise equates strictly fewer things than structural equality.
- Is usually the right kind of equality to use for comparing mutable data.

Chapter 18

Wrapping up OCaml: Designing a GUI Library

18.1 Taking Stock

Thus far, we have studied program design "in the small", where the programs we have written are at most a couple of functions, and their use is relatively straightforward. We have studied a general design strategy for developing software, which starts with understanding the problem and then uses *types* and *tests* to further refine that understanding before we actually develop code.

Throughout our studies, we have used OCaml's features to explore different ways of structuring data. First we used pure representations like lists and trees, where the primary way of processing the data is via recursive functions. Then we looked at imperative data structures, such as the queues of the last chapter, where the primary modes of operation are iteration and imperative update. Along the way, we saw many other kinds of structured data: tuples, options, records, functions, *etc.*., which give us tools for thinking about how to decompose the data of a problem into an appropriate form for computing with it. We have also encountered several styles of abstraction—hiding of detail—that can help when structuring larger programs, including generic types and functions, the idea of an abstract type implemented in a module, and the use of hidden (or encapsulated) state of an object.

In subsequent chapters, we will explore these concepts again, this time

from the point of view of Java programming, where we will see that all of the same ideas apply. Here, however, we investigate how to put together all of the tools we developed in OCaml to produce a useful tool, namely a (rudimentary) paint program. Implementing such a paint program isn't that difficult, given the appropriate library for graphical user interface (GUI) design. We make this design process more interesting by developing the GUI library too—that is, we start from OCaml's native graphics library, which doesn't provide any of the familiar components (like buttons, text boxes, scroll bars, *etc.*.) that are used to create a GUI application like the paint program. On top of that simple graphics library, we will build a useable GUI library, modeled after Java's Swing libraries.

"[The OCaml GUI/Paint project was my favorite because] I really enjoyed understanding how graphics works. I thought building up a graphics library from class lectures and on my own was both a very exciting and great learning experience. We don't usually get to see the inner working of libraries but this project afforded me the opportunity to understand what was going on lower level in the code with few abstractions." — Anonymous CIS 120 Student

There are several reasons for going through this design exercise:

- It demonstrates that, even with just the programming techniques we have studied so far, we can build a pretty serious application.
- It illustrates a more complex design process than what we have seen so far.
- As we shall see, that design process leads to the *event-driven model* of reactive programming, which can be applied in many different contexts.
- It motivates why there are features, such as classes and inheritance, in object-oriented languages like Java.
- It shows how a real GUI library, like Java's Swing, works.

18.2 The Paint Application

As a first step towards designing a GUI library, let us consider an example application that might be built using such a library.

Figure 18.1 shows an example of the kind of paint program that we are aiming to build with the techniques presented in this Chapter.[1] We are

[1] We will get started with the design, and leave the rest up to the project associated with this part of the course.

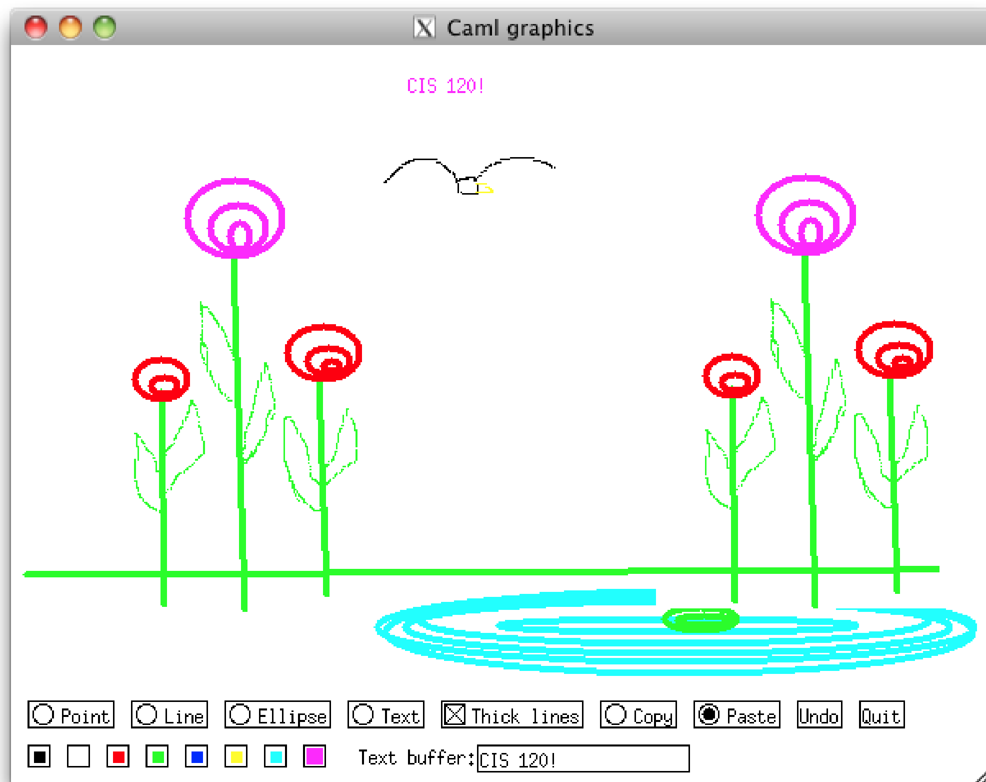


Figure 18.1: An example of a paint program built using the GUI developed in this Chapter.

all familiar with such simple paint programs, which let the user draw pictures with the mouse by clicking the button create lines, ovals, rectangles, and other basic shapes. As the picture shows, this GUI application has "buttons" (like Undo and Quit), "checkboxes" (like Thick Lines), "radio buttons" (like those used for Point, Line, Ellipse, and Text), a "text entry field", custom buttons for color selection, and a "canvas" area on which the user draws his or her picture. These and many other kinds of *widgets* are ubiquitous in applications developed for user interaction.

18.3 OCaml's Graphics Library

What do we need to do to build a library that lets us create buttons, text boxes, *etc.* for use in the paint application? First, let's take a look at the OCaml graphics library, to see what we have have to work with.[2]

Note: *To compile a program that uses the graphics library, be sure to include* `graphics.cma` *(for bytecode compilation) or* `graphics.cmxa` *(for native compilation) as a flag to the* `ocamlc` *compiler.*

Among other things, OCaml's graphics library provides basic functions for creating a new window, `open_graph`, erasing the picture displayed in the window, `clear_graph`, setting the window's size `resize_window`, and getting the current window's width, `size_x`, and height, `size_y`.

The graphics library provides the type `color`, and a variety of predefined color values like `black`, `white`, `red`, `blue`, *etc.*. Importantly, the library manages the graphics window in a stateful way—there are notions of the current "pen color", which can be set using `set_color`, and a the current "pen location", which can be changed by using `move_to`. You can draw a single pixel with the current color at coordinates (x,y) by using `plot x y`. Similarly, you can draw in the current color starting at the current pen location and ending at the coordinates (x,y) by using `line_to x y`. In a similar vein, there are functions for drawing rectangles, ellipses, arcs, and filled versions of these shapes. There are also functions for adjusting the line width used to draw the shapes, ways to put text at a certain location in the window, and work with bitmap images.

So much for drawing things on the screen. Examining the graphics library further, we see that it also provides a type called `event`, whose values describe the the mouse and keyboard. An `event` is just a record that indicates the current coordinates of the mouse, whether its button is up or down, whether a key is pressed, and which key. There is also a function that causes the program to wait for a new event to be generated by the user—moving the mouse, pressing the mouse button, or pressing a key.

The graphics library also supports a technique called "double buffering", which is used to prevent flicker when changing or animating parts

[2]See the graphics library documentation at `http://caml.inria.fr/pub/docs/manual-ocaml/libref/Graphics.html`.

of the window's displayed graphics. The idea is that, when double buffering is enabled, the graphics drawing commands affect a second copy of the window, which is not displayed on the screen. After the entire window is updated, the `synchronize` function causes the hidden buffer to be displayed (and re-uses the displayed memory space for subsequent drawing). This prevents flicker that would be caused by changing the graphics displayed in the window as they are drawn piece-meal using the primitive graphics operations.

What the graphics library does not provide is any kind of higher-level abstraction like "button" or "text box"—our GUI library will have to implement these features by using the graphics primitives.

18.4 Design of the GUI Library

How do we go from simple drawing operations and rudimentary information about the mouse and keyboard to a full-blown GUI application like the paint program? There are several issues to consider. It's clear that one of the jobs of a GUI library is to provide easy ways to construct buttons and other widgets that a program like paint can use. Since the typical GUI application involves lots of different kinds of buttons, we must consider how to share the code that is common to all (or at least many) buttons—for example, we might want a button to have an associated string (like "Undo" or "Quit") that is displayed as its label. Traditionally, GUI widgets like buttons include a rectangular border (or other visual cue) that separates the button from other regions of the window. One design challenge is thus how to conveniently package the visual attributes of widgets so that they can be reused.

A related issue is how those buttons and other widgets are positioned on the the screen relative to one another. Given the drawing primitives in the graphics library, we could imagine painstakingly drawing each line of every button using the "global" coordinate system of the window, but this would be extremely tedious and very brittle—one small change to how we want to layout the buttons of the application might prompt large, intertwined changes to the program responsible for drawing the entire window. Moreover, it's not clear how we would write that program so that, for example, just one part of the window could easily be changed (for example to put an X in a checkbox widget).

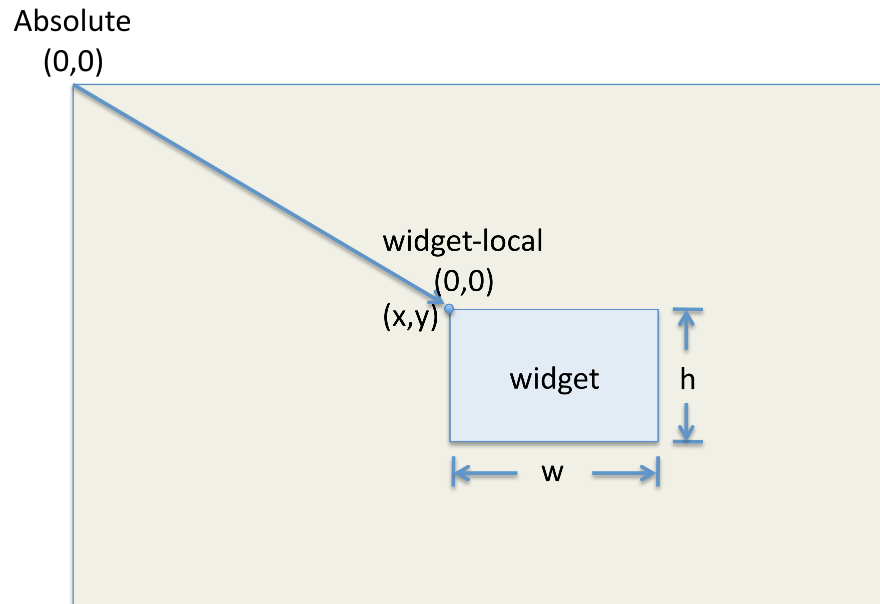


Figure 18.2: A widget draws itself in terms of widget-local coordinates, which are made relative to the global coordinate system by using a `Gctxt.t` value, which, among other things, contains the (x,y) "offset" from the origin of the graphics window. Here the grey region represents the entire graphics window . Each widget also keeps track of its width and height, which are needed for computing layour information.

Finally, there is the issue of how to process the user-generated mouse and keyboard inputs. This also turns out to be related to how the widgets are arranged in the window, since the GUI library will need to determine, based on the coordinates of the mouse click, which widget the user intended to interact with. For example, when a user clicks on a part of the window occupied by a button widget, our GUI library will have to somehow determine that the button was clicked. Moreover, since each button will typically trigger a different behavior, each button should somehow be associated with a piece of code that gets run when the mouse click occurs.

The next several sections tackle each of these problems in turn.

18.5 Localizing Coordinates

The first challenge we'll tackle is how to structure the code for drawing widget components so that it can be reused. The key idea is to make each widget in charge of drawing itself, but arrange it so that any positional information used to draw the widget can be specified using a *widget-local* coordinate system. The code for drawing a widget can therefore be written as though the widget is always located with its upper-left corner located at (0,0); in reality, the commands used to draw the widget will transparently be translated by some offset (x,y) from the actual origin of the window. Figure 18.2 shows this situation pictorially.

To implement this idea, we create a module called `Gctx` (for "graphics context") whose main type `t` represents the "contextual information" needed to draw a widget. For now, that context is just the (x,y) offset from the upper-left corner of the window. The main functionality provided by the `Gctx` module is to translate between the widget-local coordinates used by the widget drawing function and the coordinate system used by the OCaml graphics module.

There's one other discrepancy between the global coordinate system provided by the OCaml graphics library and the one we'd prefer for GUI applications: the graphics library uses Cartesian coordinates where the origin (0,0) is located at the lower-left corner of the graphics window, and the y axis increases upwards in the vertical direction. This set up is handy for plotting mathematical functions—it agrees with the usual way we think of the coordinate system when we graph a function in algebra or calculus. Unfortunately, locating (0,0) in the lower-left is not very good for GUI applications. The issue is that when a user resizes the window, the typical GUI behavior is to make more space available to the application. Importantly, the menu bars, etc. that appear at the top of the screen should remain fixed—the extra space obtained by resizing the window should appear at the bottom of the window, not the top. This means that for GUI applications, the global origin should be located in the top-left corner of the window.

These considerations lead us to a design in which the `Gctx` module provides a type `t` that stores the position relative to which a widget should be drawn. Its `x` and `y` coordinates are with respect to the GUI-global origin (0,0), located at the top-left corner of the graphics window, and for which `y` values increase downwards in the vertical direction. The `Gctx`

module also provides functions that translate from widget-local to OCaml graphics coordinates, relative to the `Gctx.t` offset. The `Gctx` module also "wraps" each of the primitive OCaml graphics routines to do the translation from widget-local to OCaml coordinates—this means that all of the widget drawing code can be written in a position-independent way.

Since the OCaml graphics library also maintains a current pen color, it is useful to add a color component to the `Gctx.t`, which will allow our GUI library to (potentially) define a widget's visual appearance relative to the color in addition to making them relative to their position in the window. The `Gctx` drawing operations therefore also set the OCaml graphics pen color accordingly.

Here is a sample of the resulting code for the `Gctx` module (the full implementation wraps more drawing routines than just the ones shown below):

```
(* Gctx.ml *)

type t = {
  x:int; y:int;           (* offset from (0,0) in GUI coords *)
  color:Graphics.color;   (* the current pen color *)
  }

(* This function takes widget-local coordinates (x,y) to OCaml
   graphics coordinates, relative to the graphics context. *)
let ocaml_coords (g:t) ((x,y):int*int) : (int*int) =
  (g.x + x, Graphics.size_y()-(g.y + y))

(* This function takes OCaml Graphics coordinates (x,y)
   to widget-local graphics coordinates, relative to the
   graphics context *)
let local_coords (g:t) ((x,y):int*int) : (int*int) =
  (x - g.x, (Graphics.size_y () - y) - g.y)

type color = Graphics.color

(* Produce a new Gctx.t with a different pen color *)
let set_color (g:t) (c:color) : t =
  {g with color=c}

(* Set the OCaml graphics library's internal state so that it
   agrees with the Gctx settings. *)
let set_graphics_state (g:t) : unit =
  Graphics.set_color g.color

(* Each of these functions takes inputs in widget-local
   coordinates, converts them to OCaml coordinates, and then
   draws the appropriate shape.  *)
let draw_line (g:t) (p1:int*int) (p2:int*int) : unit =
  set_graphics_state g;
  let (x1,y1) = ocaml_coords g p1 in
  let (x2,y2) = ocaml_coords g p2 in
    Graphics.moveto x1 y1;
    Graphics.lineto x2 y2

let draw_string (g:t) (p:int*int) (s:string) : unit =
  set_graphics_state g;
  let (_, height) = Graphics.text_size s in
  let (x,y) = ocaml_coords g p in
    Graphics.moveto x (y - height); (* Ocaml coords *)
    Graphics.draw_string s
```

18.6 Simple Widgets & Layout

The `Gctx` module lets us relativize drawing to a widget-local coordinate system. Now, let us see how we can use the `Gctx.t` structure to address the challenge of laying out widgets on the window. At a minimum, a widget will need to be able to draw itself relative to a graphics context. Since each widget will occupy some region of the window, a widget should also be able to report its size, so that we can position one widget relative to another.

This leads us to the following type for (simple) widget objects[3]:

```
(* The widget module *)

type t = {
  repaint : Gctx.t -> unit;
  size      : Gctx.t -> (int * int);
}
```

The `repaint` function asks the widget to draw itself using the `Gctx` drawing primitives—we call it "repaint" because, eventually, we repeatedly draw widgets to the screen, which allows for animation or changes to the visual state of the GUI application. For example, a `checkbox` widget's `repaint` function might draw an X depending on whether the checkbox is selected (as shown in the "Thick Lines" checkbox of Figure 18.1).

Given just the widget type above, we can already create some simple widgets. The simplest widget does nothing but occupy space in the window—it's repaint function does nothing, and its size is determined by parameters passed in when constructing the widget object.

```
(* The simplest widget just occupies space. *)
let space ((w,h):int * int) : t =
  {
    repaint = (fun _ -> ())
    size = (fun _ -> (w,h));
  }
```

Another simple widget just draws a string:

[3]In §18.10 we will extend this type to deal with user events

```
(* Display a string on the screen. *)
let label (s:string) : t =
  {
    repaint = (fun (g:Gctx.t) -> Gctx.draw_string g s);
    size    = (fun (g:Gctx.t) -> Gctx.text_size g s)
  }
```

The `label` widget's `repaint` function just uses the `draw_string` operation provided by `Gctx`, its `size` is just the size of the string when drawn on the window.

Another useful widget simply exposes a fixed-sized region of the screen as a "canvas" that can be painted on by using the `Gctx` drawing routines. The canvas widget is just a widget parameterized by the `repaint` function:

```
let canvas ((w,h):int*int) (paint : Gctx.t -> unit) : t =
   {
     repaint = paint;
     size = (fun _ -> (w,h))
   }
```

The three widgets above don't yet do anything very interesting to the window display. Since we will eventually want to draw buttons and other complex widgets with lines indicating their boundaries, it is useful to create a widget `border` wrapper. Given a widget `w`, the widget `border w` simply draws a rectangular border around the outside of `w`. The border widget therefore calls the wrapped widget's `repaint` method and also adds its own code to draw the rectangle around the inner widget. The only wrinkle is that we have to be a bit careful with the graphics context. Figure 18.3 illustrates the situation—the border widget's `repaint` function should call `w`'s repaint function, but, since `w`'s upper-left corner is not located at the origin, we have to *translate* the graphics context slightly before passing it on to `w`'s `repaint`. This functionality is easy to add to the `Gctx` module:

```
(* Gctx.ml *)
(* Shifts the gctx by (dx,dy) *)
let translate (g:t) ((dx,dy):int*int) : t =
  {g with x=g.x+dx; y=g.y+dy}
```

Given this `translate` function, it is then easy to implement the functionality of the `border` widget: the `size` function simply pads the size of the inner widget by four pixels in each direction; the `repaint` function

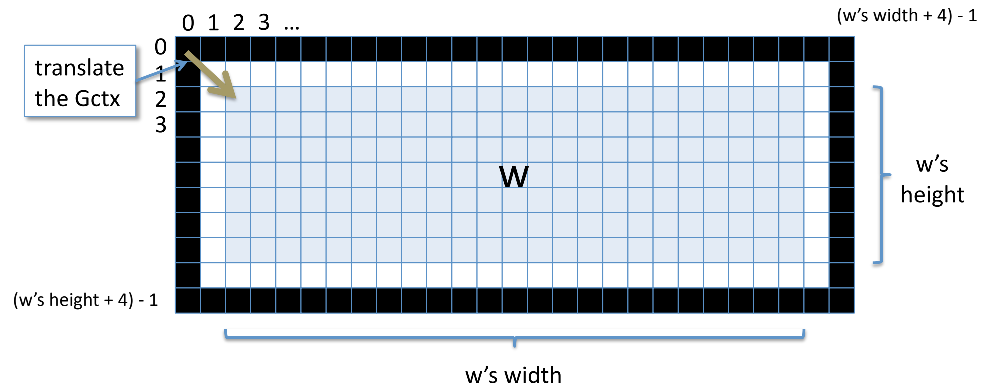


Figure 18.3: The `border` widget wraps another widget `w` with a one-pixel thick line set one pixel away from the inner widget. The `border` widget's `repaint` funtion calls `w`'s `repaint`, but must use `Gctx.translate` to "shift" the inner widget's local coordinate system to (2,2), relative to the border widget's local origin.

draws four lines for the border, translates the `Gctx.t` and then calls `w`'s repaint.

```
(* Adds a one-pixel border to an existing widget. *)
let border (w:t) : t =
  {
  repaint = (fun (g: Gctx.t) ->
                 let (width,height) = w.size g in
                 (* not +4 because we count from 0 *)
                 let x = width + 3 in
                 let y = height + 3 in
                   Gctx.draw_line g (0,0) (x,0);
                   Gctx.draw_line g (0,0) (0,y);
                   Gctx.draw_line g (x,0) (x,y);
                   Gctx.draw_line g (0,y) (x,y);
                   let g = Gctx.translate g (2,2) in
                     w.repaint g)

  size = (fun (g: Gctx.t) ->
              let (width,height) = w.size g in
                width + 4, height + 4);
  }
```

The `translate` function also suggests how we can create a "wrapper" widget that will lay out two widgets side-by-side in the window. The idea is to simply translate the right widget horizontally by the width of the left

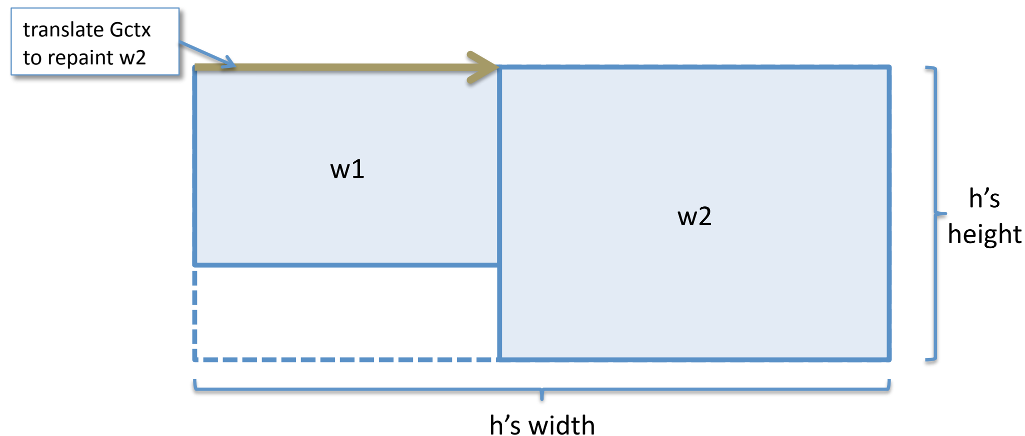


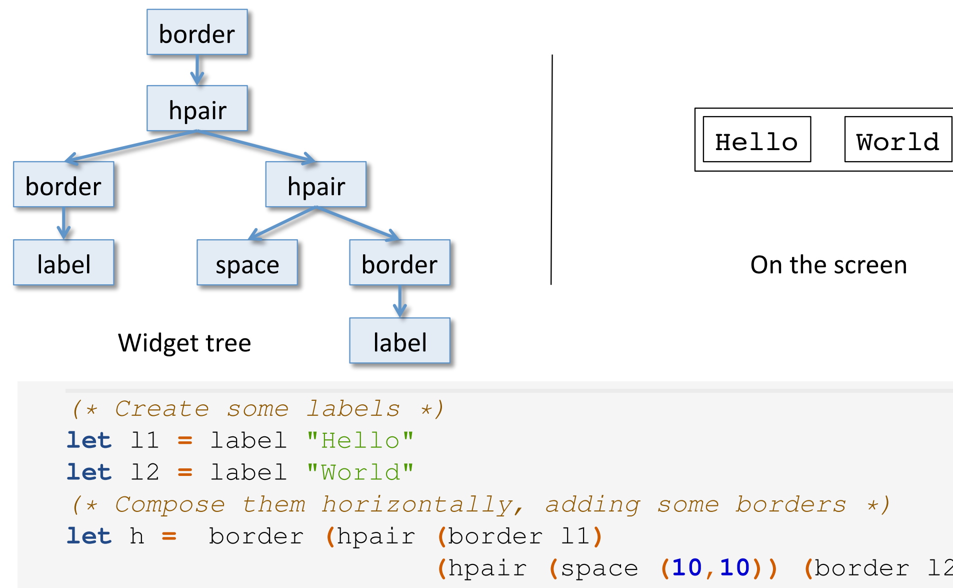
Figure 18.4: The call `hpair w1 w2` yields a widget `h` comprised of `w1` and `w2` layed out horizontally adjacent to one another. The right widget's `Gctxt.t` must be translated by the width of the left widget.

widget, as shown in Figure 18.4. The `size` is simply the sum of the two widgets' heights and maximum of their heights. The resulting "horizontal pair" widget code looks like this:

```
(* The hpair widget lays out two widgets horizontally.  They
   are aligned at their top edge. *)
let hpair (w1:t) (w2:t) : t =
  {
  repaint =
      (fun (g:Gctx.t) ->
         w1.repaint g;
         let g = Gctx.translate g (fst (w1.size g),0) in
           w2.repaint g);
   size =
      (fun (g:Gctx.t) ->
         let (x1,y1) = w1.size g in
         let (x2,y2) = w2.size g in
           (x1 + x2, max y1 y2))
  }
```

18.7 The widget hierarchy and the `run` function

So far, we have tackled the problem of layout and modular reuse of




```
(* Create some labels *)
let l1 = label "Hello"
let l2 = label "World"
(* Compose them horizontally, adding some borders *)
let h =  border (hpair (border l1)
                       (hpair (space (10,10)) (border l2)))
```

Figure 18.5: Widgets form a tree: each "wrapper" widget like a `border` or `hpair` contains one or more children. The tree on the left above corresponds to the widget `h` created by the program above. When painted to the graphics window, `h` would be displayed as shown on the right.

widget code. The `Gctx` and `Widget` modules together give us a way of constructing more complex widgets out of smaller ones by arranging them spatially in the graphics window.

When we build an application (such as the paint program) that uses the `Widget.t` types to construct a user interface, we can think of the widget instances as building a *tree*—the leaves of the tree are the "primitive" widgets like `layout`, `space`, and `canvas` (which don't have any sub-widgets inside them), and the nodes of the tree are those widgets, like `border` and `hpair`, that "wrap" one or more children. Figure 18.5 shows pictorially what such a widget tree looks like for a concrete widget program.

To paint a widget hierarchy to a graphics window, all we have to do is invoke the `repaint` function of the root widget, giving it an initial graphics context. For example, calling `h.repaint` for the program of Figure 18.5 will cause the image shown on the right-hand side of the Figure to be displayed.

As a first cut for the top-level program, we can thus create a `run` function that takes in the root widget, creates a new OCaml graphics window,

asks the widget to repaint itself, and then goes into an infinite loop. (We have to use some kind of loop, otherwise the program would draw the widget and then exit too quickly for us to see the resulting graphics!)

```
(* A program that displays a widget in the window *)
let run (w:Widget.t) : unit =
  Graphics.open_graph "";        (* open the window *)
  let g = Gctx.top_level in       (*  the top-level Gctx *)

  let rec loop () =
    loop ()
  in
    w.repaint g;
    loop ()                      (* let us see the results *)
```

As we shall see next, we will modify this top-level loop so that it can process user-generated GUI events, such as mouse motions.

18.8 The Event Loop

We have successfully addressed one of the design challenges for building a GUI library: the combination of `Gctx` and `Widget` modules provide a clean way of creating re-usable graphical components that can be positioned relative to one another in the window. The remaining challenge is how to process user-generated *events*, such as mouse clicks, mouse motion, or key presses.

The first step toward solving this problem is to replace the `run` function we just saw with something more useful. Consider the paint application, for example. Unless the user provides some input, by using the mouse to click a button or draw in the paint canvas, the paint program itself does nothing—it passively waits for a event that it knows how to process, processes the event, which might cause the visual display of the paint program to be altered, and then goes back to waiting for another event. Clearly, the infinite loop of the `run` function should be replaced by some code that waits for a user event to process and then somehow updates the internal state of the application. At the start of each loop iteration, we can clear the entire graphics window and then ask the root widget to repaint itself. This leads to a `run` function with the following structure:

```
open Widget;;
(*

This function takes a widget, which is the "root" of the GUI
interface.  It starts with "top-level" Gctx, and then it goes into
an infinite loop.  The loop simply repeats these steps forever:

     - clear the graphics window
     - ask the widget to repaint itself
     - wait for a user input event
     - forward the event to the widget's event handler
*)
let run (w:Widget.t) : unit =
  Graphics.open_graph "";           (* open the window *)
  Graphics.auto_synchronize false; (* draw to hidden buffer *)
  let g = Gctx.top_level in          (* the top-level Gctx *)

  let rec loop () =
    Graphics.clear_graph ();
    w.repaint g;
    Graphics.synchronize ();             (* show freshly painted window *)
    let e = Gctx.wait_for_event g in (* wait for user input event *)
      w.handle g e; loop ()               (* widget handles the event *)
  in
    loop ()
```

Here, the OCaml graphics library is set to use double buffering by turning off auto synchronization. Double buffering is a technique used to eliminate flicker caused when graphics are written incrementally to the display device; rather than do that, all of the graphics are written to a "backing buffer", which is then displayed all at once when the `Graphics.synchronize` function is invoked.

The new part of this `run` function is what allows the GUI program to be interactive— it consists of two lines of code, which make use of some new functionality that we will add to the `Gctx` and `Widget` modules, as explained in more detail below:

```
let e = Gctx.wait_for_event g in (* wait for user input event *)
   w.Widget.handle g e; loop ()   (* widget handles the event *)
```

First, the new function `wait_for_event` tells the program to wait for a new user-generated mouse or keyboard event. Second, once an event is received, we call the root widget's `handle` function to ask it to process the

event.

18.9 GUI Events

An event is just a value that represents a signal from the underlying operating system to the OCaml graphics library. As shown by the definition of the type `status`, found the OCaml graphics module, events carry information about the mouse coordinates, button status, and keyboard information:

```
(* The event status information from OCaml's graphics library. *)
type status = {
  mouse_x : int;      (* X coordinate of the mouse *)
  mouse_y : int;      (* Y coordinate of the mouse *)
  button : bool;      (* true if a mouse button is pressed *)
  keypressed : bool;  (* true if a key has been pressed *)
  key : char;         (* the character for the key pressed *)
}
```

Since such events carry coordinate information about where in the window the event occurred (i.e. where the mouse is), we need to extend the `Gctx` module to provide a way of mapping OCaml event coordinates to widget-local events. We can also create some helper functions for accessing the information about a given event, relative to a graphics context. It is straightforward to add these features to the `Gctx` module, like so:

```
type event = Graphics.status

(* Waits for a mouse or key event *)
let wait_for_event (g:t) : event =
  Graphics.wait_next_event
    [Graphics.Mouse_motion;
     Graphics.Button_down;
     Graphics.Button_up;
     Graphics.Key_pressed]

(* Determine whether the event has the button pressed *)
let button_pressed (g:t) (e:event) : bool =
  e.Graphics.button

(* Get the widget-local coordinates of an event *)
let event_pos (g:t) (e:event) : int * int =
  local_coords g (e.Graphics.mouse_x, e.Graphics.mouse_y)

(* Determine whether the event has a key pressed *)
let is_keypressed (g:t) (e:event) : bool =
  e.Graphics.keypressed

(* Get the keyboard character associated with an event *)
let get_key (g:t) (e:event) : char =
  e.Graphics.key
```

18.10 Event Handlers

Once an event has been received by the top-level event loop (the `run` function), it asks the root widget `w` to handle the event. To allow widgets to react to events, we need to extend their type with a new function, called `handle`. This function takes in a `Gctx.t` and a `Gctx.event`, and processes the event in a widget-specific fashion.

The type of widgets is thus modified from what we had earlier to be:

```
(* The interface type common to all widgets *)
type t = {
  repaint: Gctx.t -> unit;
  handle: Gctx.t -> Gctx.event -> unit;
  size: Gctx.t -> int * int
}
```

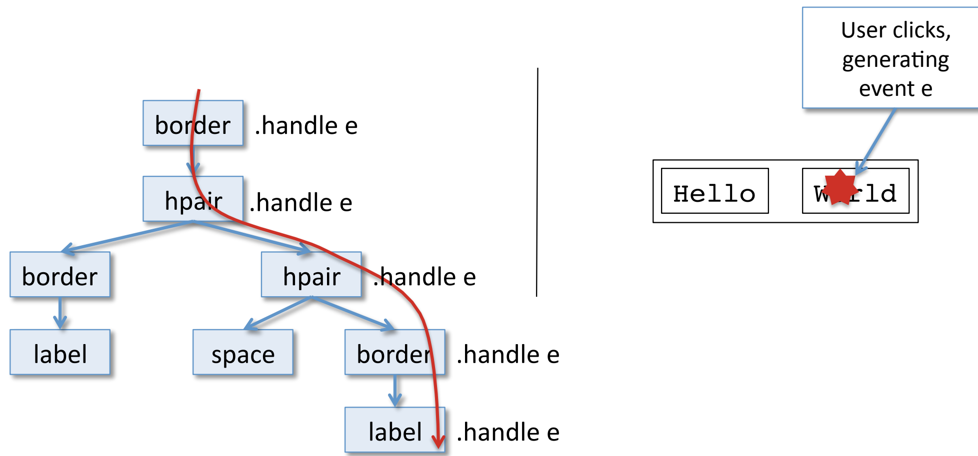


Figure 18.6: When a user clicks some place in the GUI window, the resulting event is routed through the widget heirarchy starting at the root. Each node forwards the event by calling a child widget's `handle` function after suitably translating the `Gctx.t` value. Which child gets the event is determined by their size, layout, and the coordinates of the event.

How does our program know what to do when the user clicks on a region of the window that displays a particular button, like "Undo" in the paint program? Clearly each button will (in general) need to be associated with some bit of code that gets executed when it is clicked, but there must also be some way to "route" the event to the appropriate widget of the widget hierarchy.

We have actually already encountered a similar problem twice before, albeit in very different contexts. First, recall that when inserting a new value into a binary search tree we exploited the ordering structure of the values in the nodes of the tree to "route" the new value to its proper location. Second, recall that, for Homework 4 we used quad trees to partition a 2D region of the plane and that inserting a point into the quad tree essentially amounted to "routing" the point to its proper location in the tree based on its coordinates.

For routing events to widgets, we use a similar idea: the "container" widgets like `border` and `hpair` use the spatial information about the layout of their subwidgets to decide which of the children's `handle` functions should be called. This situation is depicted in Figure 18.6.

For example, the `border` widget simply passes any events received by

its handler to its only child, but only after suitably translating the `Gctx.t` so that the child can interpret the event relative to its own local coordinate system.

```
(* Modified version of border that handles events *)
let border (w:t):t =
  {
    repaint = ...;   (* same as before *)
    size = ...;      (* same as before *)
    handle = (fun (g:Gctx.t) (e:Gctx.event) ->
       w.handle (Gctx.translate g (2,2)) e);
  }
```

Similarly, the `hpair` widget checks which of its two children should receive the event and forwards it to the appropriate one. Note that, since there is some "dead space" created if one of the two children is shorter than the other, it is possible that neither child will receive the event. Events that occur in the "dead space" are simply dropped by the hpair widget. Thus we have:

```
(* Determines whether a given event is within a
   region of a widget whose upper-left hand corner is (0,0)
   with width w and height h. *)
let event_within (g:Gctx.t)
                 (e:Gctx.event)
                 ((w,h):int*int) : bool =
  let (mouse_x, mouse_y) = Gctx.event_pos g e in
    mouse_x >= 0 && mouse_x < w && mouse_y >= 0 && mouse_y < h

let hpair (w1:t) (w2:t) : t =
  {
    repaint = ...;   (* same as before *)
    size = ...;       (* same as before *)
    handle =
         (fun (g:Gctx.t) (e:Gctx.event) ->
           if event_within g e (w1.size g)
           then w1.handle g e
           else
             let g = (Gctx.translate g (fst (w1.size g), 0)) in
               if event_within g e (w2.size g)
               then w2.handle g e
               else ());
  }
```

Modifying the `space`, `label`, and `canvas` widgets is straightforward—

each of their `handle` functions simple does nothing. Of course, we might like to add the ability for a widget of one of these types to handle events as well, but we will do so using the `notifier` widget described below. First, though, we need to consider how widgets with local state can be constructed.

18.11 Stateful Widgets

Consider the simple label widget that we saw earlier.

```
(* Display a string on the screen. *)
let label (s:string) : t =
  {
    repaint = (fun (g:Gctx.t) -> Gctx.draw_string g s);
    size     = (fun (g:Gctx.t) -> Gctx.text_size g s);
    handle = (fun (g:Gctx.t) (e:Gctxt.event) -> ())
  }
```

This widget is *stateless* in the sense that, once the string has been associated with the label, the string never changes.

If we wanted to be able to modify the string displayed by a label, we could use the techniques of §17 to give the label widget some *local state*, like this:

```
let label (s:string) : t =
  let lbl : string ref = ref s in
  {
    repaint = (fun (g:Gctx.t) -> Gctx.draw_string g !lbl);
    size     = (fun (g:Gctx.t) -> Gctx.text_size g !lbl);
    handle = (fun (g:Gctx.t) (e:Gctxt.event) -> ())
  }
```

However, although the code above creates a mutable `string ref` in which to store the label's string, code external to the label widget cannot modify the contents of `lbl`. To do that, we need to also expose a function that lets other parts of the program update `lbl`. We call such a function a *label controller* since it controls the string displayed by a label. Given the type of `label_controller` records, it is easy to modify the `label` function to create a widget (of type `Widget.t`) *and* a `label_controller`:

```
type label_controller = { set_label : string -> unit }

let label (s:string) : t * label_controller =
  let lbl : string ref = ref s in
  ({
    repaint = (fun (g:Gctx.t) -> Gctx.draw_string g !lbl);
    size     = (fun (g:Gctx.t) -> Gctx.text_size g !lbl);
    handle = (fun (g:Gctx.t) (e:Gctxt.event) -> ())
   },
  {set_label = (fun (r:string) -> lbl := r)})
```

Now when we call the `label` function, it returns a pair consisting of a widget and a label controller that can be used to change the string displayed by the label.

More generally, any stateful widget will have its own kind of controller that can be used to modify the widgets state.

18.12 Listeners and Notifiers

Different widgets may need to react to different kinds of events. For example, a button needs to "listen" for mouseclick events and then run some appropriate code; a scrollbar might have to "listen" for mouse drag events (mouse motion with the button down); a textbox widget might have to "listen" for key press events. Moreover, the way that a button (or other widget) responds to a certain event might change depending on application context. In general, whether or not a widget should "listen" for a particular event might also depend on application-specific state.

How can we easily capture the wide variety of possible ways that a widget might want to interact with user-generated events? How can we modularly add or remove code for processing events? Our solution (which is based on Java's Swing library) is to introduce a new kind of widget called a *notifier*. The idea is that a notifier widget "eavesdrops on" or "listens to" the events flowing through a part of the widget hierarchy. It manages a list of *event listeners*, which are a bit like `handle` functions except that they don't really participate in routing events through the widget hierarchy—they simply listen for a particular kind of event, react to it, and then either propagate the event or stop it from being further processed. Figure 18.7 shows pictorially, how we might add a `notifier` widget to the "Hello World" example.

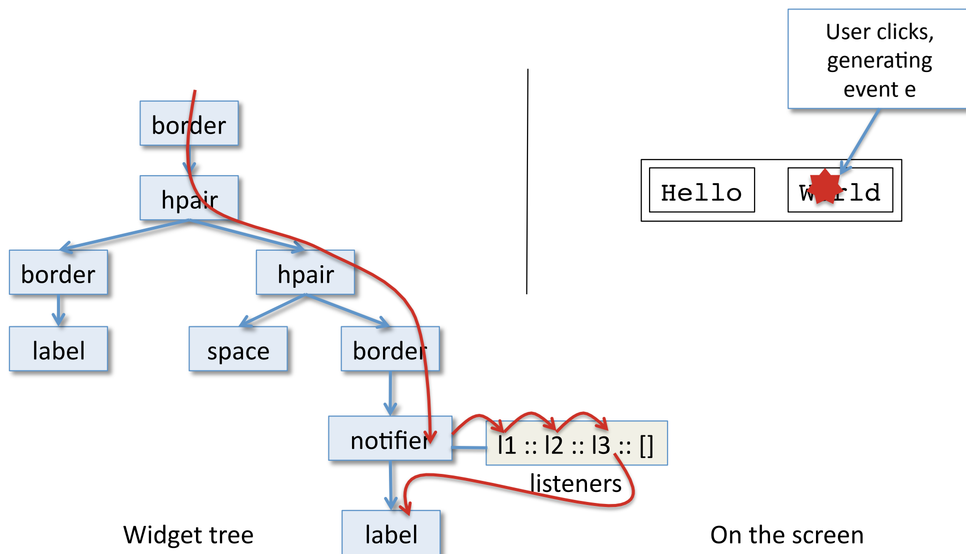


Figure 18.7: A notifier widget maintains in its local state a list of "listeners" that get a chance to process events that flow through the widget tree. Which listeners are associated with the notifier can be changed using a notifier controller.

Notifiers, like the stateful version of `label` widgets discussed above, maintain some local state—in this case a list of `event_listener` objects. Thus, each notifier widget comes equipped with a `notifier_controller` that can be used to modify the list of listeners. For simplicity in the code below, the `notifier_controller` only allows new `event_listener` objects to be *added* to the `notifier`; it is easy to extend this idea to allow `event_listener`s to be removed as well.

These design criteria lead us to create a type for `event_listener` functions that looks like this:

```
type event_listener_result =
  | EventFinished
  | EventNotDone

type event_listener =
  Gctx.t -> Gctx.event -> event_listener_result
```

An `event_listener` is just a function that, like a `handle` method of a widget, takes a `Gctx.t` and `Gctx.event` and processes the event. Unlike a handler, which always returns `unit`, an `event_listener` can return either

`EventFinished`, which signals to the `notifier` that the event should not be further processed by listeners that are "down stream" in the widget tree, or `EventNotDone`, which signals that the event should be propagated.

Once we have defined the type of `event_listener`s, it is straightforward to define the behavior of the `notifier` widget and its `notifier_controller`:

```
(*
  A notifier_controller is associated with a notifier widget.
  It allows the program to add event listeners to the notifier.
*)
type notifier_controller = {
  add_event_listener: event_listener -> unit
}

(*

  A notifier widget is a widget "wrapper" that doesn't take up
  any extra screen space -- it extends an existing widget with
  the ability to react to events.  It maintains a list of
  "listeners" that eavesdrop on the events propagated through
  the notifier widget.

   When an event comes in to the notifier, it is passed to each
   listener in turn until one of them declares the event
   to be "finished".
*)
let notifier (w: t) : t * notifier_controller =
  let listeners = ref [] in
    {         repaint = w.repaint;
         handle =
         (fun (g:Gctx.t) (e: Gctx.event) ->
            let rec loop (l: event_listener list) : unit =
              begin match l with
                | [] -> w.handle g e
                | h::t -> begin match h g e with
                      | EventFinished -> ()
                      | EventNotDone -> loop t
                    end
              end in
              loop !listeners);
         size = w.size
    },
  {
    add_event_listener =
      fun newl -> listeners := newl::!listeners
  }
```

With this infrastructure in place, it is easy to define specialized event listeners that react to particular kinds of events. For example, it is useful to define a `mouseclick_listener`, parameterized by an action to perform

when the mouse is clicked:

```
(* Performs an action upon receiving a mouse click. *)
let mouseclick_listener (action: unit -> unit) : event_listener =
  fun (g:Gctx.t) (e: Gctx.event) ->
    if Gctx.button_pressed g e
    then (action (); EventFinished)
    else EventNotDone
```

A mouse_listener, in contrast, exposes another convenient way of reacting to mouse events—it unpacks the mouse button and position information from an event e and passes that data to an action function. A client application like the Paint program might add a mouse_listener to a canvas widget to allow the canvas to react to user-generated mouse events on the canvas.

```
(*
  A mouse_listener takes an action that responds to mouse events
  of all kinds - the action determines what to do if the mouse
  button is down and the mouse cursor is at a particular location
*)
let mouse_listener
    (action : bool -> int*int -> event_listener_result)
      : event_listener =
  fun (g:Gctx.t) (e:Gctx.event) ->
    action (Gctx.button_pressed g e) (Gctx.event_pos g e)
```

18.13 Buttons (at last!)

Our GUI library finally has enough functionality to implement a traditional button widget: A button is just a label widget wrapped in a notifier. The resulting widget has both a label_controller and a notifier_controller, which can be used to change the state of the button.

```
(*
 A button has a string, which can be controlled by the
 corresponding label_controller, and a notifier_controller,
 which can be used to add listeners (e.g. a mouseclick_listener)
 that will perform an action when the button is pressed.
*)
let button (s: string) : t * label_controller
                           * notifier_controller =
  let (w, lc) = label s in
  let (w', nc) = notifier w in
    (w', lc, nc)
```

To add the ability to react to a mouse click event to the button, which is typically the desired behavior, we can simply use the `notifier_controller`'s `add_event_listener` function to add a `mouseclick_listener`. For example, to create a button that prints `"Hello, world!"` to the console each time it is clicked, we could write the following code:

```
let (hw_button, _, hw_nc) = button "Print It!"

let print_hw () : unit =
  print_endline "Hello, world!"

;; hw_nc.add_event_listener (mouseclick_listener print_hw)
```

The `hw_button` widget could then be added to a larger widget tree using layout widgets, or, simply run as the entire "application". The latter would be accomplished by doing:

```
;; Eventloop.run hw_button
```

18.14 Building a GUI App: Lightswitch

We're finally in position to build an application on top of the GUI library. Figure 18.8 shows the basic structure of an application built using the GUI library—the application never has to interact directly with the underlying OCaml primitives; instead it works with the operations provided by the `Gctx`, `Widget`, and `Eventloop` modules. Together those three components provide an appropriate collection of abstractions for building GUI programs. Of course, the feature set we have seen in this Chapter is far from

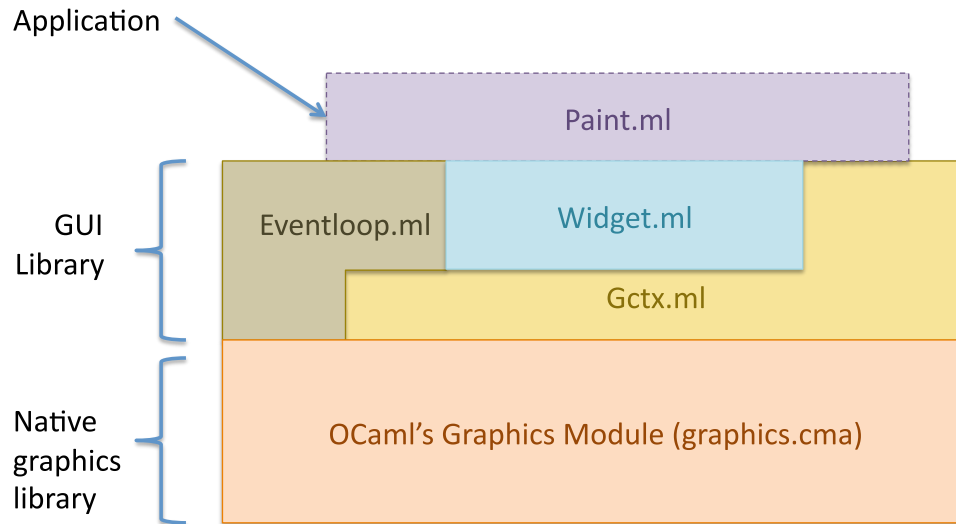


Figure 18.8: The resulting software "architecture" for an application built on top of the GUI library. An application like the Paint program should never interact with the OCaml graphics library directly; it should instead call functions in the `Gctx`, `Widget`, and `Eventloop` modules.

complete—a fully fledged GUI library would provide much more functionality, including, other graphics drawing primitives, layout options, and widgets. Adding such features is simply a matter of extending the `Gctx` and `Widget` modules, following more or less the same pattern as we have seen above. The Paint program GUI project associated with this part of the course explores how to make such changes, to the extent that we can implement the paint program pictured in Figure 18.1.

Even though the GUI library is rather primitive, we can still use it to demonstrate how all of the pieces fit together. The program below builds a "lightswitch" application, shown in Figure 18.9.

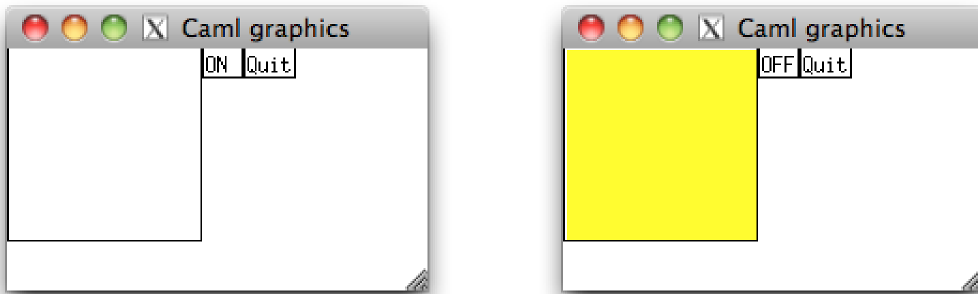


Figure 18.9: The "lightswitch" application both before (left) and after (right) the On/Off button has been pressed.

```
(* This program demonstrates how to build an application
   on top of the CIS 120 GUI library.  It assumes that
   the eventloop.ml*, gctx.ml*, and widget.ml* files
   are all present. *)


open Widget

(* Create the state affected by the light switch *)
let switched_on : bool ref = ref false

(* Create a lightswitch button, initially labeled "ON " *)
let (b,l,n) =
   Widget.button "ON "

(* The action associated with clicking the switch. *)
let flip () : unit =
  switched_on := not (!switched_on);
  if !switched_on then
    l.set_label "OFF"
  else
    l.set_label "ON "

(* Add the flip action as a mouseclick_listener *)
;; n.add_event_listener (Widget.mouseclick_listener flip)
```

```
(* Create the "QUIT" button *)
let (b2,_,n2) =
    Widget.button "QUIT"

(* The action associated with the QUIT button *)
let quit () : unit =
   exit 0

(* Add the quit action as a mouseclick_listener *)
;;  n2.add_event_listener (Widget.mouseclick_listener quit)

(* Create the "lightbulb" canvas *)
let repaint_light (g:Gctx.t) : unit =
  if !switched_on then
    let g = Gctx.with_color g Gctx.yellow in
    Gctx.fill_rect g (0,0) (100,100)
  else
     ()

let (light,_) = Widget.canvas (100,100) repaint_light


(* Package all the pieces together into a root widget *)
let w : Widget.t =
  Widget.border (Widget.border (hpair
    (Widget.border light)
    (hpair (Widget.border b) (Widget.border b2))))

(* Start waiting for events *)
;; Eventloop.run w
```

Chapter 19

Transition to Java

19.1 Farewell to OCaml

"Transitioning from one mindset to another was challenging; I forgot syntax and kept thinking about everything in Java as values like we do in OCaml, so I kept leaving "return" off of everything. I also forgot about how instance variables in a class are used. But after I realized those key things, I felt great in Java again." — Anonymous CIS 120 Student

In this chapter we begin our transition to Java programming. As we shall see, many of the concepts and ideas that we have explored in the context of OCaml arise again for Java—the two languages, despite being very different superficially and having different "feels" are actually more similar than you might expect. Understanding OCaml programming well will serve as a good foundation for understanding Java.

By now we have actually seen most of OCaml's important features—the language itself is not very big. But we have left out a few things:

- One of OCaml's strengths is its module system, which provides support for large-scale programming. We saw just the tip of the iceberg here in §9 when we studied *structures* and *signatures*. The key feature we haven't seen is called *functor*, which is a function from one structure to another.
- The "O" in OCaml stands for "object". OCaml does include a powerful system of classes and objects, similar to those found in other *object-oriented* OO languages. We have left them out so that we can study OO programming in Java, without the potential for confusion.
- OCaml's type system also provides very strong support for *type inference*. Almost all of the type annotations that we've been writing as

part of our OCaml code can be completely omitted; the compiler's type checker is able to figure out the types of every expression by looking at how the program is structured.

The functional programming style that OCaml promotes emphasizes several key concepts: the idea that program computations should be thought of as computing with values, which, for the most part are immutable tree-structured data. The main ways of working with such data are pattern matching, which lets the program examine the structure of the data, and recursion, which is the natural way to process inductively defined trees.

In this context, we have seen the importance of using abstract types to preserve invariants while manipulating mutable data structures (as in the `queue` operations of §16.)

The functional style is good for expressing simple, elegant descriptions of complex algorithms and/or data structures. The limited use of mutability, and persistence-by-default, makes functional programming well-suited for parallelism, concurrency, and distributed applications (though we haven't touched on these aspects in this course). OCaml's tree-structured datatypes are also well-suited for a variety of "symbolic processing" tasks, including building compilers, theorem provers, *etc.*.

Dialects of ML, and other functional programming languages like Scheme and Haskell, have had a big impact on the design of "modern" programming languages like C#, Java, and, to a lesser extent, C++. The use of generic programming and type inference, for example, was pioneered in ML. Object-oriented languages like Java and C# are beginning to incorporate features that support functional-programming idioms—for example, future versions of Java will include closures (*i.e.* anonymous functions) that are so pervasive in OCaml. Moreover, while C#, C++, and Java encourage the use of mutable state with their defaults, many "best practice" approaches to software development in these language actively discourage using imperative state.

Finally, some languages strive to merge the functional and the object-oriented styles of programming. These "hybrid" languages including Scala and Python, for example, offer functional programming as one possibility among many styles of writing code.

For all of these reasons, learning OCaml will give you a new perspective on how to go about solving problems in *any* programming language

that you encounter. Understanding when functional programming is an appropriate solution to a problem can let you write better code no matter what language you use to express that solution.

"Having to worry about null was probably the hardest [part of the transition]. Also I really missed being able to match things." — Anonymous CIS 120 Student

The succinctness and clarity of OCaml for these kinds of tasks, can be shown by comparing this (hopefully by now!) straightforward OCaml program that defines the `tree` type and an `is_empty` function along with a use of it, to its equivalent in Java.

```
type 'a tree =
  | Empty
  | Node of ('a tree) * 'a * ('a tree)

let is_empty (t:'a tree) =
  begin match t with
    | Empty -> true
    | Node(_,_,_) -> false
  end

let t : int tree = Node(Empty,3,Empty)
let ans : bool = is_empty t
```

The corresponding Java program is much more verbose (and isn't even fully operational, since I have omitted the methods needed to access the tree's values). (Don't worry about understanding the code yet, the next several chapters will explain all of the necessary pieces.)

```
interface Tree<A> {
  public boolean isEmpty();
}
class Empty<A> implements Tree<A> {
  public boolean isEmpty() {
     return true;
  }
}
class Node<A> implements Tree<A> {
  private final A v;
  private final Tree<A> lt;
  private final Tree<A> rt;

  Node(Tree<A> lt, A v, Tree<A> rt) {
    this.lt = lt; this.rt = rt; this.v = v;
  }

  public boolean isEmpty() {
    return false;
  }
}

class Program {
  public static void main(String[] args) {
    Tree<Integer> t =
    new Node<Integer>(new Empty<Integer>(),
     3, new Empty<Integer>());
    boolean ans = t.isEmpty();
  }
}
```

Though working with OCaml and the functional style are a good way to broaden your mental toolbox, it is also essential to be able to work fluently in other programming paradigms. Just as the program above can be written very cleanly in the functional style, there are Java programs that would be difficult to express cleanly in OCaml. For this reason, we will study object-oriented programming using Java.

Java, of course, offers the benefits of being a widely-used, "industrial strength" programming language with a large ecosystem of libraries, tools, and other resources designed for professional software developers. Java as a programming language is itself a rather large and complicated entity (with good reason!), which has evolved over the years and is continuing to change.

The goal of studying object-oriented programming in Java is not, therefore, so that you become an expert Java developer. Instead, the goal is to give you an understanding of the essence of object-oriented languages, and how their features can be used to address programming design problems.

19.2 Objects and Classes

The fundamental difference between OCaml and Java is Java's pervasive use of *objects* as the means of structuring data and code. Java syntax provides a convenient way to *encapsulate* some state along with operations on that state. To see what this means, consider this variant of the `counter` program that we saw earlier when discussing local state (see §17):

```
(* The type of a counter's local state. *)
type counter_state = {mutable cnt : int}

(* Methods for interacting with a counter object. *)
type counter = {
  inc: unit -> int;
  dec: unit -> int;
}

(* Constructs a fresh counter object with its own local state. *)
let new_counter () : counter =
  let s : counter_state = {cnt = 0} in
  {
    inc = (fun () -> s.cnt <- s.cnt+1; s.cnt) ;
    dec = (fun () -> s.cnt <- s.cnt-1; s.cnt) ;
  }

(* Create a new counter and then use it. *)
let c : counter = new_counter () in
;; print_int (c.inc ())
;; print_int (c.dec ())
```

This OCaml program illustrates the three key features of an object:

- An object *encapsulates* some local, often mutable state. That local state is visible only to the methods of the object.
- An object is defined by the set of *methods* it provides—the only way

to interact with an object is by invoking (*i.e.* calling) these methods. Moreover, the type of the encapsulated state does not appear in the object's type.

- There is a way to construct multiple *instances*—*new* object values—that behave similarly (and therefore share implementation details).

In the OCaml example above, the first feature is embodied by the use of the type `counter_state`, which describes that local state associated with each counter object. The second feature is realized by the type `counter`, which exposes only the two methods available for working with counter objects. Finally, the function `new_counter` provides a way to create new `counter` instances (*i.e.* values of type `counter`). The use of local scoping ensures that the state `s` is only available to the code of the `inc` and `dec` methods, and therefore cannot be touched elsewhere in the program—the state `s` is encapsulated properly.

Java's notation and programming model make it easier to work with data and functions in this style. A Java class combines all three features—local state, method definitions, and instantiation—into one construct. Think of a class as a template for constructing instances of objects—classes are not values, they describe how to create object values. The Java code below shows how to define the class of counter objects that is analogous to the OCaml definitions above:

```
public class Counter {

  private int cnt;      // the encapsulated local state

  // constructs a new Counter object
  public Counter () {
    cnt = 0;
  }

  // the inc method
  public int inc () {
    cnt = cnt + 1;
    return r;
  }

  // the dec method
  public int dec () {
    cnt = cnt - 1;
    return cnt;
  }
}
```

Here the notation `public class Counter { ... }` defines a new type, *i.e.* class, of counter objects called `Counter`. Here, and elsewhere, the keyword `public` means that the definition is globally visible and available for other parts of the program to use. A class consists of three types of declarations: fields, constructors, and methods.

A *field* (also sometimes called an *instance variable*) is one component of the object's local state. In the `Counter` class, there is only one field, called `cnt`. The keyword `private` means that this field can only be accessed by code defined inside the enclosing class—it is used to ensure that the state is encapsulated by the object.

A *constructor* always has the same name as its enclosing class—it describes how to build an object instance by initializing the local state. Here, the constructor `Counter` simply sets `cnt` to `0`.

A *method* like `inc` or `dec` defines an operation that is available on objects that are instances of this class. In Java, a method is declared like this:

```
public T method(T1 arg1, T2 arg2, ..., TN argN) {
    ...
    return e;
}
```

Here, `T` is the method's *return* type—it says what type of data the method produces. `T1 arg1` through `TN argN` are the method's parameters, where `T1` is the type of `arg1`, *etc.* The `return e` statement can be used within the body of a method to yield the value computed by `e` to the caller. Here again, the keyword `public` indicates that this method is globally visible. Note that the field `cnt` is available for manipulation within the body of these methods. As we will see, methods can also define local variables to name the intermediate results needed when performing some computation.

Unlike in OCaml, in which code can appear "at the top level", in Java *all* code lives inside of some class. A Java program starts executing at a specially designated `main` method. For example, a program that creates some counter objects and uses their functionality might be created in a class called `Main` like this:

```
public class Main {
    public static void main(String[] args) {
        Counter c = new Counter();
        System.out.println(c.inc());
        system.out.println(c.dec());
    }
}
```

The type of `main` must always be declared as shown above—we'll see the meaning of the `static` keyword in §21. The keyword `void` indicates that the `main` method does not return a useful value—it is analogous to OCaml's `unit` type. To create an instance of the `Counter` class, the code in the `main` invokes the `Counter` constructor using the `new` keyword. The expressions `c.inc()` and `c.dec()` then invoke the `inc` and `dec` methods of the resulting object, which is stored in the local variable `c`.

19.3 Mutability and `null`

By default, every field or local variable defined in Java is mutable—its value can be modified in place. Java's notation for in-place update is the `=` operator, and we already saw a use of it in the `inc` and `dec` methods of the `Counter` class. The statement:

```
cnt = cnt + 1;
```

is equivalent to the OCaml expression:

```
s.cnt <- s.cnt + 1
```

Another significant distinction between OCaml and Java is that in Java variables and fields that are references to objects are initialized to a special `null` value. The `null` value indicates the *lack* or *absence* of a reference. However, since Java's type system considers `null` to be an appropriate value of any reference type, any variable declared to contain an object might contain `null` instead. Trying to use a field or value via a `null` reference will cause your program to raise a `NullPointerException` when you run it. Here is an example using the `Counter` objects defined above:

```
Counter c;                  // at this point, c contains null

if (c == null) {            // this test will succeed
  System.out.println("c is null");
  c.inc();                  // this method invocation will raise
                            // NullPointerException
}
```

To avoid `NullPointerException` errors, you should either check to make sure that the reference is not `null` before trying to use one of its fields or methods, or structure your program so that you maintain an invariant that guarantees that the reference is not `null`.

Any use of a reference can potentially result in a `NullPointerException`, and, moreover, Java cannot detect such potential errors statically. For example, consider this well-typed client program that provides a method `f` that accepts a `Counter` object as its argument:

```
class Foo {
  public int f (Counter c) {
    return c.inc();
  }
}
```

If `o` is an object of class `Foo`, the call `o.f(null)` will cause the program to raise a `NullPointerException`.

OCaml's use of the `option` types eliminates this problem. Although the option value `None` plays the same role as `null`, its type, `'a option` is *distinct* from `'a`, and it is therefore not possible to use a `'a option` as though it is of type `'a`. Java's type system does not make such a distinction, thereby

creating the possibility that `null` is erroneously treated as an object.

19.4 Core Java

We conclude this transitional Chapter by examining some of the core pieces of Java syntax. The goal is not to be comprehensive, but rather to cover the basic features of the language, emphasizing the similarities and differences with OCaml.

Statements vs. Expressions

Java is a *statement* language—we think of the program as consisting of a series of *commands* that execute one after another. Java's statements themselves are built from *expressions*, which, as in OCaml, evaluate to values. Statements in Java are *terminated* by semi-colons '`;`'. (Recall that in OCaml, '`;`' *separates* a command on the left from an expression on the right.)

Local variable declarations and imperative assignment to a local variable are statements, as illustrated by these examples, which might appear in the body of some method:

```
int x = 3;   // declare x and initialize it to 3
int y;       // declare y; it gets the default value 0

y = x + 3;   // update y to be the value of x plus 3

// declare c and initialize it to a new Counter
Counter c = new Counter();

Counter d;   // declare d; it gets the default value null

d = c;       // update d to be the value of c
c = null;    // set c to null
```

Conditional tests are statements in Java—they are not expressions and don't evaluate to values. As a consequence, the `else` block can be omitted, as shown by the two examples below:

```
if (cond) {
    stmt1;
    stmt2;
    stmt3;
}

if (cond) {
    stmt1;
    stmt2;
} else {
    stmt3;
    stmt4;
}
```

Within the body of a method, the `return e;` statement indicates that the value of expression `e` should be the result yielded to the caller of the method. If a method's return type is `void`, then the return statement can be omitted entirely.

Expressions in Java are built using the usual arithmetic and logical operations like `x + y`, `x && y`, and literals like `1.0`, `true`, and `"Hello"`. A method invocation whose return type is non-`void` can be used as an expression of the corresponding type. For example, since `inc` returns an `int`, `c.m() + 3` is a legal Java expression. Constructor invocations, using the `new` keyword, are also expressions, as in: `(new Counter()).inc() + 5`.

Primitive Types

Java supports a large variety of *primitive* types:

```
int                  // standard integers
byte, short, long    // other flavors of integers
char                 // unicode characters
float, double        // floating-point numbers
boolean              // true and false
```

Java supports essentially the same set of arithmetic and logical operators that OCaml does, as summarized in table of Figure 19.1

Unlike in OCaml, some of Java's operations are *overloaded*—the same syntactic operation might cause different code to be executed. This means that the arithmetic operators `+`, `*`, *etc.*, can be applied to all of the numeric types. Java will also introduce automatic conversions to change one numeric type to another:

OCaml	**Java**	description
`=` `==`	`==`	equality test
`<>` `!=`	`!=`	inequality
`<` `<=` `>` `>=`	`<` `<=` `>` `>=`	comparisons
`+`	`+`	addition
`-`	`-`	substration
`/`	`/`	division
`*`	`*`	multiplication
`mod`	`%`	remainder (modulus)
`not`	`!`	logical "not"
`&&`	`&&`	logical "and"
`||`	`||`	logical "or"

Figure 19.1: Java's primitive operations compared to OCaml's.

```
    4 / 3   ⟹  1
4.0 / 3.0   ⟹  1.3333333333333333
  4 / 3.0   ⟹  1.3333333333333333
```

Moreover, `+` is also overloaded to mean `String` concatenation, so we also have:

`"Hello," + "World!"` ⟹ `"Hello, World!"`.

Overloading is a much more general concept than indicated by just these examples—we will study it it more detail later.

Equality

Just as OCaml includes two notions of equality—*structural* (or deep) equality, written `v1 = v2`, and *reference* (or *pointer*) equality, written `v1 == v2`—Java also has two notations of equality. Java uses the same notation, `v1 == v2`, to check for reference equality of objects or equality of primitive datatypes. Java supports structural equality only for objects. Every object in Java has a `equals` method that should be used for structural comparisons: `o1.equals(o2)`

In particular, `String` values in Java, although they are written using quote notation, should be compared using the `equals` method:

`"Hello".equals("Hello")` ⟹ **`true`**.

`"Hello".equals("Goodbye")` ⟹ **`false`**.

Identifier Abuse

Java uses different namespaces for class names, fields, methods, and local variables. This means that it is possible to use the *same* identifier in more than one way, where the meaning of each occurrence is determined by context. This gives programmers greater freedom in picking identifiers, but poor choice of names can lead to confusion. Consider this well-formed, but hard to understand, example:

```
public class Turtle {
    private Turtle Turtle;
    public Turtle() { }

    public Turtle Turtle (Turtle Turtle) {
        return Turtle;
    }
}
```

Books about Java

There are literally hundreds of Java books available. For a good, succinct introduction to the language, we recommend the first section of Flanagan's *Java in a Nutshell*, published by O'Reilly Media [4] (the second section is mostly description of the Java libraries). It is available electronically free of charge from the University of Pennsylvania library web site. For guidelines on using Java in good style and "best practices" approaches to Java development, we recommend Bloch's *Effective Java* [2].

Chapter 20

Interfaces and Subtypes

Programming languages use *types* to constrain how different parts of the code interact with one another. Primitive types, like integers and booleans, can only be manipulated using the appropriate arithmetic and logical operations. OCaml provides a rich vocabulary of structured types—tuples, records, lists, options, functions, and user-defined types like trees—each of which comes equipped with a particular set of operations (field projection, pattern matching, application) that define how values of those types can be manipulated. When the OCaml compiler typechecks the program, it verifies that each datatype is being used in a consistent way throughout the program, thereby ruling out many errors that would otherwise manifest as failures with the program is run.

Java too is a strongly typed language—every expression can be given a type, and the compiler will verify that those types are used consistently throughout the program, again preventing many common programming errors. We saw in §19 that the primary means of structuring data and code in Java is by use of *objects*, which collect together data *fields* along with *methods* for working with those fields. A Java *class* provides a template for creating new objects.

Importantly, a class is also a *type*—the class describes the ways in which its instances can be manipulated. A class thus acts as a kind of contract, promising that its instance objects provide implementations of the class's public methods. Any instance of class `C` can be stored in a variable of type `C`.

20.1 Interfaces

Java provides a second way of giving objects types: *interfaces*. An interface gives a type to an object based on what the object *does*, not how the object is implemented. Unlike a class, an interface provides only signatures for the set of methods that must be supported by an object. As such, an interface represents a "point of view" about an object, not a prescription about how that object is implemented.

As an example, consider the following interface, which might be used to describe objects that have 2D Cartesian coordinates and can be moved:

```
public interface Displaceable {
    public int getX ();
    public int getY ();
    public void move(int dx, int dy);
}
```

Note that the interface, like a class, has a name—in this case `Displaceable`—but, unlike a class, the interface provides no implementation details. There are no fields, no constructors, and no method bodies.

Since the `Displaceable` interface is a type of any "moveable" object, we can write code that works over displaceable objects, regardless of how they are implemented. For example, we might have a method like the one below:

```
public void moveItALot(Displaceable s) {
    s.move(3,3);
    s.move(100, 1000);
    s.move(s.getX(), s.getY());
}
```

This method will work on an object created from any class, so long as the class implements the `Displaceable` interface.

To tell the Java compiler that a class meets the requirements imposed by an interface, we use the **`implements`** keyword. For example, the follow `Point` class implements the `Displaceable` interface:

```
public class Point implements Displaceable {
    private int x, y;
    public Point(int x0, int y0) {
        x = x0;
        y = y0;
    }
    public int getX() { return x; }
    public int getY() { return y; }
    public void move(int dx, int dy) {
        x = x + dx;
        y = y + dy;
    }
}
```

In addition to the private fields `x` and `y` and the constructor `Point` (which takes in the initial position for the point and sets `x` and `y` accordingly), this class provides implementations for the three methods required by the `Displaceable` interface—if one of them was omitted, or included but with a different signature, the Java compile would issue an type checking error.

In the case of `Point`, the class provides only the methods of the `Displaceable` interface. A class that implements an interface may supply *more* methods than the interface requires. For example, the `ColorPoint` class also includes a color field and a way to access it:

```
class ColorPoint implements Displaceable {
  private Point p;
  private Color c;
  ColorPoint (int x0, int y0, Color c0) {
     p = new Point(x0,y0);
     c = c0;
              }
  public void move(int dx, int dy) {
     p.move(dx, dy);
  }
  public int getX() { return p.getX(); }

  public int getY() { return p.getY(); }

  public Color getColor() { return c; }
}
```

Note that this implementation provides a different implementation of the `Displaceable` methods—it *delegates* their implementation to the `Point`

object `p`. It is easy to see how we might create other classes of objects like `Circle` or `Rectangle` that have similar implementations and all implement the `Displaceable` interface.

20.2 Subtyping

Because interfaces, like classes, are types, Java programs can declare variables, method arguments, and return values whose types are given by interfaces. For example, we might declare a variable and use it like this:

```
Displaceable d;
d = new Point(1,2);
d.move(-1,1);
```

Note that, since every `Point` object satisfies the `Displaceable` contract, this assignment makes sense—`d` holds `Displaceable` objects and all `Point` objects are `Displaceable`, so `d` can store a `Point`. Similarly, since `ColorPoints` are also `Displaceable`, we could continue the program fragment above with:

```
d = new ColorPoint(1,2, new Color("red"));
d.move(-1,1);
```

However, because `d`'s type is `Displaceable`—which only offers the `getX`, `getY`, and `move` methods—it would be a type error to try to use the `ColorPoint` method `getColor` on it:

```
Color c = d.getColor();     // Error!  d may not be a ColorPoint
```

This situation illustrates the phenomenon of *subtyping*: A type `A` is a *subtype* of `B` if an object of type `A` can meet all of the obligations that might be required by the interface or class `B`. Intuitively, an `A` object can do anything that a `B` object can, or more succinctly still: an `A` is a `B`.

Since `Point` implements the `Displaceable` interface, every `Point` is `Displaceable`, *i.e.* `Point` is a subtype of `Displaceable`. We also say that `Displaceable` is a *supertype* of `Point`.

Subtyping, as we have already seen, justifies updating a variable of type `Displaceable` to contain a `Point` object. Similarly, for a method such as `moveItALot`, it is permissible to pass an object of any subtype as the arguments of a method invocation. Thus, both of the following would be allowed:

```
o.moveItALot(new Point(0,0));
o.moveItALot(new ColorPoint(10,10, new Color("red")));
```

20.3 Multiple Interfaces

A class may implement more than one interface. This makes sense because an interface offers a "point of view" about the objects it describes, and there may be more than one point of view about the objects in a class.

For example, we might have another interface for working with shapes that have a well-defined area:

```
public interface Area {
    public double getArea();
}
```

A `Circle` class might implement both the `Displaceable` and `Area` interfaces. To do so, it simply has to satisfy the method requirements of *both*. Note that multiple interfaces are given in a comma-separated list after the `implements` keyword:

```
public class Circle implements Displaceable, Area {
    private Point center;
    private int radius;

    public Circle (int x0, int y0, int r0) {
       r = r0; p = new Point(x0,y0);
    }

    public double getArea ()  {
       return 3.14159 * r * r;
    }

    public int getRadius () { return r; }

    public getX() { return center.getX();  }

    public getY() { return center.getY();  }

    public move(int dx, int dy) {
        center.move(dx,dy);
    }
}
```

`Rectangle` would be implemented similarly:

```
public class Rectangle implements Displaceable, Area {
    private Point lowerLeft;
    private int width, height;

    public Rectangle(int x0, int y0, int w0, int h0) {
         lowerLeft = new Point(x0,y0);
         width = w0;
         height = h0;
    }

    public double getArea ()  {
       return width * height;
    }

    public getX() { return lowerLeft.getX();  }

    public getY() { return lowerLeft.getY();  }

    public move(int dx, int dy) {
       lowerLeft.move(dx,dy);
    }
}
```

Classes like `Rectangle` and `Circle` that implement multiple interfaces have multiple supertypes. The following examples are all permitted:

```
Circle c = new Circle(10,10,5);
Displaceable d = c;
Area a = c;

Rectangle r = new Rectangle(10,10,5,20);
d = r;
a = r;
```

Of course, since `Rectangle` and `Circle` are not in any subtype relation (neither is a supertype of the other) it doesn't make sense to store a `Rectangle` in a variable declared to hold `Circle` objects. Similarly, even though it is possible that a `Displaceable` variable contains a `Circle`, it is not possible to store a `Displaceable` in a `Circle` variable:

```
Circle c = new Circle(10,10,5);
Rectangle r = c;     // not OK

Displaceable d = c; // OK
Circle c2 = d;       // not OK, even though d contains a Circle
```

Chapter 21

Static vs. Dynamic

In Java, the term *static* refers to any property that is determined purely by examining the text (syntax) of the program at compile time, *i.e.* when the compiler type checks the program and generates executable bytecode. In contrast, the term *dynamic* refers to a property that is determined only when the program is actually running.

Understanding the difference between static and dynamic properties is essential to understanding the behavior of Java programs. We have already seen two instance where a distinction between the static and dynamic features is important:

- `static` methods versus "ordinary" dynamically dispatched methods, and
- the static type of an expression versus the dynamic class of the object the expression denotes

This Chapter explores each of these in turn.

21.1 Static vs. Dynamic Methods

Most of the time in Java, executable code is packaged together with an object that provides some encapsulated local state (its member fields) that is modified or otherwise used by the methods. A typical example is the `move` method of the `Point` class—it updates a `Point` object's local coordinates by displacing them by some amount. Crucially, the code of a method like

move makes sense only in the context of a Point or other Displaceable object. Therefore the move method must always be invoked relative to some receiving object o as in the expression o.move(10,10).

This "ordinary" method invocation is a *dynamic* property of the program—its behavior can't be determined until the program is actually running. To see why, recall that a variable of type Displaceable might store an object of *any* class that meets the required interface obligations. This means that, in general, many different possible implementations of the move method might be called when the expression o.move(10,10) is evaluated. Here is a simple example:

```
Displaceable o;
if (...) {
      o = new Point(10,10);
} else {
      o = new Rectangle(10,10,5,20);
}
o.move(2,2);   // method called depends on the conditional
```

Suppose the the omitted conditional test depended on user input. Then whether a Point or Rectangle is assigned to o can't be known until the program is actually run and the input is resolved. Therefore the method invocation o.move(2,2)) after the conditional might require executing either the Point version of move or the Rectangle version. This is called dynamic dispatch—the method invocation "dispatches" to some version of move depending on the *dynamic class* of the object stored in o.

Java also supports a notion of **static** methods (and fields), which are associated with a class and *not* object instances of the class. The standard example is the main method, which must be declared with the following signature (in some class):

```
public static void main(String[] args) {
    ...
}
```

As the use of the keyword **static** implies, which code is called when a **static** method is invoked can be determined at compile time (without running the program). The way this works is that, rather than invoking a static method from an object, **static** methods are called relative to the class in which they are defined. For example, suppose that a class C defines a static method m, like this:

```
public class C {
    public static int m(int x) {
        return 17 + x;
    }
}
```

`C`'s `m` method can be invoked from anywhere by using an expression of the form `C.m(3)`. Here, the class name `C` says which method named `m` will be called (in general, there can be many methods named `m`, perhaps defined in different classes).

Thus, `static` methods act like globally-defined functions. Similarly, `static` fields behave like global variables that can be referenced by "projecting" from the name of the defining class. Such `static` fields cannot be initialized in a constructor for the class (since they aren't associated with objects) and so must be initialized in the class scope itself. For example, we might modify the class `C` above to use a static field like this:

```
public class C {
    private static int big = 23;

    public static int m(int x) {
        return big + x;
    }
}
```

A `static` method cannot access non-static fields or call non-static methods associated with the class because, as there is not object to provide the state for the non-static fields of the class, those methods might not be well defined. For example, the following example will cause the Java compiler to issue an error:

```
public class C{
    private static int big = 23;

    private int nonStaticField;

    private void setIt(int x) {
      nonStaticField = x + x;
    }

    public static int m(int x) {
        setIt(x);      // can't be called because m is static
        return nonStaticField + x; // nonStaticField cannot be used
    }
}
```

When should static methods be used? Generally they should be used for implementing functions that don't depend on any objects' states. One good source of examples is the Java `Math` library, which defines many standard mathematical functions like `Math.sin`, or the various "type conversion" operations, like `Integer.toString` and `Boolean.valueOf`.

21.2 Static Types vs. Dynamic Classes

The discussion above implicitly relies on another key idea: the difference between the *static type* of an expression, and the *dynamic class* of the value that the expression denotes. Again, the difference is whether the information is determined at compile time, or whether it isn't available until the program is run.

Consider the following example code, which uses the `Displaceable` and `Area` interfaces along with the shape classes `Point` and `Circle` that were introduced earlier.

```
Point p = new Point(10,10);
Circle c = new Circle(10, 10, 10);
Displaceable d1 = p;
Displaceable d2 = c;
Displaceable d3;
if (...) {
  d3 = p;
} else
  d3 = c;
}
Area a1 = c;
Area a2 = d2;        // This assignment will cause a type error
```

Here, the type annotations on the variable declarations indicate the static type information about the variable that the compiler checks while type checking the program. For example, `p` has the static type `Point`, while `d1`, `d2`, and `d3` all have static type `Displaceable`.

At run time, the variable `d1` will always store a value whose class is `Point`, but the variable `d3` might end up with a value whose dynamic class is `Point` or `Circle`.

Note that it is the static type of a variable that restricts how the value can be used by the program. For example, even though `d2` always stores a `Circle` at run time, because `d2`'s static type is `Displaceable`, it isn't possible to assign the value in `d2` to `a2`, which expects to be given an object of type `Area`.

Also note that the static type is associated with a program *expression*, which can be a complex construction involving multiple method calls or other operations. For example and expression like:

```
(new Circle(10,10,10)).getCenter()
```

has static type `Point`. Note that, due to *subtyping*, such an expression might have more than one valid static type—for example this particular expression also has type `Displaceable`, because `Point` is a subytpe of `Displaceable`.

At run time, such expressions evaluate to values, which, if they are objects (and not a primitive value like an integer), are always associated with a single *dynamic class*, namely the class whose constructor was invoked to create the object. For this reason, the dynamic class of the variable `a1` above will be `Circle`. As we saw in the discussion above about the difference between dynamic and static method invocation, it is the dynamic class of the object that determines which method will be invoked at run

time. After running the code fragment above, `d3.move()` will either call the `Point` or the `Circle` implementation of `move`, depending on which way the conditional branch evaluates at runtime.

Chapter 22

Java Design Exercise: Resizable Arrays

Before moving on to Java's more advanced object-oriented features, like inheritance and overriding, let us first work through a design exercise that emphasizes the use of *encapsulation* as a means to enforce program invariants. Along the way, we'll introduce Java's array types, which provide a convenient way to collect together a sequence of similar data values.

22.1 Arrays

Java, like most programming languages (including OCaml), provides built-in support for *arrays*. An array is a sequentially ordered collection of element values that are arranged in the computer's memory in such a way that the elements can be accessed in constant time. The array elements are *indexed* with integer positions as shown in Figure 22.1. Arrays thus provide a very efficient way to structure large amounts of similar data.

In Java, array types are written using square brackets after the type of the array's elements. So `int[]` is the type of arrays containing `int` values, ancd `Counter[]` is the type of arrays containing `Counter` values.

If `a` is an array, then `a[0]` denotes the element at index `0` in the array. Similarly, if `e` is any expression of type `int` and `e` $\Longrightarrow$ `i`, then `a[e]` denotes the element at index `i`. Note that Java array indices start at `0`, so a 10-element array `a` has values `a[0]`, `a[1]`, ..., `a[9]`.

If you try to access an array at a negative index or an index that is larger

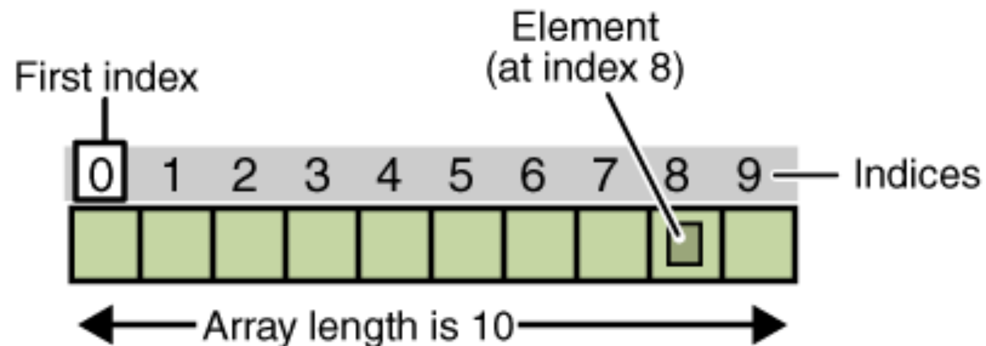


Figure 22.1: Arrays and array indices.

than (or equal to) the array's length, Java signal that no such element exists by raising an `ArrayIndexOutOfBoundsException`. Every array object has an `length` field, which can be accessed using the usual "dot" notation: the expression `a.length` will evaluate to the number of elements in the array `a`. Note that `a[a.length]` is always out of bounds—the largest legal array index is always `a.length - 1`.

Array elements are mutable—you can update the value stored in at array index by using the assignment command: `a[i] = v;`

The **`new`** keyword can be used to create a new array object by specifying the array's length in square brackets after the type of the object. The program snippet below declares an array of ten `Counter` values:

```
Counter[] arr = new Counter[10];
```

Note that array types, like `Counter[]`, never include any size information, but the length of the array is *fixed* when it is created using the **`new`** operation. Once created, an array's length never changes.

When an array is created, the elements of the array are initialized to the default value of the corresponding element type. For numeric values like integers and floating points, the default values is zero, for objects, the default is **`null`**. Java also provides syntax for *static initialization* of arrays, for the case where the array values are known when writing the program. In this case, the elements are written as a comma-separated sequence inside of `{` and `}` brackets. Here are some examples:

```java
int[] a = new int[4];
int[] b = a;
a[2] = 7;
System.out.println(b[2]);
```

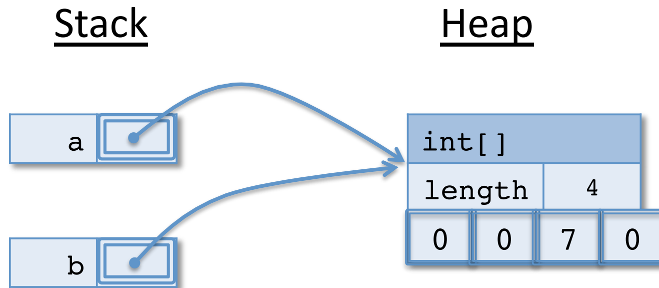


Figure 22.2: This figure shows a portion of the stack and heap configuration that would be reached by running the program above. Observe that, like other kinds of mutable reference values, array references can alias.

```java
int[] myArray = {10, 20, 30, 40, 50, 60, 70, 80, 90, 100};

String[] yourArray = { "foo", "bar", "baz" };

Point[] herArray = { new Point(1,3),  new Point(5,4) };
```

As you would expect, creating a new array object allocates space in the program's heap in which to store the array values—the amount of space required is proportional to the length of the array. The array object also includes an immutable field, `length`, which contains the size of the array when it was created. When you declare a variable of array type (or of any other object type, for that matter), Java creates a *reference* to the array on the stack. Because array elements are mutable, all of the issues with aliasing (recall §14) arise here as well. Figure 22.2 shows the program stack and heap configuration that arises in a typical array program. We will study the Java version of the abstract stack machine in more detail later (see §23).

Array iteration

Arrays provide efficient and convenient "random access" to their elements, since you can simply look up a value by calculating its index. Often, however, it is useful to process *all* (or many) of the elements of an array by *iterating* through them. Like most imperative languages, Java provides `for` loops for just that purpose. Java's `for` loop syntax is inherited from the C and C++ family of languages. Here is an example:

```
public static double sum(double[] arr) {
    double total = 0;
    for (int i = 0; i < arr.length; i++) {
        total = total + arr[i];
    }
}
```

This method takes an array of `double` width floating-point values. The `for` loop shows the canonical way of iterating over every element of the array. The `int i = 0;` part initializes a loop index variable `i` to `0`, the starting index of the array. The loop *guard* or *termination condition*, `i < arr.length;` indicates that this loop should execute while the value of `i` is less than array length. The loop variable update `i++` is short-hand for `i = i + 1`; it says that index `i` should be incremented by one after each loop iteration. The body of this `for` loop simply accumulates the total value of all the elements in the array.

The general form of a `for` loop is:

```
for (init; cond; update) {
    body
}
```

Here, `init` is the initializer, which may declare new variables (as in the `sum` example above). The `cond` expression is a `boolean` valued test that governs how many times the loop executes—when the `cond` expression becomes `false`, the loop terminates. The `update` statement is executed after each loop iteration; it typically modifies the loop variables to move closer to the termination condition.

Java also provides a `while` loop construct. The example above could be written using `while` loops like this:

```
public static double sum(double[] arr) {
    double total = 0;
    int i = 0;     // loop index initialization
    while (i < arr.length) {     // loop guard
        total = total + arr[i];
        i++;       // loop index update
    }
}
```

Using `for` loops to iterate over arrays is often quite natural, since the number of times the loop body executes is usually determined by the length of the array. As we shall see (§27.4), Java also provides support for iterating over other kinds of datatypes, particularly collections and streams of data.

Whether you use `for` or `while` loops, pay particular attention to the loop guards and indexing—a common source of errors is starting or stopping iteration at the wrong indices, which can lead to either missing an array element or an `ArrayOutOfBoundsException`.

Multidimensional Arrays

Since an array type like `C[]` is itself a type, one can also declare an array of arrays—each element of the "outer" array is itself an array. Such arrays have more than one dimension, so Java allows multiple levels of indexing to access the elements of the "inner" array:

```
String[][] names =
      {{"Mr. ", "Mrs. ", "Ms. "},
       {"Smith", "Jones"}};

// prints "Mr. Smith"
System.out.println(names[0][0] + names[1][0]);
// prints "Ms. Jones"
System.out.println(names[0][2] + names[1][1]);
```

This example shows that array initializers can be nested to construct multi-dimentional arrays. An expression like `names[0][2]` should be read as `(names[0])[2]`, that is, first find the array at index `0` from `names` and then find the element at index `2` from that array. For the example above, `names[0][2]` $\Longrightarrow$ `"Ms."`.

This example also demonstrates that the inner arrays need not be all

of the same length—Java arrays can be "ragged". This is a simple consequence of Java's array representation: An object of type `C[][]` is an array, each of whose elements is itself a reference to an array of `C` objects. There isn't necessarily any correlation among the lengths or locations of the inner arrays. Note that this is in stark contrast to languages like C or C++ in which multidimensional arrays are represented as "rectangular" regions of memory that are laid out contiguously.

The static array initialization syntax suggests that we should read the indices of a two-dimensional array as "row" followed by "column". That is, in the expression `names[row][col]`, the first index selects an inner array corresponding to a row of values, and the second index then picks the element at the corresponding column.

This view of 2D arrays is very convenient when working with many kinds of data, but for some applications, it is useful to think of the first index as the "x" coordinate and the second index as the "y" coordinate of elements in a plane. For example, for image-processing applications, we might represent an image as a 2D array of `Pixel` objects. That would allow us to write `img[x][y]` when thinking in cartesian coordinates, which looks more natural than the "row"-major `img[y][x]` indexing suggested by the row/column analysis above.

Both of these ways of thinking about 2D arrays are simply useful conventions. So long as you write your program to consistently use either the `arr[row][column]` view or the `arr[x][y]` view, it will be correct. Problems arise when these two different points of view are confused in the same program.

Note that since the inner arrays might not be all of the same length, you must use some care when writing nested loops to process all of the array elements. For example, the following program might be incorrect if the array is not rectangular:

```
int rows = arr.length;
int cols = arr[0].length;      // will fail if arr.length == 0
for (int r=0; r < rows; r++){
    // may fail if array is not rectangular
    for (int c=0; c < cols; c++) {
       arr[r][c] = ...
    }
}
```

A simpler and more robust way of iterating over all of the elements of

an array is to always use the `length` field, like this:

```
for (int r=0; r < arr.length; r++){
    // use the length of the inner array
    for (int c=0; c < arr[r].length; c++) {
       arr[r][c] = ...
    }
}
```

To create multidimensional arrays without using a static initializer, it is necessary to create an array of arrays and then initialize each of the inner arrays separately, like this:

```
int[][] products = new int[5][];
for(int row = 0; row < 5; row++) {
        products[row] = new int[row+1];
        for(int col = 0; col <= col; col++) {
                products[row][col] = row * col;
        }
}
```

This program is equivalent to the following one that uses a static initializer:

```
int[][] products =
  { { 0 },
    { 0, 1 },
    { 0, 2, 4 },
    { 0, 3, 6, 9 },
    { 0, 4, 8, 12, 16 }
  }
```

A common pitfall that you should avoid is accidentally sharing a single inner array among all of the elements of the outer array, like this:

```
int[][] arr = new int[5][];
int[] shared = new int[10];
for(int i = 0; i<5; i ++) {
  arr[i] = shared;
}
```

22.2 Resizable Arrays

The built-in arrays provided by Java are very convenient, except for one thing: their size is fixed at creation time. Sometimes, however, the number of elements that will need to be stored in the array is not known early enough. For example, suppose you wanted to keep track of an inventory of comic books, which are given issue numbers starting at 0 and increasing as each new issue is produced. Since the number of issues isn't determined in advance, it is not easy to allocate an array ahead of time. One could imagine allocating a very large array in anticipation of many issues, but that will consume a lot of wasted memory in the case that there only a few issues of the comic book produced.

A different alternative is to implement a kind of "adaptive" or "resizable" array datatype that strikes a balance between over provisioning and array-like performance. We'll call this datatype a `ResArray` for "resizable array"[1]. For the purposes of this design exercise, we will simplify things a bit and assume that the array stores only **`int`** values (which would be sufficient to keep track of the number of comics of a particular issue number in the inventory, but would not be as general as you might want).

The basic idea of the `ResArray` structure is to use a "backing" array to store the data and provide access to the elements via `set` and `get` operations. Unlike regular arrays, however, a `ResArray` acts as though its capacity is unbounded—a client of the `ResArray` can `set` and `get` any non-zero index. Of course, since the `ResArray` will use a real array internally, the `ResArray` structure will have to adjust the size of the backing array to ensure that it has enough space. This might occasionally cause the `ResArray` to have to copy the contents of the backing array to a new backing array with more room.

The `ResArray` interface

The next step of designing the `ResArray` data structure is, as usual, to specify the types of its methods. In this case, the job is made simpler because a `ResArray` acts almost as if it is a regular array—it therefore needs `set` and `get` methods that take **`int`** indices. We might also want to ask about the location of the largest non-zero element of the `ResArray`. Also, for ease

[1] Resizable arrays are loosely similar to the Java library's `ArrayList` structures.

of inter-operation with regular Java libraries, we might want to provide a way to extract an ordinary Java array from a `ResArray`. This leads us to the following skeleton for the `ResArray` class:

```
**
 * An "infinitely large array" of integers; the backing buffer
 * automatically resizes itself as new values are added.
 */
public class ResArray {

        public ResArray()

        /** access the array at position i. If position i has
         * not yet been initialized, return 0.
         *
         * @param i - index into the array
         * @return value of the array at that index
         */
        public int get(int i) {
            return 0;    // TODO: this is a stub
        }

        /** Modify the array at position i to contain value v.
         *
         * @param i - index
         * @param v - new value
         */
        public void set(int i, int v) {
            return;   // TODO: this is a stub
        }

        /** the "extent" is the size of an array that would be
         *  necessary to store the smallest prefix
         *  that contains all of the nonzero data.
         *
         * @return extent
         */

        public int getExtent() {
                return 0;  // TODO: this is a stub
        }

}
```

`ResArray` Test Cases

Programming against this stubbed-out specification, we can easily create a collection of test cases that help explain the expected behavior. We record these cases as Java JUnit tests—the `import` clauses at the top of the program bring the needed classes and operations into scope. Programming environments such as Eclipse know how to run such JUnit tests automatically, providing visual feedback about which tests pass and which tests fail. Running the tests against the stub implementation will reveal that almost all of the tests fail, as expected.

The important thing about the process of generating such tests is that it forces us to consider how the operations of a `ResArray` should interact. For example, the `getExtent` method must return `0` if we set and then zero-out an element of the `ResArray`, as shown in `TestExtent3`. Similarly, if we set two elements and then zero-out the second one, the extent should "shrink" back to the one past the index of the first element, as shown in `TestExtent4`.

The ability to design a set of good tests cases takes practice. It's often not clear how many tests to generate or when to stop making them up. A good rule of thumb is to start with tests for all of the "obvious" behaviors that match your intuitions (such as `testGet0` and `testGet1`), and then add more as you develop and debug your program. It is good practice to record each bug you find as a test case so that future changes to the program don't re-introduce old bugs.

Following these guidelines, we generate the following test program for the `ResArray` structure.

`ResArray` Implementation

Developing the test cases has revealed some of the issues we must address with the `ResArray` implementation. First, the size of the underlying array—the *backing buffer*—might be larger than the index returned by `getExtent`. The reason is shown by `testExtent3`, which first `set`s the value at index `47` to `2` and then resets it to `0`. Presumably, we must grow the backing buffer to hold at least `48` elements in order to write at index `47`, but it would be unecessary to re-size the backing buffer after resetting the value to `0`.

This analysis suggests that the `ResArray` class contains two pieces of

local state: the backing buffer itself, which is an array of integers, and an integer value that we'll call `extent`, which keeps track of (one past) the index of the last non-zero element of the `ResArray`.

How are the extent and the buffer related? The buffer must always be at least big enough to hold elements indexed up to (but not including) the extent. If the extent is smaller than the length of the buffer, then all of the elements whose indices are greater than (or equal to) the extent must be `0`. If `extent - 1` is positive, then `buffer[extent - 1]` must be non-zero.

Putting all of these considerations together, we arrive at the following `ResArray` invariant, which are defined in terms of two private fields `extent` and `buffer`:

- `0 <= extent <= buffer.length`
- if `extent = 0` then all of the elements of the `ResArray` are `0`.
- if `extent > 0` then `buffer[extent - 1]` is the last non-zero value in the `ResArray`.

Guided by these invariants, we can easily implement the first few parts of the `ResArray` class. There are two private fields, and the constructor simply creates an "empty" `ResArray`.

```
public class ResArray {

        private int[] buffer;
        private int extent;

        public ResArray() {
                extent = 0;
                buffer = new int[0];
        }
```

The `get` operation is slightly more interesting—it uses the fact that if `i < extent` then `buffer[i]` is well defined (unless `i` is negative, in which case it is appropriate to throw an array bounds exception). If the index is greater than or equal to `extent`, then the invariants tell us that we should always return `0`.

```
        /** Access the array at position i. If position i has
         * not yet been initialized, return 0.
         *
         * @param i - index into the array
         * @return value of the array at that index
         */
        public int get(int i) {
                if (i < extent) {
                        return buffer[i];
                } else {
                        return 0;
                }
        }
```

The `set` operation is more interesting still. Here there are several cases to consider. First, if the value being set is `0` and the index `i` is above the `extent`, then nothing needs to be done—all of the array invariants are preserved and there is no reason to change the backing buffer.

Second, if `i` is too big to fit into the existing `buffer` space, we must `grow` the buffer—for now, we defer defining this operation, but observe that it should modify the `buffer` field so that it's size is at least equal to `i+1`.

After possibly growing the array, it is safe to update the `i`th index to contain the value `v`. Doing so might break the `ResArray` invariants, though, so we must add code to re-establish them. In particular, if `v` is non-zero then we have to adjust `extent` in the case that `i` is larger than the old `extent`. On the other hand, if `v` is `0` and the index we're updating happens to be the *last* non-zero element of the array, then we have to search backwards through the array to find the remaining largest non-zero element. This is done in the method `shrinkExtent`.

Both `grow` and `shrinkExtent` should be **`private`** methods—they are used only internally to the `ResArray` class and so should not be exposed to outside code.

Putting all of this together, we arrive at:

```java
/** Modify the array at position i to contain value v.
 *
 * @param i - index
 * @param v - new value
 */
public void set(int i, int v) {
        if (v == 0 && i >= extent) {
                return;
        }
        if (i >= buffer.length) {
                grow(i);
        }
        buffer[i] = v;
        if (i >= extent) {
                extent = i+1;
        }
        if (v == 0 && extent == i+1) {
                shrinkExtent(i);
        }
}
```

Writing `shrinkExtent` is fairly straightforward. Its job is to search backwards through `buffer` starting from index `i` to find the last non-zero element. Once it is found (if any) then `extent` is set to one more than its index, which re-establishes the invariants:

```java
/* Adjust the extent when assigning a 0 to the buffer */
private void shrinkExtent(int i) {
        int j = i;
        while (j >= 0 && buffer[j] == 0) {
                j--;
        }
        extent = j+1;
}
```

The `grow` method must, at a minimum, create a new backing array large enough to store `i+1` elements, and then copy the old contents into the new array. A straightforward implementation would be the following:

```
/* Grow the backing buffer to accommodate up to i */
private void grow(int i) {
        int newlen = i+1;
        int [] newbuffer = new int[newlen];
        for (int j=0; j<extent; j++) {
                newbuffer[j] = buffer[j];
        }
        buffer = newbuffer;
}
```

The above implementation is perfectly functional and will meet the requirements of the `ResArray` invariant, but if you use it in practice, it might give poor performance in the case that you set a sequence of consecutive indices: `a.set(0,v0); a.set(1,v1); a.set(2,v2); ....` The problem is that each subsequent call will make a copy of *all* the preceding elements. This means that the total amount of work done to `set` an n element sequence is proportional to n^2.

A better strategy (which, incidentally, is used in many other contexts, including networking protocols like TCP) is to *double* the previous size of the array. Doing so means that, when setting a consecutive sequence as in the case above, the `grow` method will not make so many copies of the preceding elements. Intuitively, doubling the array over provisions the space needed to store the elements, but, is at most twice as much space as would otherwise be needed. Note that to maintain the `ResArray` invariants, we must still ensure that at least index `i` is included in the new buffer. We are thus led to the following implementation of `grow`:

```
/* Grow the backing buffer to accommodate up to i */
private void grow(int i) {
        int newlen = Math.max(i+1, 2 * buffer.length);
        int [] newbuffer = new int[newlen];
        for (int j=0; j<extent; j++) {
                newbuffer[j] = buffer[j];
        }
        buffer = newbuffer;
}
```

The `getExtent` method is trivial to implement:

```
        /** the "extent" is the size of an array that would be
         *  necessary to store the smallest prefix
         *  that contains all of the nonzero data.
         *
         * @return extent
         */

        public int getExtent() {
                return extent;
        }
```

Chapter 23

The Java ASM and encapsulation

The Abstract Stack Machine model that we used in chapter 15, provided a model of computation for OCaml programs that use mutable state. This stack machine lets us trace through execution in an abstract manner so that we can understand what our program does.

It turns out that we can use the *same* model for Java programs, with a few minor alterations. Like the OCaml abstract machine, the Java machine includes the same components: the *workspace*, the *stack*, and the *heap*.

23.1 Differences between OCaml and Java Abstract Stack Machines

- Almost everything, including variables stored on the stack is mutable.
- Heap values include (only) arrays and objects. Java does not include lists, options, tuples, other datatypes, records or first-class functions.
- Java includes a special reference called *null*.
- Method bodies are stored in an auxiliary component called a *class table*. For our initial discussion, we will omit some of the details about how the class table works. We will come back to it in a later chapter.

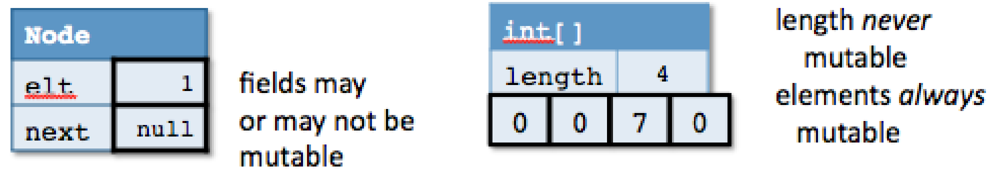


Figure 23.1: A picture of the heap of the Java Abstract Stack Machine containing two reference values: an object value (on the left) and an array value (on the right).

Reference values in the Java ASM

Java primitive types are much like the primitive types of OCaml.

There are two sorts of Java values that are stored on the heap: objects and arrays.

Objects are stored on the heap similarly to OCaml record values. The object value contains a value for each of its fields (instance variables). Fields may or may not be mutable, if they are mutable (the default in Java) we mark them with a heavy black box.

For example the Java class:

```
class Node {
   private int elt;
   private Node next;

}
```

creates object values that are represented in the heap as in the left side of Figure 23.1. Note that the only members of the class that are part of the heap value are the name of the class and the field members. The constructors and methods in the class are stored in the class table.

Likewise, Java arrays are represented in the abstract stack machine as in the right half of Figure 23.1.

```
int[] a = { 0, 0, 7, 0 };
```

Array values contain the length of the array as well as one location for each array value. The array locations are always mutable, but the length of the array never can be changed. Arrays also record the type of the array, this restricts the array to storing only certain types of values.

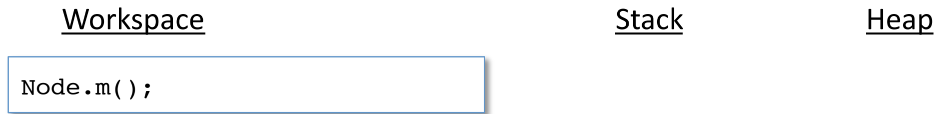


Figure 23.2: The initial state of the Java ASM example

Object Aliasing example

As in OCaml, Java object references can alias eachother. In otherwords, two reference values can point to the same location in the heap.

In what follows, we will walk through an example of object aliasing.

```
class Node {
   private int elt;
   private Node next;
   public Node (int e0, Node n0) {
     elt = e0;
     next = n0;
   }

   public static int m () {
     Node n1 = new Node (1,null);
     Node n2 = new Node (2, n1);
     Node n3 = n2;
     n3.next.next = n2;
     Node n4 = new Node (4, n1.next);
     n2.next.elt = 17;
     return n1.elt;
   }
}
```

Consider the code above. What will be the result of a call to the static method `Node.m()`? The Abstract Stack Machine can help us figure out the answer.

We start the ASM by putting the code on the workspace. The Java ASM simplifies a static method call in much the same way as it simplifies an OCaml function call.

1. The first step is to save the current workspace on the stack, marking the spot where the `ANSWER` should go with a "hole". In our example,

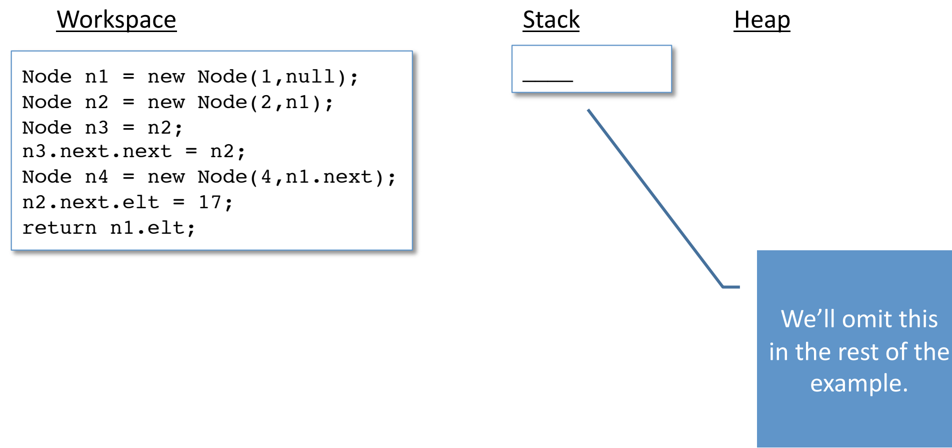


Figure 23.3: After the static method call—saving the workspace on the stack and putting the method body in the workspace.

since the original workspace was `Node.m()`, the saved workspace is merely ____.

2. The next step is to find the method definition. The Java uses the class table for this purpose, looking up the `Node` class and its method named `m`.

3. Next, the ASM adds new stack bindings for each of the called method's parameters. These stack bindings are *always* mutable in Java. In our running example, the method `m` takes no parameters, so there will be nothing pushed on the stack.

4. Next, the workspace is replaced by the body of the method.

Once the method body is in the workspace, simplification continues as usual. This process may involve adding more bindings to the stack, doing yet more method calls, or allocating new data structures in the heap. When the code of the function body reaches a `return v` statement, and the returned value has been computed, then the value is returned as the `ANSWER` to the saved workspace on the stack, plugging the hole, and popping all bindings off the stack.

Each local variable initialization in the method body adds a new (mutable) binding to the stack, which continues as long as the variable is in

scope. If a variable is declared without being initialized, Java uses the default values for the type to initialize the variable. The default value for reference types is always `null`.

The constructor invocation `new Node(1,null)` allocates and initializes an object value on the heap. We won't go into the details about how this process works at this point (we will return to these details later). For now, we can just assume that this invocation creates the object with the appropriate values for the fields.

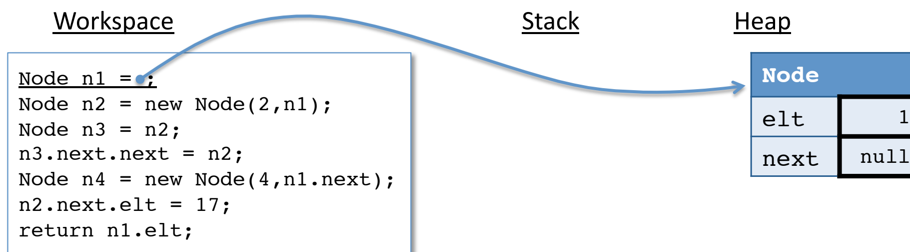


Figure 23.4: After the creation of an object. In the next step, the ASM will add the (mutable) variable `n1` to the stack, with a reference to this object.

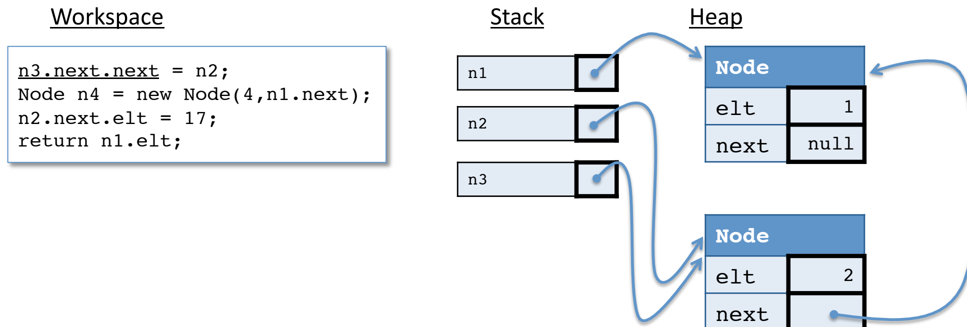


Figure 23.5: After a few more steps. The variable `n3` contains an *alias* to a previously allocated object.

After Figure 23.7, we can see what the final result of the method will be. The value of `n1.elt` is 17 when the method returns.

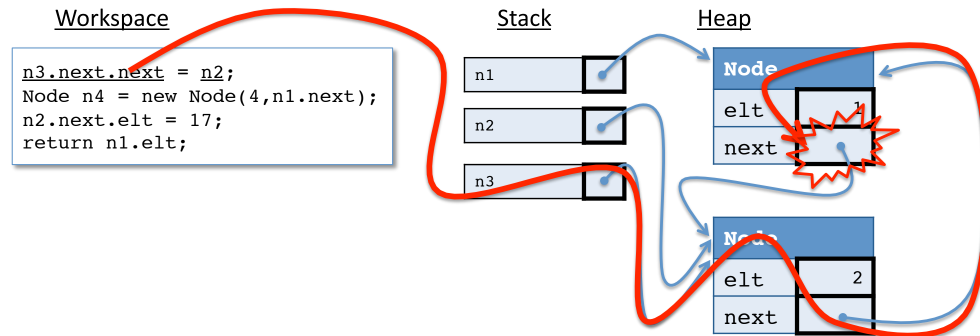


Figure 23.6: An update to a field of an object, tracing the references.

Encapsulation

Encapsulation is the process of controlling access to the state of an object. In particular, in an encapsulated object, all changes to an objects state should be done directly by the methods of the object.

The purpose of encapsulation is to preserve the *invariants* of an object. Recall that an invariant of a data structure is a property that should always hold, for any instance of that data structure, throughout its lifespan.

For example, in Chapter 22, resizable arrays have the following invariant, which states a relationship between the two private fields `extent` and `buffer`:

- `0 <= extent <= buffer.length`
- if `extent = 0` then all of the elements of the `ResArray` are `0`.
- if `extent > 0` then `buffer[extent - 1]` is the last non-zero value in the `ResArray`.

The establishment of the invariant guided the implementation of the `ResArray` class, including the code that updates the `extent` field as the `buffer` is modified.

If the class `ResArray` is encapsulated, then to assure ourselves that this invariant holds for all instances of this class, we only need to consider the methods of the object. If external methods cannot modify the state of

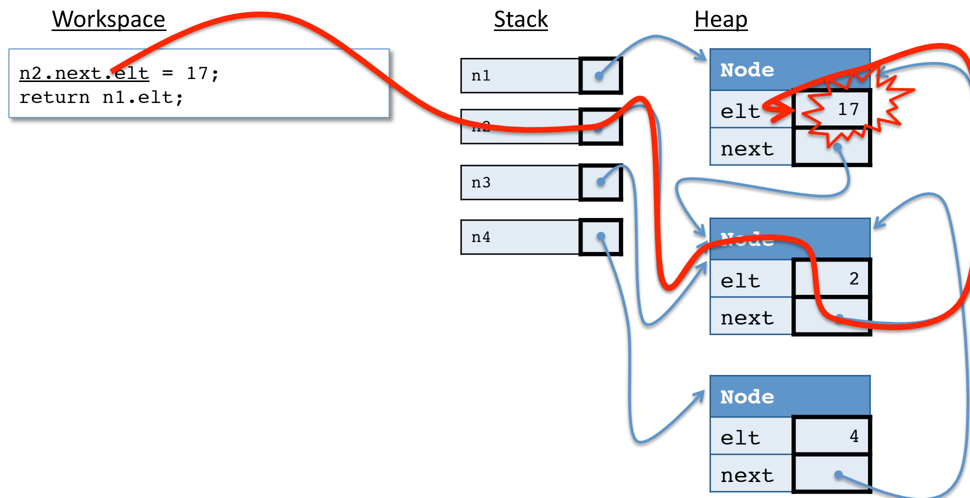


Figure 23.7: After a few more steps, the final update.

`extent` or `buffer`, then we need not look for invariant violations outside of the object's code.

However, making sure that an objects state is encapsulated can be tricky. Because of aliasing, to preserve encapsulation, we must be sure that no aliases to any of the objects internal state escape.

For example, consider extending the `ResArray` class with a values method, which returns a regular Java array, containing the nonzero values of the dynamic array. The size of this array is the `extent`.

```
        /** Return the smallest prefix of the ResArray
         *  that contains all of the nonzero values
         *  as a normal array.
         *
         * @return an array containing the values,
         *          a copy of the internal data.
         */
        public int[] values() {
            return null;  // TODO: this is a stub
        }
}
```

To implement this method, it is simple enough to create a copy, like so:

```
/** Return the smallest prefix of the ResArray
 *  that contains all of the nonzero values
 *  as a normal array.
 *
 * @return an array containing the values,
 *         a copy of the internal data.
 */
public int[] values() {
        int[] values = new int[extent];
        for(int i=0; i<extent; i++) {
                values[i] = buffer[i];
        }
        return values;
}
```

There is one slight subtlety: it would be erroneous to return the backing buffer itself. First, the extent might not agree with the size of the backing buffer. Even if the backing buffer is the same size as the `extent`, returning an alias to the encapsulated buffer would allow client programs to break the invariants by manipulating the backing buffer directly. That is, the following appealing "optimization" is wrong:

```
/** Return the smallest prefix of the ResArray
 *  that contains all of the nonzero values
 *  as a normal array.
 *
 * @return an array containing the values,
 *         a copy of the internal data.
 */
public int[] values() {
        // This is a bogus optimization: a client
        // could break ResArray invariants by
        // modifying the buffer externally.
        if (buffer.length == extent) {
                return buffer;
        }
        int[] values = new int[extent];
        for(int i=0; i<extent; i++) {
                values[i] = buffer[i];
        }
        return values;
}
```

Chapter 24

Extension and Inheritance

We saw in Chapter 20 that Java's use of interfaces to separate the *specification* of an object from its *implementation* induces a notion of *subtyping*—each class is a subtype of the interfaces it implements. In this Chapter, we will see that Java's subtyping relation is actually richer still. First, it is possible for one interface to *extend* another, by adding extra methods that must be supported. Second, one class can also *extend* or *inherit from* another, allowing the two to share common implementation code. Both of these mechanisms create new subyping relationships.

We explore each of these possibilities in turn below.

24.1 Interface Extension

Suppose, as in §20, that we were developing a paint program for working with various kinds of shapes. Previously, we had defined two interfaces, `Displaceable` and `Area` as follows:

```
public interface Displaceable {
    public int getX ();
    public int getY ();
    public void move(int dx, int dy);
}

public interface Area {
    double getArea();
}
```

If we wanted to define an interface of shapes that had both the methods

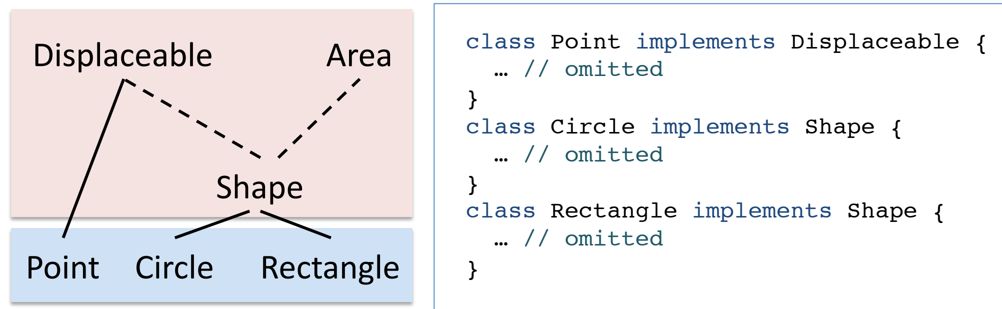


Figure 24.1: An example subtyping hierarchy for the three classes shown on the right. Subtyping induced by the `implements` keyword is shown as a solid line; that induced by the `extends` keyword is shown as dotted. Interface `Shape` extends both `Displaceable` and `Area`. Classes `Circle` and `Rectangle` are both subtypes of `Shape`, and, by transitivity, both are also subtypes of `Displaceable` and `Area`. `Point` is not a subtype of `Shape` or `Area`.

of the `Displaceable` and `Area` interfaces, along with a new method for obtaining the shape's `boundingBox`, we could define it directly like this:

```
public interface Shape {
    public int getX ();
    public int getY ();
    public void move(int dx, int dy);
    public double getArea();
    public Rectangle getBoundingBox();
}
```

However, this approach is not very good. Even though this `Shape` interface has all of the methods of both the `Displaceable` and `Area` interfaces, if we wanted to implement a class that could be used either as a `Shape` or as `Area` or as a `Displaceable` object, we would have to explicitly declare that it is a subtype of all three interfaces in the class's `implements` clause, like this:

```
public class SomeShape implements Shape, Area, Displaceable {
  ...
}
```

This is neither elegant, nor is it very scalable—in a large development, we might have to write down many interfaces, even though many of them

share method specifications. Moreover, this approach forces us to duplicate those shared methods signatures in multiple interfaces (*i.e.* `getX()` appears once in `Displaceable` and once in `Shape`).

Interface extension solves this problem by allowing one interface to extend others, possibly also adding additional required methods. For example, we could define a better version of the `Shape` interface like this:

```
public interface Shape extends Displaceable, Area {
    public Rectangle getBoundingBox();
}
```

This declaration says that the `Shape` interface is a subtype of *both* the `Displaceable` and `Area` interfaces, and thus any class that implements `Shape` must supply all of their methods, as well as the `getBoundingBox` method required by `Shape`.

The use of interface extension means that the subtyping hierarchy can be quite rich. One interface may extend several others, and interfaces can be layered on top of one another, forming an (acyclic) graph of types related by **`extends`** edges. The resulting subtyping relationship follows these chains transitively: if `A` is a subtype of `B` and `B` is a subtype of `C`, then `A` is a subtype of `C`. Figure 24.1 shows an example hierarchy.

24.2 Inheritance

Classes, like interfaces, can also extend one another. Unlike in the case of interfaces, however, a class may only extend *one* class. Here is an example:

```
class D {
  private int x;
  private int y;
  public int addBoth() { return x + y; }
}

class C extends D {     // every C is a D
  private int z;
  public int addThree() {return (addBoth() + z); }
}
```

The way to think about class inheritance is that the *subclass*, in this case `C`, gets its own copy of all the fields and methods of the *superclass* that it **`extends`**. Given the definitions above, when we create an instance of `C`, the

resulting object will have three **private** fields (x, y, and z) and two **public** methods (addBoth) and addThree.

Just as with interface extension, a class is a subtype of the class it **extends**, so, in this case, C is a subtype of D. For this reason, inheritance should be used to model the "is a" relation between two classes: every C is a D, and wherever the program requires a D it should be possible to supply a C instead. When considering how to represent a program concepts using classes, carefully examine whether there is an "is a" relationship to be found. For example, every truck is a vehicle, so one might consider making a classes Vehicle and Truck such that Truck **extends** Vehicle.

Note that the keyword **private** means that the field or method is visible only within the enclosing class. Therefore, even though C **extends** D, the fields x and y cannot be mentioned directly within C's methods. Java provides a different keyword, **protected**, which designates a field or method as visible within the class and all subclasses, no matter where they are defined. The **protected** keyword should be used with care, however, since it is not in general possible to know how a class will be extended—if the object's local state requires the fields to satisfy an invariant, making them **protected** means that all subclasses will have to preserve the invariants. This may not be feasible when the person who's writing the subclass does not even necessarily have access to the source code of the superclass.

Constructors and super

One issue with class inheritance is that it is not possible to inherit a constructor from the superclass—a constructor must have the same name as the class, and the superclass must have a different name than a class that extends it. However, constructors often establish invariants on an object's local state, so when one class inherits from another, it is usually necessary to let the the superclass initialize the private fields provided by the superclass.

Java therefore provides a keyword **super**, which can be invoked as a method. The effect is to call the superclass constructor. Here is an example of how it could be used:

```
class D {
    private int x;
    private int y;

    public D (int initX, int initY) {
         x = initX;
         y = initY;
    }

    public int addBoth() { return x + y; }
}

class C extends D {
    private int z;

    public C (int initX, int initY, int initZ) {
        super(initX, initY);   // call D's constructor
        z = initZ;
    }

    public int addThree() { return (addBoth() + z); }
}
```

Note that `C`'s constructor uses the **`super`** keyword to invoke `D`'s constructor, and thereby initialize the private `x` and `y` fields. Such a call to **`super`** must be the first thing done in the constructor body.

An example of inheritance

Returning to the shapes example, we might consider how to share some of the implementation details among several classes. For instance, the `Point`, `Circle`, and `Rectangle` classes all implement the same functionality to meet the `Displaceable` interface. We could *share* those implementation details by creating a common superclass, like this:

```
class DisplaceableImpl implements Displaceable {
   private double x;
   private double y;
   public DisplaceableImpl(double initX, double initY) {
      x = initX;
      y = initY;
   }
   public double getX () {
      return x;
   }
   public double getY () {
      return y;
   }
   public void move (double dx, double dy) {
      x = x + dx;
      y = y + dy;
   }
}

class Point extends DisplaceableImpl {
   public Point (double initX, double initY) {
      super(initX, initY);
   }
}

class Circle extends DisplaceableImpl implements Shape {
    private double r;
    public Circle (double initX, double initY,
                 double initR) {
      super(initX, initY);
      r = initR;
    }
    public double getArea () {
      return 3.14159 * r * r;
    }
    public double getRadius () {
      return r;
    }
    public Rectangle getBoundingBox () {
      return new Rectangle(getX()-r,getY()-r,getX()+r,getY()+r);
    }
}
```

```
class Rectangle extends DisplaceableImpl implements Shape {
    private double w, h;    // width and height
    public Rectangle (double initX, double initY,
                      double initW, double initH) {
      super(initX, initY);
      w = initW;
      h = initH;
    }
    public double getArea () {
      return w * h;
    }
    public double getWidth () {
      return w;
    }
    public double getHeight () {
      return h;
    }
    public Rectangle getBoundingBox () {
      return new Rectangle(getX(),getY(),w,h);
    }
}
```

24.3 `Object`

The root of the class hierarchy is a class called `Object`. All types in Java are a subtype of `Object`—even interfaces and those classes that aren't declared to `extend` any other class. Note that since Java supports only "single inheritance" for classes, *i.e.* each class may inherit from at most one superclass or, implicitly from `Object`, the classes of the subtype hierarchy for a tree. `Object` is the root of the tree.

The `Object` class provides a few methods that are supported by *all* objects: `toString`, which converts the object to a `String` representation, and `equals`, which can be used to test for structural equality.

End:

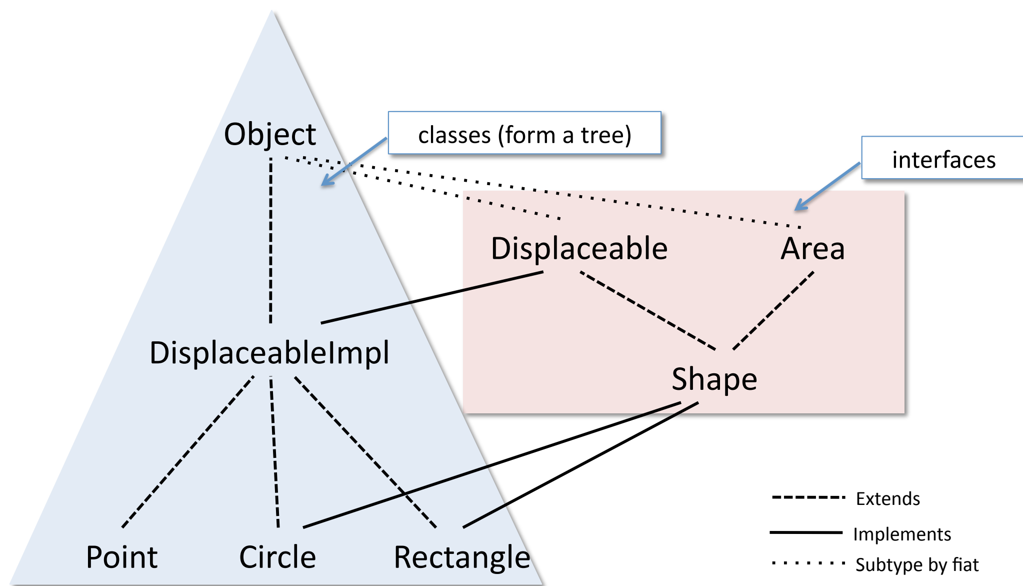


Figure 24.2: A picture of (part of) the Java subtype hierarchy. Interfaces *extend* (possibly many) interfaces. Classes *implement* (possibly many) interfaces and (except for `Object`) extend exactly one class (`Object` implicitly). Interfaces are "subtypes by fiat" of the `Object` class. Classes form a tree, rooted at `Object` (shown in blue). Interfaces (shown in pink) do not form a tree.

Chapter 25

The Java ASM and dynamic methods

In Chapter 23, we looked at simple form of the Java Abstract Stack Machine. That form included the representation of objects and arrays in the heap and explained the operation of static methods. However, in those examples, we were deliberately vague about how constructors and nonstatic method worked. We understood them at an intuitive level, but could not model them precisely in the ASM.

In this chapter we fill in the details about an essential aspect of the operation of Java, called *dynamic dispatch*. Dynamic dispatch describes the evaluation of normal method calls. It is dynamic because it is controlled by the *dynamic class* of an object—the actual class that created the object determines what code actually gets run in a method call. In a method call `o.m()`, there may be several methods in the Java Class Table called `m`. Dynamic dispatch resolves this ambiguity.

Another difficulty with modeling method calls are references in the method to the fields of the object that invoked the method. In particular, if the method `m` includes the line:

```
x = x + 1;
```

where `x` is intended to be a field, how does the ASM find that field and update it?

Understanding dynamic methods also helps us to model the execution of *Constructors* in the Java ASM, especially those from classes that extend other classes. The last refinement we make to the Java ASM in this chapter is an explanation of how constructors initialize objects when they are

created.

Our goal in this chapter is to extend the ASM enough so that we can precisely model the following example:

```
public class Counter {
  private int x;
  public Counter () {
    x = 0;
  }
  public void incBy(int d) {
    x = x + d;
  }
  public int get() {
    return x;
  }
}

public class Decr extends Counter {
  private int y;
  public Decr (int initY) {
    y = initY;
  }
  public void dec() {
    incBy(-y);
  }
}

//  somewhere in main:
Decr d = new Decr(2);
d.dec();
int x = d.get();
```

Figure 25.1: Running example of dynamic dispatch

This example includes two classes `Counter` and `Decr`, which extends `Counter`. The execution that we would like to model is at the bottom of the listing—how is it that we can create an instance of the `Decr` class? What does the constructor invocation look like? What happens when we invoke its `dec` method? What about its inherited `get` method?

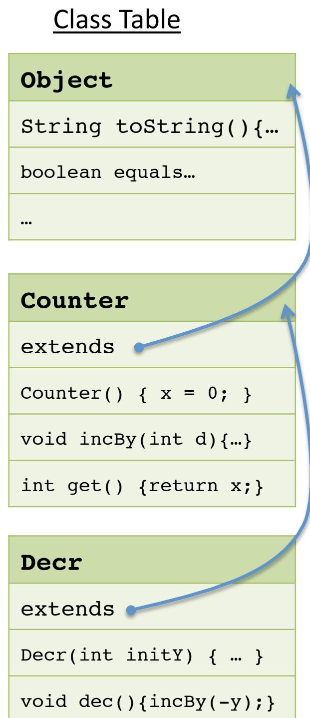


Figure 25.2: The class table for the example above

25.1 Refinements to the Abstract Stack Machine

In this chapter we explicitly add the *Class Table* as an explicit part of the Java ASM, together with the workspace, stack and heap. The class table is a special part of the heap that is initialized when the Java ASM starts. The purpose of the class table is to model the extension hierarchy (i.e. tree) among classes. Each class in the class table includes a reference to its parent or super class. It also includes code for the constructors and methods defined in that class, as well as any static class members.

The next refinement concerns the code that we run in the workspace itself. Every dynamic method allows a references to the object that invoked through a special variable called `this`. In fact, every field access `x`, is short for `this.x`. We will only execute code in the workspace where these references have been made explicit. In other words, even though we may write the code in Figure 25.1, the code that we will actually use is the one in Figure 25.3.

The code in the figure also makes one more feature of Java explicit. The first line of every constructor should start with an invocation of the constructor of the superclass of the class, using the keyword `super`. (Recall that if a class does not have an explicit superclass, its superclass is `Object`). Often we will write this invocation explicitly, especially if the superclass requires arguments. However, even if we leave it out, Java will implicitly add it. For the ASM, we will assume that all code for constructors starts with a call to `super`.

25.2 The revised ASM in action

To demonstrate how the ASM models constructors and method invocation, we next step through the operation of the ASM for the example program. The starting configuration has the given class table and puts the given code in the workspace.

Constructor invocation

The next step is to invoke the constructor for the `Decr` class, giving it the argument 2. Figure 25.5 shows the result of this action. A constructor invocation is a lot like a function call in OCaml, or a static method call in Java. It saves the current workspace, puts bindings for the parameters on the stack, and puts the body of the constructor in the workspace. In this case the `Decr` constructor takes one argument, so the binding `initY` is added to the stack.

The difference is that constructors also *allocate*, i.e. create the new object in the heap. The object values store the values of the fields (or instance variables) of the object. For a class like `Decr`, which extends another class, the fields include the fields declared in the `Decr` class plus all fields declared in the super classes (i.e. `Counter`). Furthermore, the variable `this`, a reference to the newly created object is also pushed onto the stack. Finally, the object value

Next, the code in the constructor begins to execute. The first line of the `Decr` constructor is to invoke the constructor of its super class, i.e. the `Counter` constructor. The result of this step is in Figure 25.6. As before, the current workspace is saved to the stack, and the code for the `Counter` constructor is copied from the class table to the workspace. However, this

time, because the object is already allocated, the constructor merely pushes a **this** pointer to the same heap object onto the stack. Furthermore, the `Counter` object takes no parameters, so no additional variables are added to the stack.

The first step of the `Counter` constructor is to call the constructor for *its* superclass, `Object`. Here we will skip this step—the object constructor has no effect in the ASM model. The next step is to update the `x` field of the newly created object, as shown in Figure 25.7. As the default value of the field was 0, this action does not actually change the object.

After the `Counter` constructor completes its execution, the ASM pops the stack and restores the saved workspace. The next step is to continue the execution of the `Decr` constructor, as shown in Figure 25.8.

The **this** reference directs the initialization of the `y` field of the object. This time, the field is updated to the value that was originally provided as the argument to the constructor, as shown in Figure 25.9.

After the `Decr` constructor finishes its execution, the ASM again restores the saved workspace. The reference to the new object is pushed onto the stack as the value of the local variable `d`.

The important things to remember about how the ASM treats a constructor call are:

- The newly allocated object value is tagged with the class that created it.
- The newly allocated object value includes fields that are declared in the class that constructs the object as well as *all superclasses of that class*.
- The first step of a constructor invocation is to call *its* superclass constructor. Even if the source code does not do this explicitly, Java will implicitly add this call.
- Each constructor pushes its own **this** pointer onto the stack so that it can modify the newly allocated object. However, all of these pointers refer to the same object value.

Dynamic method call

The next line of code is a dynamic method call to the `dec` method. Dynamic method calls again work like static method calls—they save the

current workspace and push their parameters onto the stack. There are two important differences between dynamic and static method calls.

1. The method body that is placed on the workspace is retrieved from the class table. The class tag in the object value (`Decr` in this case) tells the ASM where to look for this method. Because the `Decr` class contains a `dec` method, the code from that method declaration is the one that is run. This process of finding the correct method is called *dynamic dispatch* because it depends on the dynamic class of the object that calls the method. It is illustrated in Figure 25.11.

2. As well as pushing the parameters on the stack, the method also pushes a reference to the object which called the method. This reference is called **`this`**, and the method code can refer to the **`this`** variable whenever it needs a reference to the current object.

The execution of the `dec` method continues as before. However, when the code gets to the invocation of the `incBy` method, the ASM must do another dynamic dispatch to determine what code to run. This time, the method is not defined in the `Decr` class, so the ASM searches for the method in the superclasses of `Decr`. The method is found in the `Counter` class, so that is the code that is placed on the workspace for this call.

The `incBy` method modifies the value of the `x` field of the object. Note that this field is private to the `Counter` class. That means that methods in the `Decr` class cannot modify the field directly. However, even though the `Decr` class *inherits* the `incBy` method, it may modify the `x` field because it was *declared* as a part of the `Counter` class.

After the completion of the `incBy` (and `dec`) method calls, the next step in the computation is a call to the `get` method of the object. Again, the ASM uses dynamic dispatch to find the code to run (this time from the `Counter` class).

The get method returns the value -2, which is the value of the local variable `x` placed on the stack.

The important points to remember from this example are:

- When objects method is invoked, as in `o.m()`, the code that runs is determined by `o`'s dynamic class.

- The dynamic class,which is just a pointer to a class, is included in the object structure in the heap.

- If the method is inherited from a super class, determining the code for `m` might require searching up the class hierarchy via pointers in the class table.

- This process is called dynamic dispatch.

Once the code for `m` has been determined, a binding for **`this`** is pushed onto the stack. The **`this`** pointer is used to resolve field accesses and method invocations inside the code.

Static fields

Some fields in Java can be declared as **`static`**. This means that the values of those fields are stored with the *Class Table* instead of with individual objects.

```
public class C {
  public static int x = 23;
  public static int someMethod(int y) {
    return C.x + y;
  }
  public static void main(String args[]) {
  ...
  }
}

C.x = C.x + 1;
C.someMethod(17);
```

Like static methods, static fields can be accessed without having an object around. Essentially, the class table itself serves as a container for the data. Because all objects refer to the same class table, all objects have *aliases* to the static fields. Changes to the value of a static field in a method called by one object will be visible to all other objects. As a result, static fields are like *global variables*, and generally not a good idea.

The best use of static fields is for constants, such as `Math.PI`.

What about static methods?

We have already seen how static methods execute in the ASM, and the refinements of this chapter do not change that execution. But, we can now

look more closely at the difference between static and dynamic methods in Java.

In particular, the biggest difference is that static methods do not have access to a `this` pointer while they are executing. This is because they are invoked directly via the class name (e.g. `C.m()`) instead of through an object. There is no object around for the `this` pointer to refer to.

As a result, static methods cannot refer to the fields in the class that they are defined in (because there is no way for the method to access those fields) nor can they call nonstatic methods directly. (Of course, they could create a new object and call the nonstatic methods using that object.)

```java
public class Counter extends Object {
  private int x;
  public Counter () {
    super();
    this.x = 0;
  }
  public void incBy(int d) {
    this.x = this.x + d;
  }
  public int get() {
    return this.x;
  }
}

public class Decr extends Counter {
  private int y;
  public Decr (int initY) {
    super();
    this.y = initY;
  }
  public void dec() {
    this.incBy(-this.y);
  }
}

//  somewhere in main:
Decr d = new Decr(2);
d.dec();
int x = d.get();
```

Figure 25.3: Example with explicit uses of `this` and `super`.

Figure 25.4: The initial state of the Java ASM example

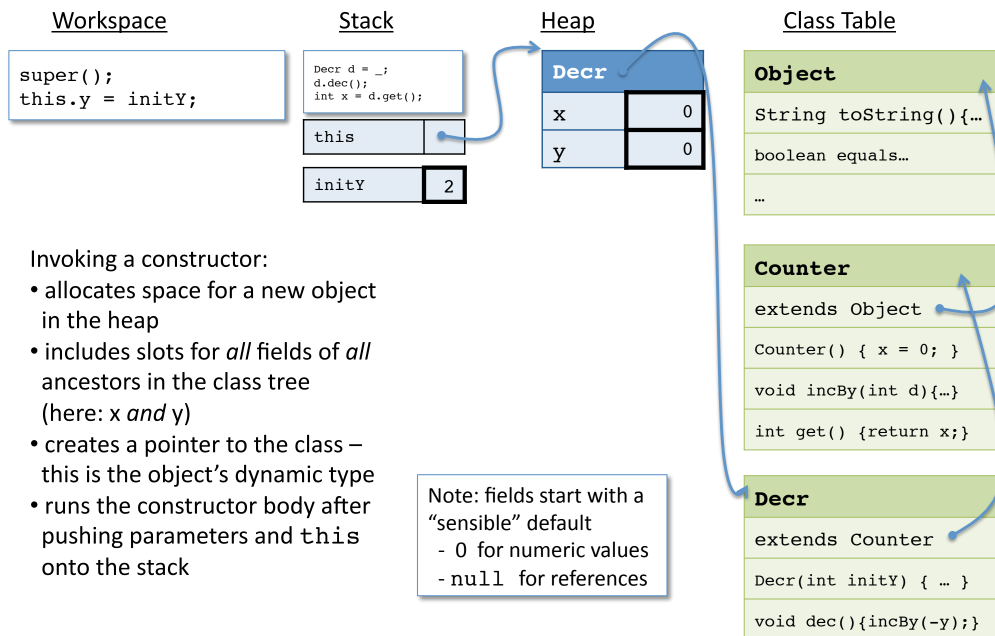


Figure 25.5: Invoking the constructor for the `Decr` class

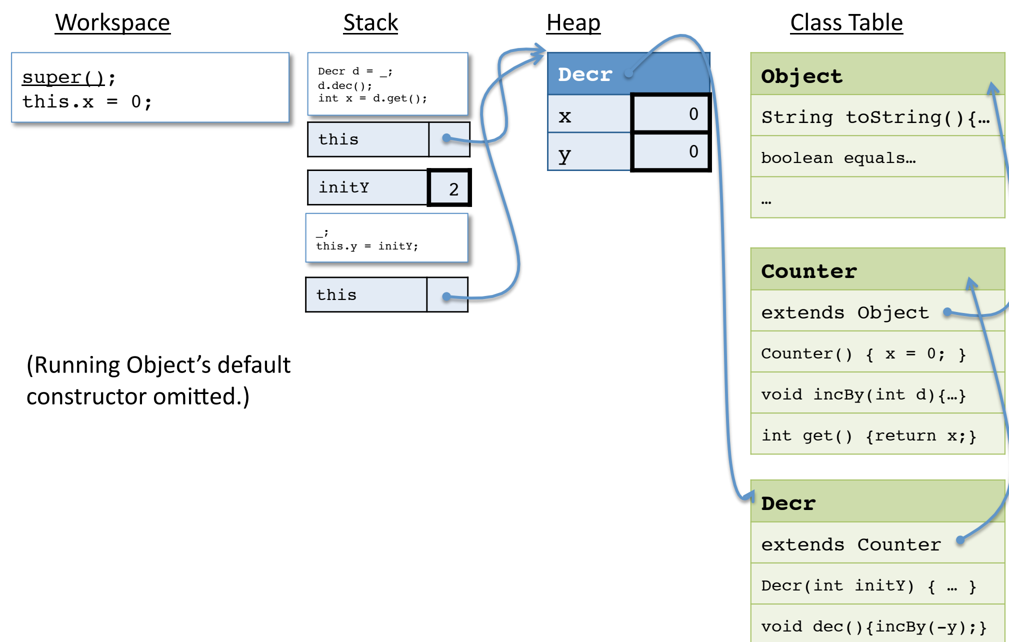


Figure 25.6: Invoking the constructor for the `Counter` class

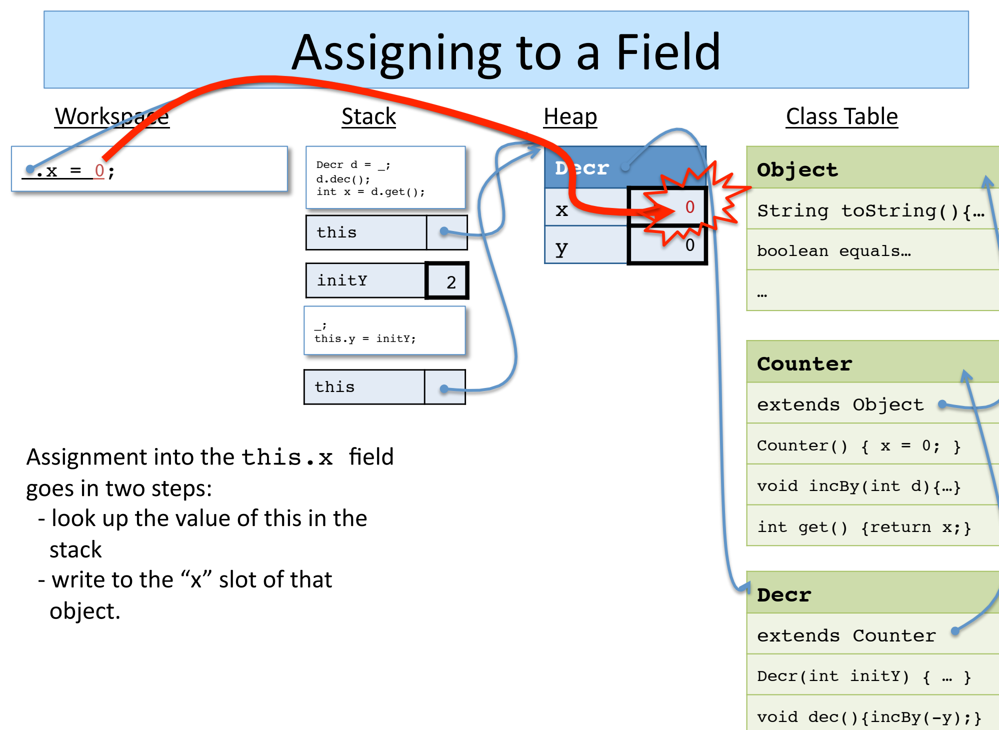


Figure 25.7: Executing the code from the `Counter` constructor

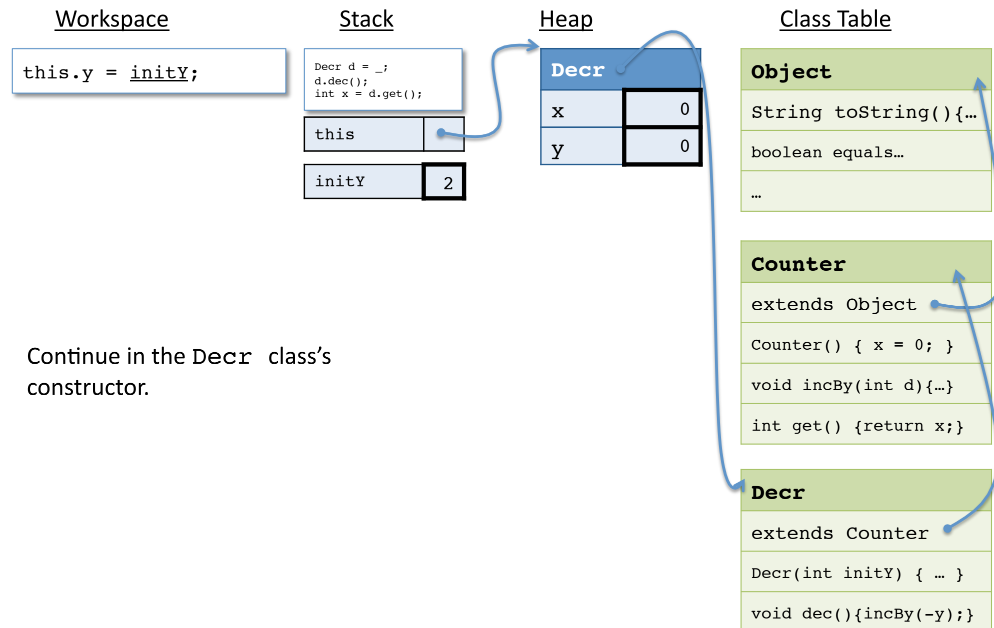


Figure 25.8: Executing the code from the `Decr` constructor

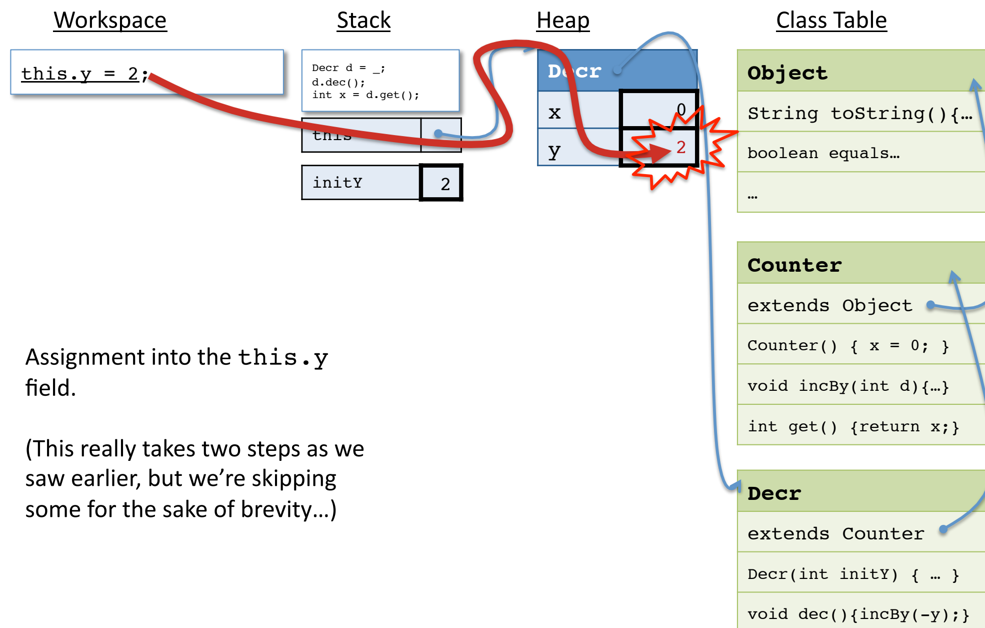


Figure 25.9: Initializing the `y` field.

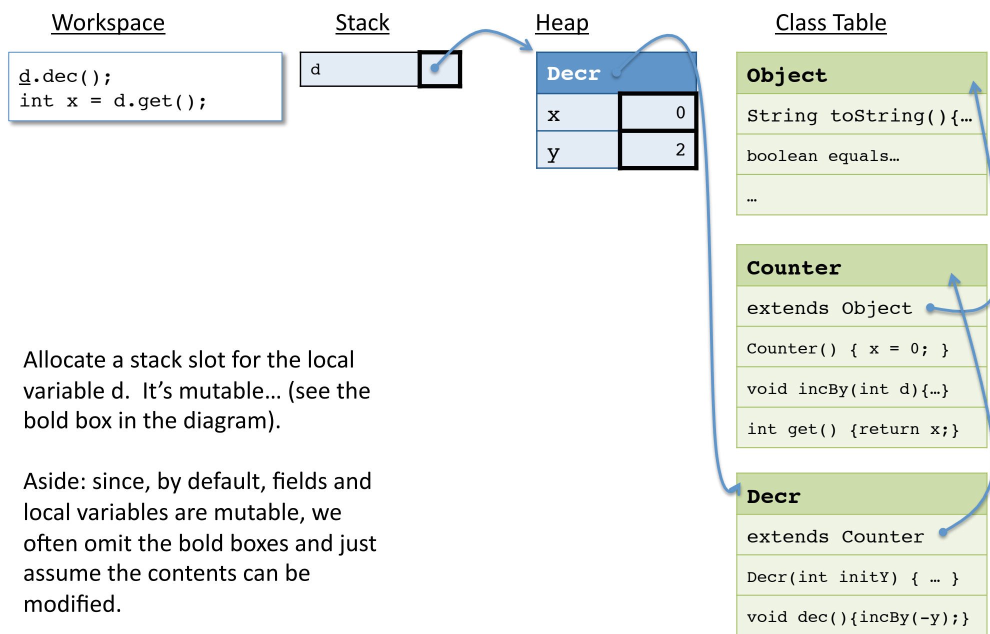


Figure 25.10: Ready to call method `dec`

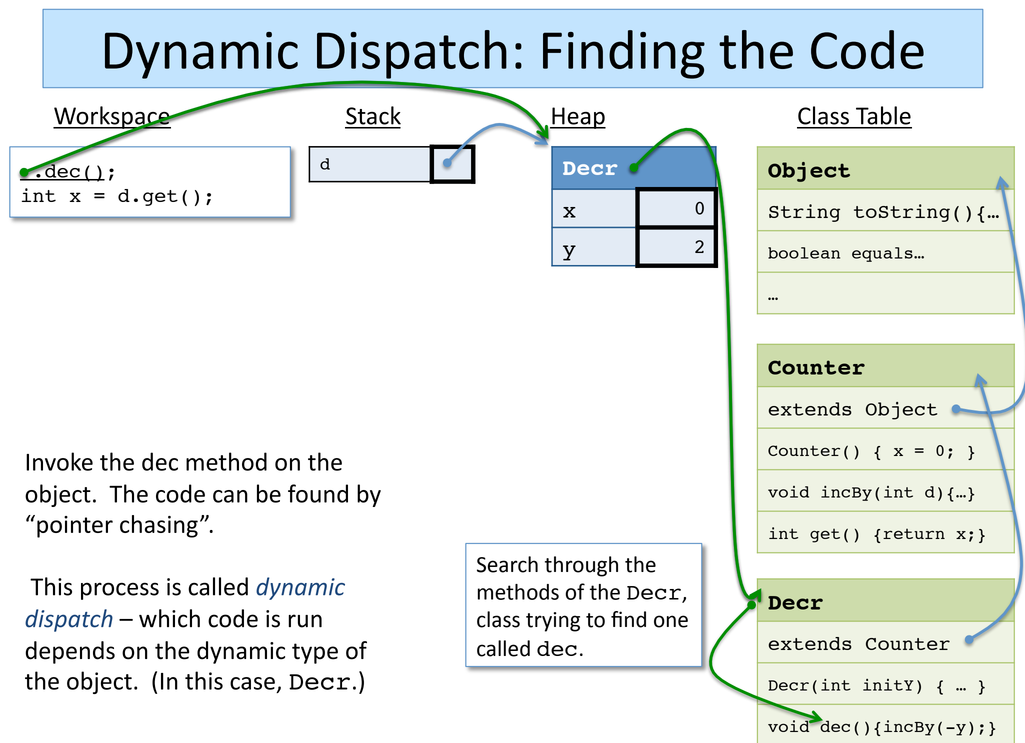


Figure 25.11: Dynamic Dispatch

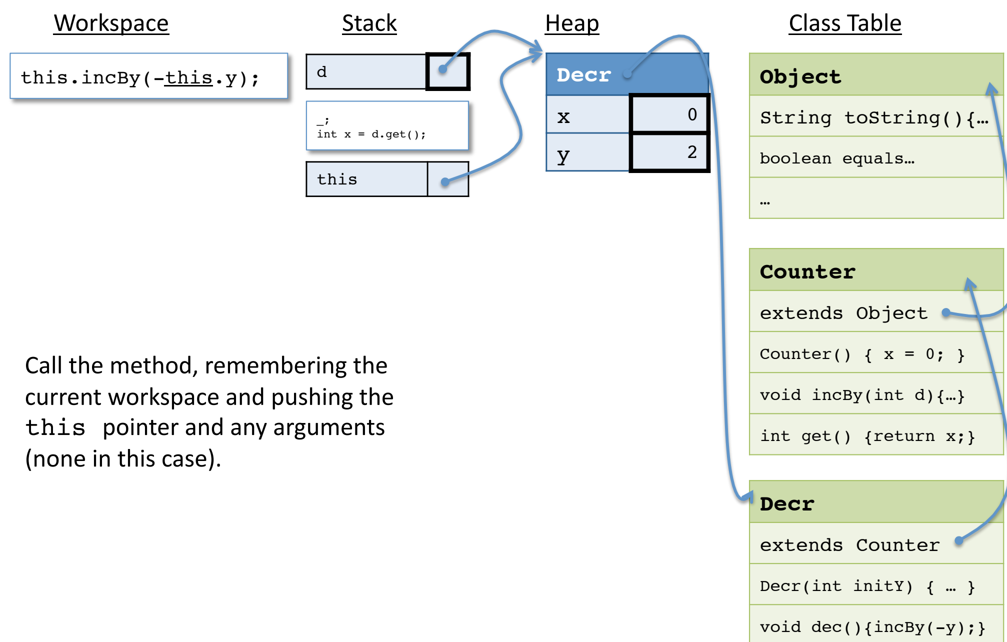


Figure 25.12: During the execution of the `dec` method

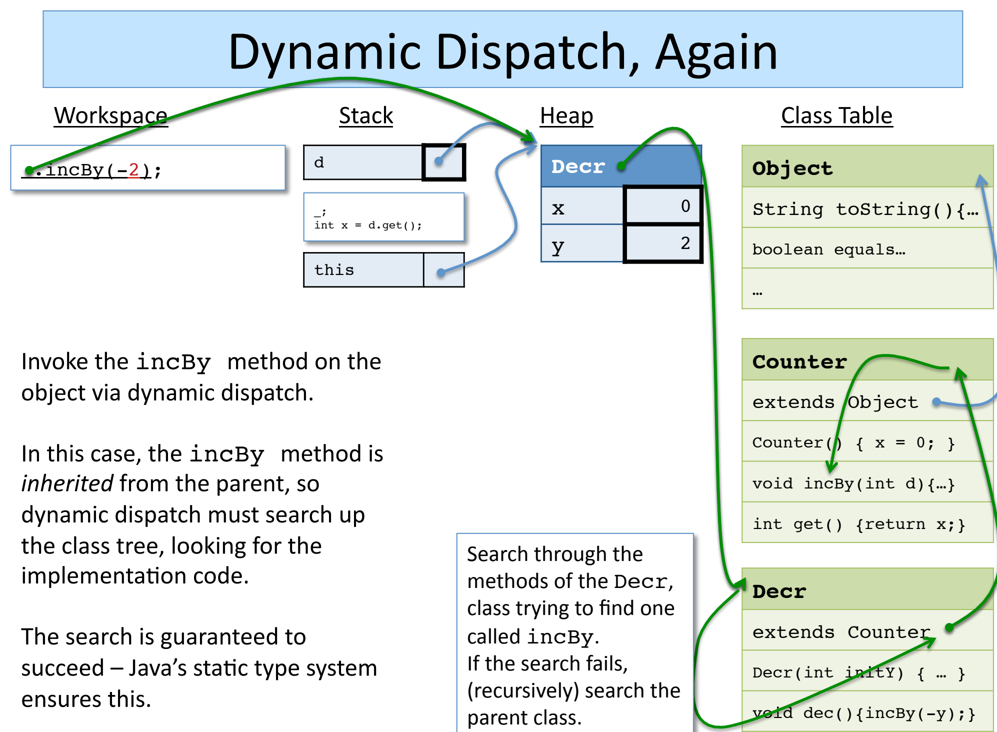


Figure 25.13: Invoking the `incBy` method

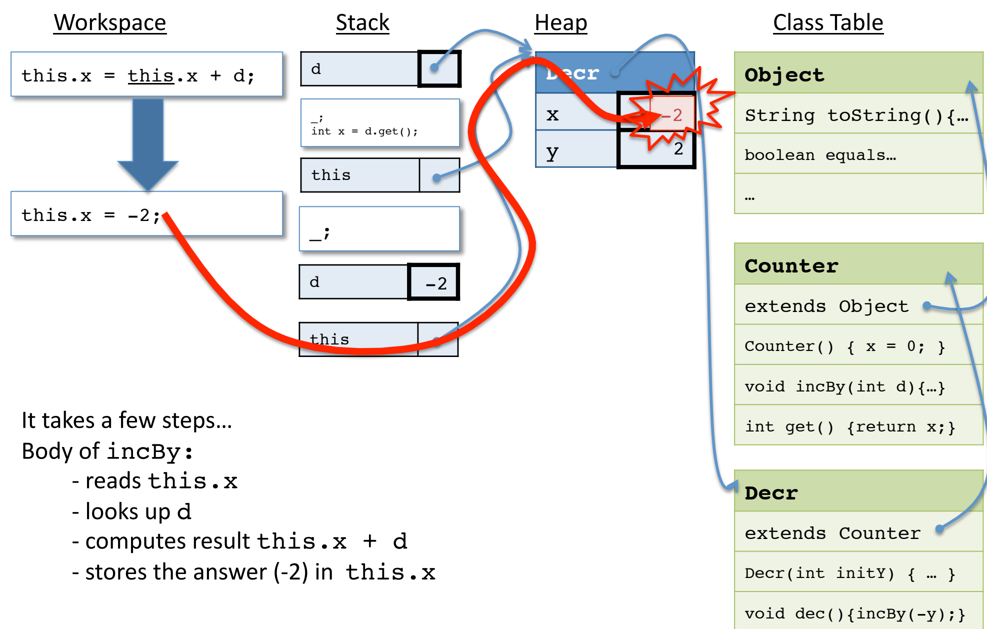


Figure 25.14: Executing the `incBy` method

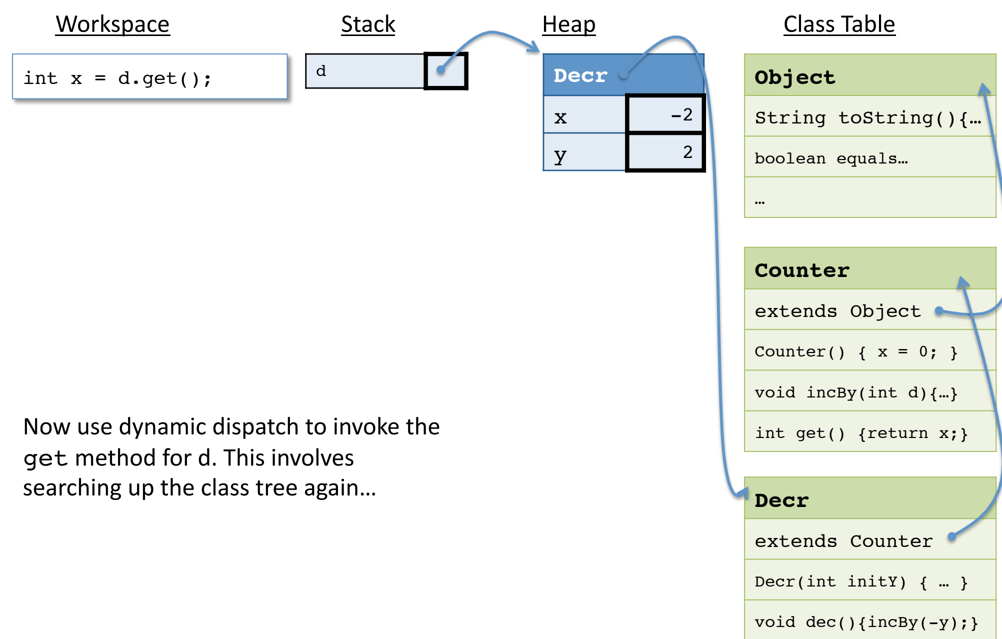


Figure 25.15: Last step: the `get` method

Chapter 26

Encapsulation and Queues

In Chapter 9, we discussed the power of type abstraction. By hiding the implementation of a data structure we can better preserve the invariants about its implementatin. For example, there are many ways to represent the mathematical idea of a set of elements—different implementations leads to different properties that must be preserved by the set operations. For example, if we represent sets by lists, we may have to maintain the invariant that there are no duplicate elements. Alternatively, if we represent sets by binary-search trees, then we must maintain the binary search tree invariants to know that our set implementation is correct.

Type abstraction allows us to reason about invariants compositionally. Because the actual representation type is abstract, no client of the set module can modify its representation. Therefore, set operations, such as `member` and `equal` can assume that their arguments satisfy the appropriate invariants, without worrying that the clients are supplying bad sets.

Java too supports this idea of hiding the implementation of a data structure so that its invariants may be maintained. In Java, this representation hiding is done via the access modifiers on the class members. Typically, the fields are declared **`private`** and access to them is provided through the **`public`** methods of the class.

If there is no way to violate the invariants of an object, then we say that the methods *encapsulate* the object state.

26.1 Queues in ML

One example of a data structure that benefitted from type abstraction in OCaml was the queue data structure. Externally, we defined queues to be an abstract type, satisfying the following OCaml interface.

```
module type QUEUE = sig
  (* type of the data structure *)
  type 'a queue

  (* Make a new, empty queue *)
  val create : unit -> 'a queue

  (* Determine if the queue is empty *)
  val is_empty : 'a queue -> bool

  (* Add a value to the tail of the queue *)
  val enq : 'a -> 'a queue -> unit

  (* Remove the head value and return it (if any) *)
  val deq : 'a queue -> 'a
end
```

Internally, queues were implemented using two related data structures—the top level queue structure itself, represented as as mutable record:

```
type 'a queue = {
    mutable head : 'a qnode option;
    mutable tail : 'a qnode option;
  }
```

and the linked list of `qnode`s:

```
type 'a qnode = {
    v : 'a;
    mutable next : 'a qnode option;
  }
```

Because the users of queues had no idea about the linked list of qnodes, there was no way for them to violate the invariants of the queue data structure. Recall:

Definition 26.1 (Queue Invariant). *A data structure of type* `'a queue` *satisfies the queue invariants if (and only if), either*

1. *both* `head` *and* `tail` *are* `None`*, or,*

2. `head` *is* `Some n1` *and* `tail` *is* `Some n2`, *and*

    - `n2` *is reachable by following* `next` *pointers from* `n1`
    - `n2.next` *is* `None`

26.2 Queues in Java

We can also implement queues in Java, and use the Java mechanisms for encapsulation to maintain the queue invariants. In this section we will walk through a Java implementation of queues and discuss how it compares to the OCaml version.

The Java queue implementation will implement the following interface.

```
public interface Queue<E> {

   /** Determine if the queue is empty*/
    public boolean is_empty ();

    /** Add a value to the end of the queue */
    public void enq (E elt);

    /** Remove the front value and return it (if any) */
    public E deq ();

}
```

We start with the interface, because this definition specifies only what a queue should do, not how to implement it. Java interfaces provide the flexibility that OCaml interfaces do. There can be several different class that satisfy this interface, and they may not necessarily implement queues with linked lists of queue nodes. However, unlike in OCaml, we cannot use the interface for encapsulation. Instead, in our implementation, we must carefully use our access modifiers so that we can restrict access to the internal state of this implementation.

Let us compare the Java interface a bit more to the OCaml interface before we continue to the implementation. The analogue to the OCaml abstract type `'a queue` is the interface name itself `Queue<E>`. Like OCaml, this type is parameterized by the elements of the queue. The Java type parameter `<E>` is the analogue of the OCaml `'a`.

In the Java interface definition, the type parameter `E` can appear throughout the interface. When this interface type is instantiated, such as `Queue<String>`, then the methods use `String` instead of `E` in their types. For example, if the variable `o` had type `Queue<String>`, then the method invocation `o.enq(x)` would only type check if `x` had type `String`.

Java Generics (i.e. the type parameters) work similarly to OCaml generics in this example. However, there is one big difference between the OCaml interface and the Java interface. The OCaml interface includes a way to construct queues, the `create` function. There is no analogue to this declaration in the Java interface. The reason is that the only type of value that could satisfy this interface in Java is an object. And objects are always created by the constructors of their classes. Interfaces do not include the constructors, and there is no easy way to abstractly construct objects.[1].

26.3 Implementing Java Queues

Now let's consider how the implementation of this interface uses encapsulation to preserve its invariants. As in the OCaml version, the Java implementation has two parts: the top-level queue data structure called `QueueImpl` (containing references to the head and the tail of the queue), and a linked list of queue nodes, each of which is an instance of the `QNode` class.

It is the responsibility of the `QueueImpl` class to encapsulate the queue nodes. Therefore, the queuenodes themselves are rather lightweight. For simplicity, we make the fields of this class public. The `QueueImpl` class will ensure that no outside code can modify the `next` reference. (For similarity with the OCaml implementation, we declare that the value stored in the node is immutable with the **`final`** keyword.)

[1]Advanced Java programmers use *factory methods* to control object creation. For more information see the classic book on Java Design Patterns [5]

```
public class QNode<E> {

   public final E  v;     // the value in the qnode is immutable

   public QNode<E> next; // next can be arbitrarily changed

   public QNode (E v0, QNode<E> next0) {
      this.v    = v0;
      this.next = next0;
   }
}
```

Note that the type of the `next` field is a `QNode<E>`, whereas in OCaml it was a `'a qnode option`. The `QNode<E>` type includes a null reference, so we do not need to make the field an explicit option. That is more convenient because we can directly access the next queue node without pattern matching. However, it is more error prone because we have to remember to make sure to check that the next queue node is not null.

The constructor for this class merely initializes the two fields.

The top-level class is responsible for encapsulating the state of the queue. Like the OCaml version, its state includes two mutable references to the head an the tail of the queue. This class is also the one that implements the queue interface—the methods of this class implement the queue operations.

```
public class QueueImpl<E> implements Queue<E> {

  private QNode<E> head;
  private QNode<E> tail;

  /** Constructor */
  public QueueImpl () {
    head = null;
    tail = null;
  }

  /** Determine if the queue is empty. */
  public boolean is_empty() {
    return (head == null);
  }

  /** Add a new value to the end of the queue */
  public void enq(E x) {
    ...
  }

  /** Remove the front value and return it (if any).
  *
  * @throws a NullPointerException if the queue is empty. */
  public E deq() {
    ...
  }
```

To encapsulate the queue state note that the `head` and `tail` fields of this class are declared private. No classes outside of this one are allowed to access these components. Furthermore, none of the methods of this class mention the `QNode` class. Other parts of the program that use this class may as well not know about the existence of the `QNode` class.

The methods of the `QueueImpl` class follow the OCaml implementation. An empty queue is one where the head and tail refernces do not point to queue nodes. We can determine whether a queue is empty by determining if the head is null.

Adding a new value to the queue means making a new queue node. Then, if the queue is empty, then we make the head and tail references point to the new node. Otherwise, we modify the node at the tail of the queue to point to the new node, and also update the tail reference.

```
/** Add a new value to the end of the queue */
public void enq(E x) {
  QNode<E> newnode = new QNode<E>(x, null);
  if (tail == null) {
    head = newnode;
    tail = newnode;
  } else {
    tail.next = newnode;
    tail      = newnode;
  }
}
```

Likewise, dequeueing from the queue modifies the linked list of queue nodes.

```
/** Remove the front value and return it (if any).
 *
 * @throws a NullPointerException if the queue is empty. */
public E deq() {
  E x = head.v;
  QNode<E> next = head.next;
  head = next;
  if (next == null) {
    tail = null;
  }
  return x;
}
```

Note that if the queue is empty, this method will throw a `NullPointerException` with the access `head.next`. There are better ways to deal with this situation in Java, but we have not covered them yet.

How could this class violate encapsulation? What could it do that would be wrong? One bad idea would be to make the head and tail public members of the `QueueImpl` class. Then, any method, not just `enq` and `deq` could modify the links in the list. It would be equally bad to for the `QueueImpl` class to return a reference to any queue node from any public method.

```
public QNode<E> getNode(int i) {
  ... find and return the ith node in the list...
}
```

Chapter 27

Generics, Collections, and Iteration

In the next few lectures, we will be covering parts of the Java standard library. In particular, we will focus on three main parts of the library:

1. The Java Collections Framework, which include a number of data structures for aggregating data values together.
2. The IO libraries, which allow Java programs to process input and output from various sources.
3. Swing, a Java GUI library.

We cover these libraries in CIS 120 for a number of reason. The formost is that they are *useful*. Part of being a good programmer is knowing how to use library code so that you don't have to write everything from scratch. Not only is the code faster to write, but it has already been debugged. It is also good style: using the abstractions of standard libraries means that others will be able to more quickly understand your code.

However, knowing *how* to use libraries is a skill in itself. The libraries are documented, and you will need to know how to read that documentation. The libraries also include features of Java that we haven't yet covered—some of those features, such as packages, generics, exceptions and inner classes, we will cover in lecture, and these libraries provide examples of those features. However, some language features you will have to learn on your own if you would like to use that part of the library.

Finally, the library designs themselves are worthy of study. This code is designed for reusability. What about the interface makes it reusable? Where does it succeed? What design patterns can you learn from it?

27.1 Polymorphism and Generics

Polymorphism is a feature of typed programming language that allows functions to take different types of arguments. The name, polymorphism, comes from the Greek word for "many shapes" because polymorphic functions work for many different shapes of data.

The Java language includes two different forms of polymorphism, *subtype polymorphism* and generics (also called *parametric polymorphism*). Although both sorts are available, it turns out that generics are more appropriate for container data types, such as found in the collections framework. To see why, consider the following comparison between the two different sorts of polymorphism in the queue interface.

Subtype polymorphism is enabled by using the type `Object` in the interface of a data structure, such as in the type of the `enq` and `deq` methods. Because every (reference) type is a subtype of `Object`, then this queue can store any type of value.

```
public interface ObjQueue {
  public void enq(Object o);
  public Object deq();
  public boolean isEmpty();
  public boolean contains(Object o);
  |
}
```

Alternatively, we can also make the queue data structure polymorphic by using Generics, as we did in Chapter 26. In this case, we parameterize the interface by the type `E`. This type appears as the argument of the `enq` method and the result of the `deq` method.

```
public interface Queue<E> {
  public void enq(E o);
  public E deq();
  public boolean isEmpty();
  public boolean contains(E o);
  |
}
```

To see the difference between these two interfaces, compare how queues with these interfaces may be used.

```
ObjQueue q = |;

q.enq("CIS 120");
___A___  x = q.deq();                   // What type for A? Object
System.out.println(x.toLowerCase()); // Does not type check!
q.enq(new Point(0.0,0.0));
___B___  y = q.deq();                   // Type for B is also Object
```

```
Queue<String> q = |;

q.enq("CIS 120");
___A___  x = q.deq();                   // What type for A? String
System.out.println(x.toLowerCase()); // Type checks
q.enq(new Point(0.0,0.0));              // Does not type check
___B___  y = q.deq();                   // Only get Strings from q
```

In the first example, suppose we have a queue `q` of type `ObjQueue`. Then we can add a string to this queue with the `enq` method because the type `String` is a subtype of the type `Object`. However, when we remove the first element of the queue, with the `deq` method, its static type is `Object`, as declared by the interface. Therefore, the next line in the example, `x.toLowerCase()`, contains a type error. The `toLowerCase` method is part of the `String` class, not and not supported by `Object`.

Alternatively, if we use generics as in the second example, then we must instantiate the queue with a specific type, such as `String`. This means that when we `deq` from the queue, then the Java type checker knows that the resulting value must be a `String`. So in this case, the method call `x.toLowerCase()` type checks.

However, the cost of permitting this method call is that all elements of the elements in the queue must be a subtype of the `String` type. Going back to the first example, it is allowed to `enq` a point into a queue that contains strings. The reason is because they are both considered `Object`s when they are in the queue. In the second example, the line `q.enq(new Point(0.0,0.0))` contains a type error. The type `Queue<String>` prevents the queue from storing anything except for (subtypes of) strings.

We could recover the heterogeneity of the first queue by instantiat-

ing the second interface with the `Object`. A queue of type `Queue<Object>` would behave exactly the same as one of type `ObjQueue`. Therefore, to give programmers the choice of behaviors, the Java Collections Framework uses Generics to give the data structures defined in the library flexible interfaces.

27.2 Subtyping and Generics

When considering the subtyping relationship between Generic types in Java, the rule to remember is that the type parameter must be *invariant*. In other words, even though type `B` may be a subtype of type `A`, it is not the case that the type `Queue<B>` is a subtype of `Queue<A>`.

Although this restriction seems counter-intuitive, without it the type checker could miss serious bugs in your code. For example, consider this program:

```
Queue<String> qs = ...                          // OK
Queue<Object> qo = qs;                          // OK?

qo.enq(new Integer(4));
String s = qs.deq(0);
s.toLowerCase();                               // oops!
```

In the first line, we create an object `qs` that stores strings. We then rename it (alias it) so that its type claims that the queue stores objects. This line is only allowed if `Queue<String>` is a subtype of `Queue<Object>`. That relationship is not true in Java, but suppose for a moment that it is allowed. Then, in the next line, we can use the alias to add a new object to the queue, because `Integer` is a subtype of `Object` (as usual). However, when we remove that integer from the queue using `qs`, and name that value `s`, we can fool Java. The static type of `s` is `String`, because that is what the result of `deq` is for in the interface `Queue<String>`. However, the dynamic class is actually `Integer`, because that is actually the value that we put in and removed from the queue. That is a problem because the dynamic class is always supposed to be a subtype of the static type. If Java thinks that an integer has type `String`, then it won't stop us from trying to call string methods on the integer.

> The basic problem with covariant generics [*i.e. making*

> `Queue<B>` *a subtype of* `Queue<A>`] is that it lets you take a bowl of apples and call it a bowl of fruit. Thats fine if youre just going to eat from the bowl and you like all fruits, but what if you decide to contribute a banana to the fruit bowl? Someone taking from the bowl expecting an apple might choke. And someone else could call the same bowl a bowl of things then put a pencil in it, just to spite you. [1]

The Java Tutorial[2] includes additional information about the behavior and meaning of Generics in Java. It goes much beyond what is covered in CIS 120.

27.3 The Java Collections Framework

The Java Collections Framework includes implementations of several of the data structures that we have already seen in CIS 120. In particular, it includes binary search trees (from Chapter 6), linked lists (from Chapter 16) and resizable arrays (from Chapter 22). It uses these data structures to implement deques, sets and finite maps (from Chapter 9).

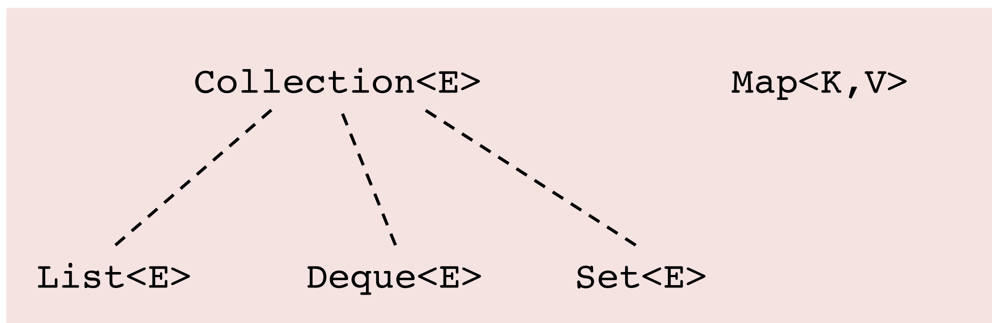


Figure 27.1: A few of the interfaces in the Java collections library.

The most important interfaces in this library appear in Figure 27.1. In particular, the collections framework unifies the interfaces for (muta-

[1] Chung-chieh Shan, "Why not covariant generics?" at `http://conway.rutgers.edu/~ccshan/wiki/blog/posts/Unsoundness/`

[2] `http://docs.oracle.com/javase/tutorial/java/generics/index.html`

ble) lists, deques, and sets under a common interface called a `Collection`. The library also includes a separate interface for finite maps, `Map`. The `Collection` interface is parameterized by `E`, the type of elements stored in the collection. While the finite map interface is parameterized by two different types `K` for the type of keys, and `V` for the type of values.

A portion of the collections interface itself is shown below.

```
public interface Collection<E> ... {
  // basic operations
  int size();
  boolean isEmpty();
  boolean add(E o);
  boolean contains(Object o);   // why not E?*
  boolean remove(Object o);

  ...
}
```

This interface includes operations for returning the number of elements in the collection (its size), determining whether the collection is empty or not, adding a new element to the collection, determining whether the collection contains a specified element, and removing a specified element. This interface is very similar to the ones that we used in the OCaml part of the course for Queues and Deques.

Note that contains and remove both take arguments of type `Object` instead of type `E`. This is because these methods use the `.equals` method from the object class to determine whether the supplied object matches the element in the collection. It could be that the provided element is not a subtype of `E`, but is still equivalent to it. (The equality method does not have to return false for objects of different types, it may only look at the common parts of the object values to determine if they are "equal". Most applications only store and remove one type of element in a collection, in which case this subtlety never becomes an issue.)

Lists and Deques are both ordered collections of elements, which we will informally call *Sequences*. Figure 27.2 shows several implementations of these interfaces in the Java Collections Framework. In particular, the `ArrayList` and `ArrayDeque` implementations are similar to the resizable array class which we developed in Chapter 22. The `LinkedList` class is similar to the deque implementation from Chapter 26. When you are reasoning about how long operations take for these implementations, you should

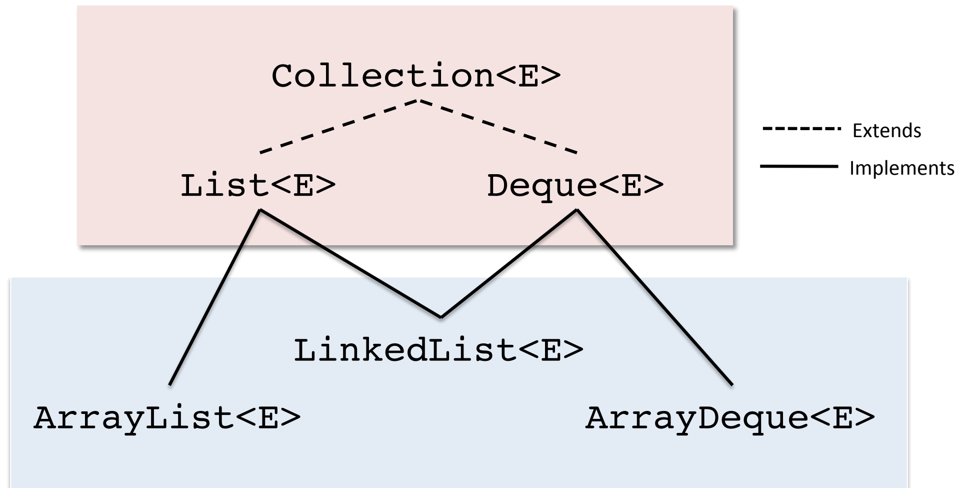


Figure 27.2: Sequence implementations.

use the related implementations as a model. For example, both `ArrayList` and `LinkedList` allow you to access elements by their position in the sequence. However, since the `ArrayList` stores the elements in an array, it is just as fast to access the last element in the list as the first one. On the other hand, accessing the last element in an `LinkedList` can take a long time if the implementation has to trace through the entire list of linked nodes to find the element. Conversely, both implementations also allow you to add elements to the middle of the sequence. In this case, it is much faster to add an element in the middle of a linked list (where it needs to only create a new node), then to add one to the middle of an array (which needs to allocate a new array and copy the elements over).

Figure 27.3 shows some implementations of the set interface (also a collection) and the map interface. The most important implementations of these interfaces are based on *hashing* (in `HashSet` and `HashMap`) and *binary search trees* (in `TreeSet` and `TreeMap`). The BST based implementation of sets actually implements a subinterface of set called `SortedSet`. This is the interface of sets that requires an ordering on elements in the set. In OCaml, all data values were ordered by the pre-defined $<$ operator. However, in this library design, each can use a specific comparison method for the elements in the set. That makes these BSTs (and BST-based maps) more

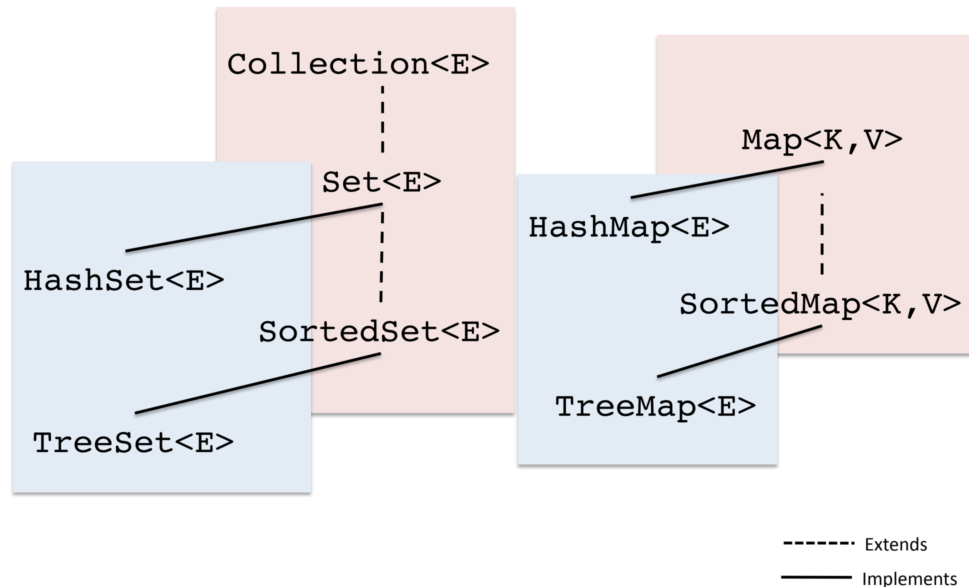


Figure 27.3: Set and Map implementations.

flexible.

Hashing is an implementation technique that is beyond the scope of this course—you'll learn more about it in CIS 120. We include those implementations here to point out the difference between sets and sorted sets.

27.4 Iterating over Collections

Java provides a powerful tool for working with all of the elements of a collection in a generic way.

As shown in Figure 27.4, the `Collections` interface in Java itself extends another interface, called `Iterable`.

This interface specifies that all collections must provide an iterator. I.e. the definition of an iterable object is one that provides at least the method `iterator`, which itself provides access to an *iterator* for the object.

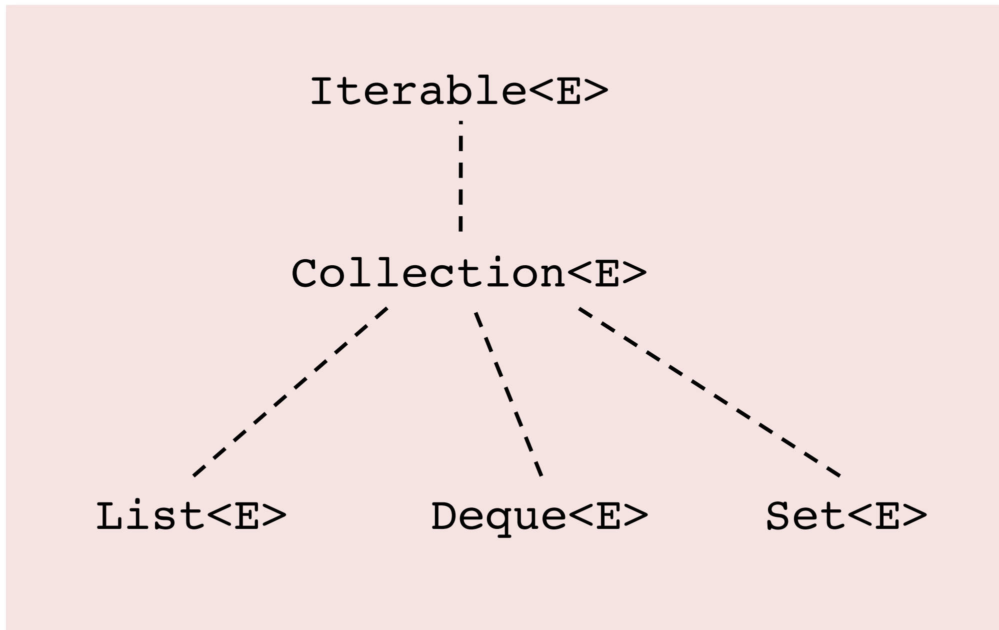


Figure 27.4: The iterable interface.

```
interface Iterable<E> {
        public Iterator<E> iterator();
}
```

An iterator object is an object whose sole purpose is to provide access to a sequence of elements. It must satisfy the `Iterator` interface, shown below.

```
interface Iterator<E> {
        public boolean hasNext();
        public E next();
        public void delete();    // optional
}
```

An iterator provides access to elements through repeated calls to the `hasNext` and `next` methods. The first method determines if there is a next element for the iterator to produce, the second method actually produces it.

For example, consider this code that catalogs the books contained in a list (called `shelf`).

```
List<Book> shelf = |  // create a list of Books

// iterate through the elements on the shelf
Iterator<Book> iter = shelf.iterator();
while (iter.hasNext()) {
        Book book = iter.next();
        catalogue.addInfo(book);
        numBooks = numbooks+1;
}
```

Because `List` is a subinterface of `Collection`, and `Collection` is a subinterface of `Iterable`, we know that the shelf has an `iterator` method. This method returns an `Iterator` for the elements in the list (which are books). The while loop executes as long as not all of the books have been accessed (or iterated). For each book supplied by the iterator (using `iter.next()`), the example adds the book information to the catalog and increments the number of books.

The iterator means that we can access the elements of the list in sequence, regardless of the actual implementation of the list. What matters is that we are doing something for each element of the list—not how we are accessing each of those elements. The iterator hides the complexity of array indexing (if we are working with an `ArrayList` for example) or of walking down a list of nodes (if we are working with a `LinkedList`). If we were to change our implementation from one implementation to another, we would not need to update this code. At the same time, it provides efficient access to the elements in either version.

Java also provides special "syntactic sugar" for working with objects that are `Iterable`. The code below uses a "for-each" loop to catalogue all of the books on the shelf. This code is equivalent to the previous two versions. The loop successively binds the variable `book` of type `Book` to each element in the collection `shelf`. For each of these elements, it executes the body of the loop.

```
// iterate through the elements on a shelf
for (Book book : shelf) {
        catalogue.addInfo(book);
        numBooks = numbooks+1;
}
```

The general for a for-each loop is shown below. The for loop declares the variable `var` of type `type` with a scope that includes the body of the

for-each loop. The loop iterates once for each element in the collection `coll`.

```
// repeat body for each element in collection
for (type var : coll) {
  body
}
```

A nice feature of Java is that arrays are also `Iterable`. For example, we could use a for-each loop to count the non-zero elements of an array with the following code:

```
int[] arr = | // create an array of ints

for (int elt : arr) {
        if (elt != 0) { cnt = cnt+1; }
}
```

In fact, *any* object that satisfies the `Iterable` interface can be used with a for-each loop.

Chapter 28

Exceptions

Programs often have to deal with unexpected inputs.

- Some methods require that their arguments satisfy certain preconditions. For example, a function that calculates the maximum value of an array requires that its input not be null or the empty array.

  ```
  public static int max(int[] arr) {
    int v = arr[0];
    for (int x : arr) {
       if (x > v) {
         x = v;
       }
    }
    return v;
  }
  ```

  The Java type checker cannot prevent this method from being called with null or a zero-length array. How can this method deal with that situation?

- Some interfaces have optional methods. This means that a class can implement this interface with a "stub" method. For example, the iterator interface includes a delete operation (which modifies the underlying sequence, removing the element most recently returned by `next` from the collection).

```
interface Iterator<E> {
   public boolean hasNext();
   public E next();
   public void delete();   // optional
}
```

However, not every iterator can support that operation. Those classes *must* define a `delete` method to implement the interface, but the only thing that method should do is report an error that it shouldn't be called with that particular iterator.

- Some parts of the program deal with external resources. Those resources may or may not be present. For example, a Java method in the IO library may try to open and read from a file on the computer's hard drive. But what if that file doesn't exist?
- Sometimes during the execution of a Java program, a resource may be exhausted. What if the program tries to write to a file on the disk, but the disk is full? What if the program makes too many nested method calls and runs out of stack space?
- Some parts of the program just may not be finished. How can a programmer indicate that some methods are not ready to be used and only present in an object to satisfy interfaces?

All of the above are *exceptional* circumstances.

28.1 Ways to handle failure

There are several ways to design methods to deal with failure and exceptional circumstances.

The simplest way is for the method to return an *error* value in the case of failure. For example, the `Math.sqrt` method returns NaN ("not a number") if its given input is less than zero. Likewise, it is common for methods to return the null reference, if the appropriate value cannot be computed.

However, there is a drawback to this practice. It puts the burden on the caller of the method to check the result to make sure that it is not the *error*

value. However, it is very easy for the caller to forget to do such checks, leading to subtle bugs in the program.

In Chapter 11 we introduced an alternative method. The option type in OCaml allows partial functions to indicate that they are partial in their type. Callers cannot forget to check the return value because the type forces them to pattern match the option. It is possible to replicate this solution in Java, using a special class to indicate possible failure.

However, although this strategy is much more likely to produce correct code, it can be tedious. The caller of every method *must* check the return value, even if there is nothing that caller can do to mitigate the failure. Furthermore, the logic of the program that deals with failure is scattered throughout the normal execution of the method, making the method difficult to read.

This chapter presents a new alternative to dealing with exceptional circumstances—the use of a language feature called *exceptions*. Most modern languages, including OCaml and Java, support this feature. Exceptions allow functions and methods to signal errors in a way that any indirect caller can handle the error, not just the most immediate one. Furthermore, if the error is not handled, the program immediately terminates, meaning that bugs are easier to detect and therefore *not* as subtle.

28.2 Exceptions in Java

An `Exception` in Java is an object (Surprise!) that represents an abnormal circumstance. The dynamic class and fields of the object indicate information about what went wrong. For example, the Java language and libraries include a number of pre-defined Exception classes to indicate run time errors. Calling a method or acessing a field with a null reference triggers a `NullPointerException`. Many IO functions return `IOException`s. Exceptions can be created, just like other objects with the **`new`** keyword.

For example, you could create your own instance of the null pointer exception class (i.e. an object with dynamic class `NullPointerException`) by invoking its constructor.

```
NullPointerException ex = new NullPointerException();
```

Creating an exception like this does not do anything special. The unique aspect of exceptions is that they can be thrown.

Throwing an exception is like an emergency exit from the current method. The exception propagates up the invocation stack until it either reaches the top and the stack, in which case the program aborts with the error, or the exception is caught. We've already seen exception throwing before, we just didn't call it that. In OCaml, the `failwith` expression throws an exception (called `Failure`) in that language. To throw an exception in Java, use the **`throw`** keyword. For example, to exit a method when its argument is three, we could throw an illegal argument exception.

```
static void m (int x) {
  if (x == 3) {
     throw new IllegalArgumentException();
  }
  // here we know that x cannot be 3.
}
```

Catching an exception lets callers take appropriate actions to handle the abnormal circumstances. This lets programs have a chance to recover from the abnormal situation and try something else. Exceptions thrown within the block of a **`try`** statement are caught using a **`catch`** block.

For example, if a file for reading is not found, then the program can give an alert to the user and ask for a different filename. For example, consider this method from a GUI for displaying images (such as in HW07). The constructor of the `Picture` class takes a string that is the filename of a new picture to show in the window. If the file is successfully read, then the new picture is displayed by this method (we've ommitted that code). However, because this process could fail, it is put inside of a **`try`** block.

```
void loadImage (String fileName) {
  try {
    Picture p = new Picture(fileName);     // could fail
    // ... code to display the new picture in the window
    // executes only if the picture is successfully created.
  } catch (IOException ex) {
    // Use the GUI to send an error message to the user
    // using a dialog window
    JOptionPane.showMessageDialog(
      frame,               // parent of dialog window
      "Cannot load file\n" + ex.getMessage(),
      "Alert",                     // title of dialog
      JOptionPane.ERROR_MESSAGE    // type of dialog
      );
}
```

If the `Picture` constructor cannot read from the specified file, then it throws an `IOException`, indicating that trouble. The **`catch`** expression after the try block includes code to execute in the case that an `IOException` is thrown. The actual exception that was thrown is bound to the variable `ex`, and information that it carries, such as an error message, can be accessed in the catch. For example, the code above uses `ex.getMessage()` to get further details about why the file cannot be read.

Note that constructors must return a reference to a newly constructed objects. They do not have the choice to return null instead, indicating an error. Exceptions are the only way that constructors can indicate difficulties in object construction.

28.3 Exceptions and the abstract stack machine

The Java ASM gives us a model of how exceptions work by demonstrating both how exception objects are represented in the heap, and what happens when exceptions are thrown and caught. Each exception handler (i.e. **`catch`** clause) is saved on the stack whenever execution reaches the accompanying try block in the workspace. When exceptions are thrown, the stack is popped until a saved exception handler is found. If there are no exception handlers, then program execution terminates. The program stops with an error.

The lecture slides from Lecture 29 demonstrate the ASM in action.

28.4 Catching multiple exceptions

The code in a try block may encounter many different sorts of exceptional circumstances. For example, when trying to load a file, several errors could occur. Perhaps the file might not exist, or perhaps the file does exist but is locked by the operating system and unable to be read. In these situations, the method for loading the file would throw different exceptions, indicating the different sources of trouble.

For this reason, Java allows try blocks to have *multiple* catch clauses. Each clause can catch a different class of exception. For example, suppose we wished to separate the handling of file not found exceptions from other sorts of IO exceptions. Then we could use the following code:

```
try {
    // do something with the IO library
} catch (FileNotFoundException e) {
    // handle an absent file
} catch (IOException e) {
    // handle other kinds of IO errors.
}
```

In this example the first clause catches any `FileNotFoundException` and the second clause catches all other `IOException`s that could be thrown by the body of the **`try`** block. Other exceptions, such as `NullPointerExceptions`, are not caught and propagate outward.

This example demonstrate that it is the *dynamic class* of the thrown exception object that determines what handling code is run. The exception handlers are examined in order and any one that binds an exception variable with some type that is a superclass of the dynamic class of the thrown exception is the one that is triggered.

Note that, in the Java IO libraries, the class `FileNotFoundException` extends the `IOException` class. This means that `FileNotFoundException` is a subtype of `IOException`. As a result, if we had exchanged the order of the exeception handlers above:

```
try {
    // do something with the IO library
} catch (IOException e) {
    // all kinds of IO errors
} catch (FileNotFoundException e) {
    // execution could never reach here
}
```

then the handler for `FileNotFoundException`s would never be triggered. If such an exception were thrown, it would be handled by the first compatible exception hander, in this case `IOException`.

All exceptions in Java are subclasses of a special class called `Exception`. This means that it is possible to create an exception handler that can handle any sort of exception whatsoever:

```
try {
    // do something
} catch (Exception e) {
    // triggers for any sort of exception
}
```

However, such exception handlers are usually *not a good idea*. If you use one in your code, you are claiming that you know how to handle any sort of exceptional circumstance whatsoever, and that is not usually the case. It is much more likely that your code will handle some sorts of exceptions, but completely ignore other sorts, such as `NullPointerException`s. This is particularly bad because those exceptions will not propagate to the top level, hiding bugs that exist in your code. And, if you cannot detect bugs early, they are much more difficult to eliminate.

28.5 Finally

Exception handlers can also include **`finally`** clauses.

A finally clause of a try/catch/finally statement always gets run, regardless of whether there is no exception, a propagated exception, a caught exception, or even if the method returns from inside the try.

These sorts of clauses are most often used for releasing resources that might have been held/created by the "try" block. For example, whenever files are opened for reading in Java, they must also be closed afterwards. The operating system can only allow so many files to be open at once,

closing the file when reading is complete means that other file operations are more likely to succeed. The code below uses the `FileReader` class to open and read from a file.

While reading from the file, an `IOException` may occur. If that is the case, then the method should still close the file afterwards.

```
public void doSomeIO (String file) {
  FileReader r = null;
  try {
    r = new FileReader(file);
    // do some IO
  } catch (FileNotFoundException e) {
    // handle the absent file
  } catch (IOException e) {
    // handle other IO problems
  } finally {
    if (r != null) {
         // don't forget null check! If the file
         // doesn't exist, then r will be null.
      try { r.close(); } catch (IOException e) {
         // handle an IOException if caused by closing the file
      }
    }
  }
}
```

28.6 The Exception Class Hierarchy

Figure 28.1 shows the relationship between several classes in Java.

All exceptions in java are instances of some class that is a subtype of the `Exception` class.

Not just exceptions can be thrown. Any object that is a subtype of the `Throwable` class is throwable. This includes exceptions and `Error`s. Errors are intended to stop the execution of a Java program. They indicate a serious runtime violation or abnormal condition. They should never be caught by the program.

Exceptions are all subtypes of the `Exception` class. Java uses the class hierarchy to make a further distinction—exception classes that extend `RuntimeException`s are treated differently than other exceptions.

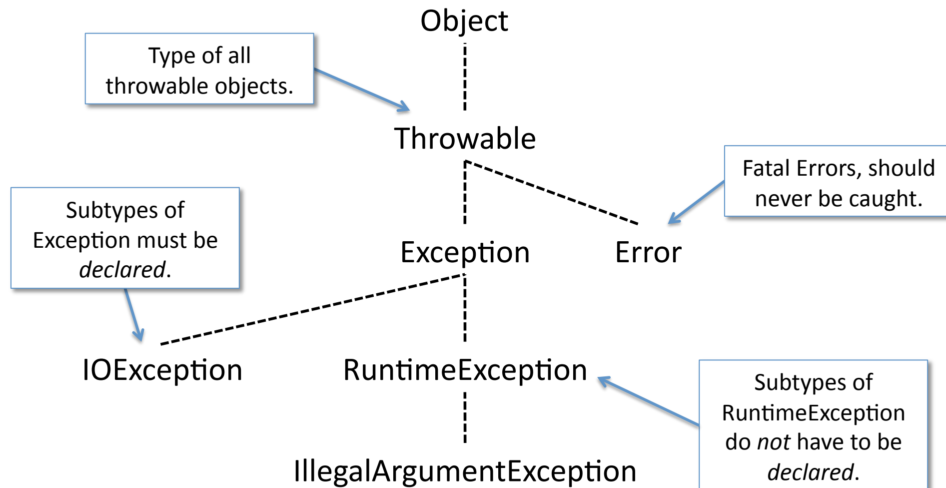


Figure 28.1: The Java Exception class hierarchy

28.7 Checked exceptions

Most methods that possibly throw exceptions (and do not handle them) must include this information in their interface, or method *declaration*. They do so by adding a **throws** clause, after the parameters of the method.

For example, this method below could throw an instance of the `AnException` class.

```
public void maybeDoIt() throws AnException {
  if (this.moonIsFull()) {
     throw new AnException();
  } else {
     // do something else
  }
}
```

Therefore, it must *declare* that this is a possiblity. If the programmer forgets to add the **throws** `AnException` to the method header, then the compile will not compile. The Java type checker will check that this exception must be declared whereever it may be thrown.

This throws clause is part of the method type. In an interface that includes the `maybeDoIt` method, the throws clause must be present as well.

```
public interface Doable {
  public void maybeDoIt() throws AnException;
}
```

That is so that every caller of the method knows that the exception could be triggered by this method call.

Methods that call other methods that could throw exceptions must also include throws annotations. For example, the method below might call the method above, and because `maybeDoIt` might throw `AnException`, this method might throw `AnException`.

```
public void maybeNot() throws AnException {
  if (this.starsAligned()) {
    this.maybeDoIt();
  } else {
    // do something else
  }
}
```

Note that if a method successfully handles a potentially thrown exception, it does not need to declare it. For example, even though the method call below, could throw `AnException`, that code is in a try block that will catch the exception and prevent its propagation to the users of the `definitelyNot` method.

```
public void definitelyNot() {
  try {
    this.maybeDoIt();
  } catch (AnException ex) {
    // ... handle the exception ...
  }
}
```

28.8 Undeclared exceptions

However, some exceptions do not need to be declared. (There is an exception to the rule about exceptions.) Methods that could throw `RuntimeExceptions` do not need to indicate this possibility.

Many of the exceptions that you have already seen are runtime exceptions. They do not need to be declared by the objects that throw them. These include `NullPointerException`, `IndexOutOfBoundsException` and

`IllegalArgumentException`. Requiring methods to declare that they could throw these exceptions would be excessive. Almost every method that uses an array or object could potentially throw a `NullPointerException`.

Why are only some exceptions declared? The original intent was that runtime exceptions represent disastrous conditions from which it was impossible to sensibly recover. Furthermore, the prevalence of potential null pointer exceptions means that every method would have to declare them, making the declaration itself tedious but meaningless.

As you define your own exception classes, you have the choice of whether they should be declared or not, by choosing whether to extend `RuntimeException` or `Exception`. Declared exceptions provide better documentation for users—the interface for the method is more explicit when it includes the potential exceptions that the caller should be aware of. With undeclared exceptions, it is easier for the callers to not even know that they should set up exception handlers for exceptional circumstances.

However, the cost of declared exceptions is that it is more difficult to maintain code as this interface must be adapted each time the code is modified. In practice, developers have not seen much benefit to declared exceptions. Test-driven development encourages a "fail early/fail often" model of code design and lots of code refactoring, so undeclared exceptions are prevalent.

As a compromise, only general purpose libraries tend to use declared exceptions, as the explicit interfaces are much more important in that situation.

28.9 Good style for exceptions

We end this chapter with some style suggestions for using exceptions sucessfully in your code.

- In Java, exceptions should be used to capture exceptional circumstances. Exception throwing and handling (i.e. Try/catch/throw) incurs performance costs and, more importantly complicates reasoning about the program. Dont use them when better solutions exist.

- Just as it is good style to re-use data structures from the standard library, it is also good style to re-use existing exception types when

they are meaningful to the situation. For example, if you are implementing your own sort of container class, it makes sense to use the `NoSuchElementException` from the Java collections framework.

- However, for exceptional circumstances that are specific to your application, it makes sense to define your own subclasses of `Exception`. By doing so, you can convey useful information to possible callers that can handle the exception. At the very least, you will give a new name to the exception class, so that it can be caught independently of other exceptions. Furthermore, because exceptions are objects, the fields of the object are a great place to store additional information about the exceptional circumstance, such as the values of variables that were in scope when the exception was thrown. Such values can help exception handlers recover from the situation more gracefully, and if necessary, provide more insightful error messages to users.

- It is often sensible to catch one exception and re-throw a different (more meaningful) kind of exception. For example, suppose you were implementing a class that provides an `Iterator` interface to a file. This class would read from the file one word at a time, and would provide the next work with each call of the `next` method. However, file reading could trigger an `IOException` if there are no more characters to be read. It is much better style if this iterator catches that exception and replaces it with a `NoSuchElementException`.

- Catch exceptions as near to the source of failure as makes sense. You could put all of your exception handling code in the `main` method of your application, which would prevent your program from being terminated by a thrown exception. However, the `main` method probably does not have enough information to recover from the situation. It is better to handle the exception as soon as possible.

- As mentioned above, it is best to catch exceptions with as much precision as possible. In other words, don't do: `try  catch (Exception e)   `. Instead use an exception handler tailored to the specific class of exception that you would like to handle. For example, `try  catch (IOException e)`

See the Java Tutorial[1] for further information about exceptions in Java.

[1] `http://docs.oracle.com/javase/tutorial/essential/exceptions/`

Chapter 29

IO

Like Chapter 27, this chapter concerns a very useful part of the Java Standard library, the *IO* library. The abbreviation *IO* is short for Input-Output. This library contains classes so that Java programs can interact with its environment.

The fundamental abstraction in the IO library is a *stream*, a communication channel to the outside world. A stream is a sequence of values that the program processes in order, and may not have a fixed end (unlike a list or array). For example, an *input* stream may consist of the sequence of characters being read from the keyboard or a file, a sequence of bytes read from a network connection. *Output* streams allow the program to send information, either to the console, a text window, a file, or the network. Values are read or written to the stream in order, one-by-one.

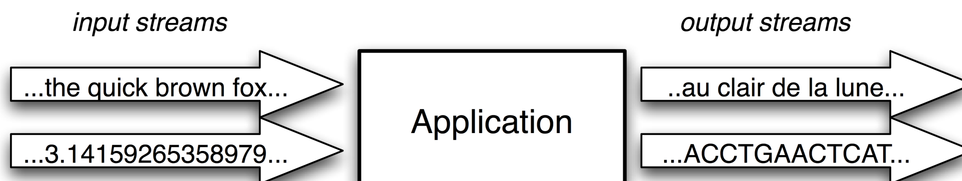


Figure 29.1: Input and Output streams

Java unifies all of these sources and sinks of data using a common type. The two abstract classes `InputStream` and `OutputStream` describe the inter-

face for basic operations in the stream hierarchy[1]

The basic operation from the `InputStream` class is a method to read data from the stream.

```
abstract int read()
  throws IOException;          // read the next byte of data
```

Likewise, the `OutputStream` class is based on writing writing data.

```
abstract void write(int b)
  throws IOException;   // writes the byte b to the output
```

In both cases, these methods are abstract—they just declare the methods that subclasses must implement. Subclasses that extend these classes will implement `read` and `write` to actually pull data from and push data to the appropriate devices (files, keyboards, network, etc.) Furthermore, although the types of these methods look like they read and write integer values, in actuality, input and output is based on *bytes*. Bytes are 8-bit integers—i.e. numbers in the range 0-255. The `read` method will never return the integer 256 or higher. Likewise, the write method will produce strange results when given numbers that are not within 0-255.

The `read` method returns -1 when the end of the stream is detected and there are no more bytes to be read.

Both of these methods could throw an `IOException` if the stream cannot be read or written to.

29.1 Working with (Binary) Files

The classes `FileInputStream` and `FileOutputStream` extend the `InputStream` and `OutputStream` classes respectively. These two classes allow you to read and write data to files. Note however, that these classes are for working with *binary* data, such as the bytes in an image file.

For example, here is a constructor for class that represents black-and-white images. The constructor for this class creates an image by reading

[1]Abstract classes are a cross between interfaces and classes. Like interfaces, they include method declarations and cannot be instantiated. However, they may also include method definitions, constructors and fields, like regular classes. They are useful for describing an interface and providing basic functionality to all subclasses based on that interface. Abstract classes are most useful in building big libraries, which is why we arent focusing on them in this course.

in a file in binary format. This simple format contains first the width, then the height of the image. After that, each byte represents a pixel in the image. Because this is a black-and-white image each pixel only contains one component, a value between 0 (pure black) and 255 (pure white).

```
class Image {
   private int width;
   private int height;
   private int[][] data;   // each value is in the range 0-255

   // create a new input stream from the specified file
   public Image(String fileName) throws IOException {
      InputStream fin = new FileInputStream(fileName);
      width  = fin.read();     // read the width
      height = fin.read();     // read the height
      if (width == -1 || height == -1) {
         throw new IOException("Height/Width not available.");
      }
      data = new int[width][height];
      for (int i=0; i < width; i++) {
         for (int j=0; j < height; j++) {
            int ch = fin.read();
            if (ch == -1) {
               throw new IOException("File ended too early.");
            }
            data[i][j] = ch;
         }
      }
      fin.close();
   }
```

Note that after the constructor is finished with the file, it *closes* the input stream. File streams are managed by the operating system. Only *open* files may be read from or written to. Furthermore, the operating system should be notified when the file access is complete by *closing* the file. This lets the operating system know that it no longer needs to manage the file.

Furthermore, this constructor explicitly throws *exceptions* indicating that the file is not in the correct format. Furthermore, some of the operations that the constructor uses to read the file could also throw exceptions. For example, if the file could not be opened by the `FileInputStream` constructor, then that constructor may throw an `FileNotFoundException`. Likewise, the `read` methods could throw `IOException`s for other sorts of errors.

Input streams provide the most basic support for working with input and output, but some times you would like more help. For example, we could make the code above more efficient by using a `BufferedInputStream`. This class groups together multiple reads into one file access, while still providing the data byte by byte. Because setting up the file access is the slow part, its much faster to read multiple bytes at once.

Using a `BufferedInputStream` is simple. This class just "wraps" another `InputStream` to add buffering. For example, we could add buffering to the example above by changing the beginning of the constructor, thus:

```
public Image(String fileName) throws IOException {
      FileInputStream fin1 = new FileInputStream(fileName);
      InputStream fin = new BufferedInputStream(fin1);
```

A `BufferedInputStream` is also an input stream, so anywhere an input stream could be used, a buffered input stream is permissible.

Buffering is not the only functionality that the Java libraries builds on top of IO streams. For example, many Java applications need to process character-based input, instead of raw binary data. Therefore, the Java library includes many more classes and higher-level abstractions to assist with this sort of stream processing.

29.2 PrintStream

For example, one class that you have already used (without even knowing about it) is the `PrintStream` class.[2] This class extends `OutputStream` with methods for printing various different types of values to the console, including `String`s. Unlike the `OutputStream` above, which only `write`s bytes, this stream can write any Java value.

The `System` class (part of Java) contains a static field called `out` that is an instance of this class.[3] So every use of `System.out.println` is calling a method of the `PrintStream` class.

The `println` method is an example of an *overloaded* method. This means

[2]See `http://docs.oracle.com/javase/6/docs/api/java/io/PrintStream.html` for the documentation of the `PrintStream` class.

[3]Note that System.out is a static member of the class System—this means that the field out is associated with the class, not an instance of the class. Recall that static members in Java act like global variables, as described in §21.

that there are actually several different variants of the `println` method in the `PrintStream` class. For example, the javadocs for this class include the following methods, useful for printing both primitive values and objects.

```
 void println()
//       Terminates the current line by writing the
//       line separator string.
 void println(boolean x)
//       Prints a boolean and then terminate the line.
 void println(char x)
//       Prints a character and then terminate the line.
void  println(double x)
//       Prints a double and then terminate the line.
 void println(float x)
//       Prints a float and then terminate the line.
 void println(int x)
//       Prints an integer and then terminate the line.
void  println(Object x)
//       Prints an Object and then terminate the line.
 void println(String x)
//        Prints a String and then terminate the line.
```

With overloading, the number and *static types* of the arguments determine which version of the method executes. The java IO library uses overloading of constructors pervasively to make it easy to glue together the right stream processing routines.

Other useful methods in the `PrintStream` class include a method to print values without terminating the line (overloaded like `println` above)

```
void print(String x)
// write x without terminating the line
       (output may not appear until the stream is
```

and a method to `flush` the output.

```
void flush()
// actually output any characters waiting to be se
```

What does the `flush` method do? The `PrintStream` class actually *buffers* output to the console. What this means is that it gathers together the output of several calls to `print` in a special data structure called a buffer. It does not display them immediately. Only when the buffer is *flushed* does the output appear in the console. Flushing happens when the buffer is full, with every newline, and when it is flushed explicitly with the `flush` method.

29.3 Reading text

For reading streams text, Java provides another abstract class called a `Reader`. Just like subclasses of the `InputStream` class provide streams of bytes, the subclasses of the `Reader` class provide streams of characters.[4] While bytes can be stored with 8 bits, characters are 16 bit values. Like `InputStream`s, `Reader`s have a `read` method. However, this time, the `read` method returns an integer in the range 0 to 65535. Also like `InputStream`s, the `read` method returns -1 when there are no more characters to be read from the input stream.

The `FileReader` class creates a reader that reads characters from an input text file. For example, given a string `fileName`, we can construct a reader as follows:

```
Reader reader = new FileReader(fileName);
```

If we would like to add buffering to the file access (as we did for input streams above) the `BufferedReader` class serves the same role as the `BufferedInputStream` class. For example, creating a file reader with buffering can be done just by wrapping the file reader with a buffered reader:

```
Reader reader = new BufferedReader(new FileReader(fileName));
```

We will walk through an example using the `Reader` class in §histogram.

29.4 Writing text

The output analogue to the `Reader` class is the abstract `Writer` class. This class is similar to the `OuputStream` class. Instead of writing data byte-by-byte, this class writes data character-by-character. For example, if you would like to write text to an output file, it would be a good idea to create an instance of the `FileWriter` class, a subclass of the `Writer` class that sends its output to a specified file.

The Java library provides a very useful form of writer called a `StringWriter`. Recall in Java that strings are immutable. Suppose you were creating a string of characters, one by one, but you didn't know in advance what the

[4] In some languages characters and bytes are the same thing. However, Java distinguishes these two sorts of values.

string would be (perhaps you are reading in the characters from an input file).

For example, you could use the following code to read all of the text of a file into a new string value.

```
Reader in = new FileReader("one.txt");
int ch = in.read();
String str = "";
while (ch != -1) {
   str = str + ((char)ch);  // cast to primitive char type
   ch = in.read();
}
```

This code opens a new file reader for a specified input file. It then reads the file character-by-character until there are no characters remaining in the file. As each character is read, it updates the string value to include that new character at the end by concatenating the current string and the new character. (Note that before the int character can be added to the string it must be casted to the primitive Java type `char`. This makes sure that the value is interpreted as a character as it is appended to the string instead of a number.)

All of these appends make this routine very slow. As each character is read, a new string must be allocated in the heap and all of the previous values of the old string, plus the new string must be copied into it.

Instead, the `StringWriter` class uses a resizable array to read into a string of unknown size more efficiently. This class doubles the array each time that it needs to grow, so it does not need to do as many copies as the code above. The `StringWriter` is a `Writer`, so its `write` method appends its input to the end of the output. The `toString` method of the `StringWriter` class returns the current output as a `String` value.

```
Reader in = new FileReader("one.txt");
int ch = in.read();
StringWriter outputWriter = new StringWriter();
while (ch != -1) {
   outputWriter.write(ch);
   ch = in.read();
}
String output = outputWriter.toString();
```

29.5 Histogram demo

In the last part of this chapter, we use a design exercise to put these ideas together. This exercise also demonstrate the use of the Java Collections framework at the same time, and is good preparation for homework HW09.

Consider the following problem:

> Write a command-line program that, given a filename for a text file as input, calculates the frequencies (i.e. number of occurrences) of each distinct word of the file. The program should then print the frequency distribution to the console as a sequence of word: freq pairs (one per line).

For example, if the above text were stored in an input file, then the console should print

```
Histogram result:
The : 1
Write : 1
a : 4
as : 2
calculates : 1
command : 1
console : 1
distinct : 1
distribution : 1
e : 1
each : 1
file : 2
filename : 1
for : 1
freq : 1
frequencies : 1
frequency : 1
given : 1
i : 1
input : 1
line : 2
```

```
number : 1
occurrences : 1
of : 4
one : 1
pairs : 1
per : 1
print : 1
program : 2
sequence : 1
should : 1
text : 1
that : 1
the : 4
then : 1
to : 1
word : 2
```

Note that in this example, punctuation is ignored. The program merely breaks up the input into words and then counts the number of occurrences of each word in the program.

When thinking about this exercise, we must ask a number of questions.

- how do we input data?
- what do we need to do to process it?
- what is a word?
- how do we iterate over the input?
- how do we keep track of the frequencies?
- how do we extract the output data?

The answers to these questions help us break up the whole problem into subproblems and define the interface between those components. Furthermore, in answering these questions, we may need to look at the Java libraries to see if there is some pre-defined functionality that can help us.

One observation that we can make is that since we need to work through the file word-by-word, we need some way to return each of

the words in the file until there aren't any more. The `Iterator` interface describes a class that returns a sequence of values, because we will represent words by strings, we will create a class that implements the `Iterator<String>` interface.

WordScanner

We will call this class a `WordScanner`. We have already defined some of the interface for this class—it should be an iterator.

The other part of the interface to define is the constructor for the class. How will we create a `WordScanner`?

The constructor for this class needs a place to read the words from. Since we are working with words, we will want an input stream that works with text. Therefore, we will make the constructor take a `Reader` as an argument. The class itself will read characters from the reader and assemble them into words.

Before we write the code, we should create a few test cases. These test cases will need to supply a `Reader` to the word scanner:

For example, given a blank input, we should not be able to read any words from it. Because the word scanner will use a reader, we can test it with any subclass of reader. Even though in the full program we will read from files, for the test cases, we can use a `StringReader` to test the `WordScanner`.

```
@Test public void testEmpty() {
      WordScanner ws = new WordScanner(new StringReader(""));
      assertFalse(ws.hasNext());
   }
@Test public void testOne()  {
      WordScanner ws = new WordScanner(new StringReader("one"));
      assertTrue(ws.hasNext());
      String word = ws.next();
      assertTrue(word.equals("one"));
      assertFalse(ws.hasNext());
   }
```

After designing a few test cases, including those with multiple words, and with punctuation or other non-letter characters, we can then move onto the implementation of the word scanner. The implementation of this class is in Figure 29.2 and Figure 29.3.

The `WordScanner` class needs two pieces of private state. It needs to remember the `Reader` that it gets the input from (we'll call this `r`) and the last character read (we'll call this `c`) from this `Reader`.

The invariant is that if the character is -1 then there are no more characters in the input, and therefore no more words to return. Therefore, the `hasNext` method can just return whether this character is -1.

Furthermore, the reader will be in a state such that the last character read is always a letter. If any of the methods reads a nonletter character, the method will keep reading until it either reaches -1 or a letter. In fact, the private method `skipNonLetters` does just this. Note that the static method `Character.isLetter` can determine whether a specific character is a letter or not.

Every time the `next` method is invoked, the wordscanner needs to read all of the letter characters in the input and return that as a word. Also, to maintain the invariant about `c`, it needs to skip all of the nonletter characters following the word. This method uses a `StringWriter` to gather the read characters as above.

Histogram

Figure 29.4 uses the `WordScanner` class to solve the design problem in its `main` method. The first step is to make sure that the method was called correctly. The arguments to the main method are the *commandline arguments* to the program. In this case, we want to make sure that the program is called with a single argument, the name of the file to process. This file name should be stored in the first position of the `args` array, i.e. `args[0]`. If this argument is not present, or if there are additional arguments, the program terminates with a message describing how to run the program from the command line.

Next, the method uses a word scanner to read in the words from the file one by one. As each word is read, the method stores the word in a *finite map*. The Collections framework includes `TreeMap` an implementation of the `Map` interface. In this case we would like to map words to their number of occurrences. So the `Map` that want to create maps `String` keys to `Integer` values.

The `Reader` for the `WordScanner` is created inside of a **`try`** block. This is because this process could fail. If the given filename (`args[0]`) then

```
public class WordScanner implements Iterator<String> {
   private Reader r;
   private int c;

   // Invariant:
   //    c   is -1 for EOF, otherwise the int representation
   //        for a *letter* starting the next word.

   public WordScanner (Reader initR) {
      r = initR;
      c = 0;
      skipNonLetters();
   }

   // scans the file to make c equal to the next letter
   // character in the file. If there are none left, c is -1.
   private void skipNonLetters() {
      try {
         c = r.read();   // returns -1 at the end of the file
         while (c != -1 && !Character.isLetter(c)) {
            c = r.read();
         }
      } catch (IOException e) {
         c = -1;         // use -1 for other IOExceptions
      }
   }

   /**
    *  Returns true if there is a word available.
    */
   public boolean hasNext() {
      return (c != -1);
   }
```

Figure 29.2: Implementation of the `WordScanner`(Part I)

```
    /**
     * Returns the next word available from the Reader and
     * then skips all non-letter characters.
     *
     * Throws "NoSuchElementException" if there is not
     * another character
     *
     * @see java.util.Iterator#next()
     */
    public String next() {
       if (c == -1) throw new NoSuchElementException();
       // a "buffer" to read the word into
       StringWriter buf = new StringWriter();
       try {
          while (Character.isLetter(c)) {
             buf.write(c);
             c = r.read();
          }
       } catch (IOException e) {
          throw new NoSuchElementException();
       }
       skipNonLetters();
       return buf.toString();
    }

    /**
     * Unsupported by this iterator.
     */
    public void remove() {
       throw new UnsupportedOperationException ();

    }

}
```

Figure 29.3: Implementation of the `WordScanner` (Part II)

the constructor for the `FileReader` class will throw an exception. For efficiency, we wrap the file reader inside a `BufferedReader` before passing it to the constructor of the `WordScanner` class.

The main loop of the program iterates through all of the words in the file. If the word is already found in the `TreeMap`, the program increments its value. Otherwise, the program inserts the word into the `TreeMap` with the value 1, indicating that the word has been seen once.

After the entire file has been processed, the program closes the input file, using the `close` method of the `FileReader` class. The program then uses the iterator of the `TreeMap` class to print out the frequencies of all of the words. The for-each expression gives access to all the entries in the finite map, represented using the class `Map.Entry`. The `getKey()` method of this class gives the key (i.e. the word) and likewise `getValue()` returns the number of occurrences that were calculated. Because we used a binary search tree to implement the map, the entries are returned in alphabetical order.

```
public class Histogram {
   public static void main(String[] args) {
      if (args.length != 1) {
         System.out.println("Usage: java Histogram <file>");
         return;
      }

      // Basic idea:
      // Use a WordScanner to read in the words from the file.
      // Then use a TreeMap to calculate the word frequencies
      // Then use a PrintWriter to display the key value data
      Map<String,Integer> freq = new TreeMap<String, Integer>();
      try {
         FileReader freader = new FileReader(args[0]);
         Reader reader = new BufferedReader(freader);
         WordScanner s = new WordScanner(reader);

         while (s.hasNext()) {
            String word = s.next();
            if (freq.containsKey(word)) {
               Integer i = freq.get(word);
               freq.put(word, i+1);
            } else {
               freq.put(word,1);
            }

         }
         // release the resources used by the input streams
         reader.close();

         // Print the result
          System.out.println("Histogram result:");
          for (Map.Entry<String,Integer> kv : freq.entrySet()) {
            System.out.println(
               kv.getKey() + " : " +  kv.getValue().toString());
          }


      } catch (FileNotFoundException e) {
         System.out.println("File not found.");
      } catch (IOException e) {
         System.out.println("IO error from closing reader.");
      }
   }
}
```

Figure 29.4: Implementation of the `Histogram` class.

Chapter 30

Swing: GUI programming in Java

The third major library that we cover in CIS 120 is *Swing*, the Java library for developing GUI applications.

We study this library in CIS 120 for a number of reasons:

- It is an example of the event-based (or reactive-) programming model.

  One goal of CIS 120 is to expose students to many different forms of programming, from functional (computation produces values), to iterative (where computation does something, such as modify data structures and print output), to object-oriented (where computation is the interaction between state-hiding components) to reactive (where computation means installing actions that happen in response to events).

- It is also an example of Object-Oriented programming.

  The Swing library makes extensive use of subtyping, inheritance and overriding to define its abstractions and produce reusable code. In §18 we designed a GUI library in OCaml without any of these features. So we will be able to make a good comparison so that we understand and justify these constructs.

- As with the other libraries that we have covered, we can look at how this one is organized to learn a bit about library design.

- It enables fun applications! The Swing library gives us a rich set of classes to work with.

In this chapter we introduce the basic framework of the Swing library and discuss how it compares to the OCaml GUI library that we developed earlier in the course. For more information about Swing and developing GUI applications in Java, we recommend the Java Swing Tutorial[1].

	OCaml	Java
Graphics Context	Gctx.t	Graphics
Widget type	Widget.t	JComponent
Basic Widgets	button label checkbox	JButton JLabel JCheckBox
Container Widgets	hpair, vpair	JPanel, Layouts
Events	event	ActionEvent MouseEvent KeyEvent
Event Listener	mouse_listener mouseclick_listener (any function of type event -> unit)	ActionListener MouseListener KeyListener

Figure 30.1: Concepts from OCaml GUI library have analogues in Swing

The OCaml GUI library that we developed in §ch:gui was intended to serve as a model of Swing. There were three important parts of that model.

The fundamental building block of the OCaml GUI library was the *widget*, defined in `Widget.ml`. This module provided an abstract type of "things" displayed on the screen, giving a uniform interfaces to the building blocks of GUIs (buttons, canvas, labels, checkboxes, etc.). Other widgets provide layout of these basic elements, by placing them beside or above each other. In this way, the widgets divide up the window of the

[1]`http://docs.oracle.com/javase/tutorial/ui/index.html`.

application in a hierarchical fashion. The widget interface included three components: a function from drawing the widget on the screen (called `repaint`), a function for calculating the size of the drawn widget (called `size`), and a function for handling events that occur in the widget's area of the window.

For drawing, GUI library included the idea of a *graphics context*, implemented in `Gctx.ml`. This module provided the drawing operations that we used to build the rest of the library. It also relativized the coordinates for drawing, so that each widget could draw itself using the same code, no matter where the widget was located in the final application. Finally the graphics context tracked the state necessary for drawing, such as the current pen color or line thickness.

To handle events, the GUI library stored certain higher-order functions as *event listeners*. When an event is triggered, the event listeners would update the state of the application. The program would then communicate the changed state by redrawing all of the widgets.

The concepts that you saw in the OCaml GUI library have analogues in the Java Swing library, as listed in Figure **??**. In OCaml, the concepts were usually represented by a module, type definition or functions. In Java, they are always classes.

30.1 Drawing with Swing

We will first just discuss how to draw pictures using the Swing libraries. Using the OCaml GUI library, we could draw a picture by creating a widget where the `repaint` method uses the graphics context to draw pictures every time the widget is displayed.

For example, the following OCaml code creates a window with a line

and point.

```
let w_draw : Widget.t =
{
   repaint = (fun (gc:Graphics.t) ->
       Graphics.draw_line gc (0, 0) (100, 100);
       Graphics.draw_point gc (3,4)) ;

   size    = (fun (gc:Graphics.t) -> (200,200));

   handle  = (fun () -> ())
}

;; Eventloop.run w_draw
```

In our Java example, we will draw in roughly the same way. The Swing analog to widget is the class `JComponent`. All Swing GUI components are subclasses of this class. To create a component that draws a picture in the window, like the one above, we need to create a subclass of this class that *override*s the methods that correspond to `repaint` and `size` above.[2]

The two analogous methods in the `JComponent` class are:

```
public void        paintComponent(Graphics g);
      // corresponds to repaint above
public Dimension        getPreferredSize();
      // corresponds to size above
```

(A `Dimension` is an object that contains an height and width fields.)

For example, to produce a component that displays as in Figure 30.2 we would first create the class shown in Figure 30.3. During the execution of the `paintComponent` method, this class uses the graphics context `Graphics` to set the pen color to green and draw lines in the window. The coordinate system for drawing is relative to the upper-left corner of the `Component`. The tree starts drawing at pixel position (75,100) and each recursive call to the `fractal` method adds to the image.

Note that the first line of the `paintComponent` method is an invocation of `super.paintComponent`. This method definition overrides the definition of `paintComponent` in the superclass. However, that method definition already does things like paint the background of the component. Therefore, the first line of the new method is a call to the old method. After the background is painted, then the tree is painted on top.

[2]Recall that method overriding is when a subclass replaces a method definition with a definition of its own.

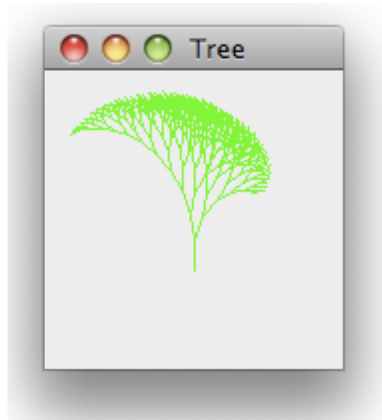


Figure 30.2: A fractal tree

In OCaml, to display a widget, we just needed to start the eventloop with the widget at the toplevel. In Java, to get things started we need to create a `JFrame`, a container that stores the toplevel `JComponent`. This frame corresponds to the window that your operating system uses to display GUI applications. The same Java code will produce different windows on different platforms because it uses the window drawing routines supported by that platform.

The `run` method below demonstrates this process. This method creates and displays the window containing the drawing defined above.

```
class DrawingApplication implements Runnable {
   public void run() {
     // a frame is a top-level window
     JFrame frame = new JFrame("Tree");
     // set the content of the window to be the drawing
     frame.add(new Drawing());
     // make sure the application exits when the frame closes
     frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
     // resize the frame based on the size of the canvas
     frame.pack();
     // show the frame
     frame.setVisible(true);
   }
}
```

The argument to the `JFrame` constructor is the name on the "title" of the window (usually shown at the top of the window by most operating

```java
public class Drawing extends JComponent {

    /* This paint function draws a fractal tree
     * Google "L-systems" for more explanation / inspiration.
     */
    private static void fractal(Graphics gc, int x, int y,
                                double angle, double len) {
        if (len > 1) {
            double af = (angle * Math.PI) / 180.0;
            int nx = x + (int)(len * Math.cos(af));
            int ny = y + (int)(len * Math.sin(af));
            gc.drawLine(x, y, nx, ny);
            fractal(gc, nx, ny, angle + 20, len - 2);
            fractal(gc, nx, ny, angle - 10, len - 1);
        }
    }

    public void paintComponent(Graphics gc) {
        super.paintComponent(gc);

        // set the pen color to green
        gc.setColor(Color.GREEN);

        // draw a fractal tree
        fractal (gc, 75, 100, 270, 15);
    }

    // get the size of the drawing panel
    public Dimension getPreferredSize() {
        return new Dimension(150,150);
    }

}
```

Figure 30.3: Source code for Figure 30.2.

systems.) The window shown in Figure 30.2 has the word "Tree" at the top. This line is where that is specified. The next line of the program instructs the frame that it should display a new instance of the `Drawing` class. Frames are containers, in that they hold a single `JComponent`, and this line determines the component that will be displayed in the frame. The next line instructs the frame what to do when the user clicks the close button on the window (on Mac OS, the red circle in the titlebar). The program doesn't end automatically when this happens unless we include this line in the code. After that, the frame must calculate how big it should be. The `pack` method performs this calculation automatically, using the `getPreferredSize` method of the component, and making sure that the frame is exactly big enough to contain the drawing panel.

Finally, although we have created the frame, it doesn't display until we tell it to show itself. This `setVisible` actually makes the frame appear on the screen. This frame can be interacted with using the standard windowing commands (move, resize, minimize, maximize, close) provided by the operating system. But it doesn't really do anything else.

We start a Swing application using the following main method. The static method `SwingUtilities.invokeLater` makes sure that the GUI starts correctly. Its argument must be something that implements the `Runnable` interface (which only requires the class to have a `run` method).

```
public class Main {
    public static void main(String[] args) {
        SwingUtilities.invokeLater(new DrawingApplication());
    }
}
```

30.2 User Interaction

Next, we show how to use reactive programming to develop applications that react to standard user input, such as mouse events and key presses. Our eventual goal will be to re-create something like the Paint program using Swing instead of our hand-rolled OCaml GUI.

However, before we do that, we will start with something much more modest. We will go back to the lightswitch demo that we used for OCaml (back in Figure 18.9) and rewrite it using the Swing libraries. Our first goal is the simple design problem:

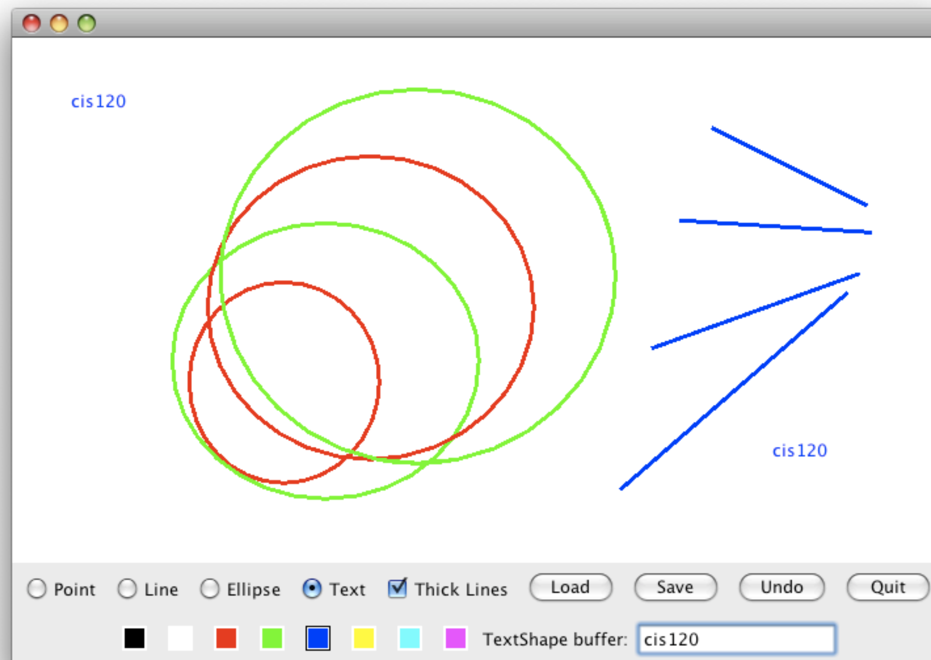


Figure 30.4: Java Paint

> Program an application that displays a button. When the button is pressed, it toggles a lightbulb on and off.

To implement the lightswitch, we will first assemble the components and then add the interactivity. In this case, our application is composed of two different components, a button (labeled On/Off) and a section of the window that changes color from yellow to black. For the latter, we will create a component similar to the `Drawing` above. The difference is that this time the object will include a boolean field, called `isOn`, that will determine whether to repaint itself yellow or black. If the value of `isOn` is true, then the object will repaint itself with a yellow rectangle. Otherwise, it will use black. (Note that this time we omit the call to **`super`**`.paintComponent` because our painting completely fills the component. We don't need the background.)

The method `flip` changes the state of the lightbulb. If it is on, then it turns off and vice versa. However, there is one more important part

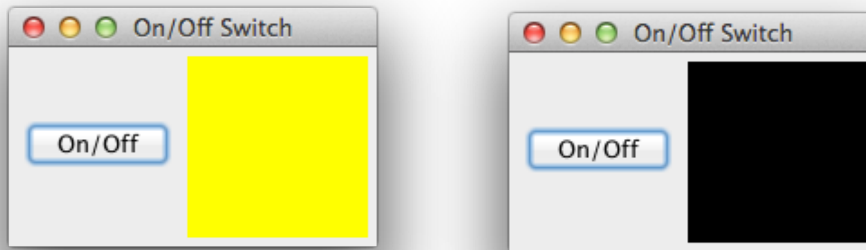


Figure 30.5: Java Lightbulb, both on and off states

to `flip`. The last line of the method, the call `repaint`, tells the swing event loop that this component should be redisplayed. Unlike the OCaml GUI library, which repaints every widget at every time step, the Swing library only repaints the components that indicated that they need to be repainted.

We can get this started as we did for the fractal tree drawing above. Again we create a frame and specify its title. However, this time we would like to add *two* components to the frame, a button and an instance of the `Lightbulb` class above. A frame can only contain one component, if we tried to add two of them, then only the most recently added component would be displayed.

Therefore, we need a *Container* component that can hold multiple components. The class `JPanel` is such a component. We create after creating an instance of this class, we can add both a button (created from the class `JButton`) and a lightbulb to the panel. (By default, these two components will appear side-by-side, as in a `hpair`.)

```
public class Lightbulb extends JComponent {
    private boolean isOn;

    public void paintComponent(Graphics gc) {
        if (isOn) {
            gc.setColor(Color.YELLOW);
            gc.fillRect(0, 0, 100, 100);
        } else {
            gc.setColor(Color.BLACK);
            gc.fillRect(0, 0, 100, 100);
        }
    }

    public Dimension getPreferredSize() {
        return new Dimension(100,100);
    }

    public void flip() {
        isOn = !isOn;
        // In Swing you have to explicitly ask to repaint
        // a component.  You should call repaint whenever
        // the display of the component changes.
        this.repaint();
    }
}
```

Figure 30.6: The Lightbulb class

```java
public class OnOff implements Runnable {
    public void run() {
        JFrame frame = new JFrame("On/Off Switch");

        JPanel panel = new JPanel();
        frame.add(panel);

        JButton button = new JButton("On/Off");
        panel.add(button);

        Lightbulb bulb = new Lightbulb();
        panel.add(bulb);

        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.pack();
        frame.setVisible(true);
    }
}
```

However, although the code above will display the button and the lightbulb, it doesn't *do* anything. Pushing the button has no effect. That is because we haven't added an event handler to the button to *react* to button presses.

30.3 Action Listeners

In the OCaml GUI library, we used *notifiers* to handle events. These widgets maintained a list of higher-order functions that would execute in the case of an event.

The Swing library works in a similar manner. However, because Java lacks higher-order functions, the library uses objects instead. Any class that implements the `ActionListener` interface (see below) can be used to handle events that are triggered as the result of user actions. This interface is defined in the package `java.awt.event`.

```java
interface ActionListener {
   public void       actionPerformed(ActionEvent e);
}
```

In our lightswitch example, we would like to implement an `ActionListener` that changes the state of the lightbulb by calling its `flip` method. To do

this, we create a class that stores a reference to the lightbulb (using the private field `bulb`) and then, in the `actionPerformed` method, calls the flip method of the bulb object. The only other part of this class is a constructor that initializes the field.

```
class BulbSwitch implements ActionListener {
    private Lightbulb bulb;

    public BulbSwitch(Lightbulb b0) {
        this.bulb = b0;
    }

    public void actionPerformed(ActionEvent e) {
        bulb.flip();
    }

}
```

The final step is to "wire up" the button. In other words, we need to create an instance of the `BulbSwitch` class and tell the button to store this object as an action listener. We do this by adding the following line to the `main` method, after the `button` and `bulb` components have been created. The `addActionlListener` method of the `JButton` class adds its argument to the list of action listeners to the button. Whenever the button is pressed, the `actionPerformed` methods of all of the action listeners will be called.

```
 // "wire up" the bulb to the button
  button.addActionListener(new BulbSwitch(bulb));
```

30.4 Timer

Suppose we would like to implement a blinking light bulb, one that turns itself on and off without waiting for user input. To control this light bulb, we use a `Timer` object, as in the code below.

```
    // Add a blinking light too
    Lightbulb blinker = new Lightbulb();
    panel.add(blinker);
    // Create a timer that will call actionPerformed() on the
    // given TimerAction object every second (1000ms)
         Timer timer = new Timer(1000, new BulbSwitch(blinker));
         timer.start();
```

The `Timer` constructor takes two arguments, an time interval (in milliseconds), and an `ActionListener` to run whenever the timer "goes off". The method `start` gets the timer started.

Note that both the `Timer` and the `JButton` can use the same `ActionListener`. We didn't have to change anything in the `BulbSwitch` class. This is a nice property of the Swing library design, it separates the description of *what* should happen (toggling a lightbulb) from the description of *why* it should happen (because the user pressed a button, or because a timer went off.)

In fact, even though in the example above, the button and timer control different bulbs, there is no reason that they can't control the same bulb—-which bulb changes is specified by the argument of the `BulbSwitch` constructor. Likewise, a button or timer could control multiple bulbs, merely by adding additional action listeners.

Chapter 31

Swing: Layout and Drawing

31.1 Layout

A GUI application typically is composed of several components: buttons, canvases, checkboxes, textboxes etc. One challenge for the programmer is specifying how these components should be arranged on the screen in relation to each other.

In the OCaml GUI library, we used `vpair` and `hpair` to simply put widgets side-by-side, or on top of each other. By nesting these operations, we can lay out widgets in many different configurations. However, this work is tedious.

In Swing, we can use layout managers to more succinctly develop standard component arrangements. In this chapter, we demonstrate how to use a few of the layout managers. However, we don't intend to be comprehensive. For more information about various forms of layout see the Java Swing Tutorial "Lesson: Laying Out Components Within a Container" available at: `http://docs.oracle.com/javase/tutorial/uiswing/layout/index.html`.

In this section, we will demonstrate the effect of three different layout managers, used to assemble a graphical application. Our application in these examples just consists of five different buttons. These buttons won't do anything, we will just arrange them in different configurations.

The listing below includes the entire code of the application. The `run` method below creates a frame and five buttons. It also creates a container component, a `JPanel` to store the five buttons. The frame contains the

panel, and the panel contains the buttons.

```
public class LayoutExample implements Runnable {

    public void run() {
        JFrame frame = new JFrame ("LayoutExample");
        JButton b1 = new JButton("one");
        JButton b2 = new JButton("two");
        JButton b3 = new JButton("three");
        JButton b4 = new JButton("four");
        JButton b5 = new JButton("five");

        JPanel panel = new JPanel();
        frame.add(panel);

        panel.add(b1);
        panel.add(b2);
        panel.add(b3);
        panel.add(b4);
        panel.add(b5);

        frame.pack();
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setVisible(true);
    }

    public static void main(String[] args) {
        SwingUtilities.invokeLater(new LayoutExample());
    }

}
```

The default layout for a `JPanel` is called `FlowLayout`. This layout means that all components added to the panel will be positioned in order left-to-right. If there is not enough width to fit all of the components, then the layout starts a new line of components. Within each line, all of the included components are centered on the line.

Because `FlowLayout` is the default layout, we do not have to do anything special to use this configuration. Furthermore, the `JPanel` does not fix its width, so the layout puts all of the buttons on a single line, and resizes the panel to fit around all of the buttons. The effect of this layout is shown on the left side of Figure 31.1. However, as the window (and panel) changes width, the buttons rearrange themselves. The "flow" so that they

Figure 31.1: FlowLayout before and after window resize

still fit into the narrower window, as in the right side of the figure.

The next layout we will demonstrate is the `GridLayout`. This layout puts the buttons into a grid configuration. Changing the application to use this layout merely requires adding the line

```
panel.setLayout(new GridLayout(3,2));
```

to `run` method right before all of the buttons are added to the panel. This command instructs the panel to divide itself into a grid containing three rows and two columns. The resulting window of buttons is shown in Figure 31.2.

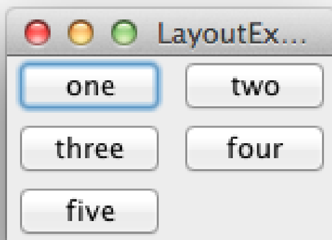


Figure 31.2: GridLayout

No matter how this window is resized, it maintains this grid configuration. Furthermore, as the window is enlarged or shrunk, the buttons

resize themselves to always take one sixth of the size of the window.

The last layout that we will demonstrate is the border layout. This layout has exactly five positions, as demonstrated in Figure 31.3. The application can use this layout by first instructing the panel to use a `BorderLayout`, and then changing how the buttons are added to the panel. Each button must specify which position it should occupy. (Not all positions must be occupied in the border layout.)

```
panel.setLayout(new BorderLayout());
panel.add(b1,BorderLayout.LINE_START);
panel.add(b2,BorderLayout.CENTER);
panel.add(b3,BorderLayout.LINE_END);
panel.add(b4,BorderLayout.PAGE_START);
panel.add(b5,BorderLayout.PAGE_END);
```

As the window is resized, the button in the center position is resized to accommodate the new space. The components on the top and bottom retain their heights, but adjust their widths. The components on the left and right retain their widths, but adjust their heights.

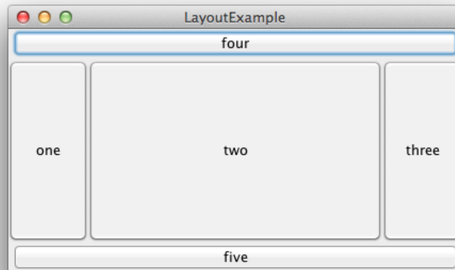


Figure 31.3: BorderLayout

31.2 An extended example

In the next section, we will go through a few variations of a simple graphical application. These variations will provide a bigger example of GUI layout, but also discuss design patterns for object-oriented programming.

To begin, let's consider the following application. The window looks like this:

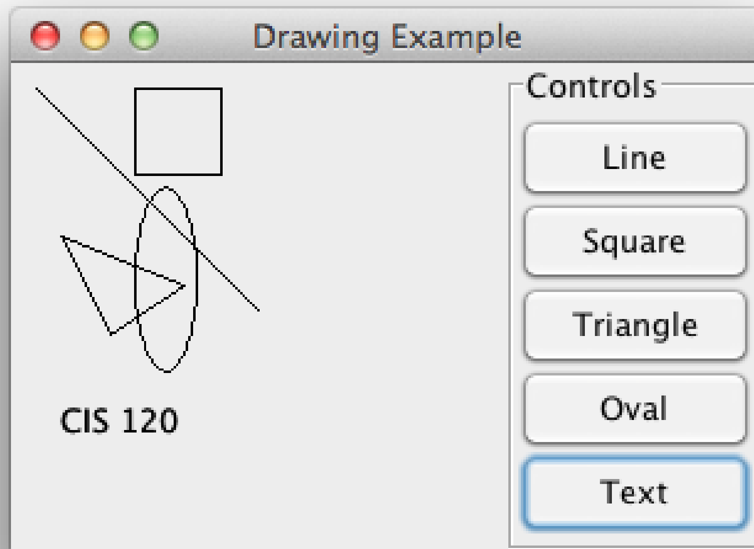


Figure 31.4: A Simple GUI Application

The idea behind this application is that it a *very* simplified paint program. The part of the window on the left is a canvas, and the right are the controls. Each button adds a new drawing at a fixed location to the canvas. Figure 31.4 shows the result after all of the buttons have been pressed.

Initial design

The first question is, what components should we use to build the interface? The drawing area to the left should be a component where we override the `paintComponent` method so that it displays the shapes. We'll design the class `DrawingExampleCanvas` to do that. For the buttons on the right, we can use the standard `JButton` class. As above, we will store references to these components in the fields of a class that uses a `run` method

to set up the interface.

```
public class DrawingExample implements Runnable {
        // Buttons
        public JButton b1, b2, b3, b4, b5;
   // Canvas
        public DrawingExampleCanvas drawingCanvas;

   public void run() { .... }
}
```

Next we should think about how we layout such an application. As we change the size of the window, we would like the canvas to grow in size. This implies that we should use a `BorderLayout` for the top-level frame and insert the canvas in the center position. The buttons, on the right of the window, will need to be stored in their own *container* component, a `JPanel`, so that they can be inserted into the frame at the position `JFrame.LINE_END`. We would like to lay out the buttons vertically within the panel. Therefore we will use a `GridLayout` with 5 rows and a single column.

```
public void run() {
    JFrame frame = new JFrame("Drawing Example");

    // Create the buttons
    b1 = new JButton("Line");
    b2 = new JButton("Square");
    b3 = new JButton("Triangle");
    b4 = new JButton("Oval");
    b5 = new JButton("Text");

    // canvas for displaying the shapes
    drawingCanvas = new DrawingExampleCanvas(this);

    // panel to contain the buttons
    JPanel eastPanel = new JPanel();
    eastPanel.setBorder(
       BorderFactory.createTitledBorder("Controls")
    );
    // layout the panel with a grid of 5 rows and 1 column
    // each grid cell will be the same size and the buttons
    // will expand to fill each cell
    eastPanel.setLayout(new GridLayout(5, 1));
    eastPanel.add(b1);
    eastPanel.add(b2);
    eastPanel.add(b3);
    eastPanel.add(b4);
    eastPanel.add(b5);

    frame.setLayout(new BorderLayout());
    frame.add(drawingPanel, BorderLayout.CENTER);
    frame.add(eastPanel, BorderLayout.LINE_END);

    // Put it on the screen
    frame.pack();
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    frame.setVisible(true);
}
```

Next, we should consider the interactions between the components. What sort of state should be shared between them? How will the `DrawingExampleCanvas` know what shapes to display? How will the buttons toggle that state?

For our first cut, we will add some boolean fields to the `DrawingExample` class to remember which buttons have been pushed. The idea is that can-

vas will look at these fields to determine which shapes to draw, and the buttons will set the appropriate one to true when pressed.

```
public class DrawingExample implements Runnable {
    ...
    // these are public so that they can be accessed by the
    // canvas.
    public boolean drawLine     = false;
    public boolean drawSquare   = false;
    public boolean drawTriangle = false;
    public boolean drawOval     = false;
    public boolean drawText     = false;
```

Note that we make these boolean fields public because they must be read and modified by other classes. The class for the `DrawingExampleCanvas` must read them to find out which shapes to display. To do this, it must also keep a reference (called `owner`) to the object that contains these boolean fields. (See the line above which creates the canvas—the constructor is invoked with the **`this`** reference.)

```
class DrawingExampleCanvas extends JComponent {
    // must include a reference to the "top-level"
    // application to be able to access its state.
    private DrawingExample owner;

    public DrawingExampleCanvas (DrawingExample p) {
        owner = p;
    }

    // We override this method to perform our own custom drawing.
    public void paintComponent(Graphics gc) {
        super.paintComponent(gc);

        if (owner.drawLine) {
            // code for drawing lines
        }

        if (owner.drawSquare) {
            // code for drawing squares
        }

        if (owner.drawTriangle) {
            // code for drawing triangles
        }

        if (owner.drawOval) {
            // code for drawing ovals
        }

        if (owner.drawText) {
            // code for drawing text
        }
    }

    public Dimension getPreferredSize() {
        return new Dimension(200,200);
    }
}
```

Finally, we need to add the action listeners to the buttons so that they modify the correct boolean value. We use the same listener for each button, and pass a reference to the button that the listener is associated with. When the button is pressed, we compare this reference to each of the buttons in the application to determine which shape to draw.

```
class DrawingExampleListener implements ActionListener {
    private DrawingExample owner;
    private JButton button;

    // Store a reference to our "owner object"
    public DrawingExampleListener(DrawingExample p,
                                  JButton b) {
        owner = p; button = b;
    }

    public void actionPerformed(ActionEvent e) {
        // Find out which button generated the event, and
        // use this information to set the appropriate
        // flag in the owner object
        if (button.equals(owner.b1)) {
            owner.drawLine = true;
        } else if (button.equals(owner.b2)) {
            owner.drawSquare = true;
        } else if (button.equals(owner.b3)) {
            owner.drawTriangle = true;
        } else if (button.equals(owner.b4)) {
            owner.drawOval = true;
        } else if (button.equals(owner.b5)) {
            owner.drawText = true;
        }

        // Notify Swing that the drawing panel needs
        // to be repainted
        owner.drawingCanvas.repaint();
    }
}
```

Again, because the execution of the `actionPerformed` method accesses the fields of the top-level object, the constructors for the `DrawingExampleListener` class must take in a reference to it.

In the `run` method, after the buttons are created, we add their action listeners with the following code.

```
  // Attach actions to the buttons.
  b1.addActionListener(new DrawingExample0Listener(this, b1));
  b2.addActionListener(new DrawingExample0Listener(this, b2));
  b3.addActionListener(new DrawingExample0Listener(this, b3));
  b4.addActionListener(new DrawingExample0Listener(this, b4));
  b5.addActionListener(new DrawingExample0Listener(this, b5));
```

Variant A: Using Dynamic Dispatch

The code above gets the job done and works correctly, but it is clumsy. In the next few subsections, we discuss how to refactor the code to improve the object-oriented style of its implementation. One problem with this implementation is the lack of encapsulation: we made the fields public in the top-level class so that they could be accessed by the helper class. (We could improve things with "package" scope, but as we will see, there are still better strategies.) Another problem concerns extensibility: if we wanted to add a new shape-drawing button, then what must we modify? We must not only add a new button, a new boolean field for that shape, but we must also modify the if statements in the methods `paintComponent` and `actionPerformed` to test this new boolean value.

In general, sequences of boolean tests, of the form

```
if (test1) {
  // do something
}
if (test2) {
  // do something
}
```

are not extensible. Instead, it is much better to refactor the application to avoid these if statements. In this section, we will demonstrate how to use *dynamic dispatch* to eliminate this sequence in `paintComponent`. In the next section, we will refactor the code again to remove the if statements in `actionPerformed`.

The code in the `paintComponent` class must display all of the shapes that have been selected by the buttons. This code dispatches on the boolean values to do the shape drawing, instead we would like to refactor the code so that instead of using a fixed set of boolean fields to remember which shapes to draw, we will use a mutable collection of shapes. In other words, we will replace the boolean fields with the single field:

```
public List<Shape> shapes = new LinkedList<Shape>();
```

That way we can refactor the `paintComponent` method to look like this:

```
public void paintComponent(Graphics gc) {
    super.paintComponent(gc);
    for (Shape shape : owner.shapes) {
        shape.draw(gc);
    }
}
```

Here, the toplevel applications uses the field `shapes` to store a collection of object that all implement the `Shape` interface. Because `owner.shapes` is a collection, we can use the "for-each" notation to iterate through each shape in the collection. Each shape knows how to draw itself, so that is what the `paintComponent` method does for each shape in the collection.

What is a shape? It is an object that knows how to draw itself. Therefore, we can create an interface that declares exactly this:

```
interface Shape {
  public void draw(Graphics gc);
}
```

and define various classes that implement that interface:

```
class Line implements Shape {
    public void draw(Graphics gc ) {
        gc.drawLine(10, 10, 100, 100);
    }
}
class Oval implements Shape {
    public void draw(Graphics gc ) {
        gc.drawOval(50, 50, 25, 75);
    }
}
```

Variant B: Refactoring references

Now there is one more change to finish out the first variation—we need to modify the `actionPerformed` method of the event listener to add shapes to the collection instead of modifying boolean values.

```
class DrawingExampleListener implements ActionListener {
    private DrawingExample owner;
    private JButton button;

    // Store a reference to our "owner object" and the button
    public DrawingExampleListener(DrawingExample p, JButton b) {
        owner = p; button = b;
    }

    public void actionPerformed(ActionEvent e) {
        // Find out which button generated the event, and
        // use this information to set the appropriate flag in
        // the owner object
        if (button.equals(owner.b1)) {
            owner.shapes.add (new Line());

        } else if (button.equals(owner.b2)) {
            owner.shapes.add(new Square());

        } else if (button.equals(owner.b3)) {
            owner.shapes.add(new Triangle());

        } else if (button.equals(owner.b4)) {
            owner.shapes.add(new Oval());

        } else if (button.equals(owner.b5)) {
            owner.shapes.add(new Text());

        }

        // Notify Swing that the drawing panel needs
        // to be repainted
        owner.drawingCanvas.repaint();
    }
}
```

However, this code is still a series of if expressions! Like the original version of the `paintComponent` it is not automatically extensible. If we want to add more shapes, we have to modify this code.

However, our previous modification suggests a simple fix. Instead of remembering the button that was pressed to figure out which shape to add, ActionListener should just skip a step and remember the actual shape to add. In other words, we'll change the class definition to include a shape reference, which should be initialized by the constructor.

As an added bonus, we can avoid adding the same shape twice. We can test whether the collection already contains the shape before adding it.

```
class DrawingExampleListener implements ActionListener {
    private DrawingExample owner;
    private Shape shape;

    // Store a reference to our "owner object"
    public DrawingExampleListener(DrawingExample p, Shape s) {
        owner = p; shape = s;
    }

    public void actionPerformed(ActionEvent e) {
        if (!owner.shapes.contains(shape)) {
            owner.shapes.add(shape);
        }

        // Notify Swing that the drawing panel needs
        // to be repainted
        owner.drawingCanvas.repaint();
    }
}
```

Now, when we add the action listeners to the buttons, we need to merely create the appropriate shape for each button. Whenever the button is pressed, that shape will be added to the collection.

```
// Attach actions to the buttons.
b1.addActionListener(
    new DrawingExampleListener(this, new Line()));
b2.addActionListener(
    new DrawingExampleListener(this, new Square()));
b3.addActionListener(
    new DrawingExampleListener(this, new Triangle()));
b4.addActionListener(
    new DrawingExampleListener(this, new Oval()));
b5.addActionListener(
    new DrawingExampleListener(this, new Text()));
```

Variant C: Using a helper method

Now we can see that there is some redundancy in button creation. First the method calls the `JButton` constructor five times to create the buttons.

Then it calls the `addActionListener` five times to attach the action listeners to the buttons. The next refactoring we will do will be do create a helper method to factor out this repeated code.

The helper method should create a button and add its action listener. To do this, the method needs to know two things: the name of the button, and the shape that should be drawn when the button is pressed. Therefore we add the following method to the `DrawingExample` class.

```
// Create a new button, given the name of the shape and
// a shape object for drawing
private JButton makeButton(String name, Shape shape) {
     JButton b = new JButton(name);
     b.addActionListener(new DrawingExampleListener(this,shape));
     return b;
}
```

Now we can go back to the `run` method and replace the repeated code with calls to this method. Now the button creation code read succinctly:

```
// Create the buttons
JButton b1, b2, b3, b4, b5;
b1 = makeButton("Line", new Line());
b2 = makeButton("Square", new Square());
b3 = makeButton("Triangle", new Triangle());
b4 = makeButton("Oval", new Oval());
b5 = makeButton("Text", new Text());
```

This code is better because it is easier to read. Each button declares exactly what is important about it, compared to the other buttons. We know that all the buttons roughly do the same thing, because they are all created by the `makeButton` method. But we know that they differ in their name, and in the shape that they create.

Variant D: Inner Classes

These previous refactorings solved the problem of extensibility and code redundancy, but the encapsulation problem still remains. The canvas and the action listeners still need to access the fields of the `DrawingExample` object. Therefore, these fields are public.

Furthermore, these two classes are very related to the `DrawingExample` class. They don't include a lot of code (especially after the refactoring) and only make sense in conjunction. In this section, we will see how the Swing

feature of *Inner classes* can tie these classes together.

Basically, *Inner classes*[1] allow classes to contain each other. They are useful in situations where two objects require "deep access" to each other's internals. The basic idea is that we can define one class inside of another. The inner class can then refer to the fields of the outer class. As a result, instances of the inner class must be created using instances of the outer class.

For example, a cut down version of the drawing example stores the list of shapes in instances of the class `DrawingExample`, and displays them in instances of the class `DrawingCanvas`.

```
public class DrawingExample implements Runnable {
    public List<Shape> shapes = new LinkedList<Shape>();
    private DrawingPanel drawingPanel;
    public void run() {
        JFrame frame = new JFrame("Drawing Example");
        drawingPanel = new DrawingPanel(this);
        ...
    }
}
class DrawingCanvas extends JComponent {
    private DrawingExample owner;
    public DrawingCanvas (DrawingExample p) { owner = p; }
    public void paintComponent(Graphics gc) {
        super.paintComponent(gc);
        for (Shape s: owner.shapes) {
            s.draw(gc);
        }
    }
    ...
}
```

Alternatively, we can make `DrawingCanvas` an *Inner class* of `DrawingExample`. Note below that the canvas class is defined inside the `DrawingExample` class.

[1] Also called "dynamic nested classes."

```
public class DrawingExample implements Runnable {
    public List<Shape> shapes = new LinkedList<Shape>();
    private DrawingPanel drawingPanel;
    public void run() {
        JFrame frame = new JFrame("Drawing Example");
        drawingPanel = new DrawingPanel();
        ...
    }

    class DrawingCanvas extends JComponent {
        public DrawingCanvas () {  }
        public void paintComponent(Graphics gc) {
            super.paintComponent(gc);
            for (Shape shape : shapes) {
                shape.draw(gc);
            }
            ...
        }
    }
```

Likewise, we can move the `DrawingExampleListener` class to be inside the `DrawingExample` class and eliminate its `owner` field.

Inner classes are a replacement for tangled workarounds like "owner" references (as in the drawing example). They help clarify the code: the solution with inner classes is easier to read. Furthermore, they better support encapsulation as there is no need to allow public access to instance variables of outer class.

For example, consider this definition, with outer class `Outer` and inner class `Inner`. The inner class constructor can refer to the field `outerVar`, even though it is a private field. That is because `Inner` itself is a part of `Outer`. The full name of the inner class is actually `Outer.Inner`, which is the type of objects that this class creates.

```
public class Outer {
  private int outerVar;
  public Outer () {
    outerVar = 6;
  }
  public class Inner {
    private int innerVar;
    public Inner(int z) {
      innerVar = outerVar + z;
    }
  }
}
```

Inner classes must be creating using instances of the outer classes. Even though the class name is `Outer.Inner`, we cannot create an instance of this class with

```
Outer.Inner inner = new Outer.Inner(3);
```

Usually, instances of the inner class will be created in the nonstatic methods of the outer class. This situation looks like normal object creation:

```
public Outer () {
        Inner b = new Inner ();
}
```

but it is not quite the same. Just like a field access `x` is different than a local variable and really short for `this.x`, the above code is short for using the current outer object to create a new inner one.

```
public Outer () {
        Inner b = this.new Inner ();
}
```

Outside the outer class, we need to specify what object to use to create instances of the inner class. For example, in some other file, we could first create an object `a`, and then use it to create an inner object `b`.

```
Outer a = new Outer();
Outer.Inner b = a.new Inner();
```

Variant E: Anonymous Inner Classes

For the last refactoring of the `DrawingExample` class, we will observe a few things about the inner classes that we defined above (`DrawingExampleListener` and `DrawingExampleCanvas`). Both of these classes are fairly short (now that we've made them inner classes), and both of them are used *only once* to create new objects. Given these two properties, it is annoying that we have to name these classes and define them far away from their single use. In deed, in OCaml we might just use an anonymous function as an event listener, and forgo a separate definition.

Anonymous Inner Classes are the closest Java feature to OCaml's anonymous higher-order functions. This feature will let us refactor the `DrawingExample` so that the code that gets run by the button can be defined at the same place where the button's action listener is installed.

We'll do this by revising the button creation helper method (which is the code that installs the action listeners for the button). Instead of `new DrawingExampleListener(shape)`, we will inline the actual code for the `DrawingExampleListener` class right there. This code only defines the `actionPerformed` method, so we will defined that method in place.

```
// Create a new button, given the name of the shape and
// a shape object for drawing
private JButton createButton(String name, final Shape shape) {
    JButton b = new JButton(name);
    b.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                if (!shapes.contains(shape)) {
                    shapes.add(shape);
                }
                drawingCanvas.repaint();
            }
        });
    return b;
}
```

Note that the class `DrawingExampleListener` goes away. With the definition above we can remove this class definition. Instead, we have an anonymous class—all we know about it is that it is an instance of the interface `ActionListener`. This is the one place in Java where we can put an interface name after the `new` keyword. The reason that we can do this is that we immediately satisfy that interface with the definition of the

actionPerformed method.

Note that this method definition is allowed to refer to the variable shape, which is a parameter to the createButton method. Java allows this reference only for **final** variables, those with values that cannot be mutated.

Anonymous Inner classes are a new expression form that defines a class and instantiates it (creating an object) all at once. The general format starts with the **new** keyword. However, after that comes either an interface or a class name, followed by the definition of perhaps multiple members for that class (i.e. fields and methods.)

```
new InterfaceOrClassName() {
    public void method1(int x) {
        // code for method1
    }
    public void method2(char y) {
        // code for method2
    }
}
```

Note that there is a limitation here: the anonymous class cannot define a constructor. Such classes rarely need them however, the scoping rules mean that the objects fields can be initialized using values in the current context. The static type of this expression is the interface or class that was used to create the object. This static type is a supertype of the object's dynamic class, however the object's dynamic class is *anonymous*, we didn't give it a name. Therefore, Java will not allow us to refer to it.

Why do we say that anonymous Java's inner classes are similar to OCaml's higher-order functions? Both of these features create anonymous structures. In Java we don't define the actual name of the class that constructs the object above, and in OCaml, we don't define a name for the function when we say `fun (x:int) -> x + 1`.

Furthermore, both of these language features create "delayed computation" that can be stored in a data structure and run later. For example, in Java these are commonly used in GUI programming for event listeners, while in OCaml, we used first-class functions for the same purpose. The computation is delayed because it only runs when the button is pressed, and could run once, many times, or not at all.

Finally, the scoping of these constructs is similar. Both sorts of computation can refer to variables in the current scope. In OCaml, all variables

are immutable, so OCaml first class functions could refer to any available variable. In Java, these methods can refer to only instance variables (fields, which are referenced through the immutable variable this) and variables marked final (which are also immutable).

Chapter 32

Swing: Interaction and Paint Demo

In this chapter we will use a larger example to demonstrate user interaction with Swing. In particular, we will go back to the paint program that we developed with the OCaml GUI library and reimplement it with Java's swing library. This application requires installing mouse listeners to interact with the drawing on the screen. It will also give us a chance to make some comparisons between the OCaml and Java languages, with respect to Object-Oriented programming.

32.1 Version A: Basic structure

The basic structure of the application is similar to the drawing example of the previous chapter. The top-level application contains the 'state' of the application, in this case the color for drawing (will only be black in this chapter) and the list of shapes that should be drawn. Shapes are drawn on the `canvas`, an area of the screen. Below the canvas is the *modeToolbar*, the section of the screen used to select the shape to be drawn. The run method of this application creates the GUI elements and lays them out in the window.

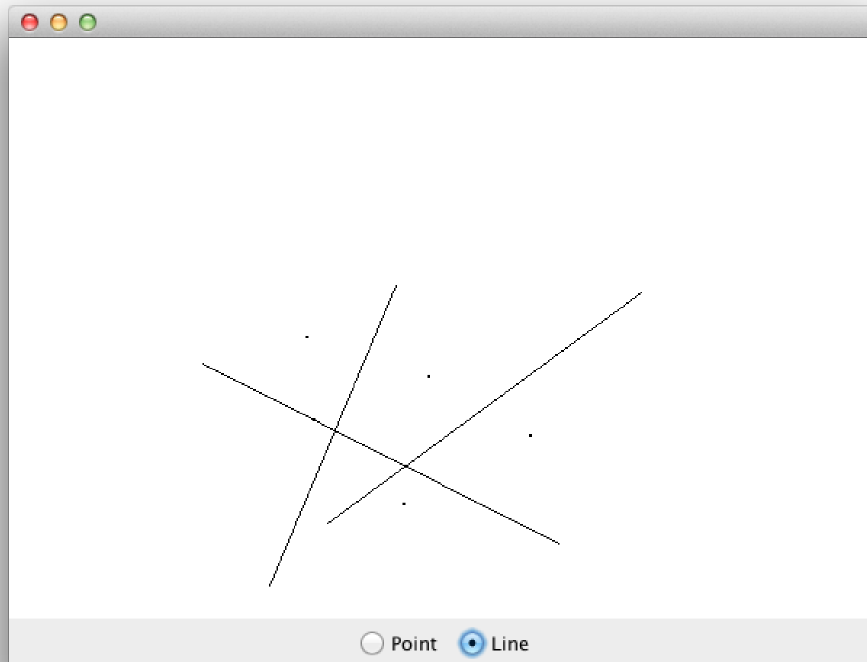


Figure 32.1: A Simple Paint Application

```
public class Paint implements Runnable {

    /** Area of the screen used for drawing */
    private Canvas canvas;

    /** Current drawing color */
    private Color  color  = Color.BLACK;

    /** The list of shapes that will be drawn on the canvas. */
    private List<Shape> shapes = new LinkedList<Shape>();
```

```
    public void run () {

        JFrame frame = new JFrame();
        canvas = this.new Canvas();

        frame.setLayout(new BorderLayout());
        frame.add(canvas,              BorderLayout.CENTER);
        frame.add(createModeToolbar(), BorderLayout.PAGE_END);

        frame.pack();
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setVisible(true);
    }

    public static void main(String[] args){
        SwingUtilities.invokeLater(new Paint());
    }

}
```

The code for the canvas, is almost identical to that of the previous chapter. As before, it is an inner class so that it can access the private fields of the top-level application. It also inherits from the `JPanel` class so that it can set the background color to be white in the constructor.

```
private class Canvas extends JComponent {

    public void paintComponent(Graphics gc) {
        super.paintComponent(gc);
        for (Shape s : shapes) {
            s.draw(gc);
        }
    }
    public Dimension getPreferredSize() {
        return new Dimension(600,400);
    }

    public Canvas(){
        super();
        setBackground(Color.WHITE);
    }

}
```

The mode toolbar is created by an auxiliary method in the `Paint` class.

This method constructs a panel to store the controls. So far, the controls include two radio buttons (one for drawing shapes, one for drawing lines). Radio buttons are UI elements, where only one can be selected at a time. We could extend the application to include more shapes by adding more radio buttons to this toolbar.

```
private JPanel createModeToolbar() {

    JPanel modeToolbar = new JPanel();

    // Create the group of buttons that select the mode
    ButtonGroup group = new ButtonGroup();

    // create buttons for points and lines, and add them to the list
    JRadioButton point =
        makeShapeButton(group, modeToolbar, "Point");
    JRadioButton line =
        makeShapeButton(group, modeToolbar, "Line");
    // add more shapes here

    // start by selecting the buttons for points
    point.doClick();
    return modeToolbar;
}
```

The `makeShapeButton` method actually creates the radio buttons. The radio buttons must be added to both the toolbar panel and the `ButtonGroup` group. This button group controls which radio buttons are associated with one another. As one button is selected within the group, the others are deselected.

This method also adds an action listener to each button. This code runs whenever the button is selected. We use an anonymous inner class to define the action listener class inline. Currently the action listener does nothing.

```
private JRadioButton makeShapeButton(ButtonGroup group,
                                     JPanel modeToolbar,
                                     String name) {
    JRadioButton b = new JRadioButton(name);
    group.add(b);
    modeToolbar.add(b);
    b.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                // does nothing
            }
        });
    return b;
}
```

32.2 Version B: Drawing Modes

Our first goal is to add some interaction to this application. In the example from the previous chapter, shapes were drawn at fixed positions by pressing buttons. Here we'd like to use the mouse to select the spots on the canvas to draw a number of points and lines.

In our first cut, we'll copy the implementation of the OCaml Paint program that we built in HW 6. In OCaml, we used a datatype to keep track of the current *mode* of the application. In Java, we can implement similar behavior with an `enum`. Below, we add the following two members to the `Paint` class—note that the enum is a nested class, and `mode` is a new field. We'll initialize this field with `Mode.PointMode` to record that we start drawing points.

```
public enum Mode {
    PointMode,
        LineStartMode,
        LineEndMode
        }

private Mode mode = Mode.PointMode;
```

Furthermore, we'll modify the action listener for the radio buttons so that when they are selected, they will change the mode. We can do this by adding an additional parameter to the `makeShapeButton` method. This parameter must be marked as **`final`** so that it may be referred to in the

`actionPerformed` method of the anonymous inner class.

```
private JRadioButton makeShapeButton(ButtonGroup group,
    JPanel modeToolbar, String name, final Mode buttonMode) {
        JRadioButton b = new JRadioButton(name);
        // ... add to button group and toolbar
        b.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                mode = buttonMode;
            }
        });
        return b;
}
```

When the radio buttons are created, we also need to specify their initial mode.

```
// create buttons for points and lines,
// and add them to the list
JRadioButton point =
    makeShapeButton(group, modeToolbar, "Point",
                    Mode.PointMode);
JRadioButton line =
    makeShapeButton(group, modeToolbar, "Line",
                    Mode.LineStartMode);
```

32.3 Version C: Basic Mouse Interaction

Now that the application can keep track of the drawing *mode*, we can set up the canvas so that it will respond to mouse clicks and movements. The behavior that we want is this: when we are in point mode, and the user clicks the mouse, we want to add a point at that particular location. When we are in `LineStartMode` and the user clicks the mouse, we want to remember where the user clicked and change the mode to `LineEndMode`. When we are in `LineEndMode`, we want to draw a line from the saved point to the point where the user clicked and revert back to `LineStartMode`.

We can implement this behavior with a mouse listener object that has methods for reacting to mouse behavior. The `MouseListener` interface describes the methods that such an event listener must include.

This interface includes several methods that run in response to mouse events such as mouse clicks, mouse button presses and mouse button re-

leases. Because we only want to add a listener for mouse clicks, we can implement this interface by extending the `MouseAdapter` class. This class has "stub" methods for the required methods in the `MouseListener` interface. In our implementation, we will *override* only the `mouseClicked` method.

```
private class Mouse extends MouseAdapter
  implements MouseListener {
    /** Location of the mouse when it was last pressed. */
    private Point modePoint;

    @Override
            public void mouseClicked(MouseEvent arg0) {
        Point p = arg0.getPoint();
        switch (mode) {
        case PointMode:
            shapes.add(new PointShape(color,p));
            break;
        case LineStartMode:
            mode = Mode.LineEndMode;
            modePoint = p;
            break;
        case LineEndMode:
            shapes.add(new LineShape(color,p,modePoint));
            mode = Mode.LineStartMode;
            break;
        }
        canvas.repaint();
    }
}
```

After the canvas is created (in the `run` method) we can add the mouse listener so that it can react to mouse events.

```
canvas.addMouseListener(new Mouse());  // mouse clicked events
```

32.4 Version D: Drag and Drop

The interaction in the previous version was very primitive. What if we'd like to emulate the drag-and-drop behavior of the OCaml Paint program? We can do so by copying that implementation.

For drag-and-drop, we need to add a new field, called `preview`, that stores a preview shape, to the `Paint` class.

```
  /** an optional shape for preview mode */
  private Shape preview;
```

During the `paintComponent` method of the `Canvas` class, we also need to draw this shape, if it is non-null.

```
public void paintComponent(Graphics gc) {
  super.paintComponent(gc);
  if (shapes != null){
    for (Shape s : shapes) {
      s.draw(gc);
    }
  }
  if (preview != null) {
    preview.draw(gc);
  }
}
```

Furthermore, we need to change how the canvas listens to mouse events, using the mouse class. Before, all the action was in the `mouseClicked` method. This time, we need to spread the action between several methods. We don't need to change the behavior for points too much, but for lines, we need to refine the behavior by overriding the `mousePressed`, `mouseReleased` and `mouseDragged` methods.

If the mode is `LineStartMode`, then when the mouse is pressed we need to save the location of the current click (in the field `modePoint`). We then switch modes to await dragging and the eventual end of the line (signaled by releasing the mouse button). Finally, we store the current location of the line

```
public void mousePressed(MouseEvent arg0){
    Point p = arg0.getPoint();
    switch (mode) {
    case LineStartMode:
        mode      = Mode.LineEndMode;
        modePoint = p;
        break;
    }
    canvas.repaint();
}
```

When the mouse is dragged, and we are in `LineEndMode`, we need to update the preview shape to reflect the current mouse position. As the mouse moves across the screen, the preview line will be continuously re-

drawn.

```
public void mouseDragged(MouseEvent arg0) {
    Point p = arg0.getPoint();
    switch (mode) {
    case LineEndMode:
        preview = new LineShape(color,modePoint,p);
        break;
    }
    canvas.repaint();
}
```

Only when the mouse is finally released do we add a new shape to the canvas. In this case we set the preview shape to null and switch back to `LineStartMode` to await a new shape.

```
public void mouseReleased(MouseEvent arg0) {
    Point p = arg0.getPoint();
    switch (mode) {
    case PointMode:
        shapes.add(new PointShape(color,p));
        break;
    case LineEndMode:
        mode = Mode.LineStartMode;
        shapes.add( new LineShape(color,modePoint, p));
        preview = null;
        break;
    }
    canvas.repaint();
}
```

32.5 Version E: Keyboard Interaction

In this next version of the paint program, we'd like to add the ability to control the program using the keyboard. In particular, we'd like to be able to switch between modes by pressing buttons. If the user presses the 'l' or 'L' button, the application should immediately switch to line mode (if it isn't there already). Likewise, if the user presses the 'p' or 'P' button, the application should switch to point drawing.

Giving a GUI component access to keyboard events requires that component to have the keyboard *focus*. There could be several different parts of the application that would like to react to key presses, but usually only

one should be reacting at a time. For example, if there are several text input areas in a window, only the selected one should display the typed text.

The component that we would like to add keyboard control to is the `modeToolbar`, the panel that contains the two radio buttons. We can add this functionality in the method that creates the component:

```
private JPanel createModeToolbar() {

    JPanel modeToolbar = new JPanel();

    // Create the group of buttons that select the mode
    ButtonGroup group = new ButtonGroup();

    // create buttons for points and lines,
    // and add them to the list
    final JRadioButton point =
        makeShapeButton(group, modeToolbar, "Point",
                        Mode.PointMode);
    final JRadioButton line =
        makeShapeButton(group, modeToolbar, "Line",
                        Mode.LineStartMode);

    // Add Keyboard control to application
    modeToolbar.setFocusable(true);
    modeToolbar.addKeyListener(new KeyAdapter() {
            public void keyTyped(KeyEvent arg0) {
                char c = arg0.getKeyChar();
                if (c == 'l' || c == 'L') {
                    line.doClick();
                } else if ( c == 'p' || c == 'P' ) {
                    point.doClick();
                }
            }
        });
    modeToolbar.requestFocusInWindow();
    // start by selecting the buttons for points
    point.doClick();
    return modeToolbar;
}
```

Note that after the radio buttons are created, we have inserted new code to add keyboard control. This code code first sets the panel to be “focusable”, i.e. able to receive the keyboard focus. It then adds a key

listener to the panel using an anonymous inner class. This anonymous class extends the `KeyAdapter` class and overrides the `keyPressed` method. When the panel has the focus and when a key is pressed on the keyboard, this method will be called.

The action of this method is to change the mode. We do that by forcing a click on the `point` and `line` radio buttons. We are faking button presses with these calls. Note that for the radio buttons to be accessible in the inner class, they must be declared to be **`final`** when they are initialized.

The last step is to actually request the keyboard focus, using the line `modeToolbar.requestFocusInWindow()`. No other component in the application will request the focus, so it will stay with the `modeToolbar`.

For more information about how to use key board control, see the Java tutorial: `http://docs.oracle.com/javase/tutorial/uiswing/events/keylistener.html`.

32.6 Interlude: Datatypes and enums vs. objects

How does our treatment of shape drawing in our current Java Paint example compare with the treatment in the OCaml GUI project?

In Java, we have

- An interface called `Shape` for drawable objects:

```
public interface Shape {
    public void draw(Graphics gc);
}
```

- Classes implement that interface (and describe how to draw themselves).

```
public class PointShape implements Shape { | }
public class LineShape implements Shape { | }
```

- A canvas that uses dynamic dispatch to draw the shapes

```
private class Canvas extends JPanel {
    public void paintComponent(Graphics gc) {
        super.paintComponent(gc);
        for (Shape s : actions)
            s.draw((Graphics2D)gc);
        if (preview != null)
            preview.draw((Graphics2D)gc);
    }
}
```

Alternatively, in the OCaml implementation, we had

- A datatype that specifies variants of drawable objects

```
type point = int * int
type shape =
  | Point   of Gctx.color * int * point
  | Line    of Gctx.color * int * point * point
```

- A Canvas that uses pattern matching to draw the shapes

```
let repaint (g:Gctx.t) : unit =
  let actions = List.rev paint.shapes in
  let drawit d =
    begin match d with
    | Point (c,t,p) ->
             Gctx.draw_points (set_params g c t) p
    | Line (c,t,p1,p2) ->
          Gctx.draw_line (set_params g c t) p1 p2
    end in
  List.iter drawit actions
```

The difference between these two versions is extensibility. The Java code can easily add more shapes by adding more classes. In the OCaml code, we add more shapes by adding more variants, and by modifying the repaint code in the canvas for drawing.

This example demonstrates the differences between using datatypes and objects. With datatypes, the focus is how the data is stored. It is easy to add more operations that work with the data, but harder to add more variants, or different types of data. Conversely, with objects, the focus is on how the data is used. It is east to add more variants (i.e. another class), but harder to add more operations to the interface, because all existing classes need to be modified.

Datatypes are better for situations where the structure of the data is fixed, such as in binary search trees. However, objects are better when the interface to the data is fixed, such as drawing shapes.

Returning to our paint example, we've used objects for storing the shapes, but we are still using variants for the drawing modes. That's not very extensible, as we add new shapes we'll have to add new drawing modes, and modify the mouse listener to react to those new modes. Can we do better?

32.7 Version F: OO-based Refactoring

In this section we will replace the enum for the drawing mode with a new definition that will be more extensible. It will require refactoring the code in several places, as well as a bit of indirection (or delegation) in the mouse listener.

Currently, the mouse listener uses a switch statement (on the current mode) to decide what code to execute. We would like to change that to use dynamic dispatch with the mode itself. In otherwords, we would like to modify the mouse listener so that the current mode determines the action in each situation.

```
private class Mouse extends MouseAdapter {
    public void mousePressed(MouseEvent arg0) {
        mode.mousePressed(arg0);
        canvas.repaint();
    }
    public void mouseDragged(MouseEvent arg0) {
        mode.mouseDragged(arg0);
        canvas.repaint();
    }
    public void mouseReleased(MouseEvent arg0) {
        mode.mouseReleased(arg0);
        canvas.repaint();
    }
}
```

What this implies is that *the mode is itself a mouse listener*. Therefore, we will change the definition of the mode type to reflect that:

```
interface Mode extends MouseListener, MouseMotionListener {  }
```

The various modes themselves (such as `PointMode`, `LineStartMode`, etc.) will be classes that implement the `Mode` interface. Because this interface requires that the class implement the `mousePressed`, `mouseDragged` and `mouseReleased` methods, the code above makes sense.

The advantage of this design means that we can think about the mouse reactions in each mode independently. For example, the `PointMode` class needs to only react to `mouseReleased` events. All other events are ignored in this mode.

```
class PointMode extends MouseAdapter implements Mode {
    public void mouseReleased(MouseEvent e) {
        Point p = e.getPoint();
        shapes.add(new PointShape(color,p));
    }
}
```

The line drawing modes work together, defined by the following pair of classes. The starting mode just switches the current mode to the ending mode, saving the current point by passing it as a constructor to the newly created mode.

The ending mode updates the preview field as the mouse is dragged (creating the line using the saved point in the mode as well as the current location of the mouse.) When the mouse is released, it switches the mode back to the starting mode, adds the new shape to the list of shapes to draw, and resets the preview field.

```
class LineStartMode extends MouseAdapter implements Mode {
    public void mousePressed(MouseEvent e) {
        mode    = new LineEndMode(e.getPoint());
    }

}

class LineEndMode extends MouseAdapter implements Mode {
    Point modePoint;
    LineEndMode (Point p) { modePoint = p; }

    public void mouseDragged(MouseEvent arg0) {
        Point p = arg0.getPoint();
        preview = new LineShape(color,modePoint,p);
    }

    public void mouseReleased(MouseEvent arg0) {
        mode = new LineStartMode ();
        Point p = arg0.getPoint();
        shapes.add( new LineShape(color,modePoint, p));
        preview = null;
    }
}
```

Compared to the previous version, the switching based on the mode is implicit. The mouse listener uses dynamic dispatch to figure out the correct reaction to mouse events.

As we add new shapes to this application, we can add new modes just by adding new classes. We don't need to modify any existing classes. Indeed, these classes are defined as inner classes of the `Paint` application for convenient access to the `mode`, `shapes` and `preview` fields. However, we could move these class definitions outside of the class as long as each mode had a way of accessing these components.

Bibliography

[1] David Bayles and Ted Orland. *Art & Fear: Observations on the Perils (and Rewards) of Artmaking*. Image Continuum Press, 1993.

[2] Joshua Bloch. *Effective Java (Second Edition)*. Adison-Wesley, 2008.

[3] Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, and Shriram Krishnamurthi. *How to Design Programs: An Introduction to Programming and Computing*. MIT Press, 2003. Available online at `http://www.htdp.org`.

[4] David Flanagan. *Java in a Nutshell (5th Edition)*. O'Reilly Media, 2005.

[5] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1 edition, November 1994.