How to Design Classes

DRAFT: June 15, 2012

How to Design Classes

Data: Structure and Organization

Matthias Felleisen
Matthew Flatt
Robert Bruce Findler
Kathryn E. Gray
Shriram Krishnamurthi
Viera K. Proulx

Contents

Preface

The goals were to provide modules and to eliminate assignment altogether.

—Alan Kay, *History of Smalltalk*

Diagrams

Draw them by hand. Diagrams are the programmer's doodling language. Using a tool that draws them for you from the existing code defeats the purpose. Assigning students to do so defeats the purpose of learning to doodle.

Stuff

Conventional text books at this level present object-oriented programming as an extension of imperative programming. The reasoning is that computations interact with the real world where physical objects change their state all the time. For example, people get sick and become patients; balls drop from a hand and change location; and so on. Therefore, the reasoning goes, computational objects, which represent physical objects, encapsulate and hide the state of the world for which they are responsible and, on method calls, they change state. Naturally, people begin to express computations as sequences of assignment statements that change the value of variables and fields (instance variables).

We disagree with this perspective and put classes and the design of classes into the center of our approach. In "How to Design Programs" we defined classes of data. As we developed larger and larger programs, it became clear that the design of a program requires the introduction of many classes of data and the development of several functions for each class. The rest is figuring out how the classes of data and their functions related to each other.

In this volume, we show students how object-oriented programming

languages such as C# and Java support this effort with syntactic constructs. We also refine the program design discipline.

What you will learn

What you won't learn

Java.

It is a good idea to study the programming language that you use on a daily basis and to learn as much as possible about it. We strongly believe, however, that it is a bad idea to teach the details of any programming language in a course. Nobody can predict which programming language you will use. Therefore, time in a course is better spent on studying the general principles of program design rather than the arcane principles of any given programming language.

Why Java?

Why ProfessorJ?

ProfessorJ is not Java; it is a collection of relatively small, object-oriented programming languages made up for teaching the design of classes, i.e., the essence of object-oriented programming. For the most part, they are subsets of Java but, to support the pedagogy of this book, they also come with constructs for playing with examples and testing methods.

ProfessorJ is useful for the first four chapters. After that, it is essential that you switch to a full-fledged programming language and an industrial programming environment. This may mean switching to something like Java with Eclipse or C# with Microsoft's Visual Studio.

Acknowledgments

Daniel P. Friedman, for asking the first author to co-author *A Little Java, A Few Patterns* (also MIT Press);

Richard Cobbe for many nagging questions on Java

typos: David van Horn

PICTURE: should be on even page, and even pages must be on the left

Purpose and Background

The goal of this chapter is to develop data modeling skills.

We assume that students have understood data definitions in Parts I, II, and III of *How to Design Programs*. That is, they should understand what it means to represent atomic information (numbers, strings), compounds of information (structures), unions of information (unions), and information of arbitrary size (lists, trees); ideally students should also understand that a program may rely on an entire system of interconnected data definitions (family trees, files and folders).

In the end, students should be able to design a system of classes to represent information. In particular, when given a system of classes and a piece of information, they should be able to create objects and represent this information with data; conversely, given an instance of a class in the system, they should be able to interpret this object as information in the "real" world.

TODO

— add examples that initialize fields between sections 1 and LAST

— add exercises that ask students to represent something with classes that isn't completely representable; they need to recognize what to omit when going from information to data. do it early in the chapter.

— add modeling exercise to Intermezzo 1 that guides students through the process of modeling Java syntax (information) via Java classes (data) start with an exercise that says

class ClassRep String name; ClassRep(String name) this. name = name;

and ask them to translate a class with fields and one without.

I   The Varieties of Data

In *How to Design Programs*, we learned that the systematic design of a program requires a solid understanding of the problem information. The first step of the design process is therefore a thorough reading of the problem statement, with the goal of identifying the information that the requested program is given and the information that it is to compute. The next step is to represent this information as data in the chosen programming language. More precisely, the programmer must describe the classes of *all* possible input data and all output data for the program. Then, and only then, it is time to design the program itself.

Thus, when you encounter a new programming language, your first goal is to find out how to represent information in this language. In *How to Design Programs* you used an informal mixture of English and Scheme constructors. This book introduces one of the currently popular alternatives: programming in languages with a notation for describing classes of data within the program. Here the word "class" is short for "collection" in the spirit in which we used the word in *How to Design Programs*; the difference is that the designers of such languages have replaced "data" with "object," and that is why these languages are dubbed object-oriented.

Of course, for most problems describing a single class isn't enough. Instead, you will describe many classes, and you will write down how the classes are related. Conversely, if you encounter a bunch of related classes, you must know how to interpret them in the problem space. Doing all this takes practice—especially when data descriptions are no longer informal comments but parts of the program—and that is why we dedicate the entire chapter to the goal of designing classes and describing their relationships. For motivational previews, you may occasionally want to take a peek at corresponding sections of chapter II, which will introduce functions for similar problems and classes as the following sections.

ProfessorJ:
Beginner

1   Primitive Forms of Data

Like Scheme, Java provides a number of built-in atomic forms of data with which we represent primitive forms of information. Here we use four of them: int, double, boolean, and *String*.[1]

How to Design Programs uses *number* to represent numbers in the problem domain. For integers and rational numbers (fractions or numbers with a decimal point), these representations are exact. When we use Java—and most other programming languages—we give up even that much precision. For example, while Java's int is short for integer, it doesn't truly include all integers—not even up to the size of a computer. Instead, it means the numbers

$$\text{from } -2147483648 \text{ to } 2147483647 \ .$$

If an addition of two ints produces something that exceeds 2147483647, Java finds a *good enough*[2] number in the specified range to represent the result. Still, for our purposes, int is often reasonable for representing integers.

In addition to exact integers, fractions, and decimals, *How to Design Programs* also introduced inexact numbers. For example, the square root function (sqrt) produces an inexact number when given 2. For Java and similar languages, double is a discrete collection of rational numbers but is used to represent the real numbers. That is, it is roughly like a large portion of the real number line but with large gaps. If some computation with real numbers produces a number that is in the gap between two doubles, Java somehow determines which of the two is a *good enough* approximation to the result and takes it. For that reason, computations with doubles are inherently inaccurate, but again, for our purposes we can think of doubles as a strange form of real numbers. Over time, you will learn when to use ints and when to use doubles to represent numeric information.

As always, the boolean values are true and false. We use them to represent *on/off* information, *absence/presence* information, and so on.

Finally, we use *String*s to represent symbolic information in Java. Symbolic information means the names of people, street addresses, pieces of conversations, and similarly symbolic information. For now, a *String* is a sequence of keyboard characters enclosed in quotation marks; e.g.,

---

[1]For Java, *String* values are really quite different from integers or booleans. We ignore the difference for now.

[2]Java uses modular arithmetic; other languages have similar conventions.

```
"bob"
"#$%ˆ&"
"Hello World"
"How are U?"
"It is 2 good to B true."
```

Naturally, a string may not include a quotation mark, though there are ways to produce *String*s that contain this character, too.

2   Classes

For many programming problems, we need more than atomic forms of data to represent the relevant information. Consider the following problem:

> . . .   Develop a program that keeps track of coffee sales at a specialty coffee seller. The sales receipt must include the kind of coffee, its price (per pound), and its weight (in pounds). . . .

The program may have to deal with hundreds and thousands of sales. Unless a programmer keeps all the information about a coffee sale together in one place, it is easy to lose track of the various pieces. More generally, there are many occasions when a programmer must represent several pieces of information that always go together.

Our sample problem suggests that the information for a coffee sale consists of three (relevant) pieces: the kind of coffee, its price, and its weight. For example, the seller may have sold

1. 100 pounds of Hawaiian Kona at $20.95/pound;

2. 1,000 pounds of Ethiopian coffee at $8.00/pound; and

3. 1,700 pounds of Colombian Supreme at $9.50/pound.

In *How to Design Programs*, we would have used a class of structures to represent such coffee sales:

(**define-struct** *coffee* (*kind price weight*))
;; *Coffee* (sale) is:
;; — (*make-coffee String Number Number*)

The first line defines the shape of the structure and operations for creating and manipulating structures. Specifically, the definition says that a *coffee* structure has three fields: *kind*, *price*, and *weight*. Also, the constructor is called *make-coffee* and to get the values of the three fields, we can use the

functions *coffee-kind*, *coffee-price*, and *coffee-weight*. The second and third line tells us how to use this constructor; it is applied to a string and two numbers:

> (*make-coffee* `"Hawaiian Kona"` 20.95 100)
> (*make-coffee* `"Ethiopian"` 8.00 1000)
> (*make-coffee* `"Colombian Supreme"` 9.50 1)

Note the use of ordinary numbers, which in Beginning Student Scheme denote precisely the decimal numbers written down.

In Java, we define the CLASS in figure 1 for the same purpose. The left column of the figure shows what such a class definition looks like. The first line is a Java-style end-of-line comment, spanning the rest of the line; the slashes are analogous to ";;" in Scheme. The second line

> **class** *Coffee* {

announces a class definition. The name of the class is *Coffee*. The opening brace "{" introduces the items that belong to the class. The next three lines

> *String kind*;
> int *price*; // *cents per pound*
> int *weight*; // *pounds*

state that an element of the *Coffee* class (think collection for the moment) has three FIELDs. Their names are *kind*, *price*, and *weight*. The left of each field declaration specifies the type of values that the field name represents. Accordingly, *kind* stands for a *String* and *price* and *weight* are ints.

The two lines for *price* and *weight* definitions end with comments, indicating which monetary and physical units that we use for *price* and *weight*. Valuating prices in integer cents is good practice because Java's calculations on integers are accurate as long as we know that the numbers stay in the proper interval.[3]

The next four lines in *Coffee* define the CONSTRUCTOR of the class:

> *Coffee*(*String kind*, int *price*, int *weight*) {
>   **this**.*kind* = *kind*;
>   **this**.*price* = *price*;
>   **this**.*weight* = *weight*;
> }

---

[3]If you wish to design programs that deal with numbers properly, you must study the principles of numerical computing.

```
// the receipt for a coffee sale
class Coffee {
  String kind;
  int price; // cents per pound
  int weight; // pounds

  Coffee(String kind, int price, int weight) {
    this.kind = kind;
    this.price = price;
    this.weight = weight;
  }
}
```

Coffee
String *kind* int *price* [in cents per pound] int *weight* [in pounds]
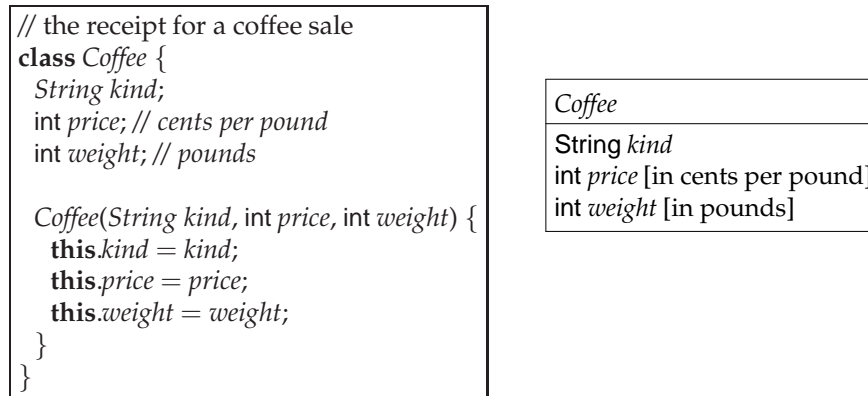
Figure 1: A class definition and a class diagram

Unlike Scheme, Java doesn't define this constructor automatically. Instead, the programmer must add one manually. For now, we use the following "cookie-cutter" approach to defining constructors:

1. A constructor's body starts with an opening brace ({).

2. The parameters of a constructor look like the fields, separated by commas (",").

3. Its body is a semicolon-separated (";") series of "equations" of the shape[4] "**this**.*fieldname* = *fieldname*;" and there are as many "equations" as fields.

4. The last line of a constructor body is a closing brace (}).

The last line of the class definition is }, closing off the class definition.

Even though a Java data representation looks more complex now than a corresponding Scheme data definition, the two are actually equivalent. In *How to Design Programs*, we used a structure definition *with* a data definition to describe (in code and in English) how a piece of data represents some information. Here we combine the two into a single class definition. The only extra work is the explicit definition of the constructor, which **define-struct** introduces automatically. Can you think of situations when the ability to define your own constructor increases your programming powers?

---

[4]No, they are not really equations, they just look like that.

After we have defined a class, it is best to translate some sample pieces of information into the chosen representation. This tests whether the defined class is adequate for representing some typical problem information and, later on, we can use these examples to test our program(s). For example, to create an object of the *Coffee* class, you apply the constructor to as many values as there are parameters:

new *Coffee*(`"Hawaiian Kona"`,2095,100)

The notation differs a bit from Scheme; here the three values are enclosed in parentheses and separated by commas.[5]

The application of the constructor creates an INSTANCE—also known as an OBJECT—of the *Coffee* class. Here we obtain the first sample of information for our problem, that is, the sale of 100 pounds of Kona for US ¢ 2095 per pound. The other two samples have similar translations:

new *Coffee*(`"Ethiopian"`, 800, 1000)

and

new *Coffee*(`"Colombia Supreme"`, 950, 1)

Finally, before we move on to a second example, take a look at the right side of figure 1. It is a pictorial illustration of the *Coffee* class, which is useful because of the notational overhead for Java's class definition. The rectangle has two pieces. The upper portion names the class; the lower portion lists the fields and their type attributes. This picture is called a CLASS DI-AGRAM.[6] These diagrams are particularly useful while you analyze problems and for discussions on how to represent information as data. You will encounter many of them in this book.

Here is another excerpt from a typical programming problem:

. . . Develop a program that helps you keep track of *daily* . . .

The word "daily" implies you are working with many dates. One date is described with three pieces of information: a day, a month, and a year.

In Java, we need a class with three fields. We name the class *Date* and the fields *day*, *month*, and *year*. Using ints to count days, months, and years is natural, because that's what people do. This suggests the class diagram and the class definition in figure 2.

Let's look at some instances of *Date*:

---

[5]This mimics ordinary mathematical notation for functions of several arguments.

[6]Work on object-oriented languages has produced a number of diagram languages. Our class diagrams are loosely based on the Unified Modeling Language (UML).

Date
int *day*
int *month*
int *year*

```
// calendar dates
class Date {
  int day;
  int month;
  int year;

  Date(int day, int month, int year) {
    this.day = day;
    this.month = month;
    this.year = year;
  }
}
```
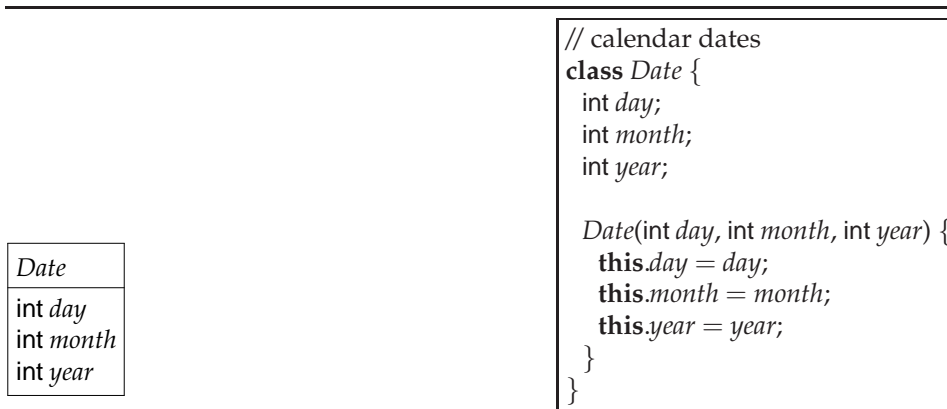
Figure 2: Representing dates

---

1. **new** *Date*(5, 6, 2003) stands for June 5, 2003;

2. **new** *Date*(6, 6, 2003) stands for June 6, 2003; and

3. **new** *Date*(23, 6, 2000) stands for June 23, 2000.

Of course, we can also write **new** *Date*(45, 77, 2003). This expression creates an instance of *Date*, but not one that corresponds to a true calendar date.[7]

Let's take a look at a third problem statement:

> ... Develop a GPS-based navigation program for cars. The GPS device feeds the program with the current location at least once a second. The location is given as latitude and longitude.
>
> Examples:
>
> 1. latitude 33.5, longitude 86.8;
>
> 2. latitude 40.2, longitude 72.4; and
>
> 3. latitude 49.0, longitude 110.3.
>
> ...

The relevant information is a called a GPS location.

---

[7]Recall the question on page 2 suggesting that a constructor can perform additional tasks. For *Date*, your constructor could check whether the dates make sense or whether it is a valid date in the sense of some (given) calendar. Explore Java's *Date* class via on-line resources.

```
// GPS locations
class GPSLocation {
  double latitude; // degrees ;
  double longitude; // degrees

  GPSLocation(double latitude, double longitude) {
    this.latitude = latitude;
    this.longitude = longitude;
  }
}
```

Location
double *lattitude* [in degrees]
double *longitude* [in degrees]
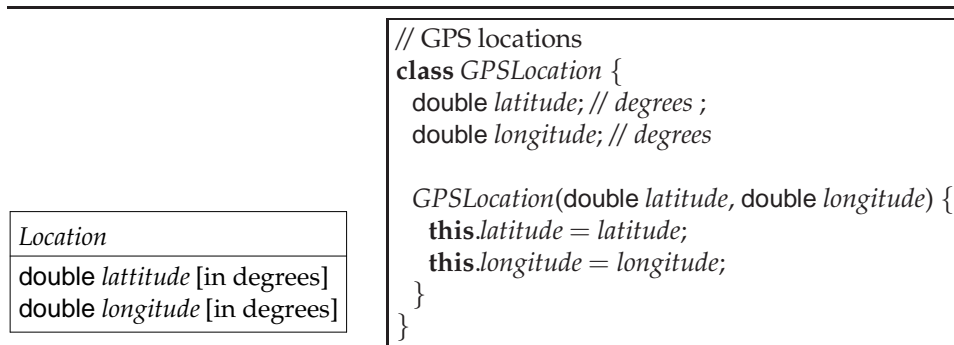
Figure 3: Representing navigations in a GPS

A class for GPS locations needs two fields: one for latitude and one for longitude. Since both are decimal numbers that are approximate anyway, we use doubles to represent them. Figure 3 contains both the class diagram and the class definition. As for turning data into information (and vice versa) in this context, see exercise 2.1.

```
// moving balls on a pool table
class Ball {
  int x;
  int y;
  int RADIUS = 5;

  Ball(int x, int y) {
    this.x = x;
    this.y = y;
  }
}
```

Ball
int *x*
int *y*
int *RADIUS*
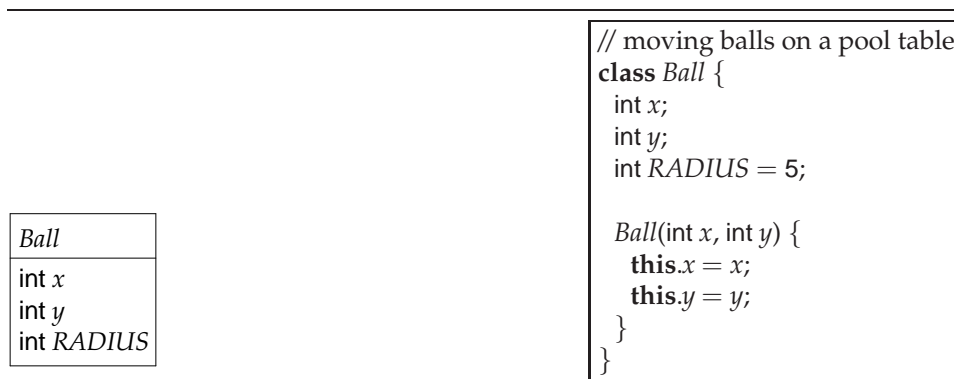
Figure 4: Representing pool balls

On occasion, a class describes objects that all share an attribute:

... Develop a simulation program for balls on a pool table. ...

Clearly, we need a class to describe the collection of balls. Movement on a two-dimensional surface suggests that a ball object includes two fields, one per coordinate. In addition, the balls also need a radius, especially if the program is to compute bounces between balls or if it is supposed to draw the balls.

Now the radius of these balls is going to stay the same throughout the simulation, while their coordinates constantly change. To express this distinction and to simplify the creation of instances of *Ball*, a programmer adds an initialization "equation" directly to a field declaration: see *RADIUS* in figure 4. By convention, we use uppercase letters for the names of fields to indicate that they are constant, shared attributes.

If a field comes with an initialization "equation", the constructor does *not* contain an equation for that field. Thus, the *Ball* constructor consumes two values, *x* and *y*, and contains two "equations:" one for *x* and one for *y*. To create instances of *Ball*, you can now write expressions such as **new** *Ball*(10,20), which produces an object with three attributes: *x* with value 10; *y* with value 20; and *RADIUS* with value 5. Explore the creation of such objects in the interactions window.

```
// collect examples of coffee sales
class CoffeeExamples {
  Coffee kona = new Coffee("Kona",2095,100);
  Coffee ethi = new Coffee("Ethiopian", 800, 1000);
  Coffee colo = new Coffee("Colombian", 950, 20);

  CoffeeExamples() { }
}
```

Figure 5: An example class

Before moving on, let's briefly turn to the administration of examples. A good way to keep track of examples is to create a separate class for them. For example, figure 5 shows how to collect all the sample instances of *Coffee* in a single *Examples* class. As you can see, all fields are immediately initialized via "equations" and the constructor has neither parameters nor constructor "equations." Creating an instance of the *CoffeeExamples* class also creates three instances of the *Coffee* class, which ProfessorJ's interactions window visualizes immediately so that you can inspect them. Of course, such example classes thus introduce a concept that we haven't covered yet: objects that contain objects. The next section is all about this idea.

2.1 Finger Exercises on Plain Classes

Exercise 2.1 Formulate the examples of information from the GPS problem as instances of *GPSLocation*.

ProfessorJ: Examples

What is the meaning of **new** *GPSLocation*(42.34,71.09) in this context? Why does this question (as posed) not make any sense?

If we tell you that the first number is associated with north and the second one with west, can you make sense out of the question? Can you find this place on a globe? On a map of the USA? ∎

Exercise 2.2 Take a look at this problem statement:

> . . . Develop a program that assists bookstore employees. For each book, the program should track the book's title, its price, its year of publication, and the author's name. . . .

Develop an appropriate class diagram (by hand) and implement it with a class. Create instances of the class to represent these three books:

1. Daniel Defoe, *Robinson Crusoe*, $15.50, 1719;

2. Joseph Conrad, *Heart of Darkness*, $12.80, 1902;

3. Pat Conroy, *Beach Music*, $9.50, 1996.

What does **new** *Book*(`"D. P. Friedman"`,`"The Little LISPer"`,900,1974) mean? Does the question make sense? What do you need to know to interpret this piece of data? ∎

Exercise 2.3 Add a constructor to the following partial class definition and draw the class diagram (by hand):

```
// represent computer images
class Image {
  int height; // pixels
  int width; // pixels
  String source; // file name
  String quality; // informal
  . . .
}
```

The class definition was developed for this problem statement:

> . . . Develop a program that creates a gallery from image descriptions that specify the height, width, and name of the source file, plus some additional information about their quality. . . .

Explain what the expressions mean in the problem context:

> **new** *Image*(5, 10, `"small.gif"`, `"low"`)
> **new** *Image*(120, 200, `"med.gif"`, `"low"`)
> **new** *Image*(1200, 1000, `"large.gif"`, `"high"`) .

Suppose the web designer decides that the *Image* class should also specify to which gallery the images belong so that functions for this class of data have the information handy. Modify the class definition appropriately. Assume that the gallery is the same for all instances of *Image*. ∎

Automobile
String *model*
int *price* [in dollars]
double *mileage* [in miles per gallon]
boolean *used*
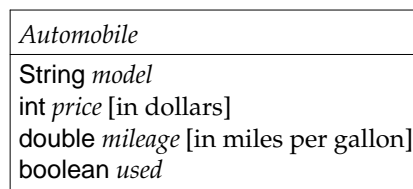
Figure 6: A class diagram for automobiles

Exercise 2.4 Translate the class diagram in figure 6 into a class definition. Also create instances of the class. ∎

Exercise 2.5 Create three instances of the following class:

> // introducing the concept of gravity
> **class** *Apple* {
>   int *x*;
>   int *y*;
>   int *RADIUS* = 5;
>   int *G* = 10; // *meters per second square*
>
>   *Apple*(int *x*, int *y*) {
>   **this**.*x* = *x*;
>   **this**.*y* = *y*;
>   }
> }

How many attributes describe each instance? How many arguments does the constructor consume? ∎

2.2   Designing Classes

This first section on classes suggests that the design of a class proceeds in three steps:

1. Read the problem statement. Look for statements that mention or list the attributes of the objects in your problem space. This tells you how many fields you need for the class definition and what information they represent. Write down your findings as a class diagram because they provide a quick overview of classes, especially when you are dealing with many classes or when you return to the problem in a few months from now.

2. Translate the class diagram into a class definition, adding a purpose statement to each class. The purpose statement should explain what information the instance of the class represent (and how).

   This translation is mostly mechanical. Only one part needs your full attention: the decision whether a field should have the same value for *all* instances of this class. If so, use an initialization equation with the field; otherwise, add the field as a parameter to the constructor and add an equation of the shape **this**.*field* = *field* to the constructor.

3. Obtain examples of information and represent them with instances of the class. Conversely, make up instances of the class and interpret them as information.

   Warning: Keep in class that some data examples have no explanation in the problem space. You may wish to warn future readers of your code about such instances with a comment. ∎

   This design recipe for creating classes corresponds to the first step in the design recipe of *How to Design Programs*. There, we said that all designs begin with a thorough understanding of the classes of data that a problem statement describes. The resulting descriptions were informal, and we dealt with simple forms of data. Here we use the programming language itself (Java) to describe classes of data, making this step much more rigorous than before. The rest of this chapter introduces complex forms of data and refines this design recipe.

3 Class References, Object Containment

What we have seen so far is that a class in Java is somewhat like a structure in Scheme. Each instance compounds several pieces of data into one. Your experience—both in programming and in the real world—should tell you that this kind of compounding can happen at several levels, that is, a piece of information may contain some other piece of information that consists of many pieces of information and so on. Let's look at an example:

> ... Develop a program that manages a runner's training log. Every day the runner enters one entry about the day's run. Each entry includes the day's date, the distance of the day's run, the duration of the run, and a comment describing the runner's post-run disposition. ...

Clearly, a log entry consists of four pieces of information: a date, a distance, a duration, and a comment. To represent the last three, we can use Java's primitive types: double (for miles), int (for minutes), and *String*. As we have seen in section 2, however, the natural representation for dates consists of three pieces; it is not a basic type.

Let's make up some examples before we formulate a data definition:

on June 5, 2003	5.3 miles	27 minutes	feeling good
on June 6, 2003	2.8 miles	24 minutes	feeling tired
on June 23, 2003	26.2 miles	150 minutes	feeling exhausted
...	...	...	...

The three recordings are from three distinct dates, with widely varying mileage and post-practice feelings.

If we were to represent these forms of data in Scheme we would formulate two structure definitions and two data definitions:

(**define-struct** *entry* (*date distance duration comment*))
;; *Entry* is:
;; — (*make-entry Date Number Number String*)

(**define-struct** *date* (*day month year*))
;; *Date* is:
;; — (*make-date Number Number Number*)

The first pair specifies the class of *Entry*s, the second the class of *Date*s. Just as our analysis of the problem statement says, the data definition for *Entry* refers to the data definition for *Date*s.

```
                                              // an entry in a runner's log
                                              class Entry {
                                                Date d;
 ┌─────────────────────────┐                    double distance; // miles
 │ Entry                   │                     int duration; // minutes
 ├─────────────────────────┤                     String comment;
 │ Date d                  │
 │ double distance [in miles] │                  Entry(Date d,
 │ int duration [in minutes]  │                        double distance,
 │ String comment          │                          int duration,
 └─────────────────────────┘                          String comment) {
                                                  this.d = d;
        ┌─────────────┐                           this.distance = distance;
        │ Date        │◁─────                      this.duration = duration;
        ├─────────────┤                            this.comment = comment;
        │ int day     │                          }
        │ int month   │                        }
        │ int year    │
        └─────────────┘                      // calendar dates
                                             class Date {
                                               int day;
                                               int month;
                                               int year;

                                               Date(int day, int month, int year) {
                                                 this.day = day;
                                                 this.month = month;
                                                 this.year = year;
                                               }
                                             }
```

Figure 7: Representing a runner's log

Using these data definitions as guides for making up data, translating the three examples into Scheme data is straightforward:

(*make-entry* (*make-date* 5 6 2003) 5.3 27 "Good")
(*make-entry* (*make-date* 6 6 2003) 2.8 24 "Tired")
(*make-entry* (*make-date* 23 6 2003) 26.2 150 "Exhausted")

When instances of structures are nested, like in this case, it often makes sense to create the structures in a stepwise fashion, using definitions to give names to the values:

(**define** *d1* (*make-date* 5 6 2003))
(**define** *e1* (*make-entry d1* 5.3 27 "Good"))

For practice, construct the last two examples in the same fashion.

We already know how to express the data definition for *Date*s with a class diagram. It is just a rectangle whose name portion is *Date* and whose field portion contains three pieces: one for the day, one for the month, and one for the year.

The diagram for *Entry*s is—in principle—also just a box. The field portion contains four pieces: one for *Date*, one for the miles, one for the minutes, and one for the post-run condition. The difference between *Entry* and classes such as *Date* or *GPSLocation* is that the type of the first field is not a primitive Java type but another class. In the case of *Entry*, it is *Date*. In class diagrams we explicate this relationship with a CONTAINMENT AR-ROW from the *Date* field to the box for *Date*, which indicates that instances of *Entry* contains an instance of *Date*. The left side of figure 7 displays the complete diagram.

The right side of figure 7 shows how to translate this data definition into class definitions. Roughly speaking, we just add constructors to each box. The constructor for *Entry* consumes four pieces of data; the first of those is an instance of the class *Date*. Conversely, if we wish to construct an instance of *Entry*, we must first construct an instance of *Date*, just like in Scheme. We can either just nest the uses of the constructors:

new *Entry*(**new** *Date*(5, 6, 2003), 5.3, 27, "Good")

or, we can construct the same value with an auxiliary definition:

Date d1 = **new** *Date*(5, 6, 2003);
Entry e1 = **new** *Entry*(*d1*, 5.3, 27, "Good");

Like every definition in Java, this definition starts with the type of the values (*Date*) that the variable (*d1*) may represent. The right-hand side of the definition is the value for which *d1* is a placeholder.

In Java, it is best to introduce definitions for all values:

Date d2 = **new** *Date*(6, 6, 2003);
Date d3 = **new** *Date*(23, 6, 2003);

Entry example2 = **new** *Entry*(*d2*, 2.8, 24, "Tired");
Entry example3 = **new** *Entry*(*d3*, 26.2, 150, "Exhausted");

It makes it easy to refer to these values in examples and tests later.

Let's practice our data design skills with another example:
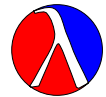
ProfessorJ:
More on Examples

```
                                              // a restaurant in Manhattan
                                              class Restaurant {
                                                String name;
                                                String kind;
   Restaurant                                  String pricing;
                                               Place place;
   String name
   String kind                                 Restaurant(String name,
   String pricing                                          String kind,
   Place place                                             String pricing,
                                                           Place place) {
                                                 this.name = name;
                                                 this.kind = kind;
      Place                                      this.pricing = pricing;
                                                 this.place = place;
   int ave                                       }
   int street                                 }

                                              // a intersection in Manhattan
                                              class Place {
                                                int ave;
                                                int street;

                                                Place(int ave, int street) {
                                                  this.ave = ave;
                                                  this.street = street;
                                                }
                                              }
```

Figure 8: Representing restaurants in Manhattan

. . . Develop a program that helps a visitor navigate Manhattan's restaurant scene. The program must be able to provide four pieces of information for each restaurant: its name, the kind of food it serves, its price range, and the closest intersection (street and avenue).

Examples: (1) La Crepe, a French restaurant, on 7th Ave and 65th Street, moderate prices; (2) Bremen Haus, a German restaurant on 2nd Ave and 86th Street, moderate; (3) Moon Palace, a Chinese restaurant on 10th Ave and 113th Street, inexpensive; . . .

Again, three of the pieces of information can be represented with Java's primitive types, but the location consists of two pieces of information.

Our problem analysis suggests that we need two data definitions: one for restaurants and one for places. Both consist of several pieces of information, and the former refers to the latter. This suggests the class diagram on the left side of figure 8. You can derive the corresponding pair of class definitions on the right side in a mechanical manner. Of course, the arrow doesn't show up in the text; it is implicit in the type of the field.

Now that we have a class definition, we turn our examples of information into examples of data:

> *Place p1* = **new** *Place*(7, 65);

> *Restaurant ex1* =
>   **new** *Restaurant*(`"La Crepe"`, `"French"`, `"moderate"`, *p1*);

We have used two definitions for the first example; translate the others on your own using the same style.

Object containment is not restricted to one level:

> ... Develop a program that can assist railway travelers with the arrangement of train trips. The available information about a specific train includes its schedule, its route, and whether it is local. (The opposite is an *express* train.) The route information consists of the origin and the destination station. A schedule specifies the departure and the arrival times. ...

Obviously, this problem statement refers to many different classes of data, and all these classes are related.

The most important one is the class of *Train*s. A train has three attributes. To represent whether a train is local, we just use a boolean field, called **local**. For the other two fields, it's best to introduce new classes—*Route*s and *Schedule*s—because each represents more than what a primitive Java type can represent. A *Route* consists of two pieces of information, the origin and the destination stations; for those, *String*s are fine representations. For *Schedule*s, we need the (clock) time when the train leaves and when it arrives. Usually, a *ClockTime* consists of two ints: one for the hour (of the day) and one for the minutes (of the hour).

As we conduct the data analysis, it is natural to draw a class diagram, especially for cases like this one, where we need four classes with three connections. Figure 9 shows the final result of this effort. From there, it
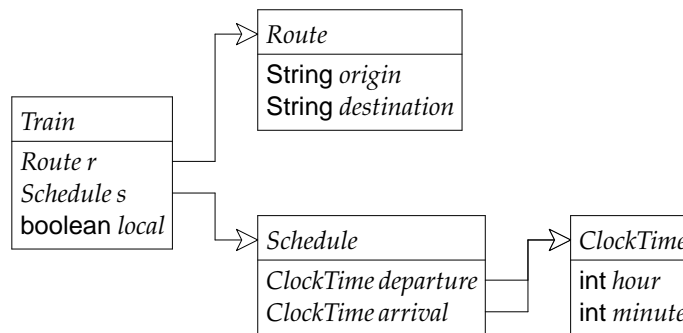
Figure 9: A class diagram for train schedules

```
// trains from a train company
class Train {
  Route r;
  Schedule s;
  boolean local;

  Train(Route r, Schedule s, boolean local) {
    this.r = r;
    this.s = s;
    this.local = local;
  }
}
```

```
// a train schedule
class Schedule {
  ClockTime departure;
  ClockTime arrival;


  Schedule(ClockTime departure,
           ClockTime arrival) {
    this.departure = departure;
    this.arrival = arrival;
  }
}
```

```
// a train route
class Route {
  String origin;
  String destination;

  Route(String origin, String destination){
    this.origin = origin;
    this.destination = destination;
  }
}
```

```
// the time on a 24-hour clock
class ClockTime {
  int hour;
  int minute;

  ClockTime(int hour, int minute){
    this.hour = hour;
    this.minute = minute;
  }
}
```

Figure 10: Classes for a train schedule

is a short step to an equivalent class definition; see figure 10 for the four definitions.

Let's look at some examples:

Route r1 = **new** *Route*(`"New York"`, `"Boston"`);
Route r2 = **new** *Route*(`"Chicago"`, `"New York"`);

ClockTime t1 = **new** *ClockTime*(23, 50);
ClockTime t2 = **new** *ClockTime*(13, 20);
ClockTime t3 = **new** *ClockTime*(10, 34);
ClockTime t4 = **new** *ClockTime*(13, 18);

Schedule s1 = **new** *Schedule*(*t1,t2*);
Schedule s2 = **new** *Schedule*(*t3,t4*);

Train train1 = **new** *Train*(*r1*, *s1*, true);
Train train2 = **new** *Train*(*r2*, *s2*, false);

This collection of definitions introduces two trains, two schedules, four clock times, and two routes. Interpret these objects in the context of the original problem and determine whether someone can reach Chicago from Boston in a day according to this imaginary train schedule.

3.1 Finger Exercises on Object Containment

Exercise 3.1 Design a data representation for this problem:

> ... Develop a "real estate assistant" program. The "assistant" helps real estate agents locate available houses for clients. The information about a house includes its kind, the number of rooms, its address, and the asking price. An address consists of a street number, a street name, and a city. ...

Represent the following examples using your classes:

1. Ranch, 7 rooms, $375,000, 23 Maple Street, Brookline;

2. Colonial, 9 rooms, $450,000, 5 Joye Road, Newton; and

3. Cape, 6 rooms, $235,000, 83 Winslow Road, Waltham. ∎

Exercise 3.2 Translate the data definition in figure 11 into classes. Also obtain examples of weather information and translate them into instances of the matching class. ∎

WeatherRecord
Date d
TemperatureRange today
TemperatureRange normal
TemperatureRange record

Date
int *day*
int *month*
int *year*

TemperatureRange
int *high*
int *low*
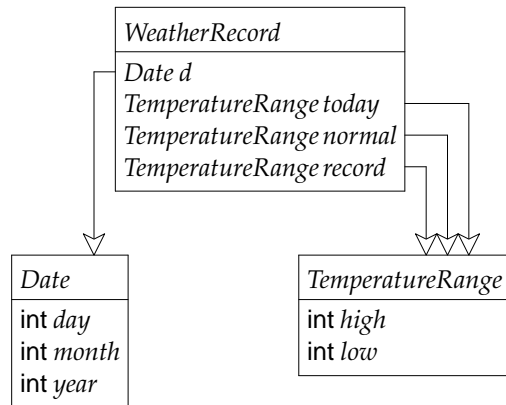
Figure 11: A class diagram for weather records

Exercise 3.3 Revise the data representation for the book store assistant in exercise 2.2 so that the program can find additional information about authors (name, year of birth). Modify the class diagram, the class definition, and the examples. ∎

3.2   Designing Classes that Refer to Classes

This section has introduced problem statements that imply a need for two, or sometimes even several, classes of related information. More precisely, the description of one kind of information in a problem refers to other (non-primitive) information. In that case you must design a class that refers to another class.

As before you first draw a class diagram, and you then indicate with arrows which classes refer to other classes. When you translate the diagram into class definitions, you start with the class(es) from which no arrows originate. We sometimes call such classes *primitive* to indicate that they don't depend on others. The others are called *compound classes* here; you may also encounter the word aggregate in other readings. For now, make sure that the diagrams don't contain loops (aka, cycles).

When the problem involves more than one class, you need to make examples for all classes, not just one. It is best to begin with primitive classes, because making examples for them is easy; the constructors consume basic kinds of values. For compound classes, you can reuse the examples of the

primitive classes when we make up examples. As in the advice for design-
ing basic classes, it is important to transform information into data and to
understand how data represents information. If things look complex, don't
forget the advice from *How to Design Programs* on creating data representa-
tions via an iterative refinement of more and more complex diagrams.

4   Unions of Classes

Our railroad example in the preceding section distinguishes between two
kinds of trains with a boolean field. If the field is true, the instance repre-
sents a local train; otherwise, it is an express train. While a boolean field
may work for a simple distinction like that, it really isn't a good way to
think about distinct kinds of trains. It also doesn't scale to large problems,
like those of real train companies, which offer a wide variety of trains with
many distinct attributes, e.g., city, local, regional, long distance, long dis-
tance express trains, and so on.

   In this section, we show how to use classes to represent distinct yet
related kinds of information, such as kinds of trains. Even though the train
problem would benefit from this reformulation, we use a new problem and
leave the train example to an exercise:

> . . .   Develop a drawing program that deals with three kinds of
> shapes on a Cartesian grid: squares, circles, and dots.
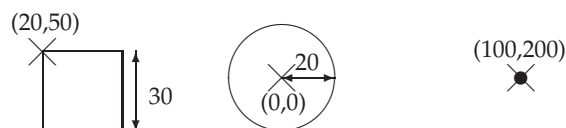
> A square's location is specified via its north-west corner (see ✕)
> and its size. A circle's essential properties are its center point
> (see ✕) and its radius. A dot is drawn as a small disk of a fixed
> size with a radius of 3 pixels. . . .

   The problem says that there are three different kinds of shapes and the
collection of all shapes. In the terminology of *How to Design Programs*, the
collection of shapes is the UNION of three classes of shapes, also called
VARIANTs. More specifically, if we were to develop the shape class in *How
to Design Programs*, we would introduce four data definitions, starting with
one for shapes:

;; A *Shape* is one of:
;; — a *Dot*
;; — a *Square*
;; — a *Circle*

This first definition does not tell us how to create actual shapes. For this, we need the following, concrete definitions:

(**define-struct** *dot* (*loc*))
;; A *Dot* is a structure:
;; — (*make-dot CartPt*)

(**define-struct** *square* (*loc size*))
;; A *Square* is a structure:
;; — (*make-square CartPt Number*)

(**define-struct** *circle* (*loc radius*))
;; A *Circle* is a structure:
;; — (*make-circle CartPt Number*)
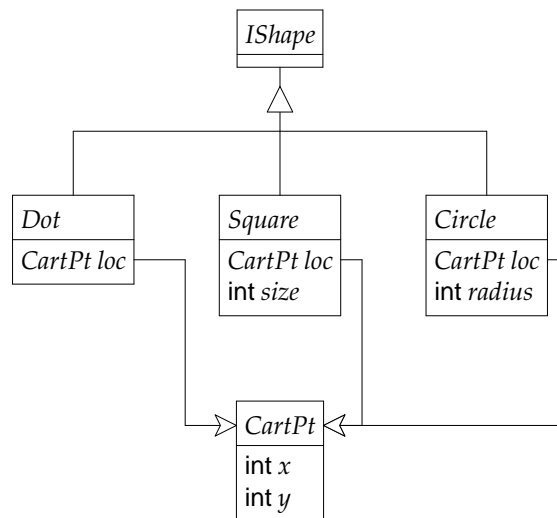
Figure 12: A class diagram for geometric shapes

Since the Scheme data definition consists of four definitions, a Java data definition or class diagram should consist of four boxes. Drawing the boxes

for the last three is easy; for *Shape* we just use an empty box for now. Since the relationship between *Shape*, on one hand, and the *Dot*, *Square*, and *Circle*, on the other, differs from anything we have seen, however, we actually need to introduce two new concepts before we can proceed.

```
// geometric shapes
interface IShape {}
```

```
// a dot shape
class Dot
  implements IShape {
  CartPt loc;
  Dot(CartPt loc) {
    this.loc = loc;
  }
}
```

```
// a square shape
class Square
  implements IShape {
  CartPt loc;
  int size;

  Square(CartPt loc,
         int size) {
    this.loc = loc;
    this.size = size;
  }
}
```

```
// a circle shape
class Circle
  implements IShape {
  CartPt loc;
  int radius;

  Circle(CartPt loc,
         int radius) {
    this.loc = loc;
    this.radius = radius;
  }
}
```

```
// Cartesian points on a computer monitor
class CartPt {
  int x;
  int y;

  CartPt(int x, int y) {
    this.x = x;
    this.y = y;
  }
}
```

Figure 13: Classes for geometric shapes

The first new concept is that of an INTERFACE, which is like a special kind of class. Here *IShape* is such an interface because it is the name for a union of variant classes; it is special because it doesn't contribute any objects to the complete collection. Its sole purpose is to represent the complete collection of objects, regardless of which class they come from, to the rest of the program. The second novelty is the arrow that represents the

relationship between the interface and specific classes. To distinguish this relationship from containment arrows, e.g., between *Square* and *CartPt*, the diagram uses INHERITANCE ARROWS, which are arrows with hollow heads. Figure 12 contains one of these arrows with three endings, one for *Dot*, *Square*, and *Circle*.

In Java, interfaces look almost like classes. Instead of **class**, their definition is introduced with **interface**, and in this chapter, there is never anything between the braces { ... } for an interface. The INHERITANCE relationship between a class and an interface is expressed with the keyword **implements** followed by an interface name:[8]

class *Dot* **implements** *IShape* { ... }

Figure 13 displays the complete set of interface and class definitions for figure 12. Once you accept the two syntactic novelties, translating class diagrams into text is as mechanical as before.

4.1   Types vs Classes

To make the above discussion perfectly clear, we need to take a close look at the notion of TYPE. Thus far, we just said that whatever you use to specify the nature of a field is a type. But what can you use to specify a field's type? In Java, a type is either the name of an interface, a class, or a primitive type (int, double, boolean or *String*). When we write

IShape s

we are saying that *s* has type *IShape*, which means that it is a placeholder for some (yet) unknown shape. Similarly, when we introduce an example such as

IShape s = **new** *Square*(...)

we are still saying that *s* has type *IShape*, even though we know that it stands for an instance of *Square*.

In general, if the program contains "*Ty inst*" or "*Ty inst* = **new** *Cls*(...)"

1. then the variable *inst* has type *Ty*;

2. it stands for an instance of *Cls*; and

3. the "equation" is only correct if *Cls* is *Ty* or if it *Cl* **implements** *Ty*. Otherwise, the statement is a TYPE ERROR, and Java tells you so.

---

[8]Using a leading *I* for the names of interfaces is a convention in this book; it makes it easy to distinguish interfaces from names when you don't have the definition handy.

So a class is a general description of a collection of objects that provides a mechanism for constructing specific objects. An interface is a uniform "face" for several classes, which you sometimes wish to deal with as if it were one. A type, finally, describes for what kind of objects a variable or a parameter (such as the ones in constructors) may stand. Remember: every class is a type, but not every type is a class because interfaces and primitive types are types, too.

Exercise

Exercise 4.1 Translate the three graphical shape examples from the problem statement into objects in the context of figure 13. Conversely, sketch the following instances on a grid: **new** *Dot*(**new** *CartPt*(−3,4)); **new** *Circle*(**new** *CartPt*(12,5),10); and **new** *Square*(**new** *CartPt*(30,−60),20). ∎

Exercise 4.2 Consider this Java rendition of a union:

```
interface ISalesItem {}

class DeepDiscount implements ISalesItem {
  int originalPrice;
  ...
}

class RegularDiscount implements ISalesItem {
  int originalPrice;
  int discountPercentage;
  ...
}
```

Say, in this context, you encounter these examples:

```
ISalesItem s = new DeepDiscount(9900);
ISalesItem t = new RegularDiscount(9900,10);
RegularDiscount u = new RegularDiscount(9900,10);
```

What are the types of *s*, *t*, and *u*? Also, someone has written down the following examples:

```
RegularDiscount v = new DeepDiscount(9900);
DeepDiscount w = new RegularDiscount(9900,10);
RegularDiscount x = new RegularDiscount(9900,10);
```

Which of them are type correct and which one are type errors? ∎
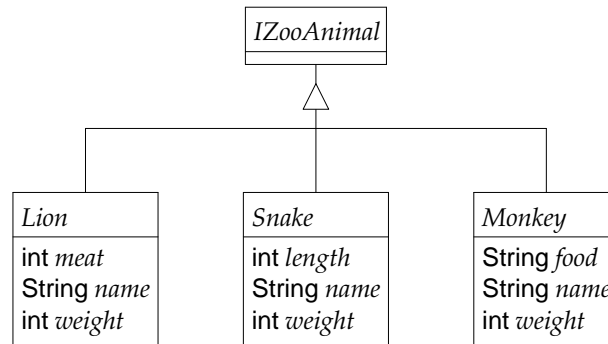
Figure 14: A class diagram for zoo animals

4.2   Finger Exercises on Unions

In *How to Design Programs*, we encountered many definitions of unions. Here is one of them:

> . . .   Develop a program that helps a zoo keeper take care of the animals in the zoo. For now the zoo has lions, snakes, and monkeys. Every animal has a name and weight. The zoo keeper also needs to know how much meat the lion eats per day, the length of each snake, and the favorite food for each monkey. Examples:
>
> 1. Leo weighs 300 pounds and consumes 5 pounds of meat every day.
>
> 2. Ana, the snake, weighs 50 pounds and is 5 feet long.
>
> 3. George is a monkey. He weighs 120 pounds and loves kiwi.
>
> . . .

The problem statement implies that the zoo program deals with an *Animal* class and that there are three distinct classes: *Lion*, *Snake*, and *Monkey*. The three classes have two properties in common: *name*, which is represented with a *String* field, and *weight*, which is an int field (the number of pounds that an animal weighs). Figure 14 shows how we can express these classes and their relationship graphically.

Translating this diagram into class definitions is again straightforward. Each box becomes an interface or a class; each line in the box becomes one field in the class. Each class is annotated with **implements** *IZooAnimal*. In turn, all three constructors consume three values each and their bodies contain three "equations" each. See figure 15 for the full definitions.

```
// zoo animals
interface IZooAnimal{}
```

```
// a zoo lion
class Lion
   implements
   IZooAnimal{
 int meat;
 String name;
 int weight;

 Lion(String name,
       int weight,
       int meat){
  this.name = name;
  this.weight = weight;
  this.meat = meat;
 }
}
```

```
// a zoo snake
class Snake
   implements
   IZooAnimal{
 int length;
 String name;
 int weight;

 Snake(String name,
        int weight,
        int length){
  this.name = name;
  this.weight = weight;
  this.length = length;
 }
}
```

```
// a zoo monkey
class Monkey
   implements
   IZooAnimal{
 String food;
 String name;
 int weight;

 Monkey(String name,
         int weight,
         String food){
  this.name = name;
  this.weight = weight;
  this.food = food;
 }
}
```

Figure 15: Classes for representing zoo animals

Lastly, we should represent the examples from the problem statement with objects:

IZooAnimal leo = **new** *Lion*(`"Leo"`, 300, 5);
IZooAnimal boa = **new** *Snake*(`"Ana"`, 150, 5);
IZooAnimal george = **new** *Monkey*(`"George"`, 150, `"kiwi"`);

All three objects, *leo*, *boa*, and *george*, have type *IZooAnimal*, yet they are instances of three different classes.

Exercises

Exercise 4.3 Modify the representation of trains in figure 10 so that local and express trains are separate classes. ∎
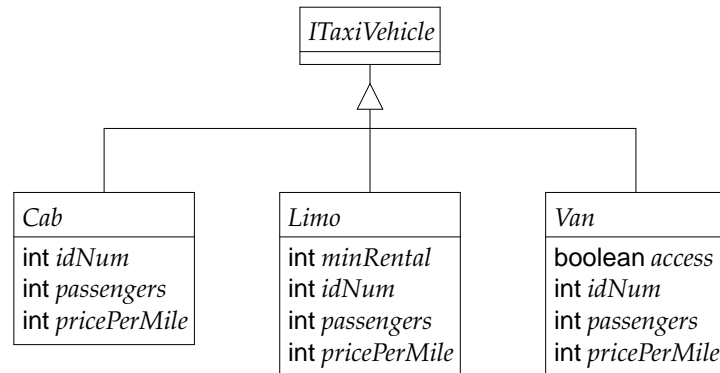
Figure 16: A class diagram for taxis

Exercise 4.4 Design a data representation for this problem:

> ... Develop a "bank account" program. The program keeps track of the balances in a person's bank accounts. Each account has an id number and a customer's name. There are three kinds of accounts: a checking account, a savings account, and a certificate of deposit (CD). Checking account information also includes the minimum balance. Savings account includes the interest rate. A CD specifies the interest rate and the maturity date. Naturally, all three types come with a current balance. ...

 Represent the following examples using your classes:

1. Earl Gray, id# 1729, has $1,250 in a checking account with minimum balance of $500;

2. Ima Flatt, id# 4104, has $10,123 in a certificate of deposit whose interest rate is 4% and whose maturity date is June 1, 2005;

3. Annie Proulx, id# 2992, has $800 in a savings account; the account yields interest at the rate of 3.5%. ∎

Exercise 4.5 Consider this generalization of exercise 2.3:

> ... Develop a program that creates a gallery from three different kinds of media: images (gif), texts (txt), and sounds (mp3).

All have names for source files and sizes (number of bytes). Images also include information about the height, the width, and the quality of the image. Texts specify the number of lines needed for visual representation. Sounds include information about the playing time of the recording, given in seconds. ...

Develop a data representation for these media. Then represent these three examples with objects:

1. an image, stored in `flower.gif`; size: 57,234 bytes; width: 100 pixels; height: 50 pixels; quality: medium;

2. a text, stored in `welcome.txt`; size: 5,312 bytes; 830 lines;

3. a music piece, stored in `theme.mp3`; size: 40,960 bytes, playing time 3 minutes and 20 seconds. ▪

Exercise 4.6 Take a look at the class diagram in figure 16. Translate it into interface and class definitions. Also create instances of each class. ▪

Exercise 4.7 Draw a diagram for the classes in figure 17 (by hand). ▪

4.3 Designing Unions of Classes

When a collection of information consists of n disjoint collections, the best way to represent it is via an interface with n implementing classes. The easiest way to recognize this situation is to study examples of information. If you have several pieces of information that ought to belong to one and the same collection but have different kinds of attributes or consist of different kinds of pieces, it is time to represent the information as the union of distinct classes.

To design a union of classes, we proceed as before. We draw a diagram with one interface and as many implementing classes as there are distinct kinds of objects. The interface represents the collection of information in its entirety; the implementing classes represent the distinct variants (or subsets) of information. To indicate the relationship between the classes-as-boxes in this diagram, we use the refinement arrow. Since this kind of diagram resembles a tree, we speak of class hierarchies.

Naturally, some of the classes involved in a union may refer to other classes. Indeed, in our very first example *IShape* referred to a *CartPt*, because we needed to represent the location of a shape. When this happens

```
                            // tickets to a science museum
                            interface ITicket {}
```

```
// museum tickets          // omnimax admission        // laser admission
class Admission            class OmniMax               class LaserShow
  implements ITicket {       implements ITicket {        implements ITicket {
  Date d;                    Date d;                     Date d;
  int price;                 int price;                  int price;
                            ClockTime t;                ClockTime t;
                            String title;               String row;
  Admission(Date d,                                      int seat;
            int price) {    OmniMax(Date d,
    this.d = d;                     int price,          LaserShow(Date d,
    this.price = price;             ClockTime t,                int price,
  }                                 String title) {             ClockTime t,
}                             this.d = d;                       String row,
                              this.price = price;               int seat) {
                              this.t = t;                 this.d = d;
                              this.title = title;         this.price = price;
for ClockTime,                }                           this.t = t;
see figure 10             }                               this.row = row;
                                                          this.seat = seat;
                                                        }
                                                      }
```

Figure 17: Some classes

and things begin to look complicated, it is important to focus on the design of the union diagram first and to *add* the containment portion later, following the suggestions in section 3.2.

After we have a complete class diagram, we translate the class diagram into classes and interfaces. The box for the interface becomes an **interface** and the others become **class**es where each class **implements** the interface. Equip each class with a purpose statement; later we may write a paragraph on the union, too.

Finally, we need to make up examples. While we cannot instantiate the interface directly, we use it to provide a type for all the examples. The latter are created from each of the implementing classes. For those who may take over the program from us in the future, we should also explain in a comment what the objects mean in the real world.

5   Unions, Self-References and Mutual References

Thus far, all the programming problems involved objects that consist of a fixed number of objects. They dealt with one receipt for a coffee sale; with one entry in a runner's log, which contains one date; or one train, which contains one schedule, one route, and two dates. They never dealt with an object that contains—or consists of—an unknown and arbitrary number of objects. We know, however, that real-world problems deal with such cases. For example, a runner's log doesn't consist of a single entry; it is an ever-growing list of entries, and the runner may become interested in all kinds of aspects of this list. Similarly, real train schedules for a real train company aren't about one train, but many trains at the same time.

In this section, we study several ways of representing such collections of information as one piece of data. In *How to Design Programs*, we use the terminology "arbitrarily large data" and studied those in two stages: lists and complex generalizations. The two subsections here correspond to those two stages.

5.1   Containment in Unions, Part 1
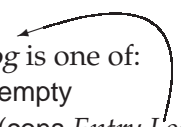
Recall the problem concerning a runner's log:

> . . .   Develop a program that manages a runner's training log. Every day the runner enters one entry concerning the day's run. Each entry includes the day's date, the distance of the day's run, the duration of the run, and a comment describing the runner's post-run disposition. . . . . . .

Naturally the program shouldn't just deal with a single log entry but sequences of log entries. After collecting such entries for several seasons, a runner may, for example, wish to compute the mileage for a month or the pace of a daily workout.

We already know how to represent individual log entries and dates. What we need to figure out is how to deal with an entire list of entries. According to *How to Design Programs*, we would use a list to represent a complete runner's log:

```
;; A Log is one of:
;; — a empty
;; — a (cons Entry Log)
```

assuming *Entry* is defined. Using this data definition, it is easy to represent arbitrarily large lists of data.

The data definition says that *Log* is a collection of data that consists of two distinct sub-classes: the class of the empty list and the class of consed lists. If we wish to represent this information with classes, we clearly need to define a union of two variant classes:

1. *ILog*, which is the type of all logs;

2. *MTLog*, which represents an empty log; and

3. *ConsLog*, which represents the construction of a new log from an entry and an existing log.

From Scheme, we know that an *MTLog* doesn't contain any other information, so it doesn't contain any fields. A *ConsLog*, though, consists of two values: an entry and another list of log entries; therefore the *ConsLog* class requires two field definitions: one for the first *Entry* and one for the rest.

← check   can we make the backarrow look different for this one picture?
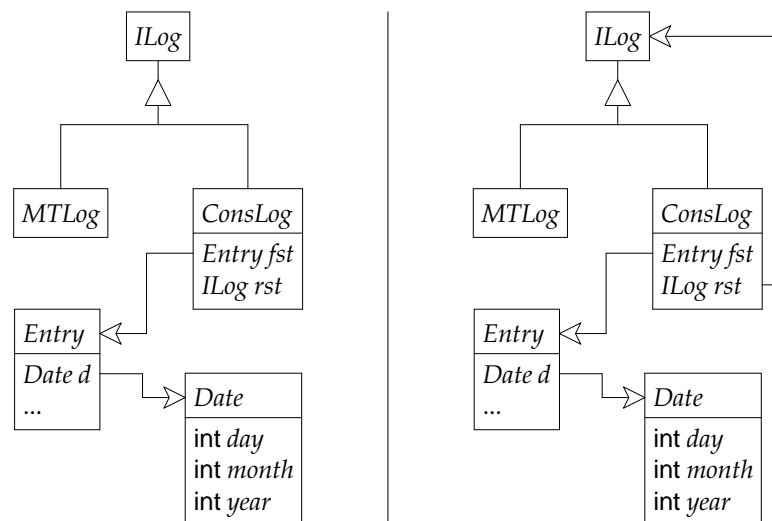
Figure 18: A class diagram for a complete runner's log

The class diagram for the union appears on the left in figure 18. Both *MTLog* and *ConsLog* refine *ILog*. The only class that contains fields is *ConsLog*. One field is *fst* of type *Entry*; a containment arrow therefore connects

the first field in *ConsLog* and *Entry*. The other field is *rst*; its type is *ILog*. This left class diagram misses the arrow from *rst* to its class in this diagram, which the informal Scheme definition above contains. Drawing it, we obtain the diagram in the right column in figure 18. The result is a diagram with a loop or cycle of arrows. Specifically, the diagram defines *ILog* as the union of two variant classes with a reference from one of the two classes back to *ILog*. We have created a pictorial data definition that is just like the corresponding data definitions for lists that we know from *How to Design Programs*.

Even if it weren't a part of the design recipe for classes, we know from our prior experience with *How to Design Programs* that we need examples for self-referential data definitions. Otherwise, we never know whether they make sense. Before we can make up examples of logs, however, we need to translate the data definition (diagram) into a class hierarchy. Fortunately, it suffices to apply what we already know, because ==arrows merely emphasize the type of a field==, which is always a part of the class definition anyway. See figure 19 for the result.

```
// a runner's log
interface ILog {}
```

```
// the empty log
class MTLog implements ILog {
  MTLog() {}
}
```

```
// adding an entry to a log
class ConsLog implements ILog {
  Entry fst;
  ILog rst;

  ConsLog(Entry fst, ILog rst) {
    this.fst = fst;
    this.rst = rst;
  }
}
```

```
// an individual entry
class Entry { ... }
```
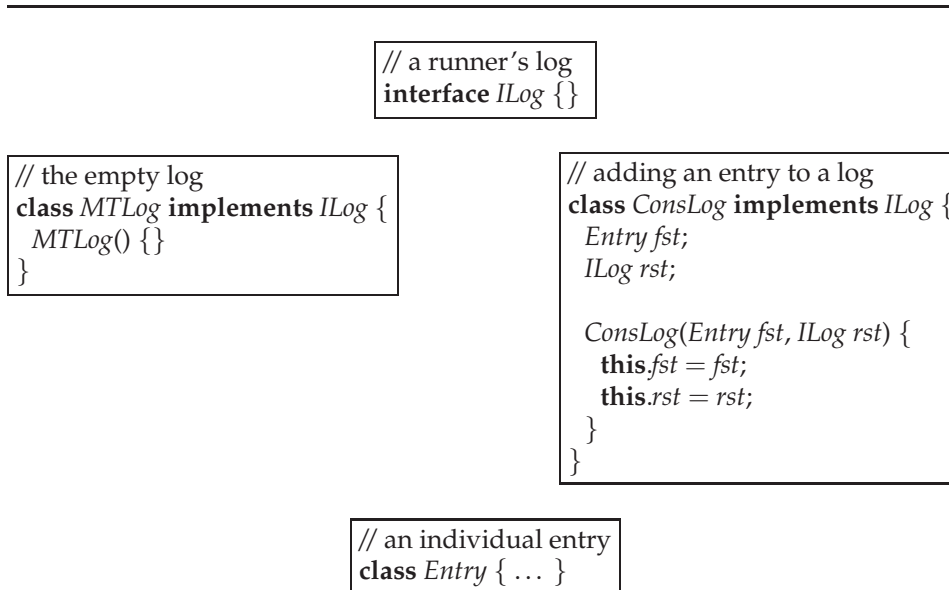
Figure 19: Classes for a runner's log

Now we can look at an actual log (information) and attempt to translate it into objects. Recall our sample log:

on June 5, 2003	5.3 miles	27 minutes	feeling good
on June 6, 2003	2.8 miles	24 minutes	feeling tired
on June 23, 2003	26.2 miles	150 minutes	feeling exhausted
...	...	...	...

We had already translated these three entries into objects:

Date d1 = **new** *Date*(5, 6, 2003);
Date d2 = **new** *Date*(6, 6, 2003);
Date d3 = **new** *Date*(23, 6, 2003);

Entry e1 = **new** *Entry*(d1, 5.3, 27, `"Good"`);
Entry e2 = **new** *Entry*(d2, 2.8, 24, `"Tired"`);
Entry e3 = **new** *Entry*(d3, 26.2, 150, `"Exhausted"`);

The last step is to connect the three log entries in a single *Log*:

ILog l1 = **new** *MTLog*();
ILog l2 = **new** *ConsLog*(e1,l1);
ILog l3 = **new** *ConsLog*(e2,l2);
ILog l4 = **new** *ConsLog*(e3,l3);

Each of these examples represents a concrete log in the runner's world. The first one, *l1*, represents the empty log, before the runner has completed the first entry. The last one, *l4*, represents the series of all three entries: *e3*, *e2*, and *e1*.
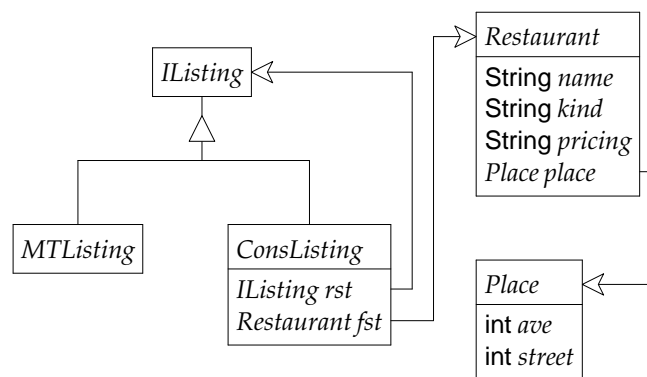
Figure 20: A class diagram for listing restaurants

With the creation of these examples, we have just completed the standard design recipe for classes. We have also verified that a circular diagram describes a perfectly fine way of representing information as data. Naturally it's never clear whether such a definition does what we expected it to do, so you should always experiment with several examples.

Exercises

Exercise 5.1 Translate these two objects of type *ILog*

> *ILog l5* = **new** *ConsLog*(*e3,l1*);
> *ILog l6* = **new** *ConsLog*(*e3,l2*);

into the runner's world of logs. Assume these examples were constructed in the context of the four examples above. ∎

Exercise 5.2 Represent the following runner's log as objects:

1. on June 15, 2004: 15.3 miles in 87 minutes, feeling great;

2. on June 16, 2004: 12.8 miles in 84 minutes, feeling good;

3. on June 23, 2004: 26.2 miles in 250 minutes, feeling dead;

4. on June 28, 2004: 26.2 miles in 150 minutes, good recovery.

∎

For a second example in the same vein, let's resume our discussion of the program that assists a visitor with restaurant selection in Manhattan:

> ... Develop a program that helps visitors navigate Manhattan's restaurant scene. The program must provide four pieces of information per restaurant: its name, the kind of food it serves, its price range, and the closest intersection (street/avenue). ...

Clearly, this program should deal with lists of restaurants, because a visitor may, for example, wish to learn about all German restaurants in a certain area or all Thai restaurants in a certain price range.

If we were designing a representation in Scheme, we would again use a cons-based representation of restaurant listings:

> ;; *A List of Restaurants (ILoR) is* is one of:
> ;; — a empty
> ;; — a (cons *Restaurant ILoR*)

assuming the class of *Restaurant*s is already defined.

To express this data definition with a class diagram, we follow the reasoning in the preceding example. A restaurant listing is a union of two classes: those for empty listings and those for constructed listings. The latter consist of two pieces of data: the (first) restaurant and the rest of the listing. The class diagram in figure 20 shows how all this works.

Figure 21 sketches the class definitions that correspond to the diagram. As we have already seen, the cycle in the diagram doesn't affect the definitions at all. It just shows up in the class *ConsListing*, which both implements *ILoR* and contains a field of type *ILoR*.

```
// a list of restaurants
interface ILoR { }
```

```
// the empty list
class MTListing implements ILoR {
  MTListing(){ };
}
```

```
// adding a restaurant to a list
class ConsListing implements ILoR {
  Restaurant fst;
  ILoR rst;

  ConsListing(Restaurant fst, ILoR rst) {
    this.fst = fst;
    this.rst = rst;
  }
}
```

```
// an individual restaurant
class Restaurant { … }
```
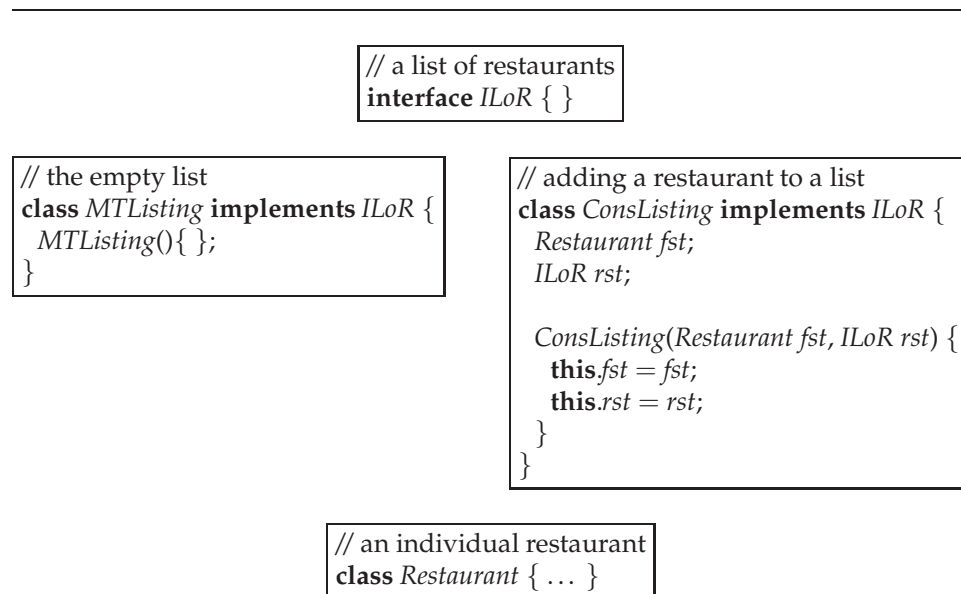
Figure 21: Classes for representing restaurant listings

Let's use the following restaurants to build a restaurant listing:

```
Restaurant ex1 =
  new Restaurant("Chez Nous", "French", "exp.", new Place(7, 65));
Restaurant ex2 =
  new Restaurant("Das Bier", "German", "cheap", new Place(2, 86));
Restaurant ex3 =
  new Restaurant("Sun", "Chinese", "cheap", new Place(10, 113));
```

Next the listings *l1*, *l2*, and *l3* contain French, German, and Chinese restaurants, respectively; the last listing contains all restaurants:

ILoR mt = **new** *MTListing*();
ILoR l1 = **new** *ConsListing*(*ex1,mt*);
ILoR l2 = **new** *ConsListing*(*ex2,mt*);
ILoR l3 = **new** *ConsListing*(*ex3,mt*);
ILoR all =
  new *ConsListing*(*ex1,***new** *ConsListing*(*ex2,***new** *ConsListing*(*ex3,mt*)));

Exercises

Exercise 5.3 Consider a revision of the problem in exercise 3.1:

> . . . Develop a program that assists real estate agents. The program deals with listings of available houses. . . . . . .

Make examples of listings. Develop a data definition for listings of houses. Implement the definition with classes. Translate the examples into objects. ∎
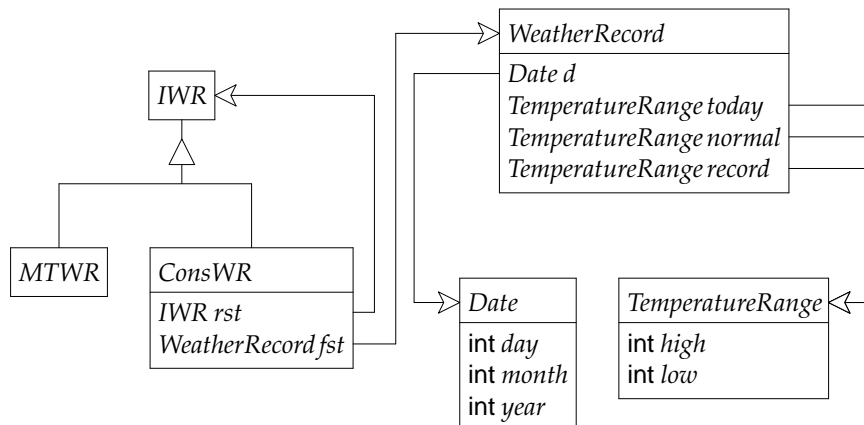
Figure 22: A class diagram for weather reports

Exercise 5.4 Consider a revision of the problem in exercise 2.2:

> ...   Design a program that assists a bookstore manager with
> reading lists for local schools. ......

Develop a class diagram for a list of books (by hand). Translate the diagram
into classes and interfaces. Create two lists of books that contain at least one
of the books in exercise 2.2 plus one or more of your favorite books. ∎

Exercise 5.5 Take a look at figure 22, which contains the data definition
for weather reports. A weather report is a sequence of weather records
(see exercise 3.2). Translate the diagram into classes and interfaces. Also
represent two (made-up) weather reports, one for your home town and
one for your college town, as objects. ∎

5.2   Containment in Unions, Part 2

Lists are by no means the only form of information that requires a class
diagram with cycles for an accurate descriptions. Let's take another look at
the problem of drawing shapes (page 27):

> ...   Develop a drawing program that deals with at least three
> kinds of shapes: dots, squares, and circles. ...In addition, the
> program should also deal with overlapping shapes. In the fol-
> lowing figure, for example, we have superimposed a circle on
> the right side of a square:
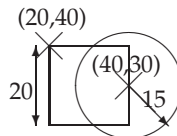
> We could now also superimpose this compounded shape on an-
> other shape and so on. ...

The new element in this problem statement is the goal of combining
two shapes into one. This suggests a new class that refines *IShape* from
figure 12. The purpose of the new class is to represent the combination of
two shapes. We call it *SuperImp* for that reason.

Figure 23 contains the class diagram for our revised problem. Like the
diagrams for lists, this diagram contains a cycle, specifying a self-referential
data definition; but unlike the list diagrams, this one has *two* arrows that go
from an implemented class back to the interface. As before, this difference
doesn't pose any problems for the translation into class definitions: see
figure 24.
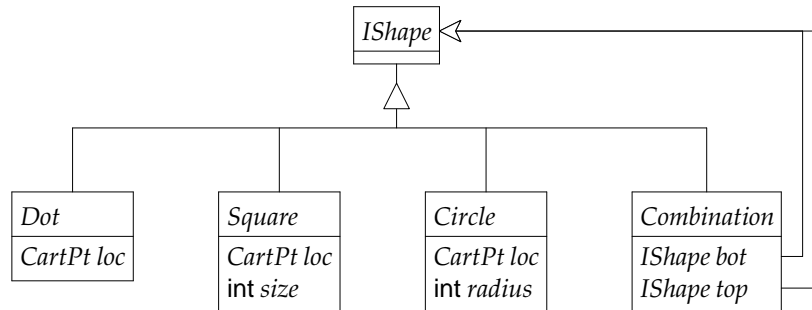
Figure 23: A class diagram for combination shapes
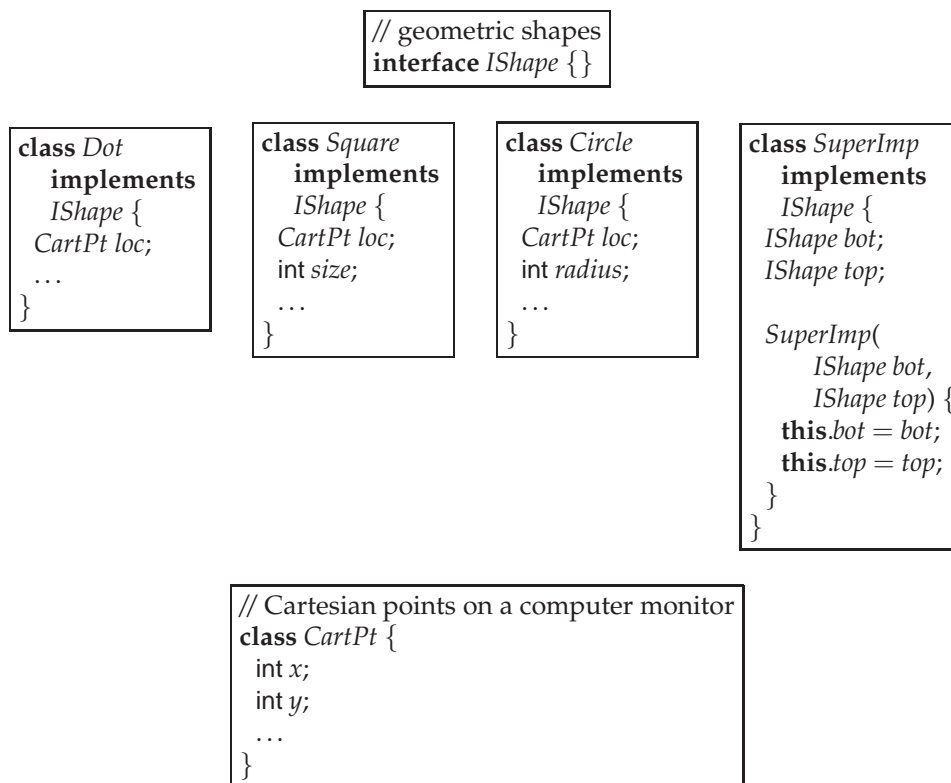
Figure 24: Classes for combination shapes

Here is the object that represents the shape from the problem statement:

new *SuperImp*(**new** *Square*(**new** *CartPt*(20,40),20),
                                    new *Circle*(**new** *CartPt*(40,30),15))

The *SuperImp* object combines a square and a circle, placing the circle on top of the square. Now look at these definitions:

CartPt cp1 = **new** *CartPt*(100, 200);
CartPt cp2 = **new** *CartPt*(20, 50);
CartPt cp3 = **new** *CartPt*(0, 0);

IShape s1 = **new** *Square*(cp1, 40);
IShape s2 = **new** *Square*(cp2, 30);
IShape c1 = **new** *Circle*(cp3, 20);

IShape sh1 = **new** *SuperImp*(c1, s1);
IShape sh1 = **new** *SuperImp*(s2, **new** *Square*(cp1, 300));
IShape sh3 = **new** *SuperImp*(s1, sh2);

To understand the purpose of these classes and the meaning of these specific objects, interpret *sh1*, *sh2*, and *sh3* as figures on a grid.

The need for self-referential data definitions, like those of reading lists and restaurant listings, also comes about naturally for data such as family trees and river systems:

```
          t
   s     /
    \    tributary [2]
  main [3] /
      \ /
       \/
       b        u
        \       /
      main [3]  tributary [1]
          \   /
           \ /
            a
            |
         main [4]
            |
            m
```
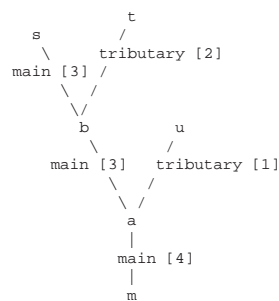
... The environmental protection agency monitors the water quality of river systems. A river system consists of a river, its tributaries, the tributaries of the tributaries, and so on. The place where a tributary flows into a river is called a confluence. The river's end—the segment that ends in the sea or perhaps another river— is called its mouth; the initial river segment is its source. ...

Even a cursory look confirms that this is by far the most complex form of information that we have encountered so far. When we are confronted with something like that, it is best to make up at least one small example and to study it in depth, which is why the problem comes with an artificial map of a made-up river system. The example has three sources—s, t, and u—and two confluences—at b and a. The mouth of the entire system is at

`m`. The map depicts the system as if it had one `main` river, with two direct tributaries. Each segment is labeled with a number, which represents its name.
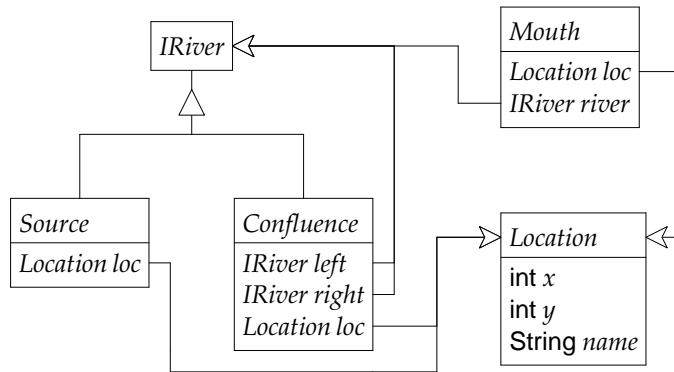
Figure 25: A class diagram for river systems

If we were to take a trip up the river, we would first encounter the mouth `m` of the river and the river proper. Here, the river proper comes from a confluence of two rivers, namely `a`; in general, though, the river may come straight from a source, too. Moving along `main` from `a`, we see again that it comes from a confluence; this time the location is `b`. Of course, it, too, could have come from a source.

Thus, our problem analysis suggests several classes of information or data. First, the mouth of a river system is a combination of two things: a location and a river (proper). A river is one of two things: a confluence of two rivers or a source (segment). Finally, a confluence consists of a location and two rivers. That is, the description for a confluence refers to river, which in turn is related to confluence.

Figure 25 depicts the class diagram that captures all the elements of our problem analysis. The *Mouth* class represents the entire river system. It refers to the *IRiver* interface, which is the union of two concrete classes: *Source* and *Confluence*. Like the mouth of a river system, these two classes have a *Location* field; but *Confluence* also describes which two rivers are flowing together at this location. The class at the bottom right describes a location as a point on a grid with a name.

```
// the end of a river                    // a location on a river
class Mouth{                             class Location{
  Location loc;                            int x;
  IRiver river;                            int y;
                                           String name;
  Mouth(Location loc, IRiver river){
    this.loc = loc;                        Location(int x, int y, String name){
    this.river = river;                      this.x = x;
  }                                          this.y = y;
}                                            this.name = name;
                                           }
                                         }
```

```
// a river system
interface IRiver{ }
```

```
// the source of a river                 // a confluence of two rivers
class Source implements IRiver {         class Confluence implements IRiver{
  Location loc;                            Location loc;
                                           IRiver left;
  Source(Location loc){                    IRiver right;
    this.loc = loc;
  }                                        Confluence(Location loc,
}                                                     IRiver left,
                                                      IRiver right){
                                             this.loc = loc;
                                             this.left = left;
                                             this.right = right;
                                           }
                                         }
```

Figure 26: Classes for representing river systems

A translation of figure 25 into class and interface definitions appears in figure 26. As with any complex data definition, this one also needs validation via a transliteration of information into data. Figure 27 shows a class that defines a series of data examples, which represent the information from the problem statement (page 46), including *mth*, the entire river system.

```
class RiverSystemExample {
  Location lm = new Location(7, 5, "m");
  Location la = new Location(5, 5, "a");
  Location lb = new Location(3, 3, "b");
  Location ls = new Location(1, 1, "s");
  Location lt = new Location(1, 5, "t");
  Location lu = new Location(3, 7, "u");

  IRiver s = new Source(ls);
  IRiver t = new Source(lt);
  IRiver u = new Source(lu);

  IRiver b = new Confluence(lb,s,t);
  IRiver a = new Confluence(la,cb,u);

  Mouth mth = new Mouth(lm,ca);

  RiverSystemExample() { }
}
```

Figure 27: A sample river system

5.3 Finger Exercises on Containment in Unions

Exercise 5.6 Consider the following problem:

```
                                Bob
???    ??? ???    ???    ???   1917
 |      |   |      |      |      |
+--+--+   +--+--+    +--+--+
   |          |          |
Angela    Robert      Annie
 1936      1935        1938        ???
   |          |          |          |
   +----+----+          +----+----+
        |                    |
      Janet                Paul
      1958                 1956
        |                    |
        +----------+----------+
                   |
                 Peter
                 1980
```
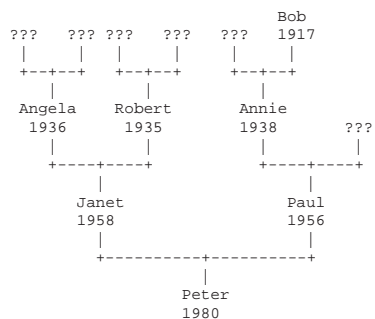
... Develop a program that helps with recording a person's ancestor tree. Specifically, for each person we wish to remember the person's name and year of birth, in addition to the ancestry on the father's and the mother's side, if it is available. The tree on the left is an example; the nodes with "???" indicate where the genealogist couldn't find any information. ...

Develop the class diagram (by hand) and the class definitions to represent ancestor family trees. Then translate the sample tree into an object. Also draw your family's ancestor tree as far as known and represent it as an object. Hint: Represent "???" with a class called *Unknown*. ∎

Exercise 5.7 Research the tributaries of your favorite river. Create a data representation of the river and its tributaries. Draw the river system as a schematic diagram. ▪

Exercise 5.8 Modify the classes that represent river segments, mouths, and sources so that you can add the names of these pieces to your data representation. Can you think of a river system that needs names for all three segments involved in a confluence? Represent such a confluence with the revised classes. ▪
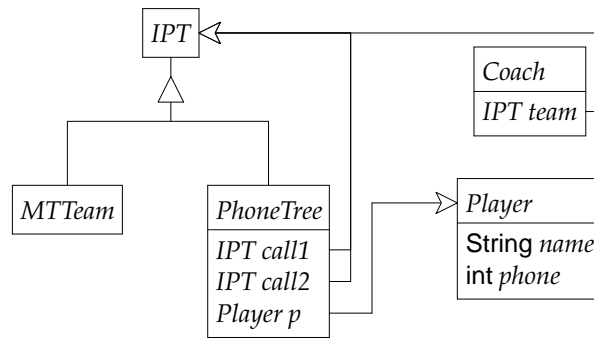
Figure 28: A class diagram for a phone tree

Exercise 5.9 Soccer leagues arrange its soccer teams into phone trees so that they can quickly inform all parents about rain-outs. The league calls the coach, who in turn calls the parents of the team captain. Each parent then calls at most two other parents.

The class diagram in figure 28 contains the data definition for a program that manages phone trees. Given these classes, one could create the data in figure 29. Draw the phone tree there as a circle-and-arrow diagram. Each circle corresponds to a player or coach. An arrow means that a player calls some other player; it goes from the caller to the callee. Then develop the class definitions that correspond to the given data definition. ▪

6   Designing Class Hierarchies

In the preceding sections we have discussed more and more complex examples of class hierarchies. Designing a class hierarchy is the first and the

Player
  cpt = **new** *Player*(`"Bob"`, 5432345);
Player
  p1 = **new** *Player*(`"Jan"`, 5432356);
Player
  p2 = **new** *Player*(`"Kerry"`, 5435421);
Player
  p3 = **new** *Player*(`"Ryan"`, 5436571);
Player
  p4 = **new** *Player*(`"Erin"`, 5437762);
Player
  p5 = **new** *Player*(`"Pat"`, 5437789);

IPT mt = **new** *MTTeam*();

IPT pt =
  new *PhoneTree*(
    cpt,
    new *PhoneTree*(
      p1,
      new *PhoneTree*(*p2*,*mt*,*mt*),
      new *PhoneTree*(*p3*,*mt*,*mt*)),
    new *PhoneTree*(
      p4,
      new *PhoneTree*(*p5*,*mt*,*mt*),
      mt));

Coach ch = **new** *Coach*(*pt*);

Figure 29: A phone tree for a soccer team

most important step in the design of an object-oriented program. Getting it right often avoids big hassles later when you (or your successor) need to improve or extend or revise the program. To help with this process, we offer a reasonably general design recipe here.

The purpose of a class is to represent collections of related pieces of information from our domain of interest. Recall from *How to Design Programs* that a piece of information is a statement about the problem world, also known as the DOMAIN OF INTEREST. Once we have the information represented as data, our programs can compute new data and we can interpret this data as information. One way to think about this process is that the original data represents a *problem* and that the newly computed data represents a *solution* to the problem. Since the point of involving a computer is to solve many problems, not just a single problem, it is natural to speak of classes of information and thus classes of data.

To discover a good data representation, we recommend that you proceed in four steps:

problem analysis The problem statement is all you have. You must therefore analyze the problem statement closely and determine with what kind of information your program must cope. It is important to identify classes of information, not specific pieces of information at this stage. The *outcome* of this step is a list of names for the relevant classes of information and a brief description of each class.

People who solve problems need examples. You should therefore supplement the description of classes of information with examples. This is particularly important when the problem statement is complex and involves many distinct forms of information.

class diagrams (data definition) The key step is to translate the informal class descriptions into a rigorous class diagram that exposes all the relationships between classes and interfaces. The diagram consists of boxes and two kinds of arrows that connect the boxes. Each box names an interface or a class of data; a class box also describes the properties that all instances of this class share. When a property of a class refers to some other class, you indicate this with a containment arrow, which indicates that an instance of this class contains an instance of the other class. When a class is a part of a union of classes, you add an refinement arrow from the class to the interface.

For simple problems, like those in this chapter, coming up with a diagram is a matter of practice. It requires recognizing five situations and translating them properly into boxes and arrows.

1. Use a *primitive* type if there is an obvious correspondence between the information and an atomic type that the language always supports.

   Examples: numeric measurements, names, on/off switches

2. Introduce a *new class* if you need to represent some information that consists of several other pieces of information. This is typically the case with things that have several properties. Each property becomes a field in the class.

   Examples: positions in space, addresses

3. Have a class *A refer to another class B* if some component of a piece of information is itself the composition of several pieces of information. When you create an instance of class *A*, it contains an instance of class *B*.

   Examples: a log entry contains a date; a train refers to a schedule

4. Use a *union* of classes if a collection of information consists of several distinct subclasses. The union represents the entire collection of objects to the rest of the program (as a type).

   Examples: local and express trains, geometric shapes

5. The class diagram is *self-referential* if you need to represent pieces

of information that consists of an unknown and unlimited number of other pieces.

Examples: reading lists, family trees, river systems, file folders

Warning: The example of river systems shows the importance of distinguishing between the boundary of a self-referential union and the union itself. We decided that a river system has a single mouth and many confluences and sources. Hence, the *Mouth* class referred to *IRiver* but it wasn't a part of the union itself. Getting this distinction right takes practice; indeed, sometimes both solutions work and only further exploration reveals whether one is better than the other.

As problems become more and more complex, you shouldn't expect that your first data representation is the best possible representation. Instead, you should test the representation with some simple explorative programming. These tests often suggest small refinements and occasionally radical revisions. In short, you must expect to use the process of iterative refinement and revision from *How to Design Programs* to get the data representation right.

class definitions, purpose statements Obtaining classes from the class diagram is an almost mechanical affair. Each box is translated according to a standard recipe. Containment arrows correspond to the type specifications of fields; refinement arrows become **implements** specifications; and all boxes, except for those of interfaces, are equipped with a constructor. You must add a one-line purpose statement to each class that explains which information the class represents or a short paragraph that explains the entire collection of classes. As representations become complex, these explanations should provide detailed explanations for the move from information to data (and back).

Only one part in his step needs your full attention: whether a field should have the same value for *all* instances of this class. If so, use an initialization equation with the field; otherwise, add the field as a parameter to the constructor and add an equation of the shape **this**. *field = field* to the constructor.

examples (representation and interpretation) After you have formulated the classes, you must translate the informal examples from the second step into objects. Doing so validates that you have built a class hierarchy that can represent the relevant information.

Furthermore, you should make up examples of objects and interpret
them in the problem domain. This step ensures that you understand
the results that you get back from the computations. After all, the
computation produces either a primitive type or an instance of one of
your classes.

Warning: In general, a data representation is too liberal. That is, there
are often objects without any meaning in the problem domain. For
example, you could create a *Date* object (page 20) such as this

new *Date*(45, 77, 2003)

and attempt to interpret it in the real world. Of course, this object
does not correspond to a date in your world; there is no 77$^{\text{th}}$ month
and no month has 45 days. Warn people of such problems, if needed.

Many problem statements suggest well-known data representations: fixed
hierarchies of objects, lists, trees, and so on. For those cases, the design
recipe often explains why you want to use a certain data representation. In
other cases, however, you will find an unusual form of information, and
then you should follow the design recipe as much as possible.

6.1   Exercises

Exercise 6.1 Design the data representation for a program that assists with
shipping packages for a commercial shipper. For each package the pro-
gram needs to record the box size, its weight, information about the sender
and the recipient, and a URL for the customer so that the package can be
tracked. Hint: Use a *String* to represent a URL. ∎

Exercise 6.2 Revise the data representation for a program that assists vis-
itors in Manhattan (see page 21). Assume that a visitor is interested in
restaurants, museums, and shops. We have already studied what the pro-
gram needs to represent about restaurants. Concerning museums, visitors
typically want to know the name of the museum, the price of admission,
and its hours. For shops, they also want to see its hours (assume the same
hours for every day), but also what kind of items they sell. Of course, visi-
tors also need to find the restaurants, museums, and shops; that is, the data
representation needs to contain locations. ∎

Exercise 6.3 Design the data representation for a program that manages
the schedule for one route of a train. A schedule records the departure

station and time; the destination station and estimated arrival time; and all the stops in between. For now, identify a stop on the route with its name. ∎

Exercise 6.4 Design the data representation for a program that assists a real-estate agent. A real estate agent sells several different kinds of properties: single family houses (see problem 3.1), condominiums, and town houses. A typical customer needs to know the address of the property, the living area (in square feet), and the asking price (in dollars). For a single family house, the customer also wants to know the land area and number of rooms. For a condominium, the customer wants to know the number of rooms and whether it is accessible without climbing stairs. For a town house, the client is often interested in how much garden area town houses have. ∎

Exercise 6.5 Design the data representation for a program that creates graphical user interfaces (GUIs). The basic component of a GUI is one of the following:

- BooleanView

- TextFieldView

- OptionsView

- ColorView

Each of these kinds of components contains a label and some additional data, which for now doesn't play a role.

To arrange basic GUI components in a grid, GUI software systems provide some form of table. A table consists of several rows; each row is a series of GUI components. Naturally, tables can be nested to allow for complex layouts; that is, they must also be GUI components (though without label). ∎

Exercise 6.6 Design the data representation for a program that tracks library checkouts. The library has books, CDs, and DVDs. Each item has a catalog number and a title. For each book the librarian also needs to record the name of the author, and the year the book was published. For CDs, the librarian also records the artist, and the number of tracks. For DVDs, the record indicates the kind of DVD (comedy, drama) and the length of play. ∎

Exercise 6.7 Design the data representation for a company's organizational chart. Each employee is identified by an id number, a name, and a title. If an employee supervises some other employees, the representation of this employee in the chart also points to these other employees. ∎

Exercise 6.8 A player in a board game keeps a wallet of pebbles. There are five kinds of pebbles, each represents a unique color. The player's wallet consists of an arbitrary number of pebbles. Design the data representation for a player's wallet.

The game administrator keeps track of a deck of cards. Each card is identified by five pebbles of arbitrary color and exactly one of the following three designation: normal, high, super. Design the data representation for decks of cards. ∎

Exercise 6.9 Consider the following puzzle:

> ... A number of people want to cross a dilapidated bridge at night. Each person requires a different amount of time to cross the bridge. Together they have one battery-powered flashlight. Only two people can be on the bridge at any given time, due to its bad state of repair.
>
> Given the composition of the group and the life-time of the battery, the problem is to determine whether and how the entire group can cross the bridge. ...

Solving the puzzle (manually or with a program) requires a recording of what happens as members of the group moves back and forth across the river. Let's call the status of the "puzzle world" after each individual crossing a *state*.

Design a data representation for the states of the puzzle. Which elements does a state have to record? What is the initial state for this problem? What is the final state for this problem? Express them in your representation. ∎

6.2   Case Study: Fighting UFOs

Let's study the design recipe in the context of a reasonably interesting and complex problem. Imagine you have become a game developer and your manager poses the problem of developing the first version of a game:[9]

---

[9]Your manager imagines a game freely named after H. G. Wells's science fiction novel.

... Develop a "War of the Worlds" game. A UFO descends from the sky. The player owns an anti-UFO platform (AUP), which can move left and right on the ground. It can also fire shots, straight up from the middle of the platform. If any of the shots hit the UFO, the player wins. Otherwise, the UFO lands and destroys the earth.

For good measure, your manager has asked an artist to draw a mock-up of the game world: see the picture on the left. It displays a screen shot of a single scene in this game, though your manager understands that you want considerable artistic freedom for your work. ...

lighter background   ← check

Recall that your first step is to read the problem statement until you understand what kind of information the program must represent as data. As you can see in this problem statement and the accompanying screen shot, this world of UFOs obviously contains three kinds of physical objects: UFOs, which are drawn as green flying saucers; AUPs, which appear as a red flat rectangle with a second rectangle sticking out in the middle; and shots, which look like long yellow stripes. Of course, while one specific world contains just one UFO and one AUP, it will almost always going to contain several shots. Indeed, since shots presumably appear when the player hits a button and disappear when they reach the edge of the window, the number of shots keeps growing and shrinking. In order to represent the entirety of shots and include it in the world, you need a flexible compound data object; a list of shots is the natural choice.

Now is the time to turn all these first notes into an enumeration of descriptions, adding some first imaginary details concerning the properties of the various objects, including color, size, movement, etc. For each item on the list, you should also make up an informal example:

1. The most important object is the world itself. It contains all the other objects. Including the world itself as an object gives us a way of referring to everything at once. Visually, the artist had rendered the world as a lightblue rectangle with the other objects embedded. To be concrete, let's assume that a world is represented with a canvas of 200 pixels by 500 pixels.

Example 1: The simplest world consists of just a *UFO* and an *AUP*, with the former appearing near the top of the world and the *AUP* appearing at the bottom.

Example 2: Let's add two shots to this simplest world. Since the *AUP* can fire only one shot at a time, one of the shots is higher up in the world than the other.

2. A *UFO* is a green flying saucer that appears at a random place at the top of the world and descends from there to the ground. Visually, it consists of a green rectangle with a green disk at its center. For a simulation of its physical movement, we need the *x* and the *y* coordinate of the object and the speed at which it descends. If the *UFO* is also supposed to move left or right, its representation should include a value for the horizontal speed, too.

   Example: a *UFO* that appears at (100,10) should show in the center of the world near its top. If it moves downward at, say, two pixels per clock tick without horizontal speed, it should show up at (100,12), (100,14), and so on over the next two steps.

3. An *AUP* is a self-propelled gunship that reacts to keystrokes on the keyboard. A left arrow key moves it some number of pixels to the left, and a stroke on the right arrow key moves it the same distance to the right. Visually, it is represented as two red rectangles, one flat on the ground and the other, short one vertically pointing upwards. All we need to record in the object is the position of the *AUP*, i.e., its *x* coordinate. Since it is always on the ground, the *y* coordinate is always the bottom of the world.

   Example: an *AUP* with the *x* coordinate 100 should appear at the center of the world, near the bottom. If the player hits a left key, it should appear at 97 next, assuming the horizontal moves are three pixels wide.

4. The list of shots is either

   (a) empty, because the player hasn't fired yet, or
   (b) it consists of at least one shot and possibly some others.

   Example 1: The empty list of shots is a legitimate example.

   Example 2: Another example is a pair of shots, one fired after another, without moving the *AUP* in between the shots. In that case, the visual world should contain two yellow rectangles, one above the other,

both above the upright rectangle on the *AUP*, separated by a narrow stripe of the blue background.

5. A shot is a small bullet flying upwards. Visually, it is just a long, yellow rectangle. Its representation requires the coordinates for a position; the speed at which it is moving upwards remains constant.[10]
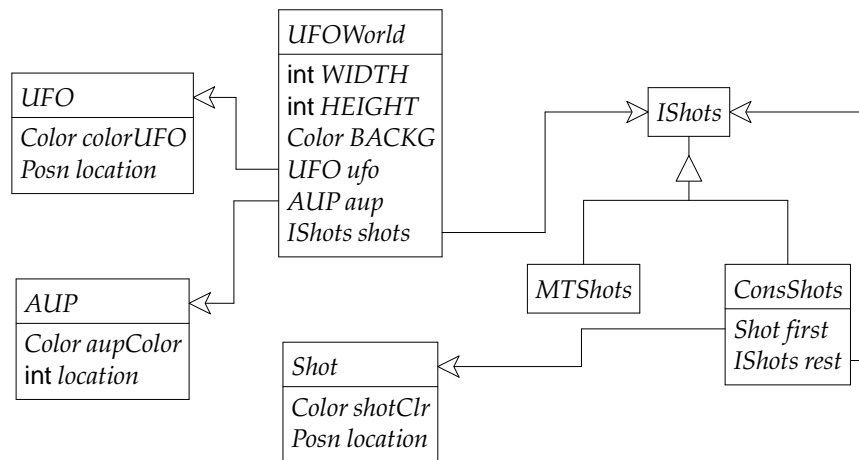
Figure 30: The World of UFOs: Class diagrams

In addition to these immediately relevant classes, we also need colors to describe objects and positions to specify their coordinates. As we have seen over the course of this first chapter, positions show up everywhere, and we have repeatedly designed similar classes. A better approach is to put such frequently used classes in a LIBRARY,[11] dubbed a PACKAGE in Java. Once these classes are in a library you can re-use them time and again, often with just one or two lines at the beginning of your program.

The specific libraries that we need here are called **geometry** and **colors**. The former defines a *Posn* class for representing positions. To include it with any of your programs, just add the line

import *geometry.∗;*

---

[10]This is a simplification of the true physics perspective. Think science fiction!

[11]Every programming language comes with a number of helpful pieces of code. Programmers have dubbed this kind of a code "library" in analogy to the physical buildings that store books of common interest to a community.
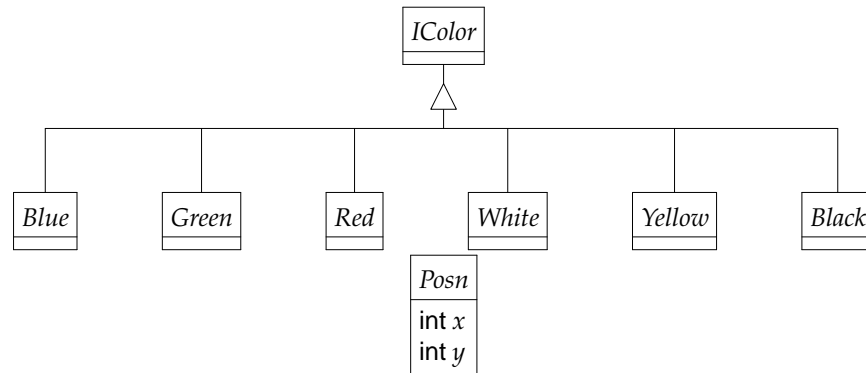
Figure 31: The colors and geometry libraries of ProfessorJ

at the top. The latter defines an interface dubbed *IColor* with classes implementing *Red*, *Green*, *Yellow*, and *Blue* (among others). To get access to these classes, use

> **import** *colors.∗*;

Figure 31 provides a diagram of the two libraries.

The second step is to draw a diagram that reflects what we know about the various kinds of information. Figure 30 contains the result of this effort. There are seven boxes: one for an interface (*IShots*) and six for classes. Three of the latter directly correspond to physical objects: *UFO*, *AUP*, and *Shot*; the other four are: *IShots*, *MtShots*, *ConsShots*, and *UFOWorld*. Roughly speaking, these four classes exist so that we can keep track of compounds of information. For example, the *UFOWorld* class contains all the pieces that play a role in the game. Similarly, *IShots* is an interface that represents all classes of list of shots. Note how the diagram treats classes from libraries just like int and *String*.

Third, you must translate the class diagram into classes and interfaces. Most of this translation is a mechanical step but remember that it also demands a decision as to which properties of objects are constant and which ones are unique and initialized during the construction of the object. To see how this decision making works, take a look at the *UFOWorld* class in figure 32. The values of the first three fields are objects whose appearance varies over time. The nature of the remaining three fields in *UFOWorld* is radically different, however. They describe aspects of the world that al-

```
// the world of UFOs, AUPs, and Shots
class UFOWorld {
    UFO ufo;
    AUP aup;
    IShots shots;
    IColor BACKG = new Blue();
    int HEIGHT = 500;
    int WIDTH = 200;

    UFOWorld(UFO ufo, AUP aup, IShots shots) {
        this.ufo = ufo;
        this.aup = aup;
        this.shots = shots;
    }
}
```

```
// an AUP: a rectangle, whose upper
// left corner is located at
// (location, bottom of the world)
class AUP {
    int location;
    IColor aupColor = new Red();

    AUP(int location) {
        this.location = location;
    }
}
```

```
// a UFO, whose center is
// located at location

class UFO {
    Posn location;
    IColor colorUFO = new Green();

    UFO(Posn location) {
        this.location = location;
    }
}
```

Figure 32: Representing UFOs and AUPs

ways remain the same; they are constants and we know their values. We therefore add initializations to the fields and omit corresponding parameters and "equations" from the constructor.

Translating the diagrams for *AUP* and *UFO* into actual classes shows that all of them have a constant color field and all other fields are object-specific: see the bottom of figure 32.

Finally, given your experience, the creation of interfaces and classes for *Shot* and list of *Shot*s are routine. They follow the pattern that we have seen several times now, without any deviation. See figure 33 for details.

```
// managing a number of shots
interface IShots {}
```

```
// the empty list of shots
class MtShots implements IShots {
    MtShots() { }
}
```

```
// a shot in flight, whose upper
// left corner is located at location
class Shot {
    Posn location;
    Color shotColor = new Yellow();

    Shot(Posn location) {
        this.location = location;
    }
}
```

```
// a list with at least one shot
class ConsShots implements IShots {
    Shot first;
    IShots rest;

    ConsShots(Shot first, IShots rest) {
        this.first = first;
        this.rest = rest;
    }
}
```

Figure 33: Representing Shots and Lists of Shots

Our last and very final step is to create examples for each class. As suggested by the recipe, this process proceeds bottom up, meaning we first create instances of those classes that don't contain any instance of the other (relevant) classes. Here we start with *AUP* and end with *UFOWorld*, which contains all kinds of objects: see figure 34. The figure shows how in a project of any size examples are collected in a separate class. Also, each example should come with a short explanation of what it represents. As we gain programming experience, we may omit such explanations from simple examples and expand those for complex examples. Still, having such explanations around strongly increases the likelihood that we can read, understand, and use these examples later when it is time to create examples of a program's inputs and outputs.

Exercises

Exercise 6.10 Collect the class definitions in this section and evaluate them in ProfessorJ. Inspect the default instance of *WoWExamples*. ∎

```
class WoWExamples {
  // an anti-UFO platform placed in the center:
  AUP a = new AUP(100);

  // a UFO placed in the center, near the top of the world
  UFO u = new UFO(new Posn(100,5));

  // a UFO placed in the center, somewhat below u
  UFO u2 = new UFO(new Posn(100,8));

  // a Shot, right after being fired from a
  Shot s = new Shot(new Posn(110,490));

  // another Shot, above s
  Shot s2 = new Shot(new Posn(110,485));

  // an empty list of shots
  IShots le = new MtShots();

  // a list of one shot
  IShots ls = new ConsShots(s,new MtShots());

  // a list of two shots, one above the other
  IShots ls2 = new ConsShots(s2,new ConsShots(s,new MtShots()));

  // a complete world, with an empty list of shots
  UFOWorld w = new UFOWorld(u,a,le);

  // a complete world, with two shots
  UFOWorld w2 = new UFOWorld(u,a,ls2);

  WoWExamples() { }
}
```

Figure 34: Some Sample Objects in the World of UFOs

Exercise 6.11 Take a look at w in figure 34. It is an instance of *UFOWorld* without any shots. Think of it as a brand new world that has just been created. Write down a new world like w assuming that the *UFO* in w has dropped by 3 pixels, that the AUP has remained at the same place, and that the player has fired one shot. A new shot is located 5 pixels above the AUP right in the middle. The width of an AUP is 20 pixels. ∎

Intermezzo 1: Classes and Interfaces

The purpose of this intermezzo is to introduce the elements of the programming language of part I. Specifically, it describes the grammar (syntax), the meaning (semantics), and the errors of the Beginner language in ProfessorJ.

Vocabulary and Grammar

The vocabulary of a language comprises the currently recognized words; its grammar governs how these words are used to form complete phrases. The following enumeration covers the phrases of Beginner that we have encountered, implicitly listing the legal words, too:

1. A program consists of **import** specifications followed by a sequence of class and interface definitions.

   Constraint: A program must not contain two class or interface definitions that use the same name. A program must not re-define an imported class or interface.

2. An interface definition introduces the name of an interface:

   > **interface** *InterfaceName* {}

   By convention, the name of an interface starts with *I* followed by a capital letter.

3. The definition of a class starts with a class header:

   > **class** *ClassName* **[implements** *InterfaceName* **]** {
   >   . . .
   > }

   By convention, the name of a class starts with a capital letter. A class may optionally specify that it implements an interface.

4. The declaration of fields follows the class header:

   > **class** *ClassName* {
   >   *Type fieldName* **[= Expr];**
   >   . . .
   > }

Each field declaration must come with a type and a name, followed by an optional initialization. The *fieldName* always starts with a low-ercase letter; all other words in the name start with an uppercase, yielding a camel-back shape for such names.

Constraint: Each field declaration introduces a name that is unique inside the class.

5. The last element of a class is a constructor definition:

       class *ClassName* {
         Type fieldName **[= Expr]**;
         . . .
         ClassName(*Type fieldName*, . . .) {
           this.fieldName = *fieldName*;
           . . .
         }
       }

   All fields without initialization are listed in the parameter part; for each parameter, the constructor body contains one "equation".

6. A *Type* is one of:

   (a) int, double, boolean, *char*
   (b) Object, *String*,
   (c) *ClassName*,
   (d) *InterfaceName*.

7. An *Expr* is one of:

   (a) a constant such as 5 or true;
   (b) a **this**.*fieldName*; or
   (c) a constructor call.

8. Constructor calls are expressions that create instances of a class:

       new *ClassName*(*Expr*,. . .)

   The sequence of expressions is as long as the sequence of constructor parameters. The first expression is for the first field parameter, the second expression is for the second field parameter, and so on.

→ a DEFINITION is one of:

  – *ImportSpec*

  – *InterfaceDefinition*

  – *ClassDefinition*

→ an IMPORTSPEC is

  import *LibN.∗*;

→ an INTERFACEDEFINITION is

  interface *InterfaceN* {}

→ a CLASSDEFINTION is:

  class *ClassN* [**implements** *InterfaceN*] {
    Type FieldN [= *Expr*];
    . . .
    ClassN(*Type FieldN*, . . .) {
      this.*FieldN* = *FieldN*;
      . . .
    }
  }

→ *LibN* is one of:

  – **colors**

  – **draw**

  – **geometry**

→ a TYPE is one of:

  – int

  – double

  – boolean

  – Object

  – *String*

  – *ClassN*

  – *InterfaceN*

→ an EXPR is one of:

  – constant (e.g., 5, true)

  – **this**.*FieldN*

  – *ConstructorCall*

→ a CONSTRUCTORCALL is

  new *ClassN*(*Expr*, . . .)

→ INTERFACEN, CLASSN, FIELDN
are alphanumeric sequences

Notes:

1. The dots (. . .) indicate a possibly empty repetition of the preceding pattern (line, expression, etc).

2. Every piece between bold face brackets is an optional part of the phrase.

3. A field declaration may not use a *FieldName* that is used for some other field declaration in the same class definition.

4. A *FieldName* whose declaration comes with an initialization "equation" may not be used as a parameter in a constructor or in an "equation" in the constructor body.

Figure 35: The grammar of ProfessorJ: Beginner

Figure 35 summarizes the grammar of ProfessorJ's Beginner language in the familiar style of informal data definitions.

Lastly, Java supports two styles of comments though we use only so-called end-of-line comments:

// two slashes turn the rest of a line into a comment

Look up "block comments" for Java on-line.

Meaning

You can specify only one form of computation in the Beginner language of ProfessorJ: the creation of objects. Specifically, a constructor call creates an instance of a class. Such an instance is a single piece of data that possibly consists of many different pieces of data. To be precise, an instance consists of as many pieces of data as there are fields in the class. Each field contains a value, which is either computed directly next to the field or in the constructor.

Consider the following program, which consists of one class definition:

```
class Posn {
  int x;
  int y;

  Posn(int x, int y) {
    this.x = x;
    this.y = y;
  }
}
```

In this context, you can evaluate a constructor call such as this one:

new *Posn*(3,4)

This creates an instance of *Posn* whose x and y fields contain the values 3 and 4, respectively. If you enter the expression at the prompt of ProfessorJ's interactions window, you see this response:

```
Posn(x = 3, y = 4)
```

In other words, the interactions window shows you the pieces of data that make up an object.

Contrast the first example with this second program:

```
class Ball {
  int radius = 3; // pixels
  int x;
  int y;
  Ball(int x, int y) {
    this.x = x;
    this.y = y;
  }
}
```

Creating an instance of *Ball* uses just two ints: one for the *x* field and one for *y*. The resulting instance, however, has three fields, that is, the interactions window displays a constructor call such as

new *Ball*(1,2)

as an object with three fields:

```
Ball(radius = 3,
     x = 1,
     y = 2)
```

Next, if your program contains the above definition of *Posn* and the following *UFO* class definition

```
class UFO {
  Posn location;
  int WIDTH = 20
  int HEIGHT = 2;
  int RADIUS = 6;

  UFO(Posn location) {
    this.location = location;
  }
}
```

then the constructor call

new *UFO*(**new** *Posn*(3,4))

creates an instance of *UFO* whose four fields contain **new** *Posn*(3,4), 20, 2, and 6, respectively. ProfessorJ displays such an instance as follows:

```
UFO(location = Posn(x = 3,
                    y = 4),
```

```
      WIDTH = 20,
      HEIGHT = 2,
      RADIUS = 6)
```

Syntax Errors, Type Errors, and Run-time Errors

Errors in ProfessorJ come in four flavors, one more than in Scheme: syntax errors, type errors, run-time errors, and logical errors. For now, we ignore the last and focus on the first three. Your program suffers from a SYNTAX ERROR if its composition violates a grammar rule. Even if the program is grammatically well-formed, ProfessorJ may still reject it and signal a TYPE ERROR, meaning the type specification of a field or a constructor parameter conflict with the kind of value that the corresponding expression produces.

If your program satisfies both the grammatical rules and the typing rules, ProfessorJ executes your program. You may then evaluate expressions in this context in the interactions window. All of these expressions just end up creating an object. Soon you will see, however, that expressions may also employ arithmetic, including division. Thus, they may attempt to divide a number by 0, in which case, ProfessorJ signals a RUN-TIME error.

Of the three errors mentioned in the section title, you are most familiar with the run-time errors of ProfessorJ's Beginner language because the semantics of this language is closely related to Scheme's. For that reason and because Beginner is too small to allow any run-time errors, the rest of this section focuses on syntax errors, which are different from those in Scheme, and type errors, which *How to Design Programs* doesn't cover at all.

Syntax or grammatical errors are usually easy to spot once they are high-lighted. For practice, let's consider a couple of grammatically ill-formed Java phrases, starting with this:

```
class Ball {
  int x = 3;
  Ball() {}
}
```

```
class Ball {
  int y;
  Ball(int y) {
    this.y = y;
  }
}
```

This sequence of definitions consists of two class definitions, both using the name *Ball*, which violates a basic constraint about programs. The following single class definition violates a similar constraint for fields, using *x* twice:

```
class Ball {
  int x;
  int x;
  Ball(int x, int y) {
    this.x = x;
    this.y = y;
  }
}
```

In addition, the constructor refers to the field *y*, which isn't declared.

Exercises

Exercise 7.1 Identify the grammatical correct programs and class definitions from the following list; for incorrect ones, explain the error message that ProfessorJ produces:

1. a program that consists of an interface and a class:

    ```
    interface Automobile { }


    class Automobile {
      int consumption; // miles per gallon
      Automobile( int consumption) {
        this.consumption = consumption;
      }
    }
    ```

2. another program that consists of an interface and a class:

    ```
    interface IVehicle { }


    class automobile implements IVehicle {
      int Consumption; // miles per gallon
      automobile(int Consumption) {
        this.Consumption = Consumption;
      }
    }
    ```

3. a program that consists of a single class definition:

```
class Automobile {
  int consumption; // miles per gallon
  Automobile(int x) {
    this.consumption = consumption;
  }
}
```

4. a second program that consists of one class:

```
class Door {
  int width;
  int height;
  String color = "yellow";
  Door(int width, int height, String color) {
    this.width = width;
    this.height = height;
    this.color = color;
  }
}
```

5. and a last program that consists of one class:

```
class Window {
  int width;
  int height;
  Window(int width) {
    this.width = width;
  }
}
```

Is there a grammatically correct program that violates a convention? ∎

Exercise 7.2 Suppose the definitions window contains one definition:

```
class Ball {
  int x;
  Ball(int x) {
    this.x = x;
  }
}
```

After clicking RUN, evaluate the following three expressions in the inter-
actions window:

1. **new** *Ball*(3,4)

2. **new** *Ball*(**new** *Posn*(3,4))

3. **new** *Ball*(3)

Predict what happens. For incorrect ones, explain the error message. ∎

Exercise 7.3 Can you spot any grammatical mistakes in these definitions:

1.
```
class Ball {
  int x;
  Ball(int x, int x) {
    this.x = x;
  }
}
```

2.
```
class Ball {
  int x;
  int y;
  int x;
  Ball(int x, int y) {
    this.x = x;
    this.y = y;
  }
}
```

3.
```
class Ball {
  int x;
  int y;
  Ball(int x, int y) {
    this.x = x;
    this.y = y;
    this.x = x;
  }
}
```

For incorrect ones, explain the error message that ProfessorJ produces. ∎

Since type errors aren't covered in *How to Design Programs*, let's study a couple of examples before we discuss their general nature. Suppose you define a class like this:

```
class Weight {
  int p = false;
  Weight() {}
}
```

Although this definition is grammatically correct, its field declaration is wrong anyway. Specifically, while the field declaration specifies that *p* is an int, the initialization "equation" has false on the right-hand side, which is a boolean value not an int.

Similarly, if you enter

new *Weight*(false)

at the prompt of the interactions window in the context of this definition:

```
class Weight {
  int p; // pounds
  Weight(int p) {
    this.p = p;
  }
}
```

ProfessorJ signals an error with the message

```
Constructor for Weight expects arguments with
type int, but given a boolean ...
```

and highlights the constructor call. The reason is that the constructor definition specifies that the constructor consumes an int but the constructor call supplies a boolean instead.

In general, type errors are mismatches between a specified (or expected) type for an expression and the type that the expression actually has. For the first example, the difference between expected type and actual type was immediately apparent. For the second example, the specified type is attached to a parameter of the constructor; the actual type of false is boolean.

To generalize properly, we need to understand what it means to determine an expression's ACTUAL TYPE. Every primitive value has an obvious

actual type. For example, 5's actual type is int; false's type is boolean; 4.27
is a double; and "four" belongs to *String*. A constructor call of the shape

> **new** *ClassName*(*Expr*, . . .)

has the actual type *ClassName*. Finally, the last kind of expression in our
grammar is the name of a field. When a field name occurs as an expression,
its actual type is the type that comes with the field declaration.

Sadly, determining the actual type of an expression isn't quite enough;
we must also discuss SUBTYPing. If the class *ClassName* **implements** the
interface *InterfaceName*, then the latter is a SUBTYPE of the former. Alter-
natively, people say *ClassName* is below *InterfaceName*. When Java matches
types, it often allows an expression with a subtype of the expected type.

Now that we understand what an actual type is and what subtyping
is, understanding a type mismatch is relatively straightforward. Here are
some canonical examples of the mismatches that can happen in ProfessorJ's
Beginner language:

1. if a field declaration comes with an initialization "equation" then the
   field type may not match the actual type of the expression on the
   right-hand side of the = sign.

   Example:

   > **class** *Ball* {
   >   int *radius* = 4.2; // int doesn't match double
   >   . . .

   The problem shows up in other guises, too. Consider the following
   (partial) union:

   Example:

   > **interface** *IVehicle* { }
   >
   > **class** *Boat* **implements** *IVehicle* {
   >   *Boat*() { }
   > }

   If the program also contains this definition,

   > **class** *SUV* {
   >   *SUV*() { }
   > }

then the following example class contains one correct and one incorrect field declaration:

```
class ExamplesOfVehicles {
  IVehicle one = new Boat();
  IVehicle two = new SUV(); // SUV is unrelated to IVehicle
  ExamplesOfVehicles() { }
}
```

Specifically, the second field declaration in *ExamplesOfVehicles* specifies on the left-hand side that *two* should always stand for objects of type *IVehicle*. The right-hand side, however, creates an instance of *SUV*, which doesn't implement *IVehicle* and is therefore unrelated to the interface as a type.

2. if a field declaration (without initialization) specifies one type and the corresponding parameter of the constructor has a different type, then the field type doesn't match the type of the expression in the corresponding constructor "equation:"

Example:

```
class Ball {
  int radius;
  Ball(double radius) {
    this.radius = radius;
  }
  . . .
}
```

3. if a constructor declaration specifies the type some parameter as one type, then the corresponding argument in a constructor call for this class must have an actual type that is below the expected type:

Example:

```
class Weight {
  int p;
  Weight(int p) {
    this.p = p;
  }
}
```

new *Weight*(`"three"`) // *Weight* expects an int, not a *String*

Here is a second type error of this kind: Example:

```
class Location {
  int x;
  int y;
  Location(int x, int y) {
    this.x = x;
    this.y = y;
  }
}
```

```
class Ball {
  double r = 3.0;
  Location p = new Location(1,this.r) ;
  Ball() { }
  . . .
}
```

In this case, *r* has type double and is used as the second argument in the gray-shaded constructor call but *Location*'s constructor expects an int in this position.

4. if a constructor declaration specifies *n* parameters and a constructor for this call has a different number of arguments, the type of the constructor call doesn't match the type of the constructor.

Example:

```
class Point1 {
  int x;
  Point1(int x) {
    this.x = x;
  }
}
```

new *Point1*(3,4) // *Point1* is applied to two arguments

While four kinds of type mismatches or type errors doesn't look like a lot, keep in mind that our language is growing and that every time you find

out about a new construct of a language, it may also introduce new kinds of type errors.

Exercises

Exercise 7.4 Identify what kind of type errors the following programs contain:

1.

```
class Employee {
  String fst;
  String lst;
  Employee(String fst, int lst) {
    this.fst = fst;
    this.lst = lst;
  }
}
```

2.

```
class Employee {
  String fst;
  String lst;
  Employee(String fst, String lst) {
    this.fst = fst;
    this.lst = lst;
  }
}
```

```
new Employee("Matthias", 1)
```

3.

```
class Customer {
  String name;
  Customer(String name) {
    this.name = name;
  }
}
```

new *Employee*(`"Matthias"`, 1)

4.

interface *IPerson* {}

class *Employee* **implements** *IPerson* {
 String fst;
 String lst;
 Employee(*String fst*, int *lst*) {
  this.*fst* = *fst*;
  this.*lst* = *lst*;
 }
}

class *Customer* {
 String name;
 Customer(*String name*) {
  this.*name* = *name*;
 }
}

class *Transaction* {
 IPerson c = **new** *Customer*(`"Kathy Gray"`);
 IPerson e = **new** *Employee*(`"Matthew"`, `"Flatt"`);
 Transaction() { }
}

Explain them in terms of the enumeration of type errors in this section. ∎

Exercise 7.5 Consider the following program:

interface *IRoomInMUD* { }

class *TowerRoom* **implements** *IRoomInMUD* { … }

class *WidowsWalk* { … }

```
class SetUp {
  IRoomInMUD room;
  SetUp(IRoomInMUD room) {
    this.room = room;
  }
}
```

If you hit RUN and evaluate the following two constructor calls, which one creates an object and which one signals a type error:

1. **new** *SetUp*(**new** *WidowsWalk*(...))

2. **new** *SetUp*(**new** *TowerRoom*(...))

What kind of type error is signaled? ∎

PICTURE: should be on even page, and even pages must be on the left

Purpose and Background

The objective of this chapter is to develop the basic skills for designing methods.

We assume that students understand the design of structurally recursive functions in the spirit of Parts I, II, and III of *How to Design Programs* plus simple accumulator-style functions from Part VI. Our methods with accumulators are relatively simple; students may understand them without the necessary background from Part VI, if they have a strong understanding of function design in general.

By the end, students should be able to add methods to a system of classes in a systematic manner. This includes conditional methods, method composition, and wish lists of methods. Adding a method systematically means designing it according to the structure of the class definition and using five steps: purpose statements and signatures; examples; templates; full definitions; and tests.

TODO

– make sure to introduce `import draw.*` in chapter 2 properly
    – does the code need `import` hints?
    – introduce and explain *String.valueOf* early
    – structural (visual) clues for the examples of section 12 (shapes)?
    – somewhere in the abstraction chapter we need to place a warning that lifting textually identical methods isn't always possible.
    – introduce error properly
    – should we encourage students to develop stubs instead of just headers?

Once you have designed a data representation for the information in your problem domain, you can turn your attention to the design of functions on this data. In an object-oriented language, functions are implemented as methods. In this chapter, you will learn to design methods following the same systematic discipline that you already know from *How to Design Programs*. It starts with a brief introduction to expressions, in general, and method calls, in particular, and then explains how to add methods to more and more complex forms of class hierarchies. The organization of this chapter is parallel to that of chapter I.

8 Expressions 1, Computing with Primitive Types

For the primitive types int, double, and boolean, Java supports a notation for expressions that appeals to the one that we use in arithmetic and algebra courses. Thus, for example, we can write

> . . . $10 * 12.50$ . . .

or

> . . . *width* $*$ *height* . . .

if *width* and *height* are method parameters of type int, or

> . . . *Math.PI* $*$ *radius* . . .

if *radius* is a method parameter of type int or double. For now, think of *Math.PI* as a name with a dot that has the expected meaning, i.e., the best possible approximation of π in Java.

    The *&&* operator computes the conjunction of two boolean expressions (are both true?); $||$ is for the disjunction of two boolean expressions (is one of them true?); and *!* is for the logical negation (is the opposite true?). For

ProfessorJ:
Interactions Window

symbol	arity	parameter types	result	example	
$!$	unary	boolean	boolean	$!(x < 0)$	logical negation
$\&\&$	binary	boolean, boolean	boolean	$a \ \&\& \ b$	logical and
$\|\|$	binary	boolean, boolean	boolean	$a \ \|\| \ b$	logical or
$+$	binary	numeric, numeric	numeric	$x + 2$	addition
$-$	binary	numeric, numeric	numeric	$x - 2$	subtraction
$*$	binary	numeric, numeric	numeric	$x * 2$	multiplication
$/$	binary	numeric, numeric	numeric	$x / 2$	division
$<$	binary	numeric, numeric	boolean	$x < 2$	less than
$<=$	binary	numeric, numeric	boolean	$x <= 2$	less or equal
$>$	binary	numeric, numeric	boolean	$x > 2$	greater than
$>=$	binary	numeric, numeric	boolean	$x >= 2$	greater or equal
$==$	binary	numeric, numeric	boolean	$x == 2$	equal

Figure 36: Some operators for numbers and booleans

example: ... $(0 < x) \ \&\& \ (x < 10)$ ... determines whether 0 is less than x (int or double) and x is less than 10.

Like mathematics (and unlike Scheme), Java comes with precedence rules so that $0 < x \ \&\& \ x < 10$ also works as expected. If you recall all these precedence rules, and if you have the courage to guess at precedences when you see new and unfamiliar operators, drop the parentheses; it you're like us, you will use parentheses anyway, because you never know who's going to read your program.

Figure 36 introduces some basic arithmetic, relational, and boolean operators for int, double, and boolean. Take a quick look now, mark the page, and consult the table when a need for these operators shows up in the following sections and exercises; it's also okay to guess on such occasions and to check your guess in ProfessorJ's interactions window.

For Java, the primitive type *String* is a class just like those we have defined in the first chapter. Each specific string is an instance of this class. The *String* class is a bit unusual in that Java doesn't require us to write

new *String*("hello world")

if we wish to create a new instance of a string. Just putting a sequence of keyboard characters between quotation marks turns them into a string.[12] Java provides some operators for processing *String*s—try `"hello"` + `"world"` at the prompt of the interaction window when you have become an Advanced ProfessorJ programmer—but in general, you really need to understand method calls to process *String*s.

9 Expressions 2, Method Calls

A method is roughly like a function. Like a function, a method consumes data and produces data. Unlike a function, however, a METHOD is associated with a class. When the method is called, it always receives at least one argument: an instance of the class with which the method is associated; we call it the method's "main" argument. Because of that, a Java programmer does not speak of calling functions for some arguments, but instead speaks of INVOKING a method on an instance or object.

Let's make this analogy concrete. Consider the Scheme function call

(string-length `"hello world"`)

The function name is string-length, which is a primitive function for strings. Its result is the number of characters in the given string, i.e., 11 in this example.

To compute the length of the same string in Java, we use the *length* method from the *String* class like this:

`"hello world"` . *length*()

That is, we write down the object first, followed by a dot, the name of the method, and a possibly empty sequence of arguments enclosed in parentheses. For emphasis, we have added optional spaces around the dot. Despite the radically different looks, the meaning of this method call is the same as that of the function call. Just like the Scheme function call to string-length, the Java method invocation of *length* consumes one string—`"hello world"`—and computes its length—11.

Like Scheme function calls, method calls can consume several arguments. For example,

`"hello"`.*concat*(`"world"`)

ProfessorJ:
Interactions Window

---

[12]While **new** *String*(`"hello"`) is an expression that produces a string, its result is not the same as `"hello"` but we ignore the difference for now.

is a method invocation of *concat* for the *String* object "hello" with a second argument: "world". The purpose is to compute the string that results from concatenating "world" to the end of the primary argument, just like (string-append "hello" "world") would. Of course, the result is "helloworld".

In general, a method call has this shape:

eObject.methodName(*expression*, . . .)

Here *eObject* is any expression that evaluates to an object and *methodName* is a method defined in the class of which the object is an instance. As this chapter shows, this could be the name of a field, the parameter of a method, or an expression that computes an object:

"hello".*concat*(" ") .*length*()

In this example, the gray-shaded part is an expression that evaluates to a *String* object, via a call to *concat*. Once the expression is evaluated (to "hello"), the *length* method is called and produces 6.

method	additional parameters	result	example
length		int	"abc".*length*()
computes the length of **this** string			
concat	String	*String*	"abc".*concat*("def")
juxtaposes **this** string and the given string			
trim		*String*	"abc".*trim*()
removes white space from both ends of **this**			
toLowerCase		*String*	"aBC".*toLowerCase*()
constructs a string with lowercase letters from **this**			
toUpperCase		*String*	"aBC".*toUpperCase*()
constructs a string with uppercase letters from **this**			
equals	String	boolean	"abc".*equals*("bc")
is **this** string equal to the given one?			
endsWith	String	boolean	"abc".*endsWith*("bc")
does **this** string end with the give suffix?			
startsWith	String	boolean	"abc".*startsWith*("ab")
does **this** string start with the give suffix?			

Figure 37: Some methods for *String*

In general, the *String* class provides many useful methods in addition to the ones we encountered so far. The table in figure 37 lists and explains

some basic *String* methods. Their names and purpose statements suggest what they compute; explore their behavior in the Interactions window.

10   Methods for Classes

In *How to Design Programs*, function bodies are expressions, involving the function parameters. If the given values are structures, function bodies also use structure selectors; if the given values fall into distinct cases, a conditional is used to distinguish the cases. The same is true for methods, and this section shows how to design methods for basic classes.

10.1   Designs through Templates

Take a look at this revised version of our very first problem:

> . . .   Design a method that computes the cost of selling bulk coffee at a specialty coffee seller from a receipt that includes the kind of coffee, the unit price, and the total amount (weight) sold. . . .

In section 2, we designed the *Coffee* class to represent the information about a coffee sale. It is now time to design the method that actually computes the cost of such a transaction.

Instead of plunging directly into the programming activity, let us recall how we would design this function using the recipes from *How to Design Programs*. The first step of our generic design recipe calls for the design of a data representation; we already have that:

> (**define-struct** *coffee* (*kind price weight*))
> ;; *Coffee* (sale) is:
> ;; — (*make-coffee String Number Number*)

The goal of the second step is to write down a contract, concise purpose statement, and a function header:

> ;; *cost : Coffee* → *String*
> ;; to compute the total cost of a coffee purchase
> (**define** (*cost a-coffee*) . . .)

Here *a-coffee* is the function parameter that stands for the instance of *coffee* structure supplied with applications of *cost*.

Next we must make up some examples, that is, function calls for *cost* that illustrate what it should produce when given certain arguments. Naturally, we use the examples from the problem statement (page 9):

(*cost* (*make-coffee* `"Kona"` 2095 100)) ; should produce
209500

Turn the other two data examples into functional examples, too.

The fourth step is the crucial one for most function designs. It requests that we refine the function header to a function template by making all the knowledge about the structure of the arguments explicit. After all, the function computes the outputs from the given information. Since *cost* consumes a structure—an instance of *coffee*—you have the structure itself and all of its field values:

(**define** (*cost a-coffee*)
    . . . (*coffee-kind a-coffee*) . . .
    . . . (*coffee-price a-coffee*) . . .
    . . . (*coffee-weight a-coffee*) . . .)

The template contains three (selector) expressions, because the *coffee* structure has three fields. Each expression extracts one value from *a-coffee*, the parameter.

The transition from the template to the full function definition—the fifth step—starts with an examination of the data that the function consumes. The function must be able to compute the result from just these pieces of data. Here we need only two of the pieces, of course:

(**define** (*cost a-coffee*)
  (∗ (*coffee-price a-coffee*) (*coffee-weight a-coffee*)))

The sixth and final step is to test the examples that we worked out above.

In Java, we don't design independent functions. As we already know from section 9, we instead design methods that are a part of a class. Later we invoke the method on an instance of this class, and this instance is the method's primary argument. Thus, if the *Coffee* class already had a *cost* method, we could write in the example section

new *Coffee*(`"Kona"`, 2095, 100).*cost*()

and expect this method call to produce 209500.

Let's try to develop this method systematically, following our well-known design recipe. First we add a contract, a purpose statement, and a header for *cost* to the *Coffee* class:

```
// the bill for a coffee sale
class Coffee {
  String kind;
  int price; // in cents per pound
  int weight; // in pounds

  Coffee(String kind, int price, int weight) // intentionally omitted

  // to compute the total cost of this coffee purchase [in cents]
  int cost() { ... }
}
```

The purpose statement for the method is a comment just like the purpose statement for the class. The contract is no longer a comment, however. It is an integral part of a Java method. In the terminology of Java, it is a METHOD SIGNATURE. The int to the left of *cost* says that we expect the method to produce an integer; the purpose statement reminds us that it is the number of cents.

At first glance, the signature also seems to say that *cost* doesn't consume anything, but remember that *cost* is always invoked on some specific instance of *Coffee* (and one could say it consumes an instance of *Coffee*). Furthermore, this instance is the primary argument to the method, and it therefore has a standard (parameter) name, **this**, which you never need to include in the parameter list explicitly. We can thus use **this** in the purpose statement—reminding readers of the role of the special argument— and method body to refer to the instance of *Coffee* on which *cost* is invoked:

inside of *Coffee* :
// to compute the total cost of **this** coffee purchase [in cents]
int *cost*() { ... **this** ... }

Note: To avoid wasting space, we show only the modified parts of a class. The underlined phrase is there to remind you where the fragment belongs. ∎

Now that we have clarified the basic nature of the method, let's reformulate the functional examples in Java:

Coffee c = **new** *Coffee*("Kona" , 2095, 100)
...
check *c.cost*() **expect** 209500

That is, in the context of an existing example, we invoke the *cost* method. Reformulate the other examples from the Scheme approach in Java.

The next step is to formulate the template. Recall that the template expresses what we know about the argument(s). In our running example, the

ProfessorJ:
Examples & Tests

"input" is **this** instance of the class. Each instance consists of three pieces of data: the *kind*, the *price*, and the *weight*. In Scheme, we use special functions, the selectors, to extract the relevant pieces. In Java, we access an object's fields with the dot notation. In general, we write:

   object . field

Since we wish to access the fields of **this** object, we write **this**.*kind*, **this**.*price*, and **this**.*weight* in the method body to create the template:

   inside of *Coffee* :
   ‾‾‾‾‾‾‾‾‾‾‾‾‾‾
   // to compute the total cost of **this** coffee purchase
   int *cost*() {
      ... **this**.*kind* ... **this**.*price* ... **this**.*weight* ...
   }

The rest is easy. We must decide which pieces of the template are relevant and how to use them. In our running example, the two relevant pieces are **this**.*price* and **this**.*weight*. If we multiply them, we get the result that we want:

   inside of *Coffee* :
   ‾‾‾‾‾‾‾‾‾‾‾‾‾‾
   // to compute the total cost of **this** coffee purchase
   int *cost*() {
      return this.*price* ∗ **this**.*weight*;
   }

The **return** keyword points the reader to the expression in a method body that produces the result of a method call. While it is obvious here because there is just one way to compute the result, you may already be able to imagine that such a hint can be useful for conditional expressions. The complete class definition including the method is shown in figure 38.

   The figure also contains an extended class diagram. In this new diagram, the box for *Coffee* consists of three compartments. The first still names the class, and the second still lists the fields that each instance has. The third and new compartment is reserved for the method signatures. If the method's name doesn't sufficiently explain the computation of the method, we can also add the purpose statement to the box so that the diagram can tell the story of the class by itself.

ProfessorJ:
Automatic Tests

   Finally, the figure extends the examples class from chapter I (page 15). In addition to the sample instances of *Coffee*, it also contains three *test* fields: *testKona*, *testEthi*, and *testColo*. Like the example fields, these test fields are also initialized. The right-hand side of their initialization "equation" is a **check** ... **expect** ... expression, which compares the result of a method call

Coffee
String *kind*   int *price* [in cents per pound]   int *weight* [in pounds]
int *cost*()

```
// represent a coffee sale:
// at which price, how much
//  coffee was sold
class Coffee {
  String kind;
  int price;
  int weight;

  Coffee(String kind,
         int price,
         int weight) {
    this.kind = kind;
    this.price = price;
    this.weight = weight;
  }

  // to compute the total cost
  // of this coffee purchase
  int cost() {
    return this.price * this.weight;
  }
}
```

```
// collect examples of coffee sales
class CoffeeExamples {
  Coffee kona =
    new Coffee("Kona",2095,100);
  Coffee ethi =
    new Coffee("Ethiopian", 800, 1000);
  Coffee colo =
    new Coffee("Colombian", 950, 20);

  boolean testKona =
    check this.kona.cost() expect 209500;
  boolean testEthi =
    check this.ethi.cost() expect 800000;
  boolean testColo =
    check this.colo.cost() expect 19000;

  CoffeeExamples() { }
}
```

Figure 38: The *cost* method for the *Coffee* class

with an expected value and produces true or false. When you place the two classes into the definitions window and run the program, ProfessorJ creates an instance of *CoffeeExamples* and determines how many of the fields with *test* in their name are true.

Some methods must consume more data than just **this**. Let us see how the design recipe applies to such problems:

> . . .   The coffee shop owner may wish to find out whether a coffee sale involved a price over a certain amount. . . .

Clearly, a method that can answer this question about any given instance of coffee must consume a second argument, namely, the number of cents with which it is to compare the *price* of the sale's record.

First we write down the purpose statement and the signature:

inside of *Coffee* :
// to determine whether **this** coffee's price is more than *amt*
boolean *moreCents*(int *amt*) { … }

The purpose statement again reminds us that *moreCents* consumes two arguments: **this** and *amt*. Second, we make a couple of examples:

check new *Coffee*("Kona", 2095, 100).*moreCents*(1200) **expect** true
check new *Coffee*("Columbian", 950, 200).*moreCents*(1000) **expect** false

To practice your design skills, explain the expected results.
    The template for this method is exactly that for *cost*:

inside of *Coffee* :
// to determine whether **this** coffee's price is more than *amt*
boolean *moreCents*(int *amt*) {
  … **this**.*kind* … **this**.*price* … **this**.*weight*
}

We do not have to include anything about the second argument, *amt*, because it is a part of the signature and its type is just int, i.e., atomic data.
    The only relevant pieces of data in the template are *amt* and **this**.*price*:

inside of *Coffee* :
// to determine whether **this** coffee's price is more than *amt*
boolean *moreCents*(int *amt*) {
  return this.*price* > *amt*;
}

Don't forget that the last step of the design recipe is to run the examples and to check that the method produces the expected results. So, turn the examples into additional *test* fields in *CoffeeExamples* (from figure 38).
    Lastly, methods may also have to consume instances of classes, not just primitive values, as their secondary arguments. Take a look at this problem:

> … The coffee shop owner may also wish to find out whether some coffee sale involved more weight than some given coffee sale. …

Naturally, a method that compares the weight for two kinds of coffee consumes two instances of *Coffee*. We call them **this** and, by convention, *that* in the purpose statement:

inside of *Coffee* :
// to determine whether **this** coffee sale is lighter than *that* coffee sale
boolean *lighterThan*(*Coffee that*) { … }

Let's make up some examples:

    check new *Coffee*(`"Kona"` , 2095, 100)
      .lighterThan(**new** *Coffee*(`"Columbian"` , 950, 200)) **expect** true

    check new *Coffee*(`"Ethiopian"` , 800, 1000)
      .lighterThan(**new** *Coffee*(`"Columbian"` , 950, 200)) **expect** false

The gray-shaded boxes represent *that*, the second argument to *lighterThan*. For both cases, the answers are obvious because in order to determine whether one coffee sale involves less weight than the other, we just compare the weights in each.

In the template for a method such as *lighterThan*, we write down the fields for **this** and the fields for the instance of the other argument. Remember that *that.kind* extracts the value from the *kind* field of the object *that*:

    inside of *Coffee* :
    // to determine whether **this** coffee sale is lighter than *that* coffee sale
    boolean *lighterThan*(*Coffee that*) {
      … **this**.*kind* … *that.kind* … // *String*
      … **this**.*price* … *that.price* … // int
      … **this**.*weight* … *that.weight* … // int
    }

Note how we have added comments about the types of the fields in this template. Adding type annotations is useful when you move from the template step to the method definition step.

Of course, the only relevant fields are the *weight* fields, and so we get this complete and simple definition:

    inside of *Coffee* :
    // to determine whether **this** coffee sale is lighter than *that* coffee sale
    boolean *lighterThan*(*Coffee that*) {
      return this.*weight* < *that.weight*;
    }

Now test the method to validate the examples that we made up. The complete code for the *Coffee* class is displayed in figure 39.

10.2   Finger Exercises

Exercise 10.1 Recall the class *Image* from exercise 2.3. Design the following methods for this class:

```
// represent a coffee sale:
// at which price how much coffee was sold
class Coffee {
  String kind;
  int price;
  int weight;

  Coffee(String kind, int price, int weight) {
    this.kind = kind;
    this.price = price;
    this.weight = weight;
  }

  // to compute the total cost
  // of this coffee purchase
  int cost() {
    return this.price * this.weight;
  }

// to determine whether this
// coffee's price is more than amt
  boolean moreCents(int amt) {
    return this.price > amt;
  }

  // to determine whether this coffee sale
  // involves less weight than that coffee sale
  boolean lighterThan(Coffee that) {
    return this.weight < that.weight;
  }
}
```

Coffee
String *kind*   int *price* [in cents per pound]   int *weight* [in pounds]
int *cost*()   boolean *moreCents*(int *amt*)   boolean *lighterThan*(*Coffee that*)

Figure 39: The *Coffee* class with methods

1. *isPortrait*, which determines whether the image's height is larger than its width;

2. *size*, which computes how many pixels the image contains;

3. *isLarger*, which determines whether one image contains more pixels than some other image; and

   4. *same*, which determines whether **this** image is the same as a given one.

Also draw a complete class diagram (by hand). ∎

Exercise 10.2 Develop the following methods for the class *House* from exercise 3.1:

   1. *isBigger*, which determines whether one house has more rooms than some other house;

   2. *thisCity*, which checks whether the advertised house is in some given city (assume we give the method a city name);

   3. *sameCity*, which determines whether one house is in the same city as some other house.

Before you design the method, draw a complete class diagram for *House* (by hand). ∎

Exercise 10.3 Here is a revision of the problem of managing a runner's log (see figure 7, page 20):

> … Develop a program that manages a runner's training log. Every day the runner enters one entry concerning the day's run. … For each entry, the program should compute how fast the runner ran in minutes per mile.[13] …

Develop a method that computes the pace for a daily entry. ∎

Exercise 10.4 A natural question concerning *Date*s (see figure 7, page 20) is whether one occurs earlier than another. Develop the method *earlierThan*.

   Hint: The first possibility is that year of the first date is smaller than the year of the second. Next, what do you do if the years are the same? ∎

10.3   Designing Methods for Classes

The examples in this section validate the design recipe from *How to Design Programs* again. Specifically, steps 2 through 6 of the design recipe for functions on structures work for methods in basic classes (i.e., classes without references to other classes), too. So from now on, when you are to design a method for a basic class,

---

[13]Although speed is usually measured in "miles per minute," i.e., distance over time, runners usually care more about how many minutes and seconds they need per mile than speed per se.

1. formulate a purpose statement and a method header;

2. illustrate the purpose statement with functional examples;

3. lay out what you know about the method's argument(s) in a template;

   In the case of basic classes, the template consists of the parameters, **this**, and all the fields of the class written as **this**.*fieldname*. Typically we just write down the latter.

4. define the method; and

5. run the examples as tests.

Keep these steps in mind as you encounter more complex forms of data than plain classes.

10.4 Conditional Computations

Like functions, methods must sometimes distinguish different situations and compute results according to the given situation. *How to Design Programs* poses a simple example with the problem of computing the interest earned for a bank certificate of deposit:

> ... Develop a method that computes the yearly interest for *certificates of deposit* (CD) for banks. The interest rate for a CD depends on the amount of deposited money. Currently, the bank pays 2% for amounts up to $5,000, 2.25% for amounts between $5,000 and $10,000, and 2.5% for everything beyond that. ...

Since the problem statement contains descriptions of intervals, the problem analysis includes a graphical rendering of these intervals:
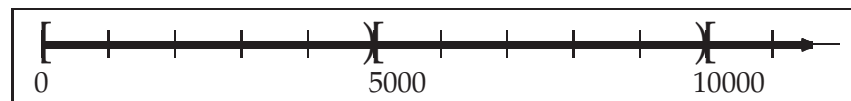
This picture is a good foundation for the construction of both the example step as well as the template step later.

First, however, we must represent the major problem information— bank certificates of deposit—in our chosen language, i.e., as a class:

```
// represent a certificate of deposit
class CD {
  String owner;
  int amount; // cents
```

```
CD(String owner, int amount) {
    this.owner = owner;
    this.amount = amount;
  }
}
```

In reality, an account would contain many fields representing the owner(s), tax information about the owner(s), account type, and many other details. For our little problem, we use just two fields: the owner's name and the deposited amount.

Translating the intervals from the problem analysis into tests yields three "interior" examples:

> **check new** *CD*("Kathy", 250000).*interest*() **expect** 5000.0
> **check new** *CD*("Matthew", 510000).*interest*() **expect** 11475.0
> **check new** *CD*("Shriram", 1100000).*interest*() **expect** 27500.0

For each example, we multiply the percentage points with the amount and divided by 100. (Why?) Add examples that determine how *rate* works for borderline examples. If the problem statement isn't clear to you, make up your mind for the two borderline cases and proceed.

Working through the examples clarifies that this method needs to distinguish three situations. More precisely, the computations in the method body depend on the deposited amount of money. To express this kind of conditional computation, Java provides the so-called IF-STATEMENT, which can distinguish two possibilities:

> **if** (*condition*) {
>   *statement1* }
> **else** {
>   *statement2* }

As the notation suggests, an if-statement tests a *condition*—an expression that produces a boolean value—and selects one of two statements, depending on whether *condition* evaluates to true or false. The only statement you know so far is a **return** statement, so the simplest **if** statement looks like this:

> **if** (*condition*) {
>   **return** *expression1*; }
> **else** {
>   **return** *expression2*; }

Of course, as their name suggests **if** statements are also statements, so replacing *statement1* or *statement2* in the schematic **if** is legitimate:

```
  if (condition) {
    return expression1; }
  else {
    if (condition2) {
      return expression2; }
    else {
      return expression3; }   }
```

Here we replaced *statement2* with the gray-shaded **if** statement. The complete statement thus distinguishes three situation:

1. if *condition* holds, the computation proceeds with **return** *expression1*;

2. if *condition* doesn't hold but *condition2* holds; then the computation proceeds with **return** *expression2*;

3. and if neither *condition1* nor *condition2* evaluates to true, then the computation proceeds with **return** *expression3*.

The analysis and the examples distinguish three situations so we do need two **if** statements as shown above:

```
inside of CD :
// compute the interest rate for this account
double interest() {
   if (0 <= this.amount && this.amount < 500000) {
      ... this.owner ... this.amount ... }
   else { if (500000 <= this.amount && this.amount < 1000000) {
      ... this.owner ... this.amount ... }
   else {
      ... this.owner ... this.amount ... }
   }
}
```

From the first design recipe we know to use **this**.*amount*; the tests come from the pictorial analysis of the problem.

The ownership data naturally plays no role for the computation of the interest rate. So finishing the definition from the template is easy:

<u>inside of *CD* :</u>
```
// compute the interest rate for this account
double interest() {
   if (0 <= this.amount && this.amount < 500000) {
      return 2.00 * this.amount; }
   else { if (500000 <= this.amount && this.amount < 1000000) {
      return 2.25 * this.amount; }
   else {
      return 2.50 * this.amount; }
   }
}
```

Your task now is to formulate an examples class and the method examples as tests in the same class. When you are done you may also ponder the following challenge:

ProfessorJ:
Testing with doubles

> ... The bank has decided that keeping track of fractional cents no longer makes any sense. They would like for *interest* to return an int. ...

Find the documentation for Java's *Math* class and read up on *Math.round*. Then modify the design of *interest* appropriately, including the tests.

Let's look at a second example. Suppose your manager asks you for some exploratory programming:

```
// represent a falling
// star on a 100 x 100 canvas
class Star {
   int x = 20;
   int y;
   int DELTA = 5;

   Star(int y) {
      this.y = y;
   }
}
```

... Develop a game based on the Grimms brothers' fairy tale called "Star Thaler." ... Your first task is to simulate the movement of the falling stars. Experiment with a single star that is falling straight to the ground at a fixed number of pixels per time unit on a 100 × 100 canvas. Once the star has landed on the ground, it doesn't move anymore. ...

For good measure, your manager has already designed a data representation according to the design recipe. A falling star has a location, which means x and y coordinates, and it moves downward at some fixed rate. Hence the class has three fields. The x coordinate and the rate of descend

DELTA are always the same for now; the *y* coordinate increases[14] continuously.

Your task is to develop the method *drop*, which creates a new star at a different position. Following the design recipe you extract a concise purpose statement for the method from the problem statement:

> inside of *Star* :
> // drop **this** Star by *DELTA* pixels,
> // unless it is on (or close) to the ground
> *Star drop*() {
>   ... **this**.*y* ... **this**.*DELTA* ...
> }

Here the purpose statement just reformulates two sentences from the problem statement. It naturally suggests that there are two distinct kinds of stars: one that is falling and one that has landed. This, in turn, means that we need at least two kinds of examples:

> *Star s* = **new** *Star*(10)
> *Star t* = **new** *Star*(100)

> **check** *s*.*drop*() **expect new** *Star*(15)
> **check** *t*.*drop*() **expect new** *Star*(100)

The first example, *s*, represents a free-falling star in the middle of the canvas. The second one, *t*, is a star on the ground; this kind of star doesn't move anymore. Of course, the two examples also point out that we don't know what happens when the star is close to the ground. Since the problem statement seems to imply that stars just land on the ground and then stop, we should add one more test case that clarifies this behavior:

> **check new** *Star*(98).*drop*() **expect new** *Star*(100)

That is, once the star is *close enough*, *drop* just creates a star that has landed.

Using an **if**-statement, we can finish the template for *drop*:

---

[14]Remember that on a computer, the origin of the Cartesian grid is in the upper left. Going to the right increases the *x* value, going down increases the *y* value.

```
// represent a falling star
// on a 100 x 100 canvas
class Star {
   int x = 20;
   int y;
   int DELTA = 5;

   Star(int y) {
      this.y = y;
   }

   // drop this Star by DELTA pixels,
   // unless it is on or close to the ground
   Star drop() {
      if (this.y + this.DELTA >= 100) {
         return new Star(100); }
      else {
         return new Star(this.y + this.DELTA); }
   }
}
```

Star
String *kind*
int *x* = 20
int *y*
int *DELTA* = 5
int *drop*()

Figure 40: A falling star

inside of *Star* :
// drop **this** Star by *DELTA* pixels, unless it is on or near the ground
Star drop() {
    if (**this**.*y* + **this**.*DELTA* >= 100)

    . . .
    else // the star is in the middle of the canvas

    . . .
}

Remember that if you can distinguish different intervals in a numeric type or different cases in another atomic type of data, a template distinguishes as many situations as there are sub-intervals (cases).

Now take a look at figure 40, which contains the full method definition for *drop*. If the condition holds, the method returns a star at height 100 (the ground level); otherwise, it actually creates a star that has dropped by a few pixels.

A bit of reflection suggests that we could have easily decided to distinguish three different situations:
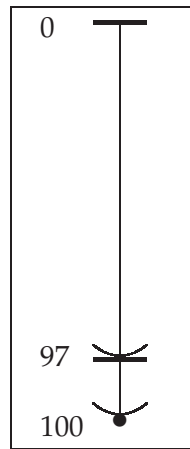
inside of *Star* :

```
// drop this Star by DELTA pixels,
// unless it is on or near the ground
Star drop() {
   if (this.y + this.DELTA < 97) {
      return ...; }
   else { if (this.y + this.DELTA < 100) {
      return ...; }
   else { // this star has landed
      return ...;
     }
   }
}
```

On the left you see a number line with the appropriate sub-intervals, just like in *How to Design Programs*. There are three: from 0 to 97 (exclusive); from 97 (inclusive) to 100 (exclusive); and 100. On the right, you see a template that distinguishes the three situations. It uses two **if** statements, one followed by another. In principle, you can chain together as many of them as are necessary; the details are explained in the next intermezzo.

Exercises

Exercise 10.5 Modify the *Coffee* class from figure 38 so that *cost* takes into account bulk discounts:

> ... Develop a program that computes the cost of selling bulk coffee at a specialty coffee seller from a receipt that includes the kind of coffee, the unit price, and the total amount (weight) sold. If the sale is for less than 5,000 pounds, there is no discount. For sales of 5,000 pounds to 20,000 pounds, the seller grants a discount of 10%. For sales of 20,000 pounds or more, the discount is 25%. ...

Don't forget to adapt the examples, too. ∎

Exercise 10.6 Take a look at this following class:

```
// represent information about an image
class Image {
    int width; // in pixels
    int height; // in pixels
    String source;

    Image(int width, int height, String source) {
        this.width = width;
        this.height = height;
        this.source = source;
    }
}
```

Design the method *sizeString* for this class. It produces one of three strings, depending on the number of pixels in the image:

1. `"small"` for images with 10,000 pixels or fewer;

2. `"medium"` for images with between 10,001 and 1,000,000 pixels;

3. `"large"` for images that are even larger than that.

Remember that the number of pixels in an image is determined by the area of the image. ∎

Exercise 10.7 Your physics professor would like to simulate an experiment involving bouncing balls. Design a class that represents a ball that is falling on a 10 x 100 canvas at a rate of *DELTA*. That is, each time the clock ticks, the ball drops by *DELTA* pixels.

When the ball reaches the bottom of the canvas, it bounces, i.e., it reverses course and travels upwards again. The bounce is perfect, meaning the ball travels the full distance during the bounce. Put different, if the ball is far enough away from a wall, it just travels *DELTA* pixels. If it is too close for that, it drops by whatever pixels are left and then reverses course for the remaining number of pixels. As it reverses course, it continues to travel at the same speed.

Design the method *move*, which simulates one step in the movement of the ball. ∎

As you design conditional methods, don't forget the design recipe from *How to Design Programs* for just this kind of function. If it is complex, draw a number line to understand all the intervals (cases, enumerated items). Pick examples from the interior of each interval and for all the borderline cases.

```
// a certificate of deposit
class CD {
   String owner;
   int amount; // cents

   CD(String owner, int amount) {
      this.owner = owner;
      this.amount = amount;
   }

   // compute the interest rate (in %) for this account
   double rate() {
      if (0 <= this.amount
          && this.amount < 500000) {
         return 2.00; }
      else { if (500000 <= this.amount
                  && this.amount < 1000000) {
         return 2.25; }
      else {
         return 2.50; }
     }
   }

   // compute the interest to be paid for this account
   double payInterest() {
      return (this.rate() * this.amount)/100;
   }
}
```

CD
String *owner*
int *amount*
int rate()
int *payInterest*()

Figure 41: A CD with interest payment

10.5 Composing methods

In *How to Design Programs*, we learned to create one function per task, especially if the tasks are complex. For example, a function for computing the average of a series of numbers must compute their sum, count the numbers, and divide the former by the latter. This means it involves two complex tasks (adding, counting) that are best turned into separate functions.

The same guideline applies to the design of methods. When a method's task is complex, identify separate tasks and design a method for each task. As you identify those auxiliary or helper tasks, create a wish list to keep track of all the things to do.

For a concrete example, consider figure 41. It displays a class for representing certificates of deposit that can also compute the amount of interest that the bank must pay. The relevant method is called *payInterest*. It first determines the appropriate interest rate with **this**.*rate*(), multiplies by the deposit amount, and finally divides it by 100 because the rate is represented as a percentage.

```
// information about an image
class Image {
   int width;
   int height;
   String source;

   Image(int width, int height, String source) {
      this.width = width;
      this.height = height;
      this.source = source;
   }

   // is this image large?
   String sizeString() {
      if (this.area() <= 10000) {
         return "small"; }
      else { if (this.area() <= 1000000) {
         return "medium"; }
      else {
         return "large"; }
      }
   }

   // determine the (pixel) area of this image
   int area() {
      return this.width * this.height;
   }
}
```

Image
int *width*
int *height*
String *source*
String *sizeString*()
int *area*()

Figure 42: Images with *area*

A second example appears in figure 42. The *Image* class contains two methods: *sizeString* and *area*. The former refers to the latter, because its result depends on the area that the image represents. Specifically, both conditions in *sizeString* evaluate **this**.*area*() which computes the area of the image

and then compare it to some given threshold.

   In summary, the design of methods benefits from factoring tasks into smaller, manageable units that you compose. Composing methods is as natural as composing functions. As you do partition tasks, however, don't forget to design each of them systematically.

Exercises

Exercise 10.8 Study this class definition:

```
// the daily percipitation of three consecutive days
class Precipitation {
  int day1;
  int day2;
  int day3;


  Precipitation(int day1, int day2, int day3) {
    this.day1 = day1;
    this.day2 = day2;
    this.day3 = day3;
  }


  // how much did it rain during these three days?
  int cumulative() {
    return this.day1 + this.day2 + this.day3;
  }
}
```

Add the method *average* to this class definition. Follow the design recipe and reuse existing methods, if possible. ∎

Exercise 10.9 Design the class *JetFuel*, whose purpose it is to represent the sale of some quantity of jet fuel. Each instance contains the quantity sold (in integer gallons), the quality level (a string), and the current base price of jet fuel (in integer cents per gallon). The class should come with two methods: *totalCost*, which computes the cost of the sale, and *discountPrice*, which computes the discounted price. The buyer gets a 10% discount if the sale is for more than 100,000 gallons. ∎

11 Methods and Object Containment

Figure 43 displays a class diagram, two class definitions, and an examples class in the lower left. The *Rectangle* class itself has three fields: two int fields and one *CartPt*. The latter type is a reference to the second class, which represents the Cartesian coordinates of a point.

Clearly the purpose of the two classes is to represent rectangles on the Cartesian plane. Such rectangles have three characteristics: their *width*, their *height*, and their location. If we assume that the rectangle's sides are parallel to the *x* and *y* axes, a single point in the plane determines the entire rectangle. Let's refer to this single point as the "anchor" point. Because we are working with canvases on a computer monitor and because software libraries place the origin in the top-left corner, we choose the rectangle's upper left corner as its anchor point.

Now consider this excerpt from a problem statement:

> ... Design a method that computes the distance of a *Rectangle* to the origin of the canvas. ...

Although a *Rectangle* has many points, your program should compute the *shortest* distance between the rectangle and the top-left corner:
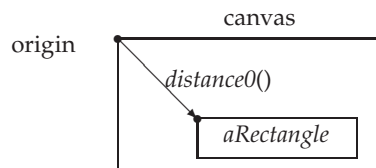
Even a cursory glance at this picture suggests that the shortest distance between the *Rectangle* and the origin is the distance between its top-left corner (anchor point) and the origin (see arrow).

The obvious conclusion from this problem analysis is that the problem is really asking for the development of *two* methods: one for *Rectangle* and one for *CartPt*. The second measures the distance of a *CartPt* to the origin and the first measures the distance of a *Rectangle* to the origin:

inside of *Rectangle* :	inside of *CartPt* :
// to compute the distance of	// to compute the distance of
// **this** *Rectangle* to the origin	// **this** *CartPt* to the origin
double *distance0*() { ... }	double *distance0*() { ... }

The two purpose statements and method signatures just restate our intentions with code. They also lead straight to the second step, the development of examples for both methods:

// a rectangle on a canvas, located
// at tlCorner, *width* pixels wide
// and *height* pixels high
class *Rectangle* {
  CartPt tlCorner;
  int *width*;
  int *height*;

  Rectangle(CartPt tlCorner,
           int *width*,
           int *height*) {
    this.*tlCorner* = *tlCorner*;
    this.*width* = *width*;
    this.*height* = *height*;
  }
}

class *ShapeExamples* {
  CartPt p = **new** *CartPt*(3,4);
  CartPt q = **new** *CartPt*(5,12);

  Rectangle r = **new** *Rectangle*(*p*,5,17);
  Rectangle s = **new** *Rectangle*(*q*,10,10);

  ShapeExamples() { }
}

// a Cartesian point on a canvas
// *x* pixels to the right of the upper
// left corner and *y* pixels down
class *CartPt* {
  int *x*;
  int *y*;

  CartPt(int *x*, int *y*) {
    this.*x* = *x*;
    this.*y* = *y*;
  }
}

Figure 43: A class of *Rectangle*s

check *p.distance0*() **expect** 5   **check** *r.distance0*() **expect** 5
check *q.distance0*() **expect** 13   **check** *s.distance0*() **expect** 13

Make sure you understand the expected result for each example and that you can associate it with the interpretation of the above sample picture.

Now it's time to turn our attention to the template. According to the basic design recipe, we first need to add one selector expression per field in each method body:

inside of *Rectangle* :
```
double distance0() {
   ... this.tlCorner ...
   ... this.width ...
   ... this.height ...
}
```

inside of *CartPt* :
```
double distance0() {
   ... this.x ...
   ... this.y ...
}
```

The method template in *Rectangle* contains three expressions because the class definition contains three fields; the same reasoning applies to the template in *CartPt*. But, remember that the purpose of a template is to translate the organization of the data definition into expressions. And given that we use diagrams as data definitions, there is clearly something missing: the arrow from *Rectangle* to *CartPt*.

A moment's thought suggest that this containment arrow suggests the natural connection between the two method templates:

inside of *Rectangle* :
```
double distance0() {
   ...  this.tlCorner.distance0()  ...
   ... this.width ...
   ... this.height ...
}
```

inside of *CartPt* :
```
double distance0() {
   ... this.x ...
   ... this.y ...
}
```

The gray-shaded method call expresses this containment arrow. It reiterates what we discovered through a careful analysis of the picture above. In general terms, the arrow says that *CartPt* is a separate class; the method call says that if we want to deal with properties of the *tlCorner*, we delegate this computational task to the corresponding methods in its class.

From here, completing the method definitions is simple. The *distance0* method in *Rectangle* invokes the *distance0* method on the *tlCorner* object. This replaces the task of computing the distance of the *Rectangle* to the origin with the task of computing the distance of a single point to the origin. Whatever the latter produces is the result of the former, too—just like our geometric reasoning suggested and the template development confirmed.

Do you remember the formula for computing the distance between two points? If not, you will have to find (and pay) a geometry expert who knows it:

$$\sqrt{x^2 + y^2}$$

The complete method definitions are displayed in figure 44. The only unusual aspect of the figure is the name of the $\sqrt{\cdot}$ operator: *Math.sqrt*. Like *Math.PI* it contains a dot; again, just think of this name as strange for now.

```
class Rectangle {
  CartPt tlCorner;
  int width;
  int height;

  Rectangle(CartPt tlCorner, …)
              int width,
              int height) { … }

  // to compute the distance of
  // this Rectangle to the origin
  double distance0() {
    return
      this.tlCorner.distance0();
  }
}
```

```
Rectangle
─────────────────
CartPt tlCorner
int width
int height
─────────────────
double distance0()
```

```
CartPt
─────────────────
int x
int y
─────────────────
double distance0()
```

```
class CartPt {
  int x;
  int y;

  CartPt(int x, int y) { … }

  // to compute the distance of
  // this CartPt to the origin
  double distance0() {
    return
      Math.sqrt(
        this.x ∗ this.x
        +
        this.y ∗ this.y);
  }
}
```

Figure 44: *Rectangle*s and *CartPt*s with methods

Let's practice this design with the weather-record example from figure 11. A weather record consists of a date, three temperature ranges, and the amount of today's precipitation. The diagram contains three classes: one for *WeatherRecord*s, one for *Date*s, and one for *TemperatureRange*s. If your company has contracts with meteorologists, you may one day encounter the following problem on your desk:

| *WeatherRecord* |
| *Date d* |
| *TemperatureRange today* |
| *TemperatureRange normal* |
| *TemperatureRange record* |
| double *precipitation* |

| *Date* |
| int *day* |
| int *month* |
| int *year* |

| *TemperatureRange* |
| int *high* |
| int *low* |
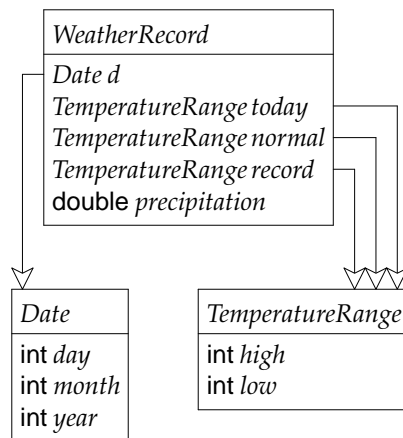
... Design a method that computes today's temperature differentials from a weather record. ...

For good measure, the problem statement includes the old class diagram for the problem. As you can tell, it contains not just one containment arrow, like the previous example, but four of them.

Given the extra complication, let's first make examples of these objects: see figure 45, left-hand side. Interpret the examples in real world; try to think of places that might have such weather records.

Now that you have a diagram and examples—it is easy to imagine the actual class definitions by now—you can start with the design recipe for methods. Figure 45 (right-hand side) contains a class diagram with proper method signatures and a purpose statement for the main method. The problem statement dictates the name and purpose statement for *Weather-Record*; for the other two classes, the diagram contains only basic signatures, because we don't know yet what we need from them.

Using the class diagram, you can develop the method templates in a straightforward manner:

```
// WeatherRecord              // TemperatureRange        // Date
int differential() {          ??? nnn() {                ??? lll() {
  ... this.date.lll() ...       ... this.low ...           ... this.day ...
  ... this.today.nnn() ...      ... this.high ...          ... this.month ...
  ... this.normal.nnn() ...   }                            ... this.year ...
  ... this.record.nnn() ...                              }
  ... this.precipitation ...
}
```

The template in *WeatherRecord* contains five expressions because there are five fields in the class definition. Due to the types of the first four fields, the first four selectors are equipped with method calls to other method templates along the containment arrows. The other two method templates are
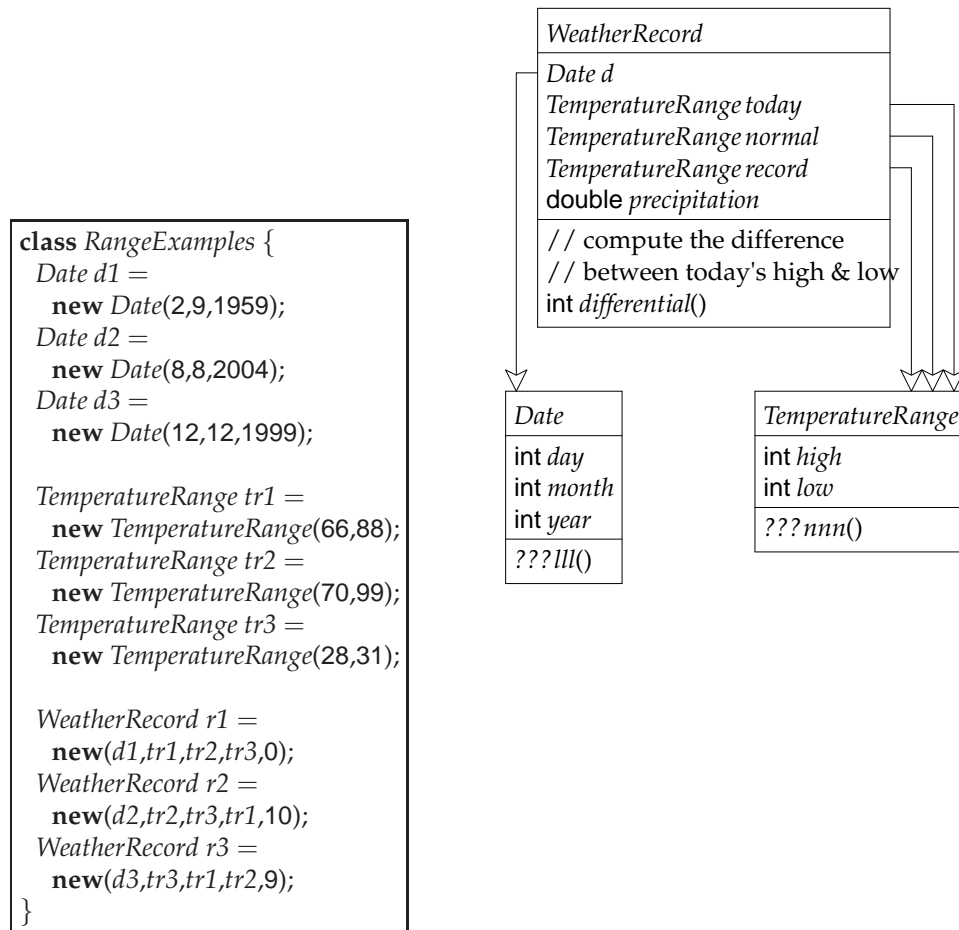
Figure 45: Recording the weather

entirely routine.

A look at the template in *WeatherRecord* suffices to complete the definition:

inside of <u>*WeatherRecord*</u> :	inside of *TemperatureRange* :
int *differential*() {	int *difference*() {
return this.*today.difference*();	**return this**.*high* − **this**.*low* ;
}	}

The *differential* method requires only one field for its computation: *today*. This field has a class type and therefore the method delegates a task to the contained class. Here, it obviously delegates the task of computing the

difference between the two temperature to the *TemperatureRange* class.

Obviously neither of the two examples is particularly complex, and you could have programmed the method without any complex design recipe. Keep in mind, however, that such examples must fit into the pages of a book and real examples are much larger than that. Without a systematic approach, it is easy to get lost.

11.1 Finger Exercises

Exercise 11.1 Recall the problem of designing a program that assists bookstore employees (see exercise 3.3). Add the following methods to the *Book* class:

- *currentBook*, which checks whether the book appeared during a given year;

- *thisAuthor*, which determines whether a book is written by the given author;

- *sameAuthor*, which determines whether this book is written by the same author as the given book. ∎

Exercise 11.2 Exercise 3.2 provides the data definition for a weather recording program. Design the following methods for the *WeatherRecord* class:

1. *withinRange*, which determines whether today's *high* and *low* were within the normal range;

2. *rainyDay*, which determines whether the precipitation is higher than some given value;

3. *recordDay*, which determines whether the temperature broke either the high or the low record. ∎

11.2 Designing Methods for Classes that Contain Classes

Every time we encountered a new form of data in *How to Design Programs*, we checked our design recipe and made sure it still worked. Usually we added a step here or a detail there. For the design of classes, we need to do the same.

In principle, the design recipe from section 10.3 applies to the case when a class contains (or refers to) another class. Now, however, the wish list, which we already know from *How to Design Programs*, begins to play a major role:

1. Formulate a method header and a purpose statement to the class to which you need to add functionality; also add method headers to those classes that you can reach from this class via containment arrows. You may or may not need those auxiliary methods but doing so, should remind you to delegate tasks when needed.

2. Illustrate the purpose statement with functional examples; you can reuse the data examples from the design steps for the class.

3. Create the template for the method. Remember that the template contains what you know about the method's argument(s). This includes schematic method calls on those selector expressions that refer to another class in the diagram. In short, follow the (direction of the) containment arrow in the diagram.

4. Define the method. If the computation needs data from a contained class, you will need to develop an appropriate method for this other class, too. Formulate a purpose statement as you place the method on the wish list.

5. Work on your wish list, step by step, until it is empty.

6. Run the functional examples for those classes that don't refer to other classes via containment arrows. Test the others afterwards.

12   Methods and Unions of Classes

The two preceding sections are reminders of the wish list and templates, two central design concepts. As we know from *How to Design Programs*, the former is essential when we design programs that consist of many functions. We can't do it all at once, so we need to keep track of what is left to do. The template is the key to organizing programs. It reflects the data organization; if the problem data changes, as it inevitably does, it is easy to change a well-organized program in a similar manner. While creating a template may seem superfluous for simple kinds of data, like those in the first two sections of this chapter, we know from experience that as soon as we work with unions of data, especially self-referential unions, templates become essential. This section covers a number of union examples and drives home once again how the data organization and the template mirror each other. The first subsection demonstrates how easy it is to define several methods once the template for a class configuration is worked out.

12.1 Example: Plain Geometric Shapes

Figure 12 (page 28) displays a class hierarchy that represents basic geometric shapes—dots, squares, and circles—for a drawing program. Recall that it is the data representation for a program that deals with such shapes, drawing them and allowing people to compute certain properties. A problem statement related to this program may include this fragment:

> ... Add the following four methods to the class hierarchy: (1) one that computes the area of shapes; (2) one that produces the distance of shapes to the origin; (3) another one that determines whether some point is inside some shapes; (4) and a final one creates the bounding box of shapes. ...

These methods must work for all shapes, which means for all objects that have type *IShape*. In an object-oriented language, we can express this requirement with the addition of a method signature to the *IShape* interface. The presence of a method header says that all classes that make up this union must contain a matching method.

Naturally, the various methods may compute their results with different strategies. For example, a *Dot* has no area because it represents a point;[15] the *area* method can either return 0 as a result or it can signal an error. In contrast, both *Square*s and *Circle*s have proper areas but they are computed with different formulas.

To improve your understanding of templates, let's first create templates for methods on shapes in general. We assume nothing about their return type for now and also assume that they consume only one value: the shape itself (**this**). Given these assumptions, take a look at the *IShape* interface in figure 46. It contains a partial signature for just such a method *mmm*. Because we now know that a signature in an interface demands that each implementing class must contain a concrete version of *mmm*, too, the figure also contains signatures for *mmm* in *Dot*, *Square*, and *Circle*. Indeed, we can go even further. Because of the containment arrow from these three classes to *CartPt* (see figure 12 on page 28), we add a signature for a method *nnn* to *CartPt*, too. Thus, if any of the other *mmm* methods need to compute something about *CartPt*, we can refer to this template and, if needed, use it to define an actual method.

As for the template themselves, the *mmm* method in *Dot* consumes an instance of *Dot* and therefore has access to the one field of *Dot*: *loc*. The

ProfessorJ:
implements is
not Java's

---

[15]Its display is a small circle and thus has an area, but the point itself doesn't.

body of *mmm* therefore contains the selector expression **this**.*loc*. Following the design recipe from section 11.2, the expression is **this**.*loc.nnn*() because we need to textually express the containment arrow from *Dot* to *CartPt* in the class diagram.

```
                                          // a Cartesian point in the plane
                                          class CartPt {
  // the class of all geometric shapes      int x;
  interface IShape {                        int y;

    ??? mmm();                              CartPt(int x, int y) {
  }                                           // omitted
                                            }

                                            ??? nnn() {
                                                ... this.x ... this.y ...
                                            }
                                          }
```

```
// a dot                    // a square                  // a circle
class Dot                   class Square                 class Circle
  implements IShape {         implements IShape {          implements IShape {
  CartPt loc;                 CartPt loc;                  CartPt loc;
                              int size;                    int radius;

  Dot(CartPt loc) {           Square(CartPt loc,           Circle(CartPt loc,
    // omitted                        int size) {                  int radius) {
  }                             // omitted                   // omitted
                              }                            }

  ??? mmm() {                 ??? mmm() {                  ??? mmm() {
    ... this.loc.nnn() ...       ... this.loc.nnn() ...       ... this.loc.nnn() ...
  }                             ... this.size ...            ... this.radius ...
                              }                            }
}                           }                            }
```

Figure 46: Classes for geometric shapes with methods and templates

---

In *Square*, we have two different fields: *loc* and *size*. The former is dealt with like *loc* in *Dot*; the latter, *size*, is just an int. Therefore we just add **this**.*size* to the template, without any schematic method call. Convince yourself that the design recipe suggests a similar treatment of *mmm* in *Circle*.

Finally, we preemptively deal with *nnn* in *CartPt*. It is invoked on an instance of *CartPt*. This class contains two fields, both of type int: *x* and *y*. A method in that class is potentially going to use those two fields; so we add **this**.*x* and **this**.*y* to remind ourselves of this possibility.

Before you read on, study the collected templates in figure 46. It contains all the code fragments that we have just discussed. The rest of this section shows how easy it is to produce actual methods from this template with the four sample requirements from the problem statement.

The *area* problem: Following the design recipe, we formulate a purpose statement and refine the signatures, starting with the one in *IShape*:

inside of *IShape* :
// to compute the area of **this** shape
double *area*();

As you read this purpose statement and the signature, keep in mind that they are located in an interface, which is a prescription for all classes that implement it. Together, the purpose statement and the signature thus describe *what* a method in an implementing class must compute. Of course, it can't say *how* the methods have to compute their results; that depends on the variant in which they are located.

Next we create functional examples. For an interface, they must cover all concrete classes that implement it. Currently there are three classes, so here are three examples, immediately formulated as tests:

```
class ShapeExamples {
  IShape dot = new Dot(new CartPt(4, 3));
  IShape squ = new Square(new CartPt(4, 3), 3);
  IShape cir = new Circle(new CartPt(12, 5), 2);

  boolean testDot = check dot.area() expect 0.0 within 0.1;
  boolean testSqu = check squ.area() expect 9.0 within 0.1;
  boolean testCir = check cir.area() expect 12.56 within 0.01;
  ShapeExamples() { }
}
```

ProfessorJ:
Testing with doubles

The result for *Dot* says that its area is 0.0. For *Square*s, we naturally just square the *size* of the side, and for *Circle*s we multiply the square of the radius with π. Note how the checks are formulated with a tolerance.

Using the template from figure 46 and the examples, we easily obtain the three concrete methods:

inside of *Dot* :	inside of *Square* :	inside of *Circle* :
double *area*() {	double *area*() {	double *area*() {
return 0;	**return this**.*size* ∗ **this**.*size*;	**return**
}	}	(*Math.PI* ∗
		this.*radius* ∗
		this.*radius*);
		}

Intuitively, the location of each shape plays no role when you compute their area, and dropping the selector expressions (**this**.*loc.nnn*()) confirms this intuition.

The only interesting aspect of testing these methods concerns the comparison of the expected value with the computed value. Recall that Java's type double represents a discrete collection of rational numbers on the number line, i.e., not all rational numbers, and that computations on these numbers is inherently inaccurate.

The *distance* **problem:** Except for *Dot*s, shapes consist of many different points, so just as for *Rectangle*s in section 11, we take this problem to mean that the method computes the distance between the origin and the *closest point*. Furthermore, let's assume that the entire shape is visible on the canvas.[16] From this, we get a purpose statement and a signature:

inside of *IShape* :
// to compute the distance of **this** shape to the origin
double *distTo0*();

For the construction of examples, we re-use the "inputs" from the *area* problem because they have easy to compute distances:

check *dot.distTo0*() **expect** 5.0
check *squ.distTo0*() **expect** 5.0
check *cir.distTo0*() **expect** 11.0

The first two expected results are obvious. The distance between a *Dot* and the origin is the distance between the *Dot*'s location and the origin; the distance between the square and the origin is the distance between the *top-left corner* of the *Square* and the origin. The third one is similar, but while we still compute the distance between the center of the *Circle* and the origin, we must also subtract the radius from this number. After all, the points on the circle are closer to the origin than its center.

---

[16]Okay, we are not only reading the problem but also simplifying it. Can you figure out what to do when we don't make this assumption?

Since all methods must compute the distance between their *loc* field and the origin, it makes sense to refine the template in *CartPt* into a *distTo0* method, too:

inside of *CartPt* :
// to compute the distance of **this** point to the origin
double *distTo0*() { … }

// Functional Examples:
check (**new** *CartPt*(4, 3)).*distTo0*() **expect** 5.0
check (**new** *CartPt*(12, 5)).*distTo0*() **expect** 13.0

The functional examples are naturally adapted from the previous ones.

At this point, you can either finish the method in *CartPt* or those in *Dot*, *Square*, and *Circle*. We start with the method in *CartPt*:

inside of *CartPt* :
// to compute the distance of **this** point to the origin
double *distTo0*(){
   return *Math.sqrt*((**this**.*x* ∗ **this**.*x*) + (**this**.*y* ∗ **this**.*y*));
}

The method computes the distance of a point to the origin in the usual fashion (see page 11). Since this method doesn't rely on any other method in our classes, you can test it immediately. Do so.

Now that we have a distance method for Cartesian points, we can easily complete the three methods in the shape classes:

inside of *Dot* :	inside of *Square* :	inside of *Circle* :
double *distTo0*() {	double *distTo0*() {	double *distTo0*() {
return	**return**	**return**
this.*loc.distTo0*();	**this**.*loc.distTo0*();	**this**.*loc.distTo0*()
}	}	− **this**.*radius*;
		}

All three delegate the task of computing the distance to the origin to the appropriate method for *loc*. The method in *Circle* performs an additional computation; the others just pass on the result of the computation in *CartPt*.

The point location problem: The third problem requests a method that can find out whether some point falls within the boundaries of a shape:

inside of *IShape* :
// is the given point within the bounds of **this** shape?
boolean *in*(*CartPt p*);

A method like *in* is useful when, among other situations, you are designing a program that must determine whether a mouse click is within a certain region of a canvas.

Given that there are three classes that implement *IShape* and exactly two distinct outcomes, there are three pairs of test cases:

1. Conceptually, a *CartPt* is within a *Dot* if the former is equal to the latter's location:

   > *IShape dot* = **new** *Dot*(**new** *CartPt*(100, 200));
   > **check** *dot.in*(**new** *CartPt*(100, 200)) **expect** true
   > **check** *dot.in*(**new** *CartPt*(80, 220)) **expect** false

2. Deciding whether a point is within *Square* is difficult if you are looking at program text only. It is therefore good practice to translate examples into graphical figures on grid paper and to check how the dots relate to the shape. Take a look at these two examples:

   > **check**
   >   **new** *Square*(**new** *CartPt*(100, 200), 40).*in*(**new** *CartPt*(120, 220))
   > **expect** true
   > **check**
   >   **new** *Square*(**new** *CartPt*(100, 200), 40).*in*(**new** *CartPt*(80, 220))
   > **expect** false

   Draw these two situations and confirm the expected results.

   Note: This is, of course, one more situation where we suggest that you interpret data as information.

3. For *Circle*s the given point is inside the circle if distance between the point and the center is less than the radius:

   > **check new** *Circle*(**new** *CartPt*(0, 0), 20).*in*(**new** *CartPt*(4, 3))
   > **expect** true
   > **check new** *Circle*(**new** *CartPt*(0, 0), 10).*in*(**new** *CartPt*(12, 5))
   > **expect** false

Recall that this kind of knowledge is domain knowledge. It is best to acquire as much basic domain knowledge as you can from courses and books; otherwise you have to find domain experts and work with them.

As usual, thinking through these examples provides hints on how to go from the template to the full definition. Let's look at *Dot* first:

<u>inside of *Dot*</u> :
```
boolean in(CartPt p) {
  ... this.loc.nnn() ...
}
```

The template reminds us that we may have to design a method for *CartPt* to complete the definition of *in*. The examples suggest that the desired method compares *p* and *loc*; if they are the same, the answer is true and otherwise it is false. From these two ideas, it is natural to put a *same* method for *CartPt* on the wish list and to define *in*:

<u>inside of *Dot*</u> :
```
boolean in(CartPt p) {
  return this.loc.same(p);
}
```

Here are three drawings that represent the question whether a given point is within a *Square*:
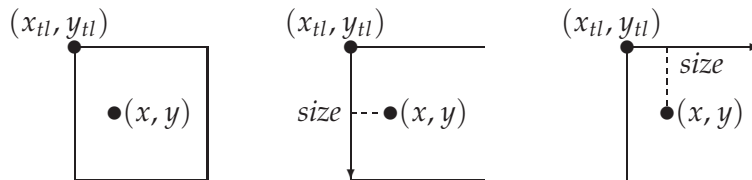
The leftmost picture depicts the actual question. Visually it is obvious that the dot is within the square, which is determined by the top-left corner and the size of the square. To do so via a computation, however, is complicated. The basic insight is that to be inside of the square means to be between two pairs of lines.

The picture in the middle and on the right show what this means graphically. Specifically, in the middle we see the top and bottom line and a vector that indicates what the distance between them is. The dashed line from the given point to the vector explains that its y coordinate has to be between the y coordinate of the top-left corner and the y coordinate of the line of the bottom; the latter is $y_{tl} + size$. Similarly, the picture on the right indicates how the x coordinate of the given point has to be between the x coordinate of the top-left corner and the x coordinate of the rightmost line; again the latter is *size* pixels to the right of the former.

Let's look at the template and see how we can translate this analysis of the examples into code:

inside of *Square* :
  boolean *in*(*CartPt p*) {
    ... **this**.*loc.nnn*() ... **this**.*size* ...
  }

The adapted template on the left contains two selector expressions, reminding us of *loc* and *size*. From the example we know that both play a role. From the former we need the coordinates. From those and *size*, we get the coordinates of the parallel lines, and the coordinates of *p* must be in between. Furthermore, because we need to check this "betweeness" twice, the "one task, one function"guideline implies that we put *between* on our wish list:

  // is *x* in the interval [*lft*,*lft*+*wdth*]?
  boolean *between*(int *lft*, int *x*, int *wdth*)

Assuming the wish is granted, finishing the definition of *in* is straightforward:

inside of *Square* :
  boolean *in*(*CartPt p*) {
    return this.*between*(**this**.*loc.x*,*p.x*,**this**.*size*)
      *&& **this**.*between*(**this**.*loc.y*,*p.y*,**this**.*size*);
  }

This brings us to *Circle*:

inside of *Circle* :
  boolean *in*(*CartPt p*) {
    ... **this**.*loc.nnn*() ... **this**.*radius* ...
  }

Given the examples for *Circle* and the experience with the previous cases, it is now almost easy to go from the template to the full definition. As we agreed above, if the distance of the given point is less than or equal to the radius, the point is within the circle. This statement and the template suggest a last addition to the wish list, namely *distanceTo* for CartPt:

inside of *Circle* :
  boolean *in*(*CartPt p*) {
    return this.*loc.distanceTo*(*p*) <= **this**.*radius*;
  }

Since we are proceeding in a top-down fashion this time, we can't test anything until we empty the wish list. It contains three methods: *same* for *CartPt*; *between* for *Square*; and *distanceTo* also for *CartPt*.

The *between* method checks a condition for three numbers:

inside of *Square* :
// is *x* in the interval [*lft,lft+wdth*]?
boolean *between*(int *lft*, int *x*, int *wdth*) {
  return *lft* $<= x$ && $x <= lft + wdth$;
}

As a helper method for the *in* method of *Square* it is located in this class.[17]

To design *same* and *distanceTo* we just follow the design recipe of section 10. Both methods are given two points: **this** and *p*, which suggests the following refinement of our template:

??? nnn(*CartPt p*){
    return ... **this**.*x* ... *p.x* ...
         ... **this**.*y* ... *p.y* ... ;
}

The rest is just an application of some geometric domain knowledge to programming: when **this** and *p* are the same and what the distance is between two points, given their coordinates. You can look this up in a geometry book or you can look ahead to section 19.1.[18]

The bounding box problem: The bounding box of a shape is the smallest rectangle that completely surrounds the given shape.[19] To make this concrete, let's look at our three concrete kinds of shapes:

1. Just as *Dot*s don't have an area, they also don't have a real bounding box. One possibility is to signal an error. Another one is to pick the smallest possible square as a representative.[20]

2. The bounding box for a *Square* is obviously the square itself.

3. For an instance of *Circle* finally, the bounding box is also a square:

---

[17]Note, however, that the method doesn't use **this** and therefore doesn't conceptually belong into *Square*.

[18]Alternatively, if you recall the Pythagorean Theorem, you can (re)construct the formula now with a little sketch on the back of an envelope. Developing this skill will serve you well.

[19]Bounding boxes play an important role in graphics software but a thorough explanation is beyond this book.

[20]Mathematicians deal with such special cases all the time. Their experience suggests that the proper treatment of special cases depends on the problem and its context.

More concretely, consider the circle on the left with radius *r*. On the right, the same circle comes with its bounding box, whose width and height are 2 ∗ *r* and whose top-left corner is one radius removed from the center of the circle in both directions.

In short, the bounding box of any one of our shapes is a square.

Let's use this problem analysis to refine the template from figure 46. First we create a header and a purpose statement in *IShape* from *mmm*:

inside of *IShape* :
// compute the bounding box for **this** shape
Square bb();

It is somewhat unusual that the return type of the method is *Square*, one of the classes implementing *IShape*, but this just reflects the observation that, in our case, the bounding boxes are just squares.

Second, we make up some examples, one per concrete shape:

check *dot.bb*() **expect new** *Square*(**new** *CartPt*(100, 200),1)
check *squ.bb*() **expect** *squ*
check *cir.bb*() **expect new** *Square*(**new** *CartPt*(10,3), 4)

The first two cases are straightforward. For the last one, draw the given situation on grid paper and determine for yourself why the expected answer is correct.

Our discussion of the problem and the examples make it easy to define the methods in *Dot* and *Square*:

inside of *Dot* :
Square bb() {
  return new *Square*(**this**.*loc*,1);
}

inside of *Square* :
Square bb() {
  return this;
}

The difficult part is the definition of *bb* in *Circle*. It demands the creation of a top-left corner for the new bounding box. From the problem analysis and the examples we know that the top-left corner of the bounding box is exactly **this**.*radius* to the top and the left of the center of the circle. A look at the template suggests that the creation of this point is an operation on *CartPt*:

inside of *Circle* :   inside of *Circle* :
Square bb() {   *Square bb*() {
   … **this**.*loc.nnn*(…) …   **return**
   … **this**.*radius* …   **new** *Square*(**this**.*loc.translate*(− **this**.*radius*),
}   2 ∗ **this**.*radius*);
                                                      }

In geometry, this operation on points (shapes actually) is called a translation. We therefore put *translate* on our wish list:

inside of *CartPt* :
// create a point that is *delta* pixels (up,left) from **this**
CartPt translate(int *delta*)

and wrap up the definition as if we had this new method.
    Defining *translate* is actually easy:

inside of *CartPt* :
// create a point that is *delta* pixels (up,left) from **this**
CartPt translate(int *delta*) {
   return new *CartPt*(**this**.*x* − *delta*, **this**.*y* − *delta*);
}

The method's primary argument is a *CartPt*. Hence, its template contains the usual ingredients: **this**.*x* and **this**.*y*, in addition to the parameter. Furthermore, the purpose statement tells us exactly what to do: subtract *delta* from **this**.*x* and **this**.*y*. Still, it would be best to follow the design recipe and to create examples and tests now.

    Figures 47 and 48 collect all the code fragments for *IShape*, *Square*, and *CartPt* into complete class definitions.

Exercises

Exercise 12.1 Collect all fragments of *Dot* and *Circle* and complete the class hierarchy in figures 47 and 48. Also collect the examples and build a working test suite for the hierarchy. ∎

Exercise 12.2 Revise the class diagram in figure 12 so that it matches the actual definitions in this section. ∎

Exercise 12.3 Design an extension for the classes in figures 47 and 48 that deals with isosceles right triangle. Assume the right angle is always in the lower right corner and that the two sides adjacent to the right angle are

```
interface IShape {                      class Square implements IShape {
  // to compute the area of this shape     int size;
  double area();                           CartPt loc;

  // to compute the distance of            Square(CartPt loc, int size) {
  // this shape to the origin                ... // omitted
  double distTo0();                        }

  // is the given point within?            double area() {
  // the bounds of this shape                return this.size * this.size;
  boolean in(CartPt p);                    }

  // compute the bounding box              double distTo0() {
  // for this shape                          return this.loc.distTo0();
  Square bb();                             }
}
                                          boolean in(CartPt p){
                                            return
                                              this.between(this.loc.x, p.x, this.size)
                                              &&
                                              this.between(this.loc.y, p.y, this.size);
                                          }

                                          Square bb() {
                                              return this;
                                          }

                                          // is x in the interval [lft,lft+wdth]?
                                          boolean between(int lft, int x, int wdth) {
                                            return lft <= x && x <= lft + wdth;
                                          }
                                        }
```

Figure 47: Classes for geometric shapes with methods (part 1)

always parallel to the two axes. The extension should cope with all the methods in *IShape*.

Remember your first design step is to develop a data representation for these triangles. Two obvious representation come to mind: one just uses the three points and the other one uses a representation similar to the one for squares in this section. Explore both with examples before you design the rest of the program. Use examples to justify your design choice. ∎

```
class CartPt {
  int x;
  int y;

  CartPt(int x, int y) { ... // omitted ... }

  // to compute the distance of this point to the origin
  double distTo0(){
      return Math.sqrt( (this.x * this.x) + (this.y * this.y));
  }

  // are this CartPt and p the same?
  boolean same(CartPt p){
      return (this.x == p.x) && (this.y == p.y);
  }

  // compute the distance between this CartPt and p
  double distanceTo(CartPt p){
      return
        Math.sqrt((this.x − p.x) * (this.x − p.x) + (this.y − p.y) * (this.y − p.y));
  }

// create a point that is delta pixels (up,left) from this
  CartPt translate(int delta) {
    return new CartPt(this.x − delta, this.y − delta);
  }
}
```

Figure 48: Classes for geometric shapes with methods (part 2)

Exercise 12.4 Design an extension for the classes in figures 47 and 48 so that a program can request the perimeter of a shape. ∎

Exercise 12.5 Combine the extensions of exercises 12.3 and 12.4. ∎

12.2 Signaling Errors

Twice we have suggested that your program could signal an error. First, we said that **new** *Date*(45, 77, 2003) doesn't represent a real date in our calendar, with the implication being that it shouldn't really produce an object. Second, we also mentioned that a *Dot* doesn't have an area and that we may wish to define a method that signals an error instead.

An error in Java is an exception. To make life simple, ProfessorJ provides a method with the name *Util.error*, which consumes a *String* and then signals an error:

inside of *Dot* :
  double *area*() {
    return *Util.error*(`"end of the world"`);
  }

Thus, if you were to evaluate **new** *Dot*(**new** *CartPt*(10,22)).*area*() for this version of *Dot*, the evaluation would terminate, display `"end of the world"`, and highlight the above expression.

We will deal with signaling errors in constructors later.

13   Types, Classes, and How Method Calls Compute

Before we continue our exploration of the design of methods, we must understand how method calls really work. The answer to this question raises a second, equally important question concerning the role of types.

13.1   Method Dispatch

Thus far we have dealt with methods as if they were functions. Specifically, when we saw a method call, we imagined substituting the values of the arguments for the respective parameters into the body of the function and then we determined the value of this new expression. While this process is a good approximation, it fails to explain how methods work for unions of classes, i.e., once we have interfaces as types and classes as implementations of interfaces.

To improve our understanding, let us study a concrete example. Imagine an interior designer who wishes to help customers create comfortable, well-balanced rooms. Instead of playing with real furniture, an interior designer is better off simulating the room, the furniture, and the layout with a program. That is, we want to study a program that allows an interior designer to create shapes (representing furniture and power outlets); to place them into another shape (representing a room); and to determine some basic properties of this simulated interior design.

Even if we cannot understand all the details of the visual editing process yet, it is easy to imagine that the following problem may come up:

```
Room
────────
int x
int y
IShape a
IShape b
IShape c
```

```
IShape
```

```
Dot
────────
CartPt loc
```

```
Square
────────
CartPt loc
int size
```

```
Circle
────────
CartPt loc
int radius
```

```
CartPt
────────
int x
int y
```

```
// tracking pieces of furniture
// in this room
class Room {
    int width;
    int height;

    IShape a;
    IShape b;
    IShape c;

    Room(int width, int height,
            IShape a, IShape b, IShape c) {
        this.width = width;
        this.height = height;
        this.a = a;
        this.b = b;
        this.c = c;
    }

    // the ratio of area covered
    // by furniture in this room
    double covered() {
        return (this.a.area()
                + this.b.area()
                + this.c.area())
               /
               (this.width * this.height);
    }
}
```

Figure 49: Keeping track of furniture

. . . Design a class that represents a room. The purpose of the
class is to keep track of the furniture that is placed into the room.
. . . Among other things, the interior designer who uses the pro-
gram may wish to know the ratio of the area that the furniture

covers. . . .

For simplicity, assume that the program deals with exactly three pieces of furniture and with rectangular rooms. For the graphical presentation of the model, assume that the furniture is represented as *IShape*s, i.e., a *Dot* (say, a floor lamp), a *Square* (say, a chair), or a *Circle* (say, a coffee table).

The problem statement suggests a class with five fields: the room's width, its height, and its three pieces of furniture. Even though we can't know which three pieces of furniture the interior designers will place in the room, we can use *IShape* as the type of the three fields because it is the type of the union of these three classes.

Equipped with a data representation for a room, we can turn to the task of designing the method that computes the ratio of the furniture's area to the room's area:

inside of *Room* :
// the ratio of area covered by furniture in **this** room
double *covered*() {
  . . . **this**.*a* . . . **this**.*b* . . . **this**.*c* . . . // all *IShape*
  . . . **this**.*width* . . . // int
  . . . **this**.*height* . . . // int
}

The template reminds us of the five fields. Making up examples is easy, too: if the room is 10 by 10 feet and the three pieces of furniture each cover 20 square feet, the result of *covered* ought to be .6. The example and its explanation suggest the following expression for the method body:

(**this**.*a*.*area*() + **this**.*b*.*area*() + **this**.*c*.*area*())
/
(**this**.*width* ∗ **this**.*height*)

In other words, the method computes and adds up the three areas covered by the furniture and then divides by the product of *height* and *width*, the area of the room.

Each gray-shaded expression is a method invocation that calls *area* on an object of type *IShape*. The method is specified in the interface and thus all implementing classes must support it. The question is how an object-oriented programming language evaluates expressions such as these or how does it decide which *area* method to use.

Suppose you play interior designer and create this *Room* object:

> *Room re* = **new** *Room*(25,12, **new** *Dot*(. . .),
> **new** *Square*(. . . ,10),
> **new** *Circle*(. . . ,5))

Then *a* stands for **new** *Dot*(. . .), *b* for **new** *Square*(. . . ,10), and *c* for **new** *Circle*(. . . ,5). From that, our substitution model of computation says a method call to *covered* proceeds like this:

> **this**.*a*.*area*() + **this**.*b*.*area*() + **this**.*c*.*area*()
> // evaluates to:
> **new** *Dot*(. . .).*area*() + **this**.*b*.*area*() + **this**.*c*.*area*()
> // evaluates to:
> 0 + **this**.*b*.*area*() + **this**.*c*.*area*()
> // evaluates to:
> 0 + **new** *Square*(. . . ,10).*area*() + **this**.*c*.*area*()
> // evaluates to:
> 0 + 100 + **this**.*c*.*area*()
> // evaluates to:
> 0 + 100 + **new** *Circle*(. . . ,5).*area*()
> // evaluates to:
> 0 + (10 ∗ 10) + (5 ∗ 5 ∗ *Math.PI*)

The gray-shaded fragments in each line are evaluated. The transition from the first line to the second explains that types don't matter during an evaluation. The shaded invocation of *area* of *a* is replaced with an invocation to a concrete object: **new** *Dot*(. . .). And at this point, it is completely clear which *area* method is meant. The same is true in the transitions from line 3 to line 4 and line 5 to line 6, except that in those cases *area* from *Square* and *Circle* are used, respectively.

Thus what really matters is how an object is created, i.e., which class follows **new**. If the given *IShape* is an instance of *Dot*, the invocation of *area* picks *Dot*'s *area*; if it is *Square*, it's *Square*'s version of *area*; and if it is *Circle*, the evaluation defers to the definition of area in *Circle*. Afterwards, we continue to replace parameters (also called identifiers) by their values and proceed with the evaluation of arithmetical expressions as we know it from primary school or *How to Design Programs*.

The mechanism of picking a method out of several based on its class is called POLYMORPHIC METHOD DISPATCH. In this context, the word "polymorphic" refers to the fact that any class that implements *IShape* supplies a method called *area* but each of the method definitions is unique. No conditionals or checks are written down in the program; instead the programming language itself chooses.

Compare this to Scheme where an *area* function would be defined with a conditional expression:

```
(define (area a-shape)
  (cond [(dot? a-shape) (dot-area a-shape)]
        [(square? a-shape) (square-area a-shape)]
        [(circle? a-shape) (circle-area a-shape)]))
```

When the function is applied to a shape, the **cond** expressions evaluates the three conditions, one by one. When one of them evaluates to true, the right-hand side expression of this **cond** line determines the answer for the entire **cond** expression. In short, where Scheme relies on conditionals to distinguish different classes of data, object-oriented languages use polymorphic method dispatch. From a computational perspective, this difference is the most important one.

13.2   The Role of Types

The preceding sections shows that types don't play any role during the evaluation of programs. The part that matters is the class[21] from which an object was instantiated. Thus, you may wonder what good they are. Let's briefly look back at *How to Design Programs*, which assumes that you produce code for yourself. The implication is that since you know what you are doing, nobody ever violates the contracts of your programs. You *always* apply your functions to exactly those kinds of data that the contract specifies, and all of your functions use the results of functions they call in the appropriate manner. As a matter of fact, the assumption is that you even use functions according to their purpose statements.

If people really acted this way, no program would ever need any safeguards. You know from experience, however, that people *do* violate contracts all the time. It is likely that you did, when you worked on some of the exercises in *How to Design Programs*. You may have violated the contracts of DrScheme's library functions. Worse, you may have forgotten about the contract of a function while you were turning its template into a complete function body. Or, you may have abused a function when you used it weeks after you first designed it. Now imagine working with other people on the same program, each of you producing a fragment of such as a class or a couple of classes, with the team eventually making these fragments work together. Or imagine being told to change some program

---

[21] In Java, **class** plays both the role of a type and data label for method dispatch. These two roles are related but they are not the same. Try not to confuse them.

fragment weeks, months or years after you first wrote them. It is just natural that people fail to respect contracts in such situations.

One purpose of types is to help overcome this problem. You write them down explicitly so that others can read them. In contrast to informal contracts, types must obey the rules of the programming languages and they are checked before you run the program. Indeed, you can't run the program if the type check fails. Other programmers (including an older "you") can then read these types with the knowledge that their use is consistent with the language's rules.

Consistency for types is similar to consistency of a program's organization with the language's grammar, something you know from *How to Design Programs*. If you write down a **define** with four pieces, something is wrong, and DrScheme signals a syntax error. In the same spirit, Java checks a program's use of types once it has checked that it obeys the grammar of the language. Java's type checker ensures that the uses of field identifiers and method parameters match their declared types. From this information, it computes a type for each expression and sub-expression in your program and always matches specified types with actual types:[22]

```
int maxLength(String s) {
  if ( s.length() ₁ >  80 ₂) {
    return s.length() ₃; }
  else {
    return 80; }
}
```

The gray-shaded expression with subscript 1 has type int, as does the gray-shaded expression with subscript 2. Since the primitive operator $<$ compares two ints and then produces a boolean, it is acceptable to use the comparison expression as the test component of an **if** statement. The gray-shaded expression labeled with 3 is the same as the expression labeled 1. Thus, no matter which of the two **return**s is used, the **if** statement—and thus the method body—always **return**s an int, just like the method signature claims. Hence the types match and the method signature correctly describes the data that the method consumes and produces.

---

[22]In sound type systems, such as Java's, type checking also implies that the result of an int expression is always (a representation of) an integer in the computer's hardware. In unsound type systems, such as the one of C++, this is not the case. While such unsound systems can still discover potential errors via type checking, you may not rely on the type checker's work when something goes wrong during your program's evaluation.

You can also ask in this context why expression 1 has type int. In this case, we know that *s* has type *String*. According to figure 37, the *length* method consumes a *String* and produces an int. In other words, the type checker can use a method's signature independently of its body to check method calls.

As the type checker performs these checks for your program, it may encounter inconsistencies. For example, if we write

```
int maxLength(String s) {
  if (s > 80)
    return s.length();
  else
    return 80;
}
```

s once again has type *String*. Its use with > (in the gray-shaded expression) conflicts with the type specification for the operator, which says that the operands have to be of type int. Your Java implementation therefore signals a type error and asks you to take a closer look at your use of *s*. In this particular case, we know for sure that comparing a string to a number wouldn't work during the evaluation and trigger an error. (Try it in DrScheme.) In general, you should think of a type checker as a spell checker in your word processor; when it finds a spelling error, the spelling is wrong, inappropriate and leads to difficulties in understanding, or intended and the spell checker is too simplistic to discover this rare scenario.

While spell checkers do find some basic mistakes, they also miss problems. Similarly, just because your program passed the type checker, you should not assume that it is correct. Do you remember how often your spell checker agreed that "there" was the correct spelling when you really meant "their" in your essays? As you proceed, keep in mind that grammar- and type-checking your programs eliminates errors at the level of typos and ill-formed sentences.[23] What they usually do not find are flaws that are comparable to problems with your chain of reasoning, the inclusion of unchecked statistics, etc. To avoid such problems, you must design your programs systematically and stick to a rigorous design discipline.

---

[23] Type systems in conventional languages can't even check some of our simple informal contracts. For example, if a function consumes a number representing a width and produces an area, we may write "*PositiveNumber* → *PositiveNumber*." Type systems usually do not include such subsets of numbers.

14   Methods and Unions of Classes (Continued)

14.1   How Libraries Work, Part 1: Drawing Geometric Shapes

The four problems concerning geometric shapes clearly call for a fifth one: a method for drawing a shape into a canvas. Of course, drawing shapes on a canvas is something the computer must do for us; it is not the task of the programming language per se. Programming languages provides libraries—such as the already mentioned **geometry** and **colors** libraries— that bridge the gap between programs and the computer.

Java provides several drawing libraries. ProfessorJ provides a refined version that matches our goal of learning to design classes systematically. More precisely, ProfessorJ provides the *Canvas* class via the **draw** library, which provides methods for drawing shapes (including circles, rectangles, lines, and strings) onto a visible computer canvas. The relevant excerpts from *Canvas* are displayed in figure 50.

You should note how the method signatures and purpose statements in this figure tell you how to use an existing class and its methods, without knowledge of the actual method definitions. To use the libraries, recall to add the following lines to your definitions window and to run this "program:"

> **import** *draw.∗*;
> **import** *colors.∗*;
> **import** *geometry.∗*;

Then you can instantiate *Canvas* in the interactions window by just supplying a width and a height:

> *Canvas c* = **new** *Canvas*(100,100);

Think of this as a field declaration with an initialization equation. Following this line, *c* is available in the interactions window and stands for a *Canvas*. In particular, you can invoke *show* on *c*:

> *c.show*()

so that the *Canvas* becomes visible. If the computer can show the window, the expression produces true; otherwise, you will encounter an error.

Once you have a canvas, you can easily place a *String* in it. Experiment the following pair of field declarations and pair of expressions in ProfessorJ's interactions window:

> *c.drawString*(**new** *Posn*(10,50), "hello world")

```
// controlling a computer canvas
class Canvas {
  int width;
  int height;
  . . .
  // show this a canvas
  boolean show()

  // draw a circle at p in this canvas
  boolean drawCircle(Posn p, int r, IColor c)

  // draw a solid disk at p in this canvas,
  // fill with color c
  boolean drawDisk(Posn p, int r, IColor c)

  // draw a width x height rectangle
  // at p in this canvas, fill with color c
  boolean drawRect(Posn p,int width,int height, IColor c)

  // draw s at position p in this canvas
  boolean drawString(Posn p, String s)
  . . .
}
```

Canvas
int *width*
int *height*
boolean *show*()
boolean *close*()
boolean *drawCircle*(*Posn*, int, *IColor*)
boolean *drawDisk*(*Posn*, int, *IColor*)
boolean *drawLine*(*Posn*, int, int, *IColor*)
boolean *drawString*(*Posn*, String)
...

Figure 50: The drawing methods in the *Canvas* class

Again, the expression produces true if the drawing action succeeds; if not, the computer will signal an error.

Exercises

Exercise 14.1 Use the libraries you have come to know (**colors**, **draw**, **geometry**) to draw (1) a box-and-circles car, (2) a match-stick man, and (3) a house in ProfessorJ's interactions window. ∎

Exercise 14.2 Develop the class *HouseDrawing*. Its constructor should determine the size of the house (width, height) and *Canvas*. Since you haven't encountered this dependence, we provide the basics:

```
class HouseDrawing {
  int width;
  int height;
  Canvas c;
  IColor roofColor = new Red();
  IColor houseColor = new Blue();
  IColor doorColor = new Yellow();

  HouseDrawing(int width, int height) {
    this.width = width;
    this.height = height;
    this.c = new Canvas(width, height);
  }
  ...
}
```

As always, the constructor for *HouseDrawing* contains one "equation" per field (without initialization "equation") but it lacks a parameter for the *Canvas*. Instead, the canvas is constructed with the help of the other parameters, *width* and *height*. This is one way in which defining your own constructor is superior to having defined it automatically.

The class should also come with a *draw* method whose purpose it is to draw an appropriately sized house onto the canvas. In other words, the house's measurements should depend on the *width* and *height* fields.

We suggest you start with a house that has a red, rectangular roof; a somewhat smaller blue, rectangular frame; a yellow door, and a couple of yellow windows. Once you can draw that much, experiment some more. ∎

Now suppose you are to add a method *show* to the *Room* for drawing the room. More precisely, the method should use a canvas to visually present the room and the furniture inside the room. Since drawing should happen on a canvas, the *Room* class needs a *Canvas* in addition to the *show* method:

```
// to keep track of the pieces of furniture in this room
class Room {
  ...
  Canvas c;

  Room(int width, int height) {
    ...
    this.c = new Canvas(this.width, this.height);
  }
  ...
}
```

We have chosen to create a canvas that is as large as the room itself. Just as in exercise 14.2, the *Canvas* is instantiated in the constructor because it depends on the values of *width* and *height*, but it is not a parameter of the constructor itself.

The design of *show* follows the design recipe, but requires one extra thought. Before it can display the room, it must show the canvas, i.e., it must invoke *c*'s *show* method:

inside of *Room* :
// show the world (the room) with its furniture
boolean *show*() {
  return this.*c.show*()
    && **this**.*a.draw*(. . .) *&&* **this**.*b.draw*(. . .) *&&* **this**.*c.draw*(. . .);
}

Then it delegates the tasks of drawing the three pieces of furniture to an imaginary method *draw* in *IShape*. As you can see, the second like of the template strictly follows from the design recipe.

The "imaginary" part means, of course, that we are adding this method to our wish list. In this case, the wish can go directly into the *IShape* interface:

inside of *IShape* :
// draw **this** shape into canvas
boolean *draw*(*Canvas c*);

The *Canvas* parameter is needed because the *draw* method needs access to it but the *Canvas* is only a part of the *Room* class. In other words, the *show* method from *Room* must communicate *c*, the *Canvas*, to the *draw* methods as an argument.

There is no true need for functional examples. We know that a *draw* method in *Square* should draw a square at the appropriate position and of

the appropriate proportions. The same is true for *Circle*s. Of course, thinking about examples does reveal that drawing a *Dot* presents the special-case problem again; let's just settle for drawing a disk with radius 3 to make it visible.

For the template step, we can reuse the template from figure 46.

inside of *Dot* :	inside of *Square* :	inside of *Circle* :
boolean	boolean	boolean
draw(*Canvas c*) {	*draw*(*Canvas c*) {	*draw*(*Canvas c*) {
... **this**.*loc.nnn*() ...	... **this**.*loc.nnn*() ...	... **this**.*loc.nnn*() ...
}	... **this**.*size* ...	... **this**.*radius* ...
	}	}

The first expression in each of these templates suggests that we can wish for a new method for *CartPt*, which is where *loc* comes from. If *loc* were a *Posn*, the template would translate itself into method bodies. All it would take is an invocation of, say, *drawCircle* on the position, the radius, and some color.

Put differently, we have a choice with two alternatives. We can either replace *CartPt* with *Posn* throughout our existing program or we can equip *CartPt* with a method that creates instances of *Posn* from instances of *CartPt*. Normally, the reuse of library classes is preferable; here we just provide the method because it is so straightforward and because it demonstrates how to bridge the small gap between the library and the rest of the code:

inside of *Dot* :	inside of *Square* :	inside of *Circle* :
boolean	boolean	boolean
draw(*Canvas c*) {	*draw*(*Canvas c*) {	*draw*(*Canvas c*) {
return	**return**	**return**
c.drawDisk(	*c.drawRect*(	*c.drawCircle*(
this.*loc.toPosn*(),	**this**.*loc.toPosn*(),	**this**.*loc.toPosn*(),
1,	**this**.*size*,	**this**.*radius*,
new *Green*());	**this**.*size*,	**new** *Red*());
}	**new** *Blue*());	}
	}	

What's left to do is to design *toPosn* in *CartPt*:

inside of *CartPt* :
Posn toPosn() {
  return new *Posn*(**this**.*x*, **this**.*y*);
}

It is so simple that we provide the definition instead of going through the process. All that you must do now is run some examples to ensure that *draw* truly draws shapes at the expected positions. Remember these aren't tests because you can't write statements that automatically compare the expected outcome with the actual outcome.

Exercises

Exercise 14.3 Complete the definition of the *Room* class. ∎

Exercise 14.4 Add isosceles right triangles to the collection of furniture shapes (see also exercise 12.3). ∎

Exercise 14.5 Modify *show* in *Room* so that it also draw a 20-point, black margin. ∎

14.2   Finger Exercises

Exercise 14.6 Recall exercise 4.5:

> . . . Develop a program that creates an on-line gallery from three different kinds of records: images (gif), texts (txt), and sounds (mp3). All have names for source files and sizes (number of bytes). Images also include information about the height, the width, and the quality of the image. Texts specify the number of lines needed for visual representation. Sounds include information about the playing time of the recording, given in seconds. . . .

 Develop the following methods for this program:

1. *timeToDownload*, which computes how long it takes to download a file at some given network connection speed (in bytes per second);

2. *smallerThan*, which determines whether the file is smaller than some given maximum size;

3. *sameName*, which determines whether the name of a file is the same as some given name. ∎

Exercise 14.7 A software house that is working with a grocery chain receives this problem statement:

> ... Develop a program that keeps track of the items in the grocery store. For now, assume that the store deals only with ice cream, coffee, and juice. Each of the items is specified by its brand name (*String*), weight (grams) and price (cents). Each coffee is also labeled as either regular or decaffeinated. Juice items come in different flavors, and can be packaged as frozen, fresh, bottled, or canned. Each package of ice cream specifies its flavor. ...

Design the following methods:

1. *unitPrice*, which computes the unit price (cents per gram) of a grocery item;

2. *lowerUnitPrice*, which determines whether the unit price of a grocery item is lower than some given amount;

3. *cheaperThan*, which determines whether a grocery item's unit price is less than some other (presumably) comparable item's unit price. ∎

Exercise 14.8 Consider this revision of our running example concerning book stores:

> ... Develop a program that assists managers of discount bookstores. The program should keep a record for each book. The record must include its title, the author's name, its price, and its publication year. There are three kinds of books with different pricing policy. Hardcover books are sold at 20% off. Books on the sale table are 50% off. Paperbacks are sold at list price. ...

 Here are your tasks:

1. Develop a class hierarchy that represents books.

2. Draw a class diagram for the hierarchy.

3. Create five sample objects.

4. Design the following methods:

    (a) *salePrice*, which computes the sale price of each book;
    (b) *sameAuthor*, which determines whether a book is by a given author. ∎

14.3 Designing Methods for Unions of Classes

It is time again to reflect on our design recipe for methods (see sections 10.3 and 11.2). The five steps work well in principle, but clearly the organization of some classes as a union suggests additional checkpoints:

1. Formulate a purpose statement and a method signature to the interface; then add the method signature to *each* implementing class.

2. Illustrate the purpose statement with examples for *each* class of the union, i.e., for each variant.

3. Lay out what you know about the method's argument(s) in each concrete method. This includes references to the fields of the class.

   Remember from section 11.2 that if any of the variants contain an instance of another class, you should place appropriate schematic method calls to methods in these other classes in the template. The purpose of these schematic calls is to remind you of the wish list during the next step. To this end, also add a tentative method header to that other class.

4. Define the methods. The parameters and the expressions in the template represent the information that may contribute to the result. Each schematic method call means that you may need to design an auxiliary method in some other class. Use the wish list to keep track of these auxiliary goals.

5. Turn the examples into tests and run them.

As this section showed, you can switch steps 1 and 3. That is, you can develop the templates just based on the structure of the class hierarchy and the classes themselves. The example in this section also showed how helpful the template-directed design is. Once you understand the template, the rest of the design task is often straightforward.

15 Methods and Classes with Mutual References

15.1 Example: Managing a Runner's Logs

Recall the problem of tracking a runner's workouts:

... Develop a program that manages a runner's training log. Every day the runner enters one entry concerning the day's run. ...

```
interface ILog {
  ??? nnn();
}
```

```
class MTLog implements ILog {
  MTLog() {}

  ??? nnn() {
    ...
  }
}
```

```
class Entry {
  Date d;
  double distance; // miles
  int duration; // minutes
  String comment;

  Entry(Date d,
        double distance,
        int duration,
        String comment) {
    ... // omitted ...
  }

  ??? mmm() {
    ... this.d.lll() ...
    ... this.distance ...
    ... this.duration ...
    ... this.comment ...
  }
}
```

```
class ConsLog implements ILog {
  Entry fst;
  ILog rst;

  ConsLog(Entry fst, ILog rst) {
    ... // omitted ...
  }

  ??? nnn() {
    ... this.fst.mmm() ...
    ... this.rst.nnn() ...
  }
}
```

```
class Date {
  int day;
  int month;
  int year;

  Date(int day, int month, int year) {
    ... // omitted ...
  }

  ??? lll() {
    ... this.day ...
    ... this.month ...
    ... this.year ...
  }
}
```

Figure 51: Classes for a runner's log

Figure 18 (page 38) displays the class diagram for a data representation of this problem. Figure 51 contains the matching class definitions. A log

is either empty, which we represent with an instance of *MTLog*, or it is constructed from an *Entry* and a log. An instance of *Entry* contains four fields: a *Date d*, a double for the *distance* in miles, an int for the *duration* in minutes, and a *String* for recording qualitative remarks.

In addition, figure 51 contains sketches of method templates: a signature for a method *nnn* in *ILog* and concrete templates for *MTLog*, *ConsLog*, *Entry*, and *Date*. Later we must refine these templates with return types and arguments.

The templates in *MTLog*, *Entry*, and *Date* are unremarkable; their design strictly follows the design recipe, especially the ones for unions and classes that refer to other classes. While this is also true for the template in *ConsLog*, that template contains the expression

> ... **this**.*rst.nnn*() ...

which refers to the method *nnn* in *ILog*. Since *rst* is a reference to some other box in the class diagram, this basically follows from the design recipe for classes that contains instances of other classes. *ILog* isn't an ordinary class, however, but an interface. Still, we know that if an interface specifies a method, then all implementing classes define this method. Therefore it is proper to reference *nnn* in this context. Indeed, by doing so, the method template matches the class diagram in a perfect manner; the class hierarchy has determined the shape of the program one more time.

From section 13 we also know what this means for the evaluation of such expressions. The value in *rst* is going to be either an instance of *MTLog* or *ConsLog*. In each case, the method call is directed to the concrete method in the corresponding class. If *rst* is an instance of *ConsLog*, the method calls itself; otherwise it calls *nnn* in *MTLog*.

Note: If you have experienced *How to Design Programs* properly, you know not to look ahead. Just trust the design recipe and its claim that methods work according to their purpose statement, both other methods (i.e., those on the wish list) and the same method (recursively). If you haven't worked through *How to Design Programs*, we need to ask you to trust us for the moment. Once you experience for yourself how smoothly the design works out, you will know why we asked you to just plunge ahead. ∎

Let's see whether working out the template first works as well for self-referential class diagrams as it did for unions. Here is a first problem:

> ... The runner will want to know the total number of miles run. ...

Here is an appropriate refinement of the method signature in *ILog*:

```
class CompositeExamples {
  Date d1 = new Date(5, 5, 2003);
  Date d2 = new Date(6, 6, 2003);
  Date d3 = new Date(23, 6, 2003);

  Entry e1 = new Entry(d1, 5.0, 25, "Good");
  Entry e2 = new Entry(d2, 3.0, 24, "Tired");
  Entry e3 = new Entry(d3, 26.0, 156, "Great");

  ILog l1 = new MTLog();
  ILog l2 = new ConsLog(e1,l1);
  ILog l3 = new ConsLog(e2,l2);
  ILog l4 = new ConsLog(e3,l3);
  CompositeExamples() { }
}
```

Figure 52: Examples for a runner's log

inside of *ILog* :
// to compute the total number of miles recorded in **this** log
double *miles*();

In addition, you can now rename the methods in *MTLog* and *ConsLog*.

For the functional examples, we use the sample objects from our original discussion of a runner's log: see figure 52, where they are repeated. Invoking the *miles* method on the *ILog*s produces the obvious results:

check *l1.miles*() **expect** 0.0 **within** .1
check *l2.miles*() **expect** 5.0 **within** .1
check *l3.miles*() **expect** 8.0 **within** .1
check *l4.miles*() **expect** 34.0 **within** .1

We use the examples to design each concrete method, case by case:

1. The examples show that in *MTLog* the method just returns 0.0. A log with no entries represents zero miles.

2. The template for *ConsLog* contains two expressions. The first says that we can compute with *fst*, which is an instance of *Entry*. The second one computes **this**.*rst.miles*(). According to the purpose statement in *ILog*, this expression returns the number of miles in the *rst* log that is included in **this** instance of *ConsLog*. Put differently, we can just add the number of miles in the *fst Entry* to those from *rst*.

Using this problem analysis, it is easy to write down the two methods:

inside of *MTLog* :	inside of *ConsLog* :
double *miles*() {	double *miles*() {
return 0;	**return this**.*fst.distance* + **this**.*rst.miles*();
}	}

All that remains to be done is to test the methods with the examples.

In light of our discussion on the differences between Scheme-based computations and a Java-based one, it is also important to see that this method looks very much like the addition function for lists:

```
(define (add-lists-of-numbers alon)
  (cond [(empty? alon) 0]
        [else (+ (first alon) (add-lists-of-numbers rest alon))]))
```

The expression from the first conditional branch shows up in *MTLog* and the one from the second branch is in the method for *ConsLog*, which is just as it should be. The conditional itself is invisible in the object-oriented program, just as described on page 13.1:

Often a runner doesn't care about the entire log from the beginning of time but a small portion of it. So it is natural to expect an extension of our problem with a request like this:

> . . . The runner will want to see his log for a specific month of
> his training season. . . .

Such a portion of a log is of course itself a log, because it is also a sequence of instances of *Entry*.

Put differently, the additional method consumes a runner's log and two integers, representing a month, and a year. It produces a runner's log—of one month's worth of entries:

inside of *ILog* :
// to extract those entries in **this** log for the given month and year
ILog oneMonth(int *month*, int *year*);

check *l1.oneMonth*(6, 2003) **expect** *l1*
check *l3.oneMonth*(6, 2003) **expect new** *ConsLog*(*e2*, *MTLog*)
check *l3.oneMonth*(6, 2003)
expect new *ConsLog*(*e3*, **new** *ConsLog*(*e2*, *MTLog*()))

As before, the examples are based on the sample logs in figure 52. The first one says that extracting anything from an empty log produces an empty

log. The second one shows that extracting the June entries from *l2* gives us a log with exactly one entry. The last example confirms that we can get back a log with several entries.

The examples suggest a natural solution for *MTLog*. The design of the concrete method for *ConsLog* requires a look at the template to remind ourselves of what data is available. The second method call in the template,

> **this**.*rst.oneMonth*(*month*, *year*)

produces the list of entries made in the given month and year extracted from the rest of the log. The other one,

> **this**.*fst.mmm*(*month*, *year*)

deals with *fst*, i.e., instances of *Entry*. Specifically it suggests that we may wish to design a separate method for computing some value about an instance of *Entry*. In this case, *Entry* needs a method that determines whether an instance belongs to some given month and year, because the *oneMonth* method should include *fst* only if it belongs to the given month and year.

To avoid getting distracted, we add an entry on our wish list:

inside of *Entry* :
// was **this** entry made in the given month and year?
boolean *sameMonthAndYear*(int *month*, int *year*) { … };

But before we design this method, let's finish *oneMonth* first.

Assuming that *oneMonth* is designed properly and works as requested, we can finish the method in *ConsLog* easily:

inside of *MTLog* :
```
ILog oneMonth(int m, int y) {
  return new MTLog();
}
```

inside of *ConsLog* :
```
ILog oneMonth(int m, int y) {
  if (this.fst.sameMonthAndYear(m, y)) {
    return
      new
        ConsLog(
          this.fst,
          this.rst.oneMonth(m, y)); }
  else {
    return this.rst.oneMonth(m, y); }
}
```

The method in *ConsLog* must distinguish two possibilities. If

> **this**.*fst.sameMonthAndYear*(*m*, *y*)

is false, the result is whatever *oneMonth* extracted from *rst*. If it is true, *fst* is included in the result; specifically, the method creates a new *ConsLog* from *fst* and whatever *oneMonth* extracts from *rst*.

  With the methods for *MTLog* and *ConsLog* completed, we turn to our wish list. So far it has one item on it: *sameMonthAndYear* in *Entry*. Its method template (refined from figure 51) is:

inside of *Entry* :
boolean *sameMonthAndYear*(int *month*, int *year*) {
  ... **this**.*d.lll*() ... **this**.*distance* ... **this**.*duration* ... **this**.*comment* ...
}

This implies that the method should calculate with *month*, *year*, and *d*, the *Date* in the given *Entry*. The suggestive method call **this**.*d.lll*() tells us that we can delegate all the work to an appropriate method in *Date*. Of course, this just means adding another item to the wish list:

inside of *Date* :
// is **this** date in the given month and year?
boolean *sameMonthAndYear*(int *month*, int *year*) { ... }

Using "wishful thinking" gives us the full definition of *sameMonthAndYear*:

inside of *Entry* :
boolean *sameMonthAndYear*(int *month*, int *year*) {
  return this.*d.sameMonthAndYear*(*month*, *year*);

  The one thing left to do is to design *sameMonthAndYear* for *Date*. Naturally, we start with a refinement of its template:

inside of *Date* :
boolean *sameMonthAndYear*(int *month*, int *year*) {
  ... **this**.*day* ... **this**.*month* ... **this**.*year* ...
}

This template tells us that *sameMonthAndYear* has five numbers to work with: *month*, *year*, **this**.*day*, **this**.*month*, and **this**.*year*. Given the purpose statement, the goal is clear: the method must compare *month* with **this**.*month* and *year* with **this**.*year*:

inside of *Date* :
// is **this** date in the given month and year?
boolean *sameMonthAndYear*(int *month*, int *year*) {
  return (**this**.*month* == *month*) && (**this**.*year* == *year*);
}

This finishes our wish list and thus the development of *oneMonth*. You should realize that we again skipped making up examples for the two "wishes." While this is on occasion acceptable when you have a lot of experience, we recommend that you develop and test such examples now.

Exercises

Exercise 15.1 Collect all the pieces of *oneMonth* and insert the method definitions in the class hierarchy for logs. Develop examples and include them with the test suite. Draw the class diagram for this hierarchy (by hand). ∎

Exercise 15.2 Suppose the requirements for the program that tracks a runner's log includes this request:

> . . . The runner wants to know the total distance run in a given month. . . .

Design the method that computes this number and add it to the class hierarchy of exercise 15.1.

Consider designing two different versions. The first should follow the design recipe without prior considerations. The second should take into account that methods can compose existing methods and that this particular task can be seen as consisting of two separate tasks. (The design of each method should still follow the regular design recipe.) Where would you put the second kind of method definition in this case? (See the next chapter.) ∎

Exercise 15.3 Suppose the requirements for the program that tracks a runner's log includes this request:

> . . . A runner wishes to know the length of his longest run ever.
> [He may eventually wish to restrict this inquiry into a particular season or runs between two dates.] . . .

Design the method that computes this number and add it to the class hierarchy. Assume that the method produces 0 if the log is empty.

Also consider this variation of the problem:

> . . . A runner wishes to know whether all distances are shorter than some number of miles. . . .

Does the template stay the same? ∎

15.2   Example: Sorting

Sorting logs is a natural idea, too, in a day and age when everything is measured, and everything is ranked:

> . . .   The runner wishes to see the log ordered according to the distance of each run, starting with the longest. . . .

As usual, we start with a choice of name, signature and purpose statement for the problem:

inside of *ILog* :
// to create a sorted version of **this** log, with entries sorted by distance
ILog sortByDist();

The method consumes a log and produces one sorted by distance.

We use the now familiar sample logs from figure 52 to create examples for this method:

check *l1.sortByDist*() **expect** *l1*
check *l2.sortByDist*() **expect** *l2*
check *l4.sortByDist*() **expect**
new *ConsLog(e2*,
     new *ConsLog(e1*,
        new *ConsLog(e3*, **new** *MTLog*())))

The first two are a bit odd; the result is identical to the "input" because the given logs are already sorted. The last one shows how the method rearranges the entries when needed.

Here are the two templates refined for our new purpose:

inside of *MTLog* :	inside of *ConsLog* :
ILog sortByDist() {	*ILog sortByDist*() {
. . .	. . . **this**.*fst.mmm*() . . .
}	. . . **this**.*rst.sortByDist*() . . .
	}

The method templates suggest how to design both methods. For *sortByDist* in *MTLog*, the result is the empty log again. For *sortByDist* in *ConsLog*, the method call

this.*rst.sortByDist*()

produces a sorted list of all entries in the rest of the log. That means we only need to insert the first entry into the sorted version of *rst* to obtain the sorted log that corresponds to the given one.

Following our wish list method, we faithfully add this "insert" method to our list:

inside of *ILog* :
// insert the given entry into **this** (sorted) log
ILog insertDist(*Entry e*);

The difference between the previous example and this one is that we are adding this wish list item to the very same class (hierarchy) for which we are already designing *sortByDist*.

Now we can use *insertDist* in *sortByDist*:

inside of *MTLog* :	inside of *ConsLog* :
ILog sortByDist() {	*ILog sortByDist*() {
return this;	**return this**.*rst.sortByDist*().*insertDist*(**this**.*fst*);
}	}

Specifically, the method first invokes *sortByDist* on *rst* and then invokes *insertDist* on the result. The second argument to *insertDist* is **this**.*fst*, the first *Entry* in the given log.

Now that we're done with *sortByDist*, we turn to our wish list, which contains *insertDist* in *ILog*. We immediately move on to the development of a good set of functional examples, starting with a new data example that contains three entries:

ILog l5 =
  new *ConsLog*(**new** *Entry*(**new** *Date*(1,1,2003), 5.1, 26, "good"),
    new *ConsLog*(**new** *Entry*(**new** *Date*(1,2,2003), 4.9, 25, "okay"),
      new *MTLog*()))

Because the addition of another instance of *Entry* to *l5* can take place at three distinct places, we develop three distinct examples:

1. The first example shows that the given *Entry* might end up in the middle of the given log:

   check *l5.insertDist*(**new** *Entry*(**new** *Date*(1,3,2003), 5.0, 27, "great"))
   expect
     new *ConsLog*(**new** *Entry*(**new** *Date*(1,1,2003), 5.1, 26, "good"),
       new *ConsLog*(**new** *Entry*(**new** *Date*(1,3,2003), 5.0, 27, "great"),
         new *ConsLog*(**new** *Entry*(**new** *Date*(1,2,2003), 4.9, 25, "okay"),
           new *MTLog*())))

2. In the second case, the given *Entry* is the first *Entry* of the resulting log:

    check *l5.insertDist*(**new** *Entry*(**new** *Date*(1,4,2003), 5.2, 24, `"fine"`))
    expect
      new *ConsLog*(**new** *Entry*(**new** *Date*(1,4,2003), 5.2, 24, `"fine"`),
        new *ConsLog*(**new** *Entry*(**new** *Date*(1,1,2003), 5.1, 26, `"good"`),
          new *ConsLog*(**new** *Entry*(**new** *Date*(1,2,2003), 4.9, 25, `"okay"`),
            new *MTLog*()))))

3. Finally, the third example explains that the Entry can also be the last one in the result:
       check *l5.insertDist*(**new** *Entry*(**new** *Date*(1,5,2003), 4.8, 23, `"bad"`))
       expect
         new *ConsLog*(**new** *Entry*(**new** *Date*(1,1,2003), 5.1, 26, `"good"`),
           new *ConsLog*(**new** *Entry*(**new** *Date*(1,2,2003), 4.9, 25, `"okay"`),
             new *ConsLog*(**new** *Entry*(**new** *Date*(1,5,2003), 4.8, 23, `"bad"`),
               new *MTLog*()))))

Still, in all cases, the resulting logs are sorted in the descending order of distances run.

    Let us look at the complete definitions:

```
class MTLog implements ILog {        class ConsLog implements ILog {
  ⋮                                    Entry fst;
                                       ILog rst;

                                       ⋮

  ILog insertDist(Entry e){            ILog insertDist(Entry e){
    return                               if (e.distance > this.fst.distance) {
      new ConsLog(e, this);                return new ConsLog(e, this); }
  }                                      else {
}                                          return
                                             new ConsLog(this.fst,
                                               this.rst.insert(e))); }
                                       }
                                     }
```

The method on the left must return a log with one *Entry* because the purpose statement promises that the given *Entry* and all the *Entry*s in the given log show up in the result. Since the given log is an instance of *MTLog*, the result must be the log that consists of just the given *Entry*.

    The method on the right must distinguish two cases. If the distance in the given *Entry* is larger than the distance in *fst*, it is larger than all the distances in the given log, and therefore the given *Entry* must show up at

the beginning of the result. If the distance in the given *Entry* is less than (or equal to) the distance in *fst*, the recursive method call inserts the given *Entry* in *rst* and the method just adds *fst* to the result of this recursive call.

Exercises

Exercise 15.4 Suppose the requirements for the program that tracks a runner's log includes this request:

> ... The runner would like to see the log with entries ordered according to the pace computed in minutes per mile in each run, from the fastest to the slowest. ...

Design this sorting method. ∎

15.3   Example: Overlapping Shapes

Recall the problem of representing overlapping shapes (page 44), which we discussed in section 5.2. The data representation that we chose is displayed in figure 23 (the class diagram) and in figure 24 (the class definitions). In this section, we design some methods to these interfaces and classes, based on problems from a programming contest for researchers.

As with the preceding examples, we start with the addition of templates to the classes. The templates for *Dot*, *Square*, and *Circle* are routine affairs. As for *SuperImp*, the two containment arrows from *SuperImp* to *IComposite* in the class diagram suggest that we need two method calls to the method template in *IComposite*. After all, a method that processes a *SuperImp* may have to process both pieces, and they are *IComposite* shapes.

Figure 53 contains the class definitions enriched with the templates; to keep the example reasonably small, *Dot* and *CartPt* are omitted as are the schematic calls to the latter. Furthermore, figure 54 presents several examples, because as we have seen now many times, it is always good to have a small set of examples around. Take a look at the four instances of *SuperImp*; they illustrate the kind of problem we are facing well. The first two just combine *Square*s and *Circle*s; the third shows that a *SuperImp* may also contain another *SuperImp*; and the last one contains not one but two *SuperImp*s.

With the template and the examples in place, we can turn to an imaginary programming problem from the above-mentioned contest:

> ... A combination of shapes represents a formation of warships. The commanding officer wants to know how close the formation is to some target. ...

```
interface IComposite {
  ??? nnn();
}
```

```
class Square                class Circle                 class SuperImp
  implements                  implements                   implements
   IComposite {                IComposite {                 IComposite {
CartPt loc;                  CartPt loc;                  IComposite bot;
int size;                    int radius;                  IComposite top;

Square(CartPt loc,          Circle(CartPt loc,          SuperImp(
       int size){                  int radius){                IComposite bot,
  this.loc = loc;             this.loc = loc;               IComposite top) {
  this.size = size;           this.radius = radius;         this.bot = bot;
}                           }                               this.top = top;
                                                          }

??? nnn() {                 ??? nnn() {                  ??? nnn() {
  ... this.loc ...            ... this.loc ...             ... this.bot.nnn() ...
  ... this.size ...           ... this.radius ...          ... this.top.nnn() ...
}                           }                             }
}                           }                             }
```

Figure 53: Classes for combination shapes, with templates

While this formulation of the problem is quite different from the original,
rather plain formulation on page 118 for the collection of basic shapes, it
is easy to recognize that they are the basically the same. The origin is the
target, and the distance to the origin is the distance to the target. If we
continue to assume that the shape is entirely on the canvas, the signature
and the purpose statement carry over from the original problem to this one:

> inside of *IComposite* :
> // to compute the distance of **this** shape to the origin
> double *distTo0*();

Indeed, the concrete methods for *Square* and *Circle* also remain the same.
The difference is the concrete method for *SuperImp*, which must compute
the distance of an entire combination of shapes to the origin.

For the classes *Square* and *Circle* the expected results are computed the
same was as those we saw earlier: see *testS1*, *testS2*, *testC1*, and *testC2* in fig-

```
class CompositeExamples {
  IComposite s1 = new Square(new CartPt(40, 30), 40);
  IComposite s2 = new Square(new CartPt(120, 50), 50);
  IComposite c1 = new Circle(new CartPt(50, 120), 20);
  IComposite c2 = new Circle(new CartPt(30, 40), 20);
  IComposite u1 = new SuperImp(s1, s2);
  IComposite u2 = new SuperImp(s1, c2);
  IComposite u3 = new SuperImp(c1, u1);
  IComposite u4 = new SuperImp(u3, u2);

  boolean testS1 = check s1.distTo0() expect 50.0 within .1;
  boolean testS2 = check s2.distTo0() expect 80.0 within .1;
  boolean testC1 = check c1.distTo0() expect 110.0 within .1;
  boolean testC2 = check c2.distTo0() expect 30.0 within .1;

  CompositeExamples() { }
}
```

Figure 54: Examples for combination shapes

ure 54. The instances of *SuperImp* make up the interesting examples. Given that a *SuperImp* contains two shapes and that we wish to know the distance of the closer one to the shape, we pick the smaller of the two distances:

> **check** *u1.distTo0()* **expect** 50.0 **within** .1
> **check** *u2.distTo0()* **expect** 30.0 **within** .1
> **check** *u3.distTo0()* **expect** 50.0 **within** .1
> **check** *u4.distTo0()* **expect** 30.0 **within** .1

The distance of *u2* to the origin is 30.0 because the *Square s1* is 50.0 pixels away and the *Circle c2* is 30.0 pixels away. Convince yourself that the other predicted answers are correct; draw the shapes if you have any doubts.

Our reasoning about the examples and the template for *SuperImp* imply that the method just computes the distance for the two shapes recursively and then picks the minimum:

> inside of *SuperImp* :
> ```
> double distTo0(){
>     return Math.min(this.bot.distTo0(), this.top.distTo0());
> }
> ```

As suggested by its name, *Math.min* picks the smaller of two numbers.

Here is a related problem (from the same contest):

> ... Assuming the shapes represent those points that are reach-
> able with anti-aircraft missiles, the commanding officer wishes
> to know whether some point in the Cartesian space falls within
> the boundaries of the formation's outline. ...

If you prefer a plainer problem statement, see page 119 for the analogous problem for basic shapes.

Clearly the methods for *Square* and *Circle* can be used "as is." The purpose statement and header from *IShape* can also be transplanted into the new interface:

inside of *IComposite* :
// is the given point within the bounds of **this** shape
boolean *in*(*CartPt p*);

inside of *CompositeExamples* :
check *u1.in*(**new** *CartPt*(42,42)) **expect** true
check *u2.in*(**new** *CartPt*(45,40)) **expect** true
check *u2.in*(**new** *CartPt*(20,5)) **expect** false

The examples illustrate that "being within the boundary" means being within one or the other shape of a *SuperImp*.

And again, we use the template and the examples to assemble the concrete method for *SuperImp* in a straightforward manner:

inside of *SuperImp* :
double *in*(*CartPt p*){
    return this.*bot.in*(*p*) || **this**.*top.in*(*p*);
}

The method computes the results for both of the shapes it contains and then checks whether one *or* the other works. Recall that *b1* || *b2* computes whether *b1* or *b2* is true.

Without ado, here is a restatement of the last geometric problem for basic shapes (see page 123):

> ... The Navy wishes to know approximately how large an area
> a group of objects covers. It turns out that they are happy with
> the area of the bounding box, i.e., the smallest rectangle that
> contains the entire shape. ...

Since we already have a solution of this problem for *Dot*s, *Square*s, and *Circle*s, it looks like we just need to solve the problem for *SuperImp*, the variant in the union.

When we solved the bounding-box problem for *IShape*, we started with a careful analysis of what it means to compute a bounding box. Let's extend this analysis to *SuperImp*. So, imagine the square and circle in the left figure as the two components of an *SuperImp*:
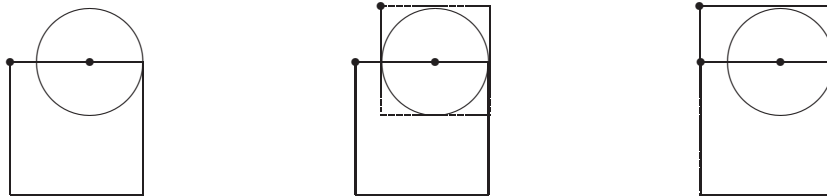
The circle's center is on the top-line of the square. In the central drawing, we see the square's and the circle's individual bounding boxes; the square's bounding box is itself and the circle's bounding box is the dashed square. The right drawing shows the bounding box for the entire *SuperImp* shape as a dashed rectangle; as you can see, it's a rectangle proper.

The immediate conclusion of our analysis is that the bounding box of a composite shape is a rectangle, not a square. Hence, *Square* can no longer serve as the representation of bounding boxes; instead we need something that represents rectangles in general. There are two obvious choices:

1. We can extend our shape datatype with a variant for representing rectangles. That is, we would add a variant to the existing union that that represents rectangles in exactly the same fashion as an instance of *Square* represents a square and an instance of *Circle* represents a circle.

2. We can define a class that is tailored to creating bounding boxes. Since bounding boxes are rectangles, such a class would also represent a rectangle. It doesn't have to implement *IComposite*, however, imposing fewer burdens on the design process. In particular, we do not have to add all the methods that *IComposite* demands from its implementing classes.

Each has advantages and disadvantages, even without considering the context in which your classes are used. Stop to think briefly about an advantage/disadvantage for each alternative.

Here we develop a solution for the second alternative; see exercise 15.5 for the first one. As a matter of fact, we don't even commit to the fields of the new class; all we assume for now is that it exists:

class *BoundingBox* { ... }

Even with that little, we can get pretty far, starting with a contract proper for the *bb* method in *IComposite*:

> inside of *IComposite* :
> ─────────────────
> // compute the bounding box for **this** shape
> *BoundingBox bb*();

The second step is to calculate out some examples, except that we don't know how to express them with *BoundingBox*. We therefore write them down in a precise but informal manner:

1. *s1.bb*() should produce a 40-by-40 rectangle at (40,80);

2. *s2.bb*() should produce a 50-by-50 rectangle at (120,50);

3. *c1.bb*() should produce a 40-by-40 rectangle at (30,100);

4. *c2.bb*() should produce a 40-by-40 rectangle at (10,20).

Not surprisingly, these rectangles are squares because they are bounding boxes for circles and squares. Still, the descriptions illustrate how to work out examples without knowledge about the result type.

Next we look at the bounding boxes of instances of *SuperImp*:

1. *u1.bb*() should produce a 110-by-70 rectangle at (40,30);

2. *u2.bb*() should produce a 70-by-50 rectangle at (10,20);

3. *u3.bb*() should produce a 70-by-120 rectangle at (10,20);

4. *u4.bb*() should produce a 70-by-120 rectangle at (10,20).

For the template step, we use the generic templates from figure 53 and refine them for this specific problem:

inside of *Square* :	inside of *Circle* :	inside of *SuperImp* :
BoundingBox bb() {	*BoundingBox bb*() {	*BoundingBox bb*() {
... **this**.*loc* ...	... **this**.*loc* ...	... **this**.*bot.bb*() ...
... **this**.*size* ...	... **this**.*radius* ...	... **this**.*top.bb*() ...
}	}	}

Without further commitments to the design of *BoundingBox*, we cannot make progress on the method definitions in *Square* and *Circle*. For *SuperImp*, however, we can actually finish the definition. Here is the template again, with the purpose statements refined for each method invocation:

inside of *SuperImp* :

BoundingBox bb() {
  // compute the bounding box for *top*
  ... **this**.*top.bb*() ...
  // compute the bounding box for *bot*
  ... **this**.*bot.bb*() ...
}

These refined purpose statements tells us that the two expressions in the template produce bounding boxes for the respective shapes. Since it is clearly a complex task to combine two bounding boxes into a bounding box, we add a wish to our wish list:

inside of *BoundingBox* :

// combine **this** bounding box with *that* one
BoundingBox combine(*BoundingBox that*);

If the wish works, we can finish *bb*'s definition:

inside of *SuperImp* :

BoundingBox bb() {
  return this.*top.bb*().*combine*(**this**.*bot.bb*());
}

Before you continue, contemplate why *combine* is a method in *BoundingBox*.

   To make progress, we need to reflect on the *bb* methods in *Circle* and *Square*. Both must produce instances of *BoundingBox* that represent squares or, in general, rectangles. Before we commit to a concrete definition of *BoundingBox*, let's briefly discuss possible representations of rectangles:

1. In the past we have represented rectangles with three pieces of information: the anchor point, its width, and its height.

2. One obvious alternative is to represent it with the four corners. Since we committed to have the sides of rectangles run parallel to the axes, we actually just need two opposing corners.

3. Based on the first two alternatives, you can probably think of a mixture of others.

   Before you commit to a choice in this situation, you should explore whether the other operations needed on your class are easy to calculate. Here we just need one: *combine*, which turns two rectangles into the smallest rectangle encompassing both. When you are faced with such a choice, it helps to plan ahead. That is, you should see how easily you can calculate

with examples or whether you can calculate examples at all. For this particular example, it helps to draw pictures of rectangles and how you would combine them, i.e, surround them with one large rectangle. See the three drawings above that explain how to get one such rectangle for one instance of *SuperImp*.

Drawing such pictures tells you quickly that the *combine* method has to pick the extreme left line, right line, top line, and bottom line from the four sides. As it turns out, this insight is easy to express for the second alternative but takes quite some work for the first. Specifically, if instances of *BoundingBox* contain the coordinates of two opposing corners, *combine* could, for example, use the extreme left and the extreme top coordinate for one new corner and the extreme right and the extreme bottom coordinate for the other. Indeed, this consideration implies that it suffices to record these four numbers in a *BoundingBox*; after all, they determine the rectangle and they allow a straightforward definition of *combine*.

Here is the basic idea then:

```
// representing bounding boxes in general
class BoundingBox {
  int lft;
  int rgt;
  int top;
  int bot;

  ...
  // combine this bounding box with that one
  BoundingBox combine(BoundingBox that) { ... }
}
```

Before we design the method, though, we should formulate the examples for the *bb* method for this choice to ensure we understand things properly:

inside of *CompositeExamples* :
boolean *test1* = **check** *s1.bb*() **expect new** *BoundingBox*(40,80,30,70);
boolean *test2* = **check** *s2.bb*() **expect new** *BoundingBox*(120,170,50,100);
boolean *test3* = **check** *c1.bb*() **expect new** *BoundingBox*(30,70,100,140);
boolean *test4* = **check** *c2.bb*() **expect new** *BoundingBox*(10,50,20,60);

boolean *test5* = **check** *u1.bb*() **expect new** *BoundingBox*(40,170,30,100);
boolean *tets6* = **check** *u2.bb*() **expect new** *BoundingBox*(10,80,20,70);
boolean *test7* = **check** *u3.bb*() **expect new** *BoundingBox*(10,80,20,140);
boolean *test8* = **check** *u4.bb*() **new** *BoundingBox*(10,80,20,140);

The first four examples show how to compute a bounding box for *Square*s and *Circle*s; not surprisingly, these bounding boxes represent squares in the Cartesian plane. The fifth and sixth are for *SuperImp*s but they are easy to compute by hand because they just combine the bounding boxes for basic shapes. The last two expected bounding boxes require some calculation. For such examples, it is best to sketch the given shape on drawing paper just to get a rough impression of where the bounding box should be.

Now that we the functional examples and a template, we can define the methods for the basic classes, *Square* and *Circle*:

inside of *Square* :	inside of *Circle* :
BoundingBox bb() {	*BoundingBox bb*() {
return	**return**
new *BoundingBox*(	**new** *BoundingBox*(
this.*loc.x*,	**this**.*loc.x* − **this**.*radius*,
this.*loc.x*+*this. size*,	**this**.*loc.x* + **this**.*radius*,
this.*loc.y*,	**this**.*loc.y* − **this**.*radius*,
this.*loc.y*+*this. size*);	**this**.*loc.y* + **this**.*radius*);
}	}

Computing the margins for a *Square* is obvious. For *Circle*s, the left margin is one **this**.*radius* to the left of the center, which is at located at **this**.*loc*; similarly, the right margin is located one **this**.*radius* to the right of the center. For the top and the bottom line, the method must conduct similar computations.

There is one entry left on our wish list: *combine* in *BoundingBox*. Recall that the purpose of *combine* is to find the (smallest) *BoundingBox* that contains **this** and *that BoundingBox*, where the latter is given as the second argument. Also recall the picture from the problem analysis. Clearly, the left-most vertical line of the two bounding boxes is the left-most line of the comprehensive bounding box and therefore determines the *lft* field of the combined box. This suggests, in turn, that *combine* should compute the minimum of **this**.*lft*—the left boundary of **this**—and *that.lft*—the left boundary of *that*:

... *Math.min*(**this**.*lft*,*that.lft*) ...

Before you move on: what are the appropriate computations for the pair of right-most, top-most, and bottom-most lines?

Putting everything together yields this method definition:

inside of *BoundingBox* :

BoundingBox combine(*BoundingBox that*) {
  return new *BoundingBox*(Math.min(**this**.*lft*,*that.lft*),
                          Math.max(**this**.*rgt*,*that.rgt*),
                          Math.min(**this**.*top*,*that.top*),
                          Math.max(**this**.*bot*,*that.bot*))
}

These methods are obviously complicated and require thorough testing. In addition, you may wish to add drawing methods that visualize the process, but keep in mind as you do so that a visual inspection does *not*—we repeat *not*—represent a test proper.

Exercises

Exercise 15.5 When we discussed the design of *BoundingBox*, we briefly mentioned the idea of adding a *Rectangle* class to the *IComposite* hierarchy figure 53:

```
class Rectangle implements IComposite {
  CartPt loc;
  int width;
  int height;

  Rectangle(CartPt loc, int width, int height){
    this.loc = loc;
    this.width = width;
    this.height = height;
  }

  ??? nnn() {
    ... this.loc ...
    ... this.width ...
    ... this.height ...
  }
}
```

Now re-design the method *bb* for *IComposite* using this signature and purpose statement:

inside of *IComposite* :

// compute the bounding box for **this** shape
Rectangle bb();

Hint: Extend *Rectangle* with auxiliary methods for computing the combination of bounding boxes.

Note: This exercise illustrates how a decision concerning the representation of information—the bounding boxes—can affect the design of methods, even though both design processes use the same design recipe after the initial decision. ∎

Exercise 15.6 Drawing shapes and their bounding boxes is a graphical way of checking whether *bb* works properly. Equip all classes from this section with a *draw* method and validate the results of *bb* via visual inspection. Add the class *Examples*, which creates examples of shapes and contains a method that first draws the bounding box for a given shape and then the shape itself. Use distinct colors for bounding boxes and shapes. ∎

15.4 Example: River Systems

The last example in this chapter concerns the problem of monitoring the nation's river systems (page 46):

> ... The EPA's software must represent river systems and monitor them. ...

Figure 55 contains the classes and interfaces for representing a river system. The code is an enrichment of the one in figure 26 with method templates where needed. Before you proceed, take another look at the example in the original problem and the object representation of this example.

Let's use these templates and examples of "inputs" to solve this problem:

> ... An EPA officer may wish to find out how many sources feed a river system. ...

First we make up some examples for the *sources* method:

check *s.sources*() **expect** 1
check *a.sources*() **expect** 3
check *b.sources*() **expect** 2
check *m.sources*() **expect** 3

```
// the end of a river
class Mouth{
  Location loc;
  IRiver river;
  ...

  ??? nnn() {
    ... this.loc.mmm() ...
    ... this.river.nnn() ...
    }
}
```

```
// a location on a river
class Location{
  int x;
  int y;
  String name;
  ...
  ??? nnn() {
      ... this.x ... this.y ... this.name ...
  }
}
```

```
// a river system
interface IRiver{
  ??? nnn();
}
```

```
// the source of a river
class Source implements IRiver {
  Location loc;
  ...



  ??? nnn() {
    ... this.loc.mmm() ...
  }


}
```

```
// a confluence of two rivers
class Confluence implements IRiver{
  Location loc;
  IRiver left;
  IRiver right;
  ...

  ??? nnn() {
    ... this.loc.mmm()
    ... this.left.nnn() ...
    ... this.right.nnn(...) ...
  }
}
```

Figure 55: Methods for a river system

Each source contributes 1 to the total count. For each confluence of two rivers, we add up the sources of both tributaries. And the sources that feed a mouth are the sources of its river.

Here are the methods, including signatures and purpose statements for *Mouth* and *IRiver*:

inside of *Mouth* :
// count the number of sources
// that feed **this** *Mouth*
int *sources*() {
  return this.*river.sources*();
}

inside of *IRiver* :
// count the number of sources
// for **this** river system
int *sources*();

The method for *Mouth* just calls the method for its *river*, following the containment arrow in the class diagram. Also following our design rules for unions of classes, the method in *IRiver* is just a signature.

    Next we define the methods for *Source* and *Confluence*:

inside of *Source* :
int *sources*(){
  return 1;
}

inside of *Confluence* :
int *sources*(){
  return this.*left.sources*() + **this**.*right.sources*();
}

The templates and the method examples suggest these straightforward definitions. You should make sure sure that these methods work as advertised by the examples.

    The next problem involves the locations that are a part of river systems:

> . . . An EPA officer may wish to find out whether some location
> is a part of a river system. . . .

Take a look at figure 56. It contains the refined templates for the relevant five classes: *Mouth*, *IRiver*, *Confluence*, *Source*, and *Location*. Specifically,

1. the refined methods have names that are appropriate for the problem;

2. they have complete signatures;

3. and they come with purpose statements.

The *Location* class is also a part of the class hierarchy because the problem statement implies that the search methods must be able to find out whether two *Location*s are the same.

    Our next step is to work through some examples:

  check *mouth.onRiver*(**new** *Location*(7,5)) **expect** true

After all, the given location in this example is the location of the mouth itself. Hence, we also need an example where the location is not the mouth:

  check *mouth.onRiver*(**new** *Location*(1,5)) **expect** false

```
// the end of a river
class Mouth {
  Location loc;
  IRiver river;
  . . .

  boolean onRiver(Location aloc) {
    . . . this.loc.same(aloc) . . .
    . . . this.river.onRiver(Location aloc) . . .
  }
}
```

```
// a location on a river
class Location {
  int x;
  int y;
  . . .
  . . .
  boolean same(Location aloc) {
    . . . this.x . . . aloc.x . . .
    . . . this.y . . . aloc.y . . .
  }
}
```

```
// a river system
interface IRiver{
  boolean onRiver(Location aloc);
}
```

```
// the source of a river
class Source implements IRiver {
  Location loc;
  . . .



  boolean onRiver(Location aloc) {
    . . . this.loc.same(aloc) . . .
  }


}
```

```
// a confluence of two rivers
class Confluence implements IRiver{
  Location loc;
  IRiver left;
  IRiver right;
  . . .


  boolean onRiver(Location aloc) {
    . . . this.loc.same(aloc)
    . . . this.left.onRiver(aloc) . . .
    . . . this.right.onRiver(aloc) . . .
  }
}
```

Figure 56: Methods searching river systems for locations

This time, the given location is the source of *t*, which joins up with *s* to form *b*, and that in turn flows into *a* and thus *mouth* itself. A complete set of examples would contain method calls for the templates in *Source*, *Confluence*, and *Location*.

The examples suggest that *onRiver* in *Mouth* checks whether the given location is the location of the mouth or occurs on the river system:

inside of *Mouth* :
// does *aloc* occur
// along **this** river system?
boolean *onRiver*(*Location aloc*){
  return this.*loc.sameloc*(*aloc*) ||
       this.*river.onRiver*(*aloc*);
}

inside of *IRiver* :
// does *aloc* occur
// along **this** river system?
boolean *onRiver*();

Again, the two method calls are just those from the template; their combination via || is implied by the examples. Both follow the containment arrows in the diagram, as suggested by the design recipe.

The methods in *Source* and *Confluence* follow the familiar pattern of recursive functions:

inside of *Source* :
boolean *onRiver*(*Location aloc*){
  return this.*loc.sameloc*(*aloc*);
}

inside of *Confluence* :
boolean *onRiver*(*Location aloc*){
  return this.*loc.sameloc*(*aloc*) ||
         this.*left.onRiver*(*aloc*) ||
         this.*right.onRiver*(*aloc*);
}

In *Source*, the method produces true if and only if the given location and the location of the source are the same; in *Confluence*, the given location could be on either branch of the river or it could mark the location of the confluence.

Last but not least we must define what it means for two instances of *Location* to be the same:

inside of *Location* :
// is **this** location identical to *aloc*?
boolean *same*(*Location aloc*) {
  return (**this**.*x* == *aloc.x*) && (**this**.*y* == *aloc.y*);
}

Like the methods above, this one is also a simple refinement of its template.

Note: If we wanted to have a more approximate notion "sameness", we would of course just change this one definition. The others just defer to *Location* for checking on sameness, and hence changing "sameness" here would change it for every method. What you see in action is, of course, the principle of "single point of control" from *How to Design Programs*. ∎

Here is the final problem concerning river systems:

> ...   An EPA officer may request the number of miles of a river
> system, either starting from the river's mouth or any of its con-
> fluence points. ...

Because our existing representation of river systems doesn't include any information about the lengths of the segments, the problem requires a modification of the data representation before we even consider the design of the required method.

```
            t
 s         /
  \      length: 2
length: 3 /
    \    /
     \  /
      \/                  IRiver s = new Source(3,new Location(1, 1));
      b          u        IRiver t = new Source(2,new Location(1, 5));
       \        /         IRiver u = new Source(1,new Location(3, 8));
   length: 3  length: 1
        \    /            IRiver b =
         \  /               new Confluence(3,new Location(3, 3),s,t);
          a              IRiver a =
          |                new Confluence(4,new Location(5, 5),b,u);
       length: 4
          |              Mouth mouth =
          m                new Mouth(new Location(7, 5), a);
```
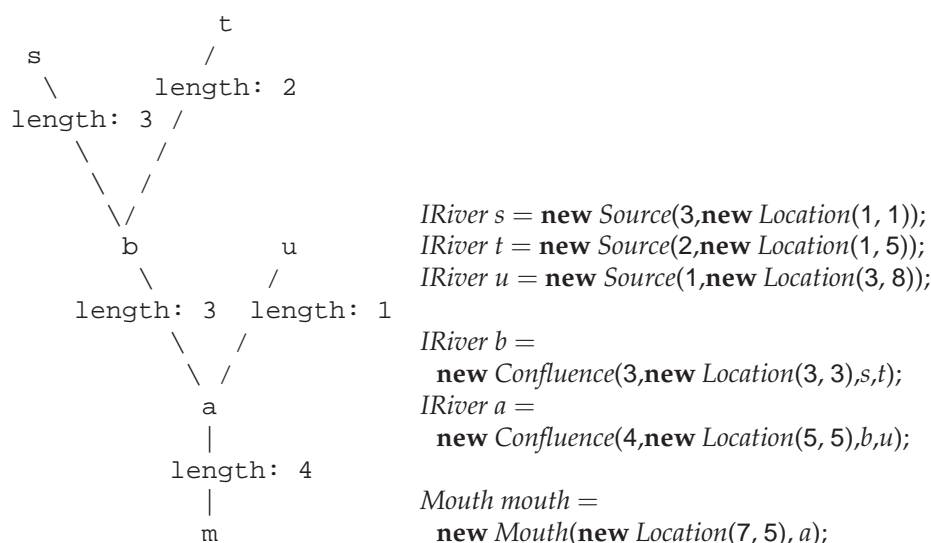
Figure 57: A sample river system with length measurements

It is easy to add the lengths of river segments to the picture of a river system because the segments are clearly visible. The picture on the left of figure 57 shows how to do this for the artificial example in the original problem statement (page 46). In our data representation from figure 55 or equivalently 26, however, we don't represent the segments at all. Instead we just represent three kinds of points of interest: the mouth, the confluences of rivers, and their sources. The segments are implied as connections between these points.

This situation forces a choice on us: to include segments in our data representation or to work with what we have. Changing the representation means adding a class that represents segments and modifying *Mouth*, *IRiver*, *Source*, and *Confluence* to refer to segments. In all likelihood the existing methods won't work after this reorganization. Alternatively, we can

add fields to the existing classes that record the length of associate segments. Still, even if we leave the overall structure of the hierarchy alone, we face another choice:

1. for any point of interest, we can record the length of the downward segment; or

2. for any point of interest, we can record the length of the upward segment.

Here we choose to leave the structure of the class hierarchy alone and to record the length of a segment in its origination point (choice 1); exercise 15.9 explores the second choice.

```
class Mouth {
  Location loc;
  IRiver river;
  …

  ??? nnn() {
    … this.loc.mmm() …
    … this.river.nnn() …
}
```

```
interface IRiver {
  ??? n();
}
```

```
class Source implements IRiver {
  int miles ;
  Location loc;
  …


  ??? nnn() {
    … this.miles …
    … this.loc.mmm() …
  }


}
```

```
class Confluence implements IRiver {
  int miles ;
  Location loc;
  IRiver left;
  IRiver right;
  …

  ??? nnn() {
    … this.miles …
    … this.loc.mmm()
    … this.left.nnn() …
    … this.right.nnn(…) …
  }

}
```
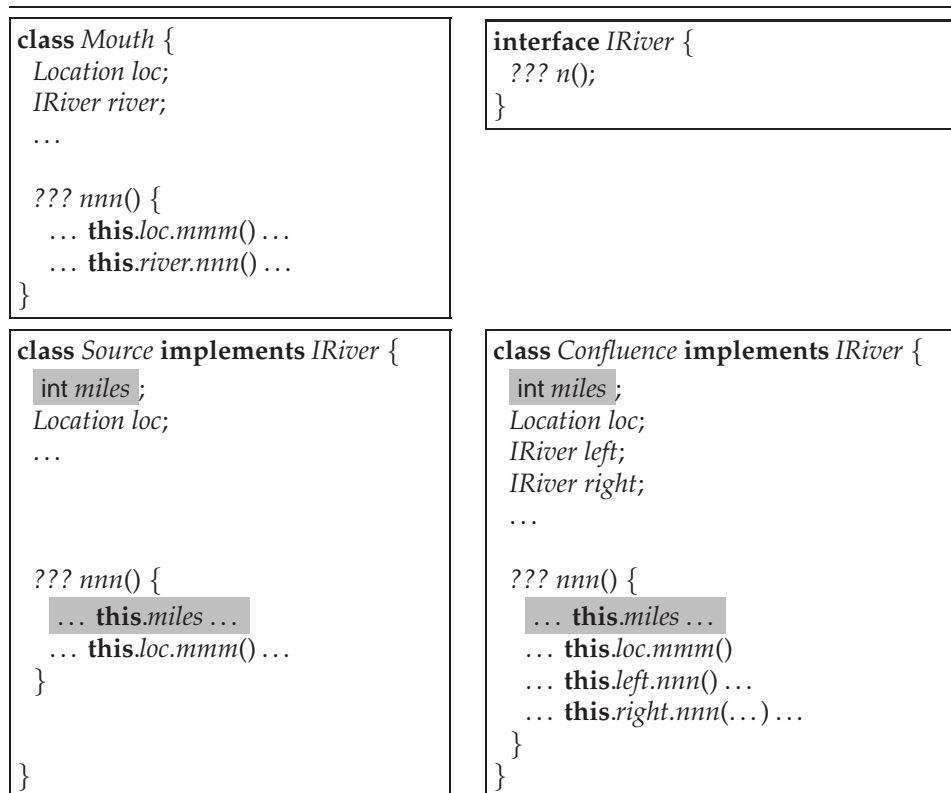
Figure 58: Adding the length of a river segment to the data representation

Figure 58 shows the adapted classes for a river system. The new fields in *Source* and *Confluence* denote the length of the down-river segment that

is adjoined to a point of interest. There is no *length* field in *Mouth* because the mouth of a river is its end. The figure also contains the adapted method templates. Both *nnn* in *Source* and *nnn* in *Confluence* require one extra line. The added fields and lines are shaded gray.

For examples that describe the behavior of *length*, we need to look back at the left part of figure 57 and adapt the examples from the original problem (page 49); the right part of figure 57 shows the new representation. Based on these specifications, the expected results for *length* are as follows:

> **check** *s.length*() **expect** 3
> **check** *t.length*() **expect** 2
> **check** *u.length*() **expect** 1
> **check** *b.length*() **expect** 8
> **check** *a.length*() **expect** 13
> **check** *m.length*() **expect** 13

From here it is easy to develop a method that computes the total length of a river system from a certain point:

inside of *Mouth* :   inside of *IRiver* :
// the total length of   // compute the total length of the
// the river system   // waterways that flow into **this** point
int *length*(){   int *length*();
  return this.*river.length*();
}

The method in *Mouth* still defers to the method of its *river* field to compute the result; and that latter method is just a signature, i.e., it is to be defined in the variants of the union.

All that is left to define are the methods in *Source* and *Confluence*:

inside of *Source* :   inside of *Confluence* :
boolean *length*(){   int *length*(){
  return this.*miles;*   **return this**.*miles* +
}   **this**.*left.length*() +
                                                     this.*right.length*();
                                             }

The total length for the *Source* class is the value of the *length*. The total length of the river flowing to a *Confluence* is the sum of the total lengths of the two tributaries and the *length* of this river segment.

Exercises

Exercise 15.7 The EPA has realized that its case officers need a broader meaning of "finding locations along the river system" than posed in this section:

> ... An EPA officer may wish to find out whether some location is within a given radius of some confluence or source on a river system. ...

Modify the existing *onRiver* method to fit this revised problem statement. ∎

Exercise 15.8 Design the following methods for the class hierarchy representing river systems:

1. *maxLength*, which computes the length of the longest path through the river system;

2. *confluences*, which counts the number of confluences in the river system; and

3. *locations*, which produces a list of all locations on this river, including sources, mouths, and confluences. ∎

Exercise 15.9 Design a representation of river systems such that each place (mouth, confluence, or source) describes how long the segments are that flow into it. Hint: For a confluence, you will need two lengths: one for the left tributary and one for the right. ∎

15.5 Finger Exercises

Exercise 15.10 Design a data representation for shopping lists. Start from the class of grocery items developed in exercise 14.7. Add the following methods:

1. *howMany*, which computes the number of items on the shopping list;

2. *brandList*, which produces the list of all brand names; and

3. *highestPrice*, which determines the highest unit price among all items in the shopping list. ∎

Exercise 15.11 Figure 59 contains a class diagram that describes the GUI hierarchy of exercise 6.5. Note that one of its interfaces, *ITable*, refines another interface *IGUIComponent*.

Add templates to each box in this diagram. Do not assume anything about the return type; do not design classes. ∎
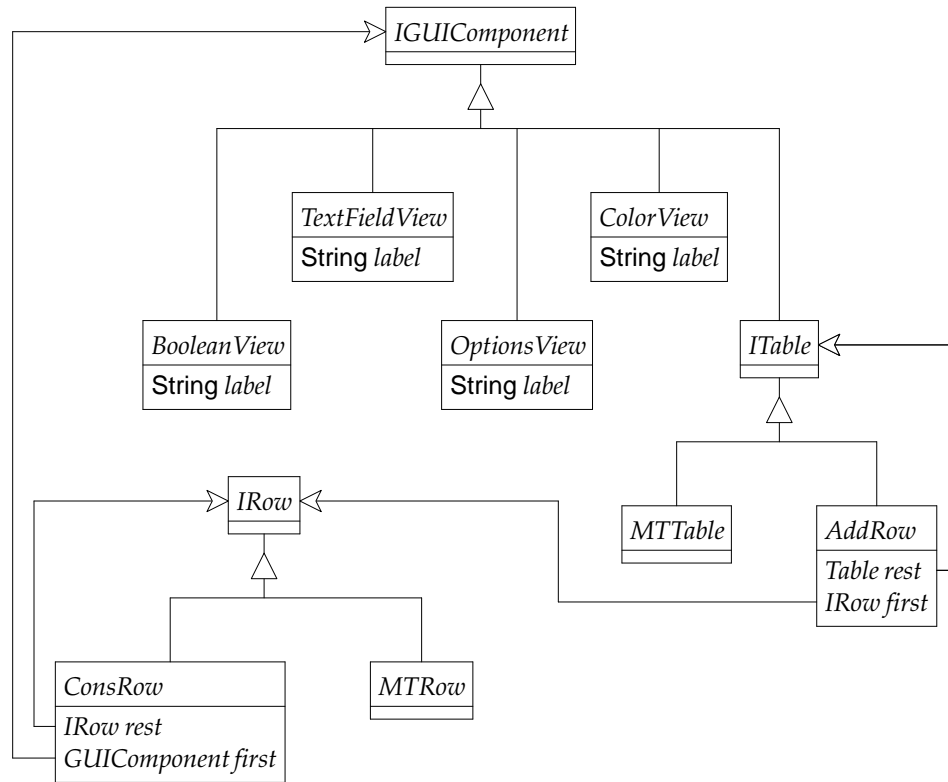
Figure 59: A hierarchy of GUI components

Exercise 15.12 Develop a program for managing discount bookstores (see exercise 14.8):

1. design a representation for lists of books;

2. write down (in English) three examples of book lists and their corresponding data representations;

3. develop the method *thisAuthor*, which produces the list of books that this author has authored.

Modify the data representation for books so that books can have an arbitrary number of authors. Then adapt the methods above. ∎

16 <mark>Designing Methods</mark>

<mark>The purpose of a method is to produce data from the data in the given object and the method's arguments.</mark> For that reason, the design recipe for methods—adapted from the design recipe from *How to Design Programs* for functions—focuses on laying out all the available pieces of data as the central step:

purpose & signature When you design a method, your first task is to clarify what the method consumes and what it produces. The method signature specifies the classes of (additional) "inputs" and the class of "outputs." You should keep in mind that a method always consumes at least one input, the instance of the class on which it is invoked.

Once you have a signature, you must formulate a purpose statement that concisely states *what* the method computes with its arguments. You do *not* need to understand (yet) *how* it performs this task. Since the method consumes (at least) the instance of the class in which the method is located, it is common practice to write down the purpose statement in terms of **this**; if the names of parameters are useful, too, use them to make the statement precise.

functional examples The second step is the creation of examples that illustrate the purpose statement in a concrete manner.

template The goal of the template step is to spell out the pieces of data from which a method can compute its result. Given that a method always consumes an instance of its class, the template definitely contains references to the fields of the class. Hint: Annotate these "selector" expressions with comments that explain their types; this often helps with the method definition step.

If any of the fields have a class or interface type, remind yourself with an appropriate expression that your method can—and often must—use method calls on these objects.

Additionally, if your method's extra arguments have class or interface types, add a schematic method call to the method parameter (*p*):

```
AType m(AClass p, ...) {
  ... p.lll()
}
```

Last but not least, before you move on, keep in mind that a method body may use other already defined methods in the surrounding class or in the class and interaces of its parameters.

method definition Creating the method body is the fourth step in this sequence. It starts from the examples; the template, which lays out all the available information for the computation; and the purpose statement, which states the goal of the computation. If the examples don't clarify all possible cases, you should add examples.

tests The last task is to turn the examples into executable tests. Ideally these tests should be evaluated automatically every time you edit the program and get it ready to run.

The five steps depend on each other. The second step elaborates on the first; the definition of the method itself uses the results of all preceding steps; and the last one reformulates the product of the second step. On occasion, however, it is productive to re-order the steps. For example, creating examples first may clarify the goal of the computation, which in turn can help with the purpose statement. Also, the template step doesn't depend on the examples, and it is often possible and beneficial to construct a template first. It will help a lot when methods come in bunches; in those cases, it is often straightforward to derive the methods from the templates then.

16.1   The Varieties of Templates

Class arrangements come in four varieties and so do templates. Let's take a quick look at each of these situations because it will help you develop templates from the organization of the classes.

Basic Classes

A basic class comes with a name and some properties of primitive type:

Basic
String *s*
int *i*
double *d*

Basic
String *s*   int *i*   double *d*
??? nnn() {   ... **this**.*i* ...   ... **this**.*d* ...   ... **this**.*j* ... }

Adding a template to such a basic class is straightforward, as the elaborated class diagram on the right shows. The method template enumerates all fields using the **this**-dot syntax, which indicates that the values come from the instance that is the implicit first argument to *mmm*.

Containment

The next case concerns a class that refers to (instances of) other classes:

```
Containment          ▷ Basic
Basic b                String s
double d               int i
                       double d
```
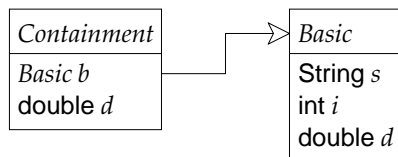
Here *Containment* refers to *Basic*. A computation concerning *Containment* may therefore need some data from (or about) *Basic*. Of course, this data may or may not play the same role, so this doesn't mean that we need the exact same method in *Basic* that we have in *Containment*. We therefore create a template with distinct method stubs in the two classes:

```
Containment                ▷ Basic
Basic b                       String s
double d                      int i
                              double d
??? mmm() {
  ... this.b.nnn() ...       ??? nnn() {
  ... this.d ...       }       ... this.i ...
                               ... this.d ...
                               ... this.j ... }
```
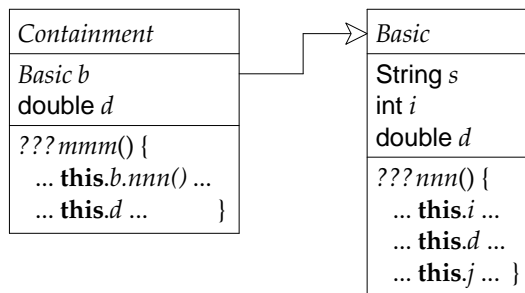
When we eventually define *mmm*, the expression **this**.*b.nnn*() reminds us that we may have to add a method to *Basic*. If so, we put it on our wish list; the template already exists and makes it easy to define the method.

Unions

Here is the diagram for the third situation:

IUnion

Basic1
boolean *b*
double *d*

Basic2
int *i*
double *d*

Basic3
int *i*
String *s*
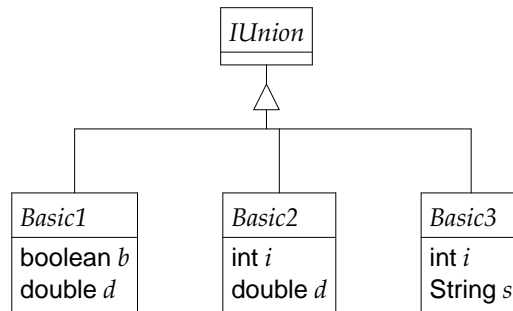
The diagram specifies *IUnion* as the interface for the union of three concrete classes: *Basic1*, *Basic2*, and *Basic3*. An interface such as *IUnion* represents a common facade for the three classes to the rest of the program.

The template for a union of classes requires a method signature in the interface and basic templates in the concrete variants of the union:

IUnion

Basic1
boolean *b*
double *d*
??? mmm() {
 ... **this**.*b* ...
 ... **this**.*d* ... }

Basic2
int *i*
double *d*
??? mmm() {
 ... **this**.*i* ...
 ... **this**.*d* ... }

Basic3
int *i*
String *s*
??? mmm() {
 ... **this**.*i* ...
 ... **this**.*s* ... }
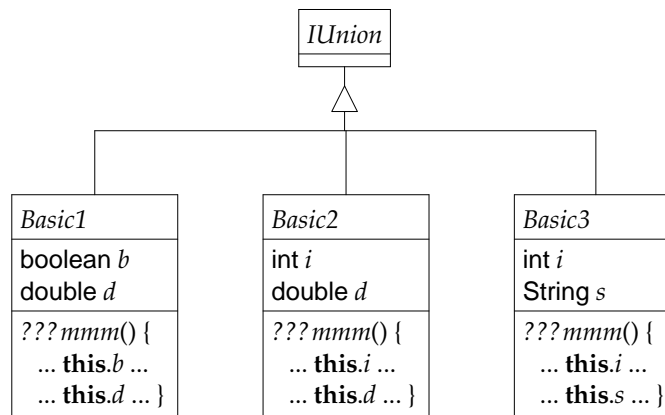
The method templates for the basic classes mention all fields of the respective classes.

Self-References and Mutual References

The last case is the most complex one. It deals with self-references and mutual references in class diagrams via refinement and containment arrows. The following diagram shows a single self-reference in a class hierarchy:
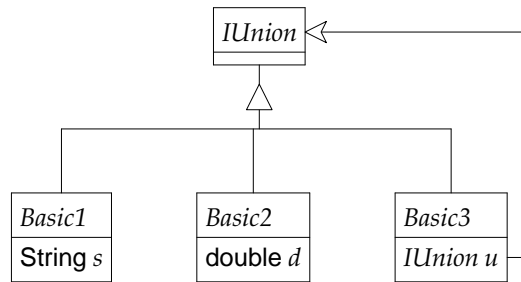
A closer look at the diagram shows that this last particular example is a composition of the containment and the union case.

Accordingly the creation of the method templates is a composition of the actions from the preceding two cases. Here is the resulting diagram:
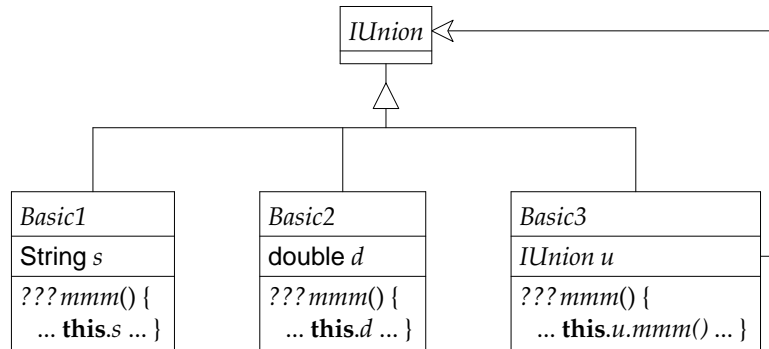
The *IUnion* interface contains a method signature for *mmm*, specifying how all instances of this type behave. The three concrete classes contain one concrete template for *mmm* each. The only novelty is in the template for *Contain3*'s *mmm* method. It contains a method invocation of *mmm* on *u*. Since *u* is of type *IUnion*, this recursive invocation of *mmm* is the only natural way to compute the relevant information about *u*. After all, we chose *IUnion* as *u*'s type because we don't know whether *u* is going to be an instance of *Atom1*, *Atom2*, or *Contain3*. The recursive invocation therefore represents a dynamic dispatch to the specific method in the appropriate concrete class.

Even though the sample diagram is an example of a self-referential class hierarchy, the idea smoothly generalizes to complex cases, including class hierarchies with mutual references. Specifically, an interface for a union of classes always contains a general method signature. As we follow the (reverse) refinement arrows, we add concrete versions of the method signature to the classes. The body of the template contains one reference to each field. If the field refers to another class in the hierarchy along a containment

arrow, we turn the field reference into a method call. If, furthermore, the reference to the other class creates a cycle in the class hierarchy, the method invocation is recursive. Note how for a collection of classes with mutual references along containment arrows, the design recipe creates natural mutual references among the methods.

16.2 Wish Lists

A wish list is a programmer's most important instrument. This is true for people who design programs, functions, or class hierarchies with methods. The very first wish on the list is the method that we are asked to design. As we develop the method for the class hierarchy, we often encounter situations when we just wish we had some additional method that computes something about some other object. When that happens, we enter another wish on our wish list. More precisely, we identify to which class or interface we should add the method and formulate a rough purpose statement and signature for the method. Those two pieces of information generally suffice to construct the method later. In the meantime, we act as if we had that method and finish the definition of the method that we are working on. Obviously, we can't test this method because it depends on a non-existing method. Hence, our next step is to pick an item from our wish list and to work on it.

When our wish list is empty, we can test all of our examples. We start testing those methods that don't depend on any other methods. Once we are confident that these basic methods work well, we move on to the methods that depend on the ones we have tested. Put differently, we first work our way down from the initially requested method to the most basic ones, and then we test our way back up, starting from the most basic methods.

There are two different situations that call for the addition of a method to the wish list:

1. If a template contains a method call, say **this**.*u*.*mmm*() where **this**.*u* belongs to some other class or interface, and if the respective containment arrow does not create a self-reference in the class hierarchy, we add an entry to the wish list. Specifically, we add a purpose statement and signature to the type of **this**.*u*.

2. If you discover that a method definition becomes complex, you are probably trying to compute too many things at once. Less charitably, you are about to violate the "one task, one function" rule. In this case,

it is critical to identify the independent tasks and to formulate them as entries on the wish list.

Some experienced programmers can cope with long and complex wish lists. You shouldn't. You should avoid large gaps between method definitions and testing. Without quick testing, beginning programmers simply can't gain enough confidence about the progress that has been made. You should therefore learn to recognize basic methods and to design and test them as quickly as possible.

Experienced programmers also recognize when some other class or the programming language itself already provides the functionality of a wish. For example, Java comes with a library of classes that deal with positions so that we don't really have to design a class for positions and methods for computing the distance to the origin. A programmer with a lot of experience will evaluate those classes and deliberately decide whether to design a new class or reuse the existing one.

16.3   Case Study: Fighting UFOs, with Methods

Recall the "war of the worlds" problem:

> . . .   Develop a "War of the Worlds" game. A *UFO* descends from the sky. The player owns an anti-*UFO* platform (AUP), which can move left and right on the ground. It can also fire shots, straight up from the middle of the platform. If any of the shots hit the *UFO*, the player wins. Otherwise, the *UFO* lands and destroys the earth. . . .

In section 6.2, we developed a data representation for this world of *UFO*s. All game objects belong to an instance of *UFOWorld*. Three classes deal with three kinds of physical objects: *UFO*s, *AUP*s, and *Shot*s. The others represent the collection of all shots that the player has fired.

In this section, we add some methods. The purpose isn't to develop a complete game (yet), but to study the design recipe for methods in action. Just like for the design of classes, you need to read the problem statement carefully to figure out what methods are needed. Our problem statement contains some direct requests for methods:

1. The *UFO* descends, i.e., it moves downward on the canvas.

2. The player can move the *AUP* left or right.

```
// the world of UFOs, AUPs, and Shots
class UFOWorld {
  UFO ufo;
  AUP aup;
  IShots shots;
  IColor BACKG = new Blue();
  int HEIGHT = 500;
  int WIDTH = 200;

  UFOWorld(UFO ufo, AUP aup, IShots shots) { ... }

  // draw this world
  boolean draw(Canvas c) { ... }
  // move the objects in this world
  UFOWorld move() { ... }
  // fire a shot in this world
  UFOWorld shoot() { ... }
}
```

```
// an anti-UFO platform: its left corner is
// location pixels from the left,
// at the bottom of the world
class AUP {
  int location;
  IColor aupColor = new Red();

  AUP(int location) { ... }

  // draw this AUP
  boolean draw(Canvas c) { ... }
  // fire a shot from this AUP
  Shot fireShot(UFOWorld w) { ... }
}
```

```
// a UFO: its center is at location

class UFO {
  Posn location;
  IColor colorUFO = new Green();

  UFO(Posn location) { ... }

  // draw this UFO
  boolean draw(Canvas c) { ... }
  // drop this UFO
  UFO move() { ... }
}
```

Figure 60: *UFO*s and *AUP*s with preliminary method specifications

3. The player can also fire shots from the *AUP*.

4. The shots fly upwards, presumably in a straight line.

5. If a shot hits the *UFO*, the player wins the game.

6. If the *UFO* has landed, the player has lost the game.

Implicitly, the problem also asks for methods that draw the various objects onto the canvas. Here we discuss how to develop methods for drawing the objects of the world in a canvas, for moving them, and for firing a shot. The remaining methods are the subject of exercises at the end of the section.

Figures 60 and 61 display the classes that represent the entire world, *UFO*s, *AUP*s, and lists of shots; the routine constructors are elided to keep the figures concise. When you add purpose statements and signatures to such a system of classes, start with the class that stands for the entire world. Then you follow the containment and refinement arrows in the diagram as directed by the design recipe. If you believe that a method in one class requires a method with an analogous purpose in a contained class, it's okay to use the same name. These additions make up your original wish list.

Now take a look at the *draw* method in *UFOWorld*. Its purpose is to draw the world, that is, the background and all the objects contained in the world. The design recipe naturally suggests via the design template that *draw* in *UFOWorld* use a *draw* method in *UFO*, *AUP*, and *IShots* to draw the respective objects:

```
inside of UFOWorld :
boolean draw(Canvas c) {
  ... this.BACKG ... this.HEIGHT ... this.WIDTH ...
  ... this.ufo.draw(...) ...
  ... this.aup.draw(...) ...
  ... this.shots.draw(...) ...
}
```

An example of a canvas for *UFOWorld* came with the original problem statement (see page 56). The image shows that the world is a rectangle, the *UFO* a flying saucer, and the *AUP* a wide horizontal rectangle with a short vertical rectangle sticking out in the middle. Before we can define the *draw* method, however, we must also recall from section 14.1 how the drawing package works. A *Canvas* comes with basic methods for drawing rectangles, circles, disk and so on. Hence, a *draw* method in our world must consume an instance of *Canvas* and use its methods to draw the various objects.

With all this in mind, we are now ready to define *draw*:

<u>inside of *UFOWorld* :</u>
boolean *draw*(*Canvas c*) {
 return
  c.drawRect(**new** *Posn*(0,0), **this**.*WIDTH*,**this**.*HEIGHT*,**this**.*BACKG*)
  && **this**.*ufo.draw*(*c*)
  && **this**.*aup.draw*(*c*)
  && **this**.*shots.draw*(*c*);
 }

It combines four drawing actions with *&&*, meaning all four must succeed before *draw* itself signals success with true.

```
// managing a number of shots
interface IShots {
  // draw this list of shots
  boolean draw(Canvas c);
  // move this list of shots
  IShots move();
}
```

```
// a shot in flight, whose upper
// left corner is located at location
class Shot {
  Posn location;
  IColor shotColor = new Yellow();

  Shot(Posn location) { … }
  // draw this shot
  boolean draw(Canvas c) { … }
  // move this list of shots
  Shot move(UFOWorld w) { … }
}
```

```
// the empty list of shots
class MtShots implements IShots {



  MtShots() { }

  boolean draw(Canvas c) { … }
  IShots move(UFOWorld w) { … }
}
```

```
// a list with at least one shot
class ConsShots implements IShots {
  Shot first;
  IShots rest;

  ConsShots(Shot first, IShots rest) { … }

  boolean draw(Canvas c) { … }
  IShots move(UFOWorld w) { … }
}
```

Figure 61: *Shot*s and lists of shots with preliminary method specifications

---

The *draw* methods in *AUP* and *UFO* just draw the respective shapes. For example, *draw* in *UFO* draws a disk and a rectangle placed on its center:

```
inside of UFO :
// draw this UFO
boolean draw(Canvas c) {
  return
    c.drawDisk(this.location,10,this.colorUFO) &&
    c.drawRect(new Posn(this.location.x − 30,this.location.y − 2),
               60,4,
               this.colorUFO);
}
```

Note how the calculations involve for the positions plain numbers. Following the advice from *How to Design Programs*, it would be better of course to introduce named constants that the future readers what these numbers are about. Do so!

In comparison, *IShots* contains only a method signature because *IShots* is the interface that represents a union. Its addition immediately induces two concrete *draw* methods: one in *MtShots* and one in *ConsShots*. Here are the two templates:

```
inside of MtShots :           inside of ConsShots :
boolean draw(Canvas c) {       boolean draw(Canvas c){
 ...                             ... this.first.draw() ... this.rest.draw() ...
}                              }
```

Since the purpose of the *draw* method is to draw *all* shots on a list of shots, it is natural to add a *draw* method to *Shot* and to use **this**.*first.draw*() in *ConsShots*. Like all other *draw* methods, these, too, consume a *Canvas* so that they can use the drawing methods for primitive shapes.

Exercises

Exercise 16.1 Define the *draw* methods for *MtShots*, *ConsShots*, and *Shot*. Remember that on screen, a *Shot* is drawn as a thin vertical rectangle. ∎

Exercise 16.2 Define the *draw* method in *AUP*. As the screen shot in the problem statement on page 56 indicates, an *AUP* consists of two rectangles, a thin horizontal one with a short, stubby vertical one placed in its middle. It is always at the bottom of the screen. Of course, your boss may require that you can change the size of *UFOWorld* with a single modification, so you must use a variable name not a number to determine where to place the *AUP* on the canvas. Hint: Add a parameter to the *draw* method's parameter

list so that it can place itself at the proper height. Does this change have any implications for *draw* in *UFOWorld*? ▪

Exercise 16.3 When you have solved exercises 16.1 and 16.2 combine all the classes with their draw methods and make sure your code can draw a world of *UFO*s. You may wish to use the examples from figure 34. ▪

Once your program can draw a world of objects, the natural next step is to add a method that makes the objects move. The purpose of *move* in *UFOWorld* is to create a world in which all moving objects have been moved to their next place—whatever that is. Hence, if the method is somehow called on a regular basis and, if every call to *move* is followed by a call to *draw*, the player gets the impression that the objects move continuously.

Here is a draft purpose statement, a method signature, and a template for the *move* method:

inside of <u>*UFOWorld*</u> :
// to move all objects in **this** world
UFOWorld move() {
  ... **this**.*BACKG* ... **this**.*HEIGHT* ... **this**.*WIDTH* ...
  ... **this**.*ufo.move*(...) ...
  ... **this**.*aup.move*(...) ...
  ... **this**.*shots.move*(...) ...
}

The template reminds us that the world consists of three objects and, following the design recipe, that each of these objects also has a *move* method. You might conclude that the *move* method for the world moves all objects just like the *draw* method draws all objects. This conclusion is too hasty, however. While the *UFO* and the shots are moving automatically, the *AUP* moves in response to a player's instructions. We therefore exclude the *AUP* from the overall movement and focus only on the *UFO* and the shots:

inside of <u>*UFOWorld*</u> :
// to move the UFO and the shots in **this** world
UFOWorld move() {
  return
    new *UFOWorld*(**this**.*ufo.move*(),**this**.*aup*,**this**.*shots.move*());
}

That is, the *move* method creates a new *UFOWorld* from the existing *AUP* plus a *UFO* and a list of shots that have been moved.

Exercises

Exercise 16.4 Our development skipped over the example step. Use figure 34 to develop examples and turn them into tests. Run the tests when the *move* methods for *UFO*s and *IShots* are defined (see below). ∎

By following the design recipe, you have placed two methods on your wish list: *move* in *UFO* and *move* in *IShots*. The latter is just a signature; its mere existence, though, implies that all implementing classes must posses this method, too. The former is concrete; its task is to create the "next" *UFO*. This suggests the preliminary method signatures and purpose statements in figures 60 and 61.

Let's take a step at a time and develop examples for *move* in *UFO*. Suppose the method is invoked on

    new *UFO*(**new** *Posn*(88,11))

Given the dimensions of *UFOWorld*, this *UFO* is close to the top of the canvas and somewhat to the left of the center. Where should it be next? The answer depends on your boss, of course.[24] For now, we assume that *move* drops the *UFO* by three (3) pixels every time it is invoked:

    check new *UFO*(**new** *Posn*(88,11)).*move*()
    expect new *UFO*(**new** *Posn*(88,14))

Similarly:

    check new *UFO*(**new** *Posn*(88,14)).*move*()
    expect new *UFO*(**new** *Posn*(88,17))

and

    check new *UFO*(**new** *Posn*(88,17)).*move*()
    expect new *UFO*(**new** *Posn*(88,20))

While this sequence of examples shows the natural progression, it also calls into question the decision that *move* drops the *UFO* by 3 pixels *every* time it is invoked. As the problem statement says, once the *UFO* reaches

---

[24]Deep down, it depends on your knowledge, your imagination, and your energy. Many computer games try to emulate physical laws. To do so, you must know your physics. Others rely on your imagination, which you can foster, for example, with readings on history, mythology, science fiction, and so on.

the ground level, the game is over and the player has lost. Translated into an example, the question is what the expression

new *UFO*(**new** *Posn*(88,499)).*move*()

for example, should produce. With a *y* coordinate of 499, the *UFO* is close to the ground. Another plain move would put it *under* the ground, which is impossible. Consequently, *move* should produce a *UFO* that is on the ground:

check new *UFO*(**new** *Posn*(88,499)).*move*()
expect new *UFO*(**new** *Posn*(88,500))

Of course, if *move* is invoked on a *UFO* that has landed, nothing happens:

check new *UFO*(**new** *Posn*(88,500)).*move*()
expect new *UFO*(**new** *Posn*(88,500))

```
// the world of
// UFOs, AUPs, and Shots
class UFOWorld {
  UFO ufo;
  AUP aup;
  IShots shots;
  IColor BACKG = …;
  int HEIGHT = 500;
  int WIDTH = 200;
  …
  // move the objects in this world
  UFOWorld move() {
  return
   new UFOWorld(this.ufo.move( this ),
                 this.aup,
                 this.shots.move());
  }
}
```

```
// a UFO, whose center is
// located at location
class UFO {
  Posn location;
  IColor colorUFO = …;
  …
  // drop this UFO by 3 pixels
  UFO move(UFOWorld w) {
   if (this.landed(w)) {
    return this; }
   else { if (this.closeToGround(w)) {
    return
     new UFO(
      new Posn(this.location.x,
               w.HEIGHT)); }
   else {
    return
     new UFO(
      new Posn(this.location.x,
               this.location.y+3)); }
   }
  }
}
```

Figure 62: The *move* methods for *UFOWorld* and *UFO*

Here is the preliminary template for *move*:

```
inside of UFO :
// to move this UFO
UFO move() {
  ... this.location ... this.colorUFO ...
}
```

The template points out that an instance of *UFO* contains two fields: one records the current location, which *move* must change, and the other one records the *UFO*'s color, which is irrelevant to *move*. It currently does *not* have any data, however, that helps decide whether the *UFO* is close to the ground or whether it is on the ground.

To overcome this problem, we can choose one of these solutions:

1. equip the *UFO* class with a field that represents how far down it can fly in the world. This also requires a modification of the constructor.

2. add a parameter to the *move* method that represents the height of the world. Adapting this solution requires a small change to the *move* method in *UFOWorld*, i.e., the addition of an argument to the invocation of *UFO*'s *move* method.

3. Alternatively, *move* in *UFOWorld* can also hand over the entire world to *move* in *UFO*, which can then extract the information that it needs.

We choose the third solution. For an analysis of the alternatives, see the exercises below.

With the change, we get this revised template:

```
inside of UFO :
// to move this UFO
UFO move(UFOWorld w) {
  ... w ... this.location ... this.colorUFO ...
}
```

If you wanted to, you could now add the numeric case distinction that the study of examples has suggested; check how to develop conditional methods in section 10.4.

The complete definitions for *UFOWorld* and *UFO* are displayed in figure 62. The *move* method in *UFOWorld* invokes *move* from *UFO* with **this**: see the gray-shaded occurrence of **this**. It is the first time that we use **this** as an argument but it shouldn't surprise you. We have used **this** as a **return** value many times, and we always said that **this** stands for the current object, and objects are arguments that come with method calls. It is perfectly natural to hand it over to some other method, if needed.

The *move* method in *UFO* first checks whether the *UFO* has already landed and then whether it is close enough to the ground. Since these two computations are really separate tasks, we add them to our wish list:

> inside of *UFO* :
> // has **this** UFO landed yet?
> boolean *landed*(*UFOWorld w*) { … }
>
> // is **this** UFO about to land?
> boolean *closeToGround*(*UFOWorld w*) { … }

The two are relatively simple methods, and the examples for *move* readily induce examples for these two methods.

Exercises

Exercise 16.5 Design the method *landed* for *UFO*. ∎

Exercise 16.6 Design the method *closeToGround* for *UFO*. ∎

Exercise 16.7 Revise the examples for *move* in *UFO* and test the method, after you have finished exercises 16.5 and 16.6. ∎

Exercise 16.8 You have decided to double the height of the world, i.e., the height of the canvas. What do you have to change in the *move* methods?

You are contemplating whether players should be allowed to resize the canvas during the game. Resizing should imply that the *UFO* has to descend further before it lands. Does our design accommodate this modification?

Change the *move* methods in *UFOWorld* and *UFO* so that the former hands the latter the height of **this** world. Does your answer to the question stay the same? ∎

Exercise 16.9 Change the *UFO* class so that it includes an int field that represents the height of the world, i.e., how far down it has to fly before it can land. Remove the *UFOWorld* parameter from its *move* method. Hint: The changes induces changes to *UFOWorld*, too.

Once you have completed the design (i.e, run the tests), contemplate the question posed in exercise 16.8.

In response to this question, you may wish to contemplate whether the *UFO* class should include a field that represents the entire *UFOWorld* in which it exists. What problem does this design choice pose? ∎

Exercise 16.10 Players find your *UFO* too predictable. They request that it should randomly swerve left and right, though never leave the canvas. Would the current *move* method easily accommodate this change request?

Note: We have not provided enough information yet for you to design such a *move* method. For the impatient, see exercise 19.18. ∎

Designing the *move* method for the collection of shots is much more straightforward than *move* for *UFO*. Recall the method signature in *IShots*:

inside of *IShots* :
// move **this** list of shots
IShots move();

It says that the method consumes a collection of shots and produces one. The implementing classes contain concrete method definitions:

inside of *MtShots* :
IShots move() {
  return this;
}

inside of *ConsShots* :
IShots move() {
  return
    new *ConsShots*(**this**.*first.move*(),**this**.*rest.move*());
}

The one in *MtShots* just returns the empty list; the other one moves the *first* shot and the *rest* via the appropriate *move* methods. Since *first* is an instance of *Shot*, **this**.*first.move*() uses the yet-to-be-defined method *move* in *Shot*; similarly, since *rest* is in *IShots*, **this**.*rest.move*() invokes *move* on a list of shots.

The *move* method in *Shot* is suggested by the design recipe, specifically by following the containment link. Consider the following examples:

check new *Shot*(**new** *Posn*(88,17)).*move*()
expect new *Shot*(**new** *Posn*(88,14))
check new *Shot*(**new** *Posn*(88,14)).*move*()
expect new *Shot*(**new** *Posn*(88,11))
check new *Shot*(**new** *Posn*(88,11)).*move*()
expect new *Shot*(**new** *Posn*(88,8))

It expresses that an instance of *Shot* moves at a constant speed of 3 pixels upwards. Let's assume that it doesn't matter whether a shot is visible on the canvas or not. That is, the height of a shot can be negative for now. Then the method definition is straightforward:

<u>inside of *Shot* :</u>
// lift **this** shot by 3 pixels
Shot move() {
  return new *Shot*(**new** *Posn*(*this.location.x*,*this.location.y* $-$ 3));
}

In particular, the definition doesn't create any wishes and doesn't require any auxiliary methods. You can turn the examples into tests and run them immediately.

Exercise

Exercise 16.11 Modify the *move* methods in *IShots* and implementing classes so that if a shot has a negative *y* coordinate, the resulting list doesn't contain this shot anymore. Write up a brief discussion of your design alternatives. ∎

It is finally time for the addition of some action to our game. Specifically, let's add methods that deal with the firing of a shot. As always, we start with the *UFOWorld* class, which represents the entire world, and check whether we need to add an appropriate method there. The purpose of *UFOWorld* is to keep track of all objects, including the list of shots that the player has fired. Hence, when the player somehow fires another shot, this shot must be added to the list of shots in *UFOWorld*.

The purpose statement and method signature for *shoot* in figure 60 express just this reasoning. Here is the template for the method:

<u>inside of *UFOWorld* :</u>
UFOWorld shoot() {
  . . . **this**.*BACKG* . . . **this**.*HEIGHT* . . . **this**.*WIDTH* . . .
  . . . **this**.*ufo.shoot*(. . .) . . .
  . . . **this**.*aup.shoot*(. . .) . . .
  . . . **this**.*shots.shoot*(. . .) . . .
}

It is of course just a copy of the template for *draw* with new names for the methods on contained objects. But clearly, *BACKG*, *HEIGHT*, *WIDTH*, and *ufo* don't play a role in the design of *shoot*. Furthermore, while the *shots* are involved, they don't shoot; it is the *aup* that fires the shot. All this, plus the problem analysis, suggests this code fragment:

inside of *UFOWorld* :
UFOWorld shoot() {
  return
   new *UFOWorld*(**this**.*ufo*,
                 this.*aup*,
                 new *ConsShots*(**this**.*aup*.*fireShot*(. . .),**this**.*shots*));
}

In other words, firing a shot creates a new *UFOWorld* with the same *ufo* and the same *aup* but a list of shots with one additional shot. This additional shot is fired from the *aup*, so we delegate the task of creating the shot to an appropriately named method in *AUP*.

    The screen shot on page 56 suggests that the *AUP* fires shots from the center of its platform. That is, the shot visually originates from the stubby, vertical line on top of the moving platform. This implies the following refinement of the purpose statement and method signature:

inside of *AUP* :
// create a shot at the middle of **this** platform
Shot fireShot(. . .) { . . . }

Thus, with an *AUP* such as this:

  new *AUP*(30)

in a conventional world, we should expect a shot like this:

  new *Shot*(**new** *Posn*(42,480))

or like that:

  new *Shot*(**new** *Posn*(42,475))

The 42 says that the left corner of the shot is in the middle of the *AUP*; the 480 and the 475 say that the top of the shot is somewhere above the *AUP*. The true expected answer depends on your boss and the visual appearances that you wish to achieve. For now, let's say the second answer is the one your manager expects.

    The third step is to develop the template for *fireShot*:

inside of *AUP* :
Shot fireShot(. . .) {
  . . . **this**.*location* . . . **this**.*aupColor* . . .
}

The template contains references to the two fields of *AUP*: its *x* coordinate and its color. While this is enough data to create the *x* coordinate of the new shot, it doesn't help us with the computation of the *y* coordinate. To do so, we need knowledge about the world. Following our above design considerations, the method needs a *UFOWorld* parameter:

> inside of *AUP* :
> *Shot fireShot(UFOWorld w)* {
>   **return new** *Shot*(**new** *Posn*(**this**.*location* + 12,*w.HEIGHT* − 25));
>   }

Again, consider naming the constants in this expression for the benefit of future readers.

With *fireShot* in place, we can finally develop some examples for *shoot* and complete its definition. Here are some of the examples in section 6.2:

> *AUP a* = **new** *AUP*(90);
> *UFO u* = **new** *UFO*(**new** *Posn*(100,5));
> *Shot s* = **new** *Shot*(**new** *Posn*(112,480));
> *IShots le* = **new** *MtShots*();
> *IShots ls* = **new** *ConsShots*(*s*,**new** *MtShots*());
> *UFOWorld w1* = **new** *UFOWorld*(*u*,*a*,*le*);
> *UFOWorld w2* = **new** *UFOWorld*(*u*,*a*,*ls*);

In this context, you can create several examples for *shoot*, including

> **check** *w1.shoot*()
> **expect new** *UFOWorld*(*u*,*a*,**new** *ConsShots*(**new** *Shot*(102,475),*le*))

Create additional functional examples and create a thorough test suite.

Exercises

Exercise 16.12 Design *move* for *AUP*. The method consumes a *String s*. If *s* is "left", the method moves the *AUP* by 3 pixels to the left; if *s* is "right", the *AUP* moves 3 pixels to the right; otherwise, it remains as it is.

Modify the method so that if the *AUP* were to move out of bounds (of the current world), it doesn't move. Hint: Add a *UFOWorld* parameter to the method's signature. ∎

Exercise 16.13 Design the method *hit* for *IShots*. The method consumes a *UFO* and produces a boolean. Specifically, it produces true, if any of the shots on the list have hit the given *UFO*; it produces false otherwise.

To make things simple,[25] we say that a shot hits a *UFO* if the shot's location is close enough to the *UFO*, meaning the distance between the *UFO*'s center and the location of the shot is less than 10.0 pixels. ∎

Exercise 16.14 Develop the class *Player*. The class should contain examples of *UFO*s, *AUP*s, *Shot*s, and *UFOWorld*s. Also add the method *play* of no arguments. The method creates an instance of *Canvas*, uses it to draw instances of *UFOWorld*s, moves all the objects, fires a shot, and repeats these actions two or three times. If you have solved exercise 16.12, you may also want *play* to move the *AUP* left and right. ∎

As you can see from this case study, following the design recipe is highly helpful. Once you have read and analyzed the problem statement, it provides guidance and hints on how to get things done. The case study also shows, however, that you need to learn to use the steps of the design recipe in a flexible manner. No design process is perfect; the design recipe provides you the most valuable guidance in our experience.

While the design template is almost always useful, making up examples and even the statement of a precise purpose statement and method signature may have to wait until you have explored some of the auxiliary methods. For example, the design of *move* showed how you sometimes can't know the true signature of a method until you are ready to define the method's body. The design of *shoot* showed how you may have to postpone the development of examples for a method and its testing until an auxiliary method is defined. Otherwise you may not be able to make up proper examples. Still, even when you decide to reorder the steps or to explore the design of auxiliary methods before you define a method proper, you must make sure to make up examples *before* you design the method body. If you don't, you run the danger to believe what you see—and that almost never works properly.

---

[25]The notion of "a shot hitting the *UFO*" is something that you would decide on after some reflection on the geometric shapes of your objects and consultations with someone who understand geometry well. Put differently, both *Shot* and the *UFO* would come with a method that converts these objects into geometric shapes, and if the two shapes overlap in any form, we might say that the shot has hit the *UFO*.

Intermezzo 2: Methods

Intermezzo 1 summarizes the language of class and interface definitions in ProfessorJ's Beginner language. In this intermezzo, we extend this language with methods. This time, though, the language discussion requires four parts not just three: syntax, type checking, semantics, and errors.

→ an INTERFACEDEFINTION is:

    interface *InterfaceN* {
     MethodSignature;
     . . .
    }

→ a CLASSDEFINTION is:

    class *ClassN* [**implements** *InterfaceN*] {
     Type FieldN [= *Expr*];
     . . .
     ClassN(*Type FieldN*, . . .) {
      this.*FieldN* = *FieldN*;
      . . .
     }
     MethodDefinition
     . . .
    }

→ a METHODSIGNATURE is:

    Type MethodN(*Type ParameterN*, . . .)

→ a METHODDEFINITION is:

    MethodSignature {
     Statement
    }

→ METHODN, PARAMETERN are alphanumeric sequences

→ a STATEMENT is one of:

  – **return** *Expr*;

  – an **if** statement:

    if (*Expr*) {
     Statement }
    else {
     Statement }

→ an EXPR is one of:

  – constant,
   e.g., 5, true

  – primitive expressions,
   e.g., true || false

  – *ParameterN*

  – **this**

  – *Expr . FieldN*

  – *ConstructorCall*

  – *MethodCall*

→ a METHODCALL is one of:

  – a plain method call
   Expr.MethodN (*Expr*, . . .)

  – a dotted method call
   ClassN.MethodN (*Expr*, . . .)

Notes (in addition those in figure 35):

1. A class or interface definition must not use the same method name in two distinct method signatures.

2. A method signature must not use the same *ParameterN* twice.

Figure 63: The grammar of ProfessorJ: Beginner with methods

Vocabulary and Grammar for Methods

Figure 63 presents the extension of the grammar presented in Intermezzo 1 (figure 35 on page 66). The comparison of these two figures shows that Beginner contains just a few keywords for the formulation of methods, most importantly **return**, **if**, **else**, and **this**. Other than that, the extension is mostly about allowing programmers to write additional phrases, such as method signatures or **return** statements, and adding them to classes and interfaces.

Let's take a look at these changes, one by one:

1. Interfaces now contain a series of method signatures, each ending in a semicolon:

   > **interface** *InterfaceName* {
   >   *Type methodName(Type parameterName, . . .)*;
   >   . . .
   > }

   Each method signature starts with a *Type*, is followed by the *methodName*, and concludes with a parameter specification. The parameters, also called formal arguments, are sequences of types and names, separated by commas, and enclosed in parentheses.

   Constraint: An interface in Beginner must not contain two method signatures that use the same method name.

   Constraint: A method signature must not contain two identical parameter names.

2. Classes may contain method definitions. In ProfessorJ, you place such definitions below the constructor of the class:

   > **class** *ClassName* {
   >   *Type fieldName* **[= Expr]**; ...
   >   *ClassName(Type fieldName, . . .)* {
   >   *this.fieldName = fieldName*; ...
   >   }
   >   // **method definitions following:**
   >   *Type methodName(Type parameterName, . . .)* {
   >   *Statement*
   >   }
   >   . . .
   > }

Each method consists of a signature and a body, which is a single
statement enclosed in braces.

Constraint: A class in Beginner must not contain two method defini-
tions that use the same method name nor a method with the name of
the class.

If a method signature occurs inside an interface that the class **imple-
ments**, then the class must contain a method with that signature.

3. A *Statement* is one of:

    (a) a **return** statement, which is the **return** keyword followed by an
         expression:

          return *Expr*;

    (b) an **if** expression with statements:

          if (*Expr*) {
            Statement }
          else {
            Statement }

    The description is self-referential, which signals that you can nest
    *Statement*s arbitrarily deep. In the preceding chapter, we have, in par-
    ticular, seen **if** statements nested several times.

4. The language of expressions has also changed: *Expr* is one of:

    – constant, e.g., 5, true
    – primitive expressions, e.g., true || false
    – *ParameterN*, that is, the parameters in method signatures
    – **this**, which is the current object
    – *Expr . FieldN*, which is a selector expression
    – *ConstructorCall*, which we have seen before
    – *MethodCall*

    The description is now self-referential, meaning *Expr*s can be nested
    arbitrarily deep just like *Statement*s.

    Constraint: The field initialization "equations" in a class in Beginner
    must not be the expression **this**.

    A constructor call is now applied to a number of *Expr*s from this
    larger set of expressions.

Type Checking

The first intermezzo (7.3) deals with type checking in a simplistic manner, matching the simplicity of the language covered in part I. Section 13.2 briefly discusses the role of types as a tool for preventing abuses of methods and thus potential run-time errors.

A short summary of those sections is that the type checker should help you writing programs. Specifically, it should have rules that disallow the addition of true to 55 (true + 55) because this makes no sense and would only trigger a run-time error. Similarly, if a class *C* doesn't define method *m*, then your program should never contain expressions that invoke *m* on an instance of *C*.

```
import draw.*;

interface ILoc {
  int xasix();
  int yaxis();
  double dist(ILoc l);
}

class Loc implements ILoc {
  int x;
  int y;
  Loc(int x, int y) {
    this.x = x;
    this.y = y;
  }

  double dist(ILoc l) {
    return ... ● ...
  }
  ...
}
```

Figure 64: A sample program

The Java type checker guarantees all that. If a program passes its rules, you can safely assume that an expression of type boolean always produces true or false and that an expression of type *C* (a class) always reduces to an instance of *C*. In other words, the type checker prevents a whole range of potential run-time errors à la *How to Design Programs*.

Understanding how type checking works is essential if you work with a typed language. It helps you recover from type errors—if your Java program has type errors, you can't run it—but more importantly, it helps you understand what the type system proves about your program and what it doesn't.

We believe that the best way to understand a type checker is to study the design and implementation of programming languages in depth. Ideally, you should implement a type checker. Doing so, however, is an extremely complex task with the portion of Java that you know. Therefore this section presents the rules of type checking and signaling type errors with structured and stylized English.

The core of type checking programs is a collection of rules for determining the type of an expression and for making sure actual and specified types match up as appropriate. Describing these rules requires the discussion of two preliminary notions: type contexts and signature tables. A TYPE CONTEXT is simply a list of all field names with their types and all parameters with their types that surround the expression you want to check.

Say you are type checking the program in figure 64. When you focus on expression where where you see •, the type context looks like this:

1. *x* in *Loc* has type int;

2. *y* in *Loc* has type int;

3. *l* has type *ILoc*.

Note how fields come with information about the class in which they are contained; parameters don't have and don't need such information.

Signatures is something you know. Every method requires one. We say that operators such as $+$ or $||$ and constructors have signatures, too. Given that, you write down a signature table for the above program as follows:

Class (if any)	Name	Domain	Range
—	$+$	int, int	int
—	$||$	boolean,boolean	boolean
World	*onTick*	—	*World*
World	*bigBang*	int, int, double	boolean
ILoc	*xaxis*	—	int
ILoc	*yaxis*	—	int
ILoc	*dist*	*ILoc*	double
Loc	**new**	int, int	*Loc*
Loc	*dist*	*ILoc*	double

The line separates those parts of the table that originate from Java and the library from those that originate from explicit interface and class definitions. Before you proceed, take a close look at the table to understand its pieces.

Once you have a type context and a signature table, you check each kind of expression according to how it is constructed:

1. As discussed before, every constant has an obvious type.

2. To determine the type of a primitive expression, first determine the types of all sub-expressions and the signature of the operator. If the types of the sub-expressions match the types of the operator's domain (in the right order), the expression's type is the range type of the operator. If not, the type checker signals a type error.

   Example 1:

   > 32 + 10

   The sub-expressions are 32 and 10 and have the types int and int. The operator's signature is int and int for the domain, which matches the types of the sub-expressions. Therefore the type of the entire expression is int.

   Example 2:

   > (32 + 10) || true

   The sub-expressions are (32 + 10) and true. We know from the first example that the type of the first sub-expression is int; the type of the second one is boolean by the first rule. The signature of || is boolean and boolean for the domain. While the type of the second sub-expression matches the corresponding type in the operator's domain, the first one doesn't. Hence, you have just discovered a type error.

3. To determine the type of a parameter, look in the type context.

4. The type of **this** is always the name of the class in which it occurs.

5. To determine the type of a field selection *e.fieldName*, first determine the type of *e*. If it isn't the name of a class, the type checker signals an error. If it is, say *C*, and if the type context contains a field of name *fieldName* with type *T* for *C*, then the type of the entire expression is *T*.

Example 3:

> ... *l.x* ...

If this expression occurs in lieu of the ● in figure 64, then the parameter *l* has type *ILoc* according to the type context. Since ==*ILoc* is an interface and not a class, it can't support field access==, so what you see here is an example of a type error concerning field access.

If you change the program just slightly, like this:

Example 4:

```
double dist(Loc l) {
    ... l.x ...
}
```

then everything works out. The parameter *l* now has type *Loc*, which is a class and which contains a field named *x*. The field's type is int so that *l.x* has type int.

6. For a constructor call, the type checker determines the types of all the arguments. It also determines the signature of the **new** constructor in the signature table. Now, if the types of the arguments match the types of the constructor's domain (in the proper order), the expression's type is the class of the constructor. If not, the type checker signals a type error.

7. Finally, to determine the type of a method call *ec.methodName(e1,...)*, the type checker starts with the expression *ec*. Its type must be the name of a class or the name of an interface. If so, the signature can be extracted from the signature table. If not, the type checker signals an error.

   Second, the type checker determines the types of the arguments. They must match the types of the methods domain (in the proper order). If so, the expression's type is the range type from the method signature; otherwise, there is a type error.

   Example 5:

   > **new** *Loc*(10,20).*dist*(**new** *Loc*(30,40))

Here the object expression is **new** *Loc*(10,20), which by the rules for a constructor expression has type *Loc*. According to the signature table, this class contains a method called *dist* with domain *ILoc* and range double.

Next, the argument expression is **new** *Loc*(30,40), and it has type *Loc*. Since *Loc* implements *ILoc*, the types match. The type of the example expression is therefore double.

Several of these clauses use the phrase "the types match." What this means, for now, is that the types are either equal or that the first type is a class type and **implements** the second one, which is an interface type.

Type checking statements is like type checking expressions. Again, we look at statements case for case, using type contexts and signature tables:

1. A **return** statement consists of a keyword and an expression. This implies that the type checker must determine the type of the expression. After that, it must match that type and the specified range of the method that contains the **return** statement. If the types match, the statement type checks; otherwise, the type checker reports an error.

   Example 6:

   > . . . **return** 4.0; ...

   In the context figure 64, the type of 4.0 is double, which is also the range of the signature of *dist*. Hence, this **return** statement type-checks.

   In contrast, Example 7:

   > . . . **return this**.*x*; ...

   this has type *Loc* and **this**.*x* has therefore type int, which is not a match for double.[26]

2. An **if** statement is more complex than a **return** statement and, yet, type-checking it is straightforward. The type checker determines the type of the test expression. If it isn't boolean, the type checker signals a type error. If it is, the type checker makes sure that the statements

---

[26]In Java, an int is automatically converted into a double in such situations. This automatic conversion simplifies life for a programmers, but it also introduces opportunities for mistakes for novices. Therefore ProfessorJ does not match int and double.

in the two branches of the **if** statement type check (relative to the method context).

Example 8:

```
...
  if (l.xaxis()) {
    return 4.0; }
  else {
    return 10.0; }
```

Since *l* i of type *ILoc* in the given type context, *xaxis* is an available method. Accordingly, the type of *l.xaxis*() is int (why?), which isn't boolean. Therefore the type checker must report an ill-formed **if** statement, even though the two branches are properly typed.

Finally field declarations require type checking when they come with an initialization "equation." This kind of type checking computes the type of the expression on the right-hand side and matches it with the declared type of the field. "Equations" in the constructor are checked likewise.

Meaning: Evaluating Method Calls

In *How to Design Programs*, the meaning of an expression is its value, e.g., (+ 3 2) has the value 5. The process of determining the value of an expression is called EVALUATION; sometimes we also use the fancy language of an expression is REDUCED TO a value. For simple arithmetic expressions, such as (*first* (cons 1 empty)) or (+ 3 2), the laws of arithmetic suffice to determine the value. For expressions with calls to functions that you defined, we need to recall a tiny bit of algebra. Expressions are always evaluated in the context of some function (and variable) definitions, i.e., the content of the definitions window. Then if you need to evaluate an expression that is a function application, you replace it with the body (right-hand side) of the function and substitute the (values of the) arguments for its parameters.

For the Beginner language of ProfessorJ, the context in which you evaluate expressions is a sequence of classes and interfaces, optionally prefixed with imports from libraries. The values you get are either instances of classes or the same old values you know (ints, doubles, booleans, *String*s). As for the rules for evaluating an expression in ProfessorJ, they are—in principle—the same as those you know from *How to Design Programs*. Java's syntax, however, is so much more complex than Scheme's that it is difficult to formulate and use such rules in the context of this book. Instead,

we sketch the evaluation rules and practice them with examples. Together with your understanding of the evaluation of Scheme programs, this sketch should suffice for thinking about the Java programs in this book.[27]

Let's study the collection of expressions case-by-case to see how to evaluate them. Instead of going through them in order, we start with constants and constructor calls:

1. Constants such as `"hello world"`, 5, '8', true are primitive values.

2. As Intermezzo 1 shows, a constructor call evaluates to an instance of the specified class if the argument expressions are values; if not, we must reduce them to values first. We loosely use the word VALUE for these instances, too; many other texts use value for constants only.

    To jog your memory, we distinguish two kinds of constructor calls. The first concerns a class without initialization "equations."

    Example 1:

    ```
    class Dog {
      String name;
      int age;
      Dog(...) { ... }
    }
    ```

    In this context, the constructor **new** *Dog*(`"woof"`,2) call determines the instance completely and immediately. We can use the call in place of the instance.

    The second concerns a class with initialization "equations."

    Example 2:

    ```
    class Ball {
      int x;
      int y;
      IColor c = new Red();
      Ball(...) { ... }
    }
    ```

---

[27] For those readers interested in becoming good programmers, we recommend taking a college course on programming languages that demonstrates how to implement evaluation rules.

Here , the constructor call **new** *Ball*(3,2) triggers an instance creation. The resulting object has three fields, initialized to 3, 2, and an instance of *Red*, respectively. Indeed, every time we evaluate the constructor call, we get an instance like that. So for now, we can still identify the call with the instance.

 The next step is to look at two complex kinds of expressions:

3. You know how to evaluate primitive expressions from your grade school courses in arithmetic. That is, we assume that you can evaluate all those expressions that involve ints, doubles, and booleans for now, plus the operations of figure 36.

4. A method call consists of a TARGET expression, a method name, and a bunch of argument expressions:

    targetExpression.methodName(*argumentExpression1*, ...)

    It is ready to be evaluated if all these expressions are values. Proceeding from here depends on the value of *targetExpression*. If it were an int, a double, or a boolean, the evaluation would be stuck; fortunately this can't happen because of the type system. If it is *String*, we need to perform the appropriate operation (see figure 37). Otherwise it is an instance of a class in the definitions window:[28]

    new *aClass*(*someValue1*, ...)

    where *someValue1* and so on are the values that went into the construction of the instance.

    Now look up the method definition in *aClass* and evaluate the method call with the method body after substituting the parameters of the method with the values *argumentExpression1*, etc. The implicit first parameter—**this**—is replaced with the value of the target expression. The value of the method's body is the value of the method call.

    Studying this last paragraph also reveals why we call Java a complex programming language. As you know by now, the method body is a statement not an expression. This makes it difficult to calculate as easily and as simply as in algebra and arithmetic. Indeed, we can't really understand how to evaluate a method's body until we have

---

[28]We act as if classes defined in libraries are part of the definitions window.

studied the evaluation of statements. For now, we just imagine that we are dealing with **return** statements and that the value of such a statement is the value of its expression.

Say your definitions window contains this class definition:

Example 3a:

```
class Sample {
  Sample() { }

  // just a number
  int stuff() {
    return 10;
  }
}
```

If you were to evaluate 3 ∗ (**new** *Sample*().*stuff*()) you would proceed like this:

(a) You notice that 3 is a value. So your next task is to evaluate the method call, wrapped in optional parentheses.

(b) The target expression is a value, **new** *Sample*(), and therefore you look up the body of *stuff*, which is **return** 10.

(c) You replace the method call with the method body, replacing all parameters with their values. The method body, however, doesn't contain any parameters. You are left with

3 ∗ (**return** 10)

(d) Since we agreed to identify the value of a **return** statement with the value of its expression, you continue with

3 ∗ 10

and you get 30.

Naturally you can perform all these calculations in your head because this method call is simple. You do need the rules for when things get complicated.

Consider this slight variation of the class definition:

Example 3b:

```
class Sample {
  Sample() { }
```

```
  // just a number
  int stuff() {
    return 10 + 20;
  }
}
```

Evaluating the method call $3 * (\textbf{new } Sample().stuff())$ now requires the evaluation of a primitive expression inside the **return** statement:

```
3 * (new Sample().stuff())
;; reduces to
3 * (return 10 + 20)
;; reduces to
3 * (return 30)
;; reduces to
3 * 30
```

and so the value is 90 as expected.

Here is a another example:

Example 4:

```
class Rectangle {
  int w;
  int h;
  Rectangle(int w, int h) { } // conventional

  // the area of this rectangle
  int area() {
    return (this.w) * (this.h);
  }
}
```

We have surrounded the factors of the multiplication in *area* with optional parentheses to clarify the rest of the example.

After clicking *RUN*, you enter **new** *Rectangle*(3,5).*area*() in the interactions window and expect to see 15. Here is how this comes about. The target expression is a value so we can just look at the method body and substitute the parameters:

$$(\textbf{new } Rectangle(3,5). \; w) * (\textbf{new } Rectangle(3,5). \; h)$$

The method has only one parameter: **this**. As prescribed, **this** has been replace by the value of the target expression, which we identify with the constructor call. The next two expression we must evaluate are field selector expressions.

For the remaining expressions, we can now proceed more rapidly:

5. An expression that is just a parameter name (*ParameterN*) can't show up according to our rules. Such a name is just a placeholder for a value. As soon as we match up a placeholder and its actual value, we replace the name with the value.

6. The preceding remark applies to **this**, too, because **this** is just a special parameter.

7. This leaves us with expressions for field access. It is similar to a method call but simpler than that. Like a method call, a field access expression starts with a target expression. The rest, however, is just the name of some field:

    targetExpression.fieldName

    Before you proceed, you must reduce *targetExpression* to a value. Proceeding from here depends on that value. If it is an instance of a class in the definitions window, the value has roughly this shape:

    new *aClass*(*someValue1*, . . .)

    where *someValue1* and so on are the values that went into the construction of the instance.

    To finish the evaluation of the field access expression, we must distinguish between two cases of constructor calls (see intermezzo 7.2):

    (a) If the constructor call involves a class that has no initialization "equations," then the constructor call has as many argument values as there are fields in the class. It is thus possible to extract the desired field value from the list of constructor arguments.

    Example 5a:

```
class Rectangle {
  int w;
  int h;
  Rectangle(int w, int h) {
    this.w = w;
    this.h = h;
  }
}
```

The expression **new** *Rectangle*(2,5).*h* uses a constructor call as the target expression. Furthermore, the class of the constructor call uses has no initialization equations. Hence, the second argument of the call corresponds to the second field of the class, and therefore, the result of the field access expression is 5.

Given this much, we can also finish the evaluation of example 4. We had stopped at the expression

(**new** *Rectangle*(3,5). *w*) ∗ (**new** *Rectangle*(3,5). *h*)

because it required two field accesses (in the same context as example 5a). Now we know how to continue:

```
(new Rectangle(3,5). w) ∗ (new Rectangle(3,5). h)
;; reduces to
3 ∗ (new Rectangle(3,5). h)
;; reduces to
3 ∗ 5
;; reduces to
15
```

(b) If the class has initialization "equations" for fields, you need to look at the entire created target object, i.e., the value of the target expression, to extract the value of the field. This can take two different forms. On one hand, you can have an initialization "equation" directly with a field:

Example 5b:

```
class Rectangle {
  int w;
  int h;
  IColor displayColor = new Blue();
  Rectangle(int w, int h) {
    this.w = w;
    this.h = h;
  }
}
```

Here the class comes with three fields, one of which is initialized immediately and independently of the constructor (and its parameters). If a field access expression uses *displayColor*, you can read it off from the definition of the class.

On the other hand, we have seen that a class may have a field that is initialized within the constructor via a computation. The first concrete example came up on page 14.2 when we created a canvas whose size depended on the parameters of the class constructor. Another small variation on example 5a illustrates the point just as well:

Example 5c:

```
class Rectangle {
  int w;
  int h;
  int area;
  Rectangle(int w, int h) {
    this.w = w;
    this.h = h;
    this.area = w * h;
  }
}
```

If you evaluate the expression **new** *Rectangle*(2,5), you now get a class with three fields, even though your constructor call has just two arguments. To extract the *area* field, you have no choice but to look at the entire target object:

```
Rectangle(x = 2, y = 5, area = 10)
```

The field *area* is initialized to 10, which is thus the result of the field access expression.

If the value of the *targetExpression* were an int, a double, a boolean or even a *String* the evaluation would be stuck. Fortunately, this can't happen because of the type system.

To finish off the set of computation rules, we need to look at the two forms of statements we have in the Beginner language:

1. A **return** statement consists of a keyword and an expression. We have therefore agreed to identify it with the expression and to just evaluate the expression instead.

2. In contrast, an **if** statement consists of several pieces: an expression and two statements. Determining its value is still straightforward. You reduce the test expression until it is either true or false. Then you pick the appropriate statement and determine its meaning. Eventually, this process leads you to a **return** statement, and you know how to evaluate that.

   Example 7:

   ```
   if (0 <= 8 && 8 <= 9) {
     if (5 <= 8 && 8 <= 9) {
       return "large"; }
     else {
       return "small"; }
   else {
     return "not a digit"; }
   ```

   This **if** statement consists of the test expression (0 $<=$ 8 $\&\&$ 8 $<=$ 9), an **if** statement in the *then* branch, and a **return** statement in the **else** branch. The test expression reduces to (true $\&\&$ true) and thus to true. So the next statement to evaluate is

   ```
   if (5 <= 8 && 8 <= 9) {
     return "large"; }
   else {
     return "small"; }
   ```

   Here we have an **if** statement that consists of the test expression (5 $<=$ 8 $\&\&$ 8 $<=$ 9) and two **return** statements. The test expression again evaluates to true so that we are left with a return statement:

   ```
   return "large";
   ```

Since this last **return** statement contains a plain *String* value, the value of the statement is `"large"`.

Exercises

Exercise 17.1 Take a look at this program:

```
class Room {
  Loc l;
  Room(Loc l) { this.l = l; }
  // how far away is this room from (0,0)?
  double distance() { return this.l.distance(); }
}

class Loc {
  double x;
  double y;
  Loc(double x, double y) {
    this.x = x;
    this.y = y;
  }
  // how far away is this location from (0,0)?
  double distance() {
    return Math.sqrt((this.x * this.x) + (this.y * this.y));
  }
}
```

Explain how ProfessorJ determines that **new** *Room*(**new** *Loc*(3.0,4.0)) has the value 5.0. Use the numbers of the rules above for your explanation of each step. ∎

Exercise 17.2 The interface and class definitions in figure 65 sketch a phone setup where a landline may enable forwarding to a cellphone. (In some countries this mechanism reduces the cost for calls to a cellphone number.)
Explain how ProfessorJ determines that the evaluation of the expression

new *LandLine*(true,true,**new** *CellPhone*(true)).*dial*()

produces true. Use the numbers of the rules above for your explanation of each step. ∎

```
// a dialable phone line
interface IPhone {
   // dial this pone
   boolean dial();
}
```

```
class LandLine implements IPhone {
  boolean busy;
  boolean fwd;
  CellPhone fwdNumber;

  LandLine(boolean busy,
           boolean fwd,
           CellPhone fwdNumber) {
    this.busy = busy;
    this.fwdNumber = fwdNumber;
    this.fwd = fwd;
  }

  boolean dial() {
    if (this.fwd) {
      return this.fwdNumber.dial(); }
    else {
      return !(this.busy); }
    }
}
```

```
class CellPhone implements IPhone {
  boolean busy;
  CellPhone(boolean busy) {
    this.busy = busy;
  }

  boolean dial() {
    return !(this.busy);
  }
}
```

Figure 65: Another evaluation example

Syntax Errors, Type Errors, and Run-time Errors

When you click RUN, ProfessorJ checks two aspects of your program in the definitions window: its syntax and its types. Checking the grammar rules comes first. If those work out, it makes sense to check whether the program adheres to typing rules. If it does, ProfessorJ rests and waits for you to enter expressions and field definitions in the interactions window. For each of those, it also checks the consistency with grammar and typing rules.

After ProfessorJ has grammar and type-checked everything, it determines the value of the expression in the interactions window, one at a time. During this evaluation it may encounter additional errors. As you know from the section on type checking, however, there are far fewer of those than in DrScheme.

Here we enumerate some illustrative examples for all four categories, starting with a syntax mistake:

```
class Loc {
  int x;
  int y;
  Room(int x, int y) { ... }



  // is this close to ?
  int volume(int x, int x) {
    return ...;
  }
}
```

This class contains a method whose parameter list uses the same parameter name twice. This is reminiscent of the syntax errors we discussed in the first intermezzo. Run this example in ProfessorJ and carefully read its error message.

Next take a look at this grammatically correct class:

```
class Room {
  int x;
  int y;
  int area;
  IColor c;
  Room(int x, int y, AClass c) {
    this.x = x;
    this.y = y;
    this.area = x * c;
    this.c = x;
  }
}
```

The creator of this class messed up the initialization "equation" for the *area* field. The right hand side is an expression that multiplies an int (x) with a color (c). While this expression is properly formed from two parameter names and a $*$ operator, it violates the signature of $*$.

Similarly, the following two-class program is also grammatically correct but contains a type error:

```
class Room {
  int w;
  int h;
  Room(int w, int h) {
    this.w = w;
    this.h = h;
  }
  int area() { return this.w * this.h; }
}

class Examples {
  Room r = new Room();
  boolean test1 = check this.r.erah() expect 10;
  Examples() { }
}
```

The problem is that the **check** expression in *Examples* invokes a method *erah* on the target expression **this**.*r*, which has type *Room*. This latter class, however, does not support a method named *erah* and, therefore, the type checker signals an error.

```
interface IList {
  // find the ith element
  // in this list
  String find(int i);
}
```

```
class MT implements IList {
  MT() { }
  String find(int i) {
    return Util.error("no such element");
  }
}
```

```
class Cons implements IList {
  String first;
  IList rest;

  Cons(String first, IList rest) {
    this.first = first;
    this.rest = rest;
  }

  String find(int i) {
    if (i == 0) {
      return this.first; }
    else {
      return this.rest.find(i-1); }
  }
}
```

Figure 66: Run-time errors via *error*

As for run-time errors, you can run into few of those in the Beginner language. The type systems prevents most from ever happening, except

for two. First, your program may ask the computer to divide a number by 0. In general, there are some primitive operations that are not defined on all of their inputs. Division is one example. Can you think of another one from high school mathematics? Second, your program may contain a method that signals an error for certain inputs. Recall the *Dot* class from page 12.2 with its area method:

```
class Dot {
  Posn p;
  Dot(Posn p) { this.p = p; }



  double area() {
    return Util.error("end of the world");
  }
```

Its *area* method raises an error for all inputs. You may also encounter PAR-TIAL methods, which are like division in that they work for some, but not all inputs: see figure 66. There you see a data representation of lists of *String*s, with a function for extracting the *i*th element. Naturally, if *i* is too large, the method can't return such an element.

Finally, in addition to those programming mistakes that ProfessorJ can discover for you, there are also logical mistakes. Those are programming errors that don't violate the grammar, the typing rules, or the run-time conditions. They are computations that produce incorrect values, i.e., values that don't meet your original expectations. In this case, you're facing a LOGICAL ERROR. Unless you always develop tests for all possible methods and careful inspect the results of test runs, you can easily overlook those logical errors, and they may persist for a while. Even if you do test rigorously, keep in mind that the purpose of tests is to discover mistakes. They can't prove that you didn't make any.

Exercises

Exercise 17.1 Consider the following sketch of a class definition:

```
class Jet {
  String direction;
  int x;
  Jet(...) { ... }
  String control(String delta, int delta) {
    ...
  }
}
```

Is it grammatically correct (as far as shown)? Does it violate any typing rules? ∎

Exercise 17.2 The following class definition violates the typing rules:

```
class Ball {
  double RADIUS = 4.0;
  int DIAMETER = this.RADIUS;
  String w;
  Ball(int w) {
    this.w = w;
  }
}
```

Explain why, using the number of the typing rules in the subsection on type checking in this intermezzo. ∎

Exercise 17.3 The following class definition violates the typing rules:

```
interface IWeapon {
  boolean launch();
}

class Rocket implements IWeapon {
  ...
  boolean launch(int countDown) { ... }
}
```

Determine whether this sketch is grammatically correct. If so, check does it satisfy the tying rules? ∎

TODO
Should we ask students to represent all of Beginner's syntax?
Should we ask students to write a type checker?

PICTURE: should be on even page, and even pages must be on the left

Purpose and Background

The purpose of this chapter is to introduce the idea of abstraction within a class hierarchy. The focus is on creating superclasses from similar classes and on deriving subclasses when needed.

In general, students must learn that programming is *not* just the act of writing down classes and methods that work, but that programming includes reasoning about programs, "editing" them, and improving them as needed.

We assume that students are at least vaguely familiar with the idea of creating simple abstractions. That is, they should know that a programmer must define a function if two expressions are basically the same except for (pairs of distinct) values. Ideally, students should have read or studied Chapter IV of *How to Design Programs* so that they should also feel comfortable with abstracting over two procedures that look alike and using such abstractions as needed.

After studying this chapter, students should be able to create common superclasses for related and similar classes and to extend a framework of classes with subclasses

TODO

– introduce more Example stuff, plus abstracting over tests (?)

– use Java 1.5 overriding: it's okay to decrease the type in the return position (all covariant positions now): explain and use

– say: **public** is needed to implement an interface method

– do we need a section that shows how to abstract within one class? perhaps in Part II already?

– abstraction in TESTS Examples classes

III   Abstracting with Classes

Many of our classes look alike. For example, the *UFO* class has strong similarities to the *Shot* class. Both have similar purpose statements, have the same kinds of fields, and have methods that simulate a move along the vertical axis.

ProfessorJ:
Intermediate

As *How to Design Programs* already says, repetitions of code are major sources of programming problems. They typically come about when programmers copy code. If the original code contains an error, the copy contains it, too. If the original code requires some enhancement, it is often necessary to modify the copies in a similar manner. The programmers who work on code, however, are often not the programmers who copied the code. Thus they are often unaware of the copies. Hence, eliminating errors becomes a cumbersome chase for all the copies and, therefore, an unnecessarily costly process.

For these reasons, programmers should eliminate similarities whenever possible. In *How to Design Programs*, we learned how to abstract over similarities in functions and data definitions and how to reuse existing abstractions. More generally, we took away the lesson that the first draft of a program is almost never a finished product. A good programmer reorganizes a program several times to eliminate code duplications just as a good writer edits an essay many times and a good painter revises an oil painting many times. Good pieces of art are (almost) never created in a single session; this includes programs.

Class-based, object-oriented languages such as Java provide a number of abstraction mechanisms. In this chapter we focus on creating super-classes for similar classes and the derivation of subclasses from existing classes. The former is the process of abstraction; its result is an abstraction. The later is the use of an abstraction; the result is a well-designed program.

18 Similarities in Classes

Similarities among classes are common in unions. Several variants often contain identical field definitions. Beyond fields, variants also sometimes share identical or similar method definitions. In the first few sections of this chapter, we introduce the mechanism for eliminating these similarities.

18.1 Common Fields, Superclasses

Our first union (page 28) is a representation of simple geometric shapes. For convenience, the left of figure 67 reproduces the class diagram. Here are the first three lines of the three class definitions:

// a dot shape	// a square shape	// a circle shape
class *Dot*	**class** *Square*	**class** *Circle*
implements *IShape* {	**implements** *IShape* {	**implements** *IShape* {
CartPt loc;	*CartPt loc*;	*CartPt loc*;

As you can see, all three classes implement *IShape*. Each contains a *CartPt*-typed field that specifies where the shape is located. And best of all, all three use the same name.

```
                +--------+                              +--------+
                | IShape |                              | IShape |
                +--------+                              +--------+
                +--------+                              +--------+
                    |                                       |
                   / \                                     / \
                   ---                                     ---
                    |                                       |
                    |                              +------------+    +--------+
       +------------+-------------------+          | Shape      | +->| CartPt |
       |            |                   |          +------------+ |  |--------|
 +-----------+ +------------+    +------------+     | CartPt loc |--+ | int x  |
 | Dot     | | Square     |    | Circle     |     +------------+    | int y  |
 +-----------+ +------------+    +------------+          |          +--------+
 | CartPt loc |-+ | CartPt loc |--+ | CartPt loc |-+        / \
 |           | | | int s      |  | | int r      | |        ---
 +-----------+ | +------------+  | +------------+ |         |
              |                |                |    +---------+-------------+
              |                |                |    |         |             |
              |  +------------+ |                |    |         |             |
          +->| CartPt      |<-+----------------+   +-----+ +--------+   +--------+
              |------------+                       | Dot | | Square |   | Circle |
              | int x      |                       +-----+ +--------+   +--------+
              | int y      |                       |     | | int s  |   | int r  |
              +------------+                       +-----+ +--------+   +--------+
```
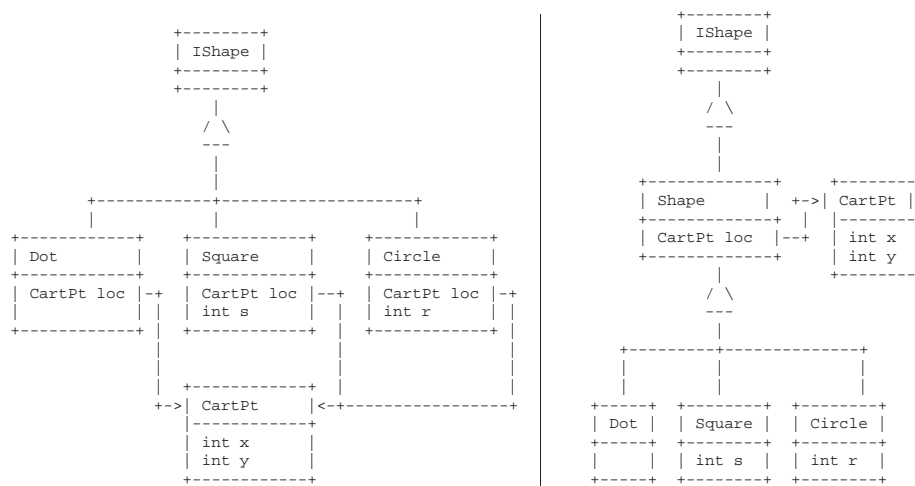
Figure 67: Lifting common fields in geometric shapes

What we want to say in such situations is that all shapes share this field and yet they represent distinct kind of things. In an object-oriented language expressing this statement is indeed possible with an enhanced form

of inheritance. More specifically, classes cannot only inherit from interfaces, they can also inherit from other classes. This suggests the introduction of a common superclass of *Dot*, *Square*, and *Circle* that represents the commonalities of geometric shapes:

```
class Shape implements IShape {
  CartPt loc;
  Shape(CartPt loc) {
    this.loc = loc;
  }
}
```

Here the class represents two commonalities: the *CartPt* field and the **implements** specification. Now, for example, if we make *Dot* an extension of *Shape*, the former inherits the *CartPt* field and the obligation to implement *IShape*:

```
// a dot shape          // a square shape        // a circle shape
class Dot               class Square             class Circle
  extends Shape {         extends Shape {          extends Shape {
```

In general, the phrase *A* **extends** *B* says that *B* inherits all of *A*'s features (fields, methods, and **implements** obligations), which include those that *A* inherited. We say that *A* is the SUPERCLASS and *B* is the SUBCLASS; we also say that *B* REFINES *A* or that *B* is DERIVED from *A*. Last but not least, **extends** is like **implements** as far as type checking is concerned; wherever the program specifies *B* (the supertype), an expressions whose actual type is *A* (the subtype) is appropriate. In short, *A* is also a subtype of *B*.

Note: In the terminology of object-oriented programming, the availability of fields and methods in subclasses is called INHERITANCE in analogy to real-life inheritance from parents to children. There are many more aspects to method inheritance than the simple example suggests. It is the purpose of this chapter to introduce method inheritance carefully as it is relevant for the systematic design of programs. ∎

The right side of figure 67 displays the revised class diagram. The new class, *Shape*, sits between the interface *IShape* and the three shape classes. That is, *Shape* refines *IShape*; *Dot*, *Square*, and *Circle* extend *Shape*. The latter contains a single field, *loc*, and because its type is *CartPt*, the diagram connects *Shape* and *CartPt* with a containment arrow. In comparison to the left, the associations between *Dot*, *Square*, and *Circle* and *CartPt*, respectively, are gone. Finally both *Square* and *Circle* still contain their other fields, which distinguishes them from each other and *Dot*.

```
// geometric shapes
interface IShape {}
```

```
// what is common to all shapes
class Shape implements IShape {
  CartPt loc;

  Shape(CartPt loc) {
    this.loc = loc;
  }
}
```

```
// a dot shape
class Dot
    extends Shape {
  Dot(CartPt loc) {
    super(loc);
  }
}
```

```
// a square shape
class Square
    extends Shape {
  int size;

  Square(CartPt loc,int size) {
    super(loc);
    this.size = size;
  }
}
```

```
// a circle shape
class Circle
    extends Shape {
  int radius;

  Circle(CartPt loc,int radius) {
    super(loc);
    this.radius = radius;
  }
}
```

Figure 68: A superclass for geometric shapes

The introduction of a superclass for geometric shapes raises the question what *Square*, for example, really looks like. We know that *Shape* contains one field and that **extends** means inherit all the fields from the superclass. Hence *Square* contains two fields: *loc* of type *CartPt* and *size* of type int. The *Square* class does not inherit *Shape*'s constructor, which leads to the question what the constructor of *Square* looks like. There are actually two answers:

```
Square(CartPt loc, int size) {             Square(CartPt loc, int size) {
  this.loc = loc;                            super(loc);
  this.size = size;                          this.size = size;
}                                          }
```

The left one initializes both fields at once, ignoring that *Square*'s declaration of the *loc* field is actually located to its superclass. The right one contains a novel construct: **super**(*loc*), known as a SUPER CONSTRUCTOR. That is, the constructor doesn't just consist of "equations," but the expression **super**(*loc*). This expression invokes the constructor of the superclass, handing over one value for the initialization of the superclass's *loc* field.

Now take a look at this expression:

    new *Square*(**new** *CartPt*(77,22),300)

No matter which constructor we use, the result is an instance of *Square*. Its location is **new** *CartPt*(77,22), and the length of its sides is 300.

   Figure 68 contains the class definitions for the revised class diagram on the right of figure 67. Note how the text completely expresses the subtle relationships among the five classes:

1. *IShape* is the interface that represents all plain geometric shapes;

2. *Shape* is the class that represents the commonalities of all plain geometric shapes;

3. *Dot*, *Square*, and *Circle* refine *Shape*. They thus inherit all fields (and future methods) from *Shape*. Furthermore, they are obliged to satisfy all obligations of *IShape*.

4. The constructors in these three classes also remind us how many fields each class has and, by using the **super** constructor, also remind us how they are alike and how they differ from the rest of the shapes.

A comparison with the original definitions suggests that we didn't save much space. But remember that this isn't the goal of the exercise. The true goal is to eliminate similar pieces of code and to express the relationships among pieces of code as precisely as possible. With the introduction of *Shape* we have succeeded in doing so. Next we need to study the precise meaning of **implements** for *Shape* because an empty interface, such as the one in figure 68, isn't good enough to understand this relationship properly.

Exercises

Exercise 18.1 Add constructors to the following six classes:

1. train schedule:

   ```
   class Train {                          class ExpressTrain extends Train {
     Schedule s;                            Stops st;
     Route r;                               String name;
   }                                      }
   ```

2. restaurant guides:

```
class Restaurant {              class ChineseRestaurant extends Restaurant {
  String name;                    boolean usesMSG;
  String price;                 }
  Place place;
}
```

3. vehicle management:

```
class Vehicle {                              class Sedan extends Vehicle {}
  int mileage;
  int price;
}
```

Assume *Schedule*, *Route*, *Stops*, and *Place* are defined elsewhere. ∎

Exercise 18.2 Abstract over the common fields of *Lion*, *Snake*, and *Monkey* in the class diagram of figure 14 (page 32). First revise the class diagram, then define the classes including the constructors. ∎

Exercise 18.3 Abstract over the common fields in the class diagram of figure 16 (page 34). Revise the class diagram and the classes. ∎

Exercise 18.4 Determine whether it is possible to abstract in the class hierarchy of exercise 4.2 (page 31). ∎

18.2   Abstract Classes, Abstract Methods

In section 12.1, we designed several methods for plain shapes, including *area*, *distTo0*, *in*, and *bb*. For unions such as those for shapes, the design of methods starts with the addition of a method signature and a purpose statement to the interface.

Let us look at this process in light of the preceding section where we added *AShape* to the same class hierarchy and said it would collect the commonalities of the concrete shapes. In addition, we specified that *AShape* **implements** *IShape* so that we wouldn't have to repeat this statement over and over again for all the classes that extend *AShape*. For *AShape* this implies that we must implement *area*, *distTo0*, *in*, and *bb* because that is what *IShape* specifies.

```
interface IShape {
  // to compute the area
  // of this shape
  double area();

  // to compute the distance of
  // this shape to the origin
  double distTo0();

  // is the given point is within
  // the bounds of this shape
  boolean in(CartPt p);

  // compute the bounding box
  // for this shape
  Square bb();
}
```

```
abstract class AShape implements IShape {
  CartPt loc;

  abstract double area();
  abstract double distTo0();
  abstract boolean in(CartPt p);
  abstract Square bb();
}
```

```
class Dot
  extends AShape {

  // cons. omitted
  ...
  double area() {
    return .0;
  }

  double distTo0() {
    return this.loc.distTo0();
  }
  ...
}
```

```
class Square
  extends AShape {
  int size;

  // cons. omitted
  ...
  double area() {
    return
      this.size *
      this.size;
  }

  double distTo0() {
    return this.loc.distTo0();
  }
  ...
}
```

```
class Circle
  extends AShape {
  int radius;

  // cons. omitted
  ...
  double area() {
    return
      Math.PI *
      this.radius *
      this.radius;
  }

  double distTo0() {
    return
      this.loc.distTo0()
      − this.radius;
  }
  ...
}
```

Figure 69: Classes for geometric shapes with methods and templates

Unfortunately, implementing methods such as *area* doesn't make any sense for *AShape* because its definition is different for each class that implements it. The two contradictory requirements—implementing *area* in *AShape* and providing a definition for every specific shape—clearly pose a problem.

To overcome this problem, object-oriented languages usually provide a mechanism for passing on the implementation requirement from classes to subclasses. In Java, the solution is to add so-called **abstract** methods to the *AShape* class. An <mark>ABSTRACT method</mark> is just like a method signature in an interface, preceded by the keyword **abstract**. The addition of an **abstract** method to a class means that the class's programmer doesn't have to define the method and thus shifts the responsibility to the developer of subclasses. Of course, it also makes no sense to create instances of the *AShape* class, because it doesn't implement all of the methods from its interface yet. To make this obvious to readers of the class definition, we make the entire class **abstract**.

Take a look at figure 69. It displays a fragment of the shape class hierarchy with methods, including the full definition of *AShape* and an interface that contains several method signature. For each signature you add to *IShape*, you also need to add an **abstract** method to *AShape*.

18.3   Lifting Methods, Inheriting Methods

What applies to fields also applies to methods. If all concrete subclasses of a union contain identical method definitions, we must lift them to the abstract class. For a concrete example, consider a class hierarchy for modeling a fleet of vehicles:

```
interface IVehicle {
  // compute the cost of refueling this vehicle,
  // given the current price (cp) of fuel
  double cost(double cp);
}

abstract class AVehicle implements IVehicle {
  double tank; // gallons
  ...
  abstract double cost(double cp);
}
```

inside of *Car* :	inside of *Truck* :	inside of *Bus* :
double *cost*(double *cp*) {	double *cost*(double *cp*) {	double *cost*(double *cp*) {
return this.*tank* ∗ *cp*;	**return this**.*tank* ∗ *cp*;	**return this**.*tank* ∗ *cp*;
}	}	}

The union has three variants: for cars, trucks, and buses. Since each variant needs to describe the tank size of the vehicle, there is a common, abstract superclass with this field. Each variant separately comes with a method for calculating the cost of filling the tank.[29] The *cost* methods have been designed systematically and are therefore identical.

When the exact same method definition shows up in all the variants of a union with a common superclass, you can replace the **abstract** method in the superclass with the method definition from the variants:

```
abstract class AVehicle {
  double tank;
  . . .
  double cost(double cp) {
    return this.tank ∗ cp;
  }
}
```

Furthermore, you can delete the methods from *Car*, *Truck*, and *Bus* because the lifted *cost* method in *AVehicle* is now available in all three subclasses.

Naturally, after editing a class hierarchy in such an intrusive manner, we must re-run the test suite just to make sure we didn't accidentally introduce a mistake. Since the test suites for *AVehicle* construct instances of *Car*, *Truck*, and *Bus* and then use *cost*, and since all three classes inherit this method, the test suite should run without error as before.

Exercise

Exercise 18.5 Develop examples for the original class hierarchy representing vehicles. Turn them into tests and run them. Then lift the common *cost* method and re-run the tests. ∎

For a second example, let's return to plain geometric shapes:

---

[29]For illustrative purposes, the methods use double to represent the cost of a tank and the price of gasoline. For realistic applications, doubles may not be precise enough.

```
interface IShape {
  // compute the area of this shape
  double area();
  // is this shape larger than that shape
  boolean larger(IShape that);
}

abstract class AShape implements IShape {
  CartPt loc;
  . . .
  abstract double area();
  abstract boolean larger();
}
```

Here the interface specifies just two methods: *larger* and *area*. The former determines whether **this** shape is larger than some other shape, and the latter actually computes the area of a shape. Here are the definitions of *area* and *larger*:

inside of *Dot* :	inside of *Square* :	inside of *Circle* :
	int *size*;	int *radius*;
	. . .	. . .
boolean *larger(IShape that)* {	boolean *larger(IShape that)* {	boolean *larger(IShape that)* {
return	**return**	**return**
this.*area()*	**this**.*area()*	**this**.*area()*
>	>	>
that.area();	*that.area()*;	*that.area()*;
}	}	}
double *area()* {	double *area()* {	double *area()* {
return 0;	**return**	**return**
}	**this**.*size* ∗	*Math.PI* ∗
	this.*size*;	**this**.*radius* ∗
	}	**this**.*radius*;
		}

We have seen *area* before and there is nothing new to it. The *larger* method naturally takes advantage of the *area* method. It computes the area of **this** shape and the other shape and then compares the results. Because this follows the "one task, one function" rule of design, all three definitions of *larger* are identical.

Figure 70 shows the result of lifting *larger* to *AShape*. Now the **abstract** superclass not only contains a shared *CartPt* field but also a shared method. Interestingly the latter refers to another, **abstract** method in the same class but this is acceptable because all shapes, abstract or concrete, implement *IShape*, and *IShape* demands that shapes define an *area* method.

```
interface IShape {
  // to compute the area of this shape
  double area();

  // to determine whether this
  // is larger than other (in area)
  double larger(IShape other);
}
```

```
abstract class AShape
    implements IShape {
  ...
  abstract double area();

  double larger(IShape other) {
    return this.area() > other.area();
  }
}
```

```
class Dot
  extends AShape {

  // cons. omitted
  ...
  double area() {
    return .0;
  }
  ...
}
```

```
class Square
  extends AShape {
  int size;

  // cons. omitted
  ...
  double area() {
    return
      this.size *
      this.size;
  }
  ...
}
```

```
class Circle
  extends AShape {
  int radius;

  // cons. omitted
  ...
  double area() {
    return
      Math.PI *
      this.radius *
      this.radius;
  }
  ...
}
```

Figure 70: Geometric shapes with common lifted method

Methods don't have to be identical for lifting. If at least some of the methods in a union are identical, it is still possible to place one copy in the abstract class and inherit—in *most* but not necessarily all subclasses. Those that need a different version of the method define one and thus OVERRIDE or replace the version from the superclass.

Take a look at the shape representation in figure 69. The three concrete classes all define *distTo0* methods. Compare the methods in *Dot* and *Square*. The two are identical. The one in *Circle*, however, differs from those two,

because the closest point to the origin is on the perimeter of the circle.[30]

```
interface IShape {
  // to compute the distance of
  // this shape to the origin
  double distTo0();
  ...
}
```

```
abstract class AShape implements IShape {
  CartPt loc;

  double distTo0() {
    return this.loc.distTo0();
  }
  ...
}
```

```
class Dot
  extends AShape {

  // cons. omitted
  ...
}
```

```
class Square
  extends AShape {
  int size;

  // cons. omitted
  ...
}
```

```
class Circle
  extends AShape {
  int radius;

  // cons. omitted

  double distTo0() {
    return
      this.loc.distTo0()
      − this.radius;
  }
  ...
}
```

Figure 71: Classes for geometric shapes with methods and templates

---

In object-oriented programming languages the common method may be defined in the abstract class. As before, every subclass inherits this definition. For those subclasses for which this method definition is wrong, you (the programmer) override it simply by defining a method with the same[31] method signature. Figure 71 shows the result of transforming the classes in figure 69 in this manner. The class *AShape* defines *distTo0* as the distance of *loc* to the origin. While *Dot* and *Square* inherit this method, *Circle* overrides it with the one that makes sense for circles.

Of course, in some sense the distance to the origin of all shapes depends on the distance to the origin of their anchor point. It just so happens that the distance between the origin and the anchor point of, say, a *Square* is the

---

[30]Remember our assumption that the shapes are completely visible on the canvas.

[31]In Java, you can use a signature whose return type is a subtype of the overridden method; in other object-oriented languages, the signatures have to be identical.

```
interface IShape {
  // to compute the distance of
  // this shape to the origin
  double distTo0();
  ...
}
```

```
abstract class AShape implements IShape {
  CartPt loc;

  double distTo0() {
    return this.loc.distTo0();
  }
  ...
}
```

```
class Dot
  extends AShape {

  // cons. omitted
  ...
}
```

```
class Square
  extends AShape {
  int size;

  // cons. omitted
  ...
}
```

```
class Circle
  extends AShape {
  int radius;

  // cons. omitted

  double distTo0() {
    return
      super.distTo()
      − this.radius;
  }
  ...
}
```

Figure 72: Classes for geometric shapes with methods and templates

distance between the shape and the origin whereas the distance between the origin and the center of a *Circle* is not. In a way, this is just what inheritance is about. The child is somewhat like the parents, but not the same. As it turns out, this idea is important in programming too, and you can express this subtle but important relationship between children and parents in object-oriented programming languages.

Figure 72 shows how to express this like-but-not-the-same relationship in classes. The gray-shaded expression is a SUPER METHOD call. The **super** refers to the superclass of *Circle*, which is *AShape*. The expression invokes the *distTo0* method in *AShape* (on **this** circle) and thus computes the distance of the anchor point of the circle to the origin; the rest of the expression subtracts the length of the *radius*, as before.

Because of the simplicity of *distTo0*, expressing the relationship as carefully as we have done here appears to be pedantic. These appearances are deceiving, however. Programming is about subtleties and expressing them

is important. It is natural in our profession that other people will read your program later and modify it. The clearer you can communicate to them what your program is about, the higher the chances are that they don't mess up these carefully thought-through relationships.

Exercise

Exercise 18.6 Complete the class hierarchy for overlapping shapes from section 15.3: *IComposite*, *Square*, *Circle*, and *SuperImp* with all the methods: *area*, *distTo0*, *in*, *bb*. Add the following methods to the class hierarchy:

1. *same*, which determines whether **this** shape is of equal size as some other, given *IShape* up to some given small number *delta*;

2. *closerTo*, which determines whether **this** shape is closer to the origin than some other, given *IShape*;

3. *drawBoundary*, which draws the bounding box around **this** shape.

Develop examples and tests for as many methods as possible. Then introduce a common superclass for the concrete classes and try to lift as many fields and methods as possible into this superclass. Make sure that the revised classes function properly after this editing step. ∎

18.4   Creating a Superclass, Creating a Union

When you develop independent classes for a program, you may end up with two (or more) classes that look alike but aren't the variants of a union. Perhaps it wasn't clear when you started out that the two classes of objects belong together, except that a second look shows that they are related after all. With a bit of renaming, they may have the same fields and a number of similar methods. The only way to eliminate these similarities is to create a union including a common superclass.

Recall the weather reporting example from figures 11 (page 26) and 22 (page 43). The class hierarchies in these figures are excerpts from a program that records information about the weather. It always includes information about the temperature (degrees, Fahrenheit) and precipitation (mm) but often also information about the air pressure (hPa[32]). Each recording typically includes the historical high, the historical low, and today's value.

---

[32]Air pressure is measured in hectopascal or hundreds of Pascals.

```
// recording air pressure
// measurements [in hPa]
class Pressure {
  int high;
  int today;
  int low;
  Pressure(int high,int today,int low) {
    this.high = high;
    this.today = today;
    this.low = low;
  }

  int dHigh() {
    return this.high − this.today;
  }

  int dLow() {
    return this.today − this.low;
  }

  String asString() {
    return String.valueOf(high)
           .concat("-")
           .concat(String.valueOf(low))
           .concat("hPa");
  }
}
```

```
// recording temperature
// measurements [in F]
class Temperature {
  int high;
  int today;
  int low;
  Temperature(int high,int today,int low) {
    this.high = high;
    this.today = today;
    this.low = low;
  }

  int dHigh() {
    return this.high − this.today;
  }

  int dLow() {
    return this.today − this.low;
  }

  String asString() {
    return String.valueOf(high)
           .concat("-")
           .concat(String.valueOf(low))
           .concat("F");
  }
}
```

Figure 73: Recording weather information

Figure 73 shows an elaborated version of two classes in this program: *Pressure* and *Temperature*. Each class comes with the three expected fields and three methods. The first two methods compute the difference between today's measured value and the historical high and low, respectively. The third method turns the recording into a *String*, which includes the historical range and the physical unit of the measurement.

Clearly, the two classes have a similar purpose and are identical up to the unit of measurement. With a common superclass, we can obviously eliminate these similarities. The superclass introduces the three common fields and the three common methods; the only remaining problem is how to deal with *asString*, the method that renders the recording as text.

```
class Recording {
  int high;
  int today;
  int low;
  Recording(int high,int today,int low) {
    this.high = high;
    this.today = today;
    this.low = low;
  }

  int dHigh() {
    return this.high − this.today;
  }

  int dLow() {
    return this.today − this.low;
  }

  String asString() {
    return
      String.valueOf(high).concat("-").concat(String.valueOf(low));
  }
}
```

```
class Pressure extends Recording {
  Pressure(int high,int today,int low) {
    super(high,today,low);
  }

  String asString() {
    return super.asString()
            .concat("mm");
  }
}
```

```
class Temperature extends Recording {
  Temperature(int high,int today,int low) {
    super(high,today,low);
  }

  String asString() {
    return super.asString()
            .concat("F");
  }
}
```

Figure 74: Recording weather information with a superclass (version 1)

The first solution is to mimic what we have seen in the previous section. It is shown in figure 74, which displays the new common superclass and the revised versions of the original classes. The latter two extend *Recording* and thus inherit its three fields (*high*, *today*, and *low*) as well as its three methods (*dHigh*, *dLow*, and *asString*). The *asString* method produces a string that

```
class Recording {
  int high;
  int today;
  int low;
  String unit;
  Recording(int high,int today,int low,String unit) {
    this.high = high;
    this.today = today;
    this.low = low;
    this.unit = unit;
  }

  int dHigh() { … }

  int dLow() { … }

  String asString() {
    return
      String.valueOf(high).concat("-").concat(String.valueOf(low)).concat(this.unit);
  }
}
```

```
class Pressure extends Recording {
  Pressure(int high,int today,int low) {
    super(high,today,low,"mm");
  }
}
```

```
class Temperature extends Recording {
  Temperature(int high,int today,int low) {
    super(high,today,low,"F");
  }
}
```

Figure 75: Recording weather information with a superclass (version 2)

separates *high* and *low* with a dash (−). Each subclass overrides the *asString* method to compute a representation that also includes the physical units. Note how the two subclasses use a **super** constructor call to initialize the objects. This creates an instance of *Pressure* or *Temperature* but initializes the fields they inherit from *Recording*.

The use of the **super** constructor also suggests an alternative solution, which is displayed in figure 75. For this second solution, *Recording* contains a complete definition of *asString* and the additional, *String*-typed field *unit*. The *asString* method uses this new field as the name of the physical unit. The constructors in the subclasses have the same signature as those in the original class but supply an appropriate unit name as the fourth argument

David says: the discussion of the two alternative developments of Recording states that a disadvantage of the second approach is that the Recording can be used in place of Pressure and Temperature, and Java cannot help to enforce consistent use of these classes. But this is a disadvantage of the first approach as well.

use the word in iv, too. using stateful classes or applicative one is one of a trade-off analysis

to the **super** constructor.

Compared to the first solution, this second solution has one advantage and one disadvantage. The advantage of this second solution is that it *forces* a programmer who introduces another subclass of *Recordings* to supply a *String* for the physical units; nothing is left to voluntary action. In contrast, the first solution relies on the programmers to override via a **super** call the *asString* method to produce a proper *String* representation. The biggest disadvantage of this second solution is that programmers can use *Recordings* in place of *Temperature* and *Pressure* (and other, yet-to-be-defined classes) that represent recordings of measurements. In turn, Java can't help check whether your programs use *Temperature* and *Pressure* consistently. In short, if you can use types to express distinctions, do so.

This DESIGN TRADE-OFF analysis shows that we really want a combination of the first and second solution. On one hand, we want a method in the superclass that performs the rendering to text; on the other hand, we want to force a subclassing programmer to supply the unit string. A moment's thought tells us that this is what **abstract** is all about. So one solution would be to make *Recording* an **abstract** class, without any other change, and thus disallow anyone from creating instances from it. But we can do even better with the addition of an **abstract** method to this **abstract** class. Adding an **abstract** method to a class forces the programmer who **extends** the class to reflect on its purpose and to supply a definition.[33]

Figure 76 displays the **abstract** version of *Recording*. Like version 1, it contains a definition of *asString* that glues together the *high* and the *low* with a dash. Like version 2, it also supplies the string that represents the unit, but via the invocation of the **abstract** method *unit*. Since *ARecording* is **abstract**, it is no longer possible to represent measurements directly with this class. Instead, a programmer must define a new subclass. To make the subclass concrete, it must contain a definition of *unit*—just like *Pressure* and *Temperature*. In short, this third solution really combines the advantages of the first and second solution.

Over the years, this organization of methods and classes, which we already encountered in a simpler manner with *larger* and *area* in figure 70, has become known as the "template and hook" pattern.[34] The *asString* method

---

[33]We really just need an abstract *field* not method to hold the unit but Java and most object-oriented languages don't support this mechanism.

[34]The idea of collecting programming patterns emerged in the 1990s. Gamma, Helm, Johnson, and Vlissides conducted "software archaeology," uncovered recurring patterns of programming, and presented them in a stylized manner so that others could benefit from them. Our design recipes emerged during the same time but are based on theoretical

```
abstract class ARecording {
  int high;
  int today;
  int low;

  ARecording(int high,int today,int low) {
    this.high = high;
    this.today = today;
    this.low = low;
  }

  int dHigh() { … }

  int dLow() { … }

  String asString() {
    return
      String.valueOf(high).concat("-").concat(String.valueOf(low)).concat(this.unit());
  }

  abstract String unit();
}
```

```
class Temperature extends ARecording {
  Temperature(int high,int today,int low) {
    super(high,today,low);
  }

  String unit() {
    return "F";
  }
}
```

```
class Pressure extends ARecording {
  Pressure(int high,int today,int low) {
    super(high,today,low);
  }

  String unit() {
    return "hPa";
  }
}
```

Figure 76: Recording weather information with a superclass (version 3)

in the superclass is a template that lays out what the string basically looks like, up to a hole; the *unit* method is a hook that empowers and forces sub-classes to fill the hole.

The template-and-hook pattern can occur more than once in a class hi-

considerations mixed with our Scheme programming experience. The critical point is that following the design recipe often produces code just like the one that software patterns suggest in the same situation.

```
                    +---------------+
                    | ARecording    |
                    +---------------+
                    | int high      |
                    | int today     |
                    | int low       |
                    +---------------+
                    | int dHigh()   |
                    | int dLow()    |
                    | String        |
                    |  asString()   |
                    | String unit() |
                    +---------------+
                          |                //
                         / \               //
                         ---               //
                          |                //
              --------------------------//-----------
              |               |       //          |
     +-------------+ +---------------+ // +---------------+
     | Pressure    | | ATemperature  | // | Precipitation |
     +-------------+ +---------------+ // +---------------+
     +-------------+ +---------------+ // +---------------+
     |String unit()| | String unit() | // | String unit() |
     +-------------+ | String name() | // +---------------+
                     +---------------+ //
                           |           //
                          / \          //
                          ---          //
                           |
                  --------------------
                  |                  |
         +-------------+    +---------------+
         | Celsius     |    | Fahrenheit    |
         +-------------+    +---------------+
         +-------------+    +---------------+
         |String name()|    | String name() |
         +-------------+    +---------------+
```
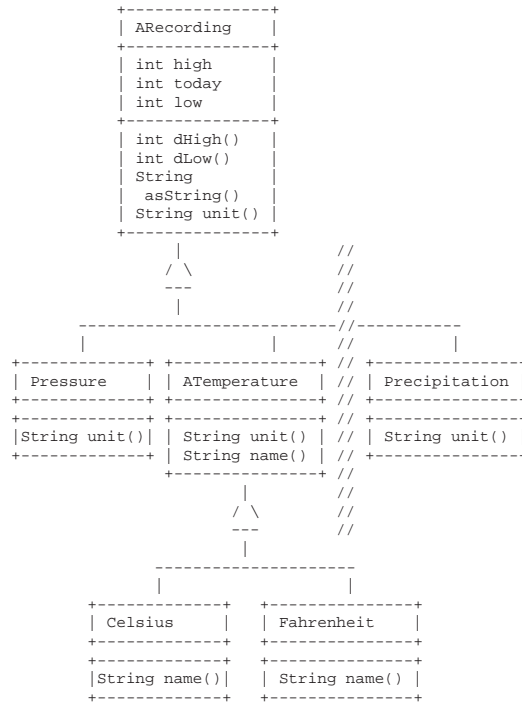
Figure 77: A class hierarchy for recordings

erarchy, and a class hierarchy may contain more than one level of classes. Suppose your manager wishes to supply temperature measurements in both Fahrenheit and Celsius degrees. In that case, a simple *Temperature* class is inappropriate. Instead, you should introduce two classes: one for Celsius and one for Fahrenheit measurements. At the same time, the two classes share a lot of things. If you were requested to add a temperature-specific method, they would probably share it, too. Put differently, as far as we can tell now, they share everything except for the name of the physical unit. Thus, the best course of action is to turn *Temperature* into an abstract class and extend it with a subclass *Celsius* and a subclass *Fahrenheit*.

Now take a look at the class diagram in figure 77. The diagram has three levels of classes, the second level extending the first, and the third extending the second. Furthermore, the extension of *ARecording* to *ATemperature* is a template-and-hook pattern, and so is the extension of *ATemperature* to

Celsius or *Fahrenheit*.

Let's inspect the actual code. Here is the new abstract class:

```
abstract class ATemperature extends ARecording {
  ATemperature(int high,int today,int low) {
    super(high,today,low);
  }

  String unit() {
    return " degrees ".concat(this.name());
  }

  abstract String name();
}
```

The class defines *unit* but only as a template, with a hole of its own. The hook for *unit* is *name*, a method that names the degrees in the string representation. As before, making *name* abstract means that the entire class is abstract and that a subclass must define the method, if it is to be concrete. The *Celsius* and *Fahrenheit* classes are concrete:

```
class Celsius                     class Fahrenheit
   extends ATemperature {            extends ATemperature {
 Celsius(int high, int today,int low) {  Fahrenheit(int high,int today,int low) {
  super(high,today,low);            super(high,today,low);
 }                                 }

 String name() {                   String name() {
  return "Celsius";                 return "Fahrenheit";
 }                                 }
}                                 }
```

The two *name* methods in the two subclasses just return "Celsius" and "Fahrenheit", respectively. Indeed, the two are so alike that they call for more abstraction, except that most programming languages don't have the mechanisms for abstracting over these similarities.[35]

Once you have a properly organized program, modifications and extensions are often easy to add. Figure 77 indicates one possible extension: a

---

[35]Physical and monetary units are ubiquitous fixtures in data representations. In principle, a programming language should provide mechanisms for annotating values with such units, but so far no mainstream programming language does. The problem of adding units to languages is complex; programming language researchers have been working on this problem for two decades.

class for measuring *Precipitation*. Measuring the amount of rain and snow is a favorite and basic topic of meteorology. Also, people love to know how much it rained on any given day, or when they travel somewhere, how much rain they may expect according to historical experience. There is a difference between precipitation and temperature, however. While the former has a natural lower bound—namely, 0—the latter does not, at least as far as people are concerned.[36]

Within the given framework of classes, creating *Precipitation* measurements is now just a matter of extending *ARecording* with a new subclass:

```
// recording precipitation measurements [in mm]
class Precipitation extends ARecording {
  Precipitation(int high,int today) {
    super(high,today,0);
  }

  // override asString to report a maximum value
  String asString() {
    return "up to ".concat(String.valueOf(high)).concat(this.unit());
  }

  // required method
  String unit() {
    return "mm";
  }
}
```

This supplies every field and method that comes with a recording. Still, because of the special nature of precipitation, the constructor differs from the usual one. It consumes only two values and supplies the third one to the **super** constructor as a constant. In addition, it overrides the *asString* method completely (without **super**), because an interval format is inappropriate for reporting precipitation.

Here is another possible change request:

> ... The string representation of intervals should be as mathematical as possible: the low end followed by a dash and then the high end of the interval. ...

_____

[36]Technically, $0^o K$ is the absolute zero point for measuring temperatures, but we can safely ignore this idea as far as meteorology information for people is concerned.

In other words, the goal is to invert the presentation of the measurements. Of course, now that we have a well-organized class hierarchy, you just change the definition of *asString* in the top-most class and the rest follows:

inside of *ARecording* :

String asString() {
  return *String.valueOf*(*low*)
        .*concat*("-")
        .*concat*(*String.valueOf*(*high*))
        .*concat*(*this.unit*());
}

All subclasses—*Pressure*, *ATemperature*, *Precipitation*, *Celsius*, and *Fahrenheit*—inherit the modified computation or override it appropriately.

The lesson of this exercise is again that creating a single point of control for a single piece of functionality pays off when it is time to change the program.

Exercises

Exercise 18.7 Figure 77 introduces a union from two classes that had been developed independently. Create an interface for the union. ∎

Exercise 18.8 Add a class for measuring precipitation to the class hierarchy of recordings (version 1) in figure 74 . Also add a class for measuring precipitation to the second version of the class hierarchy (figure 75). Compare the two hierarchy extensions and the work that went into it. ∎

Exercise 18.9 Add a class for measuring the speed of the wind and its direction to all three class hierarchies in figures 74 , 75, and 76. ∎

Exercise 18.10 Design *newLow* and *newHigh*. The two methods determine whether today's measurement is a new historical low or high, respectively. ∎

Exercise 18.11 Design the methods *fahrenheitToCelsius* and *celsiusToFahrenheit* for *Fahrenheit* and *Celsius* in figure 77, respectively. ∎

Exercise 18.12 A weather station updates weather recordings on a continuous basis. If the temperature crosses a particular threshold in the summer, a weather station issues a warning. Add the method *heatWarning* to the class hierarchy in figure 77. The method produces true if today's temperature exceeds a threshold, false otherwise. For Fahrenheit measurements, the threshold is $95^{o}F$, which corresponds to $35^{o}C$. ∎

Exercise 18.13 If the air pressure falls below a certain threshold, a meteorologist speaks of a "low," and if it is above a certaing level, the weather map display "high." Add the methods *lowPressure* and *highPressure* to the class hierarchy in figure 77. Choose your favorite thresholds. ∎

Abstracting is a subtle and difficult process. It requires discipline and experience, acquired via practice. Let's therefore look at a second example, specifically the "star thaler" problem from chapter II:

> . . . Develop a game based on the Grimms brothers' fairy tale called "Star Thaler." . . . Your first task is to simulate the movement of the falling stars. Experiment with a single star that is falling to the ground at a fixed number of pixels per time unit on a 100×100 canvas. Once the star has landed on the ground, it doesn't move anymore. . . .

After some experimentation your programming team finds that the game may sell better if something unforeseen can happen:

> . . . Modify the game so that it rains red rocks in addition to golden star thalers. If the girl is hit by one of the rocks, her energy decreases. Assume that a red rock has the same shape as a star but is red. . . .

It is your task to modify your program so that your managers can visualize the game with thalers and red rocks.

Figure 78 shows the result in two columns. The left column is the class of star thalers, as partially developed in chapter II; the right column contains the class of red rocks. The classes have been arranged so that you can see the similarities and the differences. Here are the obvious similarities:

1. Both classes have *x* and *y* of type int.

2. Both classes have identical *landed* methods.

3. Both classes have identical *nearGround* methods.

And there are also two less obvious connections between the classes:

4. The *Star* class contains a *DELTA* field for specifying how far a star thaler drops at each step; the analogous field in *RedRock* is *deltaY*. If we systematically rename *DELTA* to *deltaY* in *Star* (see gray highlighting), the two fields are also an obvious overlap between the two classes.

```
class Star {                          class RedRock {
  int x = 20;                           int x;
  int y;                                int y;
  int DELTA = 5;                        int deltaX = 3;
                                        int deltaY = 5;


  Star(int y) {                         RedRock(int x,int y) {
    this.y = y;                           this.x = x;
  }                                       this.y = y;
                                        }
  boolean draw(Canvas canv) {           boolean draw(Canvas canv) {
  … new Yellow() … }                     … new Red()) … }
  Star drop() {                         RedRock drop() {
    if (this.landed()) {                  if (this.landed()) {
      return this; }                        return this; }
    else { if (this.nearGround()) {       else { if (this.nearGround()) {
      return                                return
        new Star(100); }                      new RedRock(this.x,100); }
    else {                                else {
      return                                return
        new                                   new RedRock(this.x + this.deltaX,
          Star(this.y + this.DELTA); }                  this.y + this.deltaY); }
    }                                     }
  }                                     }
  boolean landed() {                    boolean landed() {
    return this.y == 100;                 return this.y == 100;
  }                                     }
  boolean nearGround() {                boolean nearGround() {
    return this.y + this.DELTA > 100;     return this.y + this.deltaY > 100;
  }                                     }
}
                                        boolean hit(Girl g) {
                                          return
                                            g.closeTo(new Posn(this.x,this.y));
                                        }
                                      }
```

Figure 78: Falling objects

5. Both classes also contain similar but not identical *draw* methods. One uses yellow as the drawing color, the other one uses red.

What distinguishes the two classes are the *drop* methods. While a star thaler

goes straight down, a rock drops in a diagonal line. For every five pixels that a red rock drops, it moves three to the right. Finally, *RedRock* contains one method more than *Star*: *hit*. It determines whether the rock has hit the girl.

Exercises

Exercise 18.14 The code in figure 78 comes without any purpose statements and without tests. Write down concise purpose statements for all methods, and develop a test suite for both classes. Use this definition for *Girl*:

```
class Girl {
   Girl() { }
   // is this girl close to Posn p?
   boolean closeTo(Posn p) {
     return false;
   }
}
```

This is called a stub implementation. ∎

Exercise 18.15 The method definitions for *draw* in figure 78 contain **new** expressions for colors. Introduce a name for each of these colors. Don't forget to run the test suites from exercise 18.14. ∎

The abstraction of these classes appear in figure 79. It is a superclass that represents falling objects in general; the revised versions of *Star* and *RedRock* are derived from this superclass and thus inherit its fields and methods. Based on our comparison, the superclass contains three common methods: *landed*, *nearGround*, and *draw*. It also contains the following common fields:

1. *x* and *y*, because they are common to both;

2. *deltaY*, which generalizes the *DELTA* in *Star* and the *deltaY* in *RedRock*;

   Both fields name the rate at which the respective objects drop. To create a common superclass, we have to rename one of them throughout the class. Here we chose to rename *DELTA* to *deltaY*, and the renaming is visible in the *drop* method in *Star*.

```
class Falling {
  int x;
  int y;
  int deltaY;
  IColor c;

  Falling(int x,int y,int deltaY,IColor c) {
    this.x = x;
    this.y = y;
    this.deltaY = deltaY;
    this.c = c;
  }
  boolean draw(Canvas canv) { ...  this.c  ... }
  boolean landed() { ... }
  boolean nearGround() { ... }
}
```

```
class Star {


  Star(int y) {
    super(20,y,5,new Yellow());
  }
  Star drop() {
    if (this.landed()) {
      return this; }
    else { if (this.nearGround()) {
      return
        new Star(100); }
    else {
      return
        new Star
          (this.y +  this.deltaY );}
    }
  }
}
```

```
class RedRock {
  int deltaX = 3;

  RedRock(int x,int y) {
    super(x,y,5,new Red());
  }
  RedRock drop() {
    if (this.landed()) {
      return this; }
    else { if (this.nearGround()) {
      return
        new RedRock(this.x,100); }
    else {
      return
        new RedRock(this.x + this.deltaX,
                    this.y + this.deltaY); }
    }
  }
  boolean hit(Girl g) {
    return
      g.closeTo(new Posn(this.x,this.y));
  }
}
```

Figure 79: Falling objects, revised

3. *c*, which represents the color of the falling object.

   This last field, *c*, is completely new, though its introduction is comparable to the renaming of *DELTA* to *deltaY*. Introducing *c* renders the *draw* methods identical and thus allows us to lift then method into the superclass. (See exercise 18.15.)

The signatures of the constructors of the two classes remain the same. Both constructors then call the **super** constructor, with their parameters in appropriate places and with appropriate constants elsewhere.

   Not surprisingly, the revised *RedRock* still contains the method *hit*. After all, it appears only in *RedRock* in the original program, and there is no need for this method in general. The similarities in *drop*, however, are so strong that you could have expected the method to be lifted into *Falling*. The content of the two methods differ in one aspect: while *Star*s move only along a vertical axis, *RedRock*s move in both directions. If *Falling* contained *deltaX* and *Star* set it to 0, it would still drop along a straight line. Here is the imaginary rewrite:

inside of *Star* :	inside of *RedRock* :
Star *drop*() {	*RedRock* *drop*() {
if (**this**.*landed*()) {	**if** (**this**.*landed*()) {
return this; }	**return this**; }
else { **if** (**this**.*nearGround*()) {	**else** { **if** (**this**.*nearGround*()) {
return	**return**
new *Star* (*this.x*,100); }	**new** *RedRock* (**this**.*x*,100); }
else {	**else** {
return	**return**
new *Star*	**new** *RedRock*
(**this**.*x* + **this**.*deltaX*,	(**this**.*x* + **this**.*deltaX*,
this.*y* + **this**.*deltaY*); }	**this**.*y* + **this**.*deltaY*); }
}	}
}	}

Unfortunately, lifting *deltaX* to *Falling* and generalizing *drop* accordingly still doesn't make the two methods look identical. The gray-shaded return types and constructor names emphasize how the computational rules are identical yet the method on the left uses *Star* and the right one uses *RedRock* in analogous positions.[37]

---

[37]The problem is that Java's type system—and that of almost all mainstream object-oriented programming languages—is overly restrictive. This kind of problem is the sub-

Exercises

Exercise 18.16 Use the test suite you developed in exercise 18.14 to test the program in figure 79. ∎

Exercise 18.17 Develop an interface for the newly created union of *Star*s and *RedRock*s. Does *drop* fit in? Why? Why not? ∎

Exercise 18.18 Modify the code in figure 79 so that the *drop* methods become identical except for the return types and the class names in **new** expressions. ∎

Once we can draw shapes onto a canvas, we can simulate the movement of objects, too:

```
class ExampleMove {
  Canvas c = new Canvas(100,100);
  Star f = new Star(10);

  ExampleMove() { }

  boolean testDraw = check this.c.show()
                           && this.f.draw(this.c)
                             && this.f.drop().draw(this.c)
                             && this.f.drop().drop().draw(this.c)
                         expect true;
}
```

This examples class creates two fields: a *Canvas* and a *Star*. Its one and only test field displays the canvas, draws the *Star*, drops and draws it again, and finally drops it twice and draws the result. At that point, the *Canvas* contains three yellow circles, drawn at (10,20), (10,25), and (10,30).

Imagine now the addition of a method *drawBackground*, which draws a white rectangle of 100 by 100 pixels at position (0,0) onto the canvas. Doing so every time before *draw* is called leaves just one yellow circle visible. If the method calls happen at a fast enough rate, the series of pictures that *ExampleMove* draws in this manner creates the illusion of a moving picture: a movie. A library that helps you achieve this kind of effect is a major example in the next section.

---

ject of programming languages researchers. One of their objectives is to design powerful yet safe type systems for languages so that programmers can express their thoughts in a concise, abstract and thus cost-effective manner.

```
// a point in the Cartesian plane
class CartPt {
  int x;
  int y;

  CartPt(int x, int y) {
    this.x = x;
    this.y = y;
  }

  // to compute the distance of this
  // point to the origin
  double distTo0(){
    return
      Math.sqrt(
        (this.x * this.x) +
        (this.y * this.y));
  }

  // compute the distance between this
  // CartPt and p
  double distanceTo(CartPt p){
    return
      Math.sqrt(
        (this.x − p.x) * (this.x − p.x) +
        (this.y − p.y) * (this.y − p.y));
  }

  // create a new point that is deltaX,
  // deltaY off-set from this point
  CartPt translate(int deltaX, int deltaY) {
    return
      new CartPt(this.x + deltaX,
                 this.y + deltaY);
  }
}
```

```
+-------------------------+
| Posn                    |
+-------------------------+
| int x                   |
| int y                   |
+-------------------------+
             |
            / \
            ---
             |
=============================
             |
             |
+---------------------------------+
| CartPt                          |
+---------------------------------+
+---------------------------------+
| double distanceTo()             |
| double distTo0()                |
| CartPt translate(int dX, int dY)|
+---------------------------------+
```
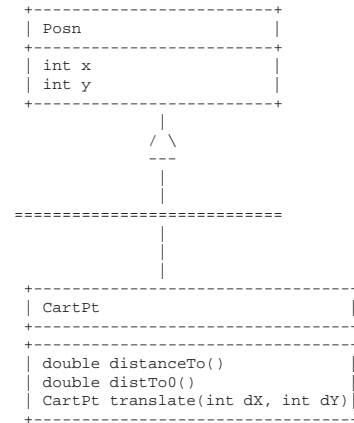
Figure 80: *CartPt* vs. *Posn*

18.5   Deriving Subclasses

In the preceding section we learned to abstract over similar classes with a superclass. To use these abstractions, we turned the original classes into

subclasses. It is often impossible, however, to reorganize a class hierarchy in this manner because it isn't under your control.

Figure 80's left column presents such a situation. The *CartPt* class (from section 12) represents the points in the Cartesian plane with two coordinates (*x* and *y*). In addition, *CartPt* provides three methods: *distanceTo*, *distTo0*, and *translate*. We developed those methods while we developed a representation for geometric shapes.

The *Posn* class in the **draw** package serves the same purpose as *CartPt*. It too represents Cartesian points, and the drawing methods in the package accept instances of *Posn*. These methods in the **draw.*** package do not, however, accept instances of *CartPt*. Thus, it appears impossible to draw shapes and to have some new computational methods on representations of Cartesian points.

In section 14.1 we solved the problem with the addition of one more method to *CartPt*: *toPosn*. This additional method converts instances of *CartPt* into *Posn*s and thus allows programs to draw shapes at this position.

The right column in figure 80 shows an alternative solution via a diagram: the derivation of a subclass of *Posn*. Deriving *CartPt* from the *Posn* library class has the immediate and obvious advantage that from a type checking perspective, every *CartPt* is a *Posn*. If a method specifies the type of a parameter as *Posn*, it also accepts an instance of a subclass. In short, this way of (re)organizing classes and fragments class hierarchies solves this problem of using program libraries and tailoring them to a special purpose. We resume and expand on this idea in the next section.

19 Designing Class Hierarchies with Methods

In the preceding section, we discussed examples of abstracting over classes via superclasses and using these abstractions via subclassing. This section formulates a design recipe that helps with the abstraction process in general and identifies conditions that describe when reusing existing abstractions is appropriate and how to go about it.

As you work through this chapter, keep in mind that our primary goal is a program that some other programmer can easily read and change. Getting to that point isn't as a one-step process, as you should recall the lesson from *How to Design Programs*; it is a process that requires many iterations. Every time you recognize a chance to abstract or to use an abstraction, edit the program and ensure that it still works with your existing test suite. If you haven't built one, do so before you edit; otherwise you may acci-

dentally introduce a mistake into a perfectly working program and fail to discover it for a long time.

The section starts with two reminders of simple but important ways to establish single points of control—a key goal of abstraction in general—within and across methods: local variables and abstraction of methods. The rest is new material.

```
// represent a Cartesian point                  class Examples {
class CartPt {                                     CartPt origin = new CartPt(0,0);
  int x;                                           CartPt other  = new CartPt(3,4);
  int y;
                                                   boolean test =
  CartPt(int x, int y) {                             check
    this.x = x;                                        this.origin.distanceTo(this.other)
    this.y = y;                                      expect 5.0
  }                                                  within .001;

  // the distance between this and other           Examples() {}
  double distanceTo(CartPt other) {             }
    return
      Math.sqrt(
        ((this.x − other.x) * (this.x − other.x)
        +
        (this.y − other.y) * (this.y − other.y)));
  }
}
```

Figure 81: Points and distance

19.1 Local Variables and Composition

Figure 81 displays a definition of the *CartPt* class. Unlike the definition in the preceding section, this one provides the method *distanceTo*, whose purpose is to compute the distance between **this** instance and some other distance. To understand the computation in *distanceTo*'s method body, you need to know a bit of domain knowledge:

given Cartesian points (x_1, y_1) and (x_2, y_2), their distance is

$$\sqrt{(x_1 − x_2)^2 + (y_1 − y_2)^2}$$

Equipped with this knowledge, reading the body of *distanceTo* should be straightforward.

Nevertheless, the method body of *distanceTo* is complex and doesn't reveal the underlying knowledge. Such deeply nested expressions are a strain on the reader. From *How to Design Programs*, you know that we use method composition and/or local variable definitions to simplify such methods. Let's try both ideas here, one at a time.

Let's start with a closer look at the domain knowledge. If you looked it up in a mathematics book, you may know that the distance between two points is equivalent to the length of the line between them. If you think of this latter line as something that starts at the origin, then the length of the line is equal to the distance of the other end point to the origin.

Given that the original definition of *CartPt* in figure 44 comes with *distance0*, we can define *distanceTo* via the composition of two methods:

inside of *CartPt* :
```
double distanceTo(CartPt other) {
  return this.line(other).distance0();
}
```

First, the *line* method computes the imaginary endpoint of the line between the two points. Second, the *distance0* method computes the distance of this endpoint to the origin, which is the desired length.

While *distance0* is already available in *CartPt*, we still need to design *line*:

inside of *CartPt* :
```
// the imaginary endpoint of the line between this and other
CartPt line(CartPt other) {
  ... this.x ... other.x ... this.y − other.y ...
}
```

From this template and the original formula (or a tiny bit of domain knowledge), we can define the actual method:

inside of *CartPt* :
```
// the imaginary endpoint of the line between this and other
CartPt line(CartPt other) {
  return new CartPt(this.x − other.x, this.y − other.y);
}
```

Composition often helps with such situations, though it also leads to the inclusion of lots of little methods in a class. In our example, we now have three methods. Two of them are clearly useful for everyone: *distance0* and

distanceTo; the third one just exists for bridging the gap between *distanceTo* and *distance0*.

An alternative to the addition of a method is the definition of LOCAL VARIABLES:

  inside of *CartPt* :
  double *distanceTo*(*CartPt other*) {
   int *deltaX* = **this**.*x* − *other.x*;
   int *deltaY* = **this**.*y* − *other.y*;
   CartPt p = **new** *CartPt*(*deltaX*,*deltaY*);
   return *p.distance0*();
  }

Here we see three of them: *deltaX*, *deltaY*, and *p*. At first glance, the definition of a local variable looks like a field definition with an initialization "equation." The difference is that a local variable definition is only visible in the method body, which is why it is called local.

In *distanceTo*, the local variables are initialized to the following values:

1. *deltaX* stands for the difference between the *x* coordinates;

2. *deltaY* stands for the difference between the *y* coordinates;

3. and *p* is a point created from the two differences.

As these definitions show, the expression on the right-hand side of initialization "equations" can use a number of values: those of the class's fields, the parameters, the fields of the parameters, and local variables whose definitions precedes the current one. Like fields, the name of the variable stands for this value, though only through the body of the method.

Based on this explanation of local variables, we can also perform a calculation to validate the equivalence of the intermediate version of *distanceTo* and the third one:

1. Since *deltaX* stands for the value of **this**.*x* − *other.x*, we can replace the occurrence in the third initialization "equation:"

    inside of *CartPt* :
    double *distanceTo*(*CartPt other*) {
     int *deltaY* = **this**.*y* − *other.y*;
     CartPt p = **new** *CartPt*(**this**.*x* − *other.x*,*deltaY*);
     return *p.distance0*();
    }

2. This works also for *deltaY*:

   inside of *CartPt* :
   double *distanceTo*(*CartPt other*) {
     CartPt p = **new** *CartPt*(**this**.*x* − *other.x*,**this**.*y* − *other.y*);
     return *p.distance0*();
   }

3. Finally, we can replace *p* in the rest of the method body with this **new** expression:

   inside of *CartPt* :
   double *distanceTo*(*CartPt other*) {
     return new *CartPt*(**this**.*x* − *other.x*,**this**.*y* − *other.y*).*distance0*();
   }

The rest is a short argument using the substitution model of computation that we know so well.

Exercises

Exercise 19.1 Can you argue that the second and first version of *distanceTo* are equivalent? ∎

Exercise 19.2 Transform the *CartPt* class provided in figure 81 into the second and then the third version, maintaining the *Examples* class as you go. ∎

19.2   Abstracting with Methods

According to *How to Design Programs*, abstracting over functions starts from concrete examples. When you recognize that two functions are similar, you highlight the differences, abstract over them with additional parameters, and re-define the originals from this general version (if applicable). Once you have those you can use the original tests to ensure that the abstraction process hasn't introduced any problems.

When you see two similar methods in the *same* class, we suggest you proceed in the same manner. Figure 82 displays a simple example. The class represents a push-button light switch. The *flip* method changes the on-off status of the switch. The *draw* method fills a canvas with an image appropriate to the switch's state. If it is on, the push-button is lit and the

```
class LightSwitch {
  boolean on;
  int width  = 100;
  int height = this.width;
  int radius = this.width/2;
  Canvas c = new Canvas(this.width,this.height);
  Posn origin = new Posn(0,0);
  Posn center = new Posn(this.radius,this.radius);
  IColor light = new Yellow();
  IColor dark  = new Blue();

  LightSwitch(boolean on) {
    this.on = on;
    this.c.show();
  }

  // turn this switch off, if on; and on, if off
  LightSwitch flip() {
    return new LightSwitch(!this. on);
  }

  // draw this light
  boolean draw() {
    if (on) {
       return this.paintOn(); }
    else {
       return this.paintOff(); }
  }

  boolean paintOff() {
   return this.c.drawRect(this.origin,this.width,this.height, this.light 1)
      && this.c.drawDisk(this.center, this.radius, this.dark 2);
  }

  boolean paintOn() {
   return this.c.drawRect(this.origin,this.width,this.height, this.dark 1)
      && this.c.drawDisk(this.center, this.radius, this.light 2);
  }
}
```

Figure 82: Similar methods in one class

surrounding background is dark; otherwise, the push-button is off and the surrounding background is lit. Since the two drawing actions consist of several expressions each, the developer has separated them into auxiliary methods, which look alike.

The four gray-shaded areas in figure 82 highlight the two pairwise differences between these auxiliary methods. The two boxes labeled with subscript 1 are the color arguments to the rectangle-drawing method that determine the background color; likewise, the two boxes labeled with subscript 2 are the color arguments to the disk-drawing method that determine the color of the button. Other than those two pairs of differences, the two methods are the same.

Following the design recipe from *How to Design Programs*, you abstract over such methods by replacing the analogous differences with two additional parameters—one per pair—in a new definition:

inside of *LightSwitch* :
// *auxiliar painting method for* **this** *light*
boolean *paint*(*IColor front*,*IColor back*) {
  return **this**.*c.drawRect*(**this**.*origin*,**this**.*width*,**this**.*height*,*back*)
    && **this**.*c.drawDisk*(**this**.*center*, **this**.*radius*,*front*);
}

In Java, we need to name the parameters and specify their types, which is *IColor* for both.

Next we must show that we can define the *paintOn* and the *paintOff* methods via *paint*:

inside of *LightSwitch* :
boolean *paintOn*() {
  return **this**.*paint*(**this**.*light*, **this**.*dark*);
}

boolean *paintOff*() {
  return **this**.*paint*(**this**.*dark*, **this**.*light*);
}

Given that each of these methods is used only once in *draw*, we can just change its definition by replacing the invocations of *paintOn* and *paintOff* with their bodies:

inside of *LightSwitch* :

boolean *draw*() {
  if (*on*) {
    return this.*paint*(**this**.*light*,**this**.*dark*); }
  else {
    return this.*paint*(**this**.*dark*,**this**.*light*); }
}

Now all that remains is to run some examples and inspect the appearances of the canvases before and after abstracting.

Exercise

Exercise 19.3 Develop an example class for figure 82 and then transform the class to use the *paint* abstraction. Why is this not a proper test? ▪
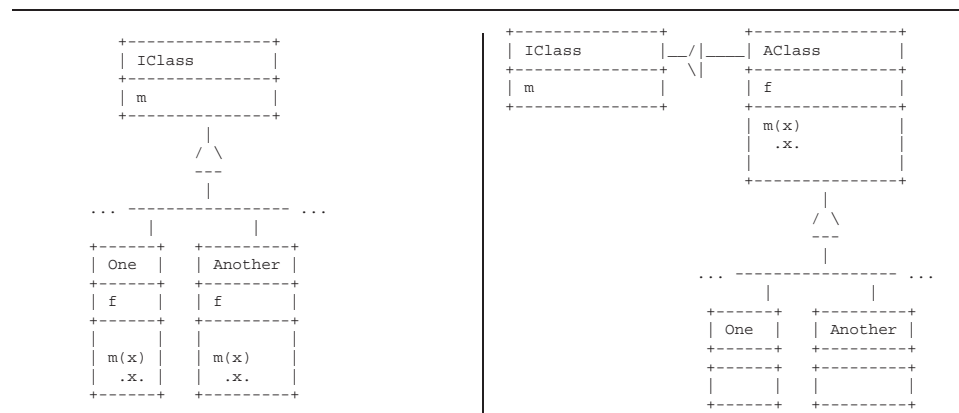
Figure 83: Lifting methods

19.3   <mark>Abstracting within Unions of Classes</mark>

Our simplest example of abstraction concerns the lifting of similar methods within unions. Figure 83 sketches the essence of this form of abstraction. When you recognize that the variants contain identical field or method definitions, you create a common abstract superclass and move the common definitions from the subclasses to the superclass. Here are the exact steps:

The comparison After you have finished designing a union hierarchy, look for identical field definitions or method definitions in the variants (subclasses). The fields should occur in *all* variants; the methods should be in at least *two* variants.

The abstraction Create the abstract class *AClass* and have it implement the union interface with abstract methods. Replace the **implements** specification in the subclasses with **extends** specifications, pointing to the superclass.

Eliminate the common field definitions from all variants; add a single copy to the superclass. Modify the constructors in the subclasses to use **super** for the initialization of shared fields.

Introduce a copy of the shared method *m* in the superclass and eliminate the methods from all those subclasses where the exact same code appears.

The super call For all those variants that contain a method definition for *m* that differs from the one in the superclass, consider reformulating the method using **super**. We have seen an example of this in section 18.3, and it is indeed a reasonably common scenario. If it is possible, express the method in this manner because it helps your successor understand the precise relationship between the two computations.

As long as you do not have sufficient experience with abstraction in an object-oriented setting, you may wish to skip this step at first and focus on the basic abstraction process instead.

The test Re-run the test suite. Since subclasses inherit the methods of the abstract class that represents the union, the subclasses did not really change, especially if you skipped the third step. Still, every modification may inadvertently introduce a mistake and therefore demands testing. Since you have a test suite from the original development, you might as well use it.

This first design recipe for abstraction is extremely simple in comparison to the one we know from *How to Design Programs*. The reason is that it assumes that the methods in the two variants are identical. This assumption is is of course overly simplistic. In general, methods are similar, not identical. If this is the case, you can often factor out the common pattern between those methods and then lift the common method.

Figure 84 is a sketch of the first step in this modified abstraction process. Its left column indicates that two variants of a union contain similar method

```
+---------------+    +---------------+          +---------------+    +---------------+
| IClass    |__/|__| AClass    |          | IClass    |__/|__| AClass    |
+---------------+   \|  +---------------+          +---------------+   \|  +---------------+
| m         |          +---------------+          | m         |          +---------------+
+---------------+          | abstract m    |          +---------------+          | abstract m    |
                           |               |                                     |               |
                           +---------------+                                     +---------------+
                                   |                                                     |
                                  / \                                                   / \
                                  ---                                                   ---
                                   |                                                     |
                         ... ---------- ...                                    ... ---------- ...
                             |          |                                          |          |
                        +------+    +---------+                              +------+    +---------+
                        | One  |    | Another |                              | One  |    | Another |
                        +------+    +---------+                              +------+    +---------+
                        +------+    +---------+                              +------+    +---------+
                        |      |    |         |                              |      |    |         |
                        |m(x)  |    |m(x)     |                              |n(x,y)|    |n(x,y)   |
                        | .x.o.|    |  .x.p.  |                              | .x.y.|    |  .x.y.   |
                        |      |    |         |                              |m(x)  |    |m(x)     |
                        +------+    +---------+                              | n(x,o)|   |  n(x,p) |
                                                                            |      |    |         |
                                                                            +------+    +---------+
```
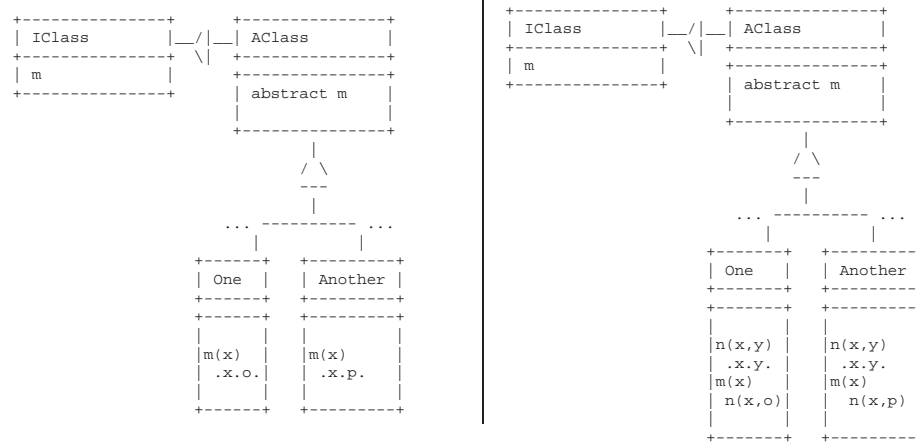
Figure 84: Lifting nearly identical methods

definitions. More specifically, the definitions differ in one place, where they contain different values: *o* and *p*, respectively. The right column shows what happens when we apply the design recipe for abstraction from *How to Design Programs*. That is,

The comparison Highlight the differences between *m* in class *One* and *m* in class *Another*.

The abstraction Define method *n* just like method *m* (in both classes) with two differences: it specifies one additional parameter (*y*) and it uses this parameter in place of the original values (*p* and *o*).

The test, 1 Reformulate the body of *m* in terms of *n*; more precisely, *m* now calls *n* with its arguments and one extra argument: *o* in *One* and *p* in *Another*.

The test, 2 Re-run the test suite to confirm that the *m* methods still function as before. You may even want to strengthen the test suite so that it covers the newly created method *n*.

At this point you have obtained two identical methods in the two classes. You can now follow the design recipe for abstraction to lift *n* into the common superclass.

A similar design recipe applies when you discover that the variants contain fields with distinct names but the same purpose. In that case, you first

rename the fields consistently throughout one class, using the correspond-
ing field name from another variant. You may have to repeat this process
until all names for fields with the same purpose are the same. Then you lift
them into the common superclasses. Of course you can only do so if you
have complete control over the classes, i.e., if you are in the middle of the
design process, or if you are guaranteed that other pieces of the program
use the interface of the union as the type and do not use individual variant
classes. Otherwise people may already have exploited the fact that your
classes contain fields with specific names. In this case, you are stuck and
can't abstract anymore without major problems.

```
                                            +---------------+     +---------+
                                            | IClass        |__/|__| SuperCC |
                                            +---------------+  \|  +---------+
                                            | m             |      | f . g   |
                                            +---------------+      +---------+
                                                                   |         |
                                                                   | m ...   |
                                                                   |         |
                                                                   +---------+
                                                                        |
                                                                      / \
                                                                      ---
                                                                       |
    +---------+   +---------+                                      +----------+
    | ClassAA |   | ClassBB |                                      |          |
    +---------+   +---------+                                      |          |
    | f . g   |   | f . g   |                              +---------+   +---------+
    +---------+   +---------+                              | ClassAA |   | ClassBB |
    |         |   |         |                              +---------+   +---------+
    | m ...   |   | m ...   |                              +---------+   +---------+
    |         |   |         |                              |         |   |         |
    +---------+   +---------+                              |         |   |         |
                                                          +---------+   +---------+
```
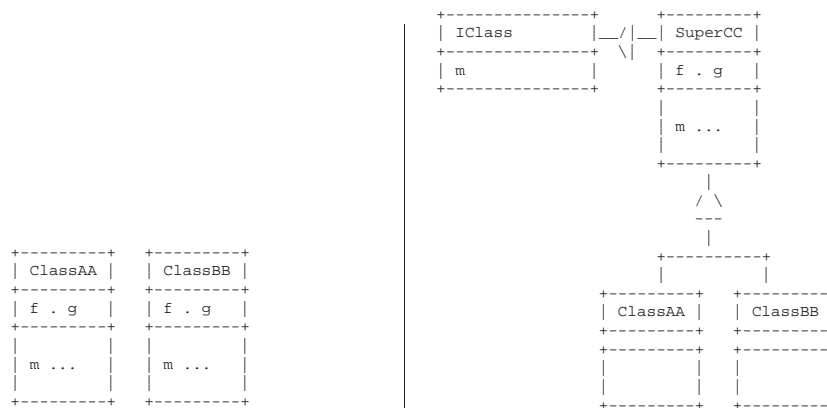
Figure 85: Creating a union retroactively

19.4 Abstracting through the Creation of Unions

On occasion, you will develop or extend a program and then find that two
independent classes are structurally similar. The classes will have a sig-
nificant overlap in fields, methods, and purpose. In other words, you will
notice *after the fact* that you need a union.

Take a look at the left of figure 85, which represents the situation graph-
ically. The right shows your goal: the creation of a common superclass. The
process of going from the left to the right column consists of three steps:

The comparison When you discover two similar classes without super-
classes,[38] such as *ClassAA* and *ClassBB*, inspect the commonalities
closely.

---

[38]In some object-oriented languages, a class may have multiple superclasses and then

Before you proceed, also re-consider the purpose statement of each class and determine whether the two classes are related in spirit. In other words, ask yourself ==whether the two classes represent related kinds of objects==. If they don't, stop.

Note 1: The two classes may have subclasses, that is, they themselves may represent a union.

Note 2: If you control *all* uses of a class, you may systematically rename fields and methods throughout the program to make two classes as similar as possible. Do so to enable abstraction as much as possible.

If you are dealing with the fragment of a program instead and others use your classes, you probably don't have the freedom to rename your fields or methods.[39]

The abstraction If you can confirm an overlap in purpose, methods, and fields, you create a union, that is, you introduce an interface (*IClass*) and a common superclass (*SuperCC*).

The interface contains the common method specifications of the two classes.. As long as two methods in *ClassAA* and *ClassBB* have the same purpose and compatible signatures, they belong into the interface of the union, which represents the union to the rest of the program.

The superclass contains the common field and method definitions from the two original classes. Formulate a general purpose statement that clarifies how this new class represents objects from both classes. Finally, turn the given classes into derived classes. In particular, you should now eliminate the fields and methods that *SuperCC* introduces and reformulate the constructors using **super**.

Note: Introduce an interface only if you believe that the class hierarchy represents a union and if you anticipate that you or others will deal with all instances of these classes from one (union) perspective.

The test Re-run the test suite for *ClassAA* and *ClassBB*. If you believe that *SuperCC* is useful in its own right, create a test suite for *SuperCC* that generalizes the examples from *ClassAA* and *ClassBB*.

---

you can abstract in all cases. Since this isn't common and Java doesn't support it, we ignore this possibility.

[39]None of the existing object-oriented programming languages truly separate the internals of a class from the visible interface; otherwise this would not be a problem.

Sometimes the creation of a common superclass for two classes may eliminate the need for the two original classes. This is particularly true if, after lifting, the two classes contain nothing but constructors. The only reason for maintaining the two is to ensure that the language itself can check the distinction. Figuring out whether this is needed takes common sense and a lot experience.

```
// a set of integers:
// contains an integer at most once
class Set {
  ILin elements;

  Set(ILin elements) {
    this.elements = elements;
  }

  // add i to this set
  // unless it is already in there
  Set add(int i) {
    if (this.in(i)) {
      return this; }
    else {
      return
        new Set(new Cin(i,this.elements));
      }
  }

  // is i a member of this set?
  boolean in(int i) {
    return
      this.elements.howMany(i) > 0;
  }
}
```

```
// a bag of integers
class Bag {
  ILin elements;

  Bag(ILin elements) {
    this.elements = elements;
  }

  // add i to this bag
  Bag add(int i) {
    return
      new Bag(new Cin(i,this.elements));
  }

  // is i a member of this bag?
  boolean in(int i) {
    return
      this.elements.howMany(i) > 0;
  }

  // how often is i in this bag?
  int howMany(int i) {
    return
      this.elements.howMany(i);
  }
}
```

Figure 86: Collections of integers

Exercises

Exercise 19.4 Compare the two classes in figure 86. A *Set* is a collection of integers that contains each element at most once. A *Bag* is also a collecion of integers, but an integer may show up many times in a bag.

Your tasks are:

1. The two class definitions use lists of integers to keep track of elements. Design a representation of lists of integers with this interface:

    ```
    // a list of integers
    interface ILin {
      int howMany(int i);
    }
    ```

    The constructors are *MTin* for the empty list and *Cin* for constructing a list from an integer and an existing list.

2. Develop examples of *Set*s and *Bag*s.

3. Develop functional examples for all methods in *Set* and *Bag*. Turn them into tests.

4. The two classes clearly share a number of similarities. Create a union and lift the commonalities into an abstract superclass. Name the union interface *ICollection*. Don't forget to re-run your test suite at each step.

5. Develop the method *size*, which determines how many elements a *Bag* or a *Set* contain. If a *Bag* contains an integer *n* times, it contributes *n* to *size*.

6. Develop the method *rem*, which removes a given integer. If a *Bag* contains an integer more than once, only one of them is removed. ∎

Exercise 19.5 Banks and international companies worry about keeping the various kinds of moneys separate. If you work at such a company, your manager might request that you implement separate classes for representing US Dollars and Euros, thus making sure that the programming language enforces this separation. That is, assuming all amounts of money are represented via objects, your program will then never accidentally add euros to dollars (or vice versa).

Implement the classes *Dollar* and *Euro*. To keep things simple, let's assume that all amounts are integer amounts and that the only two operations that matter add currencies (of the same kind) and translate them into *String*.

   The two classes obviously share a lot of code. Abstract a common super-class, say *AMoney*, from the two classes. Lift as many methods and fields as possible.

   Add a class *Pound* for representing amounts in British Pounds. ∎

19.5   Deriving Subclasses from "Library" Classes

As we know from *How to Design Programs*, using existing abstractions is much more an art than a design discipline. This is equally true for the reuse of functions as it is for the reuse of classes. In both cases, it is critical to discover that an existing abstraction can be used for a given problem situation. Here we provide a process like a design recipe that helps to stay on track with this difficult part of programming but it is not a recipe *per se*:

```
Library:                                Library:

+------------------------+              +------------------------+
| LibCC                  |              | LibCC                  |
+------------------------+              +------------------------+
| /////                  |              | /////                  |
|   ...                  |              |   ...                  |
+------------------------+              +------------------------+
| +++++                  |              | +++++                  |
+------------------------+              +------------------------+
                                                   / \
                                                   ---
                                                    |
                                                    |
                                                    |
=============================           =============================
                                                    |
                                                    |
                                                    |
+------------------------+              +------------------------+
| AA                     |              | AA                     |
+------------------------+              +------------------------+
| /////                  |              |   ...                  |
|   ...                  |              +------------------------+
+------------------------+              |   ...                  |
| +++++                  |              +------------------------+
|   ...                  |
+------------------------+
```

Figure 87: Using a "library" class

1. When your task is to design a class, follow the recipe for designing a class and its methods—up to a certain point. At a minimum, write down the purpose statement for the class, the fields for the class, and some sample instances. Then develop the signatures and purpose statements of the methods; ideally, you should also develop examples for the methods.

Going through the design recipe in this way helps you understand what this new class of data is all about. Now you are in a position where you can determine whether this class should be an extension of an existing class. Here are two situations that suggest such an extension:

(a) You already have a class that has a related but more general purpose statement. Also, the existing class provides several of the fields and methods that you need for the new class, if not by name then in spirit.

The derivation of *Precipitation* from *ARecording* exemplifies this situation. It represents the extension of an existing union of classes with an additional variant. This step is common, because we often don't think of all the variants when we first design a union.

Note: When we extend a class that is not a union, we often really want a union not just one variant of an existing class.

(b) The collections of software that are available to you on your computer or at your workplace contain a potential superclass *LibCC* whose purpose fits the bill. Furthermore, the collection contains other classes that rely on *LibCC*. If you derive your new class from *LibCC*, you benefit not just from the fields and methods in the class but also from methods in other classes in this collection.

Deriving *CartPt* from *Posn* in the drawing package is an instance of this situation. As mentioned, every instance of *CartPt* can now play the role of a *Posn* when it comes to drawing an object that needs more methods on positions than *Posn* offers.

Note: In many cases, such libraries are designed to be extended. Some times they provide **abstract** classes; the **abstract** methods in these classes specify what you need to define yourself. At other times, the classes are also fully functional, with methods that specify some reasonable default behavior. In this case, the libraries tend to come with specific prescriptions of what the programmer must override to exploit the functionality.

In both cases, the superclass is outside of your reach, that is, you can't modify it. Because this is most common with so-called software libraries we dub this situation the "library" class situation.

The left-hand side of figure 87 characterizes the situation pictorially with a diagram. The right side is an idealization of the next step.

2. The rest of the process is relatively straightforward. You identify three critical points:

   (a) the fields that the superclass (*LibCC*) provides and those that the new class (*AA*) must add;

   (b) the methods that the superclass provides with the proper functionality;

   (c) the methods that the superclass provides without or with the wrong functionality; you need to override those methods if you want your new class to play its role properly;

   (d) the methods that you must supply from scratch.

3. For the design of the new and the overridden methods, follow the design recipe. Develop examples, templates, and then define the methods. Finally, run the tests.

   Once you fully understand the overridden methods, you may wish to figure out how to reformulate the method in terms of **super** calls. This will help other programmers understand in what sense the subclass is like the superclass and how it differs.

The next two subsections present another two examples of class derivation, one example per situation.

19.6 Creating Subclasses for Special Objects

The preceding section suggests to derive classes when an existing class doesn't quite fit the purpose. This suggestion implies that we create subclasses of a class if some of the objects have a special status. Here is a concrete example:

> ... Tetris was a popular computer game. Your company's manager is thinking of a revival edition and suggests to start with a simple "block world" program. In the "block world" scenario, a block drops down from the top of the computer canvas—at a steady rate—until it lands on the ground or on blocks that have landed before. A player can steer the dropping block with the "left" and "right" arrow keys. When a stack of blocks reaches

the ceiling of the world, the game is over. The objective of this
game is to land as many blocks as possible.
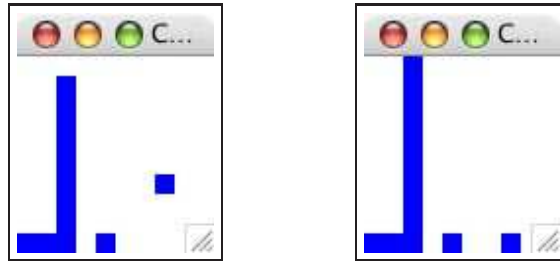
The two screen shots illustrate the idea. In the left one, you
can see how one block, the right-most one, has no support; it
is falling. In the right one, the block has landed on the ground;
an additional block has landed on top of the stack near the left
perimeter of the screen shot. This stack now reaches the top of
the canvas, which means that the game is over. ...

   A Tetris program definitely needs to represent blocks. Hence your first
task is to design a class for this purpose. These blocks obviously need to
include information about their location. Furthermore, the game program
must be able to draw the blocks, so the class should come with at least one
method: *draw*, which draws a block on a canvas at the appropriate location.

```
                                              +--------------------------+
                                              | Block                    |
                                              +--------------------------+
                                              | int down                 |
       +--------------------------+           | int right                |
       | Block                    |           +--------------------------+
       +--------------------------+           | boolean draw()           |
       | int down                 |           +--------------------------+
       | int right                |                         |
       +--------------------------+                        / \
       | boolean draw(Canvas w)   |                        ---
       +--------------------------+                         |
                                                            |
                                              +--------------------------+
                                              | DrpBlock                 |
                                              +--------------------------+
                                              | int deltaY               |
                                              +--------------------------+
                                              | DrpBlock drop()          |
                                              | boolean landed(IBlocks r)|
                                              | DrpBlock steer(String ke,|
                                              |              IBlocks r)  |
                                              +--------------------------+
```
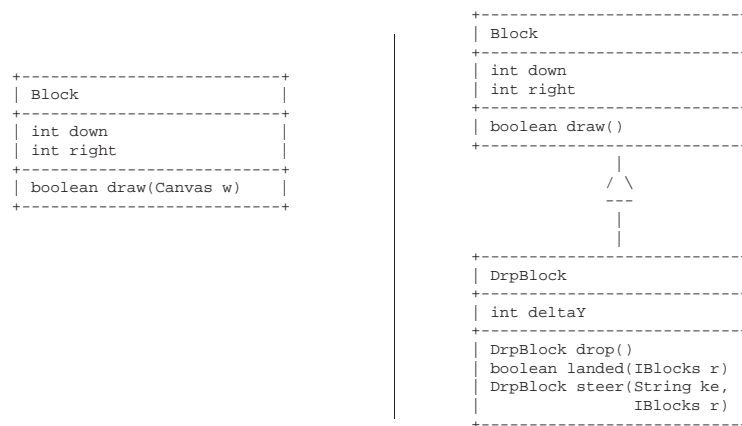
Figure 88: A class diagram for a class extension

   The left column of figure 88 contains a class diagram that summarizes
the result of this data analysis. The location is expressed as a pair of int

fields, which record how far down (from the top) and how far to the right (from the left border) the block is. The *draw* method consumes the canvas into which it is supposed to draw; all other information comes from the block itself.

Exercises

Exercise 19.6 Design the *Block* class, including a *draw* method. Include an example class with a test field that demonstrates that the *draw* method functions. ∎

Exercise 19.7 Instances of *Block* represent blocks resting on the ground or on each other. A game program doesn't deal with just one of those blocks but entire collections. Design a data representation for lists of blocks. Include a method that can draw an entire list. ∎

Of course, the *Block* class isn't enough to play the game. First, your game program needs lists of such blocks, as exercise 19.7 points out. Second, the problem statement itself identifies a second class of blocks: those that are in the process of dropping down to the ground. Since we already have *Block* for representing the class of plain blocks and since a dropping block is like a block with additional properties, we next consider deriving an extension of *Block*.

The derived class has several distinguishing properties. The key distinction is that its instances can drop. Hence, the class comes with a method that simulates the drop and possibly an additional field that specifies how fast an instance can drop as a constant:

1. *drop*, a method that creates a block that has dropped by some pixels;

2. *deltaY*, a field that determines the rate at which the block drops.

Furthermore, a dropping block can land on the ground or on the blocks that are resting on the ground and the player can steer the block to the left or right. This suggests two more methods:

3. *landed*, which consumes a list that represents the resting blocks and determines whether **this** block has landed on the ground or one of the other blocks;

4. *steer*, which also consumes the list of resting blocks and a *String*, and moves the block left or right. Whether or not the block can move in the desired direction depends of course on the surrounding blocks.

The right column of figure 88 displays the revised class diagram. It consists of two concrete classes, with *Block* at the top and *DrpBlock*, the extension, at the bottom. The diagram assumes the existence of *IBlocks*, a representation of a list of blocks.

Exercises

Exercise 19.8 Define the *DrpBlock* class and design all four methods on the wish list.

   Hint: You may wish to employ iterative refinement. For the first draft, assume that the list of resting blocks is always empty. This simplifies the definitions of *landed* and *steer*. For the second draft, assume that the list of resting blocks isn't empty. This means that you need to revise the *landed* and *steer* methods. As you do so, don't forget to maintain your wish list. ∎

Exercise 19.9 Add test cases to your example class from exercise 19.8 that exercise the *draw* methods of *Block*, *DrpBlock*, and *IBlocks*. ∎

At first glance, the derivation of *DrpBlock* from *Block* is a natural reflection of reality. After all, the set of dropping blocks is a special subset of the set of all blocks, and the notion of a subclass seems to correspond to the notion of a subset. A moment's thought, however, suggests another feasible program organization. If we think of the set of all blocks as one category and the set of all dropping blocks as a special kind of block, the question arises what to call all those blocks that are not dropping. As far as the game is concerned, the remaining blocks are those that are already resting on the ground and each other. Put differently, within the collection of blocks there are *two* kinds of blocks: the resting ones and the dropping ones; and there is no overlap between those two sets of blocks.

Figure 89 translates this additional analysis into a modified class diagram. The class hierarchy has become a conventional union arrangement. In particular, *Block* has become an abstract class with two concrete subclasses: *Resting* and *DrpBlock*. The diagram also shows that *DrpBlock* comes with one additional method: *onLanding*, which produces an instance of *Resting* from a *DrpBlock* as the dropping block lands on the ground or on one of the resting blocks.

```
+--------------------------+
| ABlock                   |
+--------------------------+
| int down                 |
| int right                |
+--------------------------+
| boolean draw(Canvas c)   |
+--------------------------+
           |
         / \
         ---
          |
     ---------------
     |             |
+---------+   +-------------------------+
| Resting |   | DrpBlock                |
+---------+   +-------------------------+
+---------+   | Resting convert()       |
              | DrpBlock drop()         |
              | boolean landed(IBlocks r)|
              | DrpBlock                |
              |  steer(String ke,       |
              |        IBlocks r)       |
              | Resting onLanding()     |
              +-------------------------+
```
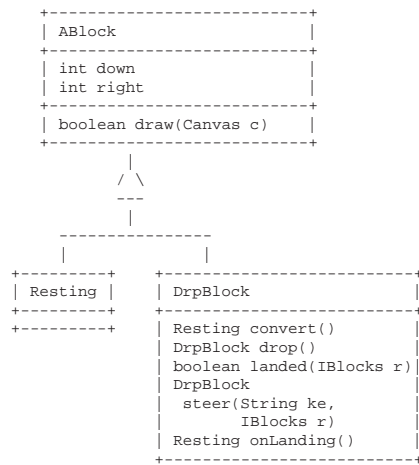
Figure 89: Two distinct classes of blocks

An interesting aspect of the revised diagram is that *Resting* comes without any additional fields or methods. All the features it needs are inherited from *ABlock*, where they are defined because they are also needed in *DrpBlock*. Still, representing blocks as a union of two disjoint classes has an advantage. Now *IBlocks* can represent a list of resting blocks, that is, it is a list of instances of *Resting* rather than *Block*. Using *Resting* instead of *Block* signals to the (future) reader that these blocks cannot move anymore. No other part of the program can accidentally move these resting blocks anymore or perform some operation on them that applies to dropping blocks only. Furthermore, as an instance of *DrpBlock* lands, it must become a member of the list of *Resting* blocks. This can only happen, however, if it is first converted into an instance of *Resting*. Thus, it is totally clear to any reader that this block has ceased to move.

In summary, the development of a subclass is often justified when we can identify a special subset of the information that we wish to represent. Pushing this analysis further, however, tends to reveal advantages of the creation of a full-fledged union rather than a single subclass. Always consider this option when you believe you must derive a subclass from an existing class in your program.

Exercises

Exercise 19.10 Develop an interface for *ABlock* for the methods that all vari-

ants in figure 89 have in common. ▪

Exercise 19.11 Modify your result of exercises 19.8 and 19.9 so that they implement the class diagram in figure 89.

In the process, implement the suggestion of changing *IBlocks* so that it represents a list of instances of *Resting*. ▪

19.7   How Libraries Work, Part 2: Deriving Classes

While you have significant freedom with the derivation of a class from your own classes, you usually have no choice when it comes to classes in libraries. The designers of object-oriented libraries often want you to derive classes and override methods to get the desired effect.

```
// to represent a world with its visual part drawn on a canvas
abstract class World {
  Canvas theCanvas = new Canvas();

  // open a width by height canvas,
  // start the clock, and make this world the current one
  boolean bigBang(int width, int height, double s) { … }

  // process a tick of the clock in this world
  abstract World onTick() { … }

  // process a keystroke event in this world
  abstract World onKeyEvent(String ke) { … }

  // draw this world
  abstract boolean draw() { … }

  // stop this world's clock
  World endOfWorld(String s) { … }
}
```

Figure 90: Animated *World*s

The **draw** package, which was introduced in section 14.1, is such a library. Recall that it provides *Canvas*, *Posn*, *IColor*, and several concrete implementations of the last one. The most complex class in this library is *Canvas* (see figure 50 (page 136)). It provides a number of methods for drawing shapes on a (computer) canvas as well as erasing them from there.

A drawing library such as **draw** has the proper ingredients for creating the illusion of an item dropping from the top to the bottom of a canvas. Recall from the end of the previous section that on a computer canvas, a dropping block is a block that is drawn over and over again at different positions on a brand new background. Consider these two images:

In the right image, the block appears at a lower place than in the left one. If the transition from left to right is fast, it appears as if the block is moving. Hence, drawing the background, drawing a block on this background, and repeating those actions in a predictable rhythm is our goal.

Drawing libraries such as **draw** abstract over this pattern of repetition with a mechanism that allows something to happen on every tick of a clock. To this end, our **draw** package provides an additional **abstract** class: *World*. Figure 90 contains the outline of *World*'s definition. It provides one field (*theCanvas*), two methods (*bigBang*, *endOfWorld*) and three abstract methods. The field refers to the canvas on which the world appears. The first concrete method, *bigBang* creates the world, makes the canvas appear, and starts the clock, and makes it click every *s* seconds. The second one, *endOfWorld* stops the clock and thus all animations. The three **abstract** methods in *World* are *onTick*, *onKeyEvent*, and *draw*. The first two are called EVENT HANDLERS because they react to events in the "world." The *onTick* method is invoked for every tick of the clock; it returns a new world. Similarly, *onKeyEvent* is invoked on the world and a keystroke, every time a person hits a key on the keyboard; the *String* parameter specifies which key has been hit. Like *onTick*, *onKeyEvent* returns a new world when it is done. The *draw* method draw this world.

The purpose of *World* is to represent animated mini-worlds. To use it, you define a subclass of *World*. Furthermore, since *World* itself is **abstract**, you must override the four abstract methods if you want to create worlds. Then, as soon as a call to *bigBang* starts the clock, the *World*'s canvas becomes alive. At each clock tick, the code in the library calls your *onTick* to create the next world and then uses *draw* for re-drawing this new world. The process continues until the clock is stopped, possibly with *endOfWorld*.

In short, your subclass and *World* collaborate according to the template-and-hook pattern.

The following table illustrates the workings of *onTick* graphically:

clock tick	0	1	2	...	n	$n+1$
this current world	a	b	c	...	w	x
result of **this**.*onTick*()	b	c	d	...	x	...

The first row denotes the number of clock ticks that have happened since *a.bigBang*() started the clock. The invocation of *a.onTick*() produces *b*, which becomes the current world. This means, in particular, that the next clock tick triggers a call to *onTick* in *b*. In general, the result of the *n*th call to *onTick* becomes the $n + 1$st world.

So, if your goal is to create an animated block world where a single block drops to the bottom of the canvas, you must define a class, say *Block-World*, that **extends** *World* and overrides: *onTick*, *onKeyEvent*, and *draw*. The simplest such subclass is sketched in figure 91:

1. the field *block* of type *DrpBlock* (see previous section) refers to the one block that is dropping in this world;

2. *onTick* drops the block and creates a new world from the resulting block;

3. *draw* draws the block onto **this** world's canvas via the block's *draw* method and its own *drawBackground* method; and

4. *onKeyEvent* returns just **this** world.

The last point is surprising at first, but remember that without a definition for *onKeyEvent BlockWorld* would be an **abstract** class. The definition in figure 91 means that hitting a key has no effect on the current world; it is returned "as is."

Now that you have an extension of *World* with working *onTick*, and *draw* methods you can run your first animation:

DrpBlock aBlock = **new** *DrpBlock*(10,20);
World aWorld = **new** *BlockWorld*(aBlock);
aWorld.bigBang(*aWorld.WIDTH*,*aWorld.HEIGHT*,.1)

The first line creates the representation of a block, located 10 pixels to the right from the origin and 20 pixels down. The second one constructs a *BlockWorld* with this *DrpBlock*. The last one finally starts the clock and thus

```
// the world of a single dropping block
class BlockWorld extends World {
  IColor BACKGROUND = …;

  DrpBlock block;
  BlockWorld(DrpBlock block) {
    this.block = block;
  }

  // drop the block in this world
  World onTick() {
    return new BlockWorld(this.block.drop());
  }

  // drop the block in this world
  World onKeyEvent(String ke) {
    return this;
  }

  // draw this world's block into the canvas
  boolean draw() {
    return this.drawBackground()
        && this.block.draw(this.theCanvas);
  }

  // paint the entire canvas BACKGROUND
  boolean drawBackground() {
    return this.theCanvas.drawRect(new Posn(0,0),…,…,…);
  }
}
```

Figure 91: An animated world with one dropping block

the mechanism that calls *onTick* and *draw* every .1 seconds. If you want the block to move faster, shorten the time between clock ticks. Why?

This first success should encourage you for the next experiment. Moving an object each tick of the clock is only one possibility. In addition, we can also have the object react to keystrokes. Remember that our dropping blocks from the preceding section should also be able to move left and right. Hitting the right arrow key should move the block to the right, and hitting the left arrow key should move it to the left. As the documentation for *World* says, each such keystroke triggers an invocation of the *onKeyEvent*

```
+---------------------+     +--------------------------+     +---------------------+
| Canvas              |<-+  | World                    |  #  | Block               |
+---------------------+  |  +--------------------------+  #  +---------------------+
| boolean drawRect(Posn) +--| Canvas theCanvas         |  #  | int down            |
+---------------------+     +--------------------------+  #  | int right           |
                            | World onTick()           |  #  +---------------------+
                            | World onKeyEvent(String k)|  #  | boolean draw(Canvas c)|
                            | boolean draw()           |  #  +---------------------+
        +----------+        +--------------------------+  #           |
        | Posn     |                  |                   #           |
        +----------+                  |                   #           |
        | int x    |                  |                   #           |
        | int y    |                  |                   #           |
        +----------+               / \                    #         / \
                                   ---                     #         ---
        +--------+                  |                      #          |
        | IColor |                  |                      #          |   +---------------------+
        +--------+                  |                      #          |   | IBlocks             |
            |                       |                      #          |   +---------------------+
          / \                       |                      #          |   | boolean draw(Canvas c)|
          ---                       |                      #          |   | boolean atop(Block b) |
           |                        |                      #          |   | boolean left(Block b) |
  ...  ---------------------- ...   |                      #          |   | boolean right(Block b)|
      |     |          |            |                      #          |   +---------------------+
     ...   ...        ...           |                      #          |
                                    |                      #          |
==========================================================#          |
                                    |                                 |
              +--------------------------+                            |
              | BlockWorld               |                            |
              +--------------------------+          +---------------------+
              | DrpBlock block           |----+---->| DrpBlock            |
              | IColor BACKG             |          +---------------------+
              +--------------------------+          | int deltaY          |
              | World onTick()           |          +---------------------+
              | World onKeyEvent(String k)|          | DrpBlock drop()     |
              | boolean draw()           |          | DrpBlock steer(     |
              | boolean drawBackground() |          |   String ke)        |
              +--------------------------+          +---------------------+
```

Figure 92: Animating dropping blocks

method in the current world. Since this current world is an instance of
BlockWorld, it uses the method from that class, which currently just returns
the current world. If it instead passes the string on to *block*'s *steer* method,
players get to control its drop:

inside of *BlockWorld* :
// what happens to **this** world on every tick of the clock
World onKeyEvent(*String ke*) {
    return new *BlockWorld*(**this**.*block*.*steer*(*ke*));
}

Like *onTick*, *onKeyEvent* first creates a new block, using its *steer* method on
ke (see figure 89). Next it constructs a new world from this block and re-
turns it as its response to the key event. Of course, this design is just the
result of following the design recipe for containment and nothing unex-
pected. Make up examples for *onKeyEvent* and test it before you run the
animation again. When you do so, use the arrow keys to control the block

as it drops.

From here it is a short step to a program that drops many blocks, lands them on the ground and each other, and allows the player to move the currently dropping block left or right. Figure 92 sketches the class diagram for this program, including both the library and the "block world" classes. Both parts consist of many interfaces and classes. On "our" side, you see *Block* and *DrpBlock* as designed in the previous section; *BlockWorld* specializes *World* to our current needs across the library boundary. For simplicity, the diagram omits the color classes in **draw** and the context of blocks on which an instance of *DrpBlock* may land. Refer to this diagram as you solve the following exercises and finish this simple game.

Exercises

Exercise 19.12 Finish the class definitions for the block world program. Start with *drawBackground*; it is a method that colors the entire background as needed. Change the colors of the world and the block. ∎

Exercise 19.13 Modify *BlockWorld*—and all classes as needed—so that the animation stops when the bottom of the dropping block touches the bottom of the canvas. **Hint**: Check out *endOfWorld* from **draw**. ∎

Exercise 19.14 Add a list of resting blocks to *BlockWorld* based on your implementation of *IBlocks* in exercise 19.11. The program should then draw these blocks and land the dropping block either on the bottom of the canvas or on one of the resting blocks. When the player steers, make sure that the dropping block cannot run into the resting block. ∎

Exercise 19.15 Design the class *Blink*, which represents blinking objects. A blinking object is a 100 by 100 square that displays two different colors at a rate of one switch per second. The *Blink* method should contain a *run* method that starts the clock and the blinking process.

   Hint: Naturally *Blink* is an extension of *World*.

   Your example class should test the *onTick* method before it invokes the *run* method expecting true. It should also create at least two distinct instances of *Blink* and display them at the same time. ∎

19.8 Case Study: Fighting UFOs, All the Way

In sections 6.2 and 16.3 we developed the core of a simple interactive game with UFOs and anti-UFO platforms. Thus far we have developed classes

that represent the various pieces of the game and methods that can draw
them or move them one step at a time. What is missing from the game
is real action: a UFO that is landing; shots that are flying; an AUP that is
moving. Since we now know how this works, it's time to do it.
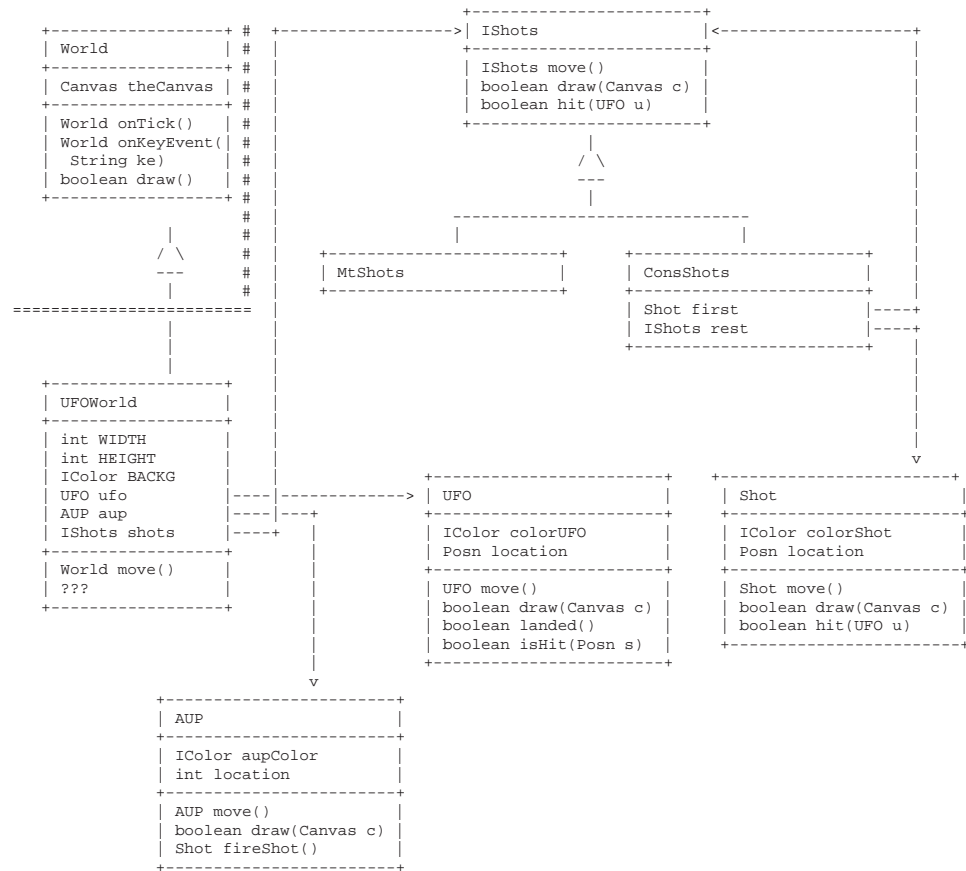
Figure 93: The World of UFOs: Class diagrams

Following the design recipe in section 19.5 and the specification of *World*
in section 19.7, we must turn *UFOWorld* into a subclass of *World* and over-
ride the three **abstract** methods. Figure 93 displays the revised class dia-
gram for our game program. The most important part of the diagram is the
refinement arrow from *UFOWorld* to *World* across the library boundary. It
reminds the reader immediately that *UFOWorld* uses a package and, in this

specific case, that the subclass must override *onTick*, *onKeyEvent*, and *draw* to achieve the desired effects.

Of these to-be-overridden methods we already have *draw*:

inside of *UFOWorld* :
boolean *draw*(*Canvas c*) {
  return
    c.drawRect(**new** *Posn*(0,0),**this**.*WIDTH*,**this**.*HEIGHT*, **this**.*BACKG*)
    && **this**.*ufo.draw*(*c*) *&&* **this**.*aup.draw*(*c*) *&&* **this**.*shots.draw*(*c*);
}

We developed this method in section 16.3 just so that we could see what the world of UFOs looks like. Its definition reflects our thinking back then that a drawing method must consume an instance of *Canvas*. Now we know, however, that a *UFOWorld* is also a *World*, which already contains a canvas:

inside of *UFOWorld* :
boolean *draw*() { // (version 2)
  return
    this.*theCanvas*
     .*drawRect*(**new** *Posn*(0,0),**this**.*WIDTH*,**this**.*HEIGHT*,**this**.*BACKG*)
    && **this**.*ufo.draw*(**this**.*theCanvas*)
    && **this**.*aup.draw*(**this**.*theCanvas*)
    && **this**.*shots.draw*(**this**.*theCanvas*);
}

Since *draw* is defined within *UFOWorld*, it has access to this inherited canvas and can use it to display the world. Furthermore, it passes this canvas to the *draw* methods of the various objects so that they can draw themselves onto the canvas.

Exercise

Exercise 19.16 The *draw* method can benefit from local variable declarations, especially if the names are suggestive of the desired action:

inside of *UFOWorld* :
boolean *draw*() { // (version 2, with local variables)
  boolean *drawBgd* = **this**.*theCanvas*
    .*drawRect*(**new** *Posn*(0,0),**this**.*WIDTH*,**this**.*HEIGHT*,**this**.*BACKG*);
  boolean *drawUFO* = **this**.*ufo.draw*(**this**.*theCanvas*);
  . . .
  return true;
}

Explain how this version of the method works. Reformulate the rest of the method and use it to ensure it still works. ∎

```
// the world of UFOs, AUPs, and Shots
class UFOWorld extends World {
 ...
  // move the ufo and the shots of this world on every tick of the clock
  World onTick() {
    if (this.ufo.landed(this)) {
      return this.endOfWorld("You lost."); }
    else { if (this.shots.hit(this.ufo)) {
      return this.endOfWorld("You won."); }
    else {
      return this.move(); }
    }
  }

  // what happens to this world on a key event
  World onKeyEvent(String k) {
    if (k.equals("up")) {
      return this.shoot(); }
    else { if (k.equals("left") || k.equals("right")) {
      return this.new UFOWorld(this.ufo,this.aup.move(this,k),this.shots); }
    else {
      return this; }
    }
  }

  // draw this world
  boolean draw() {
    return this.theCanvas.drawRect(...) // draw background
      // draw the objects
      && this.ufo.draw(this.theCanvas)
      && this.aup.draw(this.theCanvas)
      && this.shots.draw(this.theCanvas);
  }
 ...
}
```

Figure 94: Animating the *UFOWorld*

The *onTick* method is the next method that needs an overriding defi-

nition. Its purpose is to compute the next world. In our case study, this next world is a world in which the UFO has dropped a little bit more and in which the shots have risen a little bit, too. Except that if the UFO has landed or if one of the shots has hit the UFO, the game is over. Put differently, the method must distinguish three different scenarios:

- in the first scenario, the UFO has reached the ground and has landed;

- in the second scenario, one of the shots has gotten close enough to the UFO to destroy it;

- the third scenario is the typical one; the UFO didn't land and no shot got close to it.

Turning this case distinction into a method is routine work. Still it is worth considering how to get to its definition from the template:

inside of *UFOWorld* :
World onTick() {
  return . . . **this**.*ufo.move*() . . . **this**.*aup.move*() . . . **this**.*shots.move*()
}

The template's expressions remind us that a *UFOWorld* consists of a *ufo*, an *aup*, and a list of *shots*. Each of these comes with its own methods, which the template doesn't indicate here but it reminds us of this fact. When we consider the purpose statement and our discussion of it, however, everything makes sense. We can use **this**.*ufo.landed*() to find out whether we are in the first scenario. Similarly, if an invocation of **this**.*shots.hits* yields true, we know that we are in the second scenario. Lastly if neither of these conditions hold, everything proceeds normally. In the first two cases, the world stops; in the last one all the objects move along.

Figure 94 displays the complete definition of *onTick*. At this point, the class definition is complete enough to observe the animation. Specifically, the UFO should descend and the shots should fly, though the AUP won't react to keystrokes yet. Try it out! (Don't forget to supply a simplistic definition of *onKeyEvent*.)

With a first success under our belt, we can turn our attention to the player's actions. So far, your manager imagines these actions:

1. hitting the up-arrow key fires a shot;

2. hitting the left or right arrow key moves the AUP to the left or right;

3. and hitting any other key affects nothing.

While the second kind of action concerns the AUP, the first one uses the AUP and affects the collection of shots as a whole.

Here is the template that distinguishes these three arguments:

<u>inside of *UFOWorld* :</u>
World onKeyEvent(*String k*) {
  if (*k*.*equals*("up")) {
    return ... **this**.*ufo* ... **this**.*aup* ... **this**.*shots*; }
  else { **if** (*k*.*equals*("left") || *k*.*equals*("right")) {
      return ... **this**.*ufo* ... **this**.*aup* ... **this**.*shots*; }
  else {
    return ... **this**.*ufo* ... **this**.*aup* ... **this**.*shots*; }
  }
 }

As before, the expressions in the branches remind us of what data to which the method has access.

The rest of the development proceeds as before:

1. In the first scenario, the player hits the up-arrow key, which means that *onKeyEvent* has to use the *shoot* method for firing a shot.

2. In the second scenario, the player hits a left-arrow key or a right-arrow key; in that case, the key event is forwarded to the *AUP*, which interprets the key stroke and moves the *AUP* appropriately.

3. Otherwise, nothing happens.

The resulting method definition is included in figure 94.

The game is ready for you to play. Even though it is a minimalist game program, it does display a canvas; the UFO is landing; the shots are flying; and the AUP is moving under your control. You can win and, if you really want and try hard, you can lose. Enjoy, and read the rest of the section to find out how you can add some spice to the game.

Exercises

Exercise 19.17 Collect all code fragments for the classes *UFOWorld*, *UFO*, *AUP*, *IShots*, *MtShots*, and *ConsShots*.

Develop tests for the *onTick* and *onKeyEvent* methods. That is, create simple worlds and ensure that *onTick* and *onKeyEvent* create correct worlds from them.

Add the following line to your *Examples* class from figure 34:

inside of *Examples* :
boolean *testRun* = **check this**.*w2. bigBang*(200,500,1/10) **expect** true;

This displays a 200 by 500 canvas; starts the clock; and ticks it every 1/10th of a second. You should see the UFO descend and, when it gets close enough to the ground, you should see it "land." The two shots of *w2* should fly. If you press the up arrow, you should see a shot leaving the AUP. Watch it fly. Add another shot. And another one. Now press a left or a right arrow key. The AUP should move in the respective direction.

Lastly, add a *run* method to *UFOWorld* so that you can run the game easily from the interactions window. ∎

```
// randomness in the world
class Random … {
 …
 // returns the next pseudorandom, uniformly distributed boolean
 // value from this random number generator's sequence
 boolean nextBoolean() { … }

 // returns a pseudorandom, uniformly distributed int value between
 // 0 (inclusive) and the specified value (exclusive), drawn from this
 // random number generator's sequence
 int nextInt(int n) { … }
 …
}
```

Figure 95: The *Random* class

Exercise 19.18 As is, the game is rather boring. The UFO descends in a straight manner, and the AUP shoots straight up. Using random numbers, you can add an element of surprise to the game.

Modify the method *move* in *UFO* so that the object zigzags from left to right as it descends. Specifically, the *UFO* should move to the left or right by between 0 and 3 pixels every time it drops a step. As the UFO swerves back and forth, make sure that it never leaves the visible part of the world. Think hard as you create examples and reasonable tests. Even though you cannot predict the precise result once you add randomness, you *can* predict the proper range of valid results.

ProfessorJ and Java provide random values via the *Random* class. Figure 95 displays the relevant excerpt from this class for this exercise. You can use it after you import the package **java.util.Random** just like you import **geometry** and **colors**.

Hint: You may wish to employ iterative refinement. For the first draft, design a *UFO* that swerves back and forth regardless of the world's boundaries. For the second draft, design an auxiliary method that tests whether the new *UFO* is outside the boundaries; if so, it rejects the movement and tries again or it just eliminates the side-ways movement. **Note**: the technique of generating a candidate solution and testing it against some criteria is useful in many situations and naturally calls for method composition. In *How to Design Programs*, we covered it under "generative recursion." ∎

Exercise 19.19 In addition to random movements as in the preceding exercise (19.18), you can also turn the AUP into something that acts like a realistic vehicle. In particular, the AUP should move continuously, just like the UFO and the shots.

Implement this modification. Let the design recipe guide you. **Hint**: Start by removing the *move* method from the *AUP* class and add a speed field to the AUP. A positive speed moves the AUP to the right; a negative speed moves it left.

The meaning of a key stroke changes under this scenario. Hitting the left or right arrow key no longer moves the vehicle. Instead, it accelerates and decelerates the vehicle. For example, if the AUP is moving to the right and the player hits the right arrow key, then the AUP's speed should increase. If the player hits the left key, the AUP's speed should decrease. ∎

Exercise 19.20 During a brainstorming session, your team has come up with the idea that a *UFO* should defend itself with *AUP*-destroying charges. That is, it should drop charges on a random basis and, if one of these charges hits the AUP, the player should lose the game.

Implement this modification after completing exercise 19.18. A charge should descend at twice the speed of the UFO, straight down from where it has been released. ∎

Now that we have a tested, running program, the time has come to edit it. That is, we look for replications, for chances to eliminate common patterns, for opportunities to create a single point of control. The best starting point for this exercise is always the class diagram. Start with a look inside unions for common fields and methods; create a common superclass

where possible. Then compare classes that have similar purposes and appearances; check whether introducing a union makes sense.

In the "War of the Worlds" program, there is little opportunity for either kind of abstraction. It definitely lacks a union that can benefit from lifting fields and code, though the revised diagram (see figure 93) suggests one pair of similar classes: *UFO* and *Shot*. Both contain two fields with the same type. We also know that both contain *draw* and *move* methods, so perhaps they deserve some cleanup. For the exercise's sake, let's see how far abstraction takes us here.

```
// represent a descending UFO              // represent an ascending shot
+------------------------+                 +-----------------------+
| UFO                    |                 | Shot                  |
+------------------------+                 +-----------------------+
| Posn location          |                 | Posn location         |
| IColor colorUFO        |                 | IColor colorShot      |
+------------------------+                 +-----------------------+
| boolean draw(Canvas c) |                 | boolean draw(Canvas c)|
| UFO move()             |                 | Shot move()           |
+------------------------+                 +-----------------------+
```

Figure 96: Comparing UFOs and shots

Once you have identified two (or more) candidates, refine the class diagrams and juxtapose them. Also remind yourself of their purpose, just as the design recipe says. For our two candidates, figure 96 displays the refined class diagrams. The diagram lists all fields and those methods that, based on their purpose, may have a common abstraction. Based on this comparison, it is clear that both classes represents strictly vertically moving objects; that they overlap in two fields with identical types and purposes; and that they have two similar-looking methods. The remaining methods in these classes have obviously nothing to do with each other, so they have been eliminated from the diagram to keep things simple.

An actual revision of the class diagram requires a close look at the method definitions. The *draw* methods are definitely distinct, but a moving object in *UFOWorld* should definitely have one. It is therefore possible to add an **abstract** method in the common superclass that requires one from every subclass. In contract, the *move* methods are required and similar to each other, yet they don't even agree on their return types. For that reason alone, we cannot lift the methods to a new, common class.

Given this analysis, figure 97 shows a revision of the relevant portion of the class diagram. The new *AMovable* class introduces the common *color* and *location* fields, though of course, we had to perform some renaming in the process.

The two revised classes look approximately like this now:

```
+--------------------------------+
| AMovable                       |
+--------------------------------+
| Posn location                  |
| IColor color                   |
+--------------------------------+
| abstract boolean draw(Canvas c)|
+------------------------------ +
                |
              / \
              ---
                |
      +------------------------+
      |                        |
+------------+          +-------------+
| UFO        |          | Shot        |
+------------+          +-------------+
| UFO move() |          | Shot move() |
+------------+          +-------------+
```
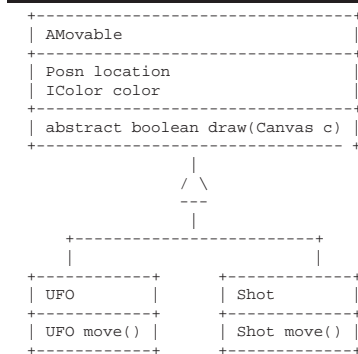
Figure 97: Abstracting over movable objects

```
class UFO {                           class Shot {
  UFO(Posn loc) {                       Shot(Posn loc) {
    super(loc,new Green());               super(loc,new Yellow());
  }                                     }

  // move this UFO down by 3 pixels     // move this Shot up by 3 pixels
  UFO move() {                          Shot move() {
      return                              return
      new UFO( this.loc.translate(0,3) );   new Shot( this.loc.translate(0,−3) );
  }                                     }
  …                                     …
}                                     }
```

The gray-shaded expressions stand out because they are nearly identical
and suggests an abstraction. Both produce a new location, a *Posn* to be
precise, for the new objects. Hence, we should naturally think of actually
adding a method to *Posn* that accomplishes this translation of a point.

   Although we cannot modify *Posn* itself, because it belongs to the un-
modifiable **draw** package, we can create a subclass of *Posn* that provides
this service:

```
class Location extends Posn {
  Location(int x,int y) { super(x,y); }

  Location moveY(int delta) {
    return new Location(this.x,this.y+delta);
  }
}
```

In turn, *AMovable* should use *Location* for the type of *location*, and *UFO* and *Shot* can then use *moveY* with 3 and −3 to move the respective objects:

```
class UFO {                           class Shot {
 …                                     …
  // move this UFO down by 3 pixels     // move this Shot up by 3 pixels
  UFO move() {                          Shot move() {
    return                               return
      new UFO(this.location.moveY(3));     new Shot(this.location.moveY(−3));
  }                                     }
 …                                     …
}                                     }
```

Figure 98 displays the class diagram with all the changes. In this diagram, you can now see *two* places where the actual program extends a class from the **draw** package: *World* and *Posn*. Study it well to understand how it differs from the diagram in figure 93.

Exercises

Exercise 19.21 Define an interface that specifies the common methods of the *Shot* and *UFO* classes. ▪

Exercise 19.22 Collect all code fragments for the classes *UFOWorld*, *UFO*, *AUP*, *IShots*, *MtShots*, and *ConsShots* and develop tests. Then introduce *AMovable*; make sure the tests still work. Finally define *Location*; again make sure the tests still work. ▪

Exercise 19.23 Exercises 19.18 and 19.20 spice up the game with random movements by the *UFO* and random counter-attacks. Create a union of movable objects in this context, starting with *UFO* and *Shot*. Then turn the class representing charges into a subclass of the *AMovable* class, too.

Compare the list of shots and the list of charges. Is there anything to abstract here? Don't do it yet. Read chapter V first. ▪

19.9   Mini Project: Worm

"Worm" is one of the oldest computer games; depending on when you were young and where you grew up, you may also know the game as "Snake." The game is straightforward. When you start it, a worm and a piece of food appear. The worm is moving toward a wall. Don't let it run

```
                        #                              #
 draw:                  #                              #  draw:
                        #                              #
                        #                              #
 +------------------+   #   +--------+                 #
 |  World           |   #   +----------> | IShots |<------------+   #  +------+
 +------------------+   #   |          +--------+               |   #  | Posn |
 | Canvas theCanvas |   #   |                                   |   #  +------+
 +------------------+   #   |              / \                  |   #  | int x|
          |             #   |              ---                  |   #  | int y|
          |             #   |               |                   |   #  +------+
          |             #   |       ------------------          |   #     |
          |             #   |       |                |          |   #     |
          |             #   |  +---------+     +-------------+   |   #     |
          |             #   |  | MtShots |     | ConsShots   |   |   #    / \
          |             #   |  +---------+     +-------------+   |   #    ----
         / \            #   |  +---------+     | IShots rest |----+   #     |
         ---            #   |                  | Shot first  |----+   #     |
          |             #   |                  +-------------+    |   #     |
 ========================== |                                    |   ===================
          |                 |                +---------------+    |      |
          |                 |                | AMovable      |    |      |
          |                 |                +---------------+    |      |
 +------------------+       |                | IColor color  |    |  +------------------+
 | UFOWorld         |       |                | Location loc  |-------->| Location         |
 +------------------+       |                +---------------+    |  +------------------+
 | int WIDTH        |       |                        |           |  +------------------+
 | int WIDTH        |       |                       / \          |  | Location moveY() |
 | int WIDTH        |       |                       ---          |  +------------------+
 | int WIDTH        |       |                        |           |
 | int HEIGHT       |       |             +-------------------+  |
 | IColor BACKG     |       |             |                   |  |
 | UFO ufo          |----|--+             |                   |  |   v
 | AUP aup          |----|-+|   +---------------+     +---------------+
 | IShots shots     |----+ |+-->| UFO           |     | Shot          |
 +------------------+      |    +---------------+     +---------------+
          v               +---------------+
 +---------------+
 | AUP           |
 +---------------+
 | IColor aupColor|
 | int location  |
 +---------------+
```
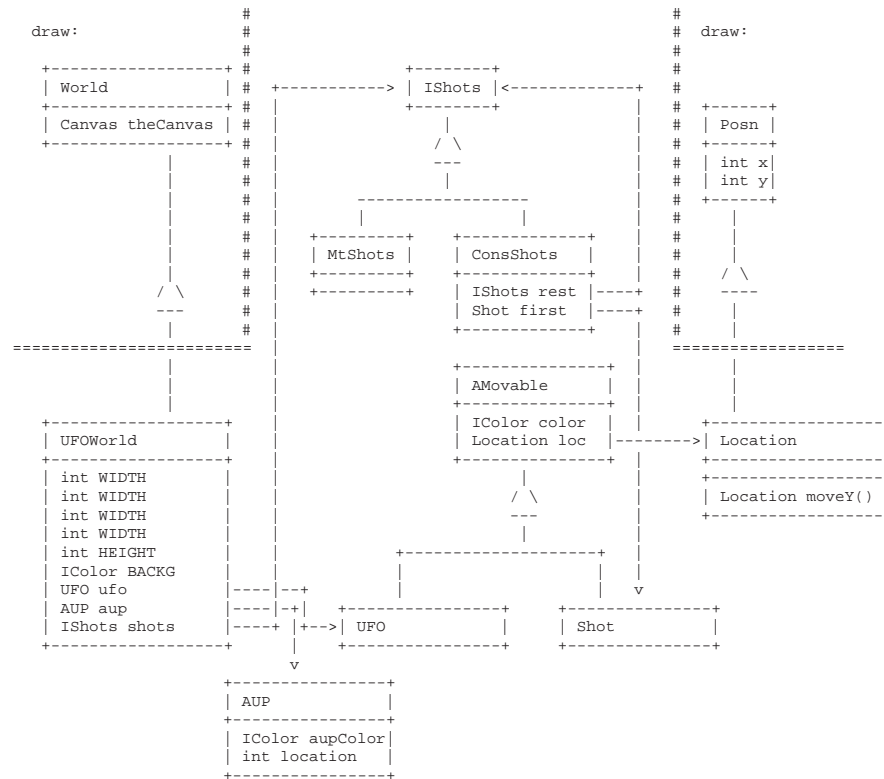
Figure 98: The Revised World of UFOs: Class diagrams

into the wall; otherwise the game is over. Instead, use the arrow keys to control the worm's movements.

The goal of the game is to have the worm eat as much food as possible. As the worm eats the food, it becomes longer; more and more segments appear. Once a piece of food is digested, another piece appears. Of course, the worm's growth is dangerous. It can now run into itself and, if it does, the game is over, too.

This sequence of screen shots illustrates how the game works in practice:
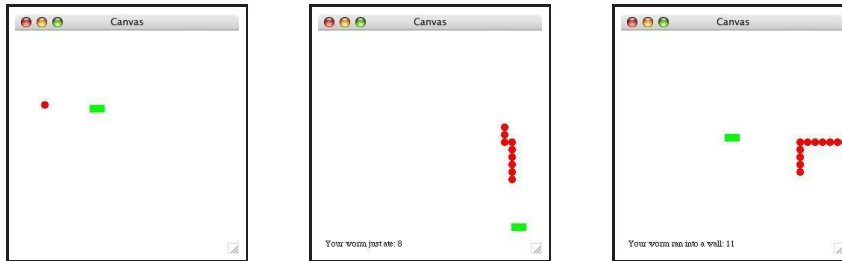
On the left, you see the initial setting. The worm consists of a single segment, its head. It is moving toward the food, which is a rectangular block. The screen shot in the center shows the situation after some "feedings." The worm now has eight segments (plus the head) and is squiggling toward a piece of food in the lower right corner. In the right-most screen shot the worm has run into the right wall. The game is over; the player scored 11 points.

The following exercises guide you through the design and implementation of a Worm game. They follow the design recipe and use iterative refinement. Feel free to create variations.

Exercises

Exercise 19.24 Design a *WormWorld* class that extends *World*. Equip the class with *onTick* and *onKeyEvent* methods that do nothing. Design the method *draw*, which should draw a yellow background for now.

Assume that neither the *Food* nor the *Worm* class has any attributes or any properties. Draw a class diagram anyway; maintain it throughout the exercise.

Make sure that the following example class works:

```
class Examples {
  Worm w = new Worm();
  Food f = new Food();
  WormWorld ww = new WormWorld(this.f,this.w);


  boolean testRun = check this.ww.run() expect true; // keep last test
}
```

That is, design a method *run* that starts the clock, displays the canvas, and invokes *draw*. ∎

Exercise 19.25 Design a data representation for a one-segment worm that continuously moves. The worm should move in a cardinal direction. Draw the worm segment as a disk and ensure that the worm moves exactly one diameter per step. Represent directions with *String*s.

Integrate your worm with *WormWorld* from exercise 19.24 and ensure that it can display a moving worm.

For this exercise, you need not worry about the worm running into the walls of the world. ▪

Exercise 19.26 Modify the program from exercise 19.25 so that the animation stops, when the worm moves runs into the walls of the world. ▪

Exercise 19.27 Design the method *steer* for the *Worm* class of exercise 19.26. Assume the method consumes a direction and sends the worm into this direction; it does not change the worm's location or speed.

Integrate your new class with *WormWorld* of exercise 19.26 so that the player can control the worm's movements with the four arrow keys. **Hint**: The *onKeyEvent* method should change the worm's direction only if it receives a cardinal direction as input. ▪

Exercise 19.28 Design a data representation for a multi-worm that continuously moves and that the player can control with arrow keys. **Hints**: The worm should now consist of a distinct head segment and an optionally empty tail of segments that are like the head except for appearances. Think of the worm's movement as a two-step process: first the head moves into the desired direction and then the segments of the tail move into the position of the preceding segment, with the first segment moving into the position of the head segment.

You may safely assume that the worm runs into the walls of the world only if the head hits the wall. For your first design, ignore the possibility that the worm may run into itself. Then create a second design that takes this possibility into account.

Integrate your worm with *WormWorld* from exercise 19.27 and ensure that it can display a moving worm. ▪

Exercise 19.29 Design a data representation for food particles. In addition to the usual *draw* method, it should have two other methods:

- *eatable*, which is given the position of the worm's head and determines whether it is close enough to eat this food particle;

- *next*, which creates the next food particle and ensures that it is not at the same spot as this one.

  Hint: This design requires a modicum of "generative recursion" (see *How to Design Programs*(Part V) or exercise 19.18).

Integrate it with the *WormWorld* from exercise 19.28, so that the display shows the food. You do not need to worry about what happens when the worm gets close to, or runs over, the food. ∎

Exercise 19.30 Design the method *eat* for *Worm* from exercise 19.28. Eating means that the head moves another step in the same direction in which the worm was moving and the worm's tail grows by one segment. Explain the assumption behind this description of "eating."

Integrate the modified *Worm* class with the *WormWorld* class from exercise 19.29. A worm should eat only if the food particle is eatable. ∎

Exercise 19.31 Design the class *Game*, which sets up a worm game with a random initial position for a piece of food and a worm with just a head, and a method for starting the game. ∎

Exercise 19.32 Edit your program, i.e., look for opportunities to abstract. ∎

20 State Encapsulation and Self-Preservation

Programs come with all kinds of assumptions. For example, the *Date* class in the first section of the first part of this book assumes that the given numbers represent a real date. Our dealings with the geometric shapes of chapter II assume that they are entirely in the quadrant represented by a canvas, and we state so several times. And, concerning the world of dropping blocks, we want this assumption to hold:

> *BlockWorld* displays a block that is dropping from near the top of the canvas to the bottom.

When an assumption concerns an entire program execution, especially one in which objects move and work, we speak of assumptions about the STATE of the program; the attributes of each object are the STATE of the object.

Establishing and maintaining such assumptions becomes an issue of self-preservation when many programmers work on many inter-related classes (for a long time). Starting in section 13.2 we assumed that because of this, our programs should include design information and possibly in

easily checkable form. Back then we explained that this is what types are all about. Types for fields and method signatures aren't enough though. In this section, we introduce three essential linguistic mechanisms from Java for establishing invariants, for maintaining them, and for protecting them.

20.1   The Power of Constructors

Here is the *Date* class with informal constraints:

```
class Date {
  int day; // between 1 and 31
  int month; // between 1 and 12
  int year; // greater than or equal to 1900

  Date(int day, int month, int year) {
    this.day = day;
    this.month = month;
    this.year = year;
  }
}
```

It represents dates via the common numeric notation, e.g., **new** *Date*(5, 6, 2003) stands for June 5, 2003. The problem is, however, that some other part of the program can—accidentally or intentionally—create an instance like **new** *Date*(45, 77, 2003), which has no meaning in the real world.

The programmer has dutifully documented this assumption but the code does not enforce them. By choosing int as the type for the fields, the programmer ensures that the *Date* always consists of Java-style integers, but this specification cannot even ensure that the given ints are positive numbers. Thus, **new** *Date*(-2,77,3000) is also an instance of the class.

In Professor's Intermediate language we can address this problem with a conditional constructor for the class: see figure 99. The constructor in this revised *Date* class consists of an **if** statement. The test ensures that the value of *day* is between 1 and 31, the value of *month* is between 1 and 12, and *year* is greater than 1900. If so, the constructor sets up the object via initialization "equations" as usual; otherwise, it signals an error with *Util.error*. Put differently, the constructor expresses the informal comments of the original design via a test and thus guarantees that each instance of *Date* satisfies the desired conditions.

Figure 100 shows a similar revision of the data representation of shapes, illustrated with the case of circles. The inside-the-quadrant assumption for shapes is attached to the interface. It is relevant, for example, for computing

```
class Date {
  int day;
  int month;
  int year;

  Date(int day, int month, int year) {
     if ((1 <= day && day <= 31)
         && (1 <= month && month <= 12)
         && (1900 <= year)) {
       this.day = day;
       this.month = month;
       this.year = year; }
     else {
       Util.error("the given numbers do not specify a date"); }
  }
}
```

Figure 99: A conditional constructor for the *Date* class

the distance of a shape to the origin. Based on this assumption, we compute
the distance of a circle like this:

inside of *Circle* :
double *distTo0*() {
  return this.*loc.distTo0*() − **this**.*radius*;
}

This method computes the distance between the origin and the center and
then subtracts the radius, because the closest point to the origin is on the
circle when the circle is inside the quadrant.

The left side in figure 100 presents a revision similar to the one for the
Date class. As for *Date*, the constructor consists of an **if** statement that tests
whether certain conditions are satisfied. Specifically, the test ensures that
radius is positive and that the center is at least far enough from the corners
of the canvas.

The right side presents an equivalent alternative that employs an aux-
iliary boolean-typed method to check whether the given values are valid.
A call from the constructor is curious because until now, we have assumed
that the object exists only after the constructor has finished its work. Here,
however, the constructor uses a method from within the class before it even
sets up the fields with proper values. Indeed, the auxiliary method cannot

```
// the class of all geometric shapes
// (completely inside the quadrant)
interface IShape {
  // draw this shape into the given canvas
  boolean drawn(Canvas c);
}
```

```
class Circle implements IShape {
  Posn loc;
  int radius; // positive

  Circle(Posn center, int radius) {
      (radius > 0)
      && (center.x − radius >= 0)
    if ( && (center.y − radius >= 0) ) {
      this.center = center;
      this.radius = radius; }
    else {
      Util.error("invalid circle!"); }
  }




  boolean draw(Canvas c) {
    return c.drawDisk(...)
  }
}
```

```
class Circle implements IShape {
  Posn loc;
  int radius;

  Circle(Posn center, int radius) {
    if (this.valid(center, radius)) {
      this.center = center;
      this.radius = radius; }
    else {
      Util.error("invalid circle!"); }
  }

  // is this circle inside the quadrant?
  boolean valid(Posn center, int radius) {
          (radius > 0)
          && (center.x − radius >= 0)
    return && (center.y − radius >= 0) ;
  }

  boolean draw(Canvas c) {
    return c.drawDisk(...)
  }
}
```

Figure 100: A conditional constructor for the *Circle* class

use the fields in ProfessorJ and, if it did, things would go wrong.[40]

For a third example, consider the following problem statement:

> ... In the world of dropping blocks, a block appears at location
> (10,20) and drops from there at the rate of one pixel per clock

---

[40] A method that doesn't use the fields of the class—and shouldn't use the fields of the class—is not really a method. It is a plain function and should be declared as such. Java empowers the programmer to express such a fact with the keyword **static**.

tick. A player can control the block with keystrokes . . .

If you ignore the remark on where these blocks should appear, the design of the class and its *drop* method is straightforward. Figure 101 displays the result on the right side.

Naturally ignoring the remark is wrong, because it violates the rules of the imagined game. Coming up with a solution, however, is impossible in the language that you know. Here are two ideas:

1. It is possible to initialize fields via "equations" directly:

   ```
   class DrpBlock {
     int x = 10;
     int y = 20;
     . . .
   }
   ```

   Unfortunately, doing so means in ProfessorJ's Intermediate language that these attributes are the same for all possible instances of *DrpBlock*, which is not what we want. The *drop* method must be able to create instances of the class whose *y* field is larger than 10.

2. Since we can have initialization equations for a field either with the field declaration or in the constructor but not both, the second idea is an obvious consequence of the first, rejected one:

   ```
   class DrpBlock {
     int x;
     int y;
     DrpBlock() {
       this.x = 10;
       this.y = 20;
     }
   }
   ```

   As we have seen, the constructor can compute the initial values of a field from its parameters or just use constants. In particular, it could just use the desired constants for an initially created block: (10,20). Of course, this solution is equivalent to the first because it, too, creates nothing but blocks with fixed coordinates.

It turns out that the second solution isn't completely wrong, though this is a topic for the next section and an improved language.

Exercises

Exercise 20.1 Refine the constructor for the *Date* class in figure 99 even more. Specifically, ensure that the created *Date* uses a value of 31 for *day* field only if the given month has that many days. Also rule out dates such as **new** *Date*(30,2,2010), because the month of February never has 30 days and therefore such a *Date* should not be created.

Hint: Remember that only January, March, May, July, August, October, and December have 31 days.

If you are ambitious, research the rules that govern leap years and enforce the proper rules for 30-day months and February, too. ∎

Exercise 20.2 Rectangles like circles are supposed to be located within the quadrant that the canvas represents. Determine conditions that ensure that a rectangle respects this constraint. Design a constructor that ensures that the given initial values satisfy these conditions. ∎

20.2   Overloading Constructors

ProfessorJ:
. . . + access

If you change ProfessorJ's language to Intermediate + access, you gain several new ways of expressing your thoughts on design.

The first and relevant one here is the power of OVERLOADING a constructor.[41] To overload a constructor means to define several constructors, each consuming different types of arguments. You can also overload methods in this manner. While this concept is also useful for methods, we explain it exclusively with constructors here. Overloading for methods works in the exact same manner; we introduce it later when needed.

Figure 101 shows on the right side a version of *DrpBlock* that includes two constructors:

1. the original one, which is needed for the *drop* method;

2. and the new one, discussed above, which creates blocks according to the original problem statement.

---

[41]Other object-oriented languages allow programmers to name constructors, which solves the problem in a more elegant manner than overloading.

```
class DrpBlock {
  int x;
  int y;
  int SIZE = 10;




  DrpBlock(int x, int y) {
    this.x = x;
    this.y = y;
  }

  DrpBlock drop() {
    return
      new DrpBlock(this.x,this.y+1);
  }
}
```

```
class DrpBlock {
  int x;
  int y;
  int SIZE = 10;

  DrpBlock() {
      this.x = 10;
      this.y = 20;
  }

  DrpBlock(int x, int y) {
    this.x = x;
    this.y = y;
  }

  DrpBlock drop() {
    return
        new DrpBlock(this.x,this.y+1) ;
  }
}
```

```
class Example {
  DrbBlock db1 =  new DrpBlock() 1;

  boolean test1 = check this.db1.drop() expect  new DrpBlock(10,21) 2;
}
```

Figure 101: Multiple constructors for one class

Code outside of *DrpBlock* can use the no-argument constructor to create blocks and code inside of *DrpBlock* can use both constructors, depending on what is needed.

Once you have two (or more) constructors, your language also needs a way to choose one of them for evaluation. Java uses the types of the arguments to choose from among the available constructors. First, it enforces that the parameter types of any two constructors differ somewhere. Second, as it type-checks the expressions in the program, it also determines the types of the arguments at each constructor call site. It matches the argument types to the parameter types, and because the latter are distinct,

picks the one, matching constructor as the intended one for future program evaluations.[42]

The example in figure 101 illustrates this point directly. The two constructors have distinct signatures. The first one consumes no arguments; the second consumes two ints. Next take a close look at the three gray-shaded constructor expressions in the figure. The one in the class itself use two ints as arguments; hence it refers to the second constructor. The constructor expression with subscript 1 takes no arguments, meaning it is a reference to the first constructor in the class. Finally, the expression with subscript 2 takes two ints again and therefore uses the second constructor.

Sadly, the introduction of overloading doesn't solve the problem completely. We need even more expressive power. While code outside of *Drp-Block* can utilize the no-argument constructor and thus obtain an appropriate block in the desired initial state, nothing forces the use of this constructor. Indeed, as the figure shows the two-argument constructor is usable and used in the *Example* class. In other words, the introduction of overloaded constructors opens possibilities for violating unspoken or informal assumptions that we cannot just ignore if we wish to reason about code.

20.3 Encapsulating and Privacy

Programming language designers have known about this problem for a long time. Therefore most programming languages provide a mechanism for ENCAPSULATING the state of an object. Encapsulating state means hiding fields, constructors, and methods from other classes as well as their programmers, a group that may include you. Presumably, the creator of a class knows how everything works and won't misuse the pieces of the class accidentally; in contrast, outsiders don't know all of our assumptions that the programmers wishes to hold for the attributes of a class. If they can't use them in any way, they can't misuse them either.

Concretely Java and many other object-oriented languages provide PRIVACY SPECIFICATIONS, which means markers that make the features of a class private or public (or something in between). From a programmer's perspective, these privacy specifications tell Java which pieces of the program (and thus who) can manipulate a part of a class. While Java and ProfessorJ type check a program, they also check the privacy specifications. If one class accidentally refers to a part of some other class in violation of the privacy specifications, Java signals an error and doesn't evaluate the

---

[42]Don't confuse overloading with overriding, which is entirely different.

```
class DrpBlock {
  private int x;
  private int y;
  private int SIZE = 10;

  public DrpBlock() {
    this.x = 10;
    this.y = 20;
  }

  private DrpBlock(int x, int y) {
    this.x = x;
    this.y = y;
  }

  public DrpBlock drop() {
    return new DrpBlock(this.x,this.y+1);
  }
}
```

```
class ExampleBad {
  DrbBlock db1 = new DrpBlock() 1;

  boolean test1 =
    check this.db1.drop()
    expect new DrpBlock(10,21) 2;
}
```

Figure 102: Privacy specifications for classes

program. Like types, privacy specifications are approximations of what programmers really want to say and don't provide perfect protection, but without them, your classes can quickly get into a trouble once they become a component of a system of classes.

A privacy specification is an adjective—that is, a descriptive (and often optional) word—for the features of a class: a field, a constructor, or a method. Java uses four privacy specifications:

private means the respective piece is accessible only inside the class;

public allows every class to use this part of the class;

protected makes the piece visible inside this class, its subclasses, their sub-classes, and so on.

If privacy specifications are omitted, the feature is package-private, a notion that we ignore in this book.[43]

---

[43] In Java, a package is a linguistic mechanism for organizing classes and interfaces. Just like constructors, fields, and methods can be visible only within a certain class, classes and interfaces can be visible only within certain packages. A package-private feature is thus visible only within the package, not outside.

The left column in figure 102 illustrates how to use these adjectives to let every reader and Java know that the argument-less constructor is useful for everyone—the world outside the current class as well as the class itself—and that the second constructor is only useful for use within the class. If some other class or the Interactions Window in ProfessorJ now contains code such as **new** *DrpBlock*($-$30,40)), Java does not run the program.

With this in mind, take a look at the right column. It contains a sample class for testing the only method of *DrpBlock*: *drop*. The class contains two gray-shaded constructor expression with subscripts; one is legal now, the other one is not. Specifically, while the expression with subscript 1 is legal because it uses the constructor without arguments, the expression with subscript 2 is illegal given that the constructor of two arguments is **private**. Thus, the *EampleBad* class does not "type check," i.e., ProfessorJ highlights the expression and explains that it is inaccessible outside of *DrpBlock*.

In addition, the code in the figure labels every field as private so that other classes can't exploit the internal data representation of the field. Thus, neither x nor y are accessible attributes outside the class. As a result, it is impossible to write the following tests in a sample class:

> **check new** *DrpBlock*().*drop*().*x* **expect** 10
> *&&*
> **check new** *DrpBlock*().*drop*().*y* **expect** 21

Put differently, the method in *DrpBlock* is not testable given the current privacy specifications and method interface. In general, in the presence of privacy specifications it occasionally becomes necessary to equip a class with additional methods, simply so that you can test some existing classes.

Still, we can reason about this code and argue why it enforces the basic assumption from the problem statement:

> *DrpBlock* represents dropping blocks, which enter the world at (10,20) and move straight down.

Because of the privacy adjectives, the first and **public** constructor is the *only* way to create an instance of *DrpBlock*. This instance contains a block that is at (10,20). An invocation of *drop* creates a block with the same x coordinate and a y coordinate that is one larger than the one in the given block. Hence, on a computer canvas, this new block would be drawn one pixel below the given one. No other class can use the second constructor to create blocks at random places, meaning we know that the described scenario is the only one possible.

```
// the world of a dropping block
class BlockWorld extends World {
  private int WIDTH = 100;
  private int HEIGHT = 100;
  private IColor BACKGROUND = new Red();

  private DrpBlock block;

  // initial constructor
  public BlockWorld() {
    this.block = new DrpBlock();
  }

  // update constructor
  private BlockWorld(DrpBlock block) {
    this.block = block;
  }

  // what happens to this
  // world on every tick of the clock
  public World onTick() {
    return new BlockWorld(this.block.drop());
  }
  ...
}
```

Figure 103: Privacy and inheritance

Privacy specifications also interact with inheritance and subclassing. If some class *A* extends some other class *B*, then the privacy specifications in *A* must make a feature equally or more visible than *B*.

Take a look at figure 103, which presents the world of dropping blocks with privacy specifications. Its attributes, including the falling block, are private. Its two overloaded constructors are public and private, just like in *DrpBlock*. In contrast, the inherited methods are **public** because that is how the *World* superclass labels its corresponding methods.

Exercises

Exercise 20.3 Design a method (or methods) so that you can test the *drop* method in *DrpBlock* in the presence of the given privacy specifications. ∎

```
// to represent a world with its visual part drawn on a canvas
abstract class World {
  protected Canvas theCanvas = new Canvas();

  // open a width by height canvas,
  // start the clock, and make this world the current one
  public boolean bigBang(int width, int height, double s) { … }

  // process a tick of the clock in this world
  public abstract World onTick() { … }

  // process a keystroke event in this world
  public abstract World onKeyEvent(String ke) { … }

  // draw this world
  public abstract boolean draw() { … }

  // stop this world's clock
  public World endOfWorld(String s) { … }
}
```

Figure 104: Animated *World*s

Exercise 20.4 Complete the transliteration of figure 92 into the definitions of figure 103 and add appropriate privacy specifications. ▮

Exercise 20.5 Suppose you wanted to enforce this revised assumption:

> The block in *BlockWorld* is always within its boundaries. Visually, the program displays a block that is dropping from near the top of the canvas to the bottom and stops there.

How do you have to change the class? ▮

The third privacy adjective (**protected**) is useful for fields and methods in classes that serve as superclasses. Let's look at *World* from **draw.ss**, which is intended as a superclass for different kinds of world-modeling classes. Figure 104 displays a view of this *World* class with privacy specifications. As you can see, the class labels its canvas with **protected**, making it visible to subclasses but inaccessible to others; after all, we don't want arbitrary methods to accidentally draw the wrong kind of shape on the canvas.

If an extension wishes to have some other class or method to have the canvas, then the methods in the subclasses must hand it out explicitly as an argument to other classes. For an example, look at the *draw* in *BlockWorld* (see figure 91); it uses **this**.*theCanvas* as an argument when it calls the *draw* method of *DrpBlock*. When a method hands out the value of a **protected** field, its programmer assumes responsibility (to the community of all programmers on this project) for the integrity of the associated assumptions.

```
+--------------------------------+  #  +------------------------------+
| World                          |  #  | Block                        |
+--------------------------------+  #  +------------------------------+
| protected Canvas theCanvas     |  #  | protected int down           |
+--------------------------------+  #  | protected int right          |
| public World onTick()          |  #  +------------------------------+
| public World onKeyEvent(String)|  #  | public boolean draw(Canvas)  |
| public boolean draw()          |  #  | private boolean paint(Canvas)|
| ...                            |  #  +------------------------------+
+--------------------------------+  #                 |
        |                           #                 |
        |                           #                 |
        |                           #                 |
       / \                          #                / \
       ---                          #                ---
        |                           #                 |
=====================================#      +-------+--- ...
        |                                   |
+--------------------------------+          |
| BlockWorld                     |          |
+--------------------------------+          |
| private DrpBlock block         |----->| DrpBlock                    |
| private IColor BACKG           |      +---------------------------+
| public int WIDTH               |      | private int deltaY        |
| public int HEIGHT              |      +---------------------------+
+--------------------------------+      | DrpBlock drop()           |
| public World onTick()          |      | DrpBlock steer(String ke) |
| public World onKeyEvent(String k)|    +---------------------------+
| public boolean draw()          |
| private boolean drawBackground()|
+--------------------------------+
```

Figure 105: Animating dropping blocks

For a second, diagramtic example of **protected** status, see figure 105. It re-displays the diagram of blocks, dropping blocks, worlds of blocks from figure 92. The figure recalls that *Block* defines the coordinates for a block and methods for drawing them. Naturally, the *draw* is **public** because *BlockWorld* must use it; *paint*, however, is **private** because it is just an auxiliary.

The fields in *Block* are **protected**. Clearly, no outside class needs to know about the coordinates, but subclasses of *Block* must be able to refer to them and create instances with different values for *x* and *y*. Therefore choosing **protected** is again the correct choice for these fields.

20.4   Guidelines for State Encapsulation

Once you start designing programs that you may keep around for a long time or that other people may read and modify, it becomes important to protect the integrity of your (assumptions about) objects. The purpose of an object is to represent information about the world. As the world evolves, the objects evolve too. If possible, your objects should initially represent the world's information properly, and as your program and its methods create new objects from the existing ones, these transitions should create objects that can be interpreted in the real world.

When objects can accidentally—or intentionally—misuse the pieces of other objects, we have no guarantees that our collection of objects represent the state of the world properly. To address this problem partly, we recommend that from now on, you always encapsulate the state of your objects via privacy specifications and well-designed constructors.

As far as privacy is concerned, our guideline is simple: equip all constructors, fields, and methods in a class with privacy attributes. To make this concrete, we suggest you choose as follows:

1. If in doubt, use **private**;

2. A method must be **public** if it is visible through an interface.

3. A method is decorated with **public** even if it isn't visible through an interface but some unrelated class must use it.

4. A field is **protected** if only subclasses should refer to it. The same holds for a method.

5. A field is **public** if some unrelated class must refer to this field, e.g., fields that describe global attributes of a program. This is rare.

6. A constructor in an abstract class is **protected**, because subclasses (may) need to refer to it.

7. A constructor is **private** unless it is acceptable that some other class creates instances with these properties.

Keep in mind that these are guidelines. Even though experience shows that they are valid in many, if not most, situations, you may encounter a class hierarchy one day where you need specifications different from those suggested. If so, analyze the situation carefully.

```
// the world of worms
class WormWorld extends World {
  public int WIDTH = 200;
  public int HEIGHT = 200;
  Segment head;

  public WormWorld() {
    this.head = new Segment(this)
  }

  private WormWorld(Segment s) {
    this.head = s;
  }

  public World onTick() { … }

  public World onKeyEvent(String ke) {
    return
      new WormWorld(
          this.head.move(this)) ;
  }

  public boolean draw() { … }
}
```

```
// one segment of the worm
class Segment extends Posn {
  IColor SEGCOL = new Red();
  int RADIUS = 5;

  public Segment(WormWorld w) {
    this.x = w.WIDTH / 2;
    this.y = w.HEIGHT /2;
  }

  private Segment(Segment s) {
    this.x = s.x;
    this.y = s.y;
  }

  public Segment move(Segment pre) {
    return
      new Segment(pre) ;
  }

  public Segment restart(
      WormWorld w) {
    return
      new Segment(w) ;
  }
}
```

Figure 106: Resolving constructor overloading

20.5   Finger Exercises on Encapsulation

Exercise 20.6 The *LightSwitch* class definitions in figure 106 each contain two constructors. Determine for each gray-shaded use of a constructor in figure 106 to which constructor definition it refers. What are the values of *x* and *y* in the resulting *Segment* objects? ∎

Exercise 20.7 The *LightSwitch* class in figure 82—or the one you obtained after abstracting out the commonalities (exercise 19.3)—employs a single constructor. Hence, every time the *flip* function is invoked, a new canvas is created and a new light switch is drawn.

Modify the class so that only one them creates a canvas and the other one exists exclusively so that *flip* can function. Add appropriate privacy

specifications.

Finally, modify the class so that it extends *World* and so that every keystroke on the spacebar flips the switch. ∎

Exercise 20.8 Take a look at the sketches of *Select*, *Presentation*, and *Factory* in figure 107. Each refers to constructors in *Item*, the class on the left.

```
class Item {                          class Select {
  int weight;                             ... new Item(w,p,q) ...
  int price;                          }
  String quality;

  public Item(int w,int p,String q) { class Presentation {
    this.weight = w;                    int p;
    this.price = p;                     ...
    this.quality = q;                   boolean draw(String s) {
  }                                       ... new Item(p,s) ...
                                        }
                                        ...
  public Item(int w,int p) {          }
    this.weight = w;
    this.price = p;                   class Factory {
    this.quality = "standard";          ...
  }                                     int inquireInt(String s) { ... }

  public Item(int p,String q) {         Item create(...) {
    this.weight = 0;                        ... new Item(
    this.price = p;                               inquireInt("pounds"),
    this.quality = q;                             inquireInt("cents")) ...
  }                                     }
  ...                                   ...
}                                     }
```

Figure 107: Resolving constructor overloading, again

Determine for each use of the constructor to which of the constructors in *Item* it refers. Determine the values of the fields for the resulting object. ∎

Exercise 20.9 The left column of figure 108 sketches a class definition for *Set* (exercise 19.4) with privacy specifications. The right column displays the class diagram for *ILin*, *Cin*, and *MTLin*, a list of ints. Assume appropriate class definitions exist.

Which constructor should the *add* method employ? Which constructor should a class employ that contains a *Set*?

```
class Set {
  private ILin elements;

  public Set() {
    this.elements = new MTLin();
  }

  private Set(ILin elements) {
    this.elements = elements;
  }

  // add i to this set
  // unless it is already in there
  public Set add(int i) { … }

  // is i a member of this set?
  public boolean in(int i) { … }
}
```

```
                           +------+
                           | ILin |<------------+
                           +------+             |
                           +------+             |
                              |                 |
                             / \                |
                             ---                |
                              |                 |
                     ----------------           |
                     |              |           |
                 +--------+    +-----------+     |
                 | MTLin  |    | Cin       |     |
                 +--------+    +-----------+     |
                 +--------+    | ILin more |----+
                               | int one   |
                               +-----------+
```
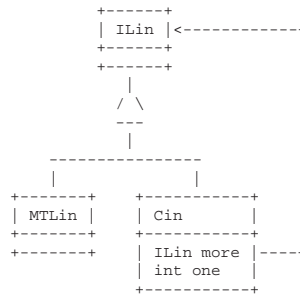
Figure 108: Enforcing assumptions

Argue that the following assumption holds, if *add* and *in* are completed appropriately:

> *Lin* does not contain any int twice.

Or show how to construct a set with the given constructors so that the assumption is wrong.

Complete the definition of *Set* with a **public** *remove* method. ∎

Exercise 20.10 Design the class *SortedList* with the following **interface**:

```
interface ISortedList {
  // add i to this list so that the result is sorted
  ISortedList insert(int i);
  // the first item on this list
  int first();
  // the remainder of this list
  ISortedList rest();
}
```

The purpose of the class is to keep track of integers in an ascending manner:

check new *SortedList*().*insert*(3).*insert*(2).*insert*(4).*first*() **expect** 2

What does

   new *SortedList*().*insert*(3).*insert*(2).*insert*(4).*rest*().*first*()

produce? Make more examples! ▪

Exercise 20.11 Re-visit the "War of the Worlds" project with encapsulation in mind. Make sure that the UFO first appears at the top of the canvas, that the AUP doesn't leave the canvas, that only those shots are in the list of shots that are still within the *UFOWorld*. Can you think of additional properties in this project that encapsulation can protect? ▪

Exercise 20.12 Inspect your solution for the "Worm" game. Add privacy specifications and ensure that you can still run all the tests. ▪

21 Extensional Equality, Part 1

Up to this point, we have relied on **check** … **expect** … to compare objects for us, without a good understanding of how it performs those comparisons. Now that we (may) have both deep, nested hierarchies of interfaces and classes as well as (partially) hidden features of objects, we should take a first, close look at the problem to appreciate at least some of its subtlety. At the end of the next chapter, we return to the idea of equality and study it in even more depth than here and an alternative view.

21.1 Equality for Plain Classes

Recall the very first class definition. It introduced the class of objects that represent bulk coffee sales. Suppose we simplify this example a bit still and just represent the kinds of coffees that are for sale:

   class *Coffee* {
     String origin;
     int *price*; // in cents per pound
     …
   }

In this business, each kind of coffee is identified via its origin country and, for sales purposes, its current price.

   The question is when two instances of this class are the same. Let's see where following the design recipe gets us:

```
// represents bulk coffee for sale
class Coffee {
  private String origin;
  private int price;

  public Coffee(String origin, int price) {
    this.origin = origin;
    this.price = price;
  }

  // is this the same Coffee as other?
  public boolean same(Coffee other) {
    return this.origin.equals(other.origin) && this.price==other.price;
  }
}
```

Figure 109: The sameness of *Coffee*

inside of *Coffee* :
// is **this** the same *Coffee* as *other*?
boolean *same*(*Coffee other*)

The purpose statement phrases the problem as a direct question about **this** object and the *other* instance of *Coffee*. After all, the idea of sameness is about a comparison of two objects.

Given that *Coffee* has two fields, creating the template is also straight-forward:

inside of *Coffee* :
// is **this** the same *Coffee* as *other*?
boolean *same*(*Coffee other*) {
  ... **this**.*origin.mmm*() ... *other.origin.mmm*() ...
  ... **this**.*price* ... *other.price* ...
}

It contains four expressions because two *Coffee* objects contain four fields. The first two indicate that it is possible to invoke another method on the value in the *origin* field because its type is *String*. From the template alone, however, you cannot figure out what sameness could mean here. You need examples, too:

```
class EqExamples {
  Coffee ethi = new Coffee("Ethiopian",1200);
  Coffee kona = new Coffee("Kona",2095);
  Coffee ethi1300 = (new Coffee("Ethiopian",1300));

  boolean test1 = check this.ethi.same(this.ethi)  expect true;
  boolean test2 = check this.kona.same(this.ethi) expect false;
  boolean test3 = check this.ethi.same(this.ethi1300) expect false;
}
```

The first example compares two instances of *Coffee* that have the exact same
attributes: both are from Ethiopia and both cost $12 per pound. Naturally,
you would expect true here. The second one compares two objects with
different origins and different prices. Unsurprisingly, the expected answer
is false. Last, even if two *Coffee*s share the origin but have distinct prices,
they are distinct objects.

Together, the examples and the template suggest a point by point—that
is, a field by field—comparison of the objects: see figure 109. For each kind
of value, you use the appropriate operation for comparisons. Here *same*
uses == for ints and *equals* for *String*s. If you solved such exercises as 10.4
or recall the design of *same* for Cartesian points in chapter II, the definition
of *same* in *Coffee* is no surprise.

Given the presence of privacy specifications in the definition of *Coffee*,
you may wonder whether the field access in *same* works out as obviously
desired. Since these instructions for hiding the features of an object concern
the class, i.e., the program text, it turns out that one instance of *Coffee* can
access the secret, hidden fields of another instance of *Coffee* just fine.

Following philosophers, we call this notion of sameness EXTENSIONAL
EQUALITY.[44] Roughly speaking, a method that implements extensional
equality compares two objects on a point by point basis. Usually, it just
compares the object one field at a time, using the equality notion that is
appropriate for the types of the fields. When you design an extensional
equality method, however, it is important to keep in mind what your class
represents and what this representation is to mean to an external observer.
Because this is what extensional equality is truly about: whatever you want
external observers to be able to compare.

A representation of mathematical sets via classes illustrates this point
well. To keep things simple, let's look at sets of two ints. Obviously, such

---

[44]Frege (1848–1925) was the first logician to investigate the idea of a predicate's extension
and extensional equality.

```
// represents a set of two numbers
class Set2 {
  private int one;
  private int two;

  public Set2(int one, int two) {
    this.one = one;
    this.two = two;
  }

  // does this set contain x?
  public boolean contains(int x) {
    return (x == this.one) || (x == this.two);
  }

  // is this the same Set2 as other?
  public boolean same(Set2 other) {
    return
      other.contains(this.one)
      && other.contains(this.two)
      && this.contains(other. one)
      && this.contains(other. two);
  }
}
```

Figure 110: The sameness of sets

a class needs two fields and, at a minimum, a method that determines whether some given int is a member of **this** set, i.e, whether the set contains the int. Designing this kind of data representation is easy for you at this point (see figure 110).

The problem of sameness for sets is quite different from that for *Coffee*. As you may recall from mathematics, two sets are the same if every element of the first set is a member of the second set and vice versa:

$$\text{sets } S1 \text{ and } S2 \text{ are equal, notation: } S1 = S2,$$
$$\text{means that}$$
$$\text{for all } e, e \in S1 \Leftrightarrow e \in S2 \ .$$

Given the *contains* method from *Set2*, this bit of mathematics has an obvious translation into Java:

>   *other*.contains(**this**.*one*)
>   *&& other*.contains(**this**.*two*)
>   *&&* **this**.*contains(other. one)*
>   *&&* **this**.*contains(other. two)*

For the full definition of *same* for *Set2*, see figure 110.

A quick comparison shows that *same* in *Set2* isn't just a field-by-field comparison of the given objects. Instead, each field in **this** set is compared to both fields of the *other* set. The success of one of these comparisons suffices. Thus, this notion of equality is significantly weaker than the field-by-field notion of equality for, say, *Coffee*. Here, it is the mathematics of sets that tells you what *same*ness really means. In other cases, you will have to work this out on your own. The question you will always need to ponder is whether someone who doesn't know how you designed the class and the method should be able to distinguish the two given objects. The answer will almost always depend on the circumstances, so our advice is to explore the problem with as many examples as you need until you know what others want from the class.

Exercises

Exercise 21.1 Develop data examples for *Set2* and turn them into behavioral examples for *contains* and *same*. Translate them into a test suite and run them. Be sure to include an example that requires all four lines in *same*. ∎

Exercise 21.2 Design the method *isSubset* for *Set2*. The method determines whether **this** instance of *Set2* contains all the elements that some given instance of *Set2* contains.

Mathematicians use $s \subseteq t$ to say that "set s is a subset of set t." They sometimes also use the notion of "subset" to define extensional set equality as follows:

$$S1 = S2 \quad \text{means} \quad (S1 \subseteq S2 \quad \text{and} \quad S2 \subseteq S1)$$

Reformulate the *same* method using this definition. ∎

Exercise 21.3 Design a class called *Factoring*. A factoring is a pair of ints. What matters about a factoring is the product of the two numbers; their ordering is irrelevant. Equip the class with one method: *prod*, which computes the product of the two ints.

Design two *same*ness methods:

1. *sameFactoring*, which determines whether one factoring is the same as some other factoring;

2. *sameProduct*, which determines whether one factoring refers to the same product as some other factoring.

Can you imagine situations in which you would use one of the methods but not the other? ∎

21.2   Equality for Inheritance

Imagine for a moment an extension of *Coffee* with a class for decaffeinated coffee in the spirit of figure 111. The class adds one private field to the two inherited ones; it records the percentage to which the coffee is decaffeinated. Its constructor calls super with two values to initialize the inherited fields and initializes the local field directly.

```
class Decaf extends Coffee {
  private int quality; // between 97 and 99

  Decaf(String origin, int price, int quality) {
    super(origin,price);
    this.quality = quality;
  }
}
```

Figure 111: The sameness for inheritance

By inheritance, the class also contains a method for comparing instances of *Decaf* with instances of *Coffee*. This method, however, does not properly compare two instances of *Decaf*. More precisely, the inherited *same* method compares two instances of *Decaf as if they were instances of Coffee*. Hence, if they differ to the degree that they have been decaffeinated, the comparison method reports that they are equal even though they are distinct instances:

```
class InhExamples {
  . . .
  Decaf decaf1 = new Decaf("Ethiopian",1200,99);
  Decaf decaf2 = new Decaf("Ethiopian",1200,98);

  boolean testD1 = check this.decaf2.same(this.decaf1)  expect false;
  boolean testD2 = check this.decaf1.same(this.decaf1)  expect true;
}
```

In this case, ProfessorJ would report one failed test, namely *testD1*. The computed result is true while the expected result is false.

Of course, when you do compare a *Coffee* with an instance of *Decaf*, the best you can do is compare them as *Coffee*s. Conversely, a comparison of a *Decaf* with a *Coffee* should work that way, too:

inside of *InhExamples* :
Coffee ethi = **new** *Coffee*(`"Ethiopian"`,1200);

boolean *testCD* = **check this**.*ethi.same*(**this**.*decaf1*) **expect** true;
boolean *testDC* = **check this**.*ethi.same*(**this**.*decaf1*) **expect** true;

Based on this analysis and the examples, you should realize that you want (at least) two methods for comparing objects in Decaf:

inside of *Decaf* :
// is **this** *Decaf* the same as *other* when viewed as *Coffee*?
public boolean *same*(*Coffee other*)

// is **this** the same *Decaf* as *other*?
public boolean *same*(*Decaf other*)

The first is the inherited method. The second is the proper method for comparing two instances of *Decaf* properly; we use the name *same* again and thus overload the method. The type comparison that selects the proper method thus decides with which method to compare coffees.

The method definition itself is straightforward:

inside of *Decaf* :
public boolean *same*(*Decaf other*) {
  return super.*same*(*other*) *&&* **this**.*quality* == *other.quality*;
}

It first invokes the **super** method, meaning the one that compares the two instances as *Coffee*s. Then it compares the two local fields to ensure that the instances are truly the same as *Decaf*s.

In general, you will have to decide what a comparison means. In any case, you will want to ensure that comparing objects is symmetric, that is, no matter in what order you compare them, you get the same result. Keep in mind, however, that your derived subclass always inherits the equality method from its superclass so this form of comparison is always legal from a type perspective. If you do not add an equality comparison for two instances of the subclass, you will get the inherited behavior—even if you do not want it.

Exercises

Exercise 21.4 Design a complete examples class for comparing instances of *Coffee* and *Decaf*. Evaluate the tests with the class definitions from figure 111 and 109. Then repeat the evaluation after adding the overloaded *same* method to *Decaf*. ∎

```
// a sale of bulk coffee
class Sale {
  Coffee c;
  int amount;

  Sale(Coffee c, int amount) {
    this.c = c;
    this.amount = amount;
  }

  // is this Sale the same as the other Sale?
  boolean same(Sale other) {
    return this.amount == other.amount
        && this.c.same(other.c) ;
  }
}
```

Figure 112: The sameness for containment

21.3 Equality for Containment and Unions

The first subsection explains the basics of extensional equality but not the whole story. It suggests that the design recipe helps a lot with the design of *same* methods, and you can probably see how *same*ness works for class containment. If class *C* contains a field whose type is that of another class *D*, the design rules apply: to compare the *D*-typed field of one object to that of another, use *same* from *D*.

Figure 112 displays a concrete example. The class *Sale* represents the sales of bulk coffee. Each sale comes with two attributes: the number of pounds sold and the coffee sold. The former is a plain int; the latter is, however, an instance of *Coffee*. Naturally, the *same* method of *Sale* uses ==

to compare the int fields and *same* from *Coffee* to compare the *Coffee*s (see the gray-shaded box).

```
                    +-----------------------+
                    | IItem                 |
                    +-----------------------+
                    +-----------------------+
                    | boolean same(IItem x) |
                    +-----------------------+
                               |
                             /   \
                             -----
                               |
                 ------------------------------
                 |                            |
   +-----------------------+    +-----------------------+
   | Coffee                |    | Tea                   |
   +-----------------------+    +-----------------------+
   | String origin         |    | String kind           |
   | int price             |    | int price             |
   +-----------------------+    +-----------------------+
   +-----------------------+    +-----------------------+
```
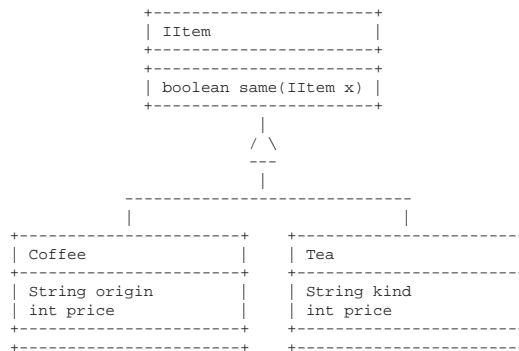
Figure 113: Coffee and tea

In the language you know, the design of an extensional equality method for unions is far more complicated than that. To make things concrete, let's consider the specific example of all items on sale in figure 113. The interface *IItem* represents the union of *Coffee* and *Tea*, i.e., it represents all the items that tea-and-coffee houses sell. The *Coffee* class is as above; the *Tea* class keeps track of the *kind* (a *String*) of the tea and its price.

Adding a method to a union requires the addition of a method header and purpose statement to the interface:

> inside of *IItem* :
> // is **this** the same *IItem* as *other*?
> boolean *same*(*IItem x*)

To make up behavioral examples, we first need to agree on the nature of extensional equality in a union. The simplest kind of extensional equality has these two characteristics:

1. An object of type *IItem* can only be the same as an instance of the same class. In other words, the method compares *Coffee*s with *Coffee*s and *Tea*s with *Tea*s but not *Coffee*s with *Tea*s (or vice versa). For the latter it reports false.

2. When *same* compares an instance of *Tea* with another *Tea*, it compares them field by field, just like *same* in *Coffee*.

These two points explain the expected results of the following examples:

```
class ItemsExamples {
  IItem ethi = new Coffee("Ethiopian",1200);
  IItem blck = new Tea("Black",1200);


  boolean test1 = check this.ethi.same(this.ethi) expect true;
  boolean test2 = check this.ethi.same(this.blck) expect false;
  boolean test3 = check this.blck.same(this.blck) expect true;
}
```

You can easily make up others; the idea should be clear.

The true nature of this design problem begins to show when you write down the templates :

inside of *Tea*:
```
boolean same(IItem other) {
  ... this.kind ... this.price ...
  ... other.mmm() ...
}
```

inside of *Coffee*:
```
boolean same(IItem other) {
  ... this.kind ... this.price ...
  ... other.mmm() ...
}
```

Both classes contain two fields each, so the templates contain the two appropriate selector expressions. Since the arguments have a non-basic type, the templates also contain a reminder that the methods can call an auxiliary method on the argument.

Unfortunately, *other* itself has type *IItem*, which is an obstacle to making progress. On one hand, we don't even know whether it makes sense to compare **this** instance of *Coffee*, for example, with *other*; after all, *other* could just be a *Tea*. On the other hand, even if we could validate that *other* is an instance of *Coffee*, it is impossible to access its fields because the *IItem* type doesn't provide any methods that access the fields of the implementing classes. Indeed, thus far it only allows a method call to *same* or *mmm*, i.e., an auxiliary method.

What we really need then is two methods: one that checks whether an *IItem* is an instance of *Tea* and another one that converts an *IItem* to a *Tea*, if it is an instance of *Tea*. After the conversion, the *same* method can use the familiar ways of comparing two *Tea*s. Analogously, we need such methods for *Coffee*, too:

```
// some grocery items
interface IItem {
  // is this the same IItem as other?
  boolean same(IItem x);
  // is this Coffee?
  boolean isCoffee();
  // convert this to Coffee (if feasible)
  Coffee toCoffee();
  // is this Tea?
  boolean isTea();
  // convert this to Tea (if feasible)
  Tea toTea();
}
```

This complete interface (and wish list) shows the need for four auxiliary methods overall but it also suggests immediately how to compare two *IItem*s:

```
inside of Tea :
boolean same(IItem other) {
  return other.isTea() && other.toTea().same(this);
}
```

The method invokes *isTea* on *other* to determine whether it is an instance of *Tea*. If it isn't, someone attempted to compare *Tea* and *Coffee*; otherwise, the method uses *toTea* to convert *other* and then uses an auxiliary method *same* for comparing just two *Tea*s:

```
inside of Tea :
// is this the same Tea as other?
boolean same(Tea other)
```

The overloading of *same* is easy to resolve here because the result type of *toTea* is *Tea* and so is the type of **this**; at the same time, reusing the name conveys the intention behind the method and points to its connection with the original version. Designing this auxiliary method is also easy; it is just like any other method for comparing two instances of a plain class.

Figure 114 contains the full definitions of *IItem*, *Tea*, and *Coffee*. The auxiliary methods for comparing *Tea* with *Tea* and *Coffee* with *Coffee* are private. While hiding the method isn't necessary, the method isn't advertised via the *IItem* interface and is thus not useful for the union as a whole. Since the rest of the program deals with the union and not its individual cases, hiding it is the correct solution here.

```
class Coffee implements IItem {            class Tea implements IItem {
  private String origin;                     private String kind;
  private int price;                         private int price;

  Coffee(String origin, int price) {         Tea(String kind, int price) {
     this.origin = origin;                      this.kind = kind;
     this.price = price;                        this.price = price;
  }                                          }

  public boolean isCoffee() {                public boolean isTea() {
    return true;                               return true;
  }                                          }

  public boolean isTea() {                   public boolean isCoffee() {
    return false;                              return false;
  }                                          }

  public Coffee toCoffee() {                 public Tea toTea() {
     return this;                              return this;
  }                                          }

  public Tea toTea() {                       public Coffee toCoffee() {
    return Util.error("not a tea");            return Util.error("not a coffee");
  }                                          }

  public boolean same(IItem other) {         public boolean same(IItem other) {
    return (other.isCoffee())                  return other.isTea()
         && other.toCoffee().same(this);            && other.toTea().same(this);
  }                                          }

  // is this the same Coffee as other?      // is this the same Tea as other?
  private boolean same(Coffee other) {       private boolean same(Tea other) {
    return                                     return
      this.origin.equals(other.origin)           this.kind.equals(other.kind)
      && this.price == other.price;              && this.price == other.price;
  }                                          }
}                                          }
```

Figure 114: The sameness of unions

Exercises

Exercise 21.5 Our chosen examples for the classes in figure 114 do not cover all possible cases. Design additional test cases to do so. ▪

Exercise 21.6 Add *Chocolate* as a third variant to the *IItem* union. The class should keep track of two attributes: *sweetness* (a *String*) and *price* (an int). What does it take to define its *same*ness method? ▪

Exercise 21.7 Abstract over the commonalities of *Coffee* and *Tea*. Don't forget to use the tests from exercise 21.5. When you have completed this step, repeat exercise 21.8. How does the abstraction facilitate the addition of new variants? How is it still painful? ▪

Exercise 21.8 Pick any self-referential datatype from chapter I and design a *same* method for it. Remember the design recipe. ▪

Exercise 21.9 On occasion, a union representation employs a string per variant that uniquely identifies the class:

> **interface** *IMeasurement* { }
>
> **class** *Meter* **implements** *IMeasurement* {
>   **private** *String name* = "meter";
>   int *x*;
>
>   *Meter*(int *x*) {
>   **this**.*x* = *x*;
>   }
> }

Add a variant to the union that measures distances in feet just like *Meter* measures them in meters. Then add a method for comparing *IMeasurement*s that ignores the actual distance; in other words, it compares only the kind of measurement not the value.

Similarly, a union may come with a method that converts an object to a string. In that case, the comparison method can employ this method and then compare the results. Equip *IMeasurement* with a *toString* method whose purpose is to render any measurement as a string. Then add an equality method to the union that compares complete measurements. ▪

Todo

discrepancies between book and ProfessorJ: can we get overridden method signatures with subtypes? is assignment still in Intermediate?
   discuss overloading in this intermezzo

Intermezzo 3: Abstract Classes, Privacy

Abstract Classes and Class Extensions

1. abstract classes

   > **abstract class** *ClassName* [**implements** *InterfaceName*] {
   >   . . .
   > }

2. abstract methods in abstract classes:

   > **abstract** *Type methodName*(*Type parameterName*, . . .);

3. subclassing

   > **class** *ClassName* **extends** *ClassName* {
   >   . . .
   > }

4. a super constructor call:

   > **super**(*Expression,*. . .)

5. a super call:

   > **super**.*methodName*(*Expression,*. . .)

   > [**abstract**] **class** *ClassName*
   >   [**extends** *ClassName*]
   >   [**implements** *InterfaceName*] {
   >   . . .
   > }

   extends, super, overriding
   purpose: the syntax of intermediate student that is used in chapter III
   type checking in this context

Privacy for Methods

abstract methods in abstract classes:

abstract [**private** | **public**] *Type methodName*(*Type parameterName, . . .*);

methods in classes:

[**private** | **public**] *Type methodName*(*Type parameterName, . . .*) {
 Statement
}

Overloading Constructors and Methods

PICTURE: should be on even page, and even pages must be on the left

Purpose and Background

The purpose of this chapter is to introduce the idea of assignment statements, to learn when to use them, and to provide a version of the design recipe that accommodates assignments.

After studying this chapter, students should be able to design classes with two-way connections (through assignment statements) and to design imperative methods analogous to those in chapter II.

The development is similar to HtDP chapters VII and VIII, which will disappear from HtDP/2e. The chapter does not assume that the student understands these chapters.

TODO

– add stacks and queues to the exercises (deck of cards looks like an ideal place to do so)

 – the following works:

```
class Circular {
  Circular one = this;
}
```

and creates a circular Object; I need to explain this

IV   Circular Objects, Imperative Methods

When you go to a book store and ask a sales clerk to look up a book whose author you remember but whose title you have forgotten, the clerk goes to a computer, types in the name of the author, and retrieves the list of books that the author has written. If you remember the title of the book but not the author, the clerk enters the title of the book and retrieves the author's name. Even though it is feasible for the program to maintain two copies of all the information about books, it is much more natural to think of a data representation in which books and authors directly refer to each other in a circular manner.

So far, we haven't seen anything like that. While pairs of *data definitions* may refer to each other, their *instances* never do. That is, when an instance O of class C refers to another object P, then P cannot—directly or indirectly— refer back to O; of course, P may refer to *other* instances of C, but that is not the same as referring to O. With what you have learned, you simply cannot create such a collection of objects, as desirable as it may be. Bluntly put, you cannot express the most natural representation of the book store program.

The purpose of this chapter is to expand the expressive power of your programming language. To this end, it introduces a new mechanism for computing, specifically, the ability to change what a field represents. This is called an assignment statement or *assignment* for short. Using assignments, your methods can create pairs of objects that refer to each other. Your new powers don't stop there, however. Once your methods can change the value of a field, you also have another way of representing objects whose state changes over time. Instead, of creating a new object when things change, your methods can just change the values in fields. This second idea is the topic of the second half of this chapter.

```
+------------+              +--------------+
| Author     | <--+   +--> | Book         |
+------------+    |   |    +--------------+
| String fst |    |   |    | String title |
| String lst | +-|---+    | int price    |
| int dob    | | |        | int quantity |
| Book bk    |--+ +--------| Author ath   |
+------------+              +--------------+
```

```
// represent authors of books      // represent books
class Author {                      class Book {
  String fst;                         String title;
  String lst;                         int price;
  int dob;                            int quantity;
  Book bk;                            Author ath;

  Author(String fst, String lst,      Book(String title, int price,
         int dob, Book bk) {                int quantity, Author ath) {
    this.fst = fst;                     this.title = title;
    this.lst = lst;                     this.price = price;
    this.dob = dob;                     this.quantity = quantity;
    this.bk = bk;                       this.ath = ath;
  }                                   }
}                                   }
```

Figure 115: The author of a book, the book of an author (version 1)

23   Circular Data

Let's turn the problem of looking up books and authors into a programming scenario, based on the bookstore problem 11.1 (page 113):

> ...   Design a program that assists bookstore employees. For each book, the program keeps a record that includes information about its author, its title, its price, and its publication year. In turn, the data representation for the author includes the author's first name, last name, year of birth, and the book written by this author. ...

The problem simplifies the real world a lot. Many authors write more than one book. Similarly, many books have more than a single author; this book has six, some have many more than that. Eventually we will have to associate an author with a list of books and a book with a list of authors.

Figure 115 displays a first-guess data representation, including a class diagram and two class definitions. The class diagram differs from every-

thing we have seen so far in that it contains two classes, mutually connected via containment arrows. Naturally, each of the two classes has four fields. The *Author* class comes with four fields: a first name, a last name, an integer that records the year of birth, and the book that the author wrote; the fields in *Book* are for the title of the book, the sales price, the number of books in the store, and the author of the book.

The next step in our design recipe calls for an exploration of examples. Here, doing so proves again the value of a systematic approach. Consider the following concrete example, a classic book in our discipline:

> Donald E. Knuth. *The Art of Computer Programming*. Volume 1. Addison Wesley, Reading, Massachusetts. 1968.

If we were to use the data representation of figure 115 and start with the author, we easily get this far:

```
new Author("Donald",
           "Knuth",
           1938,
           new Book("The Art of Computer Programming (volume 1)",
                    100,
                    2,
                    ??? ))
```

Now the *???* should be replaced with the *Author*, but of course, that means we would be starting all over again and there would obviously be no end to this process. If we start with the book, we don't get any further either:

```
new Book("The Art of Computer Programming",
         100,
         2,
         new Author("Donald",
                    "Knuth",
                    1938,
                    ??? ))
```

In this case, the *???* should be replaced with a representation of the book and that leads to an infinite process, too. At first glance, we are stuck.

Fortunately, we are not confronted with a chicken-and-egg problem; before authors are born, they can't write books. This suggests that an *Author* should be created first and, when the book is created later, the program should connect the instance of the given *Author* with the instance of the *Book*. Figure 116 displays new versions of *Author* and *Book* that work in

```
// represent authors of books
class Author {
  String fst;
  String lst;
  int dob;
  Book bk = null ;

  Author(String fst, String lst,
         int dob, Book bk) {
    this.fst = fst;
    this.lst = lst;
    this.dob = dob;
  }
}
```

```
// represent books
class Book {
  String title;
  int price;
  int quantity;
  Author ath;

  Book(String title, int price,
       int quantity, Author ath) {
    this.title = title;
    this.price = price;
    this.quantity = quantity;
    this.ath = ath;

    this.ath.bk = this ;
  }
}
```

Figure 116: The author of a book, the book of an author (version 2)

the suggested manner. A comparison with figure 115 shows two distinct differences (gray-shaded):

1. The *bk* field in *Author* initially stands for *null*, an object that we haven't seen yet. In some way, *null* is unlike any other object. Its most distinguishing attribute is that it has all class and interface types. Hence, any variable with such a type can stand for *null*. Think of *null* as a wildcard value for now.

2. More surprisingly, the constructor for *Book* consists of five "equations" even though the class contains only four fields. The last one is the new one. Its left-hand side refers to the *bk* field in the given *ath*; the right-hand side is **this**, i.e., the instance of *Book* that is just being created.

   This situation is unusual because it is the very first time two "equations" in one program—the gray-shaded ones— refer to the *same* field in the same object. The implication is that *equations in Java aren't really mathematical equations.* They are instructions to evaluate the right-hand side and to change the meaning of the field (variable) on the left-hand side: from now on the field stands for the value from the right-hand side—until the next such instruction (with the same field

on the left side) is evaluated. Computer scientists and programmers refer to these "equations" as ASSIGNMENT STATEMENTS or ASSIGN-MENTS, for short.

Concretely: the meaning of **this**.*ath.bk* = **this** in the constructor of *Book* is to *change* the value that *bk* represents. Until this assignment statement is evaluated, *bk* represents *null*; afterwards, it stands for **this** book.
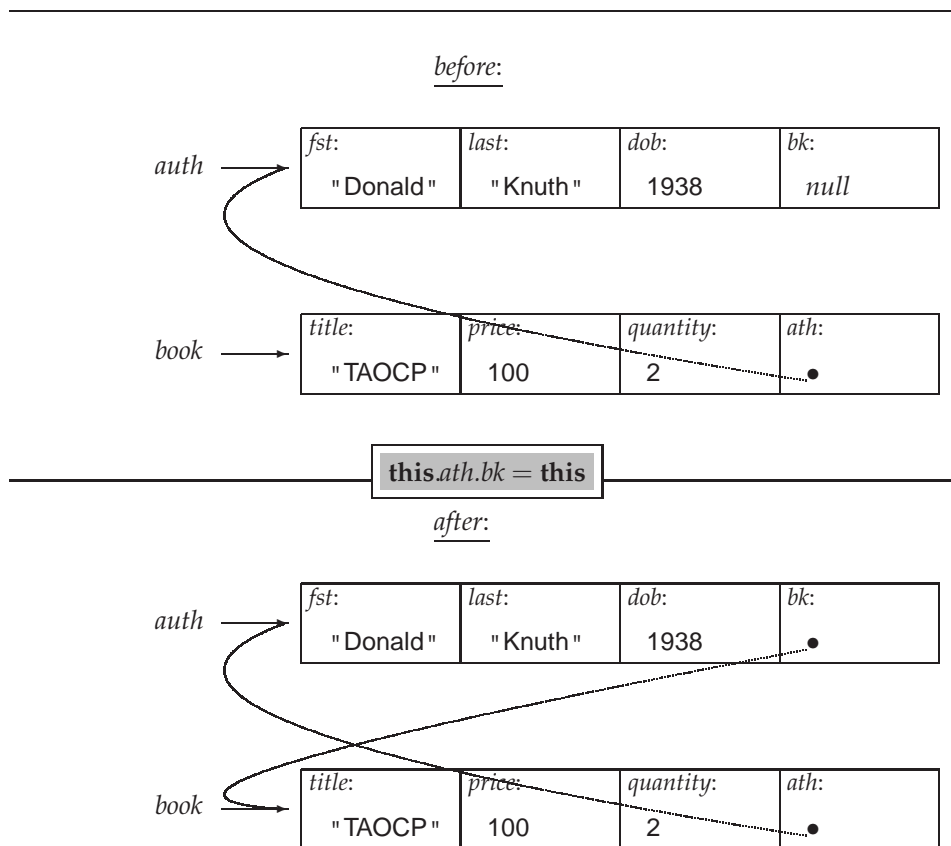
Figure 117: An object-box diagram

Creating instances of these revised classes looks just like before. The constructor of the class is called with as many values as needed, and its

result is an instance. For example, to represent the computer science classic mentioned above, you write:

Author auth = **new** *Author*(`"Donald"`,`"Knuth"`,1938);

Book book = **new** *Book*(`"TAOCP"`,100,2,*auth*);

The difference is that the evaluation of the second line affects what *auth* represents. Before the second line is evaluated, *auth*'s *bk* stands for *null*; *afterwards* it refers to *book*.

Figure 117 illustrates with the box diagrams from *How to Design Programs* what happens during the construction of the *book* object. Before the gray-shaded assignment statement is evaluated, two boxes have come into existence: the one for *auth*, with a *null* in the *bk* field, and the one for *book*, which contains *auth* in the *ath* field. The top half of the picture shows the two boxes and their four compartments. Instead of placing the box for *auth* into the compartment labeled *ath*, an arrow from the compartment to the *auth* box says that *auth* is in this compartment. The bottom half of the picture shows the boxes after the assignment has been evaluated. As you can see, the assignment places the box for *book* into the *bk* compartment of the *auth* box. Again, the figure uses an arrow from the latter to the *book* box to indicate this relationship. Indeed, now that there are arrows from the boxes to each other, it has become impossible to draw nested box diagrams.

For this reason, a Java implementation cannot easily present *auth* and *book* to programmers. When you request that ProfessorJ display *auth*, it shows the following:

```
Author(
   fst = "Donald",
   lst = "Knuth",
   dob = 1938,
   bk = Book(
         title = "TAOCP",
         price = 100,
         quantity = 2,
         ath = Author))
```

meaning that *auth* contains a *Book bk* and, in principle, the instance of *Book* contains *auth*. The latter, however, is just marked as *Author* in the *ath* field.

In general, a constructor can call other methods, especially those in other classes, to change fields elsewhere. Ideally these calls take place after the fields have been set to the given values; that is, they are located at the

```
// represent authors of books
class Author {
  String fst;
  String lst;
  int dob;
  Book bk = null;

  Author(String fst, String lst,
         int dob, Book bk) {
    this.fst = fst;
    this.lst = lst;
    this.dob = dob;
  }

  void addBook(Book bk) {
    this.bk = bk;
    return ;
  }
}
```

```
// represent books
class Book {
  String title;
  int price;
  int quantity;
  Author ath;

  Book(String title, int price,
       int quantity, Author ath) {
    this.title = title;
    this.price = price;
    this.quantity = quantity;
    this.ath = ath;

    this.ath.addBook(this) ;
  }
}
```

Figure 118: The author of a book, the book of an author (version 3)

bottom of the constructor. These calls typically involve **this**, the newly created and previously unavailable object. Other objects may have to know the new object (**this**) and informing them about it during the construction of **this** is the most opportune time.

Figure 118 demonstrates this point with a modification of the running example. The revised constructor ends in a call to the method *addBook* of *ath*, the given author. This new method has a novel return type: void. This type tells any future reader that the method's purpose is to change the field values of the object and nothing else. To signal the success of the operation, such methods return a single value, which has no other significance and is therefore invisible.

The body of *addBook* also has a novel shape for methods, though it looks almost like a simple constructor. Like a constructor, it consists of an assignment statement separated from the rest via a semicolon (;). Unlike a constructor, the method body ends with a **return** ; statement; it reminds the reader again that the method produces the single, invisible void value and

that its computational purpose is to change the object.[45]

The use of such methods is preferable to direct assignments such as in figure 116 when there is more to do than just set a field. For example, we may not want the *addBookk* method to change the *bk* field unless this field stands for *null*:

```
void addBook(Book bk) {
  if (this .bk == null) {
    this.bk = bk;
    return ;
  }
  else
    Util.error("adding a second book");
}
```

The **if** statement checks this condition with (**this** *.bk == null*), which asks whether *bk* in **this** object is equal to *null*. If so, it changes what the field represents and returns void; if not; it signals an error.

```
                                        +----------------+
                                        | Author         |<--+
                                        +----------------+   |
                                        | String first   |   |
                                        | String last    |   |
  +----------+                          | int dob        |   |
  | IBooks   |<-------------------------| IBooks books   |   |
  +----------+<----------------+        +----------------+   |
    |  |                       |                             |
   / \                         |                             |
   ---                         |                             |
    |                          |                             |
  +--+------------+            |                             |
  |              |             |        +----------------+   |
  +---------+ +--------------+ | +---->| Book           |----+
  | MTBooks | | ConsBooks    | | |      +----------------+
  +---------+ +--------------+ | |      | String title   |
              |              | | |      | int price      |
              | Book fst     |-----+    | int quantity   |
              | IBooks rst   |-+        | Author author  |
              +--------------+          +----------------+
```
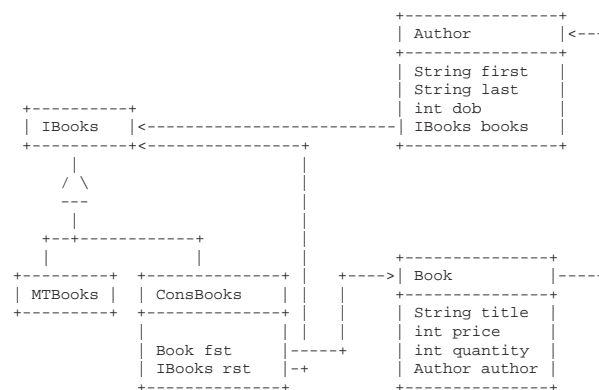
Figure 119: Classes for books and authors

Signaling an error in *addBook* says that an author can write only one book. It should remind you of the discussion following the initial problem statement. To accommodate authors who write many books, let's look at a data representation that associates an author with a list of books instead of a single book: see figure 119. In this new data representation, the *Author* class contains a field that contains lists of books, dubbed *IBooks* in the figure. This

---

[45]Many people omit such "empty" return statements; we believe it helps readers and therefore use it.

interface is the type of all lists of books. Thus, *Author* doesn't directly refer to *Book* but indirectly, through *ConsBook* and its first field. Of course, *Book* still refers to *Author*. Otherwise this class diagram doesn't differ much from those we have seen plus the new element of a cycle of containment arrows.

Translating the class diagram of figure 119 into class definitions poses the same problem as translating the class diagram at the top of figure 115 into code. Like then, let's agree that the constructor for *Author* doesn't consume a list of books. Instead, the *books* field is initially set to **new** *MTBooks*(), the empty list of books. As books are added to the collection of data, the list must be updated.

```
// authors of books
class Author {
  String fst;
  String lst;
  int dob;
  IBooks books = new MTBooks();

  Author(String fst, String lst,
          int dob) {
    this.fst = fst;
    this.lst = lst;
    this.dob = dob;
  }

  void addBook(Book bk) {
    this.books = new ConsBooks(bk, this.books);
    return ;
  }
}
```

```
// lists of books
interface IBooks { }

class MTBooks
  implements IBooks {
    MTBooks() {}
}

class ConsBooks
  implements IBooks {
    Book fst;
    IBooks rst;

    ConsBooks(Book fst,
              IBooks rst) {
      this.fst = fst;
      this.rst = rst;
    }
}
```

Figure 120: An author of many books, the many books of an author

We can implement the rest of the changes to the data definition with the usual interfaces and classes for a list of books and a small change to the *addBook* method in *Author* (see gray-shaded method in figure 120). The revised method still receives a book, which it is to add to the list of books for the author. To accomplish this, it forms a new, extended list from the given book *bk* and the current list of *books*:

new *ConsBooks*(*bk*, **this**.*books*)

Afterwards, it assigns this value to the *books* field in *Author*. In other words, from now on the *books* field stands for the newly created list. The rest of *Book* class remains the same.

Exercises

Exercise 23.1 Explain why the definition of *Book* remained the same when we changed *Author* to be associated with an entire list of *Book*s. ∎

Exercise 23.2 Create data representations for the following list of classics in computer science:

1. Donald E. Knuth. *The Art of Computer Programming*. Volume 1. Addison Wesley, Reading, Massachusetts. 1968.

2. Donald E. Knuth. *The Art of Computer Programming*. Volume 2. Addison Wesley, Reading, Massachusetts. 1969.

3. Donald E. Knuth. *The Art of Computer Programming*. Volume 3. Addison Wesley, Reading, Massachusetts. 1970.

Draw a box diagram like the one in figure 117 for this example.
   Or, do it for these books:

1. Henk Barendregt. *The Lambda Calculus*. North Holland, Amsterdam, The Netherlands. 1981.

2. Daniel P. Friedman. *The Little LISPer*. SRA Press, Chicago, Illinois. 1974

3. Guy L. Steele Jr. *Common LISP, the Language*. Digital Press, Bedford, Massachusetts. 1990.

Computer scientists should know (about) them, too. ∎

Exercise 23.3 Modify the data representation for authors and books in figure 120 so that a new book is added to the end of an author's list. ∎

Exercise 23.4 Encapsulate the state of *Book* and *Author* for the code in figures 116 and 118. ∎

23.1   Designing Classes for Circular Objects, Constructors

The examples in this section suggest that designing classes whose instances refer to each other in a circular manner follows mostly the same process as the design of classes in general:

1. Read the problem statement and determine how many classes of information there are and how they are related. Make up examples of information and how people will use it. This should help you determine whether there is a need to access a piece of information I from some piece of information J and vice versa.

2. Develop the interface and class diagram.

   Doing so gives you a second chance to discover the existence of object cycles. Specifically, inspect the class diagram for cycles of containment arrows. Sometimes the cycle is direct and uninterrupted, as in figure 115. In this case, you are almost guaranteed to have pieces of data that are in a circular relationship. What you need to confirm is an intention that each instance points to some other object that points back. Other times, such as in figure 119, there is no direct cycle; instead you must traverse an inheritance arrow in the reverse direction to construct a cycle. In those cases, you may or may not have to construct circular collections of objects. Only examples from the first step can help here.

3. Translate the class diagram into class definitions naively. Don't forget to add a purpose statement to each class as usual.

4. Now modify your class definitions so that they can accommodate circular relationships among objects, if needed.

   First, try to translate the examples of circular information into data. If you can translate all the information examples into data (using the regular constructors), you don't need circularity. If there are some that you can't translate, you have confirmed the need for circularity.

   Second, determine the class *C* whose objects should come into existence first. In our running example at the beginning of the section, this class was *Author*. Then determine the field that would create a circular relationship; call it *cf* and let its type be *CT*.

   Third, initialize the *cf* field with an object of type *CT* that contains no fields of the type *C* (or its interface). In our running example, we used

the empty list of books for this purpose. Sometimes we need to use the value of last resort: *null*.

Fourth, define an *add* method that assigns new values to *cf*. For now, use the examples from this section as templates; they either replace the value of *cf* with a given value or create a list of values. You will soon learn how to design such methods in general.

Last, modify the constructors of the classes that implement *CT*. They must call the *add* method with **this** so that the circular references can be established.

5. Lastly, translate the circular examples of information into data, using just the constructors in the enforced order. Check whether the circular references exist by looking at it.

Exercises

Exercise 23.5 Design a data representation for a hospital's doctors and patients. A patient's record contains the first and last name; the patient's gender; the blood type; and the assigned primary physician. A doctor's record should specify a first and last name, an emergency phone number (int), and the assigned patients. ∎

Exercise 23.6 Design a data representation for your registrar's office. Information about a course includes a department name (string), a course number, an instructor, and an enrollment, which you should represent with a list of students. For a student, the registrar keeps track of the first and last name and the list of courses for which the student has signed up. For an instructor, the registrar also keeps track of the first and last name as well as a list of currently assigned courses. ∎

Exercise 23.7 Many cities deploy information kiosks in subway stations to help tourists choose the correct train. At any station on a subway line, a tourist can enter the name of some destination; the kiosk responds with directions on how to get there from here.

The goal of this exercise is to design and explore two different data representations of a straight-line subway line. One way to represent a line for an information kiosk is as a list of train station. Each station comes with two lists: the list of stops (names) from this one to one end of the line and the stops from this one to the other end of the line. Another way is to think

of a subway stop as a name combined with two other stops: the next stop in each direction. This second way is close to the physical arrangement; it is also an example of a *doubly linked list*. Hint: Consider designing the method *connect* for this second representation of a subway station. The purpose of this method is to connect this station to its two neighbors. The end stations don't have neighbors.
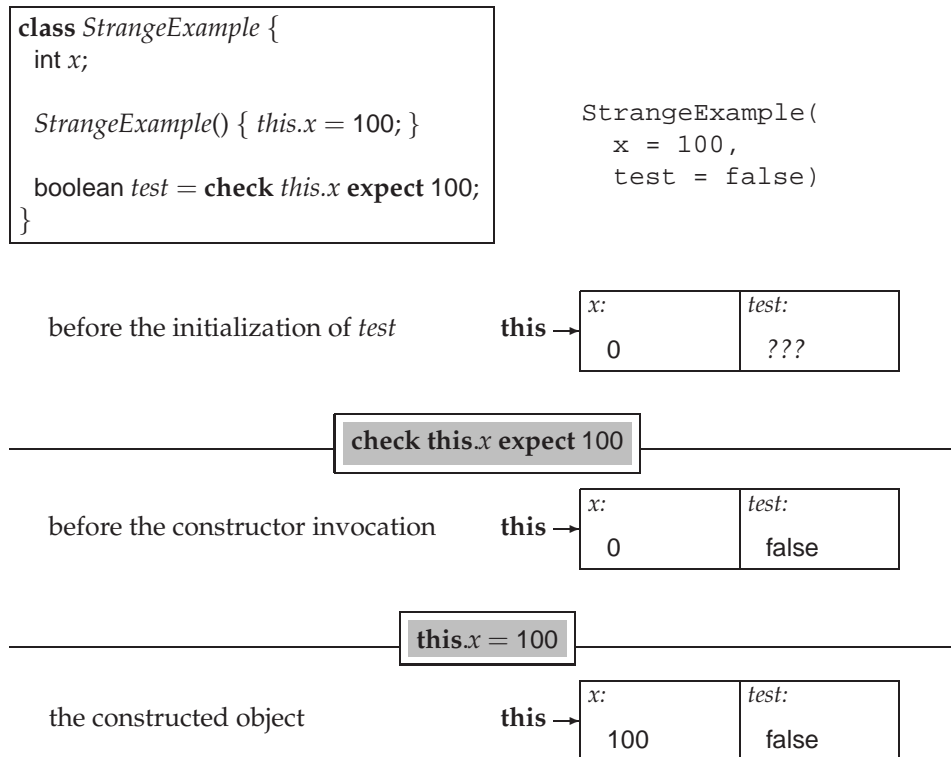
Figure 121: The behavior of constructors

Design both representations. Represent the Boston red line, which consists of the following stops: JFK, Andrew, Broadway, South Station, Downtown Crossing, Park Street, MGH, Kendall, Central, Harvard, Porter, Davis, and Alewife. ∎

23.2   The True Nature of Constructors

A full understanding of circular objects also demands a full explanation of the true nature of constructors. Until now, we have acted as if a constructor creates instances of a class. In reality, the **new** operation creates the objects; the constructor assigns values to fields and performs other actions. This just begs the question then what values the fields contain *before* the constructor is invoked.

In Java and ProfessorJ (Advanced), each field is initialized based on its type. Specifically, each int field is initially set to 0; each boolean field to false; each double field to 0.0; and each field with a class or interface type— including *String*—is set to *null*, the wildcard value.

Not surprisingly, the initialization process and its assignments can produce strange results. For one example, take a look at figure 121. Its top-left side displays a class with two fields whose constructor initializes the int field to 100 and whose *test* field compares the int field with 100. The top-right side shows how ProfessorJ displays an instance of this class, an object with two fields: *x*, which equals 100, and *test*, which equals false. Considering that *test* is the result of comparing *x* and 100 and that *x* is indeed 100, the false value in *test* demands some explanation.

The bottom half of figure 121 explains how the object is created. First, the fields are initialized, based on their type; *x* is set to 0, and *test* is false. Second, the right-hand side of the *test* field declaration is evaluated. Since *x* is 0 at this point, comparing **this**.*x* with 100 produces false, which remains the of *test*. Finally, the constructor is invoked. It contains a single assignment to **this**.*x*, which changes the value to 100. Thus, the *x* field ends up as 100 even though *test* is set to false. Naturally, this behavior has serious consequences for testing, and we encourage you to read the notes on testing before you continue.

Testing in
Advanced

23.3   Circularity and Encapsulation

If you go back to chapters I and II, you will find cycles in all self-referential class diagrams. There is (almost) no other way to create a piece of data without pre-determined limit on its size. The left side of figure 122 displays the class diagram for a list of ints; the right side shows the interface and class definitions for this list.

The cycle in the diagram suggests that creating a circular list of ints is possible. And indeed, now that we know about assignment statements, it is possible and relatively straight forward. Take a look at this class:

```
        +-------+
        | IList |<----------+
        +-------+           |
            |               |
           / \              |
           ---              |
            |               |
      +-----+------+        |
      |            |        |
  +------+    +-----------+ |
  |  MT  |    | ConsLog  | ||
  +------+    +-----------+ |
              | int fst   | ||
              | IList rst |-+
              +-----------+
```
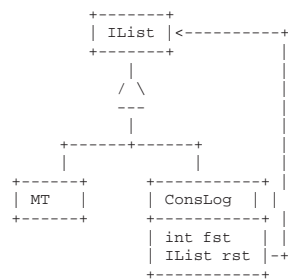
interface *IList* {}

class *MT* **implements** *IList* {}

class *Cons* **implements** *IList* {
  int *fst*;
  IList rst;

  Cons(int *fst*, *IList rst*) {
    this.fst = *fst*;
    this.rst = *rst*;
  }
}

Figure 122: Creating Circular Lists

```
class Example {
  Cons alist = new Cons(1,new MT());

  Example() {
    this.alist.rst = alist;
  }
}
```

It has a single field: *alist*, which has the initial value **new** *Cons*(1,**new** *MT*()). The constructor then assigns the value of *alist* to the *rst* field of *alist*, which makes it a circular list.

    We can visualize *Example*'s *alist* and the assignment to *alist* using the box diagram for structures from *How to Design Programs*:
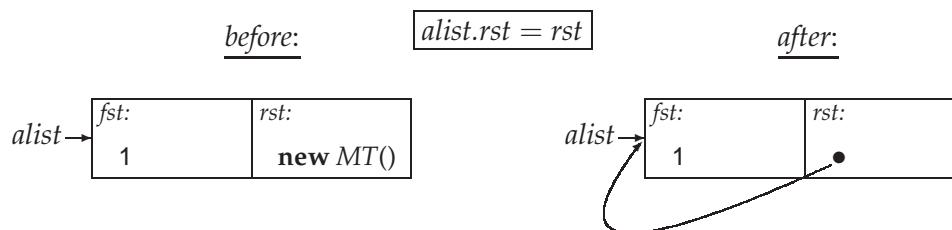
The boxes have two compartments, because *Cons* has two fields. Initially the content of the first compartment is 1; **new** *MT*() is in the second compartment. The box itself is labeled *alist*, which is its name. When the constructor assigns *alist* to *rst* of *alist*, it sticks the entire box into itself. We

can't draw such a box, because it would take us forever; but we can indicate this self-containment with an arrow that goes from the inside of the compartment back to the box, the exact same place that is called *alist*.

Exercise

Exercise 23.8 Design the method *length* for *IList*. The method counts how many ints **this** *IList* contains. After you test the method on regular examples, run it on *alist* from *Example* on the previous page:

  new *Example*().*alist.length*()

What do you observe? Why? ∎

```
class Cons implements IList {
  private int fst;
  private IList rst;

  public Cons(int fst, IList rst) {
    this.fst = fst;
    this.rst = rst;
  }
}
```

Figure 123: Privacy declarations to prevent unintended circularities

The example suggests that assignment statements not only add necessary power; they also make it easy to mess up. Until now we didn't need circular lists, and there is no obvious reason that we should be able to create them unless there is an explicit demand for it. Otherwise we can get truly unwanted behavior that is difficult to explain and, when it shows up later by accident, is even more difficult to find and eliminate.

Our solution is to use privacy declarations for all fields and for all methods as advertised at the end of the preceding part. In this case, we just need to protect the two fields in *Cons* and publicize the constructor: see figure 123. With this protection in place, it is possible to create instances of *Cons* and impossible to assign new values to these fields from the outside. Hence, it is also impossible to create circular lists.

In short: if you want circular collections of data, design them explicitly and intentionally. Use privacy declarations so others don't create circular data by chance.

23.4   Example: Family Trees

```
┌─────────────────┐   ┌─────────────────┐
│ Carl  (1926)    │   │ Bettina  (1926) │
│ Eyes:  green    │   │ Eyes:  green    │
└─────────────────┘   └─────────────────┘
```

```
┌─────────────────┐ ┌─────────────────┐ ┌─────────────────┐ ┌─────────────────┐
│ Adam  (1950)    │ │ Dave  (1955)    │ │ Eva  (1965)     │ │ Fred  (1966)    │
│ Eyes:  yellow   │ │ Eyes:  black    │ │ Eyes:  blue     │ │ Eyes:  pink     │
└─────────────────┘ └─────────────────┘ └─────────────────┘ └─────────────────┘
```

```
┌─────────────────┐
│ Gustav  (1988)  │
│ Eyes:  brown    │
└─────────────────┘
```
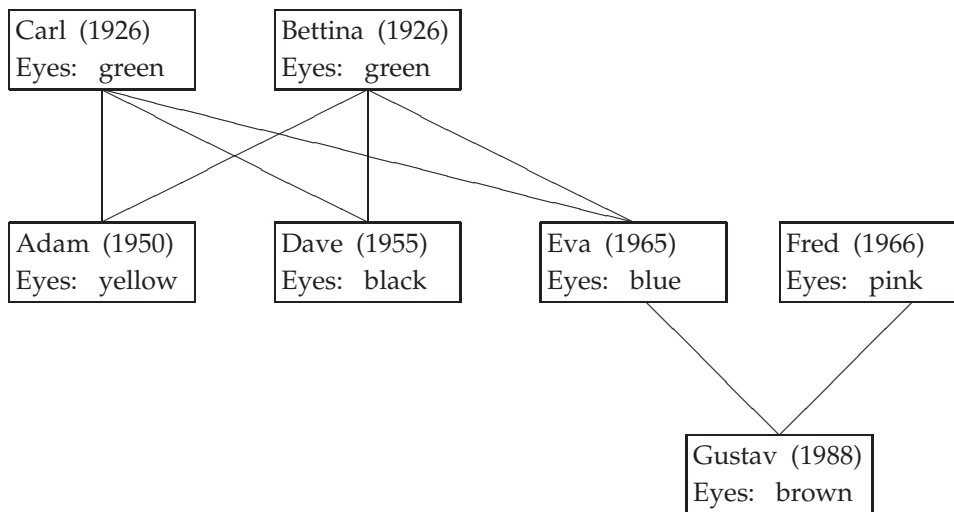
Figure 124: A sample family tree

In *How to Design Programs*, we studied both ancestor and descendant family trees. The idea was to write software that assists genealogists and others who study family trees. At the time our programs could answer such questions as "list all your ancestors" or "list all the descendants of this person" or "is there someone with blue eyes in your ancestor tree." Let's revisit this problem now:

> ... Your company has agreed to create software for a genealogy department at a research hospital. Your manager has asked you to design the data representation for family trees. To help you design this data representation, your manager has also posed two sample design problems concerning methods:
>
> 1. Locate an ancestor of **this** person with a given name.
> 2. Determine the names of the siblings of **this** person.

Finally, your manager has also provided a sample family tree:
see figure 124. . . .

Here are the two data definitions from *How to Design Programs* for an-
cestor trees and descendant trees:

1. (**define-struct** *child* (*father mother name age eye-color*))

A *Family-Tree-Node* (short: *FTN*) is either

(a) 'unknown; or

(b) (make-child *FTN FTN String Number String*)

---

2. (**define-struct** *parent* (*children name age eye-color*))

A *Parent* is a structure:

                (make-parent *LOC String Number String*)

A *list-of-children* (short: *LOC*) is either

(a) empty; or

(b) (cons *Parent LOC*)

To create the ancestor family tree of a newborn child, we call the constructor
with information about the child and representations of the mother's and
father's family tree:

    (**define** *father* . . .)
    (**define** *mother* . . .)
    (**define** *child* (make-child *father mother* "Matthew" 10 "blue"))

That is, the data for the ancestors must exist before child is created. In con-
trast, to create a descendant tree for a mother, we create a representation of
her family tree using the list of her children and other personal information:

    (**define** *loc* (list . . .))
    (**define** *mother* (make-parent *loc* "Wen" 30 "green"))

Thus, in an ancestor tree, it is easy to find all ancestors of a given person in
the tree; in a descendant tree, it is easy to find all descendants.

Even if we didn't have any experience with family tree representations,
just replacing the lines in figure 124 with arrows from children to parents

would give us the same insight. Such arrows lead to containment arrows in a class diagram, meaning a node in the family tree contains fields for parents (father, mother). This, in turn, means a method can follow these arrows only in the ancestor direction. Conversely, if we point the arrows from parents to children, the nodes contain lists of children, and the methods can easily compute facts about descendants.

```
                        +-------------+<----------------+
                        | IFamilyNode |<---------------+|
                        +-------------+                ||
                              |                        ||
                             / \                       ||
                             ---                       ||
                              |                        ||
              +----------------------------------+     ||
              |                        |         |     ||
      +------------------+     +------------------+    ||
  +->| Person           |     | UnknownPerson    |    ||
  |  +------------------+     +------------------+    ||
  |  | String name      |     | IPersons children |   ||
  |  |                  |     +------------------+    ||
  |  |                  |                             ||
  |  | FamilyNode father |--------------------------------+|
  |  | FamilyNode mother |--------------------------------+
  |  |                  |      +----------+
  |  | IPersons children |------>| IPersons |<----------------+
  |  +------------------+      +----------+                   |
  |                                 |                         |
  |                                / \                        |
  |                                ---                        |
  |                                 |                         |
  |                            +--+-----------+               |
  |                            |        |                     |
  |                     +-----------+  +-------------+        |
  |                     | MTPersons |  | ConsPersons |        |
  |                     +-----------+  +-------------+        |
  |                                    |             |        |
  +------------------------------------| Person fst  |        |
                                       | IPersons rst |-+
                                       +-------------+
```
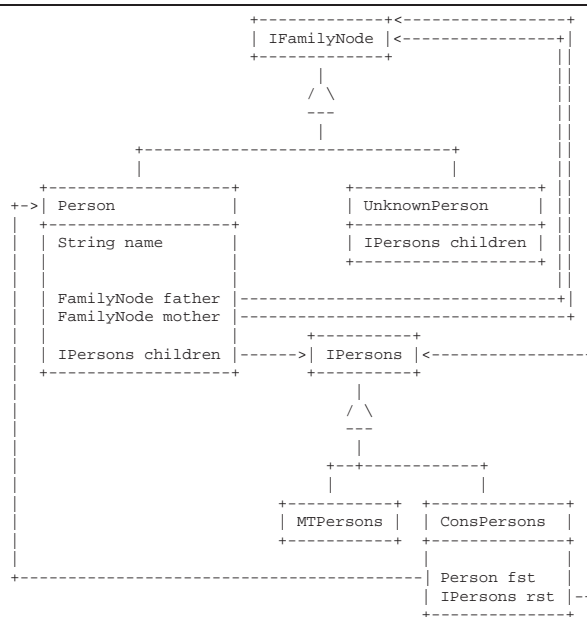
Figure 125: A family tree representation for computing siblings

What this all means is that one problem is easy and one is difficult with what we know:

- Finding all the ancestors of a given person is easy with a plain ancestor tree. The node that represents the given family member contains the family trees for *mother* and *father*, and designing a plain recursive traversal solves the problem.

- In contrast, the second problem requires more than one-directional arrows from our data representation. After all, the siblings of a node are all the immediate descendants of the parents (minus the node itself). Put differently, from each node, a sibling-computing method must be able to go up *and* down in the family tree.

Figure 125 translates this problem analysis into a class diagram. Most of the class diagram is straightforward. The upper half is the class-based equivalent of the *FTN* definition (ancestor family tree) above: the interface *IFamilyNode* is the type and the two implementing classes—*Person* and *UnknownPerson*—are the variants. The lower half is the class-based equivalent of the *Parent* and *LOC* definitions above: *Person* corresponds to *Parent* and *IPersons* corresponds to *LOC*.

```
// a family tree in which we can
// compute the siblings of a node
interface IFamilyTree {
  // add a child to this
  // node's list of children
  void addChild(Person child);
}

class Person
    implements IFamilyTree {
  String name;
  IFamilyTree mother;
  IFamilyTree father;

  IPersons children =
    new MTPersons();

  Person(String name, IFamilyTree father,
         IFamilyTree mother) {
    this.name = name;
    this.mother = mother;
    this.father = father;

    this.mother.addChild(this);
    this.father.addChild(this);
  }

  void addChild(Person child) {
    this.children =
      new ConsPerson(child, this.children);
  }
}
```

```
class Unknown
    implements IFamilyTree {
  IPersons children = new MTPersons();

  Unknown() {}

  void addChild(Person child) {
    this.children =
      new ConsPerson(child, this.children);
  }
}
```

```
// list of Persons
interface IPersons {}
```

```
class MTPersons
    implements IPersons {
  MTPersons() {}
}
```

```
class ConsPerson
    implements IPersons {
  Person fst;
  IPersons rst;
  ConsPerson(Person fst, IPersons rst) {
    this.fst = fst;
    this.rst = rst;
  }
}
```

Figure 126: Classes for a family tree representation

In the context of this diagram, the second programming problem can be expressed in concrete terms:

> ... Design the method *siblings* that, given an instance of *Person*, finds all siblings. ...

The *Person* class contains two fields for the immediate ancestor trees, both of type *IFamilyNode*. In turn, both of its implementing classes contain lists of immediate descendants. In short, it is possible to go up and down in such a family tree, and this means that a *Person* can be in a circular relationship to another *Person*.

It is now time to determine which nodes should exist first and which nodes are added later via *add* methods. If we follow our instinct, we say parents exist first, and children are added later. This is how it works in nature, and if our program implements this solution, it is natural.

Using this decision and following the new design recipe requires that we fix a good initial value for *children* in *Person*. Again, the natural choice is **new** *MTPersons*(), the empty list of children. Next we must add a method for adding new children to a family tree node. So, imagine a genealogist who has entered the data for `Bettina` and `Carl` from figure 124. The first child is `Adam`, which would be created via

> **new** *Person*(`"Adam"`,*carlFT*,*bettinaFT*)

assuming *carlFT* and *bettinaFT* are names for the family trees of `Carl` and `Bettina`. As the constructor is evaluated, it should add the new object for representing `Adam` to both parent nodes. Conversely, we need a method to add a child to the list of children:

> inside of *IFamilyTree* :
> // add a child to **this** node's list of children
> void *addChild*(*Person child*);

The actual method definition looks almost identical to the *addBook* in *Author*. It shows up in both *UnknownPerson* and *Person* because both can have children; for an instance of *UnknownPerson*, we just don't know the name or the parents.

Last but not least we must decide where to call *addChild*. Since a child is represented as an instance of *Person*, the only possible place is within the constructor for *Person*, i.e., after the constructor has initialized the *mother* and *father* fields, it calls the *addChild* method for both connecting the parents to their children.

Figure 126 displays all interface and class definitions. The left column is all about the ancestor tree; the right column is the representation of the

list of children. The last step in the design recipe is to ensure that we can represent information examples properly, including the circular ones. We leave this to an exercise.

Exercises

Exercise 23.9 Use the class definitions in figure 126 to represent the information in figure 124 as data. Ensure that you can navigate from one sibling to another via the existing methods and fields. Draw a box diagram for the resulting objects. ∎

Exercise 23.10 Create an instance of *Person* whose parent fields point to itself. Add privacy declarations to the classes in figure 126 so that the state of all objects is encapsulated and hidden from the rest of the program. Can these declarations prevent self-parenting instances of *Person*? ∎

Exercise 23.11 Abstract over the common patterns (see the *Person* and *Unknown* classes) in figure 126. ∎

Exercise 23.12 Design a data representation for a file system like the one found on your computer. Make sure you can implement methods for listing all files and folders in a folder; for changing the focus to a folder that is inside the current folder; and for changing the focus to the parent of the current folder. Which of the potential design tasks, if any, require you to design circular relationships into the data representation? ∎

Exercise 23.13 Design a data representation for a representing a river system. You must be able to go back and forth along the river, including from the confluence where a tributary joins to both of the preceding points. See section 5.2 for a simple representation of river systems. ∎

24   Methods on Circular Data

Now that we can create circular (collections of) objects, we must learn to design methods that process them. Let's return to the bookstore problem for this purpose:

> . . . The "bookstore" program assists bookstore employees with the task of finding books and authors. . . . Assume that a bookstore maintains a list of authors and their books as in figure 127.

```
+----------+
| IAuthors |<-----------------+                +-----------+          +--------------+
+----------+                  |          +--->| Author    |<---+   +-->| Book         |
     |                        |          |    +-----------+    |   |   +--------------+
    / \                       |          |    | String fst |    |   |   | String title |
    ---                       |          |    | String lst |  +--/2/-+  | int price    |
     |                        |          |    | int dob    |  | |      | int quantity |
  +--+-----------+            |          |    | Book bk    |--+ +--/1/---| Author ath   |
  |              |            |          |    +-----------+            +--------------+
+-----------+  +--------------+ |        |
| MTAuthors |  | ConsAuthors  | |        |
+-----------+  +--------------+ |        |
               | Author fst   |-|----+
               | IAuthors rst |-+
               +--------------+
```
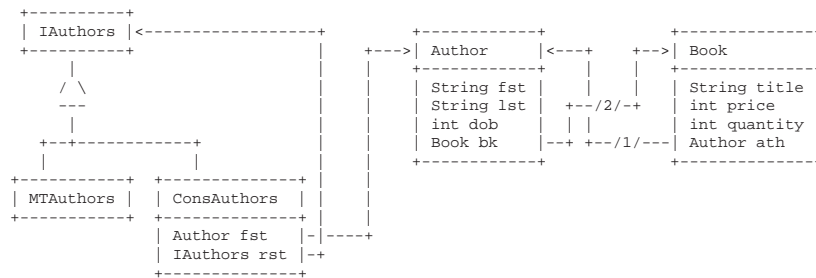
Figure 127: A simplistic bookstore: the class diagram

> Design the method *findTitle*, which given an author's last name, finds the title of the author's book. . . .

The extension is a method design problem that we encountered many times in the first three chapters of the book. Indeed, if it weren't for the mutual references between *Book* and *Author*—labeled with 1 and 2 in figure 127—the problem would be straightforward.

To understand the problem with processing circular references, let's follow the regular design recipe until we get stuck. The problem statement calls for the addition of a single method signature to the *IAuthors* interface:

inside of *IAuthors* :
// retrieve the title of *last*'s book from **this** author list; " " if none
String lookupTitle(*String last*);

If the list contains a record for Knuth's *The Art of Computer Programming* (volume 1), then

check *authorList.lookupBookByAuthor*("Knuth")
expect "The Art of Computer Programming (volume 1)"

The inheritance arrow between *IAuthors* and its implementing classes shows us where the templates go and what they look like:

inside of *MtAuthors* :   inside of *ConsAuthors* :
String lookupTitle(*String last*) {   *String lookupTitle*(*String last*) {
  . . .   . . . **this**.*fst.lookupTitle*(*last*) . . .
}   . . . **this**.*rst.lookupTitle*(*last*) . . .
                                             }

In turn, the containment arrow from *ConsAuthors* to *Author* suggests the addition of a *lookupTitle* method to that class:

inside of *Author* :
String lookupTitle(*String last*) {
  … **this**.*fst* … // *String*
  … **this**.*lst* … // *String*
  … **this**.*dob* … // int
  … **this**.*bk.lookupTitle*(*last*) …
}

Because *Book* contains a reference to he *Author*, we get a link back to *Author* if we now proceed naively:

inside of *Book* :
String lookupTitle(*String last*) {
  … **this**.*title* … // *String*
  … **this**.*price* … // int
  … **this**.*quantity* … // int
  …   **this**.*ath.lookupTitleAgain*(*last*)
}

After all, there is a containment arrow from *Book* to *Author*, labeled with 1 in figure 127.

Then again, a moment of reflection tells us that this link back from the book to its author brings us back to the very same author. Perhaps it is best to ignore this field and to proceed as if it didn't exist. In that case, the problem looks just like a lot of the problems in chapter II. As a matter of fact, the templates—without the gray-shaded method call—work out perfectly; turning them into full-fledged methods is straightforward.

Exercises

Exercise 24.1 Figure 128 displays the portion of the program that deals with a list of authors. In the bottom left corner, the figure displays a one-item wish list. Modify the class definitions of *Author* and *Book* from figure 116 or 118 to work with the fragment of figure 128. ∎

Exercise 24.2 The *lookupTitle* method returns `" "` if the list doesn't contain an instance of *Author* with the given *last* name. While this trick is possibly justifiable for the *lookupTitle* method in *IAuthors*, it does not produce well-designed programs in general.

At this point it is important to remember that the design recipe for methods does not demand a method of the same name in a contained class. It just suggests that you might need some auxiliary method (or several) in

```
interface IAuthors {
  // retrieve the title of last's book from this author list
  // return " " if no author with this last name exists
  String lookupTitle(String last);
}
```

```
class MTAuthors
    implements IAuthors {
  MTAuthors() {}

  public String lookupTitle(String l) {
    return " ";
  }
}
```

```
class ConsAuthors
    implements IAuthors {
  Author fst;
  IAuthors rst;

  ConsAuthors(Author fst, IAuthors rst) {
    this.fst = fst;
    this.rst = rst;
  }

  public String lookupTitle(String last) {
    if (this.fst.lookupTitle(last).equals(" ")) {
      return this.rst.lookupTitle(last); }
    else {
      return this.fst.lookupTitle(last); }
  }
}
```

wish list:

inside of *Author* :

// return title of the book if
// **this** author has the name *last*;
// return " " otherwise
String lookupTitle(String last)

Figure 128: Looking for a book title by author

the contained class. Use this reminder to formulate a more general template for *ConsAuthors*, *Author*, and *Book*. Then define a variant of the program that separates the task of checking for the author's last name from the task of retrieving the title of the author's book. ∎

Exercise 24.3 Add a data representation for lists of *Book*s to figure 127. Design the classes and a method for looking up the last name of an author by the book title. Also design a method for looking up how many copies of a given book title the store has. ∎

It's time to turn to the full-fledged bookstore problem where books have many authors and authors write many books. The diagram in figure 119 (page 334) solves half the problem; there, each author is associated with many books though each book has exactly one author. To attach many authors to one book, we need an appropriate field in *Book*:

IAuthors authors;

where *IAuthors* is the type of a list of *Author*s.

Figure 129 shows the suggested revision of figure 119. Two of the arrows are labeled: 1 labels the containment arrow from *Book* to *IAuthors*; 2 is attached to the corresponding arrow from *Author* to *IBooks*. Each of the two arrows can be used to create a cycle in a collection of instances of these classes. Try it out!

The translation of the class diagram in figure 129 into class definitions can start from the classes in figure 120. In this figure, the *Author* class provides a method for adding a book to an author's list of books; the constructor of the *Book* class calls this method every time a book is created for that author, thus adding the book itself to the list. This invocation of *addBook* must change, however; see figure 130. After all, a *Book* doesn't have a single author anymore, it has many. And adding the book to all authors' book lists requires a method by itself.

Thus our attempt to translate a data description into class definitions has run into a surprising obstacle:

> defining the classes requires the design of a non-trivial method.

In other words, data design and method design can now depend on each other: to define the *Book* class, we need a method that adds the book to each author, and to define this method we really need the class definitions.

Fortunately we know how to design methods, at least in principle, so let's forge ahead. Specifically, the design of methods can proceed based on the class diagram, and this is what we try here. Following the precedent from the first example in this section, we ignore the containment arrow pointing back from *Author* to *Book*—labeled 2 in figure 129—for the design of the method. Without this arrow, the design problem is just another instance of the problems we know from chapter II.

Representing a list of authors is easy at this point; we use *IAuthors* for the type of lists, and *MtAuthors* and *ConsAuthors* for the concrete variants. The method signature of *addBookToAll* and purpose statement shows up in the interface:

<u>inside of *IAuthors* :</u>
// add the book to all the authors on **this** list
void *addBookToAll*(*Book bk*);

The return type is void because the purpose of the method is modify each item on the list, not to compute something. The appropriate method templates look like this:

```
+----------+                              +----------+
| IBooks   |<-----------------+     +--->| IAuthors |<-----------------+
+----------+<------------+ |     |     +----------+                    |
     |                   | |     |          |                          |
    / \                  | |     |         / \                         |
    ---                  | |     |         ---                         |
     |                   | |     |          |                          |
  +--+------------+      | |     |       +--+------------+             |
  |               |      | |     |       |               |            |
+---------+ +--------------+ | |  |  +-----------+ +-------------+    |
| MTBooks | | ConsBooks    | | |  |  | MTAuthors | | ConsAuthors |   |
+---------+ +--------------+ | |  |  +-----------+ +-------------+   |
        +----| Book fst     | | |  |          +----| Author fst  | | |
        |    | IBooks rst   |-+ |  |          |    | IAuthors rst|-+ |
        |    +--------------+   |  |          |    +-------------+   |
        |                      |  |          |                      |
        |                      |  |          |                      |
        v                      |  |          v                      |
+------------------+           |  |  +----------------+             |
| Book             |           |  |  | Author         |            |
+------------------+           |  |  +----------------+            |
| String title     |          |  |  | String first   |            |
| IAuthors authors |--/1/-------------+  | String last    |       |
| int price        |          |  |  | int dob        |            |
| int quantity     |        +----/2/----| IBooks books   |
+------------------+                  +----------------+
```
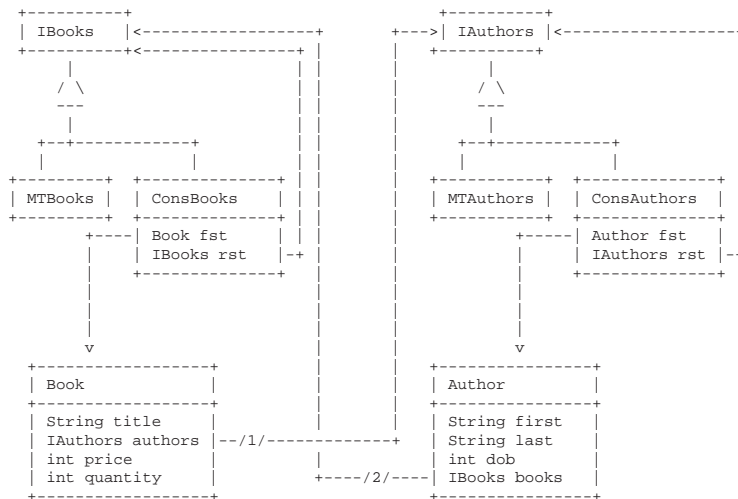
Figure 129: Books and authors: the class diagram

```
class Author {                          class Book {
  String fst;                             String title;
  String lst;                             int price;
  int dob;                                int quantity;
  IBooks bk = new MTBooks();              IAuthors ath;

  Author(String fst, String lst, int dob) {   Book(String title, int price,
    this.fst = fst;                                  int quantity, IAuthors ath) {
    this.lst = lst;                         this.title = title;
    this.dob = dob;                         this.price = price;
  }                                         this.quantity = quantity;
                                            this.ath = ath;
  void addBook(Book bk) {
    this.bk = new ConsBooks(bk, this.bk);   this.ath.???;
    return ;
  }                                       }
}                                       }
```

Figure 130: Books and authors: the classes

inside of *MtAuthors* :
```
void addBookToAll(Book b) {

  . . .

}
```

inside of *ConsAuthors* :
```
void addBookToAll(Book b) {
   . . . this.fst.addBook(b) . . .
   . . . this.rst.addBookToAll(b) . . .
}
```

```
// lists of authors
interface IAuthors {
  // add the book to all the authors on this list
  void addBookToAll(Book bk);
}
```

```
class MTAuthors
  implements IAuthors {
 MTAuthors() {}

 void addBookToAll(Book bk) {
  return ;
 }
}
```

```
class ConsAuthors
  implements IAuthors {
 Author fst;
 IAuthors rst;

 ConsAuthors(Author fst, IAuthors rst) {
  this.fst = fst;
  this.rst = rst;
 }

 void addBookToAll(Book bk) {
  this.fst.addBook(bk);
  this.rst.addBookToAll(bk);
  return ;
 }
}
```

Figure 131: Authors and *addBookToAll*

As always, the template is empty in *MtAuthors*. Similarly, the template for *ConsAuthors* contains two method invocations: one for the *fst* field and one for the *rst* field. The purpose of the latter is to add the book *book* to the authors that are on the rest of the list; the purpose of the former is to add the book to the first instance of *Author*. Of course, this second task is exactly what *addBook* does, so there is no need to define a second method. The only truly new part is that the two methods return void, and we have never combined void values before. Indeed, there is no real operation for combining void values, so we use ";" to throw them away and to return void again from the end of the method. Figure 131 contains the complete definitions, which are easy to figure out with what we have discussed so far.

Exercise

Exercise 24.4 Perform the last step of the design recipe for classes, which in

this case is also the first step of the design recipe for methods: the creation of data examples. Represent the following information about authors and books with our chosen classes:

1. Donald E. Knuth. *The Art of Computer Programming*. Volume 1. Addison Wesley, Reading, Massachusetts. 1968.

2. Donald E. Knuth. *The Art of Computer Programming*. Volume 2. Addison Wesley, Reading, Massachusetts. 1969.

3. Donald E. Knuth. *The Art of Computer Programming*. Volume 3. Addison Wesley, Reading, Massachusetts. 1970.

4. Matthias Felleisen, Robert B. Findler, Matthew Flatt, Shriram Krishnamurthi. *How to Design Programs*. MIT Press, Cambridge, Massachusetts. 2001.

5. Daniel P. Friedman, Matthias Felleisen. *The Little LISPer*. Trade Edition. MIT Press, Cambridge, Massachusetts. 1987.

Use auxiliary fields to make your life as easy as possible. ∎

Now that we have a full-fledged data representation for books and authors, let's repeat the design of the *lookupTitle* method. Naturally, this time the method doesn't return a single title but a list of titles, presumably represented as *String*s. Let's use *ITitles* for the type of all lists of book titles; you can define this representation yourself.

The first step is to write down the signature and purpose statement of the method that goes into the interface for lists of authors:

> <u>inside of *IAuthors*</u> :
> // produce the list of book titles that
> // the author wrote according to **this** list
> *ITitles lookupTitles*(*String last*);

The method signature of *lookupTitles* is the translation of the problem statement into code: the method works on lists of authors and it consumes an author's name. The result has type *ITitles*. This time we ignore link 1 in figure 129, the containment arrow from *Book* back to *IAuthors*.

The second design step is to develop functional examples. For simplicity, we use the data examples from exercise 24.4, assuming you have translated the information into data and called the resulting list *all*:

    check *all.lookupTitles*(`"Knuth"`) **expect**
     new *ConsTitles*(`"The Art of Computer Programming (volume 3)"`,
      new *ConsTitles*(`"The Art of Computer Programming (volume 2)"`,
       new *ConsTitles*(`"The Art of Computer Programming (volume 1)"`,
        new *MTTitles*())))

The method call requests all the books that Knuth wrote. The list of resulting titles is presented in reverse chronological order, because we assume that the chronological order is the one in which the books have been added to the author's list of books.[46]

For the third step, we take inventory and write down what we know for each concrete method in the two *IAuthors* classes:

inside of *MTAuthors* :	inside of *ConsAuthors* :
ITitles lookupTitles(*String lst*) {	*ITitles lookupTitles*(*String lst*) {
...	... **this**.*fst* ... // *Author*
}	... **this**.*rst.lookupTitles*() ... // *ITitles*
	}

In the case of the empty list, we know nothing else. For the case of a constructed list, we know that there is an author and, via the natural recursion, another list of titles.

You can finish the definition of *lookupTitles* in *MTAuthors* trivially; if there are no (more) authors, the result is the empty list of titles. The case of *ConsAuthors* requires a bit of planning, however. First, we distinguish two cases: the *Author* object is for the author with last name *lst* or it isn't. In the latter case, the result of the natural recursion is the result of the method. Otherwise, the search process has found an author with the matching name and therefore **this**.*fst.bk* provides access to the list of the author's books.

Put differently, we get two entries on our wish list:

1. *is*, a method that compares an author's last name with a given string:

    inside of *Author* :
    // is the last name of **this** author equal to *lst*?
    boolean *is*(*String lst*)

2. *allTitles*, a method that extracts the list of titles from a list of books

    inside of *IBooks* :
    // the list of titles from **this** list of books
    ITitles allTitles()

---

[46] An alternative is to represent the result as a set and to compare the results with set-equality; see section 21.

The first method has a simple, one-line definition; the second one belongs to lists of books and thus leads to a wish list entry.

Exercises

Exercise 24.5 Create an Example class from the test case for *lookupTitles*. Develop two additional examples/tests: one for an author who has a single book to his name and one for an author with no books. ∎

Exercise 24.6 Complete the definition of *lookupTitles* assuming the methods on the wish list exist. ∎

The design of the *allTitles* method poses the same problem as the design of *lookupTitles*. This time it is the *authors* field in *Book* that causes the circular link in the chain. Hence we ignore this field and proceed as usual:

inside of *IBooks* :
// produce the titles of **this** list of books
ITitles allTitles();

The signature shows that the method just traverses an object of type *IBooks*, i.e., a list of books; the purpose statement repeats our task statement in a concise manner.

For the example step, if *knuth* stands for the list of all of Knuth's books from exercise 24.4 (in order), we should expect this result:

check *knuth.bk.allTitles*() **expect**
  new *ConsTitles*(`"The Art of Computer Programming (volume 3)"`,
    new *ConsTitles*(`"The Art of Computer Programming (volume 2)"`,
      new *ConsTitles*(`"The Art of Computer Programming (volume 1)"`,
        new *MTTitles*())))

This is, of course, also the result of *all.lookupTitles*(`"Knuth"`). In other words, we have exploited the example for one method to create an example for an auxiliary method, which is a common way to develop tests.

Creating the template is also similar to what we did before:

inside of *MTBooks* :	inside of *ConsBooks* :
ITitles allTitles() {	*ITitles allTitles*() {
. . .	. . . **this**.*fst* . . . // *Book*
}	. . . **this**.*rst.allTitles*() . . . // *ITitles*
	}

As before, *MTBooks* contains no other fields, so the template contains no

expressions; the *ConsBooks* contains two fields and the method template contains two lines: one that reminds of the first book and one that reminds us of the natural recursion, which collects the rest of the list of titles.

The step from the template to the full definitions is again straightforward. One method returns the empty list of titles and the other one combines the title from the first book with the list of titles from the second book.

Exercises

Exercise 24.7 Design the *is* method for *Author*. Finish the design of *allTitles*; in particular, run the test for the method and ensure it works.

Finally, after both definitions are confirmed to work run the tests from exercise 24.5. ∎

Exercise 24.8 Add computer science books of two distinct authors with the same last name to the inventory. In this new context, develop a functional example that uses this new last name; turn it into a test. If the program from exercise 24.7 fails the test, modify it so that it passes this new test. ∎

Exercise 24.9 Design *ITitles* so that it represents a set of titles instead of a list. In other words, the order of entries shouldn't matter. Equip the class with the method *same*, which checks whether **this** set contains the same elements as some given set. Hint: See exercise 19.4 and Section 21.

Reformulate the test cases in exercise 24.5 using *same* and make sure that your solution of exercise 24.7 still passes these tests. What is the advantage of doing using sets? ∎

Our simple idea worked. If we just ignore the fields that create circularity and proceed with the design of methods as before, we can successfully design methods for circular data. In short, the problem is one of *viewing* the data properly and of understanding the design of the data representation from the perspective of the method design problem.

Exercises

Exercise 24.10 Design the method *findAuthors*, which given the title of a book, produces the names of its authors. ∎

Exercise 24.11 Add the following methods to the bookstore program:

1. *titlesInYear*, which produces a list of all book titles that some author with given last name published during some given year;

2. *allAuthors*, which produces the list of all last names;

3. *value*, which computes the value of the current inventory. ∎

Exercise 24.12 This exercise extends the design problem of exercise 23.6. Design the following methods:

1. *numberCourses*, which counts the courses a student is taking;

2. *takesCourse*, which helps instructors figure out whether some student (specified via a last name) is enrolled in a given course;

3. *jointCourses*, which allows the registrar to determine the enrollment common to two distinct courses as a list of last names.

  Also add enrollment limits to each course. Make sure that a method for enrolling a student in this course enforces the enrollment limit. ∎

Exercise 24.13 This exercise resumes exercise 23.5. Design the *sameDoctor* method for the class of patients. The method consumes the last name of some other patient and finds out whether the two are assigned to the same primary care physician. ∎

Exercise 24.14 Design the method *findDirection* to both data representations of a subway line from exercise 23.7. The method consumes the name of the station to where a customer would like to go and produces a short phrase, e.g., `"take the train to ..."` or `"you are  at ..."`. ∎

25 The State of the World and How It Changes

The goal of creating cyclic collections of objects has forced the introduction of a new computation mechanism and concept: assignment statements and the idea that one and the same field can stand for many different values during a program evaluation. Not surprisingly, this idea has more uses than just the creation of cyclic objects. Before we study those and how they affect the design of classes, let's stop and assess what we have learned:

1. "Equations" aren't equations; they are assignments. For now, the left side of an assignment is a field; the right side is an arbitrary expression. The field on the left is either a field in the current class, e.g.,

<u>inside of *Author* :</u>
this.*books* = **new** *ConsBooks*(*bk*,**this**.*books*);

or it is a field in some other class assuming this field is public, e.g.,

<u>inside of *Book* :</u>
this.*ath.bk* = **this**;

The purpose of an assignment statement is to change what the field represents.

2. If the return type of a method is void, the method does not communicate the results of its computation directly to its caller. Instead, it changes some fields, from which the caller or some other piece of code retrieves them. When the method is done, it returns the invisible void value.

   Your programs can't do much with the void value. It exists only as a token that signals the end of some computation. Conversely, it allows the *next* computation in the program to proceed.

3. The purpose of "*statement1* ; *statement2*" in Java programs is to say that the evaluation of *statement2* begins when *statement1* is done. The value of *statement1* is ignored. In particular, when you see

   this.*field1* = *field1*;
   this.*field2* = *field2*;
   . . .

   in a constructor, the assignments are evaluated in this order, and their results are thrown away.

All of this has profound implications. Most importantly, now time matters during program evaluation. In the past, we acted as if a field in a specific object stands for one and the same value during the entire program evaluation. After the introduction of assignment statements, this is no longer true; the value of a field can change and we must learn to figure out what the *current* value of a field is when we encounter a reference during an evaluation.

Next, if assignments change what fields represent, we can use assignments to represent changes in the world of our programs. Consider a Java object that represents a dropping ball. Thus far we have created a new object for every tick of the clock, but the fact that field values can be changed suggests we have an alternative.

Last but not least, assignments change what we mean with equality. When we update the list of books in an *Author* object, the object changes yet it remains the same. When we propose to keep track of the height of a dropping ball via assignment, we are proposing that the ball changes and yet it also remains the same.

The remaining sections explore these topics in detail. First, we show how to use our new powers to represent change. Second, we study equality, i.e., what it means for two objects to be the same.

```
class DrpBlock {                          class DrpBlock {
  int ow = 10;                              int ow = 10;
  int oh = 10;                              int oh = 10;
  int x;                                    int x;
  int y;                                    int y;
  IColor oc = new White();                  IColor oc = new White();

  DrpBlock(int x, int y) {                  DrpBlock(int x,int y) {
    this.x = x;                               this.x = x;
    this.y = y;                               this.y = y;
  }                                         }

  DrpBlock drop() {                         void drop() {
    return                                    this.y = this.y + 1;
      new DrpBlock(this.x,this.y+1);          return ;
  }                                         }

  boolean isAt(int h) {                     boolean isAt(int h) {
    return this.y + this.oh >= h;            return this.y + oh >= h;
  }                                         }

  boolean draw(Canvas c) {                  boolean draw(Canvas c) {
    return c.drawRect(...);                   return c.drawRect(...);
  }                                         }
}                                         }
```

Figure 132: Blocks with Changing State: Applicative vs. Imperative

26   Assignments and Changes in the World

Your goal for this section is to gain an intuitive understanding of what it means to represent changes in the world via assignments to fields. We com-

pare and contrast this new idea with the old approach via three examples, including a re-designed drawing package. These examples set up the next section, which presents the principles of design of imperative classes and methods.

```
class BlockWorld extends World {
  private int WIDTH  = 200;
  private int HEIGHT = 200;
  private IColor bgrdColor = new Red();

  private DrpBlock block = new DrpBlock(this.WIDTH / 2,0);

  public BlockWorld() {
   this.bigBang(this.WIDTH,this.HEIGHT,.01);
  }

   public World onTick() {
    this.block.drop();
    if (this.block.isAt(this.HEIGHT)) {
     return endOfWorld("stop!");
    }
    else {
     return this; }
   }
  ...
}
```

Figure 133: One More World of Dropping Blocks

26.1   Example: Dropping Blocks

The first example takes us back to the world of dropping blocks. Figure 132 displays two versions of our exploratory program for dropping a single block from the top of a canvas to the bottom. On the left you see the familiar program that creates a block from its *x* and *y* coordinates and that creates a new block for every clock tick.

On the right side, you find a rather different version employing void and an *assignment* statement. The two versions differ in just one place: the *drop* method. While the left-hand version creates and returns a new instance of *DrpBlock* for each clock tick, the right-hand version changes the *y* coordinate of the instance.

One way of viewing the two classes is to perceive them as two different representations of change in the world (as in, the domain with which our program acts). In the left version the current world, dubbed APPLICATIVE, is a function of time; as time goes by, the program creates new worlds, each representing another moment. This approach is typically used in physical models, chemistry, and other natural sciences as well as in engineering disciplines such as signal processing. In the right version, the world stays the same but some of its attributes change. This second approach, dubbed IMPERATIVE or STATEFUL in this book, is the electrical computer engineer's view of a changing world, based on the idea of computing as turning "switches" on and off.[47]

Exercise

Exercise 26.1 Add privacy modifiers to the imperative *DrpBlock* class. ∎

Given the new, alternative definition of *DrpBlock*, the question is how *BlockWorld* should change to accommodate the imperative version of *DrpBlock*. Clearly, the method that requires discussion is *onTick*, because it is the one that uses the *drop* method:

inside of *BlockWorld*: (applicative)
World onTick() {
  return new *BlockWorld*(**this**.*block*.*drop*());
}

Now that *drop*'s return type is void, we can no longer use its result to create a new *BlockWorld*. Instead the method for the *imperative* version of *BlockWorld* must invoke *drop* and return a *World*:

inside of *BlockWorld* : (*imperative: version* 1)
World onTick() {
  this.*block*.*drop*();
  return new *BlockWorld*(**this**.*block*);
}

If you stop to think about this new instance of *BlockWorld*, however, you quickly realize that the new *BlockWorld* contains the exact same old block

---

[47]This contrasts with a computer scientist who should be able to use and easily switch between the two views.

and is therefore the same as the old instance of *BlockWorld*. Therefore the
method might as well just produce **this**:

inside of *BlockWorld* : (*imperative: version* 2)
World onTick() {
　this.*block.drop*();
　return this;
}

Of course, we really want the block to land when its bottom reaches the
bottom of the canvas, which means that the actual method also needs to
perform some tests.

Figure 133 displays the final definition of *onTick* for the imperative ver-
sion of *DrpBlock*: see the nested box. As discussed, the method first drops
the block. Then it uses the *isAt* method in *DrpBlock* to determine whether
block has reached the specified *HEIGHT*. If so, the world ends with the block
resting on the ground; otherwise, *onTick* returns **this** world.

26.2   Example: Accounts

Imagine you are working for a bank that wishes to replace its 30-year old
COBOL accounting system with a modern-day Java version. Your manager
is still unfamiliar with this novel technology that everybody else is using
already so she wishes you to conduct a programming experiment:
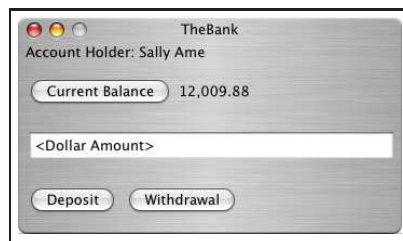
. . .   Design a simple account
class that can perform three typ-
ical bank tasks: deposit money,
withdraw money, and compute
the balance. One of the bank's
clerks has sketched a graphical
user interface for this simple ac-
count (see left). . . .

If you were to design this class according to the conventional design
recipe, you would come up with the following data definition and goals
for designing methods:

1. You need a single class, *Account*, with two fields: one that represents
   the account holder and one for the current balance. For example, the
   account from the sample user interface might be created with

   new *Account*(`"Sally Ame"`,12009.88)

The initial deposit should be larger than 0.

2. The class needs at least three publicly visible methods:

<u>inside of *Account* :</u>
// deposit the amount *a* into **this** account
Account deposit(int *a*)

// withdraw the amount *a* from **this** account
Account withdraw(int *a*)

// create a balance statement from **this** account
String balance()

Naturally the deposit and withdrawal amounts should be larger than 0 and the withdrawal amount should also be less than or equal to the money in the account.

With all your experience by now, it is straightforward to follow the design recipe and to come up with the definition of *Account* shown on the left in figure 134. The only curious tidbit concerns *balance*, which uses the *String.valueOf*[48] method for converting an int into a *String*.

Exercises

Exercise 26.2 Add privacy specifications to the applicative *Account* class. ∎

Exercise 26.3 Add conditionals to *Account*'s constructor, *deposit*, and *withdraw* methods that check on the above stated, informal assumptions about their int arguments. If any of the given amounts are out of the specified range, use *Util.error* to signal an error. ∎

To convert this class into an imperative one, let's follow the outline of the preceding subsection. It suggests that the methods whose return type is *Account* should be converted into methods that produce void. Instead of creating a new instance of *Account*, the revised methods change the *amount* field in the existing object. Specifically, *deposit* uses this assignment:

this.*amount* = **this**.*amount* + *a*;

---

[48]This method is another "dotted name" method to you.

```
class Account {                          class Account {
  int amount;                              int amount;
  String holder;                           String holder;

  // create the account                    // create the account
  Account(String holder,int deposit0) {    Account(String holder,int deposit0) {
    this.holder = holder;                    this.holder = holder;
    this.amount = deposit0;                  this.amount = deposit0;
  }                                        }

  // deposit the amount a                   // deposit the amount a
  Account deposit(int a) {                 void deposit(int a) {
    return                                   this.amount = this.amount + a;
      new Account(this.holder,               return ;
                  this.amount + a);        }
  }

  // withdraw the amount a                  // withdraw the amount a
  Account withdraw(int a) {                void withdraw(int a) {
    return                                   this.amount = this.amount − a;
      new Account(this.holder,               return ;
                  this.amount − a);        }
  }

  // a balance statement                    // a balance statement
  // of this account                        // of this account
  String balance() {                       String balance() {
    return                                   return
      this.holder.                             this.holder.
        concat(": ".                             concat(": ".
          concat(                                  concat(
            String.valueOf(this.amount)));           String.valueOf(this.amount)));
  }                                        }
}                                        }
```

Figure 134: Bank Accounts: Applicative vs. Imperative

while *withdraw* uses this one:

$$\textbf{this}.amount = \textbf{this}.amount - a;$$

In general, the constructor argument for the changing field becomes the right-hand side of the respective assignment statement.

The complete imperative definition of *Account* is displayed on the right side of figure 134. Because the imperative methods don't need to say that the *holder* field stays the same, they are actually a bit shorter than the applicative ones.[49]

Using the imperative version of *Account* is quite similar to using the applicative one:

Account a0 = **new** *Account*("I",10);	*Account a3* = **new** *Account*("I",10);
Account a1 = *a0.deposit*(20);	*a3.deposit*(20);
Account a2 = *a1.withdraw*(10);	*a3.withdraw*(10);
check *a2.balance*() **expect** "I: 20"	**check** *a3.balance*() **expect** "I: 20"

For the applicative version, we use a series of definitions. Each of them introduces a new instance (*a1*, *a2*) as the result of using *deposit* or *withdraw* on the latest account. For the imperative one, we use a series of method invocations. Each of these invocations returns void after modifying the *amount* field of *a3*. Because we don't have to care about the return values, we can use ";" to compose these statements. The significant difference between the two examples is that the old accounts with the old balances are available in the applicative version; the assignments to *amount* in the imperative versions literally destroy the relationship between *amount* and its values.

26.3 How Libraries Work 3: An Alternative World

The *Canvas* class in the **draw** package contains four methods that produce true. If they produce true, their computation has succeeded; they have drawn a shape on the canvas; the rest of the computation may proceed now. If they don't produce true, their computation has failed, because, say, a *drawRect* was requested before *start* was called. In this case, the program evaluation is stopped with an error.

When we invoke the *Canvas* methods from **draw**, we combine them with *&&* to make several changes to the canvas. For example,

Canvas c = **new** *Canvas*();
c.start(200,200) *&&*
c.drawRect(**new** *Posn*(0,0),200,200,**new** *Red*()) *&&*
c.drawString(**new** *Posn*(100,100),"hello world");

---

[49]In general, if a class had 10 fields and one of them changed, an applicative method would call the constructor with 10 arguments, nine of which stay the same; an imperative method contains only one assignment statement. Of course, a programming language could provide support for shortening applicative methods to the same length.

creates a 200 by 200 canvas with a red background and the words "hello world" close to the center.

```
// to represent a world
abstract class World {
  Canvas theCanvas = new Canvas();

  // open a width by height canvas,
  // start the clock, and
  // make this world the current one
  void bigBang(int width, int height,
                double s)

  // process a tick of the
  // clock in this world
  abstract void onTick()

  // process a keystroke
  // event in this world
  abstract void onKeyEvent(String ke)

  // draw this world
  abstract void draw()

  // stop this world's clock
  World endOfWorld(String s)

  // stop this world's clock
  World endOfTime()
}
```

```
// controlling a computer canvas
class Canvas {
  int width;
  int height;
  ..
  // display a canvas of
  void show()

  // draw a circle at p
  void drawCircle(Posn p, int r, IColor c)

  // draw a solid disk at p,
  // fill with color c
  void drawDisk(Posn p, int r, IColor c)

  // draw a width x height rectangle
  // at p, fill with color c
  void drawRect(Posn p,int width,int height,
                IColor c)

  // draw s at position p
  void drawString(Posn p, String s)

  . . .
}
```

Figure 135: The **idraw** package

Even though the use of boolean works reasonably well, the use of void as a return type would express our intentions better than boolean. After all, the methods just need to return *anything* to signal that they are done; what they return doesn't matter. Furthermore, they also effect the state of some object, in this case, the canvas on the computer screen. It is exactly for this situation that Java provides void. Similarly, using *&&* for sequencing changes to the canvas is a bit of an abuse of what boolean values are all about. Once the methods return void, it becomes natural to use ";" for sequencing effects, and this is what it symbolizes. For example, the above

piece of code would become

> *Canvas c* = **new** *Canvas*();
> *c.start*(200,200);
> *c.drawRect*(**new** *Posn*(0,0),200,200,**new** *Red*());
> *c.drawString*(**new** *Posn*(100,100), `"hello world"`);

if *start*, *drawRect*, and *drawString* returned void.

Figure 135 (right side) displays the skeleton of the *Canvas* class from the **idraw** package. In this class, all of the methods have a void return type, indicating that they work by changing the object not by producing a value. Their actual workings remain the same. To use them in order, however, you need to use ";" instead of *&&*.

The left part of figure 135 displays the *World* class from the **idraw** package. The changes to the Canvas class suggest two immediate changes to the *World* class: *draw* uses void as its return type. Surprisingly though, most of the other methods also produce void. Remember that in the **draw** package, the *bigBang* method produces true to signal its work is done; again, void expresses this better than boolean.

To explain the other changes, let us look back at the two key methods in the **draw** package:

1. *onTick* (in **draw**) consumes the current world as **this** and produces a new world, i.e., a world that represents the changes that happened during one tick of the clock.

2. *onKeyEvent* (in **draw**) consumes the current world as **this** and a *String* representation of a keyboard event; it produces a world that represents the changes that happened due to this keyboard event.

That is, the methods implement transitions and changes in our representation of a (small) world. Given this analysis, it is natural to say that these changes can be accomplished via assignments to fields rather than the creation of new objects and the **idraw** package does so. The use of void as the return type for both *onTick* and *onKeyEvent* implies that when your classes override these methods, your methods must change some of the fields in your subclasses of *World*.

Let us see how this works for our running example, the world of a dropping block. We can't quite use the classes from figures 132 (right side) and 133. Even though these classes are imperative, they don't work with **idraw** because the types of *onTick* and *onKeyEvent* are wrong.

So we start from scratch. The library design recipe says that we have to design the four required[50] methods:

```
import idraw.*;
import colors.*;
import geometry.*;

class BlockWorld extends World {
  ...
  private DrpBlock block;

  public BlockWorld()  { ... to be designed ... }

  // draw the falling block in this world
  public void draw()  { ... to be designed ... }

  // drop the block in this world by a pixel
  public void onTick() { ... to be designed ... }

  // do nothing
  public void onKeyEvent(String ke) {
    return ;
  }
}
```

The *block* field is necessary because this world of blocks contains one dropping block; the *onKeyEvent* method doesn't do anything for now, so it just returns void. Clearly, it is *onTick* (again) that requires our attention.

A brief consideration suggests that the task of *onTick* is to change the *block* field; after all it is the only thing that changes in this world. We record this insight in the purpose statement:

inside of *BlockWorld* :
// drop the block in **this** world by a pixel
// <u>effect</u>: change the block field
public void *onTick*() { ... to be designed ... }

This second line of the purpose statement is called an EFFECT STATEMENT. The objective of an effect statement is to help us design the method and to alert future readers to the change of fields that this method causes.

---

[50]Recall that labeling a method with **abstract** means that subclasses are required to implement them.

Together the purpose statement and the effect statement help us take the first step in our design:

inside of *BlockWorld* :
// drop the block in **this** world by a pixel
// <u>effect</u>: change the block field
public void *onTick*() {
  this.*block* = . . . a new block that is lower than the current one . . .
  return ;
}

From the informal right-hand side of this assignment statement we know that we need the usual *drop* method from the *applicative DrpBlock* class. Hence, the following is a minimal *onTick* method:

inside of *BlockWorld* :
// drop the block in **this** world by a pixel
// <u>effect</u>: change the block field
public void *onTick*() {
  this.*block* = **this**.*block.drop*();
  return ;
}

The mostly complete class definition for an imperative *BlockWorld* (using **idraw**) is shown on the left side of figure 136. The *draw* method is as expected. It uses the drawing methods from *Canvas* and return void. The *onTick* method performs one additional task compared to the one we design: when the block reaches *HEIGHT*, the clock is stopped and with it the animation.

The right side of the figure defines a mostly applicative *DrpBlock*. Its *drop* method is applicative, i.e., it returns a new instance of the class with new coordinates. Indeed, the class has barely changed compared to its purely applicative method. Only its *draw* method uses an imperative drawing method and ";" to sequence two statements.

Exercise

Exercise 26.4 Design methods for controlling the descent of the block with left and right arrow keys. ▪

At this point, you may wonder what it would be like to use the *imperative* version of *DrpBlock* from figure 133. Even though the method signa-

tures aren't quite right, it is after all the block whose properties change over time. Hence it should be designed using imperative methods and the rest of the design should follow from this decision.

```
// an imperative world                          // an applicative block
class BlockWorld extends World {                class DrpBlock {
  private int HEIGHT = 200;                       int ow = 10;
  private int WIDTH  = 200;                       int oh = 10;
  private IColor bgrdColor = new Red();           int x;
  private DrpBlock block =                        int y;
    new DrpBlock(this.WIDTH/2,0);                 IColor oc = new White();

  public BlockWorld() {                           DrpBlock(int x, int y) {
    this.bigBang(...);                              this.x = x;
  }                                                 this.y = y;
                                                  }
  public void draw() {
    this.block.draw(theCanvas);                   DrpBlock drop() {
    return ;                                        return
  }                                                   new DrpBlock(this.x,this.y+1);
                                                  }
  public void onTick() {
    this.block = this.block.drop();               boolean isAt(int h) {
    if (this.block.isAt(this.HEIGHT)) {             return this.y + this.oh >= h;
      this.endOfTime();                           }
      return ; }
    else {                                        boolean draw(Canvas c) {
      return ; }                                    c.drawRect(...);
  }                                                 return true;
                                                  }
  public void onKeyEvent(String ke) {            }
    return ;
  }
}
```

Figure 136: The Last World of Dropping Blocks (**idraw** package)

Figure 137 displays the revisions of the two class definitions based on this alternative design strategy. The important changes are highlighted:

1. The return type of *drop* is void because the method achieves its purpose in an imperative manner.

2. The assignment statement changes the value of **this**.*y*, which repre-

sents the physical drop of the block.

3. Afterwards the method returns the void value.

4. In *onTick* of *BlockWorld*, it is no longer possible or necessary to assign the result of **this**.*block.drop*() to **this**.*block*. First, the result is void, not a *DrpBlock*. Second, the *drop* method is invoked for its effect only, meaning it changes the block's fields. Hence, *BlockWorld* doesn't have to keep track of a new block; it just hangs on to the only one it has.

Also look at the effect statement for *drop* in *DrpBlock*. Like the one for *onTick* before, this statement announces what changes on each call to *drop* so that a reader doesn't have to study all of *drop*'s body.

We have just discussed several different designs of the world of dropping blocks, based on **draw** and **idraw** and using and mixing applicative and imperative mechanisms. Go back through this section and look at all the figures again before you go on to the next section, where we discuss the design recipe for imperative classes and methods.

Exercise

Exercise 26.5 Design methods for controlling the descent of the block with left and right arrow keys for the classes of figure 137. ∎

27   Designing Stateful Classes, Imperative Methods

Recall from sections 6 and 16 that the purpose of classes is to represent a collection of related pieces of information and their legal methods; the purpose of a method is to produce data from its arguments, including the data in a specific instance of the class. In this section, we discuss the design of stateful classes and the design of methods that can change the fields of their instances. Of course, the very existence of a design alternative (for classes and methods) raises several questions:

1. When is it preferable to use stateful classes?

2. How do we design such classes and their imperative methods?

3. Does the template play a role?

4. How does statefulness affect the abstraction step?

```
// an imperative world
class BlockWorld extends World {
  private int HEIGHT = 200;
  private int WIDTH  = 200;
  private IColor bgrdColor = new Red();
  private DrpBlock block =
    new DrpBlock(this.WIDTH/2,0);

  public BlockWorld() {
    this.bigBang(...);
  }

  public void draw() {
    this.block.draw(theCanvas);
    return ;
  }

  public void onTick() {
    this.block.drop() 4 ;
    if (this.block.isAt(this.HEIGHT)) {
      this.endOfTime();
      return ;
    }
    else {
      return ; }
  }

  public void onKeyEvent(String ke) {
    return ;
  }
}
```

```
// an imperative block
class DrpBlock {
  int ow = 10;
  int oh = 10;
  int x;
  int y;
  IColor oc = new White();

  DrpBlock(int x, int y) {
    this.x = x;
    this.y = y;
  }

  // effect: change the y field
  void 1 drop() {
    this.y = this.y+1; 2
    return ; 3
  }

  boolean isAt(int h) {
    return this.y + this.oh >= h;
  }

  void draw(Canvas c) {
    c.drawRect(...);
    return ;
  }
}
```

Figure 137: The Very Last World of Dropping Blocks (**idraw** package)

5. What is the cost of (using) imperative methods?

The following subsections answer these questions, in roughly this order, and then resume our running case study of the "War of the Worlds" game.

27.1   When to Use Stateful Classes and Imperative Methods

The last example in the preceding section suggests a simple and obvious guideline for the introduction of stateful classes and imperative methods:

> If the libraries that you use provide stateful classes and imperative methods, you must consider designing stateful classes and imperative methods, too.

library criterion[(1)]

Thus, because the *Canvas* class of **idraw** implements a number of methods with a void result type, methods that use them are naturally imperative, too. The *World* class in **idraw** is an even better example than *Canvas*. If you wish to use this stateful class to implement an animated world, you must override *onTick* and *onKeyEvent* (among others), two methods with void return type. This implies that your methods cannot return a new "current" world and have someone else[51] take care of it; instead your methods must modify the fields in **this** object (and/or the fields of some related objects) to keep track of changes and transitions.

Of course, a comparison of figures 136 and 137 shows that it isn't just a question of library versus non-library code. Consider the following division of tasks:

- We provide the **idraw** library.

- Someone recognized the imperative nature of **idraw** and designed the *BlockWorld* subclass.

- You are asked to design the *DrpBlock* class.

How you execute your task all depends on the existing *BlockWorld* class. If it follows the approach of figure 136, you are free to choose an applicative or an imperative approach; if she chooses the approach of figure 137, you must design *DrpBlock* as a stateful class, and you must make *drop* assign new values to the *y* field.

Put differently, it isn't so much about the separation of library and non-library code but about how your classes and methods are to interact with the rest of the project. This suggests a revised motto:

> If you design is to use *any* stateful classes, consider designing stateful classes and imperative methods, too.

library criterion[(2)]

---

[51]Now that you have co-designed an imperative block world with an applicative block inside (figure 136), you understand in principle how *World* itself works.

In short, it is all about the line that somebody drew between your code and the rest of the code, and this line determines the nature of your design.

While both version of the criterion are pragmatic and straightforward to understand, they actually fail to answer the true question, which is when we want to use stateful classes. After all, we still don't know why the designers of the library or the project chose to introduce stateful classes in the first place. And if we wish to become good designers, we must understand their reasoning, too.

So, let's ignore the library example from section 27 for a moment and focus on the first two: the stateful definition of *DrpBlock* and the *Account* class. In both cases, we started from the premise that the class represents information about objects that change over time. One way to recognize whether a class represents changing objects is to study the relationship among its methods, before you implement them. If one of the methods should produce different results *depending* on whether or how the methods of the class have been called *in the past*, consider using assignments to the fields. Put differently, if time matters, applicative classes represent it explicitly; stateful classes represent it implicitly with assignment statements.

Consider the *Account* class and the methods that its graphical interface suggests: *deposit*, *withdraw*, and *balance*. A moment's thought suggests that the result of *balance* should depend on what amounts the account holder has deposited and withdrawn since the account has been created. Similarly, for *DrpBlock* objects the result of *isAt* depends on the flight time of the block, i.e., on how often *drop* has been called.

Thus, our second criteria is this:

> If any of the to-be-designed methods of a class must compute results that depend on the history of method invocations (for an object), consider making the class stateful and some of its methods imperative.

| time (history) criterion

The designers of your libraries use this criterion, implicitly or explicitly, when they create classes and methods for others to use. They imagine what kind of things you want to compute with their methods or methods that augment their classes (via subclassing). If their libraries control changing physical objects, such as computer displays or sensors, they might choose an imperative approach. If they imagine that your subclasses represent changing objects, they are also likely to provide stateful classes and imperative methods.

Since you know that alternatives exist, i.e., that it is possible to represent change with applicative classes and methods, you might wonder why it is more likely that library designers use imperative approaches,[52] thus inducing you to use imperative methods. There are two different answers to this question:

1. The first answer concerns the complexity of the software. Although the stateful classes and imperative methods we discussed don't look any simpler than their applicative counterparts, this is not true in general. The hallmark of the applicative approach is that a method signature describes all possible channels on which values may flow into or out of a method. Of course, this is also a restriction and, as section 27.6 below shows, can lead to contorted signatures. In the worst case, these contortions can become rather complex, though people have not found a good way to measure this form of complexity.

2. The second answer concerns the complexity of the computation that the software performs, which are measurable quantities[53] such as time, energy, or space. To this day, people don't know yet how to make applicative classes and methods as *efficient* as imperative ones. That is, for many situations an imperative approach to state-changes over time run faster than applicative ones, consume less energy than an applicative one, and so on.

   How to Design Programs introduced the issue of running time only at the very end. As you learn to design pieces of a program that others may use or that are going to survive for a long time, you will need to pay more and more attention to this issue. This chapter is a first step in this direction. As you pursue this path keep in mind, though, that before you worry about your method's running time or energy consumption, you must get them correct and you must design them so that you and others can understand and adapt them to new circumstances and improve them as needed.

With this in mind, let's turn to the revision of the design recipe.

---

[52]This is true for Java and many object-oriented programming languages, though not for OCAML and PLT Scheme, which both support class systems, too.

[53]They have therefore gained dominance in traditional approaches to programming.

27.2   How to Design Stateful Classes, Imperative Methods

The design of stateful classes and imperative methods appears similar to the design of applicative classes and methods at first glance but a new step introduces an entirely new element. As before, your first task is still to design a data representation for the information in your problem statement (section 6), but now you also need to determine whether any of the classes are imperative. Since our second "imperativeness" guideline depends on a preliminary understanding of the methods, you also need to pay attention to those as you figure out what classes you need and how they are related:

problem analysis   Extract from your problem statement what information you need to represent and what kind of classes you need. List the potential classes by name and add a purpose statement to each.

class diagram (data definition)   Draw a **class diagram**, which means add fields to the classes and determine how they all relate to each other via containment arrows. Don't forget to use interfaces and inheritance arrows to represent unions; introduce abstract classes only if they are truly obvious.

stateful classes   Your new task is to decide whether any of your classes should be stateful and which of your methods should be imperative. To this end, you compile a wish list of methods, writing down the names of the methods that you might need according to the problem statement. Add a rough purpose statement to each method name, that is, say how the objects of the class should react to these method invocations.

Now imagine how the methods ought to work. If the results of any of the methods depends on the history of method invocations, the methods are going to be imperative. If your classes must override methods with a void return type or if their purpose statement describes them as imperative, you have also identified an imperative method.

Last but not least, try to imagine enough about the methods' workings so that you can guess which fields these imperative methods should change.

Note 1: Remember deciding whether a method is applicative or imperative can be difficult—even with the guidelines. In many cases, you can use one or the other design. If you need to introduce a number of auxiliary classes (or interfaces) for the applicative version, thus

increasing the complexity of your world, you are almost always better off with an imperative design.

Note 2: The decision to make classes stateful may actually require a change to your data definitions. This is especially true if you decide that a class ought to come with imperative methods and your part of the program doesn't have control over all the instances of the class. Section 27.8 illustrates this point.

Hint: If the collection of classes under your control has a *World*-like class that contains everything else, start with it. The verbs in the problem statement may help you figure out what kind of actions can be performed on the objects of this class. Then, using your imagination and experience, proceed along the containment and inheritance arrows to add methods in other classes that have to support those in your all-encompassing one.

class definitions Define the interfaces and classes; equip each definition with a purpose statement that explains its purpose. You may also want to add a note saying to which of the fields methods may assign new values.

data examples Translate examples of information from the problem statement into data examples. Also be sure that you can interpret data examples in the problem domain. Provide examples that illustrate how objects evolve over time.

With this new step in mind, let us take a look at the *Account* example again. When you are asked to design such a class, it is quite obvious that there is one class with (at least) two fields (*owner*, *amount*) and three methods: *deposit*, *withdraw*, and *balance*. We have already discussed how the latter's result depends on the invocation of the former methods. From this we concluded that the class should be stateful, and it is obviously the *amount* field that the methods change; the *owner* remains the same all the time (given the methods of this example).

Your second task is to design the methods with the design recipe. For applicative methods, follow the recipe that you know. For the design of imperative methods, we need to modify the recipe a bit:

signature, purpose, & effect For an imperative method, your first task is to clarify what the method consumes because you already know what

it produces: void.[54] Recall from the design recipe for applicative methods that a method always consumes at least one argument: **this**.

Once you have a signature, you can formulate a purpose and effect statement. It concisely states *what* the method computes and *how* it uses the result of the computation to *change* the object. This latter part, dubbed **effect statement**, can take several forms; for now you may just want to say which fields are changed.

Let's illustrate this step with the familiar example:

> inside of *Account* :
> // to deposit some amount of money into **this** Account
> // effect: to add *d* to **this**.*amount*
> void *deposit*(int *d*)

The signature says that the method consumes an int, which the purpose statement describes as an amount. From the effect statement, we find out that the method changes what the *amount* field represents.

functional examples The purpose of the preceding step statement is to figure out what a method is to compute. When you have that, you formulate examples. Examples of the kind we have used in the past wouldn't work, because they would all look like this:[55]

> **check** *anObject.imperativeMethod*(*arg1,...*) **expect** void

After all, void is the standard result of an imperative method.

Therefore instead of functional examples, you formulate *behavioral* examples. In contrast to a functional example, which specifies the expected result for given arguments, a behavioral example states what kinds of effects you wish to observe after the method invocation returns void.

The simplest form of a behavioral example has this shape:

> *SomeClass anObject* = **new** *SomeClass*(...)

---

[54]Some people prefer to return the object itself from an imperative method, which makes it is easy to chain method invocations. At the same time, it obscures the purpose statement because the method might just be an applicative method simulating changes with the creation of new objects.

[55]Warning: this syntax is illegal.

anObject.imperativeMethod(arg1,. . .);

check *anObject.field1* **expect** *newValue1*
check *anObject.field3* **expect** *newValue2*

That is, given an object whose properties you know because you just constructed it, the invocation of an imperative method returns void. Afterwards, an inspection of the properties ensures that the method assigned changed the desired fields and left others alone.

Sometimes it is too time consuming to create a new object. In those cases, you can observe the properties of an object and use them:

Type1 value1 = anObject.field1;
Type2 value2 = anObject.field2;

anObject.imperativeMethod(arg1,. . .);

check *anObject.field1* **expect** *newValue1*
check *anObject.field3* **expect** *newValue2*

In these **check** expressions, *newValue1* and *newValue2* are expressions that may involve *value1* and *value2*. Indeed, the two variables may also be useful as arguments to the method call.

Last but not least, you are generally better off using applicative methods to inspect the imperative effects of a method because fields are often **private** to a class:

Type1 value1 = anObject.method1();
Type2 value2 = anObject.method2();

anObject.imperativeMethod(arg1,. . .);

check *anObject.method1()* **expect** *newValue1*
anObject.method3() **expect** *newValue2*

Here *method1*, *method2*, and *method3* are methods that do not change the object. As before, they extract values that you may use in the method arguments or in the tests.

Generally speaking, the first half of your behavioral example deter-
mines the current state of the object, i.e., the values of its fields; the
second half specifies the expected changes.[56]

Let's see how this works for our running example:

Account a = **new** *Account*(`"Sally"`,10);

a.deposit(20);

check *a.amount* **expect** 30

If *amount* were **private**, we would have to use the *balance* method to
determine whether *desposit* works:

Account a = **new** *Account*(`"Sally"`,10);

a.deposit(20);

check *a.balance*() **expect** `"Sally: 30"`

This example suggests that you might want to add a method that
produces the balance as an int rather than as a part of a *String*.

Terminology: because of the role that applicative methods play in
this contex, they are also known as OBSERVER methods, or just ob-
servers. Likewise, imperative methods are known as COMMANDS.

template As always the objective of the template step is take stock of the
data to which the method has access. Remember that the template
enumerates the fields, the method parameters (standing in for the ar-
guments), and their pieces if needed. For an imperative method, you
also enumerate the fields that the method should change.

Here is our running example enhanced with a method template:

---

[56]If you also ensure that *value1* and *value2* satisfy certain conditions, people refer to the
first half as preconditions and the second half of the example as postcondition, though
these words are more frequently used for more general claims about methods than specific
examples.

> inside of *Account* :
> // to deposit some amount of money into **this** Account
> // <u>effect</u>: to add *d* to **this**.*amount*
> void *deposit*(int *d*) {
>   … **this**.*amount* = … **this**.*amount* … **this**.*owner* …
> }

The *amount* field shows up twice: on the left of the assignment statement it is a reminder that we consider it a changeable field; on the right it is a reminder that the data is available to compute the desired result. As discussed, the *owner* field should never change.

method definition Now use the purpose and effect statement, the examples, and the template to define the method. Keep in mind that you should work out additional examples if the ones you have prove insufficient. Also, if the task is too complex, don't hesitate to put additional observer and/or command methods on your wish list.

And indeed, the definition of *deposit* is straightforward given our preparation:

> inside of *Account* :
> // to deposit some amount of money into **this** Account
> // <u>effect</u>: to add *d* to **this**.*amount*
> void *deposit*(int *d*) {
>   **this**.*amount* = **this**.*amount* + *d*;
>   **return** ;
> }

tests Last but not least, you must turn the examples into automatic tests:

> **class** *Examples* {
>   *Account a* = **new** *Account*("Sally",10);
>
>   boolean *testDeposit*() {
>   *a*.*deposit*(20);
>   **return** (**check** *a*.*balance*() **expect** "Sally: 30");
>   }
> }

Testing

Run the tests when the method and all of its helper methods on the wish lists have been designed and tested.

27.3   Imperative Methods and Templates

The creation of templates for applicative methods benefits from a solid understanding of class designs. Section 16.1 discusses the four kinds of organizations for classes and how they relate to templates. Let us take a look at these four situations again and find out what role they play in the design of imperative methods. We are ignoring cyclic collections of object because we reduce the design of methods for them to acyclic collections by ignoring a link.

Basic Classes

A basic class comes with a name and fields of primitive type. In this case, an applicative template enumerates the fields (and the arguments):

```
    +-------------+              +-------------+
    | Basic       |              | Basic       |
    +-------------+              +-------------+
    | String s    |              | String s    |
    | int i       |              | int i       |
    | double d    |              | double d    |
    +-------------+              +-------------+
    | ??? mmm() { |              | void mmm(){ |
    |    this.s   |              |    this.s   |
    |    this.i   |              |    this.i   |
    |    this.d } |              |    this.d   |
    +-------------+              |             |
                                 |  this.s =   |
                                 |  this.d = } |
                                 +-------------+
```

A template for an imperative method differs from this in just one aspect: for each changeable field, introduce an assignment statement with the field on the left and nothing (or all of the fields) on the right. When you finally define the method, the template reminds you that you can use the method arguments and the fields to compute the new value for the changing fields. Also keep in mind that an imperative method may change one field or many fields.

Containment

A field *b* in class *Containment* may stand for objects of some other class, say *Basic*. In this case, it is quite common that a method in *Containment* may have to compute results using the information inside of the value of *b*:

```
    +---------------+              +---------------+
    | Containment   |              | Containment   |
    +---------------+  +-------------+  +---------------+  +-------------+
    | Basic b       |------>| Basic   |  | Basic b       |------>| Basic   |
    | double d      |  +-------------+  | double d      |  +-------------+
    +---------------+  | String s    |  +---------------+  | String s    |
    | ??? mmm() {   |  | int i       |  | ??? mmm() {   |  | int i       |
```

```
| this.d       |      | double d    |      | this.b.nnn() |      | double d    |
|              |      +-------------+      | this.d  }    |      +-------------+
| this.b.s     |                           +--------------+      | ??? nnn() { |
| this.b.i     |                                                 |   this.s    |
| this.b.d }   |                                                 |   this.i    |
+--------------+                                                 |   this.d  } |
                                                                 +-------------+
```

The template on the left says that your method could reach into the contained object via **this**.*b.s* etc. While this is indeed a possibility that we initially exploited, we have also learned that it is better to anticipate the need for an auxiliary method *nnn* in *Basic* that helps *mmm* compute its result.

In principle, we have the same choices for imperative methods. An imperative *mmm* method in *Containment* can change what *s*, *i*, or *d* in *Basic* represent; we used this power to create circular object references in section 23. For the same reasons as in the applicative case, however, it is preferable to design auxiliary methods in *Basic* that perform these changes:

```
+---------------+
| Containment   |
+---------------+          +-------------+
| Basic b       |------>|  Basic      |
| double d      |          +-------------+
+---------------+          | String s    |
| void mmm() {  |          | int i       |
|   this.b.nnn()|          | double d    |
|   this.d =    |          +-------------+
|   this.d  }   |          | void nnn() {|
+---------------+          |   this.s    |
                          |   this.i    |
                          |   this.d    |
                          |   this.s =  |
                          |   this.d = }|
                          +-------------+
```
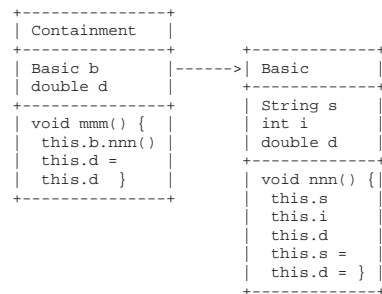
In addition, the relationship of *Containment-Basic* raises a second, entirely different question:

> If *mmm* in *Containment* is to represent changes to the state of your world (domain of interest), should *Basic* be stateful or not?

In section 26, we have seen both possibilities. Specifically, the first version of *BlockWorld* is imperative and contains an applicative version of *DrpBlock*; in the second version, *DrpBlock* itself is stateful and *BlockWorld* exploits this fact and is therefore only indirectly stateful (see figures 136 and 137).

The general answer depends on two factors. First, if someone else has designed *Basic*, you are handed an applicative or stateful class and must work with it. Even if you decide to extend *Basic* with a subclass—say to add a method—leave its state-related nature intact. Second, if you are designing the entire class (sub)system, you may choose whichever solution you wish. For now, you may just choose the design that you are comfortable with.

Unions

The third situation concerns unions of classes:

```
                          +-------------+
                          | IUnion      |
                          +-------------+
                          | void mmm()  |
                          +-------------+
                                 |
                                / \
                                ---
                                 |
              +-----------------+-----------------+
              |                 |                 |
     +-------------+    +-------------+    +-------------+
     | Basic1      |    | Basic2      |    | Basic3      |
     +-------------+    +-------------+    +-------------+
     | boolean b   |    | int i       |    | int i       |
     | double d    |    | double d    |    | String s    |
     +-------------+    +-------------+    +-------------+
     | void mmm(){ |    | void mmm(){ |    | void mmm(){ |
     |   this.b    |    |   this.i    |    |   this.i    |
     |   this.d    |    |   this.d    |    |   this.s    |
     |   this.b =  |    |   this.i =  |    |   this.i =  |
     |   this.d = }|    |   this.d = }|    |   this.s = }|
     +-------------+    +-------------+    +-------------+
```
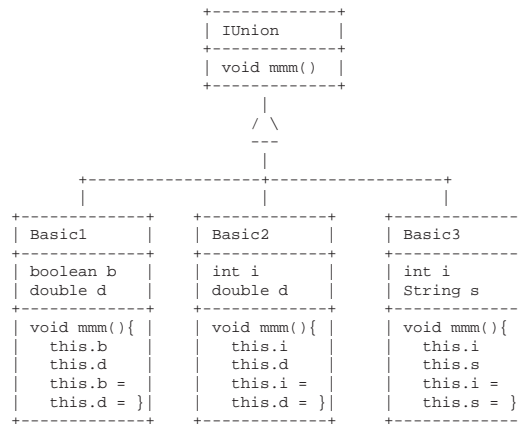
As before, the template for a union demands a method signature in the interface and template with sketches of bodies in concrete variants of the union. Just like for *Basic*, the method templates for the variants enumerate all fields of the respective classes and all changeable fields on the left side of assignment statements.

Self-References and Mutual References
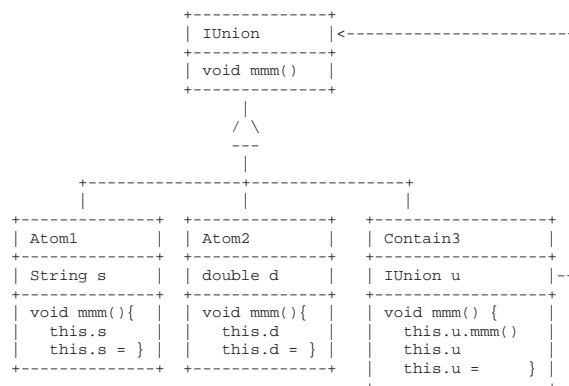
The last case is the most complex one, involving unions, containment, and self-reference all at once. Any other class organization, such as mutual references, is a composition of the four situations. Even though, this fourth situation is mostly just a composition of the preceding cases:

```
                          +--------------+
                          | IUnion       |<---------------------+
                          +--------------+                      |
                          | void mmm()   |                      |
                          +--------------+                      |
                                 |                              |
                                / \                             |
                                ---                             |
                                 |                              |
              +----------------+----------------+              |
              |                |                |              |
     +--------------+  +--------------+  +-----------------+   |
     | Atom1        |  | Atom2        |  | Contain3        |   |
     +--------------+  +--------------+  +-----------------+   |
     | String s     |  | double d     |  | IUnion u        |-+
     +--------------+  +--------------+  +-----------------+
     | void mmm(){  |  | void mmm(){  |  | void mmm() {    |
     |   this.s     |  |   this.d     |  |   this.u.mmm()  |
     |   this.s = } |  |   this.d = } |  |   this.u        |
     +--------------+  +--------------+  |   this.u =    } |
                                        +-----------------+
```

If you decide that a union must come with an imperative method, you add the method signature and purpose & effect statement to the interface and the method templates with bodies to the variants. The bodies of the templates contain the fields and assignments to the fields.

If you follow the advice for designing templates from the first three cases and strictly apply it to *Contain3*, you get three pieces in the template:

1. **this**.*u.mmm*() is the recursive method invocation that is due to the self-reference in the class diagram. Its purpose is to ask *u* to change its fields as needed.

2. **this**.*u* is a reminder that this value is available.

3. **this**.*u* = ... finally suggests that the method may change *u* itself.

Points 1 and 3 conflict to some extent. The method call changes fields in *u* and possibly fields that are contained in objects to which *u* refers. When **this**.*u* = ... changes what the field stands for, it may throw away all the computations and all the changes that the method invocation performed. Even though the use of both forms of assignment is feasible, it is rare and we won't pay much attention to the possibility in this book. The template contains both and for the method definition, you choose one or the other.

27.4 Imperative Methods and Abstraction

In chapter III, we discussed the principle of lifting common methods from the variants of a union to a common (abstract) superclass. Section 18.4 (in that part) presents an example that is impossible to deal with in this manner. Specifically, the program fragment consists of a union that represents falling stars and red rocks. The superclass contains methods for drawing such objects, for determining whether they are close to the ground and whether they have landed. The problem is, however, that the *move* methods are alike but their return types differ:

```
class Star extends Falling {          class RedRock extends Falling {
  ...                                   ...
   Star  drop() {                        RedRock  drop() {
    if (this.landed()) {                   if (this.landed()) {
      return this; }                         return this; }
    else                                   else
      ...                                    ...
}                                     }
```

Because of these differences it is impossible in Java to lift the otherwise similar methods to *Falling*, the common superclass.

```
class Star extends Falling {
  . . .
  // move this Star down
  // effect: change y
  void drop() {
    if (this.landed()) {
      return ; }
    else {
      . . . // change y
    }
  }
}
```

```
class RedRock extends Falling {
  . . .
  // move this RedRock down
  // effect: change y
  void drop() {
    if (this.landed()) {
      return ; }
    else {
      . . . // change y
    }
  }
}
```

```
class Falling {
  . . .
  // move this object down
  // effect: change y
  void drop() {
    if (this.landed()) {
      return ; }
    else {
      . . . // change y
    }
    . . .
  }
}
```

```
class Star extends Falling {
  . . .
}
```
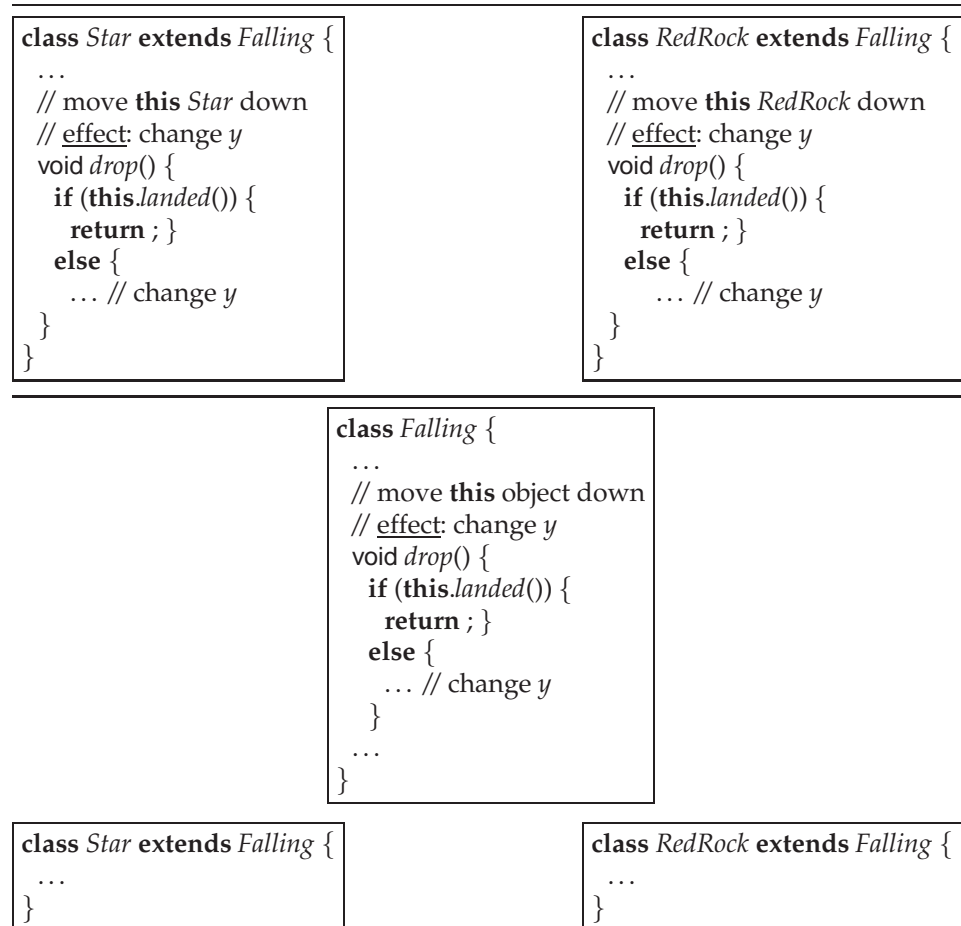
```
class RedRock extends Falling {
  . . .
}
```

Figure 138: imperative methods and abstraction

Now consider imperative versions of the two methods within stateful classes. In that case, the *move* methods would change the *y* field (and possibly the *x* field) and then return void. The top of figure 138 summarizes the situation. Since the methods are identical now, including the return type, it is possible to lift them to *Falling*. The bottom of the figure shows what the situation is like after the methods have been lifted.

In general, imperative methods remove one hurdle to abstraction: the return type. They use void and thus eliminate a potential difference from

two similar methods. Thus, when you choose to work with stateful classes and imperative methods, you may open additional opportunities for abstraction and you should use them.

27.5 Danger!

At first glance, stateful classes and imperative methods have three advantages over an applicative solution. First, they add expressive power; you can now directly represent any form of relationship among a collection of "real world" objects, even if they contain cycles. Second, they facilitate the representation of information that changes over time. Third, they enable additional opportunities for abstraction, at least in the context of Java and other popular object-oriented programming languages. Nothing comes for free, however, and that includes stateful classes and imperative methods. In this case, the very advantages can also turn into disadvantages and stumbling blocks.

```
// creating directly circular objects
class Node {
  private Node nxt = null;

  public Node() {
    this.nxt = this;
  }

  // visit each node connected to this one
  public int visit() {
    return this.nxt.visit();
  }
}
```

```
+-------------+
| Node        |<--+
+-------------+   |
| Node nxt    |---+
+-------------+
| int visit() |
+-------------+
```

Figure 139: A simple circular class

The first problem is about the creation of cycles of objects. In this part, we have illustrated why it is needed and how to create them with assignment statements. The examples made sense and were motivated by real-world applications. Take a look, however, at the class in figure 139. Creating an instance of this class and *visit*ing it has a surprising effect:

new *Node*().*visit*()

This expression has no result. If you solved exercises 23.8, you experienced this problem in a slightly different, almost realistic setting.

The true problem is that this *visit* method is well-designed according to the original design recipe. In particular, here is the template:

```
int visit() {
  ... this.nxt.visit() ...
}
```

Because the class contains a single field, the template body naturally consists of a single expression. Furthermore, because the field has the same type as the class, the design recipe requests an invocation of *visit*. The rest appears to be a straightforward step. If you designed your *length* method for exercise 23.8 in the same manner, you obtained a well-designed method for a union of classes that you used quite often.

We overcome this problem through the use of a modified design recipe. The major modification is to look out for those links in the class diagram that suggest auxiliary links for cyclic references. Cutting them—as suggested in figure 129, for example—restores order and guarantees that the original design recipe works. The step of cutting ties, however, isn't easy. It is most difficult when your task is to revise someone else's program; your predecessor may not have followed the design recipe. The class diagram for the classes might be extremely complex; then it is easy to overlook one of these links and to design a program that diverges. In short, with the introduction of assignments you are facing new problems when you test programs and when you are trying to eliminate errors.

The second disadvantage is due to the introduction of time and timing relationships into programming. While assignment statements make it easy to represent changes over time, they also force programmers to think about the timing of assignments *all the time*. More forcefully,

> assignment statements are universally destructive.

Once an assignment has changed the field of an object that change is visible everywhere and, in particular, the old value of the field is no longer accessible (through this field).

A traffic light simulation illustrates this point particularly well. Figure 140 (left, top) displays the three normal states of a traffic light in the normal order. Initially, the red light is on, the others are off, with traffic stopped.[57] Then the green light comes on, with red and yellow off, en-

---

[57]Turning on the red light in the initial state (or in a failure state) is the safest possible choice. Most physical systems are engineered in this manner.

```
    +-------+    #
    | World |    #
    +-------+    #
        |        #
       / \       #
       ---       #
        |        #
===================#
        |
+---------------------------------+
| TrafficLight                    |
+---------------------------------+    +------+
| private Bulb on                 |--+-->| Bulb |
| private Bulb next               |--+   +------+
| private Bulb third              |--+
+---------------------------------+
| public void onTick()            |
| public boolean draw()           |
| public void onKeyEvent(String k)|
+---------------------------------+
```

```
// a light bulb of a given
// a color at a given location
class Bulb {
  private int size;
  private IColor c;
  private Posn p;

  Bulb(int size,Posn p,IColor c) {
    this.size = size;
    this.c = c;
    this.p = p;
  }

  // turn this light bulb on
  public void on(Canvas c) {
    c.drawDisk(this.p,this.size,this.c);
    return ;
  }

  // turn this light bulb off
  public void off(Canvas c) {
    c.drawCircle(this.p,this.size,this.c);
    return ;
  }
}
```
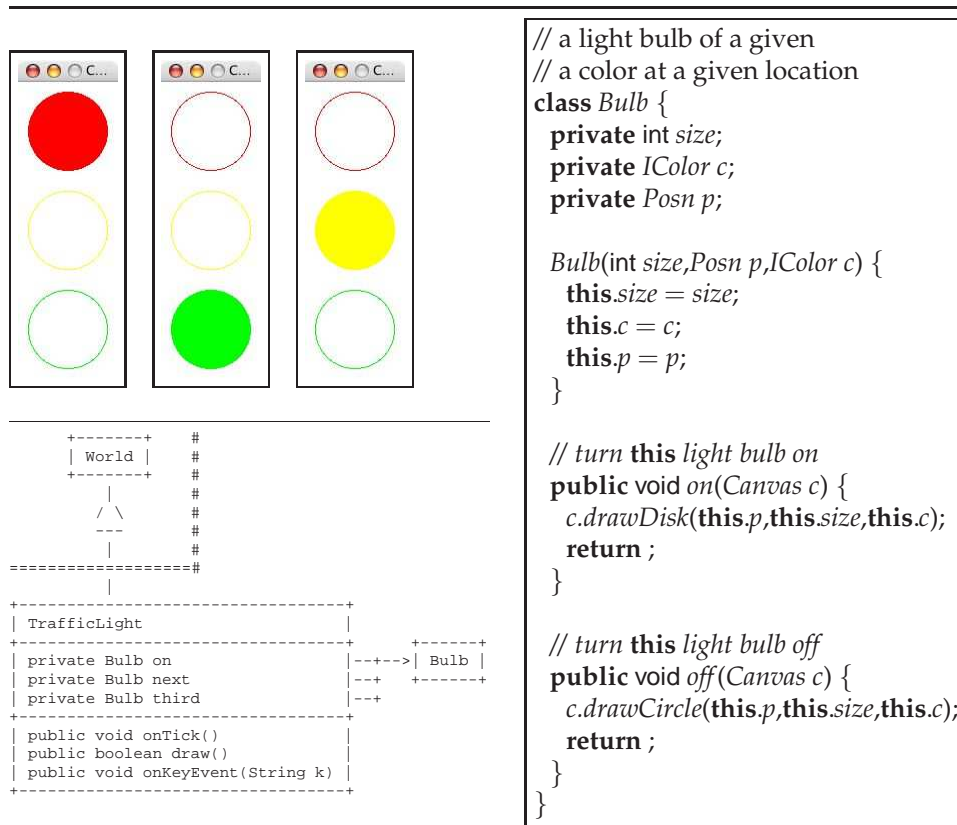
Figure 140: A simple circular class

abling the flow of traffic. Finally, the yellow light comes on to warn drivers of the upcoming switch to red. From here, the cycle repeats.[58]

The right side of the figure displays the *Bulb* class. Its three properties are: the size, the color, and the placement (on a canvas). The *on* and *off* methods consume a *Canvas* and draw a disk or a circle of the given size and at the specified position.

Now imagine that your manager asks you to design the *TrafficLight* class, which consists of three *Bulbs* and which switches colors every so often. Clearly, this class represents a little world of its own, with changes taking place every so often. Hence, you naturally design *TrafficLight* as an extension of *World* from **idraw**. This decision requires the design of three methods: *onTick*, *draw*, and *onKeyEvent*. Finally, the *TrafficLight* class has at

---

[58]For our purposes, we ignore the failure state in which the yellow or red light is blinking.

least three properties, namely the three bulbs: red, yellow, and green.

The class diagram on the bottom left of figure 140 summarize this data analysis. The three *Bulb*-typed fields are called *on*, *next*, and *third* are named to indicate which light is on, which comes next, and what the third one is. Roughly speaking, the three fields represent a three-way light switch. Obviously, the values of these fields must change over time to simulate the transitions in the state of a traffic light. To complete the data design part of the design recipe, determine appropriate initial values for these fields.

Let us now look at the design of *onTick*. Its purpose is to simulate a transition in the traffic light, that is, to rotate the roles of the three lights. To accomplish this rotation, the method affects the three *Bulb* fields:

> if *on* is a red bulb, *next* is green, and *third* is yellow, the method must change the fields so that *on* stands for green, *next* for yellow, and *third* for red.

When the *draw* method is invoked after *onTick* has finished its computation, it will turn on the green bulb and turn off the other two.

We can translate what we have into this method header and template:

inside of *TrafficLight* :
```
// a transition from one state of the light to another
// effect: change the values of on, next, and third
void onTick() {
  ... this.on ... this.next ... this.third ...
  this.on = ...
  this.next = ...
  this.third = ...
}
```

The first line reminds us that the method can use the values of the three fields for its computation and the last three lines suggest that it can assign new values to these three fields.

One apparently easy way to complete this template into a full definition is to just place the three fields in a different order on the right side of the assignment statements:
```
void onTick() {
  this.on = this.next;
  this.next = this.third;
  this.third = this.on;
  return ;
}
```

The informal description of the example suggests this idea. Unfortunately, this method definition doesn't work. To see how the method goes wrong, translate the above example into a test and invoke the *onTick* method once or twice. Then—and only then—read on.

after	*state of the object*
method entry	*on*: **new** *Bulb*(...,**new** *Red*()) *next*: **new** *Bulb*(...,**new** *Green*()) *third*: **new** *Bulb*(...,**new** *Yellow*())
this.*on* = **this**.*next*	*on*: **new** *Bulb*(...,**new** *Green*()) *next*: **new** *Bulb*(...,**new** *Green*()) *third*: **new** *Bulb*(...,**new** *Yellow*())
this.*next* = **this**.*third*	*on*: **new** *Bulb*(...,**new** *Green*()) *next*: **new** *Bulb*(...,**new** *Yellow*()) *third*: **new** *Bulb*(...,**new** *Yellow*())
this.*third* = **this**.*on*	*on*: **new** *Bulb*(...,**new** *Green*()) *next*: **new** *Bulb*(...,**new** *Yellow*()) *third*: **new** *Bulb*(...,**new** *Green*())

Figure 141: Assignment sequencing

The problem is that the first assignment statement

this.*on* = **this**.*next*;

"destroys" what *on* stand for. From now on (until the next assignment statement), the *on* field stands for the current value of the *next* field; the old value

is no longer accessible. Hence, the last assignment statement in *onTick*

>   **this**.*third* = **this**.*on*;

also forces the *third* field to stand for the value that *next* represented when
the method was called.

A graphical presentation with box diagrams for objects explains the
problem in a concise manner: see figure 141. The table shows the state
of the object in the first row and its state after each of the three assignment
statements thereafter. The first assignment uses the current value of *next*,
which is a green light bulb, and sticks it into the compartment for *on*. The
second one places the yellow bulb from *third* into *next*. And the last one
uses the current value of *on*, which is now the green bulb, to change the
value of the last compartment. The end effect is that two of the fields stand
for the green bulb, one for the yellow one, and the red one is gone.

More generally, you should notice that when we use assignments and
explain their workings, we use words that refer to points in time. For ex-
ample, we no longer just use "the value" of a field but "the *current* value."
Assignments are about a time, and they introduce time into programming.
As we just saw, a simple, natural-looking slip drops a value from our world.

To overcome this problem, we use local variables. Recall that in Java,
local variable definitions take the shape of field declarations. Their primary
use here is to remember the values of fields that the method is about to
change. In other words, even though the method changes what the field
stands for, the local variable still stands for the current value of the field and
can thus use it after the assignment statement is evaluated. In particular,
onTick can use such a local variable to temporarily remember the value of
this.*on*:

>   inside of *TrafficLight* :
>   ‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾
>   // a transition from one state of the light to another
>   // <u>effect</u>: change the values of *on*, *next*, and *third*
>   void *onTick*() {
>   *Bulb tmp* = **this**.*on*;
>
>   **this**.*on* = **this**.*next*;
>   **this**.*next* = **this**.*third*;
>   **this**.*third* = *tmp*;
>   }

Exercises

Exercise 27.1 Finish the design of the *TrafficLight* class. Run the traffic light simulation. Would it make sense for *Bulb* objects to change color? ∎

Exercise 27.2 Design an applicative version using **draw**. Compare and contrast this solution with the one from exercise 27.1. ∎

The third disadvantage concerns the information content of method signatures. The preceding section sketched how the use of void return types increases the potential for abstraction. The reason is that methods that returned an instance of the class to which they belong may return void now and may therefore look like another method in the same union.

Of course, this very fact is also a disadvantage. As this book has explained at many places, types in signatures tell the readers of the code what and how the method communicates with its calling context. In the case of void, we know that it does *not* communicate any values; it just changes some fields. Worse, the language implementation (ProfessorJ or whatever you use) checks types before it runs your program and points out (type) errors. When the return type is void and there is nothing to check. In short, the use of void disables even the minimal checks that type checkers provide. You will notice this lack of basic checking when your programs become really large or when you collaborate on large projects with other people.

The overall lesson is one of compromise. Assignments are good for the purpose of representing things that change over time. So use them, but use them with care and keep their disadvantages and dangers in mind.

27.6 Case Study: More on Bank Accounts

Figure 134 compares an applicative representation of checking accounts with an imperative one. Let us consider a simple-looking extension of the original problem statement:

> ... Equip the *Account* class with a method for transfering funds from **this** account to some other account. In other words, the method simultaneously withdraws a given amount from **this** account and deposits it in another one. ...

As you may know from your own first banking experiences, this kind of action is available at most banks (where a teller performs it for you) and via their Internet banking services.

The purpose of this section is twofold. On one hand, it illustrates the design recipe for imperative methods with a simple example. On the other hand, it demonstrates that an applicative design may require a more complex design process than an imperative design.

Let's start with the stateful class, the case that is easy and natural:

1. The signature and the purpose statement for the imperative method in the stateful version of *Account* are straightforward:

   > inside of *Account* :
   > // move the amount *a* from **this** account into the *other* account
   > // <u>effect</u>: change the *amount* field in **this** and *other*
   > void *transfer*(int *a*,*Account other*)

2. Here is an example for *transfer*'s desired behavior:

   > *Account a* = **new** *Account*(`"Matthew 1"`,100);
   > *Account b* = **new** *Account*(`"Matthew 2"`,200);
   >
   > *b.transfer*(100,*a*);
   >
   > **check** *a.balance*() **expect** `"Matthew 1: 200"`;
   > **check** *b.balance*() **expect** `"Matthew 2: 100"`;

   That is, after the transfer the target account (*a*) should have an increased balance and the balance of **this** account, which is *b*, should have decreased by an equal amount.

3. The template of *transfer* enumerates all the fields and methods of the two given accounts and the fields that may be on the left of an assignment statement:

   > inside of *Account* :
   > // move the amount *a* from **this** account into the *other* account
   > // <u>effect</u>: change the *amount* field in **this** and *other*
   > void *transfer*(int *a*,*Account other*) {
   >   ... **this**.*amount* ... **this**.*owner* ...
   >   ... *other. lll*() ... *other.amount* ...
   >   **this**.*amount* =
   >   *other. amount* =
   > }

Before we move on, recall that a method definition may always use methods from the class itself; here, this means *deposit* and *withdraw*.

4. Picking up this last idea suggests an intuitive definition for *transfer*:

   inside of *Account* :
   // move the amount *a* from **this** account into the *other* account
   // effect: change the *amount* field in **this** and *other*
   void *transfer*(int *a*,*Account other*) {
     this.withdraw(*a*);
     other.deposit(*a*);
     return ;
   }

   First the amount *a* is withdrawn from **this** account. Second, *a* is deposited in the *other* account. After that, the method returns void.

Exercises

Exercise 27.3 The design of *transfer* for the stateful *Account* class omits the last step. Turn the example into a test and run the tests. ∎

Exercise 27.4 According to the template, the *transfer* method could also directly assign new values to **this**.*amount* and *other.amount*. Define the transfer method in this way and compare it to the above definition. Which version do you think expresses the purpose statement better? ∎

For the addition of *transfer* to the applicative version of *Account*, we also use the design recipe but even the first step poses a problem. While it is clear that the arguments for *transfer* remain the same—*a*, the amount, and *other*, the target account—it is not at all obvious what the method returns:

inside of *Account* :
// move the amount *a* from **this** account into the *other* account
??? transfer(int *a*,*Account other*)

Usually a method that computes a change to an object (due to some event) returns a new instance of the same class with the fields initialized appropriately. As we know from the design of the imperative *transfer* method, however, the purpose of transfer is to change *two* objects simultaneously:

```
// a pair of accounts
class PairAccounts {
    Account first;
    Account second;

    PairAccounts(Account first, Account second) {
        this.first = first;
        this.second = second;
    }
}
```

```
+----------------+
|  PairAccounts  |
+----------------+
| Account first  |
| Account second |
+----------------+
```

Figure 142: Pairs of accounts

this and *other*. Hence, the applicative version of transfer must produce *two* instance of *Account*.

Unfortunately, methods in Java can only produce a single value,[59] we must somehow combine the two new objects into one. In other words, the systematic design of the method forces us to introduce a new form of data: pairs of *Account*s. Figure 142 shows the rather standard design.

From here, the rest of the design follows from the design recipe:

1. Here is the complete method header:

   inside of *Account (applicative version)* :
   // move the amount *a* from **this** account into the *other* account
   PairAccounts transfer(int *a*,*Account other*)

2. The adaptation of the above example requires very little work:

   Account a = **new** *Account*("Matthew 1",100);
   Account b = **new** *Account*("Matthew 2",200);

   check *b.transfer*(100,*a*) **expect**
     new *PairAccounts*(**new** *Account*("Matthew 1",200),
                                new *Account*("Matthew 2",100))

   The advantage of the applicative form of the example is that everything is obvious: the names of the owners, the associated amounts, and how they changed. Of course, the example is also overly complex compared with the imperative one.

---

[59] In Scheme and Common Lisp, functions may return many values simultaneously.

3. For the template we can almost reuse the one from the imperative design, though we need to drop the assignment statements:

> inside of *Account (applicative version)* :
> // move the amount *a* from **this** account into the *other* account
> *PairAccounts transfer*(int *a,Account other*) {
>   ... **this**.*amount* ... **this**.*owner* ...
>   ... *other. lll*() ... *other.amount* ...
> }

4. The actual definition combines the example and the template:

> inside of *Account (applicative version)* :
> // move the amount *a* from **this** account into the *other* account
> *PairAccounts transfer*(int *a*, *Account other*) {
>   **return new** *PairAccounts*(*this.withdraw*(*a*),*other.deposit*(*a*));
> }

5. You should turn the example into a test and confirm that the method works properly.

As you can see from this simple example, an applicative design can easily produce a rather complex method. Here the complexity has two aspects. First, the method design requires the design of (yet) another class of data. Second, the method must construct a complex piece of data to return a combination of several objects. Here we could avoid this complexity by designing the method from the perspective of the bank as a whole, changing several (presumably existing) classes at once. In general though, these other classes are not under your control, and thus, applicative approaches can lead to rather contorted designs.

27.7   Case Study Repeated: A Stateful Approach to UFOs

The "War of the Worlds" game program is a good case study for a conversion of a complete applicative design into an imperative one. At the same time, we can test the design guideline in the context of a reasonably large, complete program. As figure 143 reminds us, even the simplest version consists of six classes and an interface, plus a couple of dozen methods. Furthermore, it uses the *World* class from the **draw** package, which means we can use the borderline between the library and the *UFOWorld* class as the starting point for our design investigation.

```
                                      +-----------------------+
+------------------+ #  +------------->| IShots                |<-------------------+
| World            | #  |              +-----------------------+                    |
+------------------+ #  |              | IShots move()         |                    |
| Canvas theCanvas | #  |              | boolean draw(Canvas c)|                    |
+------------------+ #  |              | boolean hit(UFO u)    |                    |
| World onTick()   | #  |              +-----------------------+                    |
| World onKeyEvent(| #  |                          |                                |
|  String ke)      | #  |                         / \                               |
| World draw()     | #  |                         ---                               |
+------------------+ #  |                          |                                |
                    #  |              -------------------------------               |
          |         #  |              |                             |               |
         / \        #  |   +---------------------+     +---------------------+      |
         ---        #  |   | MtShots             |     | ConsShots           |      |
          |         #  |   +---------------------+     +---------------------+      |
========================= |                           | IShots rest         |----+ |
          |             |                             | Shot first          |----+ |
          |             |                             +---------------------+    | |
          |             |                                                        | |
+------------------+    |                                                        v |
| UFOWorld         |    |              +-----------------------+     +---------------------+
+------------------+    | +--------->| UFO                   |     | Shot                |
| int WIDTH        |    | | |          +-----------------------+     +---------------------+
| int HEIGHT       |    | | |          | IColor colorUFO       |     | IColor colorShot    |
| IColor BACKG     |    | | |          | Posn location         |     | Posn location       |
| UFO ufo          |----|---+          +-----------------------+     +---------------------+
| AUP aup          |----|---+          | UFO move()            |     | Shot move()         |
| IShots shots     |----+ |            | boolean draw(Canvas c)|     | boolean draw(Canvas c)|
+------------------+    |              | boolean landed()      |     | boolean hit(UFO u)  |
| World onTick()   |    |              | boolean isHit(Posn s) |     +---------------------+
| World onKeyEvent(|    |              +-----------------------+
|  String ke)      |    |
| boolean draw()   |    |              +-----------------------+
| Shot shoot()     |    | +--------->| AUP                   |
| World move()     |    |              +-----------------------+
+------------------+    |              | IColor aupColor       |
                       |              | int location          |
                       |              +-----------------------+
                       |              | AUP move()            |
                       |              | boolean draw(Canvas c)|
                       |              | Shot fireShot()       |
                       |              +-----------------------+
```
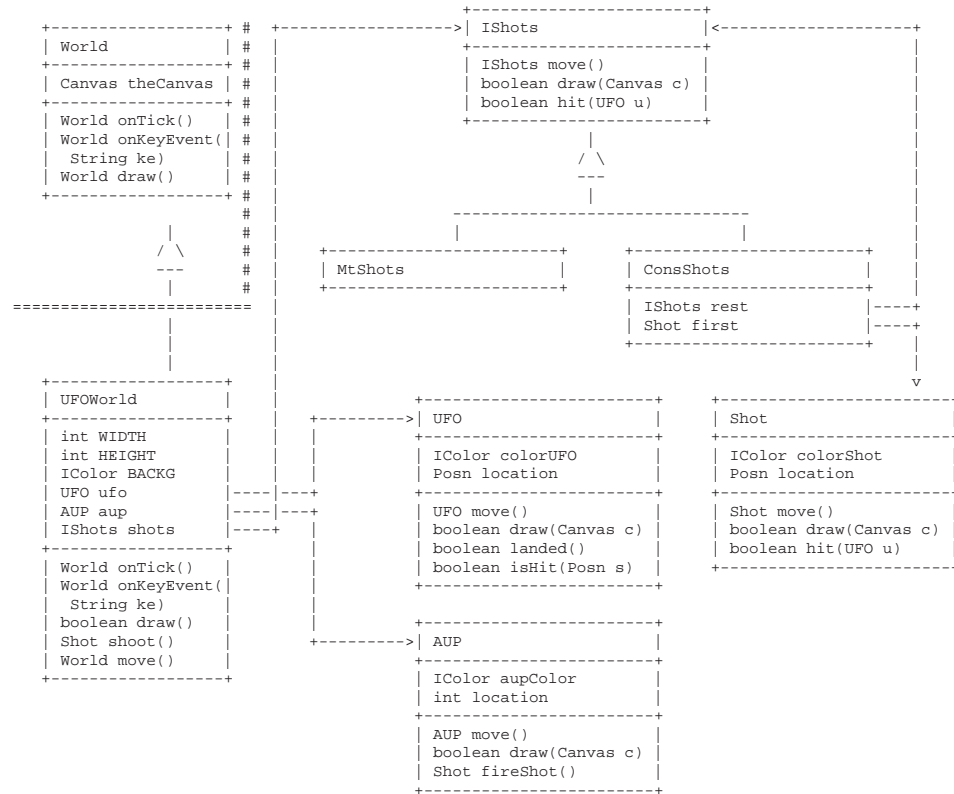
Figure 143: The Revised World of UFOs: Class diagrams

If we import *World* from the **idraw** package instead of the **draw** package, we are immediately confronted with three "type" problems: the overriding definitions of *onTick*, *onKeyEvent*, and *draw* in *UFOWorld* must now produce void results, not *World*s. In particular, this suggests that *onTick* and *onKeyEvent* can no longer produce new instances of the world, but must modify **this** instance of *UFOWorld* to keep track of changes over time.

Let's deal with each method in turn, starting with *draw*:

inside of *UFOWorld (imperative version)* :

```
// draw this world
void draw() {
  this.ufo.draw(this.theCanvas);
  this.aup.draw(this.theCanvas,this.HEIGHT);
  this.shots.draw(this.theCanvas);
  return ;
}
```

Instead of combining the drawing methods with *&&*, the imperative version uses ";" (semicolon), which implies that *draw* in *UFO*, *AUP*, and *IShots* now produce void, too. Finally, *draw* returns void.

Exercise

Exercise 27.5 Modify the methods definitions of *draw* in *UFO*, *AUP*, and *IShots* so that they use the imperative methods from *Canavs* in **idraw**. ∎

The modifications concerning *onTick* are more interesting than that:

inside of *UFOWorld (applicative)* :

```
// move the ufo and the shots of
// this world on every clock tick
World onTick() {
 if (this.ufo.landed(this)) {
  return this.endOfWorld("You lost."); }
 else { if (this.shots.hit(this.ufo)) {
  return this.endOfWorld("You won."); }
 else {
  return this.move(); }
 }
}
```

inside of *UFOWorld (imperative)* :

```
// move the ufo and the shots of
// this world on every clock tick
void onTick() {
 if (this.ufo.landedP(this)) {
  this.endOfWorld("You lost."); }
 else { if (this.shots.hit(this.ufo)) {
  this.endOfWorld("You won."); }
 else {
  this.move(); }
 return ;
 }
}
```

The method first checks whether the UFO has landed, then whether the UFO has been hit by any of the shots. In either case, the game is over. If neither is the case, however, the method moves every object in the world, using a local *move* method. Clearly, this part must change so that *move* no longer produces a new *UFOWorld* but changes **this** one instead. Hence, the imperative version of *onTick* on the right side doesn't look too different from the applicative one. The true code modifications are those in *move*.

Turning to the *move* method in *UFOWorld*, we see that its purpose is to move the UFO and all the other objects in the world so that they show up somewhere different the next time the canvas is redrawn. This naturally raises the question whether, say, the *UFO* class and the representation of lists of shots remain applicative or should be reformulated as stateful classes, too. *The choice is ours*; the line between **idraw**'s *World* and *UFOWorld* does not force either one of them.

To make things simple, we ignore the shots for a moment and focus on a world that contains only a UFO. Then the two choices—an applicative or stateful version of the *UFO* class—induce the following two feasible definitions of *move* in *UFOWorld*:

inside of *UFOWorld* :	inside of *UFOWorld* :
// move the UFO and shots	// move the UFO and shots
void *move*() {	void *move*() {
this.*ufo.move*(**this**);	**this**.*ufo* = **this**.*ufo.move*(**this**);
. . .	. . .
return ;	**return** ;
}	}

The one on the left assumes that *UFO* is stateful and its *move* method is imperative. The definition on the right assumes that *UFO* is applicative and that therefore its *move* method produces a new instance of *UFO*, which *UFOWorld* keeps around until the next tick of the clock.

Now it is up to you to make a decision of how to design *UFO*. You know from the applicative design (or from reading the problem statement) that you need at least four methods for *UFO*:

1. *move*, which moves **this** UFO for every clock tick;

2. *draw*, which draws **this** UFO into a given *Canvas*;

3. *landed*, which determines whether the UFO is close to the ground; and

4. *isHit*, which determines whether the UFO has been hit by a shot.

According to our design principle, you should consider whether calling some method influences the computations (results) of some other method in the class. After a moment of reflection, it is clear that every invocation of *move* changes what *draw*, *landed*, and *isHit* compute. First, where *draw* places the UFO on the canvas depends on how often *move* was called starting with *bigBang*. Second, whether or not the UFO has landed depends on

```
// a UFO, whose center is located at location
class UFO {
  Posn location;
  IColor colorUFO = new Green();

  UFO(Posn location) {
    this.location = location;
  }

  // move this UFO down by 3 pixels
  // effect: change the location field in this UFO
  void move(WorldUFO w) {
    if (this.landed(w)) // don't move anymore, it's on the ground {
      return ; }
    else {
      this.location = new Posn(this.location.x,this.location.y+3);
      return ;
    }
  }

  // has this UFO landed yet?
  boolean landed(WorldUFO w) {
    return this.location.y >= w.HEIGHT − 3;
  }

  // is this UFO close to the given Posn?
  boolean isHit(Posn x) {
    return this.distanceTo(x) <= 10.0;
  }

  // compute distance between l and this.location
  double distanceTo(Posn l) { … } // belongs to Location

  // draw this UFO into the Canvas
  void draw(Canvas c) {
    c.drawDisk(this.location,10,this.colorUFO);
    c.drawRect(new Posn(this.location.x-30,this.location.y-2),60,4,this.colorUFO);
    return ;
  }
}
```

Figure 144: The imperative *UFO* class

how far the UFO has descended, i.e., how often *move* has been called. Finally, the case for *isHit* is ambiguous. Since a shot moves on its own, the UFO doesn't have to move to get hit; then again, as the UFO *move*s, its chances for being hit by a shot change.

Our analysis suggests that designing *UFO* as a stateful class with *move* as an imperative method is a reasonable choice. It is also clear that *move* changes the *location* field of the class. Every time it is called it recomputes where the UFO ought to be and assigns this new *Posn* to *location*.

Figure 144 displays the stateful version of *UFO*. The key method is *move*. Its effect statement informs us that it may change the value of the *location* field. The method body consists of an **if**-statement that tests whether the *UFO* has landed. If so, it does nothing; otherwise it computes the next *Posn* and assigns it to *location*. Both *landed* and *isHit* use the current value of *location* to compute their answers. Because *location*'s value continuously changes with every clock tick, so do their answers. Finally, *draw* is imperative because we are using the **idraw** library now.

Exercises

Exercise 27.6 The definition of *landed* contains an invocation of *distanceTo*. Design the method *distanceTo* and thus complete the definition of *UFO*. Is *distanceTo* truly a method of *UFO*? Where would you place the definition in figure 98 (see page 288)? ∎

Exercise 27.7 Develop examples for the *move* method in *UFO*. Reformulate them as tests. ∎

Exercise 27.8 Develop examples and tests for *landed* and *isHit* in *UFO*. ∎

Exercise 27.9 Design two imperative versions of *shoot* in *UFOWorld*: one should work with a stateful version of *AUP* and another one that works with the existing functional version of *AUP*. Also design the stateful version of the *AUP* class. ∎

With the movement of *UFO*s under control, let's turn to the collection of shots and study whether we should make it stateful or not. The *IShots* interface specifies three method signatures: *move*, *draw*, and *hit*. By analogy, you may want to make the first imperative. The second is imperative because the **idraw** library provides imperative *draw* methods. The question

is whether repeated calls to move affect the *hit* method, and the answer is definitely "yes." As the shots move, they distance to the *UFO* (usually) changes and thus *hit* may produce a different answer.

Still, you should ask what exactly is stateful about moving all the shots. After all, it is obvious that moving the shots doesn't change how many shots there are. It also shouldn't change in what order the shots appear on the list. With this in mind, let's follow the design recipe and see what happens. Step 1 calls for a full signature, purpose and possibly effect statement:

> <u>inside of *IShots*</u> :
> // move the shots on **this** list; <u>effect</u>: fill in later
> void *move*()

Since an interface doesn't have fields, it is impossible to specify potential changes to fields. For step 2, we consider a two-item list of shots:

> *Shot s1* = **new** *Shot*(**new** *Posn*(10,20));
> *Shot s2* = **new** *Shot*(**new** *Posn*(15,30));
> *IShots los* = **new** *ConsShots*(*s1*,**new** *ConsShots*(*s2*,**new** *MtShots*()));
>
>
> *los.move*();
>
>
> **check** *s1.location* **expect new** *Posn*(10,17)
> **check** *s2.location* **expect new** *Posn*(15,27)

So far the example says that each of the shots on the list must change locations after the *move* method for *IShots* is invoked. It doesn't reflect our thinking above that *los* per se doesn't change. To express this, you could add the following line:

> **check** *los* **expect**
>   **new** *ConsShots*(**new** *Posn*(10,17),
>   **new** *ConsShots*(**new** *Posn*(15,27),
>   **new** *MtShots*()));

Of course, this example subsumes the ones for *s1* and *s2*.

The *IShots* interface is the union type for two variants: *MtShots* and *ConsShots*, which means the template looks as follows:

inside of *MtShots* :   inside of *ConsShots* :
// move these shots   // move these shots
void *move*() {   void *move*() {
 ...   ... **this**.*first.move*() ... **this**.*rest.move*() ...
}   ... **this**.*first* = ...
                                           ... **this**.*rest* = ...
                                        }

The template for *ConsShots* looks complex. It reminds us that the method
can use **this**.*first*, can invoke a method on **this**.*first*, and that the natural
recursion completes the process. The two skeleton assignments suggest
that the method could change the value of the *first* and *rest* fields.

Put differently, the method template empowers us again to express a
design decision:

Should we use a stateful representation of lists of shots?

If the representation is stateful, the assignment for **this**.*first* changes which
Shot the first field represents. If the representation remains applicative,
however, the method must demand that the invocation **this**.*first.move*()
changes *Shot*—which you have to add to your wish list or communicate
to the person who designs the new *Shot* class for you.

As discussed above, we don't think of the list per se as something that
changes. For this reason, we stick to an applicative class with an imperative
version of *move*, knowing that the imperativeness is due to the imperative
nature of *move* in *UFOWorld* and requires an imperative version of move in
Shot. Based on the examples and the template, we arrive at the following
two definitions:

        inside of *MtShots* :   inside of *ConsShots* :
        void *move*() {   void *move*() {
          return ;   **this**.*first.move*();
        }   **this**.*rest.move*();
                                                  return ;
                                                }

That is, the *move* method in *MtShots* does nothing and the one in *ConsShot*
invokes an imperative *move* method in *Shot*. The design of *move* thus adds
an imperative *move* method for *Shot* on the wish list, re-confirming that *Shot*
is stateful. At the same time, we can now remove the effect statement for
move in *IShots*.

The first step in the design of this final *move* method is to write down
its signature and purpose and effect statement:

inside of *Shot* :
// move **this** *Shot*
// <u>effect</u>: change *location*
void *move*()

The effect statement states the obvious: when an object move, its location changes. Since the example for moving a list of shots covers the *move* method for *Shot*, too, we go right on to the template:

```
void move() {
  ... this.location ... this.colorShot ...
   this.location = ...
}
```

The rest is easy and produces a method definition quite similar to the one in *UFO* (see figure 144).

Exercises

Exercise 27.10 Complete the definition of *move* in *Shot* and in *UFOWorld*. Create examples for the latter, and turn all the examples into tests. ∎

Exercise 27.11 Complete the definitions of *draw* and *hit* in the *MtShots*, *ConsShots*, and *Shot* classes. If you have done all the preceding exercises in this section, you now have a complete "War of the Worlds" game and you're ready to play. ∎

Exercise 27.12 Revise figure 143 so that it is an accurate documentation of the imperative version of the program.

Equip all pieces of all classes with privacy specifications. ∎

Exercise 27.13 Design a solution to the "War of the Worlds" game that relies on a stateful version of the list of shots and keeps the *Shot* class applicative. Compare and contrast your solution with the one above. ∎

Exercise 27.14 Modify the design of the *AUP* class so that the AUP moves continuously, just like the UFO and the shots, i.e., it acts like a realistic vehicle. Let the design recipe guide you. Compare with exercise 19.19. ∎

Exercise 27.15 Design UFOs that can defend themselves. The UFO should drop charges on a random basis and, if one of these charges hits the AUP, the player should lose the game. A charge should descend at twice the speed of the UFO, straight down from where it has been released. Compare with exercise 19.20. ∎

Exercise 27.16 Re-design the Worm game from section 19.9 in an imperative manner, using the guidelines from this chapter. ∎

// a Squadron Scramble playing card
interface *ICard* { }
	// your implementation for testing
	class *Card* **implements** *ICard* {
	  String kind;
	  int *view*;
	
	  Card(*String kind*, int *view*) {
	    this.*kind* = *kind*;
	    this.*view* = *view*;
	  }
	}

Figure 145: A game card

27.8   Case Study: A Deck of Cards

Simulated card games are popular on all kinds of computers. Your company has decided to develop a software version of "Squadron Scramble,"[60] because the marketing department has found no on-line competitors. Your boss has decided to assign you the task of designing the card deck. She has simply marked the relevant portions of the game instructions and declared this your problem statement:

> ... After the cards are distributed, the game administrator puts down one more card, face up, in the middle of the table. This starts a deck of cards, which players can use for swapping playing cards.
>
> ...
>
> For each turn, the player may choose to get a new card from the game administrator or may take as many cards as he desires from the top of the deck. The player then incorporates the(se) card(s) into his hand (according to some strategy). At the end of the turn, the players must put one card from his hand back on the deck. ...

---

[60]The game, published by US Game Systems, resembles Rummy but uses cards labeled with Axis and Allies World War II war planes.

You are told to produce an implementation of decks of cards while follow-ing the guidelines of this book. Someone else has already been asked to produce an implementation of playing cards, but since the actual nature of cards don't really matter to decks, you are to use the *ICard* interface and stub class from figure 145.[61]

The first part of the design recipe is about the identification of the dif-ferent kinds of information that play a role in your problem statement. For your problem—the simulation of a deck of cards—there appears to be just one class of information: the deck of cards. Without any further informa-tion, we can think of the deck of cards as a list of cards, especially since the sequence matters according to the problem statement. Given how routine the design of a list is, we just assume the existence of *ICards*, *MtCards* and *ConsCards* for now, without looking at the details at all.

As you now consider whether the deck of cards should be applicative or a stateful, you must take a look at the methods that you might need. From the problem statement, you know that the game program creates the deck and that (the program piece representing) the players can do three things to the deck afterwards:

1. put a card back on top of the deck, and

2. remove a number of cards from the top of the deck,

3. look at the first card of the deck.

Of course, the second item implies that a player can (and must be able to) count how many cards there are on the deck, so we add this method as a fourth one:

4. count the number of cards in a deck.

The following names for these four methods offer themselves naturally: *put*, *take*, *look*, and *count*.

The question is whether calling one of these methods influences the computation or the result of any of the other. Consider *put*, which con-sumes an *ICard* and adds it to the top of a deck. Of course this should change the result of *look*, which shows the top-most *ICard* of a deck. And it should also change the result of *count*, which computes how many cards are in a deck. The case of *take* is similar. Its task is to remove cards from the deck, and naturally, this kind of action changes what the top-most card is and how many cards there are in the deck of cards.

---

[61]That way the two of you can proceed in parallel.

Problem is that there are no obvious fields in a list that can represent the changes that *put* and *take* are to compute. First, neither *ICards* nor *MtCards* have fields. Second, the fields in *ConsCards* are to represent the first *ICard* and the ones below the first. When a player *put*s a card on top of a deck, **this** list doesn't change, only the deck that represents the cards. Put differently a card is added to the front of the list with **new** *ConsCards*(...) and somehow the deck changes. Similarly, when a player *take*s some cards, it is not that *first* or *rest* should change; only the deck changes.

Our analysis thus produced two insights. On one hand, a deck of cards needs a list of cards. On the other hand, a deck isn't just a list of cards. Instead, a deck should consist of a list of cards and the above methods. In short, *Deck* is a class with one assignable field—*content*—and four methods, two of which—*put* and *take*—may assign to this field.

```
class Deck {
  ICards content ... ;
  Deck(...) { ... }
  // place one card on top of this deck
  // effect: change content to include the
  // given card
  void put(ICard t) { ... }

  // take the first i cards from this deck
  // effect: change content to indicate the
  // removal of the cards
  ICards take(int i) { ... }

  // look at the top card on this deck
  ICard look() { ... }

  // how deep is this deck
  int count() { ... }
}
```

```
+-------------------+
| Deck              |
+-------------------+             +--------+
| ICards content    |------->| ICards |
+-------------------+             +--------+
| void put(Card c)  |                 |
| ICards take(int n)|               / \
| int count()       |               ---
| ICard look()      |                 |
+-------------------+               / \
                                   /   \
                                  /     \
                                 .........
```

Figure 146: A game card

Figure 146 summarizes our discussion with a partial class definition and diagram for *Deck*. The latter also indicates how the class is connected to the rest of our work. The figure also turns the above descriptions of the methods into purpose and effect statements plus contracts. Before we design the methods, however, we must finish the design of *Deck*. So far we don't even have a constructor for *Deck*, i.e., we can't create data examples.

Therefore the next step is to inspect the problem statement for an example of a "real world" deck of cards. As it turns out, the very first sentence explains how the deck comes about or, in programming terminology, how it is created. The game administrator places a single card on the table, which starts the deck. This suggests that *Deck*'s constructor consume a single card and from this creates the deck:

inside of *Deck* :
Deck(ICard c) {
  this.*content* = **new** *ConsCards(c,***new** *MtCards()*);
}

That is, the constructor creates a list from the given card and uses this list to initialize the *content* field.

Discussion: Given this constructor for *Deck*, you may be wondering whether we shouldn't design a data representation for non-empty lists. Take a second look at the problem statement, however. When it is a player's turn, one action is to take *all* cards from the deck. This implies that *during* a turn, the deck could be temporarily empty. It is only at the end of the turn that the deck is guaranteed to be non-empty. ∎

Using this constructor, you can now create sample objects for *Deck*:

new *Deck*(**new** *Card*("Boeing B-17 (Flying Fortress)",2))

In the actual game, this would be a card showing the second view of the Flying Fortress airplane. Unfortunately, the natural constructor doesn't allow us to build a representation of a deck of many cards. To do so, we need to design *put*, whose purpose it is to put several cards to atop the deck:

ICard c1 = **new** *Card*("Boeing B-17 (Flying Fortress)",2);
ICard c2 = **new** *Card*("Boeing B-29 (Superfortress)",1);
Deck d1 = **new** *Deck(c1)*;

d1.put(c2)

check *d1.content*
expect new *ConsCards(c2,***new** *ConsCards(c1,***new** *MtCards())*)

This behavioral example constructs two cards, uses one to construct a *Deck*, *put*s the other one on top of the first, and then ensures that this is true with a look at the *content* field. If *content* were **private**, you could use *look* instead to inspect the deck:

check *d1.look()* **expect** *c2*

After all, it is its purpose to uncover the top-most card, and in *d1*, the top-most card is the last one added.

The reformulation of the example appears to suggest a switch to the design of *look* so that we can run tests as soon as possible. While this is an acceptable strategy in some cases, this particular case doesn't require a consideration of the auxiliary method. The template of *put* is straightforward. It consists of **this**.*content.mmm*() and a skeleton assignment to *content*:

    inside of *Deck* :
    void *put*(ICard c) {
      ... **this**.*content.mmm*() ...
       this.*content* = ...
    }

The former reminds us that the value of *content* is available for the computation and that we may wish to define an auxiliary method in *ICards*; the latter reminds us that the method may assign a new value to the field.

Since the goal is to add *c* to the front of the list, the actual method just creates a list from it and **this**.*content* and assigns the result to *content*:

    inside of *Deck* :
    void *put*(ICard c) {
       this. *content* = **new** *ConsCards*(*t*,**this**. *content*);
       return ;
    }

No new wish for our wish list is required.

Equipped with *put*, you can now create a data representation of any actual deck of cards. In other words, the design recipe for a data representation and the design recipe for methods are intertwined here. Furthermore, the design of *put* implicitly created a behavioral example for *look*.

The template for *look* is like the one for *put*, minus the assignment:

    inside of *Deck* :
    ICard *look*() {
      ... **this**.*content.mmm*() ...
    }

Based on the examples, *look* is to extract the first card from the list. Hence, we add a *fst* method[62] to the wish list for *ICards* and define *look* as follows:

---

[62]In Java proper, we could name the method *first*. Because *ConsCards* implements *ICards*, the class must then include both a field called *first* and a method called *first*. Java distinguishes methods and fields and can thus reason separately about those and can (usually) disambiguate between the two kinds of references.

<u>inside of *Deck* :</u>
ICard look() {
  return this.*content.fst*()
}

Judging from *count*'s purpose statement in figure 146, the first data example is easily turned into a behavioral example:

ICard c1 = **new** *Card*(`"Boeing B-17 (Flying Fortress)"`,2);
Deck d1 = **new** *Deck*(*c1*)

check *d1.count*() **expect** 1

The other example from above is also easily modified for *count*:

ICard c1 = **new** *Card*(`"Boeing B-17 (Flying Fortress)"`,2);
ICard c2 = **new** *Card*(`"Boeing B-29 (Superfortress)"`,1);
Deck d1 = **new** *Deck*(*c1*);

d1.put(*c2*)

check *d1.count*() **expect** 2

Not surprisingly *count*'s template is almost identical to *look*'s:

<u>inside of *Deck* :</u>
int *count*() {
  … **this**.*content.mmm*() …
}

After all, like *look*, *count* is an observer and has no effects. Of course, the reference to *content*, a list of cards, implies that *count* should just delegate the task of counting cards to the list-of-cards representation. Put differently, we are adding a second wish to our wish list: *length* for *ICards*.

This is the proper time to stop and to write down the list representation for the list of cards and the wish list of methods on lists: see figure 147. It displays the class diagram for lists of cards and it uses the interface box for writing down the wish list. Specifically, the interface for *ICards* lists the two methods we have wished for so far: *fst* and *length*. The last two methods are needed for the *take* method, which we are about to design.

Exercises

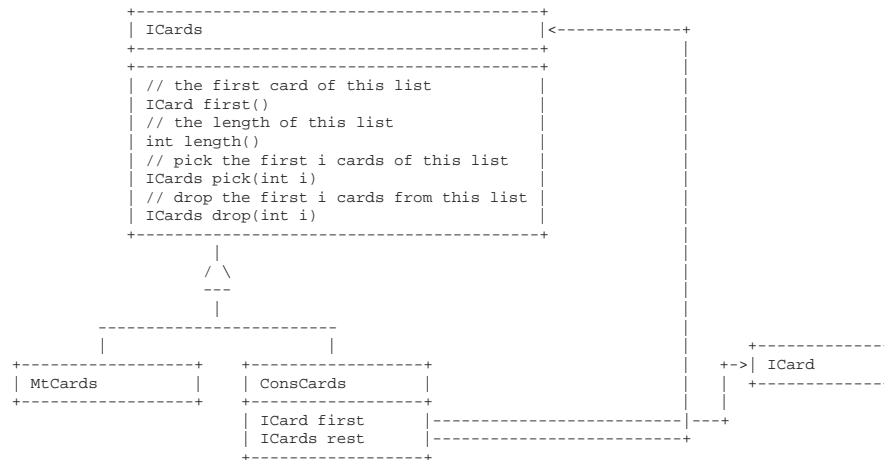**Exercise 27.17** Develop purpose statements for the *fst* and *length* methods in the *ICards* interface. ∎

```
+-----------------------------------------+
| ICards                                  |<-------------+
+-----------------------------------------+              |
+-----------------------------------------+              |
| // the first card of this list          |              |
| ICard first()                           |              |
| // the length of this list              |              |
| int length()                            |              |
| // pick the first i cards of this list  |              |
| ICards pick(int i)                      |              |
| // drop the first i cards from this list|              |
| ICards drop(int i)                      |              |
+-----------------------------------------+              |
             |                                           |
           / \                                           |
           ---                                           |
            |                                            |
      ------------------------                           |
      |                      |                           |      +-------------+
+------------------+  +------------------+               |  +->| ICard       |
| MtCards        | |  | ConsCards      | |               |  |  +-------------+
+------------------+  +------------------+               |  |
                      | ICard first  |-------------------------|---+
                      | ICards rest  |-------------------------+
                      +------------------+
```

Figure 147: A list of cards

Exercise 27.18 Design *fst* and *length* for list of cards representation. Then convert the examples for *put*, *look*, and *count* from *Deck* into tests and run the tests. Make sure to signal an error (with appropriate error message) when *fst* is invoked on an instance of *MtCards*. ∎

Our last design task is also the most complex one. The *take* method is

to remove a specified number of cards from the top of the deck.

In figure 146, this sentence has been translated into a purpose and effect statement plus a method signature. Let's look at those closely:

inside of *Deck* :
// take the first *i* cards from **this** deck
// effect: change *content* to indicate the removal of the cards
ICards take(int *i*)

The English of the purpose statement is ambiguous. The "take" could mean to remove the specified number of cards and to leave it at that, or it could mean to remove them and to hand them to the (software that represents the) player. The presence of the effect statement clarifies that the second option is wanted. That is, the method is to remove the first *i* cards from

the deck and simultaneously return them in the form of a list. In short, the method is both a command and an observer.

Now that we have a clear understanding of the role of *take*, making up behavioral examples is feasible. Let's start with the simple data example:

ICard c1 = **new** *Card(*"Boeing B-17 (Flying Fortress)"*,2);*
Deck d1 = **new** *Deck(c1)*

check *d1.take(*1*)* **expect new** *ConsCards(c1,***new** *MtCards())*

Since *take* is both an observer and a command, it is possible to specify its desired result and to specify its applicative behavior. Still, ensuring that the method invocation also affects the *contents* field requires additional observations (or post-conditions):

check *d1.count()* **expect** 0

This line demands that after taking one card from a singleton deck, there shouldn't be any cards left, i.e., *count* should produce 0. Of course, this immediately raises the question what *look* is supposed to return now:

d1.look()

The answer is that a player shouldn't *look* at the deck after requesting (all the) cards and before putting the end-of-turn card down. The situation corresponds to a request for the first item of a list and, as we know, evaluation the Scheme expression (*first* empty) raises an error. It is therefore appropriate for *look* to produce an error in this case.

For a second behavioral example, we use the deck of two cards:

ICard c1 = **new** *Card(*"Boeing B-17 (Flying Fortress)"*,2);*
ICard c2 = **new** *Card(*"Boeing B-29 (Superfortress)"*,1);*
Deck d1 = **new** *Deck(c1);*

d1.put(c2)

This deck has two cards, meaning *take* can remove one or two cards from it. Let's see how you can deal with both cases. First, the case of taking one card looks like this:

check *d1.take(*1*)* **expect new** *ConsCards(c2,***new** *MtCards())*

check *d1.count()* **expect** 1

check *d1.look()* **expect** *c1*

Second, the case of taking two cards empties the deck again:

> *d1.take*(2) **expect new** *ConsCards*(*c2*,**new** *ConsCards*(*c1*,**new** *MtCards*()))

> *d1.count*() **expect** 0

And as before, *d1.look*() raises an error.

The template for *take* is similar to the one for *put*:

> inside of *Deck* :
> *ICards take*(int *n*) {
>   **this**.*content* = ...
>   ... **this**.*content.mmm*(*n*) ...
> }

Together, the template, the examples, and the header suggest the following:

> inside of *Deck* :
> *ICards take*(int *n*) {
>   **this**.*content* = **this**.*content.drop*(*n*);
>   **return this**.*content.pick*(*n*);
> }

Before you continue now you should turn the examples into tests, complete the program, and run all the tests.

Exercises

Exercise 27.19 At this point the wish list contains two additional methods for *ICards* (also see figure 147):

1. *pick*, which retrieves a given number of cards from the front of a list;

2. and *drop*, which drops a given number of cards from the front of a list and returns the remainder.

Design these two methods. Then turn the examples for *take* into tests and run the tests. Observe that tests fail. Explain why. ∎

Exercise 27.20 Abstract over the two flavors of test cases with two items in a deck. That is, create a method that produces a deck with two cards and reuse it for the test cases. ∎

When you run the tests for this first definition of *take*, the method invocation *d1.take*(2) signals an error even though *d1* contains two cards. ProfessorJ also shows you that it is the second line in *take*'s method body that causes the error: *pick* is used on an empty list of cards. The problem is that

this.*content* = **this**.*content.drop*(*n*)

has already changed the value of the *content* field by the time, the *pick* method is used to retrieve the front-end of the list. In short, we have run into one of those timing errors discussed in section 27.5.

To overcome the timing error, it is necessary to compute the result of *take* before changing the *content* field:

inside of *Deck* :
ICards take(int *n*) {
  ICards result = **this**.*content.pick*(*n*);
  this.*content* = **this**.*content.drop*(*n*);
  return *result*;
}

Following section 27.5, the method uses a temporary variable, named *result*, to hang on to the demanded items from *content*. Afterwards it is free to change *content* as needed. At the end, it returns the value of *result*.

Exercises

Exercise 27.21 Formulate the examples for *take* as test cases and complete the definition of *Deck*. ∎

Exercise 27.22 The expression

new *Deck*(**new** *Card*("B-52",4)).*take*(2)

signals an error concerning lists. This should puzzle your colleague who uses your *Deck* class to implement an all encompassing *GameWorld* class. Here, he has accidentally invoked *take* with a number that is too large for the given *Deck*, but this still shouldn't result in an error message about *lists*. Modify the definition of *take* so that it signals an error concerning *Deck*s if invoked with bad arguments. ∎

Exercise 27.23 Find all places where our design of decks raise errors. Formulate a boolean expression, using observer methods from the *same* class,

that describes when the method is guaranteed to work properly. Add those expressions to the purpose statement prefixed with "@pre."

Note: Such conditions are known CONTRACTS. Ideally, a language should help programmers formulate contracts as part of the method signature so that other programmers know how to use the method properly and, if they fail, are told what went wrong (where). ∎

Exercise 27.24 Explain why the game administrator can always *look* at the *Deck* between the turns of players without running the risk of triggering an error. ∎

27.8.1 Caching and Invisible Assignments

The implementation of a deck of cards is an ideal place to illustrate another common use for assignments to fields: caching results. Given the current definitions of *Deck* and *ICards*, a request to *count* the number of cards in a deck becomes a method call for *length* on *content*, the list of cards. This call, in turn, visits every *ConsCards* objects of the list. Using the $O(\cdot)$ notation from *How to Design Programs* (Intermezzo 5), every request for the length of a list consumes $O(n)$ time where n is the number of cards in the list.

Fortunately, a deck of cards tends to contain just a few numbers, so that traversing the list doesn't take much time on any modern computer. Still, we can actually eliminate such traversals and thus reduce the time needed to a small constant amount of time. The key idea is to introduce a (hidden) mutable field in *Deck* that represents the number of cards on the deck. Its purpose statement implies that every method that adds or removes cards from the *Deck* must change the value of the field appropriately. And, naturally, the *count* method uses the field instead of a call to *length*. The idea of pre-computing the result of a method such as *length* is called CACHING; the extra field is a CACHE.

Figure 148 displays the modified definition of *Deck* with indexed gray shades highlighting the edits, each labeled with subscript 1 through 6:

1. A field that exists for internal purposes only should come with an explanation that tells future readers what it represents.

2. The *noCards* field is initially 0 because the *content* field initially stands for the empty list of cards.

3. The constructor changes *content* to a list of one card and must therefore change *noCards* to 1.

```
// a deck of cards
class Deck {
  private ICards content = new MtCards();
  // a cache for tracking the number of cards 1

  private int noCards = 0 2;

  public Deck(ICard c) {
    this.content = new ConsCards(c,new MtCards());
    this.noCards = 1 3;
  }

  public void put(ICard t) {
    this.content = new ConsCards(t,this. content);
    this.noCards = this.noCards+1 4;
    return ;
  }

  public ICards take(int i) {
    ICards result = this.content.take(i);
    this.content = this.content.drop(i);
    this.noCards = this.noCards-i 5;
    return result;
  }

  public int count() {
    return this.noCards 6;
  }
  ...
}
```

Figure 148: Deck with cache for number of cards

4. The *put* method adds a card to *content* and therefore increments *no-Cards* by 1.

5. The *take* method removes *i* cards from *content* and must decrement *noCards* by *i*.

6. Finally, *count* simply returns the value of *noCards* because it stands for the number of cards in the field.

As you can see, modifying the class is straightforward. To ensure that *no-*

Cards always stands for the number of cards in *content*, we check all effect statements of the methods; if it mentions *content*, we must edit the method.

Exercises

Exercise 27.25 Now that *Deck* caches its number of cards, it is possible to simplify the list representation. Do so. ∎

Exercise 27.26 A cache doesn't necessarily involve a mutable field. Add hidden fields to *MtCards* and *ConsCards* that represent how many cards the list contains. ∎

27.8.2   Merging List Traversals: Splitting Lists

Every invocation of *take* in *Deck* triggers two traversals of the list of cards to the exact same point. First, it uses *take* on the list of cards to produce a list of the first *i* cards. Second, *drop* traverses the first *i ConsCards* again and produces the list in *rest* of the *i*th object.

   While these traversals take hardly any time, the two results can be computed with a single method, and you know in principle how to do so with an applicative method. This example, however, provides a chance to demonstrate the usefulness of assignments to self-referential fields such as *rest* in *ConsCards*. Let's state the goal explicitly:

   ... Design the method *break* for *ICards*. The method consumes
   an int *i*. Its purpose is to break the list, after the *i*th object on the
   list; it is to return the second half as its result. ...

The problem statement includes a purpose statement and suggests an effect statement, too. All that's left for you here is to add a method signature:

   inside of *ICards* :
   // deliver the tail after the first *i* cards from **this** list
   // effect: change *rest* in the *i ConsCards* so that it is *MtCards*()
   ICards break(int *i*);

   Since splitting a list is only interesting when a list contains some cards, we start with a list that has two and split it in half:

   ICard c1 = **new** *Card*("black",1);
   ICard c2 = **new** *Card*("black",2);
   ICards cs = **new** *ConsCards*(c1,**new** *ConsCards*(c2,**new** *MtCards*()));

> **check** *cs.break*(1) **expect new** *ConsCards*(*c2*,**new** *MtCards*())

The question is what else happens? And if anything happens, how do we observe the change? According to the effect statement, the method is to change the rest field in the first *ConsCards* object, the one that contains *c1*. By implication, the list *cs* should no longer contain two but one card now:

> …
> **check** *cs* **expect new** *ConsCards*(*c1*,**new** *MtCards*())

That is, the value of *cs* changes even though there is no assignment statement for the variable. Think through the example again, because it is unlike anything you have seen before.

Our design recipe demands the creation of a template next:

inside of *MtCards* :	inside of *ConsShots* :
ICards split(int *i*) {	*ICards split*(int *i*) {
…	… **this**.*first* …
}	… **this**.*rest.split*(… *i* …) …
	this.*rest* = …
	}

The templates for both variants of the union are unsurprising. Because *MtCards* has no fields, its template doesn't contain any expressions. In *ConsCards*, we see three expressions: one that reminds us of the *first* field; one for *rest* and the natural recursion; and a skeleton assignment to *rest*.

At this point, we are almost ready to define the method. What we are missing is a thorough understanding of *i*, the second argument. Up to now, we have dealt with *i* as if it were an atomic piece of data. The purpose statement and examples, though, tell us that it is really a natural number and *split* processes the list of cards and this number in parallel. From *How to Design Programs*'s chapter III, we know that this requires a conditional that checks how the list gets shorter and the number smaller. In Java, however, there are no tests on the lists. Each list-implementing class contains its own *split* method so that it "knows" whether the list is empty or not.

In other words, the methods must only test how small *i* is and must decrease *i* when it recurs. When *i* becomes 1, the method has traversed exactly *i* instances of *ConsCards*, and it must split the list there:

```
inside of ConsShots :
ICards split(int i) {
  if (i == 1) {
    ... this.first ...
    this.rest = ...
  } else {
    return this.rest.split(i); }
}
```

Note how the *then* branch in this outline composes two sketches of statements. In a world of effects, this makes perfect sense.

The method outline translates the words into code. The **if**-statement distinguishes two cases:

(*i* == 1) In this case, the method is to traverse one instance of *ConsCards* and has just done so. There is no need to continue the traversal, which is why the natural recursion has been removed from method template, which is otherwise repeated unchanged.

(*i* != 1) In this case, the method hasn't traversed enough nodes yet, but it has encountered one more than before. Hence, it uses the natural recursion from the template to continue.

Given this case analysis, we can focus on the first case because the second one is done. A first guess is to use

```
if (i == 1) {
  this.rest = new MtCards();
  return this.rest; }
else ...
```

But obviously, this is nonsense. Return **this**.*rest* right after setting it to **new** *MtCards*() can't possibly work because it always return the empty list of cards. We have encountered another case of bad timing. To overcome this problem, we introduce a local variable and save the needed value:[63]

```
ICards result = this.rest;
if (i == 1) {
  this.rest = new MtCards();
  return result; }
else ...
```

---

[63] In Java, the variable could be made local to the *then* branch of the **if**-statement.

In this version, the value of **this**.*rest* is saved in *result*, then changed to **new** *MtCards*(), and the method returns the value of *result*.

For completeness, here are the definitions of the two *split* methods:

inside of *MtCards* :
```
ICards split(int i) {
  return
    Util.error(" ... an empty list");
}
```

inside of *ConsShots* :
```
ICards split(int i) {
  if (i == 1) {
    ICards result = this.rest;
    this.rest = new MtCards();
    return result;
  } else {
    return this.split(i-1); }
}
```

Exercises

Exercise 27.27 Turn the examples for *split* into tests and ensure that the method definition works properly for the examples. Add tests that explore what *split* computes when given 0 or a negative number. ∎

Exercise 27.28 Use the *split* method to re-define *take* in *Deck*. ∎

Exercise 27.29 Challenge problem: Design *splitA*, an applicative alternative to *split*. Like *split*, *splitA* consumes an int *i*. After traversing *i* instances of *ConsCards*, it returns the two halves of the list.

Hint: See section 27.6 on how to return two results from one method. Also re-read *How to Design Programs* (chapter VI) on accumulator style. ∎

27.9   Endnote: Where do Worlds Come from? Where do they go?

If you want to become a top-tier programmer or a computer scientist, you ought to wonder how the applicative and the imperative *World* classes work.[64] Alternatively, you might imagine that you are the designer of libraries for your company, which uses **idraw**, and you notice that programmers have a much easier time with applicative classes rather than imperative ones. So you ask yourself:

---

[64]We cannot really explain how to define either of the two libraries in complete detail. For that, you will need to learn more about computers and operating systems.
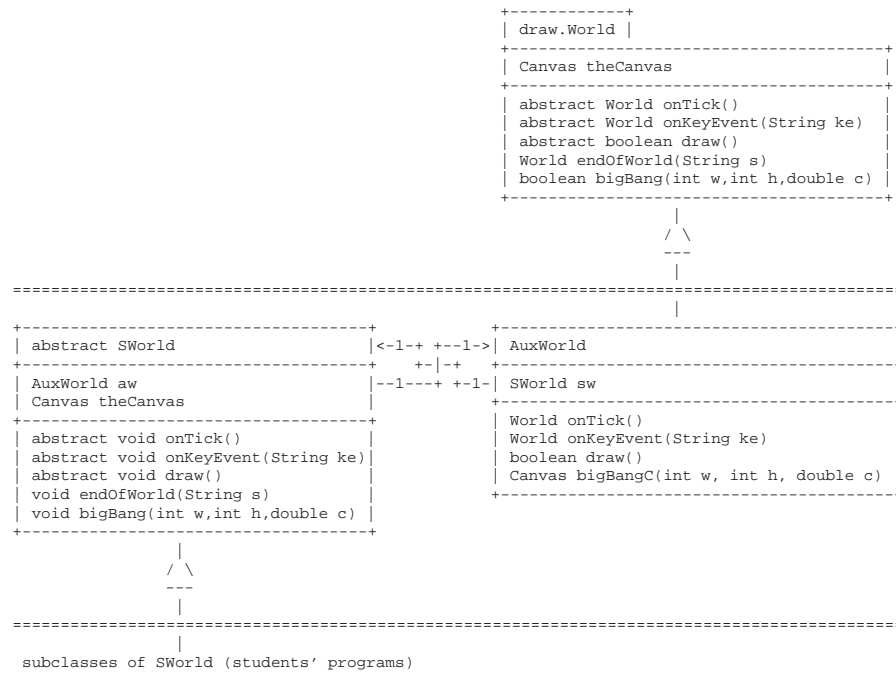
```
                                                    +------------+
                                                    | draw.World |
                                                    +---------------------------------------+
                                                    | Canvas theCanvas                      |
                                                    +---------------------------------------+
                                                    | abstract World onTick()               |
                                                    | abstract World onKeyEvent(String ke)  |
                                                    | abstract boolean draw()               |
                                                    | World endOfWorld(String s)            |
                                                    | boolean bigBang(int w,int h,double c) |
                                                    +---------------------------------------+
                                                                        |
                                                                       / \
                                                                       ---
                                                                        |
===========================================================================================
                                                                        |
+-----------------------------------+          +---------------------------------------+
| abstract SWorld                   |<-1-+ +--1->| AuxWorld                              |
+-----------------------------------+    +-|-+  +---------------------------------------+
| AuxWorld aw                       |--1---+ +-1-| SWorld sw                             |
| Canvas theCanvas                  |          +---------------------------------------+
+-----------------------------------+          | World onTick()                        |
| abstract void onTick()            |          | World onKeyEvent(String ke)           |
| abstract void onKeyEvent(String ke)|         | boolean draw()                        |
| abstract void draw()              |          | Canvas bigBangC(int w, int h, double c) |
| void endOfWorld(String s)         |          +---------------------------------------+
| void bigBang(int w,int h,double c) |
+-----------------------------------+
                    |
                   / \
                   ---
                    |
===========================================================================================
                    |
 subclasses of SWorld (students' programs)
```

Figure 149: Creating an applicative world: the diagram

> ... given *World* from the **idraw** library, design a class like the
> applicative *World* class from the **draw** library. ...

Naturally you may also imagine the opposite problem:

> ... given *World* from the **draw** library, design a class like the
> stateful *World* class from the **idraw** library. ...

For both problems, you are facing the question of how to implement a class
that has similar functionality as some given library class but entirely differ-
ent types. That is, you want to adapt an existing class to a context that ex-
pects different types rather than design an entirely new class from scratch.

   As it turns out, the second problem is slightly easier to solve than the
first one, so we start with it. The problem is unlike any design problem you
have seen before. You are given a complete specification of one class, and
you must define it, using some second class. Still, some of the ideas from
the design recipe process apply and help you here.

Figure 149 summarizes the situation with a three-tier class diagram. The top tier represents the given library. The box in this tier summarizes the given *World* class from **draw**; recall that figure 90 (page 272) specifies the same class with field declarations, method signatures, and purpose statements. The middle tier is the library that you are to design. The left box is the class that the library is to provide. Again, it is just the box representation of a class you got to know via a textual specification: see figure 135 (page 368). We have dubbed the class *SWorld*; think "stateful" world when you see it. Finally, the bottom tier represents the programs that others design based on your library. Specifically, they are subclasses of *SWorld* like *BlockWorld*, *UFOWorld*, *WormWorld*, and so on.

Exercises

Exercise 27.30 Why can't *SWorld* extend *World* directly? After all, *World* provides all the needed functionality, and programmers are supposed to extend it to use it. ∎

Exercise 27.31 Design the class *Block1* as a subclass of *SWorld*. The purpose of *Block1* is to represent a single block dropping from the top of a 100 by 100 canvas and stopping when it reaches the bottom. Make the class as simple as possible (but no simpler).

Note: This exercise plays the role of the "example step" in the design of *SWorld*. To experiment with this class, switch it to be a subclass of *World* from **idraw**. ∎

The box on the right side of the library tier in figure 149 is needed according to the design guidelines for using existing abstract classes. Remember that according to chapter III, the use of an abstract class such as *World* from **idraw** requires the construction of a subclass of *World*. The subclass inherits *bigBang*, *endOfWorld*, and *theCanvas* from *World*; it must provide concrete definitions for *onTick*, *onKeyEvent*, and *draw*. These last three methods are to implement the behavior that creates the animation. Of course, this behavior is really created by the users of the library, i.e., by instances of subclasses of *SWorld*.

In turn, we know that the *SWorld* class is abstract; it introduces three abstract methods: *onTick*, *onKeyEvent*, and *draw*; and it supplies *bigBang*, *endOfWorld*, and *theCanvas* to its subclasses. These last three, it must somehow acquire from *AuxWorld*, the concrete subclass of *World*. These consid-

erations imply that (some instances of) the two to-be-designed classes need to know about each other.

Figure 149 shows this dependence with two fields and two containment arrows. The *AuxWorld* class contains a field called *sw* with type *SWorld*, and *SWorld* contains a field called *aw* with type *AuxWorld*. The methods in the two classes use these two fields to refer to the other class and to invoke the appropriate methods from there.

Now that we understand the relationship among the two classes, we can turn to the next step in the design recipe, namely, figuring out how to establish the relationship. To do so, let's imagine how we would create an animation with this library. As agreed, an animation class (such as *Block1* from exercise 27.31) is a subclass of *SWorld*. From the last two chapters, you also know that the animation is started in two steps: first you create the world with **new** and then you invoke *bigBang* on **this** object. Hence, instances of (a subclass of) *SWorld* come into existence first. As they do, they can can create the associated instance of *AuxWorld* and establish the backwards connection.

Since all of this suggest that we need exactly one instance of *AuxWorld* per instance of *SWorld*, we create the former and immediately associate it with *aw* in the field declaration:

> **abstract class** *SWorld* {
>   *AuxWorld aw* = **new** *AuxWorld*(**this**);
>
>   . . .
> }

By handing over **this** to the constructor of *AuxWorld*, the latter can assign its *sw* field the proper value:

> **class** *AuxWorld* **extends** *World* {
>   *SWorld sw*;
>   *AuxWorld*(*SWorld sw*) {
>   **this**.*sw* = *sw*;
>   }
>   . . .
> }

It is easy to see that the field declaration creates one instance of *AuxWorld* per instantiation of *SWorld* and that this instance is associated with exactly **this** instance of (a subtype of) *SWorld*.[65]

---

[65]This design demonstrates that even though the instances of *SWorld* and *AuxWorld* are in a direct cyclic relationship, it isn't necessary to use the entire design process from the first

```
abstract class SWorld {                    class AuxWorld extends World {
  AuxWorld aw =                              SWorld ad;
    new AuxWorld(this);
  Canvas theCanvas;                          AuxWorld(SWorld ad) {
                                               this.ad = ad;
  void bigBang(int w, int h, double c) {     }
    theCanvas = aw.bigBangC(w,h,c);
    return ;                                 Canvas bigBangC(int w, int h, double c) {
  }                                            bigBang(w,h,c);
                                               return theCanvas;
  void endOfWorld(String s) {                }
    aw.endOfWorld(s);
    return ;                                 World onTick() {
  }                                            ad.onTick();
                                               return this;
  abstract void onTick();                    }

  abstract void onKeyEvent(String ke);       World onKeyEvent(String ke) {
                                               ad.onKeyEvent(ke);
  abstract void draw();                        return this;
}                                            }

                                             boolean draw() {
                                               ad.draw();
                                               return true;
                                             }
                                           }
```

Figure 150: Worlds, worlds, worlds—the imperative version

With the fields and the class relationships mostly under control, we can turn to the design of methods. Let's start with *bigBang*, which is the first method that is invoked after *SWorld* is instantiated. Even though we can't make concrete examples, we can lay out the template for the method:

---

section of this chapter. When there is a direct one-to-one, immutable relationship where one object completely controls the creation of the other, it is possible to create the relationship when the objects are created.

<u>inside of *SWorld* :</u>
```
void bigBang(int w, int h, double c) {
  … this.aw.mmm() … this.theCanvas.mmm() …
   this.aw = …
   this.theCanvas = …
}
```

Given that the class contains two fields of complex type, it is natural that both show up as potential method calls. For completeness, we have also added two assignment statements. Because we already know that *aw* provides the proper functionality with its *bigBang* method, it is tempting to just call the method, wait for it to return true, and to return void in the end:

<u>inside of *SWorld* :</u>
```
void bigBang(int w, int h, double c) {
  this.aw.bigBang(w,h,c); // throw away the value of this expression
  return ;
}
```

Although this action starts the clock and creates (displays) the canvas, it fails to make *theCanvas* available to **this** instance of *SWorld*. While the *bigBang* method in *World* assigns the newly created *Canvas* to *theCanvas*, the *theCanvas* field in *SWorld* is still not initialized. Worse, *theCanvas* in *World* is **protected**, meaning only the methods in *World* and in its subclasses can access its value. The *SWorld* class, however, cannot grab the value and assign it to its *theCanvas* field.

One solution is to add a *getCanvas* method to *AuxWorld* whose purpose is to hand out *theCanvas* when needed. It is important though that you never call this method before *bigBang* has been called. An alternative solution is to define the method *bigBangC*, which invokes *bigBang* and then returns *theCanvas*. By lumping together the two steps in one method body, we ensure the proper sequencing of actions.

Figure 150 displays the complete definitions of *SWorld* and *AuxWorld*. The code drops the "**this**." prefix where possible to make the code fit into two columns; it is legal and convenient to do so in the Advanced language of ProfessorJ and Java. The figure also shows how *AuxWorld*'s concrete methods compute their results via method calls to the abstract methods of *SWorld*. We have thus implemented a template-and-hook pattern (see section 18.4, page 238) across two classes.

Exercises

Exercise 27.32 Equip all fields and methods in *SWorld* and *AuxWorld* with privacy specifications and ensure that the classes work as advertised with your *Block1* animation (exercise 27.31). ∎

Exercise 27.33 Replace the *bigBangC* method with a *getCanvas* method in *AuxWorld* and adapt *SWorld* appropriately. ∎

```
                                    +-------------+
                                    | idraw.World |
                                    +-------------------------------------+
                                    | Canvas theCanvas                    |
                                    +-------------------------------------+
                                    | abstract void onTick()              |
                                    | abstract void onKeyEvent(String ke) |
                                    | abstract void draw()                |
                                    | void endOfWorld(String s)           |
                                    | void bigBang(int w,int h,double c)   |
                                    +-------------------------------------+
                                                      |
                                                     / \
                                                     ---
                                                      |
===========================================================================================
                                                      |
+-------------------------------------+          +---------------------------------------+
| abstract AWorld                     |<*-+  +-1-->| Keeper                                |
+-------------------------------------+   +--|-+  +---------------------------------------+
| Keeper wk                           |-*----+ +-1-| AWorld crnt                           |
| Canvas theCanvas                    |          +---------------------------------------+
+-------------------------------------+          | void onTick()                         |
| abstract AWorld onTick()            |          | void onKeyEvent(String ke)            |
| abstract AWorld onKeyEvent(String ke) |        | void draw()                           |
| abstract boolean draw()             |          | Canvas bigBangC(int w, int h, double c)|
| AWorld endOfWorld(String s)         |          +---------------------------------------+
| boolean bigBang(int w,int h,double c) |
+-------------------------------------+
                    |
                   / \
                   ---
                    |
===========================================================================================
                    |
 subclasses of AWorld (students' programs)
```

Figure 151: Creating an applicative world: the diagram

For the design of an applicative world from a stateful one, we exploit the experience from the preceding solution and copy the basic set-up. As before, we exploit ideas from the design recipe to guide the process. Figure 151 summarizes the situation. It shows the three tiers just like figure 149 but the roles have been inverted:

1. *World* is now from the **idraw** package;

2. its to-be-designed subclass *Keeper* provides concrete definitions for the imperatively typed methods *onTick*, *onKeyEvent*, and *draw*;

3. and the to-be-designed *AWorld* class—for applicative world—is like the original definition of *World* from the **draw** package.

As before, the instances of the two to-be-designed classes must know about each other. Hence, each class has a field with a type from the other class. In addition, the setup procedure works like before; the field in *AWorld* comes with an initial value and hands over **this** to the constructor of *Keeper* so that it can establish the mutual references among the object.

The key to designing *AWorld* in terms of *Keeper*—the subclass of *World*—is to determine where the results of *onTick* and *onKeyEvent* go, when the methods have finished their computations. After all, these methods create entirely new worlds to represent changes over time. To solve this problem, remember the design recipe for stateful classes such as *Keeper*. After you know the methods, you are supposed to determine whether invocations of one method depend on prior calls to others and, if so, through which field(s) they communicate the changes. In the case of *Keeper*, *draw*'s action on the canvas is usually a function of how often *onTick* has been invoked. The invocation of *onTick* is supposed to change **this** instance of *Keeper*. Furthermore, we know from the first half of this section that *onTick* really implements a template-and-hook pattern with *onTick* in *AWorld* and that the latter method is to produce a new *AWorld* every time it is invoked.

A moment of reflection on the role of the classes and their fields shows that the first instance of *Keeper* is the keeper of the current world. Its *crnt* field is the place where it stores the current world. With this in mind, designing the remaining methods in *Keeper* is straightforward now. The *onTick* and *onKeyEvent* methods are the most interesting ones. Here are a purpose and an effect statement, a method, and a template for the former:

```
inside of Keeper :
// process a tick of the clock in this world
// effect: change crnt to the current world
void onTick() {
  ... this.crnt.mmm() ...
  this.crnt = ...
}
```

As always, the template suggests that *onTick* has *crnt* available and could invoke a method on it. It also reminds us that the method may have to change the value of *crnt* via the skeleton of the assignment in the last line.

The informal description of *onTick* in *AWorld* (page 273) suggests first definitions for *onTick* and *onKeyEvent*:

inside of *Keeper* :	inside of *Keeper* :
void *onTick*() {	void *onKeyEvent*(*String ke*) {
crnt = *crnt.onTick*();	*crnt* = *crnt.onKeyEvent*(*ke*);
. . .	. . .
crnt.draw();	*crnt.draw*();
}	}

Both methods compute their results in the same way. First, they invoke the respective method on *crnt* and assign the resulting new world to *crnt*. Second, the methods invoke *crnt.draw* to draw the new world into the canvas.

Naturally, the *onTick* method in *Keeper* invokes *crnt.onTick* and the *onKeyEvent* method invokes *crnt.onKeyEvent* in the process. Remember that *crnt* is an instance of (a subtype of) *AWorld* and that the *onTick* and *onKeyEvent* methods in *AWorld* are abstract. A subclass of *AWorld* overrides these methods with concrete methods and those definitions are the ones that the two methods above end up using.

While these first drafts implement the desired behavior from the external perspective of *onTick*, they fail to re-establish the connection between **this** instance of *Keeper* and the worlds that are created. That is, when *crnt.onTick*() creates new instance of (a subtype of) *AWorld*, it also automatically creates a new instance of *Keeper*, which is unrelated to **this**. Worse, the new instance of *AWorld* doesn't know about *theCanvas* either because it is **this** *Keeper* that holds on to *the* canvas with a picture of the current world.

Ideally, the evaluation of *crnt.onTick*() should not create a new instance of *Keeper*. Of course, the constructor of an applicative class cannot distinguish the situation where a new world is created for which *bigBang* is about to be called and the situation where the successor of a current world is created. Inside the body of **this** *Keeper*'s *onTick* method the situation is clear, however. Hence, it is *its* duty to establish a connection between the newly created world and itself (and its canvas):

inside of *Keeper* :	inside of *Keeper* :
void *onTick*() {	void *onKeyEvent*(*String ke*) {
crnt = *crnt.onTick*();	*crnt* = *crnt.onKeyEvent*(*ke*);
crnt.update(**this**,*theCanvas*);	*crnt.update*(**this**,*theCanvas*);
crnt.draw();	*crnt.draw*();
}	}

In other words, thinking through the problem has revealed the need for a new method, an entry on our wish list:

inside of *AWorld* :
// establish a connection between **this** (new) world and its context
// <u>effect</u>: change *myKeeper* and *theCanvas* to the given objects
void *update*(*Keeper k*, *Canvas c*)

That is, we need a method for establishing the circularity between *AWorld*s and the keeper of all worlds after all.

Exercises

Exercise 27.34 Define *update* for *AWorld*. Collect all code fragments and define *AWorld* and *Keeper*.

Now develop the subclass *AppBlock1* of *AWorld*. The purpose of *AppBlock1* is the same as the one of *Block1* in exercise 27.31: to represent a single block that is dropping from the top of a canvas and stopping when it reaches the bottom. Parameterize the public constructor over the size of the canvas so that you can view differently sized canvases.

Use *AppBlock1* to check the workings of *AWorld* and *Keeper*. ∎

Exercise 27.35 The purpose of *update* in *AWorld* is to communicate with *Keeper*. Use the method to replace *bigBangC* in *Keeper* with *bigBang*, i.e., a method that overrides the method from *World* in **idraw**. Use *Block1App* from exercise 27.34 to run your changes.

Equip all fields and methods in *AWorld* and *Keeper* with privacy specifications making everything as private as possible. Use *AppBlock1* from exercise 27.34 to ensure that the classes still work.

Argue that it is wrong to use **public** or **protected** for the privacy specification for *update*. Naturally it is also impossible to make it **private**. Note: It is possible to say in Java that *update* is only available for the specified library but not with your knowledge of privacy specifications. Study up on `packages` and privacy specifications (and their omission) in the Java report to clarify this point. ∎

Exercise 27.36 Create an instance of *Block1App*, invoke *bigBang* on it, and do so a second time while the simulation is running. Specify a canvas for the second invocation that is larger than the first one. Explain why you can observe only the second invocation before you read on.

Modify your definition of *Block1App* so that it works with the regular **draw** library. Conduct the experiment again. ∎

Exercise 27.36 exposes a logical error in our design. The very principle of an applicative design is that you can invoke the same method twice or many times on an object and you should always get the same result. In particular, an applicative class and method should hide all internal effects from an external observer. For the particular exercise, you ought to see two canvases, each containing one block descending from the top to the bottom; the re-implementation using *World* from **draw** confirms this.

The problem with our first draft of *AWorld* is that a second invocation of *bigBang* uses the *same* instance of *Keeper*. If, say, the clock has ticked five times since the first invocation of *bigBang*, the *crnt* field contains

b.onTick().onTick().onTick().onTick().onTick()

When the second invocation of *bigBang* creates a second canvas, this canvas becomes the one for the instance of *Keeper* in **this** world. All drawing actions go there and invocations of *update* on the above successor of *b* ensure that all future successors of *b*—that is, the results of additional calls of *onTick*—see this new canvas.

With this explanation in hand, fixing our design is obvious. Every invocation of *bigBang* must create a new instance of *Keeper*:

```
abstract class AWorld {
  private Keeper myKeeper;
  protected Canvas theCanvas;

  public boolean bigBang(int w, int h, double c) {
    myKeeper = new Keeper(this);
    theCanvas = myKeeper.bigBangC(w,h,c);
    return true;
  }
  . . .
}
```

From then on, this *Keeper* object is the caretaker of the first instance of (a subtype of) *AWorld* and all of its successors created from event handling methods. Figure 152 shows how it all works. On the left, you see the complete and correct definition of *AWorld* and on the right you find the definition of *Keeper*. Study them well; they teach valuable lessons.

Exercise

Exercise 27.37 Fix your implementation of *AWorld* and conduct the experiment of exercise 27.36 again. ∎

```
abstract class AWorld {                     class Keeper extends World {
  Keeper myKeeper = new Keeper(this);         AWorld crnt;
  Canvas theCanvas;                           Keeper() { }

  boolean bigBang(int w, int h,               Keeper(AWorld first) {
                  double c) {                   this.crnt = first;
    theCanvas =                               }
      myKeeper.bigBangC(w,h,c);
    return true;                              Canvas bigBangC(int w, int h, double c) {
  }                                             bigBang(w,h,c);
                                                return theCanvas;
  void update(Keeper wk, Canvas tc) {         }
    myKeeper = wk;
    theCanvas = tc;                           void onTick() {
  }                                             crnt = crnt.onTick();
                                                crnt.update(this,theCanvas);
  boolean endOfTime() {                         crnt.draw();
    myKeeper.endOfTime();                     }
    return true;
  }                                           void onKeyEvent(String ke) {
                                                crnt = crnt.onKeyEvent(ke);
  AWorld endOfWorld(String s) {                 crnt.update(this,theCanvas);
    myKeeper.endOfTime(s);                      crnt.draw();
    return this;                              }
  }
                                              void draw() {
  abstract AWorld onTick();                     crnt.draw();
                                              }
  abstract AWorld onKeyEvent(                }
    String ke);

  abstract boolean draw();
}
```

Figure 152: Worlds, worlds, worlds—the applicative version

28 Equality

If you paid close attention while reading this chapter, you noticed that the
notion of "same" comes up rather often and with a subtly different mean-
ing from the one you know. For example, when the discussion of circular

objects insists that following some containment arrow from one field to another in a collection of objects brings you back to "the very same object." Similarly, once assignment is introduced, you also see phrases such as "the world stays the same but some of its attributes change."

Given statements that seem in conflict with what we know about equality, it is time to revisit this notion. We start with a review of extensional equality, as discussed in section 21, and then study how assignment statements demand a different, refined notion of equality.

28.1   Extensional Equality, Part 2

Figure 153 displays a minimal class definition for dropping blocks. If you ignore the gray-shaded definition for now, you see a class that introduces a single field–*height* of type int—and a single method—*drop*, which changes *height* appropriately.

If you had to compare any given instance of *DroppingBlock* with some other instance, you might come up with the *same* method in the figure. Since an instance is uniquely determined by the value of the *height* field, this method extracts those values from the two blocks—**this** and *other*—and compares them. If the values are the same, the blocks are considered the same; if they differ, the blocks are guaranteed to differ.

The idea of comparing two objects on a field-by-field basis is the essence of extensional equality as we encountered it in section 21.1. It is easy to understand and emphasizes that the attributes of an object determine its nature. As we have seen, too, you can also use the privacy specifications of a class to design *same*; this argues that only directly visible and "measurable" attributes determine whether two objects are equal. Finally, you can choose a viewpoint in the middle and base an equality comparison on some other observable property of the fields.

28.2   Intensional Equality

Now that we have reminded ourselves of the basics of extensional equality, we can turn to the problem of objects that remain the same even though they change. To make this concrete, we use the same example, the world of dropping blocks. Given the *drop* method and its imperative nature, we would have said in preceding sections that the block stays the same though its (*height*) attribute changes.

The purpose of this section is to define once and for all what this form of sameness means via the design of a method. Obviously, the header and

```
// represents dropping block in a world (stateful)
class DroppingBlock {
  private int height;
  public DroppingBlock(int height) {
    this.height = height;
  }

  // is this the same DroppingBlock as other?
  public boolean same(DroppingBlock other) {
    return this.height == other.height;
  }

  // drop this block by 1 pixel (effect: change height)
  public void drop() {
    this.height = height+1;
    return ;
  }

    // is this truly the same DroppingBlock as other?
    public boolean eq(DroppingBlock other) {
      boolean result = false;
      // save the old values
      int heightThis = this.height;
      int heightOther = other.height;

        // modify both fields
        this.height = 0;
        other.height = 1;

        // they are the same if the second assignment changes this
        result = (this.height == 1);

        // restore the fields to their original values
        this.height = heightThis;
        other.height = heightOther;

        return result;
      }
}
```

Figure 153: The intensional sameness of objects

signature for this method are similar to the ones for *same*:

> inside of *DroppingBlock* :
> ‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾
> // is **this** truly the same *DroppingBlock* as *other*?
> boolean *eq*(*DroppingBlock other*)

The name *eq* has been chosen for historical purposes.[66] The word "truly" has been added because we're not talking about extensional equality, where two blocks are the same if their fields are equal:

> **class** *IntExamples* {
>   *DroppingBlock db1* = **new** *DroppingBlock*(1200);
>   *DroppingBlock db2* = **new** *DroppingBlock*(1200);
>
>   boolean *test12* = **check** *db1.eq*(*db2*) **expect** false;
> }

Then again, comparing an instance of *DroppingBlock* with itself should definitely yield true:

> inside of *IntExamples* :
> ‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾
>   boolean *test11* = **check** *db1.eq*(*db1*) **expect** true;

Thus we have two behavioral examples but they don't tell us more than we know intuitively.

Here is an example that is a bit more enlightening:

> inside of *IntExamples* :
> ‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾
>   *DroppingBlock db3* = *db1*;
>
>   boolean *test13* = **check** *db1.eq*(*db3*) **expect** true;

Giving an instance of *DroppingBlock* a second name shouldn't have any effect on intensional equality. After all, this happens regularly while a program is evaluated. For example, if your program hands *db1* to a constructor of a *World*-style class, the parameter of the constructor is a name for the object. Similarly, if you invoke *drop* on *db1*, the method refers to the object via **this**—which is of course just a name for the object.

This last idea plus the motivating sentence "it remains the same even though it changes" suggests one more example:

> boolean *test13again*() {
>   *db1.drop*();
>   **return check** *db1.eq*(*db3*) **expect** true;
> }

---

[66]Lisp has used this name for intensional equality for 50 years.

In this example, the instance of *DroppingBlock* has two names; it is changed via an invocation of *drop*; and yet this change doesn't affect the fact that *db1* and *db3* are intensionally equal.

Indeed, this last example also suggests an idea for the method definition. If changing an instance via one name should not affect its identity under another name, perhaps the method should use changes to a field to find out whether two different names refer to the same object. First steps first; here is the template:

> inside of *DroppingBlock* :
> // is **this** truly the same as *DroppingBlock* as *other*?
> boolean *eq*(*DroppingBlock other*) {
>   … **this**.*height* … *other.height* …
>   **this**.*height* = …
>   *other. height* = …
> }

The template includes two skeleton assignments to the fields of **this** and *other*, the two names that might refer to the same object. Its header doesn't include an effect statement; an equality method shouldn't change the objects it compares. From here, we can take the first step toward a definition:

> inside of *DroppingBlock* :
> boolean *eq*(*DroppingBlock other*) {
>   // save the old values
>   int *heightThisInitially* = **this**.*height*;
>   int *heightOtherInitially* = *other. height*;
>
>   … **this**.*height* … *other.height* …
>
>   **this**.*height* = …
>   *other. height* = …
>
>   // restore the fields to their original values
>   **this**.*height* = *heightThisInitially*;
>   *other. height* = *heightOtherInitially*;
>   …
> }

First the method must save the current values of the two *height* fields via local variables. Second it may assign to them. Third, it must restore the old connection between fields and values with a second pair of assignments so that it has no externally visible effects.

The rest requires code for our idea that if "it also changes, it is the same." In more concrete words, if the method assigns a new value to **this**.*height* and *other.height* changes, then the two methods are the same. Measuring a change in value means, however, comparing *other.height* with an int and knowing what to expect. Since we can't know the value of *other.height*, let's just assign a known value to this field, too. Now if this second assignment affects the **this**.*height* again, we are dealing with one and the same object:

inside of *DroppingBlock* :
boolean *eq*(*DroppingBlock other*) {

  . . .
  this.*height* = 0;
  other.height = 1;
  // **this** and *other* are the same if:
  . . . (**this**.*height* == 1) . . .
  . . .
}

Of course, the method can't return the result yet because it first needs to restore the values in the fields. Therefore, the result is stored locally.

The gray-shaded area in figure 153 is the complete definition of the *eq* method, and it does represent the essence of INTENSIONAL EQUALITY:

> if a change to one object affects a second object, you are dealing with one and the same object.

At the same time, *eq* is a method that assigns to the fields of two objects, yet has no visible side-effect because it undoes the change(s) before it returns. Ponder this observation before you move on.

Exercises

Exercise 28.1 Why can *eq* not just modify one instance of *DroppingBlock* via an assignment? Why does it have to change **this**.*height* and *other.height*?

Develop a test case that demonstrates that a version without one of the two assignment statements would fail to act like *eq*. In other words, it would consider two distinct instances the same, contrary to the spirit of *eq*, which only identifies **this** and *other* if they were created from one and the same **new** expression. ∎

Exercise 28.2 Discuss: Is it possible to design an *eq* method for a class that has a single field whose type is some other class? Consider the following

two hints. First, *null* is the only value that has all types. Second, calling the constructor of a random class may have visible and permanent effects such as the creation of a canvas on your computer screen. ∎

As is, the *eq* method is not useful as a general programming tool. On one hand, it requires the mutation of an object, and it requires a significant amount of shuffling of values from one variable to another and back. Getting it right isn't straightforward, and even if it is right, it is a lot of work. On the other hand, it doesn't work for *null*, the special object value introduced in this section, because *null* is a constant and can't be changed. Since it is often important to discover the presence of *null* (e.g., to prevent methods from accidentally invoking a method on it or extracting a field), researchers have come up with a cheaper and equivalent way of discovering intensional equality. Specifically, two objects are intensional equal if (and only if) they are both *null* or if they were both created by the exact same evaluation of a **new** expression during the execution of a program.

Based on this insight, Java and other object-oriented programming languages automatically provide an *eq*-like method for all objects. In Java, it isn't really a method but the == comparison operator that determines whether two objects are intensionally equal. For any two instances *db1* and *db2* of *DroppingBlock* (or any other class), the expression

$db1 == db2$

evaluates to true if and only if the two objects are intensionally the same.

28.3 Extensional Equality with null

While we discourage the use of *null* as much as possible, it is occasionally useful to formulate methods in a concise manner. Recall the equality comparison for union types from section 21.3. The essence is that each variant comes with two methods, both of which must be included in the interface:

1. boolean *isSpecificVariant*(), whose purpose is to determine whether an object is instance of a specific variant;

2. *SpecificVariant toVariant*(), whose purpose is to convert any object into an instance of the class *SpecificVariant*. Naturally, the method fails because it is simply impossible to turn apples into oranges and *Tea* into *Coffee*.

```
// some grocery items
interface IItem {
  // is this the same IItem as other?
  boolean same(IItem x);

  // is this Coffee?
  boolean isCoffee();
  // convert this to Coffee (if feasible)
  Coffee toCoffee();

  // is this Tea?
  boolean isTea();
  // convert this to Tea (if feasible)
  Tea toTea();
}
```

Figure 154: An interface with methods for extensional equality

For your convenience, figure 154 repeats the interface for the *IItem* union of *Coffee* and *Tea* variants. (Also see figures 113 and 114.)

As you can see from this interface, supporting a single *same* method in a union demands the addition of two auxiliary methods per variant. Adding a variant thus requires the implementation of all these methods, plus two more for each already existing variant. Exercises 21.6 and 21.7 bring home this conundrum, also demonstrating how abstraction can reduce some of the work in this case, though not all.

Equipped with *null* and a fast mechanism for discovering its presence, we can further reduce the work. Specifically, we can combine the two methods into one because *null* has all possible types and thus passes the type system and because == can discover *null* cheaply. Concretely, we retain the conversion method and eliminate the method that is like a predicate in Scheme:

inside of *IItem* :
// convert **this** to *Coffee* (to *null* otherwise)
Coffee toCoffee();

Defining this method is straightforward. In *Coffee*, **returns this**; everywhere else, say in *Tea*, it **return**s *null*:

<div style="display:flex">
<div>

inside of *Coffee* :
Coffee toCoffee() {
  return this;
}

</div>
<div>

inside of *Tea* :
Coffee toCoffee() {
  return *null*;
}

</div>
</div>

Once we have this single method, adapting *same* is also easy:

<div style="display:flex">
<div>

inside of *Coffee* : (original version)
boolean *same*(*IItem other*) {

  return *other.isCoffee*()
      && other.toCoffee().*same*(**this**);
}

</div>
<div>

inside of *Coffee* : (new version)
boolean *same*(*IItem other*) {
  Coffee c = other.toCoffee();
  return (*null == c*)
      && c.same(**this**);
}

</div>
</div>

Like the original version (on the left), the new version (on the right) converts *other* to *Coffee*. If this conversion step produces *null*, the boolean expression evaluates to false; otherwise, it invokes the *same* method on the result of the conversion and hands over **this**, whose type is *Coffee*. The resolution of overloading thus resolves this second method call to the **private** method (see figure 114) that compares one *Coffee* with another.

Exercise

Exercise 28.3 Complete the definitions of the *Tea* and *Coffee* classes, each implementing the revised *IItem* interface. Then add the *Chocolate* from exercise 21.6 as a third variant to the *IItem* union. Finally, abstract over the commonalities in this union. ∎

28.4   Extensional Equality with Cast

Even with the use of *null* to reduce the number of auxiliary methods by half, our solution for comparing two objects (for extensional equality) from a union requires too much work. Clearly, defining *same*ness is something that comes up time and again, and for that reason, Java provides two mechanisms for making this simple. The first one is a tool for inspecting the class of an object and the second one is for converting an object's type in-line:

inside of *Tea* :
boolean *same*(*IItem other*) {
  return (*other instanceof Tea* $_1$)
      && *((Tea)other)* $_2$.*same*(**this**);
}

The gray-shaded box with subscript 1 determines whether *other* is an instance of the *Tea* class, i.e., whether it is created with **new** *Tea* (or for some subclass of *Tea*). The gray-shaded box with subscript 2 is called a CAST. It tells the type checker to act as if *other* has type *Tea* even though its actual type is *IItem*, a supertype of *Tea*. Later when the program runs, the cast checks that *other* is indeed an instance of *Tea*. If it is not, the cast raises an error, similar to the original conversion method *toCoffee* in figure 114.[67] Fortunately, here we have already confirmed with *instanceof* that *other* is a *Tea* so nothing bad will happen.

Finally, since testing extensional equality is such a ubiquitous programming task, Java defines a default method and behavior:

inside of *Object* :
public boolean *equals*(Object *other*) {
  return this == *other*;
}

The class Object is the superclass of all classes, and if a class doesn't explicitly *extend* another class, Java makes it *extend* Object. Thus, all classes come with a useless **public** definition of *equals*. To obtain mostly useful behavior, you must publicly override *equals* and define extensional equality according to your needs. Since *all* classes provide *equals*, your method may safely call *equals* on all contained objects. For classes that you defined, *equals* behaves as defined; for others, it may incorrectly return false but at least it exists.[68]

28.5 Danger! Extensional Equality and Cycles

Figure 155 displays a Scheme-like data representation for list of ints. It includes a *setRest* method, which changes **this** list's *rest* field to some given value. In addition, the data representation supports a conventional *equals* method, designed according to the structural recipe.   not in interface!

It is easy to make up examples for testing the *equals* method:

---

[67]By doing so, ProfessorJ and Java guarantee that the object acts at run-time according to what its type promises. This guarantee is called TYPE SOUNDNESS or TYPE SAFETY.

[68]Although overriding *equals* alone doesn't violate any syntax or typing rules, it is technically wrong. Overriding *equals* makes it also necessary to override the method *hashCode*. Once you understand hashtables, read the manual to understand this constraint.

```
interface IList {
  // effect: change the rest of this list
  void setRest(IList rst);

  // is this list the same as the other?
  boolean equals(IList other);

  // the first int of this [non-empty] list
  int first();

  // the rest of this [non-empty] list
  IList rest();
}
```

```
class MT implements IList {
  MT() { }

  public void setRest(IList rst) {
    Util.error("empty has no first");
  }

  public boolean equals(IList other) {
    return other instanceof MT;
  }

  public int first() {
    return Util.error("empty has no first");
  }

  public IList rest() {
    return Util.error("empty has no rest");
  }
}
```

```
class Cons implements IList {
  private int fst;
  private IList rst;

  Cons(int fst, IList rst) {
    this.fst = fst;
    this.rst = rst;
  }

  public void setRest(IList rst) {
    this.rst = rst;
  }

  public boolean equals(IList other) {
    return other instanceof Cons
      && this.fst == other.first()
      && this.rest().equals(other.rest());
  }

  public int first() {
    return this.fst;
  }

  public IList rest() {
    return this.rst;
  }
}
```

Figure 155: Potentially cyclic lists

```
class Examples {
  IList alist = new Cons(1,new MT());
  IList list2 = new Cons(1,new MT());

  boolean test1 = check alist.equals(list2) expect true;
  boolean test2 = check alist.equals(new MT()) expect false;
  boolean test3 = check new MT().equals(alist) expect false;
}
```

To test cyclic lists, you also want to add a method that creates such lists. Here is one way of doing this, there are many others:

inside of *Examples* :
// create a cyclic list with one element
IList makeCyclic(int *x*) {
   IList tmp = **new** *Cons*(*x*,**new** *MT*());
   tmp.setRest(*tmp*);
   return *tmp*;
}

As you can see, the method creates a list and immediately uses *setRest* to set the rest of the list to itself. That is, the assignment statement in *setRest* changes *tmp* just before it is **return**ed.

With this method, you can introduce two obviously equivalent cyclic sample lists and check whether *equals* can compare them:

inside of *Examples* :
boolean *test4*() {
   IList clist1 = *makeCyclic*(1);
   IList clist2 = *makeCyclic*(1);
   return check *clist1.equals*(*clist2*) **expect** true;
}

If you now run the program consisting of this four classes, ProfessorJ does not stop computing. You must need the STOP button to get a response in the interactions window.

ProfessorJ:
Memory Limit

What just happened? The method *makeCyclic* is invoked twice. Each time it creates a list that consists of a single instance of *Cons* whose *rst* field points to the object itself:
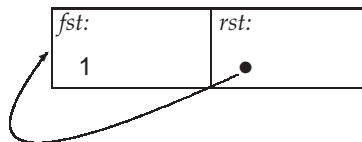
Since both *clist1* and *clist2* represent just such lists, we should thus expect that *clist1.equals*(*clist2*) produces true. To understand why it doesn't, let's step through the computation. According to the law of substitution, the method invocation evaluates to

return *clist2 instanceof Cons*
  && clist1.fst == clist2.first()
  && clist1.rest().*equals*(*clist2.rest*());

The expression uses *clist1* and *clist2* to represent the instances of *Cons*, because they are cyclic and not representable with **new** expressions directly.

The **return** statement contains a boolean expression, which in turn consists of three separate conditions. First, we must therefore evaluate *clist2 instanceof Cons*, which is true:

return true
  && clist1.fst == clist2.first()
  && clist1.rest().*equals*(*clist2.rest*());

Second, once the first part of an *&&* expression is true, we proceed to the second one, which with two steps reduces as follows:

return 1 == 1
  && clist1.rest().*equals*(*clist2.rest*());

Third, 1 == 1 also reduces to true, so we must evaluate the thid and last condition, which consists of three method calls. The key insight is that *clist1.rest*() evaluates to *clist1* and *clist2.rest*() evaluates to *clist2*. And thus we end up with

return *clist1.equals*(*clist2*);

In other words, we are back to where we started from. To determine the value of *clist1.equals*(*clist2*), we must determine the value of *clist1.equals*(*clist2*) and therefore the program runs forever.[69]

In a sense, the result isn't surprising. The first few sections in this chapter introduced the idea of cyclic data and the idea of programs that run forever or go into infinite loops. In the case where assignments exists only to create cyclic data, the solution is to ignore some links and to traverse the data as if they didn't exist. In the presence of general assignments (and changes to objects), this solution doesn't work. The programmer of any general traversal method—such as *equals*—must be prepared that the object graph contains a cycle and that a method designed according to the structural recipe does not compute a result.

The following exercises explore the topic in some depth and we strongly recommend you solve them. While they do not offer a complete solution,

---

[69]Technically, the program doesn't run forever in Java. Each method call in Java consumes a tiny bit of memory but after a while this adds up to all the memory on your computer. At that point, the program crashes and reports a "stack overflow" error. More on this topic later.

they explore the basic principle, namely to exploit intensional equality to define a general extension equality method. You will encounter this idea again in advanced courses on algorithms.

Exercises

Exercise 28.4 Add the following list to *Examples*:
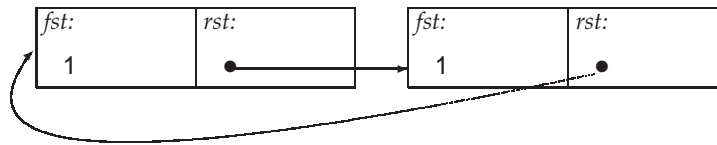
Compare it with the cyclic list containing one instance of *Cons* with 1 in the *fst* field. Are these two lists the same? If not, design a boolean-valued method *difference* that returns true when given the list with two *Cons* cells and false when given the list with one *Cons* cell. ∎

Exercise 28.5 In *How to Design Programs*, we developed several approaches to the discovery of cycles in a collection of data. The first one exploits accumulators, i.e., the use of an auxiliary function that keeps track of the pieces of the data structure it has seen so far. Here is the natural adaptation for our lists:

> inside of *IList* :
> // is **this** list the same as *other*, accounting for cycles?
> // accumulator: *seenThis*, the nodes already encountered on **this**
> // accumulator: *seenOther*, the nodes already encountered on *other*
> boolean *equalsAux*(*List other*, *IListC seenThis*, *IListC seenOther*);

That is, the auxiliary method *equalsAux* consumes **this** plus three additional arguments: *other*, *seenThis*, and *seenOther*. The latter are lists of *Cons* nodes that *equalsAux* has encountered on its way through **this** and *other*.

Design *IListC*, a data representation for lists of *Cons* objects. Then design the method *equalsAux*. While the problem already specifies the accumulator and the knowledge it represents, it is up to you to discover how to use this knowledge to discover cycles within lists of ints. Finally modify the definition of *equals*

> inside of *IList* :
> // is **this** list the same as *other*
> boolean *equals*(*List other*);

in *Cons* and *Mt* (as needed) so that it produces true for *test4* from above.
    Does your method also succeed on these tests:

inside of *Examples* :

```
boolean test5() {
  List clist1 = new Cons(1,new MT());
  List clist2 = new Cons(1,new MT());

  clist1.setRest(clist2);
  clist2.setRest(clist1);

  return
    check clist1.equals(clist2) expect true;
}
```

inside of *Examples* :

```
boolean test6() {
  List clist0 = new Cons(1,new MT());
  List clist1 = new Cons(1,clist0);

  List clist2 = new Cons(1,new MT());
  List clist3 = new Cons(1,clist2);

  clist0.setRest(clist1);
  clist2.setRest(clist3);

  return
    check clist1.equals(clist2) expect true;
}
```

Explain the examples in terms of box diagrams of lists. Think of additional ways to create lists with cycles and test your *equals* method. ∎

Exercise 28.6 A cyclic list, like a fraction with an infinite decimal expansion, has a period, i.e., it consists of a finite list that is repeated forever. Thus, another approach to determining the equality of two *IList*s is to determine their periods and to compare those. Design a method that uses this idea, but do solve exercise 28.6 first. ∎

Intermezzo 4: Assignments

purpose: the syntax of beginning student that is used in chapter IV
    interface, implements, inner classes

Purpose and Background

The purpose of this chapter is to introduce data abstraction in the context of class hierarchies. The focus is on abstracting over entire class diagrams that have a similar structure. The chapter introduces two-and-a-half techniques: an Object-based approach and an approach based on parametric polymorphism, also known as generics.

As in chapter III, the goal is to remind students that programming is *not* just the act of writing down classes and methods that work. True programming includes reasoning about programs, "editing" them, and improving them as needed.

We assume that students are at least vaguely familiar with the idea of creating simple abstractions from similar data definitions. Ideally, students should have read or studied Part IV of *How to Design Programs*.

After studying this chapter, students should be able to abstract over class diagrams with similar structure and use these common abstractions in place of the original ones.

TODO

Chapter III teaches you how to deal with classes that share fields and contain similar methods. To abstract over those similarities, you create superclasses and lift the common elements there, using inheritance to share them in many different classes. Like higher-order functions in *How to Design Programs*, superclass abstraction comes with many advantages.

You should therefore find it disturbing that this technique doesn't work for the case when everything but the types are shared. For example, we have seen lists of shots in our case study; you have created lists of worm segments for the worm game; you have designed data representations for restaurant menus, which are lists of menu items; and your programs have manipulated phone books, which are lists of names and phone numbers. At least to some extent, these lists are the same except that the type of list element differs from one case to another.

This chapter introduces techniques for abstracting over classes and systems of classes that differ in types not just in fields and methods. In terms of *How to Design Programs* (chapter IV), the chapter introduces you to the mechanisms of abstracting over data definitions; in contrast to Scheme, where you used English for this purpose, Java offers linguistic constructs in support of this form of abstraction. Once you know how to abstract over the types of data representations, you know how to design data libraries; in the last section of this chapter, we also discuss how to turn libraries into extensible frameworks.

Java 5:
Eclipse

30   Types and Similarities between Plain Classes

Over the course of the first few chapters, we have had numerous encounters with list items: log entries, menu items, phone records, team members, plain ints, and many more. Like lists themselves, entries are somehow similar, with differences mostly concerning types. We therefore begin our study

of abstraction over data with the simple case of list entries, demonstrating the ideas as much as possible before we move on to interesting cases, including lists.

```
+------------------+              +------------------+
| PhnBEntry        |              | MenuItem         |
+------------------+              +------------------+
| String name      |              | String name      |
| PhoneNumber pnum |              | int price        |
+------------------+              +------------------+
```

```
// an entry in a phone listing          // an entry on a restaurant menu
class PhnBEntry {                        class MenuItem {
  String name;                             String name;
  PhoneNumber phnu;                        int price;

  PhnBEntry(String name,                   MenuItem(String name,
            PhoneNumber phnu) {                     int price) {
    this.name = name;                        this.name = name;
    this.phnu = phnu;                        this.price = price;
  }                                        }
  ...                                      ...
}                                        }
```
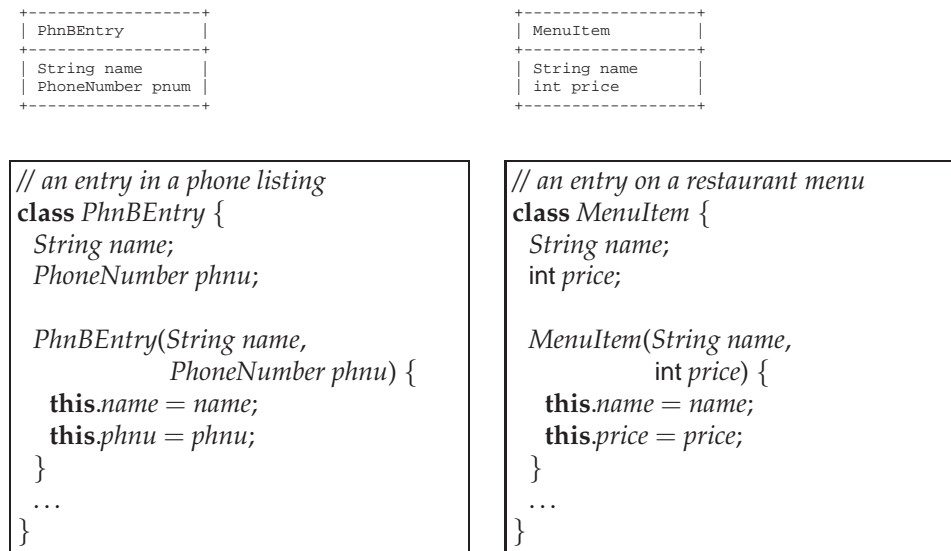
Figure 156: Entries in a phone book and menu items

30.1   Classes with Common Structure, Different Types

Figure 156 displays the class diagrams and the class definitions for two unrelated but similar fragments of classes:

1. *PhnBEntry*, which you should understand as the class of entries in an electronic phone book, like one of those found on your cell phone;

2. and *MenuItem*, which represents items on a restaurant menu.

The two classes have two fields each. One of those fields can be understood as a key and the other one is "information" associated with the key.

Specifically, the *name* field of *PhnBEntry* refers to the name of a phone acquaintance and the *phnu* field is the phone number. Typically, we would have a whole list of instances of *PhnBEntry* and we might wish to look for someone on the list or sort the list alphabetically, e.g., for display on a cell phone monitor. In a similar manner, *name* in *MenuItem* consists of the title

of some dish served in a restaurant and possibly a short description. You can imagine that a modern restaurant broadcasts its menu to your PDA and you can then search the menu, inspect its items for ingredients, or just sort it by price. The figure uses suggestive dots to indicate methods.

Clearly, the two classes are similar, and they will be used in a similar manner. If the types of the "information" fields didn't differ, we could use the design recipe from chapter III to create a common superclass with two fields. This new class would become a part of library and would then serve as the repository for all common methods of *Entry*. Programmers would derive specialized classes, such as *MenuItem* and *PhnBEntry* from this class and supplement additional, special-purpose methods. Our problem is that the types do differ, and therefore, the design recipe doesn't apply. Most object-oriented programming languages support at least one solution for this problem; Java allows two: via subtyping and via abstraction. The next two subsections compare and contrast these mechanisms, one at a time, and also display the advantages and disadvantages of each.

30.2   Abstracting Types via Subtyping

The first solution is to find a common supertype for the different types. In our example, the two relevant types are *PhoneNumber* and int. The first is a class used as a type, the second is a primitive type. In Java, these two types share one common supertype: Object,[70] the superclass of all classes that we encountered at the end of the preceding chapter.

Every object-oriented programming language has such a class. It hosts the most basic methods that are common to all objects. In Java, for example, the Object class hosts *equals* and *hashCode*, two related methods. When a class definition does not mention an **extends** clause, it is implicitly a subclass of Object. Hence, Object suggests itself as the type of the field for the superclass of *Entry*.

Reasoning along those lines leads to the class definition of *Entry* at the top of figure 157. Below the line in this figure, you see the two original classes defined as extensions of *Entry*; for such simple classes, it suffices that both constructors just pass their arguments to the **super** constructor. Re-defining the two classes in terms of *Entry* is, of course, just the last step in the design recipe for abstraction. It demands, after all, that we show how we can use abstractions to re-create the motivating originals.

---

[70]In Java, int corresponds to a class called *Integer* and it is a subclass of Object. The full story is too complex for this book. Research the terms "wrapper classes" and "implicit boxing" to understand this idea in full detail for Java. C# tells a simpler story than Java.

```
// a general entry for listings
class Entry {
  String name;
  Object value;

  Entry(String name, Object value) {
    this.name = name;
    this.value = value;
  }
}
```

```
class PhnBEntry extends Entry {
  PhnBEntry(String name,
            PhoneNumber value) {
    super(name,value);
  }
  ...
}
```

```
class MenuItem extends Entry {
  MenuItem(String name,
            int value) {
    super(name,value);
  }
  ...
}
```

Figure 157: Abstracting over entries via Object

Classes without methods are boring. Since phone books and menus need sorting, let's add the relevant methods to our classes. Sorting assumes that a program can compare the entries on each list with each other. More precisely, the representations for entries in a phone book and for items on a menu must come with a *lessThan* method, which determines whether one item is "less than" some other item:

inside of *PhnBEntry* :

```
// does this entry precede
// the other alphabetically?
boolean lessThan(PhnBEntry other) {
  return
    0 > name.compareTo(other.name);
}
```

inside of *MenuItem* :

```
// does this menu item cost
// cost less than the other?
boolean lessThan(MenuItem other) {
  return
    this.price < other.price;
}
```

The two method definitions are straightforward; the only thing worthy of attention is how to find out that one *String* is alphabetically less than some other *String*.[71]

---

[71]Search for *compareTo* in the Java documentation of *String*.

While the two *lessThan* methods in *PhnBEntry* and *MenuItem* are distinct and can't be abstracted, it is nevertheless necessary to check how to design them if the classes aren't designed from scratch but derived from *Entry*. Doing so reveals a surprise:

inside of *PhnBEntry*: (figure 157)
```
boolean lessThan(PhnBEntry other) {
  return
    0 > name.compareTo(other.name);
}
```

inside of *MenuItem*: (figure 157)
```
boolean lessThan(MenuItem other) {
  int thisPrice = (Integer)this.value;
  int othrPrice = (Integer)other.value;
  return thisPrice < othrPrice;
}
```

The definition in *PhnBEntry* is identical to the original one; the one for the derived version of *MenuItem*, however, is three lines long instead of one.

The additional lines are a symptom of additional complexity that you must understand. Ideally, you would like to write this:

inside of *MenuItem* :
```
boolean lessThan(MenuItem other) {
  return this.value < other.value;
}
```

where *value* is the name of the second, abstracted field, which replaces *price* from the original *MenuItem* class. Unfortunately, writing this comparison isn't type-correct. The *value* field has type Object in *MenuItem*, and it is impossible to compare arbitrary Objects with a mathematical comparison operator. Still, you—the programmer—know from the constructor that the *value* field always stands for an integer, not some arbitrary values.[72]

Casts—first mentioned in conjunction with Object—bridge the gap between the type system and a programmer's knowledge about the program. Remember that a cast tells the type checker to act as if an expression of one type has a different type, usually a subtype. Later when the program runs, the cast checks that the value of the expression is indeed an instance of the acclaimed type. If it is not, the cast signals an error.

Here we use casts as follows (or with variable declarations as above):

inside of *MenuItem* :
```
boolean lessThan(MenuItem other) {
  return (Integer)this.value ₁ < (Integer)other.value ₂;
}
```

---

[72] As a matter of fact, you don't really know. Someone derives a stateful subclass from your class and assigns "strange" objects to *value*. See **final** in your Java documentation.

The gray-shaded box labeled with subscript 1 requests that the type checker uses **this**.*value* as an *Integer* (i.e., an integer) value not an instance of Object, which is the type of the field-selection expression. The box labeled with subscript 2 performs the same task for the *value* field in the *other* instance of *MenuItem*. Once the type checker accepts the two expressions as *Integer*-typed expressions, it also accepts the comparison operator between them. Recall, however, that the type checker doesn't just trust the programmer. It inserts code that actually ensures that the *value* fields stand for integers here, and if they don't, this additional code stops the program execution.

In summary, the use of Object solves our "abstraction" problem. It should leave you with a bad taste, however. It requires the use of a cast to bridge the gap between the programmer's knowledge and what the type checker can deduce about the program from the type declarations. Worse, every time *lessThan* compares one *MenuItem* with another, the cast checks that the given value is an integer, even though it definitely is one.

While practical programmers will always know more about their programs than the type portion can express, this particular use of casts is arbitrary. Modern statically typed programming languages include proper mechanisms for abstracting over differences in types and that is what we study in the next subsection.

30.3   Abstracting Types via Generics

In *How to Design Programs* we encountered a comparable situation when we compared two different definitions of lists:

A *list of numbers* (LoN) is one of: 1. empty or 2. (cons ⃞*Number*⃞ LoN).	A *list of IRs* (LoIR) is one of: 1. empty or 2. (cons ⃞*IR*⃞ LoIR).

There, our solution was to introduce a data definition with a parameter, which we dubbed "parameteric" data definition:

A *[listof Item]* is one of:

1. empty or

2. (cons ⃞Item⃞ [listof Item]).

To use this definition, we need to supply the name of a class of data, e.g., (**listof** *String*), or another instantiation of a generic data definition, e.g., (**listof** (list *String Number*)).

Of course, in *How to Design Programs*, data definitions are just English descriptions of the kind of data that a program must process. Java[73] and similar object-oriented languages allow programmers to express such data definitions as a part of their programs, and the type checkers ensures that the rest of the program conforms to them.

```
// a parameteric entry for listings
class Entry<VALUE> {
  String name;
  VALUE value;

  Entry(String name, VALUE value) {
    this.name = name;
    this.value = value;
  }
}
```

```
class PhnBEntry
    extends Entry<PhoneNumber> {
  PhnBEntry(String name,
            PhoneNumber value) {
    super(name,value);
  }

  boolean lessThan(PhnBEntry other) {
    return 0 > name.compareTo(other.name);
  }
}
```

```
class MenuItem
    extends Entry<Integer> {
  MenuItem(String name,
            int value) {
    super(name,value);
  }

  boolean lessThan(MenuItem other) {
    return value < other.value;
  }
}
```

Figure 158: Abstracting over entries via generics

The top of figure 158 shows how to express a parameteric class definition in Java. There, and in other object-oriented languages, such definitions are called GENERIC CLASSES or just GENERICS. Roughly speaking, a generic is a class with one or more parameters. Unlike the parameters of a function

---

[73]This is true for version 1.5 of Java; prior releases of the language didn't include generics.

or a method, the parameters of a generic class range over types.

Thus, **class** *Entry<VALUE>* introduces a class definition with one parameter: *VALUE*. This type parameter shows up twice in the body of the class definition: once as the type of the *value* field and once as the type of the *value* parameter of the constructor.

Using a generic class such as *Entry* is like applying a function or a method. A function application supplies an argument for every parameter. In this spirit, the use of a generic class definition demands a type for every type parameter. For example, if you wish to use *Entry* as the type of some field, you must supply a type for *VALUE*. Here is an imaginary class *C* with an *Entry* field:

> **class** *C* {
> *Entry<String> twoSt*;
> . . .
> }

Similarly, if you wish to instantiate *Entry*, the **new** expression has to specify replacement types for *VALUE*. Continuing the *C* example, you might see this in a program:[74]

> **class** *C* {
> *Entry<String> twoSt* = **new** *Entry<String>*(`"hello"`,`"world"`);
> . . .
> }

Last but not least, if you wish to derive a subclass from *Entry*, you must supply a type for *VALUE*:

> **class** *C* **extends** *Entry<String>* { … }

Here, *String* is substituted for all occurrences of *VALUE*, meaning *C*'s superclass is a class with two *String*-typed fields. In general, this form of inheritance is just like the old one after the substitution has happened; it introduces all the visible fields and methods for use in the subclass.

The bottom of figure 158 displays two subclass definitions of *Entry*. On the left you see the re-creation of *PhnBEntry* from the generic *Entry* class; on the right is the re-creation of *MenuItem*. Unlike in the preceding subsection, the *lessThan* methods are just like those in the original classes. In particular,

---

[74]Leaving off the *String* part in the second *Entry* expression may actually work in Java. To understand why, you must understand how Java translates generics-based solutions into Object-based solutions; see the Intermezzo following this chapter.

MenuItem extends *Entry<Integer>*,[75] which is a class with two fields:

1. *name*, which has type *String*; and

2. *value*, which stands for *Integer*s.

Thus, it is correct to compare **this**.*value* with *other.value* in the *lessThan* method. After all, both values stand for integers, and it is okay to compare integers with $<$.

In summary, generics eliminate both the gap between what programmers know about the type of *value* and the run-time checks that the use of Object and casts in the preceding subsection imply. A generic class truly acts like a function from types to classes. Once the concrete type for the parameter becomes known, the resulting class uses the concrete types wherever the parameter shows up.

A second advantage of generic class abstraction over Object-based abstraction concerns methods. Remember that you may wish to sort lists of entries and to do so you need a method for comparing instances with a *lessThan* method. While the designer of *Entry* cannot know how to compare and rank instances of objects whose type isn't known yet, it is now possible to demand an appropriate method from each subclass:

> inside of *Entry<VALUE>* :
> // is **this** *Entry* less than the *other* one?
> **abstract** boolean *lessThan*(*Entry<VALUE>* other);

Of course, to make this work, you must add **abstract** to the definition of *Entry*, because the *lessThan* method can't have a definition.

Anybody who wishes to design a concrete subclass of *Entry<VALUE>* must provide a definition for *lessThan*. At first glance, you may think that these subclasses look just like those in figure 158. Although this is correct in principle, Java does not equate *PhnBEntry* with *Entry<PhoneNumber>* and *MenuItem* with *Entry<Integer>*.[76] Hence, in order to override the abstract *lessThan* method with a concrete one, both classes must repeat the instantiation of the superclass:

---

[75]Just like with casts, you must use *Integer* as a type when you wish to instantiate a generic class at type int.

[76]Java types are identified with the *names* of classes and interfaces, not with expressions that instantiate generic classes. This principle is called *nominal typing*.

inside of *PhnBEntry* :	inside of *MenuItem* :
boolean *lessThan*(*Entry*<*PhoneNumber*> *other*) {	boolean *lessThan*(*Entry*<*Integer*> *other*) {
return 0 > *name.compareTo*(*other.name*); }	**return** *value* < *other.value*; }

Now the signatures of *lessThan* in the concrete subclasses matches the signature of *lessThan* in the superclass, and Java accepts the code.

Exercises

Exercise 30.1 Design test cases for the original *PhnBEntry* and *MenuItem* classes. Then revise the classes in figure 158 according to the discussion and re-run the test cases for the abstracted solution. ∎

Exercise 30.2 Explain why adding

inside of *Entry (in figure 157)* :
// is **this** less than the *other*?
abstract boolean *lessThan*(Object *other*);

and making the class **abstract** doesn't express the same specification as the extension of *Entry*<*VALUE*>. To do so, design concrete subclasses of the revised *Entry* class that represent phone-book entries and menu items. ∎

When you define a generic class, you may use as many parameters as you wish:

```
class Pair<LEFT,RIGHT> {
  LEFT l;
  RIGHT r;

  Pair(LEFT l, RIGHT r) {
    this.l = l;
    this.r = r;
  }
}
```

This class definition says that a *Pair* consists of two pieces. Since we don't know the types of the two pieces, the definition uses parameters.

In the first four chapters, we have encountered many kinds of pairings, including the *Posn* class from the **geometry** library:

```
class Posn extends Pair<Double,Double> {
  Posn(double l,double r) { super(l,r); }

  // how far way is this Posn from the origin?
  double distanceToO() {
    return Math.sqrt(this.l*this. l+this. r*this. r);
  }
}
```

This *Posn* class is a subclass of *Pair<Double,Double>*,[77] that is, it inherits two fields that represents double-precision numbers. The constructor expresses this fact with two parameters, both of type double. The subclass definition also includes a method definition for *distanceToO*, which computes the distance of **this** instance of *Posn* to the origin. Because the superclass's fields have type double, the instructions in the body of *distanceToO* are exactly the same as before; there is no need for casts or other changes.

A moment's reflection suggests that *Pair* is also a natural superclass for *Entry* because an entry pairs a string with a value:

```
class Entry<VALUE> extends Pair<String,VALUE> {
  Entry(String l,VALUE r) {
    super(l,r);
  }
}
```

Naturally, *Entry* remains a class definition of one type parameter. It extends the *Pair* class with *Pair<String,VALUE>*. The first parameter here is *String*, the type of the *name* as we have called the field so far. The second parameter, though, is *VALUE*, which is the parameter of the declaration. Passing on a parameter isn't a problem of course; we have seen this in *How to Design Programs* with functions:

```
(define (g u v) ... )
(define (f x) (g x 10))
```

and in this book with methods:

```
inside of Posn :
double distance(Posn other) {
  return distanceToO(new Posn(this.l-other.l,this.r-other.r));
}
```

---

[77] *Double* is to the type double what *Integer* is to the type int.

In short, type parameters of generic classes are really just like parameters of functions and methods.

Exercises

Exercise 30.3 Create instances of *Pair<String,Integer>* and *Posn*s. Is an instance of *Pair<Double,Double>* also of an instance of *Posn*?

Explore why this expression

new *Posn*(1.0,2.0) *instanceof Pair<Double,Double>*

is illegal in Java. Note: This part is a challenging research exercise that requires a serious amount of reading for a full understanding. ∎

Exercise 30.4 Develop a version of *Pair* that uses Object as the type of its fields. Then extend the class to obtain a definition for *Posn*, including its *distanceToO* method. ∎

Even though *Pair* and *Entry* are trivial examples of abstractions over similar classes, they teach us a lot about abstracting via Object and generics. Suppose we had turned *Pair* into a library class and wanted to add some basic methods to it, i.e., methods that apply to all instances of *Pair* no matter what the concrete field types are. One of these methods is *getLeft*:

inside of *Pair<LEFT,RIGHT>* :
```
// the l field of this Pair
LEFT getLeft() {
  return this.l;
}
```

If *l* is a **private** field, *getLeft* provides access to its current value. Interestingly enough the method's signature communicates to any reader what the method does. It consumes one argument, **this** of type *Pair<LEFT,RIGHT>* and it returns a value of type *LEFT*—regardless of what these types really are. Since the given instance contains only one value of type *LEFT*, the method can only extract this value.[78]

A slightly more interesting example is *swap*, which produces a new pair with the values reversed:

---

[78]Technically, the method could also use the *null* value as something of type *LEFT*, but remember that *null* should only be used in certain situations.

inside of *Pair<LEFT,RIGHT>* :
// create a pair with the fields in the reverse order of **this**
public *Pair<RIGHT,LEFT> swap*() {
  return new *Pair<RIGHT,LEFT>*(r,l);
}

The method creates a new pair using the values in *r* and *l* in this order. Just as with *getLeft*, the signature *Pair<RIGHT,LEFT>* of *swap* actually reveals a lot about its computation. Given this instance of *Pair* whose *l* and *r* fields are of type *LEFT* and *RIGHT*, respectively, it creates an instance of *Pair<RIGHT,LEFT>*. Naturally, without any knowledge about the actual types, it can only do so by swapping the values of the two types to which it has guaranteed access.

Exercise

Exercise 30.5 Design (regular) classes for representing lists of *MenuItem*s and *PhnBEntry*s. Add methods for sorting the former by price and the latter by alphabet. ∎

31   Types and Similarities between Hierarchies

Pairs and list entries are small, artificial examples that don't make it into widely usable libraries. Lists or any compound data, however, are an entirely different matter. In the object-oriented world, lists and similar classes are called COLLECTION classes, or just collections, because they allow programs to collect a (n often arbitrary) number of objects in one. Abstracting over the types of the objects in such collections creates a general and powerful programming tool.

Take a look at figure 159. It compares the class diagrams for electronic phone books (lists of *PhnBEntry*s) and for menus (lists of *MenuItem*s). The two figures display the kind of diagram that we encountered in chapter 1, where we focused on representing information as data. A look from afar suggests that the diagrams are identical: three boxes in each; one interface with two implementing classes; two inheritance connections; and one containment arrow, from the second field of the box with two fields to the interface. Only a close-up look reveals differences in type and class names, which is why we dub this situation a "structural similarity."

If we perceive the diagrams as data definitions for lists, it is easy to ignore three of the four differences. After all, it is up to us to choose the names
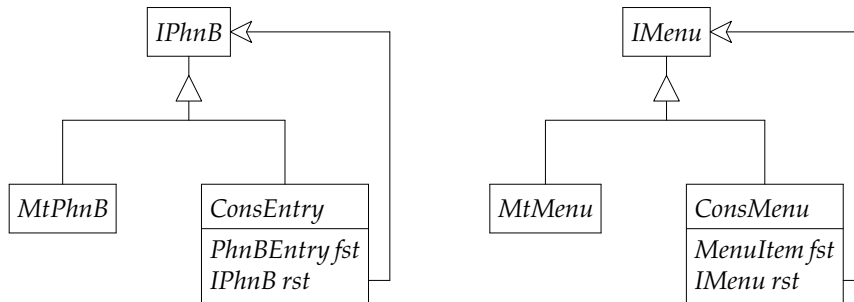
Figure 159: Class diagrams for two different lists

for the common list interface and the two implementing classes. Even after we agree on some consistent naming scheme, however—say, *IList*, *Cons*, and *Mt*—we are left with a difference: *PhnBEntry* versus *MenuItem*, the types of the items on the list.

This situation should remind you of subsection 30.2 where two classes differed in one type and we abstracted via a common supertype. Or you could think of subsection 30.3 where the process exploited generic classes. The big difference is that we are now abstracting in the context of a system of classes, not just an individual class.

31.1 Abstracting Types via Subtyping, Part 2

The top of the left column of figure 160 shows the class diagram for an abstraction of the two lists in figure 159. The data definition exploits subtyping. In other words, each occurrence of a distinguishing type in the original diagrams is replaced with Object, the common supertype. The bottom shows the complete class definitions that go with this diagram.

Of course, creating this general list representation is only half the design process. At a minimum, we must also demonstrate that we can re-define the classes of figure 159 in terms of this general list representation. Before we do this, let's create some examples; see the right side of figure 160:

1. *mt* is introduced as a name for the empty list;

2. *example1* is a list of one Object;

3. *example1* is a list of *MenuItem*s;

4. *example3* is a list of two *String*s.

```
               ┌──────┐◁─────────────┐
               │ IList│              │
               └──────┘              │
                  △                  │
          ┌───────┴────────┐         │
    ┌─────┐          ┌──────────┐    │
    │ Mt  │          │ Cons     │    │
    └─────┘          ├──────────┤    │
                     │Object fst│────┘
                     │IList rst │
                     └──────────┘
```

```
// a list of Objects
interface IList {}
```

```
// an empty list
class Mt implements IList {
  Mt() { }
}
```

```
// adding an Entry to a list
class Cons implements IList {
  Object first;
  IList rest;

  Cons(Entry first, Object rest) {
    this.first = first;
    this.rest = rest;
  }
}
```

```
class Examples {
  IList mt = new Mt();

  IList example1 =
    new Cons(new Object(),
      mt);

  MenuItem pasta =
    new MenuItem("Pasta",12);
  MenuItem pizza =
    new MenuItem("Pizza",11);
  IList example2 =
    new Cons(pasta,
      new Cons(pizza,
        mt));

  IList example3 =
    new Cons("hello",
      new Cons("world",
        mt));

  IList example4 =
    new Cons(pasta,
      new Cons(1,
        new Cons("a",
          mt)));
}
```

Figure 160: *Object*-based lists

5. and last but not least *example4* is a list consisting of three objects created from three distinct classes.

These examples show that lists of Objects may be HOMOGENEOUS, containing objects that belong to one type, or HETEROGENEOUS, containing objects that belong to many different types. Hence, if one of your methods extracts an item from such a list of Objects, it may assume only that it is an instance of Object. Using the instance in any other way demands a cast.

Our next step is to add some basic methods to our general representation of lists, just to explore whether it is feasible to turn these classes into a generally useful library. If so, it is certainly a worthwhile effort to abstract from the representation to lists of *PhnBEntry*s and *MenuItem*s.

Remember some of the simplest functions that work on all possible lists: *length*, which counts how many objects are on the list; *contains*, which checks whether some given object is on the list; and *asString*, which renders the list as a string:

```
interface IList {
  // how many objects are on this list?
  int count();
  // is the given object o on this list?
  boolean contains(Object o);
  // render this list as a String
  String asString();
}
```

We have turned this wish list into method signatures and purpose statements for our new interface, following our standard method design recipe.

Remember that when there is demand for several methods, you are usually best off developing a general template and then filling in the specific methods:

inside of *Mt* :	inside of *Cons* :
??? meth() {	*??? meth*() {
return ...	**return** ... **this**.*first* ... **this**.*rest.meth*()
}	}

Keep in mind that the expression **this**.*first* reminds you of the value in the *first* field and the fact that your method can call methods on *first*.

Filling in those templates for the first two methods is straightforward:

1. *count*:

inside of *Mt* :	inside of *Cons* :
int *count*() {	int *count*() {
return 0;	**return** 1 + **this**.*rest.count*();
}	}

Here **this**.*first* isn't used; just its presence counts.

2. *contains*:

inside of *Mt* :	inside of *Cons* :
boolean *contains* (Object *o*) {	boolean *contains*(Object *o*) {
return false;	**return** **this**.*first.equals*(*o*)
}	|| **this**.*rest.contains*(*o*);
	}

In this case, the method invokes *equals* on **this**.*first* to find out whether it is the object in question. Since the Object class defines *equals*, the method is available in all classes, even if it may not compute what you want.

3. *asString*:

The completion of the definition for *asString* demands some examples. Say you had to render the following two lists as strings:

> *IList* empty = **new** *Mt*();
> *IList alist* = **new** *Cons*("hello",**new** *Cons*("world",empty));

One possibility is to just concatenate the strings and to use the empty string for instances of *Mt*:

> *checkExpect*(*empty.asString*(),"")
> *checkExpect*(*alist.asString*(),"helloworld")

This isn't pretty but simple and acceptable, because the goal here isn't the study of rendering lists as *String*s.

As it turns out, the Object class not only defines *equals* but also the *toString* method, which renders the object as a string. Hence, all classes support *toString*, because they implicitly extend Object. The *String* class overrides it with a method that makes sense for *String*s; you are responsible for the classes that you define. In any case, invoking *toString* on **this**.*first* and concatenating the result with **this**.*rest.asString*() is the proper thing to do:

inside of *Mt* :
```
String asString() {
  return "";
}
```

inside of *Cons* :
```
String asString() {
  return
    first.toString()
      .concat(rest.asString());
}
```

Exercises

Exercise 31.1 Use the examples to develop tests for the three methods. Modify *asString* to become the *toString* method of *IList*. ∎

Exercise 31.2 Design *asString2*, which renders a list as a comma-separated enumeration surrounded with opening ("[") and closing ("]") brackets. For example, *asString2* would render *alist* from above as `"[hello,world]"`. Hint: this requires an auxiliary method for rendering a list for which the evaluation context has already added an opening bracket. When you have finished the first draft, look for opportunities to abstract. ∎

Exercise 31.3 Design an extension of *Posn* with *equals* and *toString* methods. Then use the Object-based list representation of this section to create lists of such objects and develop tests that validate that the methods work. ∎

Now we know how to use the general list representations to create examples, and how to add basic methods, we have turned it into a true library. Following the design recipe for abstraction, our next task is to demonstrate that these general lists suffice to represent menus and phone books. The obvious problem is that, in contrast to the *Entry* example, our list representation consists of more than one class: *IList*, *Cons*, and *Mt*. In short, if we wish to construct a menu representation based on the list representation, we need to extend an entire class hierarchy—sometimes called a framework—not just a single class. More on this below.

From the perspective of class diagrams, the solution is actually natural. The original menu representation looks just like a list, consisting of an interface and two classes: *IMenu*, *MtMenu*, and *ConsMenu*. These three pieces play roles that are analogous to *IList*, *Mt*, and *ConsMenu*. Thus, if we wish to construct a menu representation from the list library, it appears best to derive *IMenu* from *IList*, *MtMeny* from *Mt*, and *ConsMenu* from *Cons*. Figure 161 shows how a matching class diagram; the corresponding pieces are connected via inheritance arrows. In addition, *MtMenu* and *ConsMenu* both implement *IMenu*, which is the type of the menu representation.

The translation of the diagram into code isn't possible, however, with what you know. Doing so requires the introduction of a new—but fortunately simple—concept: the derivation of one interface from another. In our example, the code for this INTERFACE INHERITANCE looks like this:

interface *IMenu* **extends** *IList* { … }

Here the **extends** clause means that *IMenu* inherits everything from *IList*. In this specific case, *IMenu* inherits three method signatures. Otherwise the code is ordinary; see figure 162.

```
                                            // The Object List Library
            +---------------------------+
            | IList                     |<--------------------+
            +---------------------------+                     |
            +---------------------------+                     |
            | int count()               |                     |
            | boolean contains(Object x)|                     |
            | String asString()         |                     |
            +---------------------------+                     |
                          |                                   |
                         / \                                  |
                         ---                                  |
                          |                                   |
            -----------------------------------               |
            |                   |           |                 |
  +---------------------+       |   +-----------------------+ |
  | Mt                  |       |   | Cons                  | |
  +---------------------+       |   +-----------------------+ |
            |                   |   | Object first          | |
            |                   |   | IList rest            |-+
            |                   |   +-----------------------+
            |                   |              |
           / \                  |             / \
           ---                  |             ---
            |                   |              |
 =========================================================================
            |                   |         // The Menu Application
            |                   |
            |         +--------------------------------+ |
            |         | IMenu                          | |
            |         +--------------------------------+ |      +-------------+
            |         +--------------------------------+ |      | MenuItem    |
            |                   |                        |      +-------------+
            |                  / \                       |      | String name |
            |                  ---                       |      | int price   |
            |                   |                        |      +-------------+
            |         -----------------------------------
            |         |                          | |
+-----------------------+    +------------------------------+
| MtMenu                |    | ConsMenu                     |
+-----------------------+    +------------------------------+
```
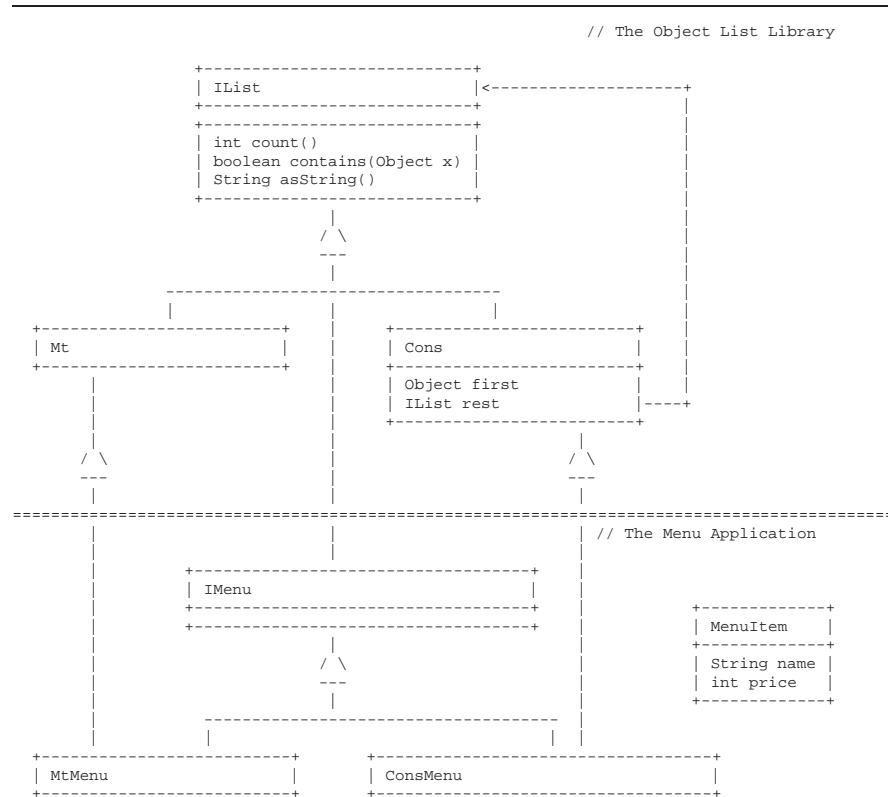
Figure 161: Representing menus based on *Object* lists

> // a list of menu items
> **interface** *IMenu* **extends** *IList* { }

```
// an empty menu              // adding an item to a menu
class MtMenu extends Mt       class ConsMenu extends Cons
  implements IMenu {            implements IMenu {

 MtMenu() {                     ConsMenu(MenuItem first, IMenu rest) {
  super();                       super(first,rest);
 }                              }
}                             }
```
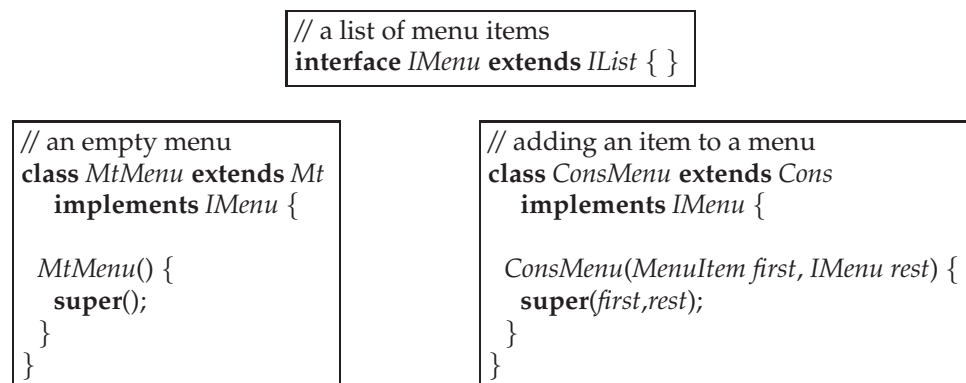
Figure 162: Creating menus via lists of *Object*s

Exercises

Exercise 31.4 Create a sample list of *MenuItem*s using the implementation of the data representation in figure 162. Then use the *count*, *contains*, and *asString* methods from the general list library to develop tests. ∎

Exercise 31.5 Define the representation of phone books as an extension of the general list library. Define *PhoneNumber* as a simple stub class and develop tests for *count*, *contains*, and *asString*. Consider reusing those of exercise 31.4. Why is this possible? ∎

31.2 Abstracting Types via Subtyping plus Interfaces

The work in the preceding section leaves us with representations of menus and phone books that do not support sorting. One idea is to add sorting methods to the specific classes. This idea poses a challenge (see section 34.2), however, and doesn't fit with the goal of putting the commonalities of menus and phone books into a single place. After all, both menus and phone books demand sorting, so the appropriate methods should really reside in the abstracted code, not in specific data representations.

Sorting requires a method for comparing the objects that we wish to sort. For data such as numbers or strings, the desired comparisons are obvious. For other kinds of objects, however, comparisons don't even exist. Consider instances of this class

```
class Add200 {
  public AddFunction() {}

  public int apply(int x) {
    return x + 200;
  }
}
```

whose purpose it is to represent functions. It doesn't make any sense to compare such functions or to sort them.

Put positively, you can only sort lists of objects if the objects also support a comparison method that determines when one object is less than some other object. In particular, the data definition of figure 160 is not quite appropriate because it allows all kinds of objects to be stored in a list. If our list abstraction is to include a sorting method, we must insist that the members of the list are comparable.

```
interface IList {
  // sort this list, according to lessThan
  IList sort();
  // insert o into this (sorted) list
  IList insert(IComp o);
}
```

```
interface IComp {
  // is this object less than o?
  boolean lessThan(Object o);
}
```

```
class Mt implements IList {
  Mt() {}

  public IList sort() {
    return this;
  }

  public IList insert(IComp o) {
    return new Cons(o,this);
  }
}
```

```
class Cons implements IList {
  IComp first;
  IList rest;

  Cons(IComp first, IList rest) {
    this.first = first;
    this.rest = rest;
  }

  public IList sort() {
    return rest.sort().insert(first);
  }

  public IList insert(IComp o) {
    if (first.lessThan(o)) {
      return new Cons(first,rest.insert(o)); }
    else {
      return new Cons(o,this); }
  }
}
```

Figure 163: Lists based on subtyping and interfaces, with sorting

In Java—and related languages—you use interfaces to express such requirements. Specifically, the *Cons* class does not use Object as the type of *first* but *IComp*, an interface that demands that its implementing classes support a method for comparing and ranking its instances. The top-right of figure 163 displays this interface; its single method, *lessThan*, is like *equals* in Object, in that it consumes an Object and compares it with **this**.

Other than the small change from Object to *IComp* the class definitions for representing lists and sorting them is straightforward. The interface and class definitions of figure 163 are basically like those in figure 160. The

methods for sorting such lists are straightforward. Their design follows the standard recipe, producing code that looks just like the one in section 15.2.

```
class MenuItem implements IComp {
  String name;
  int value;

  MenuItem(String name, int value) {
    this.name = name;
    this.value = value;
  }

  public boolean lessThan(Object o) {
    MenuItem m;
    if (o instanceof MenuItem) {
      m = (MenuItem)o;
      return this.value < m.value; }
    else { … }
  }
}
```

Figure 164: Comparable objects

Given a representation of sortable lists, we can now return to the problem of representing menus and phone books. Assuming that you are designing those from scratch, you start with classes for menu items and phone book entries and you add **implements** *IComp* clauses to each, because this is what a re-use of the chosen list library demands. This leaves you with just one question, namely, how to equip *MenuItem* and *PhnBEntry* with a *lessThan* method. The interface provides the signature and the purpose statement; here are examples:

new *MenuItem*(`"Pizza"`,10).*lessThan*(**new** *MenuItem*(`"Pasta"`,12))

should be true, because *lessThan* compares prices. Here is an unusual one:

new *MenuItem*(`"Pizza"`,10).*lessThan*(**new** *PhnBEntry*(…))

You have two choices here. The first is to signal an error, because it is impossible to compare *MenuItem*s with *PhnBEntry*s. The second is to force *lessThan* to produce some sensible result, e.g., false.

The definition of *lessThan* in figure 164 can accommodate any decision you make. The method tests whether the other object belongs to *MenuItem*

and, if so, casts it to one and compares the values of **this** and the given object. Note how the cast is performed via an assignment to a local variable.

Exercises

Exercise 31.6 Complete the definition of *lessThan* in figure 164. Also design a class for representing phone book entries so that you can reuse the library of sortable lists to represent phone books. Use the two classes to represent menus and phone books. ▪

Exercise 31.7 Is it possible to use Object as the type of *insert*'s second parameter? What would you have to change in figure 163? ▪

Exercise 31.8 Design *Apple* and *Orange* classes that represent individual fruits (weight, ripeness, etc). Both classes should implement *IComp* where *lessThan* returns false if the second input belongs to the wrong kind of class. Then show that it is possible to compare *Apple*s and *Orange*s. Is this good or bad for programming?

Next consider the proposal of changing the type of *lessThan*'s second argument to *IComp*:

```
class IComp {
  // is this object less than o?
  boolean lessThan(IComp other);
}
```

Specifically ponder the following two questions:

1. Is our original solution in figure 163 more general than this one?

2. Does this definition prevent the comparison of *Apple*s and *Orange*s?

Conduct coding experiments to find the answers.

Finally, read the sections on *Comparable* and *Comparator* in the official Java documentation. ▪

Problem is, the list library in figure 163 forces you to duplicate code. Specifically, the library duplicates the functionality of the original list library in figure 160. Because the new *Cons* class compounds objects of type *IComp*, you can no longer use this new library to represent lists of objects

```
interface IList {
  // sort this list, according to lessThan
  IList sort();
  // insert o into this (sorted) list
  IList insert(IComp o);
}
```

```
interface IComp {
  // is this object less than o?
  boolean lessThan(Object o);
}
```

```
class Mt implements IList {
  Mt() {}

  public IList sort() {
    return this;
  }

  public IList insert(IComp o) {
    return new Cons(o,this);
  }
}
```

```
class Cons implements IList {
  Object first;
  IList rest;

  Cons( Object first, IList rest) {
    this.first = first;
    this.rest = rest;
  }

  public IList sort() {

    return rest.sort().insert( (IComp)first );

  }

  public IList insert(IComp o) {
    if (other.lessThan(first)) {
      return new Cons(other,this); }
    else {
      return new Cons(first,rest.insert(other)); }
  }
}
```

Figure 165: Lists based on subtyping and casts, with sorting

whose class doesn't implement *IComp*. Worse, if programmers keep making up list representations for special kinds of objects, say *IStringable* for things that can be rendered as *String*s or *IDrawable* for objects that can be drawn into a *Canvas*, then this form of duplication quickly proliferates.

If you are in charge of both libraries, a solution is to continue using Object as the type of the list elements. Figure 165 displays the revised library code. The gray-shaded boxes highlight the changes. In particular, the type of *Cons*'s *first* field is Object. Since *insert* is invoked from *sort* using the value

in the *first* field, the argument must be cast from Object to *IComp*; see the framed and gray-shaded box. This cast allows the type checker to bless the use of *lessThan* on the current *first* field but also performs a check at run-time that the given Object implements *IComp*. If it doesn't, the program signals an error and the evaluation stops. Otherwise evaluation continues with the method invocation, using the existing object unchanged.

Exercises

Exercise 31.9 Demonstrate that the list library of figure 165 can represent menus and phone books. Add *count*, *contains*, and *asString*. ∎

Exercise 31.10 Explain the error that occurs when you try to represent a non-empty list of instances of *Add200* using the library in figure 163.

Now use the library in figure 165 to represent the same list. Can you sort the lists? What kind of error do you encounter now? ∎

31.3   Abstracting Types via Generics, Part 2

Figure 166 displays a minimal generic list representation. Like in figure 160, the top of the left side is a diagram of the library, and the bottom shows the actual interface and class definitions. The diagram uses one new element of diagramming: the parameter box; its purpose is to say that the framed class hierarchy is parameterized over the name in the box. In the figure, the three boxes of the list representation are parameterized over *ITEM*; currently this parameter shows up only as the type of the *fst* field in *Cons*.

The interface definition on the right of figure 166 uses a type parameter in its definition, just like the classes of figure 158. Generic interfaces are analogous to generic classes, meaning you typically use the notation in one of two ways:

1. *IList<Type>*, where *Type* is an actual type (but use *Integer* for int); or

2. *IList<PARA>*, where *PARA* is a type parameter bound elsewhere.

More concretely, if you wish to introduce a field that stands for a list of *String*s into some *Examples* class, you must use the second form like this:

   IList<String> = . . .

```
                              ┌──────┐
                              │ ITEM │
  ┌───────────────────────────┴──────┴──┐
  │          ┌───────┐                   │
  │          │ IList │◁──────┐           │     ┌─────────────────────────────────┐
  │          └───┬───┘       │           │     │ class Examples {                │
  │              △           │           │     │  IList<MenuItem> mt =           │
  │       ┌──────┴──────┐    │           │     │    new Mt<MenuItem>();          │
  │       │             │    │           │     │                                 │
  │   ┌───────┐    ┌─────────┴─┐         │     │  MenuItem pasta =               │
  │   │  Mt   │    │   Cons    │         │     │    new MenuItem("Pasta",12);    │
  │   └───────┘    ├───────────┤         │     │  MenuItem pizza =               │
  │                │ ITEM fst  │         │     │    new MenuItem("Pizza",11);    │
  │                │ IList rst │         │     │  IList<MenuItem> menu =         │
  │                └───────────┘         │     │    new Cons<MenuItem>(pasta,    │
  └─────────────────────────────────────┘     │      new Cons<MenuItem>(pizza,  │
                                               │      mt));                      │
  ┌─────────────────────────────────────┐     │                                 │
  │ interface IList< ITEM > { }         │     │  IList<String> los =            │
  └─────────────────────────────────────┘     │    new Cons<String>("hello",    │
                                               │      new Cons<String>("world",  │
  ┌─────────────────────────────────────┐     │      new Mt<String>()));        │
  │ class Mt<ITEM>                      │     │                                 │
  │   implements IList<ITEM> {          │     │  IList<Object> loo =            │
  │  Mt() { }                           │     │    new Cons<Object>("hello",    │
  │ }                                   │     │      new Cons<Object>(pasta,    │
  └─────────────────────────────────────┘     │      new Mt<Object>());)        │
                                               │ }                               │
  ┌─────────────────────────────────────┐     └─────────────────────────────────┘
  │ class Cons<ITEM>                    │
  │   implements IList<ITEM> {          │
  │  ITEM first;                        │
  │  IList<ITEM> rest;                  │
  │                                     │
  │  Cons(ITEM first, IList<ITEM> rest) {│
  │    this.first = first;              │
  │    this.rest = rest;                │
  │  }                                  │
  │ }                                   │
  └─────────────────────────────────────┘
```

Figure 166: A generic list

where the . . . are replaced with an actual list. Analogously, if a method specifies that its argument must be lists of strings, e.g.,

int *count*(*IList*<*String*> *los*) { . . . }

then this method must be applied to lists of the exact same type.

The second usage shows up in the **implements** clauses of *Mt* and *Cons* where *IList* is applied to the type parameter of the defined class. That is, the type parameter itself is bound in the position following the class names, i.e., in *Mt*<*ITEM*> and *Cons*<*ITEM*>. Naturally, these type parameters can be systematically renamed, just like function or method parameters, without changing the meaning of the interface or class definition. Thus,

        class *Mt<XIT>* **implements** *IList<XIT>* { … }

would be a feasible alternative for the definition of *Mt* in figure 166.

    Following the design recipe of chapter I, we must next create instances of these classes. From section 30.3 you know that this means that the constructor must be applied to concrete types; figure 166 presents a class of examples:

mt: To create an empty list, you must supply a concrete type for *ITEM*, the type parameter in the class definition.

menu: To extend an existing list with an additional *MenuItem*, your program provides *Cons* with the type of the item it adds.

los: To construct a list of *String*s instead of *MenuItem*s, it is necessary to use instantiations of *Cons* and *Mt* at *String*.

loo: To build a list from instances of different classes, instantiations of *Cons* and *Mt* must use a supertype of all involved types (Object in this case).

    In other words, we can build all the lists in a parametric manner that we built from the Object-based list library. Of course, if we now take any item from this list, it has type Object and must first be cast as a *String* or a *MenuItem* if we wish to invoke a specific method on it.

As you can tell from this enumeration, creating instances of a parametric list representation uses a more complex notation than creating instances of a list representation that uses subtyping (see figure 160). The additional complexity buys you TYPE HOMOGENEOUS lists. Thus, if some methods removes an item from *menu*, the type of the item is *MenuItem* because the lists is created from *Cons<MenuItem>* and *Mt<MenuItem>*; in contrast to the Object-based list, no cast is needed. Also, while the creation of data is more complex than in the Object-based representation, the design of methods is as straightforward as before. We therefore leave it as an exercise.

Exercise

Exercise 31.11 Here is a wish list for the generic list implementation:

    inside of *IList<ITEM>* :
    // how many objects are on **this** list?
    int *count*();

```
// is the given object o on this list (comparing via equals)?
boolean contains(ITEM o);

// render this list as a String, using toString on each item
String asString();
```

Design the methods. Explain how the *contains* method in *Cons* works via some of the examples of *IList<ITEM>* from above. ∎

Exercise 31.12 Demonstrate that this parameteric list library can be used to re-define the classes for representing menus and phone books (without sorting functionality). ∎

31.4   Abstracting Types via Generics plus Interfaces

Again, we have done our basic homework in terms of the design recipe: we can represent lists using *IList<ITEM>* and we can add methods that don't need any specific information about the *ITEM*s that the list contains. Next we need to demonstrate that we can truly re-define the lists from which we started, including the ability to sort them.

Following section 31.2, we must obviously restrict what kinds of objects a list may contain. In particular, the items must be comparable with each other in the spirit of a *lessThan* method. Because the library consists of several classes and interfaces, we must ensure that this constraint applies to all of them. Furthermore, we must communicate this constraint to all future users of this library.

To this end, generic interfaces and classes support the specification of obligations on type parameters. The syntax for doing so is whimsical:

interface *Ifc* $< T$ **extends** $S >$ { … }

class *Cls* $< T$ **extends** $S >$ { … }

That is, instead of just specifying a type parameter, the creator of an interface or class can also demand that the parameter T must be a subtype of some other type S. Thus, while the keyword **extends** reminds you of subtyping for classes and interfaces, it also applies to subtyping between classes and interfaces in this one context.[79]

---

[79]Designing the concrete syntax of programming languages is not easy. It also tends to cause more debate than deserved. We simply accept the decision of the Java designers.

```
// a list of items of type I
interface IList< I extends ICompG<I> > {
  // sort this list, according to lessThan
  IList<I> sort();
  // insert o into this (sorted) list
  IList<I> insert(I o);
}
```

```
// comparable objects
interface ICompG<T> {
  // is this object less than other?
  boolean lessThan(T other);
}
```

```
class Mt< I extends ICompG<I> >
    implements IList<I> {
  Mt() { }

  public IList<I> sort() {
    return this;
  }

  public IList<I> insert(I other) {
    return new Cons<I>(other,this);
  }
}
```

```
class Cons< I extends ICompG<I> >
    implements IList<I> {
  I first;
  IList<I> rest;

  Cons(I first, IList<I> rest) {
    this.first = first;
    this.rest = rest;
  }

  public IList<I> sort() {
    return rest.sort().insert(this.first);
  }

  public IList<I> insert(I o) {
    if (first.lessThan(o)) {
      IList<I> r = rest.insert(o);
      return new Cons<I>(first,r); }
    else {
      return new Cons<I>(o,this); }
  }
}
```

Figure 167: Generic lists, with sorting

Consider the following example:

```
interface IList <I extends IComp> {
  IList<I> sort();
  IList<I> insert(I o);
}
```

where *IComp* is a reference to the interface from figure 165. It demands that you apply *IList* to subtypes of *IComp*. Thus, each item that *insert* consumes must implement a *lessThan* method. Similarly,

```
class Cons <I extends IComp> extends IList<I> {
  I first;
  IList<I> rest;
  Cons(I first, IList<I> rest) {
    this.first = first;
    this.rest = rest;
  }
  . . .
}
```

is a class definition that extends *IList*. It does so at type *I*, which is the type parameter of the class definition and which is also required to be a subtype of *IComp*. The body of the class shows that every value in the *first* field is of type *I* and, due to the constraint on *I*, any method within *Cons* can invoke *lessThan* on *first*. Since the *lessThan* method is applicable to any other Object, *first* can now be compared to everything.

In figure 167 you can see three applications of this new syntax, all highlighted in gray. Together they enforce that a list contains only comparable objects. The first one comes with the *IList* interface itself; it specifies that its type parameter must be below some type like *IComp*, i.e., an interface that demands comparability. (For now, ignore the actual interface used in the constraint.) The informal description uses the type parameter to explain why it exists, and the formal restriction tells us that the list consists of comparable objects. It is possible, however, to implement this specification in a way that violates the intention behind the purpose statement and the type parameter.

While the interface does not determine the shape of the list, it does imply two guarantees for all designers who work with *IList*. First, the designer of every implementation of *IList* must ensure that the class satisfies this constraint on the type parameter, too. Second, the designer of every *insert* method may assume that the method's second argument implements the specified interface. As you can see below, these kinds of facts help a lot when you work with a generic library.

The primary consequence explains the shape of the class headers and the second and third use of this new syntax. Because these classes are to implement the interface—via **implements** *IList<I>*—they must ensure that *I*, their type parameter, is a subtype of *IComp*. The constraints in the type

parameter specification of these classes accomplish just that. Furthermore, the *Cons* class uses its type parameter as the type of the *first* field. Thus, each member of the list also implements *IComp*.

   This brings us to the design of *IComp* itself. The purpose of this interface is to specify that an implementing class must provide a method that can compare **this** instance with some *other* instance. From the perspective of the type system, we wish to say that the types of the objects that we compare ought to be the same. While it is impossible to say this in a world of pure subtyping—see section 31.2 and figure 164—generics add just this form of expressive power.

   For the specific case of *lessThan*, we wish to say that the type of its second argument should be the type of **this**, which is the class in which the definition of *lessThan* is located. The problem is that for our lists with *sort* methods, we wish to express this constraint via an interface; put differently, we don't know the class type of the elements of our lists and, for all we know, these classes haven't been written yet.

```
class MenuItem implements ICompG<MenuItem> {
  String name;
  int value;

  MenuItem(String name, int value) {
    this.name = name;
    this.value = value;
  }

  public boolean lessThan(MenuItem o) {
    return this.value < o.value;
  }
}
```

Figure 168: Generically comparable objects

You can see the solution in the top-right of figure 167. The interface *ICompG* is parameterized over the type *T* that the *lessThan* method uses for its second argument. Of course, since we wish to compare instances of the same class, *T* is usualy also the class that implements *ICompG*. Conversely, if the implementing class instantiates *ICompG* with itself, as in

   class *Sample* **implements** *ICompG<Sample>* { ... }

the type checker ensures that *lessThan* is invoked on an instance of *T* and is

passed a second instance of *T*. In short, apples are compared with apples only.[80]

We exploit this idea for the design of *MenuItem* in the generic world; see figure 168. The class header specifies that all instances of *MenuItem* support a *lessThan* method. Since it applies *ICompG* to *MenuItem* itself, we conclude that the *lessThan* method consumes **this** *MenuItem* and another *MenuItem*. That is, the designer of the method knows from the class and method signature that both objects have a *value* field. Hence, comparing the value fields ranks them properly. Better yet, it is unnecessary to cast *other* to *MenuItem*, and it is impossible that such a cast fails during evaluation.

In the same vein, the constraints on *IList* and its consequence for *insert* imply that *insert* can compare the objects on the list with each other—also without using a cast. Once the designer has this knowledge, it becomes easy to deal with method definitions and it is easy to argue that the method can never fail during evaluation.

It is this stark contrast with the solution in the Object and subtyping world (see figures 164 and 165) that makes people prefer the generic solution. Every time something can go wrong, it will go wrong. If a program can fail because it tries to compare apples and oranges, eventually it will, and it will do so while your most important customer is using it. In this case, the (sound) type system can prove that such a problem can *never* happen,[81] and that is a good guarantee to have.

Exercises

Exercise 31.13 Demonstrate that this parameteric list library can be used to represent menus and phone books, including their sorting functionality. ∎

Exercise 31.14 Add the *count*, *contains*, and *asString* methods to the generic list library. What do you expect when you use *asString* on a list of *MenuItem*s? What do you actually get? Can you improve this? ∎

Exercise 31.15 Exploit the type signatures and constraints on the types to argue that the result of *sort* and *insert* are lists of comparable objects. ∎

---

[80]There is still the problem of *null*, which may show up at any type; we ignore this for now.

[81]If something does go wrong, it is the fault of the type checking system in your language and some company will then fix this mistake for all users of the language.

Unfortunately, the generics-based solution suffers from two flaws. First, like the first attempt at creating a list library with sorting via subtyping (section 31.2), the design of the generic library duplicates code. Specifically, the *count*, *contains*, and *asString* methods must be duplicated in the *IList<I>* and *IList<I* **extends** *ICompG<I>>*. Ideally, we would really like to have all these methods in a single library—at least as long we are in charge of both—with the proviso that the *sort* and *insert* methods belong to *IList<I>* only if *I* is a subtype of *ICompG<I>*. That, however, is impossible. Worse, it is impossible to fix the generic library without re-introducing casts and thus lurking errors.

Second, the solution is highly inflexible. A programmer who represents a list of information via some general library may need methods that sorts the list in several different ways. For example, a program may need to sort personnel records according to alphabet, age, or salary. A moment's thought shows that the design in figure 167 does not permit this. The *sort* method always invokes the one and only *lessThan* method that the items on the list implement according to *ICompG<I>*. The next chapter will demonstrate how a systematic abstraction of traversals eliminates this specific problem. In the meantime, we are going to take a step back and extract design principles from our examples.

32 Designing General Classes and Frameworks

When (large parts of) two classes systematically differ in the types of fields and method signatures and differ only in those types, it is time to design a general[82] class, hopefully useful as a library for many projects. This is equally true for two arrangements of several classes that are structurally similar to each other up to differences concerning types. When several classes are involved, people call such arrangements FRAMEWORKS, especially if they are useful in specific situations. Of course, such strict similarities rarely come about naturally. Usually they emerge during the editing phase of the programming process and require a few additional reorganizations and abstraction steps to fully crystalize.

The construction of a general class proceeds roughly according to the design recipe for abstraction:

recognizing similarities: Let's describe the initial scenario and name the elements. For simplicity, we focus on parallels between two classes,

---

[82]Using "abstract" in place of "general" would conflict with established terminology.

but the design recipe carries over to frameworks two.

You have discovered two classes, *CollectA* and *CollectB*, that are the same except for two types, say *A* and *B*, respectively. Also assume that the fields of *CollectA* and *CollectB* are invisible from the outside. **private**.

The *goal* of the generalization process is to specific a type *T* that generalizes *A* and *B* and then use *T* to create a class *Collect* from which you can derive *CollectA* and *CollectB*. In order to do so, you may have to modify *A* and *B* (details below) and we assume that you may do so.[83]

The following sketchy class and interface diagram provides a graphical overview of the scenario:

```
              +------------+
              | Collect    |
              +------------+
              | T          |
              +------------+                   +---+
              | ... T ...  |                   | T |
              +------------+                   +---+
                    |                           / \
                   / \                         /   \
                   ---                        /     \
                    |                     +---+     +---+
         +--------------------+           | A |     | B |
         |                    |           +---+     +---+
         |                    |
  +---------------+    +---------------+
  | CollectA      |    | CollectB      |
  +---------------+    +---------------+
  | private A ... |    | private B ... |
  +---------------+    +---------------+
  | ... A ...     |    | ... B ...     |
  +---------------+    +---------------+
```
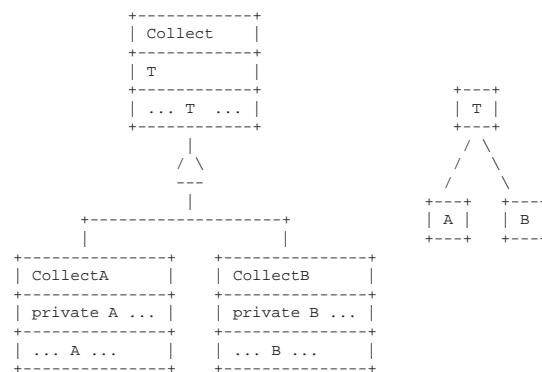
Ideally, you should have a sense that *Collect* is going to be useful in other circumstances, too. This is typically the case when *CollectA* and *CollectB* are about compounding a (fixed or arbitrary) number of objects of type *A* and *B*, respectively. Remember that object-oriented languages tend to use the terminology of a collection class for such things, and it is for that reason that we have used *Collect* in the names of the classes.

Example: The class for pairing two arbitrary objects with methods for retrieving them and switching their order is a simplistic example. The list library for arbitrary objects is a generally useful framework. This second example arose from a comparison of two concrete examples

---

[83]This assumption is realistic and exercise 32.6 shows how to get around the assumption when it doesn't hold.

that differed in the component types only: *MenuItem* played the role of *A* and *PhnBEntry* played the role of *B*.

abstracting over types: Once you understand the similarity, it is time to eliminate the differences in types with generalized types. To do so, you may use either subtyping or generics. Because this design recipe applies to both cases, the specifics of the two approaches can be found in subsections 32.2 and 32.3. For both approaches, though, it is critical to inspect *CollectA* for constraints on *A* and *CollectB* for constraints on *B*.

The methods and initialization expressions of *CollectA* (*CollectB*) may impose constraints on type *A* (*B*) in one of two ways: they may reference a field from *A* (*B*) or they may invoke a method on some object of type *A* (*B*). We deal with each case in turn.

Say *CollectA* and *CollectB* refer to some field *f* in an object of type *A* and *B*, respectively. Consider (1) adding a method *getF* to *A* and *B* that retrieves *f*'s content and (2) replacing references to *f* in *CollectA* and *CollectB* with invocations of *getF*. If doing so is impossible (because *A* or *B* are beyond your control), consider abandoning the generalization process. We ignore fields in *A* and *B* from now on.[84]

Say *CollectA* and *CollectB* invoke some method *m* in an object of type *A* and *B*, respectively. In this case, we need to inspect and compare the method signatures of *m* in *A* and *B*. This comparison produces constraints that dictate how you generalize:

1. In the first and simplest case, *m* consumes and produces types of values that are unrelated to the types under consideration:

    inside of *A* :   inside of *B* :
    int *m*(*String s*, boolean *b*)   int *m*(*String s*, boolean *b*)

    Here we can add it to an interface and use it as needed in *Collect*. Furthermore, this kind of constraint does not necessitate changes to *A* or *B* for the next design step.

2. The second example involves the type under consideration:

    inside of *A* :   inside of *B* :
    A m(*String s*, boolean *b*)   *B m*(*String s*, boolean *b*)

---

[84]If you wish to generalize in this situation, you must use a common superclass of *A* and *B* that contains *f*. This constraint is extremely stringent and hardly ever yields generally useful libraries.

Specifically, *m* produces a result of the class type itself.

For the type representing *A* and *B* in *Collect*, this implies that it must support a method whose return type is the class type. We can translate this constraint into one of the following interfaces, the left one for use with subtyping and the right one for use with generics:

interface *ISub* {   **interface** *IGen<I>* {
  ISub m(*String s*, boolean *b*);   *I m*(*String s*, boolean *b*);
}   }

Any class that implements *ISub* must define a method *m* that returns either an instance of the class or at least an instance of a class that implements *ISub*. With the generic interface *IGen* your class specification can be more precise about what it returns. If class *C* implements *IGen<C>*, its method *m* is guaranteed to produce an instance of *C*, just like *m* in *A* and *B*.

3. The third kind of example involves the type under consideration as a parameter type:

inside of *A* :   inside of *B* :
… *m*(*String s*, *A x*)   … *m*(*String s*, *B x*)

Regardless of the return type, both methods *m* consume an argument whose type is the class itself. Again, there are two ways to express the constraints via interfaces:

interface *ISub* {   **interface** *IGen<I>* {
  … *m*(*String s*, Object *x*)   … *m*(*String s*, *I x*)
}   }

In the subtyping world, we are forced to use a supertype of *A* and *B* and to cast from there to the appropriate subclass in the actual method. This suggests using Object because it is the most flexible type. In the world of generics, we can use a type parameter to express the fact that *m* must be applied to another instance of the *same* class.

Example: As seen, the representation of sortable phone books assumes the existence of a method *lessThan* in *PhnBEntry* with signature boolean *lessThan*(*PhnBEntry e*).

4. Finally for all other cases, consider abandoning the generalization effort because it imposes a level of complexity that it is unlikely to pay off.

If the comparison yields several constraints, consider combining all of them in a singe interface.

testing the abstraction: Once you have figured out how to generalize *A* and *B* and how to create the general class *Collect*, you must validate that it is correct. Following *How to Design Programs*, the very definition of generalization suggests that you must be able to define the original classes in terms of the general one. For our scenario, you must show that you can derive the old classes from the generalization. In particular, you must show that you can recover all methods directly via inheritance or indirectly via calls to **super** methods plus casts. The details depend on how you generalize *A* and *B*.

In our abstract example, you should be able to define *NewCollectA* and *NewCollectB* as extensions of *Collect*. If you formulated constraints on the generalization of *A* and *B* and if *Collect* uses these constraints, you must also ensure that *A* and *B* satisfies them. Concretely, recall the interface from case 2 above. Since *Collect* uses either *ISub* or *IGen<I>* as types in lieu of *A* and *B*, you must add an **implements** clause to *A* and *B* and adjust the method signatures accordingly.

Finally, you must use the tests for *CollectA* and *CollectB* to ensure the basic correctness of *Collect* and its methods. Eventually though, you may also wish to generalize the tests themselves; otherwise you always depend on the specific class when working on the general one.

re-formulating purpose statements: When you have convinced yourself that everything works, revisit the purpose statements for *Collect* and its method statements. Since you want to re-use this class (and the effort that went into its construction), formulate the purpose statements as generally as possible.

Keep in mind to proceed in an analogous manner for frameworks.

You may consider applying this design recipe while you are designing a compound data representation for a single case. With some experience, it often becomes clear which types deserve generalizing and which ones don't. Just be sure to finish the design of a concrete version before you generalize; for beginners, this definitely remains the safest route.

32.1 Subtyping Summarized

Both approaches to type generalization rely on subtyping to some extent, though in different ways. We therefore start with a brief overview of sub-

typing in this subsection, including a few ideas that you haven't encountered yet. While these new ideas aren't critical, they make the generalization process convenient.

Recall that in Java a type is either the name of a class or of an interface (or a primitive type such as int). To declare that some type *T* is a subtype of some type *S*, a programmer uses one of three declarations:

1. **class** *T* **extends** *S*, meaning class *T* extends some other class *S*;

2. **class** *T* **implements** *S*, meaning *T* implements an interface *S*;

3. **interface** *T* **extends** *S*, meaning *T* extends some other interface *S*.

The Java type checker also considers *T* a subtype of *S* if *T* is connected to *S* via a series of immediate subtype declarations.

If you just look at the classes in a program, you see a tree-shaped hierarchy, with Object as the top class. A class definition that doesn't specify an **extends** clause is an immediate subtype of Object. All others are immediate subtypes of their specified superclass.

New: While a class can extend only one other class, it can implement as many interfaces as needed. Hence, a class can have many supertypes, though all but one are interfaces. Similarly, an interface can extend as many other interfaces as needed. Thus, an interface can have many supertypes and all of them are interfaces.
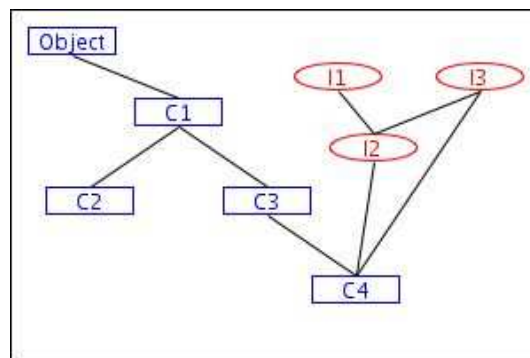
Figure 169: Subtyping in general

Although the details of multiple super-interfaces are irrelevant here, the fact that they exist means that interfaces form a more complex hierarchy

than classes. Computer scientists and mathematicians call this kind of hierarchy a DIRECTED ACYCLIC GRAPH (DAG). The important point for us is that the DAG sits on top and to the side of the tree-shaped class hierarchy. Figure 169 summarizes the subtyping hierarchy schematically.

Subtyping in Java implies slightly different consequences for classes and interfaces:

1. A class C that **extends** class D inherits all visible methods, fields, and **implements** declarations.

   New: A subclass may override an inherited method with a method definition that has a signature that is like the original one except that the return type may be a subtype of the original return type.

   Example: Here *mmm* in C overrides *mmm* in D:

   class D {   **class** C **extends** D {
     … D *mmm*(*String s*) …   … C *mmm*(*String s*) …
   }   }

2. A class C that **implements** interface I must provide **public** method definitions for all method signatures of I. If C is **abstract**, the method definitions may be **abstract**, too.

   New: A class may implement a method signature from an interface with a method declaration that has a signature that is like the original one except that the return type may be a subtype of the original one.

   Example: Here *mmm* in C implements *mmm* in I:

   interface I {   **class** C **implements** I {
     I *mmm*(*String s*) …   … **public** C *mmm*(*String s*) …
   }   }

3. An interface I that **extends** interface K inherits all features from K.

Each of these consequences plays a role in the following subsections.

32.2 Generalizing via Subtyping

Let's recall the basic scenario of this section. We are starting from *CollectA* and *CollectB*, and the former uses type A wherever the latter uses type B. The goal is to design a class *Collect* from which we can re-create *CollectA* and *CollectB*. Doing so involves one decision and three actions:

choosing a replacement type for *A* and *B*: The very first step is to choose a type with which to replace *A* and *B* in *CollectA* and *CollectB*, respectively. If you wish to stick to subtyping, you have three alternatives though only two matter:

1. If both *A* and *B* are classes, consider using Object as the replacement type. This choice allows the most flexibility and works fine as long as there are no constraints.
   Example: For *Pair*, a representation of pairs of objects, it is natural to choose Object as the type for the two fields. There aren't any constraints on the components of a pairing.

2. If both *A* and *B* are classes, you might also consider an existing superclass of *A* and *B*. This tends to tie the general *Collect* class to a specific project, however, making it unsuitable for a general library. We therefore ignore this possibility.

3. If either *A* or *B* is an interface and some of their method signatures are constraints, you have no choice; you must use an interface to generalize an interface. Similarly, if *CollectA* and *CollectB* make assumptions about methods in classes *A* and *B*, you are better of choosing an interface than Object.
   Example: For sortable lists, we tried both alternatives: Object and an interface. The latter was necessary for the creation of sortable lists because sorting assumes that the objects on the list implement a *lessThan* method.

It is possible to choose solutions that mix elements of these three alternatives, but we leave this kind of design to experienced developers.

designing *Collect*: Once you have decided which replacement type *T* you want to use for *A* and *B*, replace the designated parts of *CollectA* and *CollectB* with *T*. If you also rename the two classes to *Collect*, you should have two identical classes now.

Unfortunately, these classes may not function yet. Remember that the methods and initialization expressions of *CollectA* and *CollectB* may impose conditions on *T*. If you collected all those constraints in a single interface and if *T* is this interface, you are done.

If, however, you chose Object to generalize *A* and *B*, you must inspect all method definitions. For each method definition that imposes a constraint, you must use casts to an appropriate interface to ensure that the objects stored in *Collect* implement the proper methods.

Examples: When you use the *IComp* interface to generalize sortable lists, you don't need any casts in the methods. The interface guarantees that the items on the list implement a *lessThan* method.

When you use Object as the generalization of *A* and *B*, you must use a cast in the *sort* method to ensure that the items are implement *IComp*.

preparing *A* and *B* for re-use: If you identified any constraints during the first step of the design recipe (see page 484), you have an interface with method signatures that represent the constraints. Let' s call this interface *IConstraints*. If you choose Object as the generalized type, *IConstraints* may show up in individual methods of *Collect*; otherwise *T* is *IConstraints*.

In either case, you must ensure that *A* and *B* implement the obligations in *IConstraints*. Start with adding an **implements** *IConstraints* clause to the definition of class *A*; if *A* (*B*) is an interface, use **extends** *IConstraints* instead. Of course classes don't automatically implement the obligations specified in some interface. Here we do know that the methods exist by name; this is how we created the interface *IConstraints* in the first step of the design recipe. Their signatures may not match the interface signature, however. To fix this problem, look for signatures in *IConstraints* whose parameter types involve Object and the method definition uses *A* instead:

class *A* {   **interface** *IConstraints* {
… *m*(*String s*, *A x*) { … }   … *m*(*String s*, Object *x*)
}   }

This mismatch originates from case 3 in the constraint gathering step of the design recipe. Adjust the parameter type in the class and use a local variable to cast the parameter to *A*:

    class *A* **implements** *IConstraints* {
    … *m*(*String s*, Object *y*) { *A x* = (*A*)*y*; … }
    }

The rest of the method body remains the same. (Why?)

For method signatures in *IConstraints* that use *IConstraints* as a return type, you don't need to change *A*'s definition. (Why?)

Also adjust *B* in this manner. Then do re-run the tests for *A* and *B* before you proceed.

Example: The representation of menus in terms of the *ISort* library illustrates the case where *Collect* uses the constraint interface *IComp* as the generalized type.

No matter how you use *IComp* for the generalization, the *MenuItem* class must implement *IComp*, becaue the interface constrains which items the list may contain. Since the parameter type of *lessThan* is Object in *IComp*, we use a cast inside the method to ensure that the method compares *MenuItem*s with *MenuItem*s.

re-creating *CollectA* and *CollectB*: The re-creation of *CollectA* and *CollectB* from *Collect* again depends on your choice of replacement type *T*. The goal of this step is to define *NewCollectA* and *NewCollectB* as sub-classes of *Collect*. These new classes should support the exact same methods as *CollectA* and *CollectB* and should pass the same tests.

If *T* is Object, inspect the signatures of the inherited methods and compare them to the original:

1. There is nothing to do when the original signature doesn't involve *A* or *B*.

2. The original return type is *A* or *B*, but Object in *Collect*:

inside of *CollectA* :	inside of *Collect* :
A m(*String s*)	Object *m*(*String s*)

   Fortunately, Java's inheritance works in our favor here. You can simply override *m* with a method that has the old signature and call the **super** method to get the work done:

   inside of *NewCollectA* :
   A m(*String s*) { **return** (*A*) **super**.*m*(*s*); }

   Because **super**.*m*'s return type is Object, the overriding class uses a cast to *A*.

3. Finally, consider the case when *A* or *B* is the type of a parameter in *CollectA* or *CollectB*, respectively:

inside of *CollectA* :	inside of *Collect* :
void *m*(*A x*)	void *m*(Object *x*)

   In this case you need to the following method to *NewCollectA*:

   inside of *NewCollectA* :
   void *m*(*A s*) { ... }

```
// a cell for tracking Items
class CellItem {
  private  Item  content;
  private int counter = 0;
  public CellItem( Item  content) {
    this.content = content;
  }

  // get current content of this cell
  public  Item  get() {
    return content;
  }

  // set content of this cell to c
  public void set( Item  content) {
    counter = counter + 1;
    this.content = content;
  }

  // how often did this cell get set?
  public int howOften() {
    return counter;
  }
}
```

```
// a cell for tracking objects
class Cell {
  private  Object  content;
  private int counter = 0;
  public Cell( Object  content) {
    this.content = content;
  }

  // get current content of this cell
  public  Object  get() {
    return content;
  }

  // set content of this cell to c
  public void set( Object  content) {
    counter = counter + 1;
    this.content = content;
  }

  // how often did this cell get set?
  public int howOften() {
    return counter;
  }
}
```

```
class NewCellItem extends Cell {
  public NewCellItem(Item content) {
    super(content);
  }
  public Item get() {
    return  (Item) super.get();
  }
  public void set(Item content) {
    super.set(content);
  }
  public void set(Object content) {
    throw new RuntimeException
           ("Item expected");
  }
}
```

Figure 170: Generalizing a simple collection class

Otherwise, users of *NewCollectA* may accidentally invoke *m* on arbitrary objects, not just instances of *A*. Since overriding dictates that the types of the explicit parameters remain the same, this is actually defining an *overloaded* method. The implementation uses **super**.*m* to provide the functionality:

inside of *NewCollectA* :
void *m*(*A s*) { **return super**.*m*(*s*); }

Because this new definition overloads *m* and doesn't override it, *NewCollectA* supports one extra method:

inside of *NewCollectA* :
void *m*(Object *s*)

a method that it inherits from *Collect*. To ensure that nobody uses this method accidentally, it is best to override it, too, and make it raise an exception:

inside of *NewCollectA* :
void *m*(Object *s*) {
  throw new *RuntimeException*("A expected");
}

Section 32.5 explains exceptions in some detail.

If *T* is an interface, you proceed in the exact same manner as far as *NewCollectA* is concerned.

Example: Thus far, we have not encountered the need to override any methods or to supplement methods with overloaded versions in the classes that correspond to *NewCollectA*.

Given the lack of an example, let's work through a small, toy-size problem to illustrate this point and some others from the design recipe:

... Design a *CellItem* class that manages access to an instance of the *Item* class. The purpose of *CellItem* is to allow other classes to get the current *Item* in a *CellItem*, to put in a new one, and to find out how often the content of a *CellItem* object has been changed. ...

The left side of figure 170 shows the complete, specialized solution to this problem (minus the *Item* class). The naturally stateful class has two private fields, a public constructor, and three public methods.

Of course, such a "management" class may be useful for controlling access to objects other than *Item*s. Thus, we set out to generalize it: we inspect the class; we find no constraints on what is stored; we choose Object as the type that generalizes *Item*; and we end up with the *Cell* class on the right of figure 170. As you can see from this class definition, Object occurs in the four positions that are analogous to the occurrences of *Item* in *CellItem*.

Because there were no constraints on what *Cell* may store, all we have to do now is demonstrate that *CellItem* is a special case of the general *Cell* class. Put differently, we should show that if *Cell* were in a library, a version of *CellItem* could easily be obtained. The idea of obtaining this customized class from library code is indicated by the bold horizontal line that separates *Cell* from *NewCellItem*. For this "re-creation" step, we define two methods in terms of the super methods: *get* and *set*. The former *overrides* the **super**.*get* method and uses a cast to return an *Item* instead of an Object. The latter *overloads* the *set* method and uses **super**.*set* to implement the functionality.

Exercises

Exercise 32.1 Create a test suite for *CellItem*. Then ensure that *NewCellItem* passes this test suite, too. ∎

Exercise 32.2 Use *Cell* to derive a class that manages access to *String*s. ∎

Exercise 32.3 Modify *Cell* so that it also counts all invocations of *get*, not just *set*. This exercises shows again that abstraction creates single points of control where you can enhance, fix or improve some functionality with one modification. ∎

The case changes significantly if we add the following to *CellItem*:

inside of *CellItem* :
  private int *weight* = 0;

  // set the content of **this** cell to *c*
  public void *set*(*Item content*) {
    counter = *counter* + 1;
    weight = *weight* + content.weight() ;
    this.*content* = *content*;
  }

```
// retrieve the weight from this cell
public int getWeight() {
  return weight;
}
```

The intention of this addition is to accumulate the total "weight" of all the *Item*s that are stored in the *content* field. While the *weight* field is initially 0, the *set* method increments this field by the weight of the new *Item* that it is to store in *content*. Finally, *getWeight* is a method that retrieves the current value of *weight*.

Obviously, the *set* method assumes that the *Item* class comes with a *weight* method; see the gray-shaded method call. If we are to generalize this class, we must formulate this assumption as a constraint for all future classes that are to reuse the generalized class. We have formulated the assumption in an *ICell* interface, displayed at the top of figure 171. The interface specifies a single method signature, which is directly derived from the method call in *set*.

The next step is to choose whether we wish to use Object as the general type for a generalization of *CellItem* or whether we should use *ICell*. Figure 171 shows both alternatives, side by side for a comparison: *CellObject* and *CellIntf*. The two classes are only sketched out, displaying the one method where they differ.

In *CellObject*, the *set* method must use a cast to *ICell* because its parameter's type is Object but the body actually assumes that the object comes with a *weight* method. Using *ICell* directly as the type of objects to be stored in the *CellIntf* class eliminates this cast and all potential problems. Re-creating *CellItem* from either of these general classes produces identical classes. Indeed, they are the same as before. Before we can use them, however, we must modify the *Item* class so that it **implements** *ICell*.

At this point, you might wonder why anyone would ever design a method like *CellObject* if any of the methods impose constraints on the generalized type. Why not always use the interface? The point is that the choice of a general type demands a (design) trade-off analysis. Recall the completely analogous situation with *IList* and *ISort*. The former is a data representation of lists of arbitrary Object; the latter is one for lists of comparable objects only. If we implemented both libraries, both would support methods such as *count*, *contains*, and *asString*. We would duplicate code. And we can avoid this code duplication with the technique that we have just seen again with the design of *CellObject*.

```
interface ICell {
  int weight();
}
```

```
class Item implements ICell {
  public int weight() { … }
}
```

```
class CellObject {
  Object content;
  private int weight = 0;
…
  public void set(Object c) {
    ICell content = (ICell)c;
    counter = counter + 1;
    weight = weight + content.weight();
    this.content = content;
  }
  …
}
```

```
class CellIntf {
  private ICell content;
  private int weight = 0;
…
  public void set( ICell  content) {
    counter = counter + 1;
    weight = weight + content.weight();
    this.content = content;
  }

…
}
```

```
// redefining CellItem in terms of Cell
class CellItemObj extends CellObject {
  public CellItemObj(Item content) {
    super(content);
  }

  public Item get() {
      return (Item)super.get();
  }

  public void set(Item content) {
    super.set(content);
  }
  public void set(Object content) {
    throw new RuntimeException
          ("Item expected");
  }
}
```

```
// redefining CellItem in terms of Cell
class CellItemIntf extends CellIntf {
  public CellItemIntf(Item content) {
    super(content);
  }

  public Item get() {
    return (Item)super.get();
  }

  public void set(Item content) {
    super.set(content);
  }
  public void set(ICell content) {
    throw new RuntimeException
          ("Item expected");
  }
}
```

Figure 171: Generalizing a simple collection class, with constraints

In this specific—toy-sized—example, the only advantage of *CellObject* is that it can also store objects for which *set* is never used. Since the *set* method is the reason why *CellItem* exists in the first place, there is no argument left in support of *CellObject*. It is the wrong choice. In general though, the

choice isn't as obvious as this and you may wish to weigh the advantages and disadvantages of the alternatives before you commit to one.

When a solution like *CellObject* is the preferred one, you are accepting additional casts in your class. Each method that makes special assumption about the objects must use casts. Casts, however, come with three distinct disadvantages. First, the presence of casts complicates the program. The conscientious programmer who adds these casts must construct an argument why these casts are proper. That is, there is a reason why it is acceptable to consider an object as something of type *A* even though its declared type is Object (or something else). Unfortunately, Java doesn't allow the programmer to write down this reason. If it did, the type checker could possibly validate the reasoning, and better yet, any programmer who edits the program later would be able to re-use and re-check this reasoning.

Second, casts turn into checks during program evaluation. They perform the equivalent of an *instanceof* check, ensuring that the object belongs to the specified class or implements the specified interface. If it doesn't, the type checker was cheated and many other assumptions may fail; hence, the evaluator raises an exception in such cases. Although such tests aren't expensive, the problem is that they can fail.

Third, when casts fail, they signal an error and stop the program evaluation. As long as it is you—the programmer—who has to study and understand the error message, this isn't much of a problem. If it is your company's most important client, you are in trouble. After all, this customer typically has no computer training and just wants a functioning program for managing menus across a chain of a hundred restaurants.

In short, a failure after delivery is costly and is to be avoided if possible. Conversely, eliminating such potential cast failures should be our goal whenever it is cost-effective to do so. And the desire to eliminate casts is precisely why Java and other typed object-oriented languages are incorporating generics.

Exercises

Exercise 32.4 Complete the definition of *CellIntf* in figure 171, including a test suite. Then consider this class:

```
class Package {
  int postage;
  int value;
```

```
  Package(int postage, int value) {
    this.postage = postage;
    this.value = value;
  }

  int weight() { return value; }
}
```

Derive a class *CellPackage* for controlling access to instances of *Package*. ∎

Exercise 32.5 Exercise 32.2 asks you to derive a class from *Cell* for managing access to *String*s. Is it possible to derive such a class from *CellIntf* using the *length* method to determine the weight of a *String*? If yes, do so. If not, explain why it can't be done. ∎

Exercise 32.6 On many occasions you will encounter a situation where you would like to use a general library *L* (class or framework) for a class *C* that you can't change. The use of *L* with any class, however, presupposes that this class implements interface *I*. Since *C* doesn't, you are apparently stuck.

   The solution is to create an ADAPTER class that bridges the gap between *C* and *I*:

```
  class Adapter implements I {
    private C anInstance;
    Adapter(C anInstance) {
      this.anInstance = anInstance;
    }
    . . .
  }
```

The methods of *Adapter* naturally use the methods of *anInstance* to compute their results. Revisit exercise 32.5 and solve it with an adapter. ∎

32.3   Generalizing via Generics

The introduction of generics into Java acknowledge the importance of abstracting over types.[85] A generic class (interface) parameterizes a class (an interface) in the same sense as a function definition parameterizes an expression. Similarly, applying a generic class (interface) to arguments is just

---

[85]The designers of Java added generics only to the fifth revision of the language, even though parametric polymorphism has been studied since the 1970s and has been available in programming languages for almost the same time.

like applying a function to values. The difference is that the generics computation takes place at the level of types and the function application takes place in the realm of values. Put differently, it is the *type checker* that determines the outcome of applying a generic type to a type while it is the *evaluator* that determines the value of a function application.

When you use generics to generalize some classes and interfaces into a framework, you do not think in terms of generalizing types but in terms of types as parameters. In a sense, generalizing with generics is much like generalizing with functions in *How to Design Programs*:

specifying the parameter for *Collect*: If you choose to use generics to generalize *CollectA* and *CollectB*, you pick a type parameter per pair of corresponding differences. Our running abstract example assumes that there is one pair of such differences: *CollectA* uses *A* where *CollectB* uses *B*. Hence, we need one type parameter, say *I*. For realistic examples, you should consider choosing suggestive parameter names so that people who read your program get an idea of what is stored in *Collect*.

Assuming you have gathered all the constraints on *A* and *B* as method signatures in the interface *IConstraints*. Recall that constraints are occasionally self-referential, in which case *IConstraints* itself is parameterized. This suggests using something like the following three forms as the header of *Collect*:

1. **class** *Collect <I>* if there are no constraints;

2. **class** *Collect <I* **extends** *IConstraints>* if the constraints are not self-referential;

3. **class** *Collect <I* **extends** *IConstraints<I>>* if the constraints refer to themselves. Keep in mind that the first occurrence of *I* is the binding instance, and the second one is an argument to which the generic interface *IConstraints* is applied.

Things may get more complicated when several type parameters are involved, and you may benefit from experimentation in such cases.

Example: The generalization of *MenuItem* and *PhnBEntry* needs one type parameter because the (core of the) two classes differ only in the type of one field. In contrast, the definition of *Pair* uses two parameters. Finally, *IList<I* **extends** *ICompG<I>>* is our canonical illustration of a *Collect*-like class with a constrained interface.

designing *Collect*: Now that you have the header of *Collect*, you just re-
place *A* and *B* with *I*, respectively. Doing so for *CollectA* or *CollectB*,
you should end up with the same class. Since *I* is specified to be a
subtype of the interface of all constraints, this step is done.

preparing *A* and *B* for re-use: Reusing type *A* for the re-definition of *Col-
lectA* requires that *A* implements all the constraints:

1. If there aren't any constraints, you don't need to modify *A*.

2. If the constraints are ordinary, add **extends** *IConstraints* or **im-
   plements** *IConstraints* to *A*'s header; the former is for classes, the
   latter for interfaces.

3. If the constraints are self-referential, *IConstraints* is generic and
   to obtain a type for an **extends** or **implements** clause requires
   an application of *IConstraints* to a type. Naturally, this type is
   A itself, i.e., you add **implements** *IConstraints<A>* or **extends**
   IConstraints<A> to the header.

   With generics, adding these clauses ought to work without fur-
   ther modifications to *A*. The reason is that generic constraint in-
   terfaces capture constraints more precisely than subtyping inter-
   faces.

   Example: Consider the following constraint originating in *A*:

   > **class** *A* {
   >   ...
   >   *m(String s, A x)* { ... }
   >   ...
   > }

   Formulated as an interface, it becomes

   > **interface** *IConstraints<I>* {
   >   ...
   >   *m(String s, I x)* { ... }
   >   ...
   > }

   By applying *IConstraints* to *A* in the **implements** clause, you get
   the exact same signature back from which you started:

```
class A implements IConstraints<A> {
  …
  public m(String s, A x) { … }
  …
}
```

Of course, you must also make *m* **public**.

re-creating *CollectA* and *CollectB*: We know from the preceding section that re-creating *CollectA* and *CollectB* means extending *Collect*. Because *Collect* is generic, this implies that we must apply the class to a type to obtain a suitable class for an **extends** clause. Naturally for *NewCollectA* this type is *A* and for *NewCollectB*, it is *B*:

class *NewCollectA*	**class** *NewCollectB*
extends *Collect<A>* { … }	**extends** *Collect<B>* { … }

Convince yourself that all inherited methods automatically have the correct signatures.

For completeness, figure 172 shows how to create the generalized *Cell* class from the preceding section and its constrained counterpart, *CellWgt*. The column on the left represents the plain *Cell* case; the column on the right concerns the case with constraints. The bold horizontal line separates the library code from the definitions that create *Cell* and *CellW*, the class that keeps track of the weight of items. Take a close look at the figure because it once again illustrates how easy and how elegant generalization with generics are.

As you can see from the abstract recipe and the examples, using generics has two significant advantages. First, it means less work than with subtyping. Second, the resulting code doesn't contain any casts; all type constraints are checked before you ever run your program. From our introductory example in section 31.4—sortable lists—we also know, however, that generics may come with one serious disadvantage. If we wish to avoid casts, we have no choice but to duplicate code between plain, unconstrained lists and lists that contain comparable elements and are therefore sortable. Before we investigate this problem a bit more, though, let's look at some examples.

32.4 Finger Exercises: Sets, Stacks, Queues, and Trees Again

Over the course of this chapter, you have studied generalized list representations. While lists are a ubiquitous mechanism for keeping track of many

```
class Cell<I> {                          class CellWgt<I extends ICell> {
  private I content;                       private I content;
  private int counter = 0;                 private int weight = 0;
                                           private int counter = 0;

  public Cell(I content) {                 public CellWgt(I content) {
    this.content = content;                  this.content = content;
  }                                        }

  public I get() {                         public I get() {
    return content;                          return content;
  }                                        }

  public void set(I content) {             public void set(I content) {
    counter = counter + 1;                   counter = counter + 1;
    this.content = content;                  weight = weight + content.weight();
  }                                          this.content = content;
                                           }

  public int howOften() {                  public int howOften() {
    return counter;                          return counter;
  }                                        }
}                                        }
```

```
                                         interface ICell {
                                           int weight();
                                         }
```

```
class CellItem extends Cell<Item> {      class CellW extends CellWgt<Item> {
  public CellItem(Item content) {          public CellW(Item content) {
    super(content);                          super(content);
  }                                        }
}                                        }
```

```
                                         class Item implements ICell {
                                           public int weight() { … }
                                         }
```

Figure 172: Generic *Cell* classes

objects collectively, programs often need other views of collections of objects. Among those are unordered collections or sets of objects; queues of

objects analogous to lines at bus stops; and stacks of objects analogous to stacks of plates in your kitchen's cupboard.

   This section starts with exercises on designing such alternate collection classes. We expect you to use the list representations from this chapter to collect objects. You should start with the representations in figures 163 and 166, that is, lists based on subtyping and generics, respectively.

Exercise 32.7 Equip your list representations with the methods for counting items; for checking on the presence of a specific item (using *equals*); for rendering the list of items as strings; and for adding an item to the end of the list. Also design a method for removing an item from the list if it *equals* a given item. ▪

Exercise 32.8 In some exercises in the preceding chapters, you have dealt with representations of mathematical sets, e.g., sets of integers. Since sets don't impose any constraints on their elements other than that they support the *equals* method, it is straightforward to generalize these representations via subtyping or generics:

```
interface ISet {                          interface ISet<Element> {
  // is this set empty?                     // is this set empty?
  boolean isEmpty();                        boolean isEmpty();
  // is o a member of this set?             // is e a member of this set?
  boolean in(Object o);                     boolean in(Element e);
  // create a set from this set and x       // create a set from this set and x
  Set add(Object x);                        Set add(Element x);
  // is this a subset of s?                  // is this a subset of s?
  boolean subset(Set s);                    boolean subset(Set<Element> x);
  // number of elements in this set         // number of elements in this set
  int size()                                int size()
}                                         }
```

The interface on the left assumes that the elements of a set are Objects; the one on the right is parameterized over the type of its elements. Design classes that implement these interfaces, using lists to keep track of the set's elements. Extend and modify your chosen list libraries as needed. ▪

Exercise 32.9 If you organize your kitchen, it is likely that you stack your plates and bowls and possibly other things. Designing a data representation for a collection that represents such a form of information is mostly about designing methods that access pieces in an appropriate manner. Here are two interface-based specifications of stacks:

```
interface IStack {                        interface IStack<Item> {
 // is this stack empty?                   // is this stack empty?
 boolean isEmpty();                        boolean isEmpty();
 // put x on the top of this stack         // put x on the top of this stack
 void push(Object x);                      void push(Item x);
 // remove the top from this stack         // remove the top from this stack
 void pop();                               void pop();
 // the first item on this stack           // the first item on this stack?
 Object top();                             Item top();
 // number of items on this stack          // number of items on this stack
 int depth();                              int depth();
}                                         }
```

According to them, a stack is a bunch of objects to which you can add an-other object at the top; from which you can remove only the top-most object; and whose top-most object you may inspect. For completeness, our specifications also include a method for counting the number of items that are stacked up. Design classes that implement these interfaces. Extend your list libraries as needed. ∎

Exercise 32.10 Imagine yourself designing a program for your local public transportation organization that models bus stops. At each bus stop, people can join the queue of waiting people; when a bus shows up, the people who have waited longest and are at the front of the queue enter the bus. You may also wish to know who is at the front and how long the queue is.

The following two interfaces translate this scenario into two interfaces for general queue representations:

```
interface IQueue {                        interface IQueue<Item> {
 // is this queue empty?                   // is this queue empty?
 boolean isEmpty();                        boolean isEmpty();
 // add x to the end of this queue         // add x to the end of this queue
 void enq(Object x);                       void enq(Item x);
 // remove the front from this queue       // remove the front from this queue
 void deq();                               void deq();
 // the first item in this queue           // the first item in this queue
 Object front();                           Item front();
 // number of items in this queue          // number of items in this queue
 int length();                             int length();
}                                         }
```

As always, the specification based on subtyping is on the left, the one based on generics is on the right. Design classes that implement these interfaces. Modify and extend your chosen list libraries as needed. Describe at least two more scenarios where a general queue library may be useful. ∎

Exercise 32.11 Consolidate the two list libraries that you developed in exercises 32.8 through 32.10. Then ensure that your classes for representing sets, stacks, and queues can use one and the same list library. ∎

This last exercise illustrates how generally useful libraries come about. Programmers turn a generalized data representation into a library, use the library in many different contexts, and improve/add methods. Eventually someone standardizes the library for everyone else.[86] The next few exercises are dedicated to tree representations, a different but also widely used collection class and library.

Exercises

Exercise 32.12 Lists are just one way of compounding data. Trees are another one, and they are popular, too. In some cases, organizing data in the shape of a tree is just the best match for the information; examples include family trees and representations of river systems. In other cases, organizing data in this shape is useful for performance purposes.

Design a generalized tree representation for binary trees of objects using both generics and subtyping. A binary tree is either an empty tree or a branch node that combines information with two binary trees, called *left* and *right*. Also design the method *in*, which determines whether some given object is in the tree. ∎

Exercise 32.13 As you may recall from *How to Design Programs* (chapter III), binary search trees are even better than plain binary trees, as long as the objects in the tree are comparable to each other.

Design a generalized tree representation for binary search trees of comparable objects using both generics and subtyping. A binary search tree is either an empty tree or a node that combines two binary search trees. More precisely, a node combines a comparable object *o* with two binary trees, *l* and *r*; *all* objects that are less than *o* occur in *l* and all those that are greater

---

[86]In the case of Java, Joshua Bloch was Sun's designated library designer. He has distilled his experiences in his book "*Effective Java.*"

than (or equal to) *o* occur in *r*. Optional: If the object is in the tree, the tree remains the same. Use privacy specifications to ensure that the condition for branches always holds.

Also design the method *in*, which determines whether some given object occurs in this binary search tree. ∎

Exercise 32.14 Design the class *SetTree*, which implements *ISet* from exercise 32.8 with binary search trees as the underlying collection class. For the latter, start from the data representation designed in exercise 32.13 and modify it as needed. Also design a generic set representation that implements *ISet<Element>* from exercise 32.8.

For both cases, document where you impose the constraint that this kind of set representation works only if the elements are comparable. ∎

Before you move on, take a look at the sections on *Set* and *TreeSet* in the official Java documentation.

32.5   Errors, also known as Runtime Exceptions

Sometimes a method cannot produce a reasonable result from the given data; sometimes the design recipe calls for the insertion of methods that throw an error (see figures 170 and 171). In the preceding chapters, we have used the simple *Util.error* method from ProfessorJ's languages for this purpose. We call it simple because no matter what type the context of the expression *Util.error*(...) expects, the expression fits in.

Java does not allow you to define such general methods as *Util.error*. Instead, if you wish to signal an error, you must use the *exception system*. The general exception system is complex and differs substantially from those used in most programming languages. This book therefore introduces only one form of Java exception, a *RuntimeException*, and one operation on exceptions: **throw**.

The *RuntimeException* class supports two constructors: one doesn't consume anything and the other one consumes a string. To **throw** an instance of *RuntimeException*, you write down an expression such as

... **throw new** *RuntimeException*(`"this is an error"`) ...

in your program, wherever expression are allowed. Like *Util.error*(...) this **throw** expression takes on any desired type.

When Java evaluates a **throw** expression, it performs three steps: it creates the instance of the *RuntimeException*; it stops the program; and it

displays the string, plus some optional information about the state of the evaluation and where the **throw** expression is located in the program. The specifics depend on the Java implementation.

Here is a use of *RuntimeException*s for an example from this chapter:

<u>inside of *MenuItem* :</u>
```
public boolean lessThan(Object o) {
  MenuItem m;
  if (o instanceof MenuItem) {
    m = (MenuItem)o;
    return this.value < m.value; }
  else {
    throw new RuntimeException("incomparable with MenuItem") ; }
}
```

At the time when we first designed *lessThan* for *MenuItem* (see page 473), we went with false as the value to return when an instance of some incomparable class was given. Now that we once again have the option of signaling an error, we should do so.

Several of the exercises in the preceding section call for signaling errors, too. Consider the *pop* and *top* methods in exercise 32.9:

<u>inside of *IStack* :</u>
```
// remove the top from this stack
void pop();
// the first item on this stack?
Object top();
```

Since an empty stack has no elements, it is impossible to retrieve the first and to remove it from the stack. If you solved the exercise, your method definitions signal errors that are about empty lists. Naturally, this is unhelpful to a friend or colleague who wishes to use the stack and has no desire to know how you designed the internals of your *Stack* class.

In *How to Design Programs*, you learned how to design checked functions. Now you need to learn how to add checks to methods. We recommend two additions: first, the purpose statement should warn the future users of *IStack* that *pop*ping or *top*ping an empty stack causes an error; second, the method body should check for this situation and signal an appropriate error:

<u>inside of *IStack* :</u>
  // remove the top from **this** stack
  // ASSUMES: the stack is not empty: *!*(**this**.*isEmpty*())
  void *pop*();

<u>inside of *Stack* **implements** *IStack* :</u>
  public void *pop*() {
    // − − − contract check
    if (*this.isEmpty*())
      throw new *RuntimeException*("pop: stack is empty");
    // − − − end
    . . .
  }

Note the "ASSUMES" in the first fragment, making it clear that in addition to the signature, the method needs to enforce additional constraints on its input. Also note how in the method itself, the checking is visually separated from the method body proper via visual markers. Following the practice from *How to Design Programs*, we call these checks "contract checks."[87]

Exercises

Exercise 32.15 Equip the *sort* method from figure 163 with an explicit check that ensures that the items on the list implement *IComp*. The goal is to signal an error message that re-states the assumption of the *sort* method. ▪

can we checkExpect for
runtime exceptions?

Exercise 32.16 Complete the addition of checks to your stack class from exercise 32.9. ▪

Exercise 32.17 Inspect the method specifications in the *IQueue* interfaces of exercise 32.10 for operations that cannot always succeed. Equip the purpose statements with assumptions and the method definitions with appropriate checks. ▪

Exercise 32.18 Inspect the uses of lists of shots and charges in your "War of the Worlds" program from section 27.7. If possible, create a general list library of moving objects. If not, explain the obstacle. ▪

add exercises on using a
general list structure for
War of the Worlds game,
plus Worm game

---

[87]In programming, the word "contract" tends to describe assumptions about a method's inputs, or promises about its outputs, that cannot be expressed in the type system of the underlying language. Parnas introduced contracts in a research paper in 1972, and they have a long history.

33 Designing (to) Interfaces

The time has come to stop for a moment and think about the nature and the role of interfaces in object-oriented programming. What we have seen is that object-oriented computing is mostly about creating and probing objects, via method calls and field inspections. Of course, not every object can deal with every method call or respond to every field inspection; it can only do so if its *interface* allows so.

From this perspective, the goal of object-oriented programming is therefore to formulate interfaces for objects. For simple kinds of data, the design of a single class *is* the design of an interface for all of its instances. A class describes how to create these objects, what methods they support, and what fields they have. For complex forms of data, you design several classes, but you present them to the rest of the world via a common **interface**. With the latter you can create objects; the former dictates how you compute with them.

This book has introduced the idea of programming interfaces and programming to interfaces from the very first chapter, without spelling it out completely. In the process, we neglected two rough corners of this major programming principle. Furthermore, in this chapter we used **interface**s in a different role. With the following three subsections, we take a close look at these three issues.

33.1 Organizing Programs, Hiding Auxiliary Methods

If it is the purpose of an **interface** to announce to the world which methods it is allowed to invoke on certain objects, you should wonder why we have placed the method signatures and purpose statements of auxiliary methods into **interface**s. After all, such methods are often only useful to objects that implement the interface. Worse, their design may rely on assumptions that are unstated and that hold only when they are invoked by objects that implement the **interface**.

Consider the example of sortable lists, which we have discussed extensively in this chapter. Thus far, we have used **interface**s such as

```
interface IList<Item extends IComp> {
  // sort this list, according to lessThan in IComp
  IList<Item> sort();
  // insert i into this sorted list
  IList<Item> insert(Item i);
}
```

```
              +----------------------------+
              | IList<Item extends IComp>  |<--------------------+
              +----------------------------+                     |
              +----------------------------+                     |
              | IList<Item> sort()         |                     |
              +----------------------------+                     |
                         |                                       |
                        / \                                      |
                        ---                                      |
                         |                                       |
              +----------------------------+                     |
              | AList<Item extends IComp>  |                     |
              +----------------------------+                     |
              +----------------------------+                     |
              | AList<Item> insert(Item i) |                     |
              +----------------------------+                     |
                         |                                       |
                        / \                                      |
                        ---                                      |
                         |                                       |
              ----------------------------------                 |
              |                                |                 |
+------------------------+      +--------------------------+     |
| Mt<Item extends IComp> |      | Cons<Item extends IComp> |     |
+------------------------+      +--------------------------+     |
                                +--------------------------+     |
                                | Object first             |     |
                                | IList rest               |---+
                                +--------------------------+
```
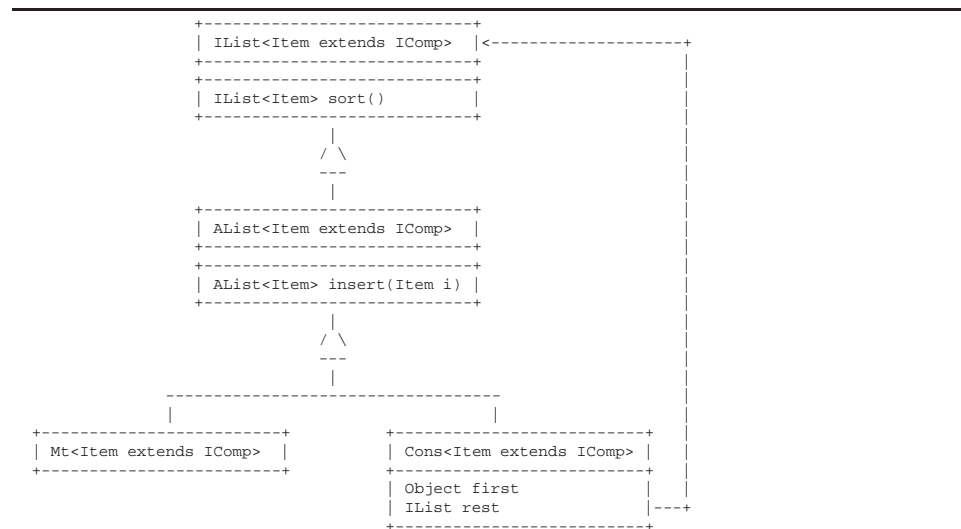
Figure 173: Hiding auxiliary methods, step 1: the class diagram

There is no question that *sort* should be a part of the *IList* interface, but the inclusion of *insert* is dubious. Its purpose statement clearly demonstrates that it is only intended in special situations, that is, for the addition of an item to a sorted list. The *insert* method is at the core of the insertion sort idea, and its presence in the common interface thus reveals how *sort* works. Although making *insert* available to everyone looks innocent enough, it is improper. Ideally, the interface should contain just the header of the *sort* method and nothing else (as far as sorting is concerned).

Overcoming this problem requires a different organization, but one that you are already familiar with. Take a look at the diagram in figure 173. It should remind you of the chapter on abstracting with classes. Like the diagrams there, it includes an abstract class between the *IList* interface and its implementing classes. This time, however, the primary purpose of the abstract class is not to abstract some common methods from the implementing classes—though it may be useful for this purpose, too. Its purpose is to hide the auxiliary *insert* method from the general audience. Specifically, the **interface** *IList* contains only *sort* now. The *insert* method header is in *AList*; furthermore, it is unlabeled, indicating that only library code may refer to it. Since everyone else will use only the interface and the public class constructors to deal with lists, *insert* is inaccessible to outsiders.

The translation into classes and interfaces appears in figure 174. It is

```
interface IList<Item extends IComp> {
  // sort this list according to lessThan in IComp
  IList<Item> sort();
}
```

```
abstract class AList<Item extends IComp> implements IList<Item> {
  // insert i into this sorted list
  abstract AList<Item> insert(Item i);
}
```

```
class Mt<Item extends IComp>
    extends AList<Item> {
  public Mt() {}

  public IList<Item> sort() {
    return this;
  }

  AList<Item> insert(Item i) {
    AList<Item> r = new Mt<Item>();
    return new Cons<Item>(i,r);
  }
}
```

```
class Cons<Item extends IComp>
    extends AList<Item> {
  private Item first;
  private IList<Item> rest;

  public Cons(Item first,
              IList<Item> rest) {
    this.first = first;
    this.rest = rest;
  }

  public IList<Item> sort() {
    AList<Item> r =
      (AList<Item>)rest.sort() ₁;
    return r.insert(first);
  }

  AList<Item> insert(Item i) {
    AList<Item> r;
    if (i.lessThan(first)) {
      return
        new Cons<Item>(i,this); }
    else {
      r = (AList<Item>)rest ₂;
      return
        new Cons<Item>(first,r.insert(i)); }
  }
}
```

Figure 174: Parameteric lists with auxiliary methods, properly organized

straightforward, though requires the two gray-shaded casts in *Cons*. Surprisingly, each cast is due to a different type problem. The one labeled with subscript 1 tells the type system that the value of *rest.sort()* can be accepted as an *AList<Item>*, even though its type is *IList<Item>* according to the signature of *sort*. The cast labeled with 2 is necessary because the *rest* field is of type *IList<Item>* yet this type doesn't support the *insert* method.

```
                    +----------------------------+
                    | IList<Item extends IComp>  |
                    +----------------------------+
                    +----------------------------+
                    | IList<Item> sort()         |
                    +----------------------------+
                                 |
                                / \
                                ---
                                 |
                    +----------------------------+
                    | AList<Item extends IComp>  |<--------------------+
                    +----------------------------+                     |
                    +----------------------------+                     |
                    | AList<Item> sort()         |                     |
                    | AList<Item> insert(Item i) |                     |
                    +----------------------------+                     |
                                 |                                     |
                                / \                                    |
                                ---                                    |
                                 |                                     |
                    ----------------------------------                 |
                    |                                |                 |
      +------------------------+       +--------------------------+    |
      | Mt<Item extends IComp> |       | Cons<Item extends IComp> |    |
      +------------------------+       +--------------------------+    |
                                       | Object first             |    |
                                       | AList rest               |---+
                                       +--------------------------+
```
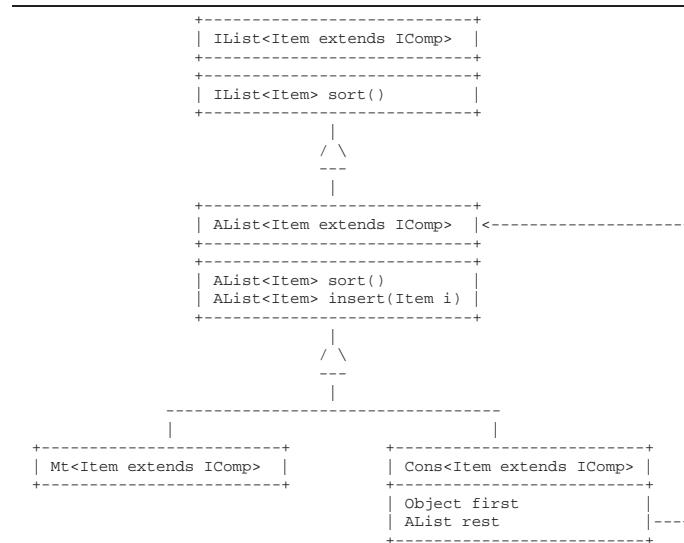
Figure 175: Hiding auxiliary methods, step 2: the class diagram

With a simple refinement of our class and interface arrangement, it is possible to replace these two casts with a single, centrally located cast. Take a look at the diagram in figure 175. The most easily spotted difference between it and the diagram in figure 173 concerns the containment arrow. It no longer points from *Cons* to *IList*, but instead goes from *Cons* to the abstract class. In terms of the class definition, this diagram suggests that the *rest* field has the more specific type *AList* rather than the publicly visible type *IList*. A second, equally important change is less visible: *AList<Item>* overrides the definition of *sort* with an abstract version. Doing so is not only legal with respect to the type system, it is also pragmatic. As we have seen, the recursive calls to *sort* must produce a list of type *AList<Item>* so that it is possible to insert the *first* item by just chaining the method calls.

Figure 176 displays the final definitions of the classes and interfaces of our list library. The one remaining cast is again gray shaded. It is located in the constructor, making sure that every value in the *rest* field is

```
interface IList<Item extends IComp> {
 // sort this list
 IList<Item> sort();
}
```

```
abstract class AList<Item extends IComp> implements IList<Item> {
 abstract public AList<Item> sort();

 // insert i into this sorted list
 abstract AList<Item> insert(Item i);
}
```

```
class Mt<Item extends IComp>           class Cons<Item extends IComp>
   extends AList<Item> {                  extends AList<Item> {
 public Mt() {}                         private Item first;
                                        private AList<Item> rest;
 public AList<Item> sort() {            public Cons(Item first,
   return this;                                       IList<Item> rest) {
 }                                        this.first = first;
                                          this.rest = (AList<Item>)rest ;
 AList<Item> insert(Item i) {           }
   IList<Item> r = new Mt<Item>();
   return new Cons<Item>(i,r);          public AList<Item> sort() {
 }                                        return rest.sort().insert(first);
}                                       }

                                        AList<Item> insert(Item i) {
                                          if (i.lessThan(first))
                                            return
                                              new Cons<Item>(i,this);
                                          else
                                            return
                                              new Cons<Item>(first,rest.insert(i));
                                        }
                                       }
```

Figure 176: Parameteric lists with auxiliary methods, properly organized

of type *AList<Item>*. Put differently, the cast ensures that every class that implements *IList<Item>* also extends *AList<Item>*—a fact that the library designer must now keep in mind for any future modifications (or extensions).

Exercises

Exercise 33.1 Design an example class that tests the sorting algorithm. This class represents a use of the list library by someone who doesn't know how it truly functions. ∎

Exercise 33.2 The definition of *insert* in *Cons* of figure 176 uses the constructor with an *Item* and an *AList<Item>*. By subtyping, this second expression is typed as *IList<Item>*. In the constructor, however, this value is again exposed to a cast. Suppose casts were expensive. How could you avoid the cast in this particular case? Does your solution avoid any other casts? ∎

Exercise 33.3 Design a list library that is based on subtyping (as opposed to generics), supports sorting, and hides all auxiliary operations. ∎

Exercise 33.4 Design a generic list representation that implements the *IList* interface from figure 176 but uses the quicksort algorithm to implement *sort*. You may add `append` to the interface, a method that adds some give list to the end of **this** list. Hide all auxiliary methods via an abstract class.

From the chapter on generative recursion in *How to Design Programs*, recall that quicksort uses generative recursion for non-empty lists *L*:

1. pick an item *P* from *L*, dubbed *pivot*;

2. create a sorted list from all items on *L* that are strictly less than *P*;

3. create a sorted list from all items on *L* that are strictly greater than *P*;

4. append the list from item 2, followed by the list of all items that are equal to *P*, followed by the list from item 3.

Use the example class from exercise 33.1 to test the sorting functionality of your new list library.

After you have designed the library, contemplate the following two questions: (1) Does it help to override *sort* in the abstract class? (2) Does it help to override `append` in the abstract class? ∎

33.2 Getters, Predicates, and Setters

If it is the purpose of an **interface** to announce to the world which common methods some classes provide, you should also wonder how programs that use the library should get hold of the values in fields. Let's reconsider the list library from the proceeding subsection. To the outside, it provides the following view:

1. *IComp*;

2. *IList<Item* **extends** *IComp>*, a generic interface;

3. **new** *Mt<Item* **extends** *IComp>*(), a constructor of no arguments;

4. and **new** *Cons<Item* **extends** *IComp>*(*Item i*, *IList l*), a constructor that combines an *Item* with a *IList*.

As we have seen in the past, programs that use such a library may have an object *los* of type *IList<Item>* constructed from *Cons* and may just need to know what the first item is. Since you have declared these fields to be **private**, the program can't get the current values from these fields, even with a cast that turns the *IList<Item>* into a *Cons<Item>*.

If a library needs to grant access to fields in classes that are presented to the rest of the program via an interface, the best approach is to add methods that can retrieve the current values of such fields. In the case of our list library, you may wish to add the following method specification to *IList*:

inside of *IList<Item ... >* :
  // retrieve the first item from **this** list
  Item getFirst();

Naturally, *getFirst* cannot work for *Mt*, one of the classes that implement *IList*; an empty list doesn't have a first item. Hence, the method should signal an error by throwing a *RuntimeException*:

inside of *Mt<Item ... >* :   inside of *Cons<Item ... >* :
Item getFirst() {   *Item getFirst*() {
  throw new *RuntimeException*("first: ...");   **return this**.*first*;
}   }

Methods such as *getFirst* are dubbed GETTERS in the object-oriented world.

While throwing an exception is definitely the proper way to deal with requests for the first element of an empty list, the library should then also allow the rest of the world to find out whether any given *IList* is empty.

This suggest the addition of a method to *IList* and the addition of an "AS-SUMES" clause to the purpose statement of *getFirst*:

> inside of *IList<Item … >* :
>   // is **this** list empty?
>   boolean *isEmpty*();
>
>
>   // retrieve the first item from **this** list
>   // ASSUMES: **this** list isn't empty: *!*(**this**.*isEmpty*())
>   *Item getFirst*();

The *isEmpty* method returns true for instances of *Mt* and false for instances of *Cons*. It is an example of a PREDICATE.

Finally, on rare occasions you may also wish to grant the rest of the program the privilege to modify the content of a field. To do that, you add a SETTER method such as *setFirst*:

> inside of *IList<Item … >* :
>   // effect: changes what the first item for **this** list is
>   // ASSUMES: **this** list isn't empty: *!*(**this**.*isEmpty*())
>   void *setFirst*(*Item newValue*);

Providing setters also has the advantage that your library can inspect the new value for a field before you commit to a change. That is, if values of a field must satisfy conditions that can't be expressed via the type system, a setter method can check whether the new value is proper for the field and signal an error if it isn't.

Exercises

Exercise 33.5 Design a stack library that provides the following:

1. the *IStack* interface of figure 177 and

2. a constructor for empty stacks that is invoked like this: *MtStack*().

Here is an *Examples* class, representing a use of the library:

```
interface IStack<Item> {
  // push i onto this stack
  IStack<Item> push(Item i);

  // is this stack empty?
  boolean isEmpty();

  // what is the top item on this stack?
  // ASSUMES: this isn't the empty stack: !(this.isEmpty())
  Item top();

  // create a stack by removing the top item from this stack
  // ASSUMES: this isn't the empty stack: !(this.isEmpty())
  IStack<Item> pop();
}
```

Figure 177: Another stack interface

```
class Examples implements Testable {
  public void tests(Tester t) {
    IStack<Integer> mt = new MtStack<Integer>();
    IStack<Integer> s1 = mt.push(1);
    IStack<Integer> s2 = s1.push(2);

    checkExpect(s2.top(),2,"top");
    checkExpect(s2.pop(),s1,"pop");
  }
  ...
}
```

Finally add a setter that changes what the top of a stack is. ∎

Exercise 33.6 Design a queue library that provides the following:

1. the *IQueue* interface of figure 178 and

2. a constructor for empty queues that is invoked like this: *MtQueue*().

As you design the library, don't forget to design a full-fledged examples class.

Design the *front* and *deq* methods so that they use auxiliary methods. (Hint: These methods should use an accumulator parameter.) Naturally,

```
interface IQueue<Item> {
  // add i to the end of this queue
  IQueue<Item> enq(Item i);

  // is this queue empty?
  boolean isEmpty();

  // what is at the front of this queue?
  Item front();

  // remove the front item from this queue
  IQueue<Item> deq();
}
```

Figure 178: Another queue interface

you should hide those auxiliary methods using the technique learned in section 33.1. ∎

33.3   Interfaces as Specifications

While a class is an interface (describing how its instances can interact with the rest of the world), a Java **interface** is a linguistic mechanism that describes the common interface of several classes. In the preceding chapters, we have acted as if **interface**s exist to describe the common methods of classes that are designed simultaneously. In this chapter, we started using **interface**s for the purpose of telling others what we expect from the classes that are to work with our libraries.

The most illustrative example is the list library with support for sorting. Specifically, the *sort* method applies only if the items in the list implement a *lessThan* method specified in the *IComp* interface (or its relatives). Thus, if we wish to represent a menu as a sortable list of *MenuItem*s, then *MenuItem* must implement the *IComp* interface. The same is true for the representation of sortable phone books. In contrast, you cannot create a sortable list of objects using a (unmodifiable) class that pre-dates *IComp*, even if the class contains a method definition for *lessThan* with the proper signature.[88]

Of course, sorting is just one thing we can imagine when we think of phone books or menus. Consider the following requests:

---

[88]While many mainstream object-oriented languages are similar to Java in this regard, some alternative languages are more flexible than Java here.

> ... Represent phone books in an electronic manner so that it is possible to extract all those entries whose *name* field starts with `"A"`. ... Represent restaurant menus electronically so that you can extract all those items that cost between $10 and $20. ...

If we wanted to use lists to represent these forms of information, they would have to support a method for creating a list of objects that have a certain property.

With this chapter's experience in program design, this kind of problem shouldn't pose any challenge. Assuming we have a list representation like the one in figure 166, we design the method *select*, which creates a new list from the given list such that the extracted objects satisfy a certain property. We express this latter part by insisting that the original list is made up of objects that have a *hasProperty* method:

```
interface IList<Item extends IPred> {            interface IPred {
  // extract the objects from this list           // does this object satisfy
  // that satisfy hasProperty from IPred            // a certain property?
  IList<Item> select();                            boolean hasProperty();
}                                                 }
```

Now you add *select* to *Cons* and *Mt*, and you make sure that *MenuItem* and *PhnBEntry* implement *IPred* in an appropriate manner. If this is still a bit fast for you, study the following exercise.

Exercise

Exercise 33.7 Design a representation for restaurant menus. A menu consists of a series of menu items; each menu item names the food and specifies a price. Include a method for extracting those items from a menu that cost between $10 and $20; the result is a menu, too.

Design a representation for phone books. A phone book consists of a series of entries; each entry names a friend and phone number (use plain numbers for this exercise). Include a method for extracting those entries from a phone book that list a friend whose name starts with A; for simplicity, the result is a phone book, too.—When you have completed the design, suggest an alternative form of data for the result of the selection method.

Abstract over the two forms of lists you have designed, including the method for selecting sub-lists. Use both a subtyping approach as well as a generic approach. Don't forget to demonstrate that you can still represent menus and phone books. This should produce a list representation that

implements the above *IList<Item* **extends** *IPred>* interface and whose items are instances of a class that implements *IPred*. ∎

Our true problem is that we now have two list libraries: one that supports sorting and one that supports selecting. If you decide to use the former to represent menus, you need to add your own methods for selecting menu items in the desired price range. If you decide to use the latter for phone books, you need to design your own sorting methods. Of course, what we really want is a list library that supports both methods. Designing such a library demands a choice, and the choices shed light on the role of interfaces as we have used them in this chapter.

The first choice is to design a list library whose elements are both comparable to each other and can be inspected for a certain property. Continuing with our generic approach, this demands the formulation of an interface that expresses these two constraints. There are two ways to do so:

```
interface ICompPred {                              interface ICompPred
  boolean lessThan(Object o);                          extends IComp, IPred { }
  boolean hasProperty();
}
```

The interface on the left includes both a *lessThan* and a *hasProperty* method signature. The interface definition on the right introduces a slightly new notation, namely an **extends** clause with comma-separated interfaces. Its meaning is that *ICompPred* simultaneously extends both interfaces and inherits their method signatures. Using this form of an **extends** clause also makes *ICompPred* a subtype of *IComp* as well as *IPred*; see section 32.1 and figure 169.

Once we have *ICompPred*, we can introduce the interface that defines a generalized list structure:[89]

```
interface IList<Item extends ICompPred> {
  // sort this list according to lessThan in ICompPred
  IList<Item> sort()

  // extract the objects from this list
  // that satisfy hasProperty from ICompPred
  IList<Item> select();
}
```

[89]Java also supports a short-hand for specifying such an interface without introducing *ICompPred*: **interface** *IList<I* **extends** *IComp & IPred>*. This notation exists for convenience, though, and is otherwise of no interest to program design.

All you have to do now is add *Mt* and *Cons* classes that implement this interface and you have a powerful list library. Since this step in the design is routine, we leave it to the exercises.

Exercises

Exercise 33.8 Finish the design of this list library using generics. Use the library to represent the menus and phone books from exercise 33.7. ∎

Exercise 33.9 Design a list library like the one in exercise 33.8 using sub-typing and interfaces. Use the library to represent the menus and phone books from exercise 33.7. ∎

If you solved the preceding exercises or if you thought through the rest of the library design, you understand that this kind of design comes with a serious drawback. Suppose your library exports the interfaces *IList* and *ICompPred* as well as the constructors *Cons* and *Mt*. You can use the constructors directly to represent some list of objects, or you can derive special-purpose interfaces and classes and then use those to make up a list. In either case, you are forced to design classes of list items that implement the *ICompPred* interface. That is, the object on such a list must define a *lessThan* and a *hasProperty* method. It is impossible to create a list of objects that are only comparable or only inspectable for a property.

In order to represent lists that can compound all kinds of objects, we need to retort to casts again. Specifically, we need to rewrite the list interface to allow arbitrary items:

```
interface IList<Item> {
  // sort this list according to lessThan in IComp
  // ASSUMES: the objects implement IComp
  IList<Item> sort()

  // extract the objects from this list
  // that satisfy hasProperty from IPred
  // ASSUMES: the objects implement IPred
  IList<Item> select();
}
```

With an interface like that, your *sort* and *select* methods check during program execution whether the items on the list implement the *IComp* or *IPred* interface, respectively. If they don't, the library raises an error.

```
interface IPred {                          interface IComp {
  // does this object satisfy the property?   // is this object less than o?
  boolean hasProperty();                       boolean lessThan(Object o);
}                                          }
```

```
interface IList<Item> {
  // sort this list according to lessThan in IComp
  // ASSUMES: the objects implement IComp
  IList<Item> sort()

  // extract the objects from this list that satisfy hasProperty from IPred
  // ASSUMES: the objects implement IPred
  IList<Item> select();
}
```

```
class Mt<Item>                             class Cons<Item>
    implements IList<Item> {                   implements IList<Item> {
  …                                          …
  Mt<Item>() { … }                            Cons<Item>(Item i, IList<Item> l) { … }
  …                                          …
}                                          }
```

```
class PlainItem {
  …
}
```

```
class CompItem implements IComp {          class PredItem implements IPred {
  …                                          …
}                                          }
```

```
class AllItem implements IComp, IPred {
  …
}
```

Figure 179: A flexible list library and four kinds of list items

The top half of figure 179 sketches what this library design looks like to the rest of the world. This version of the list library exports three interfaces (*IList*, *IPred*, and *IComp*) plus two constructors (*Mt*, *Cons*). The bottom half sketches four classes whose instances may occur in an instance of *IList*:

1. *PlainItem* is a class that doesn't implement any interface. While you can create a list from its instances, you cannot design methods that sort those lists and you can't have methods that select sub-lists from them, either.

2. *CompItem* implements the *IComp* interface. Hence, a list that consists of *CompItem* objects is sortable.

3. Similarly, *PredItem* implements the *IPred* interface. Hence, you may design methods that extract lists with certain properties from lists of *PredItem* objects.

4. Lastly, you may design classes that implement two (and more) interfaces. *AllItem* is a class that implements both *IComp* and *IPred*, and it is thus possible to have methods that sort lists of *AllItem*s as well as methods that select sub-lists.

The key to the last class definition, *AllItem*, is yet another small extension to our notation:

class *AllItem* **implements** *IComp*, *IPred* { … }

As you can see, the **implements** clause lists two interfaces separated by commas. The meaning is obvious, i.e., the class must define all methods specified in all the interfaces that it implements. This notational extension is similar to the one for **extends** for interfaces; again, see section 32.1 and figure 169 for the general idea of **extends** and **implements** with multiple **interface**s.

Exercises

Exercise 33.10 Finish the design of this list library using generics. Use the library to represent the menus and phone books from exercise 33.7. ∎

Exercise 33.11 Design a list library like the one in exercise 33.10 using subtyping. Use the library to represent the menus and phone books from exercise 33.7. ∎

Given the analysis of this scenario and the preceding chapters, there appears to be a design tension. In the preceding chapters, we have made interfaces large, adding as many methods as we wished to provide as much functionality as needed (for any given problem) for a data representation. In this chapter, and especially in this subsection, our design analysis suggests making interfaces small, adding just as many method specifications as needed to let other designers—users of our libraries—know what their classes must implement.

A close look resolves this tension easily, producing a simple guideline:

GUIDELINE ON INTERFACES

Make an **interface** as large as needed when designing a data representation for a programming problem.

Make an **interface** as small as possible when using it to express constraints for designers who wish to use a generalized data representation.

In all cases, design to the interface that comes with data representations and libraries.

Of course, just as observed at the end of section 31, we don't really want lists that can be sorted in one manner only. Similarly, we don't want lists from which we can select sub-lists in just one way. Nevertheless, the lesson on **interface**s stands and you must keep it in mind as you move forward.

34   Extensible Frameworks: Abstracting Constructors

The discovery of this chapter is that data libraries are designed by generalizing the type of structurally similar classes and class frameworks. A data library consists of interfaces and implementing classes; programmers use such libraries to represent general forms of information, re-using all the methods that the interfaces specify and the classes implement. While the chapter has dealt with just one major example, it is easy to see how creating libraries can accelerate the design of many mundane programs.

Given this description of a data library, you can and should ponder these two questions:

1. What should you do if you wish to use a library that you can't change but has almost all the functionality needed to solve some problem?

2. What should you do if you wish to use a library that you can't change and that doesn't quite deal with all the data variants needed?

The questions look strange, considering that in the past we just edited classes and added whatever we needed. In reality, however, programmers are handed libraries and are told not to edit them[90] and then they are faced with just these questions.

```
                                +-----------+                      *
                                | IIfc      |                      *  DATA EXTENSION
                                +-----------+                      *
                                     |                             *
                                    / \                            *
                                    ---                            *
                                     |                             *
    +-----------------+-----------------------+------------------+-------*-------+
    |                 |                        |                 |       *       |
    |             +-----------+            +-----------+     +-----------+ * +-----------+
    |             | ACls      |            | BCls      |     | CCls      | * | DCls      |
    |             +-----------+            +-----------+     +-----------+ * +-----------+
    |                 |                        |                 |       *       |
====|=================================================================================
    | FUNCTION EXTENSION|                      |                 |       *       |
    |                 |                        |                 |       *       |
+-----------+  /|     |                        |                 |       *       |
| IExt      |- |----+--|-------------------+-|-------------------+--------------+ |
+-----------+  \|   | |                    | |                 | |     *   | | |
            +-----------+            +-----------+         +-----------+ * +-----------+
            | AxCls     |            | BxCls     |         | CxCls     | * | DxCls     |
            +-----------+            +-----------+         +-----------+ * +-----------+
```
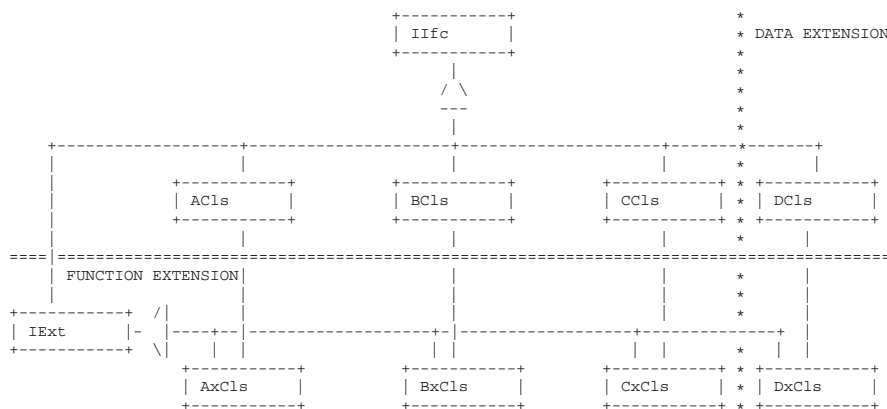
Figure 180: Data and functional extensions of frameworks

Figure 180 demonstrates the problem with a class diagram. The diagram is cut into four pieces with a vertical and a horizontal divider line. The top-left corner represents the library. As far as the world at large is concerned, this library provides an interface and three classes. The bottom-left corner shows a use of the library for a concrete purpose. For concreteness, imagine a general list library and a representations of menus, respectively. If the functionality exported via *IIfc* doesn't suffice for menus, the idea—first mentioned but not explored in section 31.2 (page 471)—is to add methods in this lower tier, unless you can edit the library.

The top-right of the figure 180 shows a fourth class that implements the *IIfc* interface from the library. It suggests another way of representing the information of interest as data. To make it compatible with the rest of

---

[90]In general, libraries don't come in a readable, textual form; even if they do, development conventions ensure that edits are ignored in the overall product.

the library, the new class implements the general interface. Finally, in the bottom right corner you see a piece of code that uses this data extension of the library and adds functionality to the overall library, too.

In short, you often don't just want libraries; you want extensible frameworks. One reason object-oriented programming languages became popular is because they promised a solution to just this kinds of programming problems. In this section we investigate this claim and how to approach the design of extensible frameworks.[91]

34.1   Data Extensions are Easy

Object-oriented programming languages do support the extension of data representations with new variants. Indeed, it is so straightforward that we just look at one example and exhibit a small problem with the exercises. Figure 181 presents a simplified list library: it serves as a "list of integers" representation, which are useful in numeric programs. For simplicity, the classes implement the familiar methods: *count*, *contains*, and *asString*; a numeric library would provide rather different methods.

As always, the world perceives the library as an interface and two implementing classes, i.e., two constructors. Given the list library, the representation of a numeric interval, say $[low, high]$, takes $high - low + 1$ instances of *Cons*. Thus it is easy to imagine that someone may ask you to address this problem:

> ... One way to make the representation of intervals compact is to design a new variant, dubbed *Range*, which represents an interval combined with a rest of a list. Do so without modifying the core library. ...

Figure 182 shows the result of such a design effort. The *Range* class implements the *IList* interface and is thus a variant of the list representation. The method definitions are straightforward:

count adds the number of integers in [*low*,*high*] to the number of integers in the rest of the list;

contains checks whether the searched-for integer is in the interval or in the rest of the list;

---

[91]For a full understanding of the solution and its implications, you will need to acquire knowledge about object-oriented program design that is beyond the scope of this book.

```
// a list of integers
interface IList {
  // how many objects are on this list?
  int count();
  // is the given object o on this list?
  boolean contains(int o);
  // render this list as a string
  String asString();
}
```

```
class Mt implements IList {
  Mt() {}

  public int count() {
    return 0;
  }

  public boolean contains (int o) {
    return false;
  }

  public String asString() {
    return "mt";
  }
}
```

```
class Cons implements IList {
  int first;
  IList rest;

  Cons(int first, IList rest) {
    this.first = first;
    this.rest = rest;
  }

  public int count() {
    return 1 + this.rest.count();
  }

  public boolean contains(int o) {
    return this.first == o || this.rest.contains(o);
  }

  public String asString() {
    return String.valueOf(first).concat(" ")
            .concat(rest.asString());
  }
}
```

Figure 181: A list of integers . . .

asString concatenates the mathematical rendering of an interval with the string that represents the rest of the list.

None of these should harbor any surprise for you.

Exercises

```
  // represents the interval [low,high] combined with an IList
class Range implements IList {
 int low;
 int high;
 IList rest;

 Range(int low, int high, IList rest) {
  this.low = low;
  this.high = high;
  this.rest = rest;
 }

 public int count() {
  return (high − low + 1) + rest.count();
 }

 public boolean contains(int o) {
  return ((low <= o) && (o <= high)) || rest.contains(o);
 }

 public String asString() {
 return "[".concat(String.valueOf(low))
        .concat("-")
        .concat(String.valueOf(high))
        .concat("] ")
        .concat(rest.asString());
 }
}
```

Figure 182: ... and a data extension

Exercise 34.1 Develop examples and tests for all methods of figure 181. Extend the test class to cope with the library extension in figure 182. ∎

Exercise 34.2 Design a *sort* method for the library of figure 181.

Then add a *sort* method to the library extension of figure 182 without modifying the core library. **Hint**: To solve this problem, you will need to design a method that inserts each element in a range into the sorted list. As you do so, remember the lessons concerning the design of functions that process natural numbers. Also see section 34.4. ∎

The second exercise indicates how a data extension may suggest a function extension as well. A moment's thought shows that the extension of the union with a *Range* variant demands the addition of a method that inserts a range into a list. We haven't covered function extensions, however; so doing so has to wait.

34.2 Function Extension: A Design that doesn't Quite Work …

As in the preceding subsection and sections, this section illustrates libraries with function extensions through a single example. Let's start with a problem statement:

> … Design a representation of menus that supports four pieces of functionality: counting the items on a menu; rendering them as a string; sorting the items by price; and selecting the sub-menu of items that cost between $10 and $20. You must start from the list library in (the top half of) figure 183. …

The last sentence emphasizes again that you are no longer designing complete programs from scratch but adding components to an existing world.

Following the general idea in figure 180, we extend *IList*, *Mt*, and *Cons* with an interface and classes for menus, respectively. This lower tier also adds the *select* methods to menus because it isn't provided by the library itself. Figure 183 displays the class diagram for this concrete case. The *IMenu* interface extends *IList* and specifies *select*. The *MtMenu* and *Cons-Menu* classes extend *Mt* and *Cons* and implement the *IMenu* interface. Note how the *ConsMenu* class refers neither back to *IMenu* nor to *MenuItem* it via a containment arrow.

Translating the bottom half of figure 183 into interfaces, classes and methods (including examples and tests) is straightforward. You may wish to practice this step again before proceeding from here; the result (minus *MtMenu*) is shown in figure 184. As you can see there, the *ConsMenu* class hands over its argument to the super constructor. Hence, the method definitions suffer from the same type-mismatch problem that we already encountered in section 30.2. Take a look at *select* in *ConsMenu*, which contains two gray-shaded casts via variable declarations. As before, we—the designers and programmers—see that the constructor always uses the **super** constructor with an instance of *MenuItem* and list of type *IMenu*. The type checker does not use our kind of reasoning, however. From the perspective

```
                                                              // The Object List Library

          +---------------------------+
          | IList                     |<------------------+
          +---------------------------+                   |      +--------------------+
          +---------------------------+                   |      | IComp              |
          | int count()               |                   |      +--------------------+
          | boolean contains(Object x)|                   |      +--------------------+
          | String asString()         |                   |      | lessThan(Object o) |
          | IList sort()              |                   |      +--------------------+
          +---------------------------+                   |
                        |                                 |
                       / \                                |
                       ---                                |
                        |                                 |
             ----------------------------------           |
             |                   |           |            |
    +----------------------+     |   +------------------------+ |
    | Mt                   |     |   | Cons                   | |
    +----------------------+     |   +------------------------+ |
             |                   |   | Object first           | |
             |                   |   | IList rest             |--+
             |                   |   +------------------------+
             |                   |                |
            / \                  |               / \
            ---                  |               ---
             |                   |                |
    ==================================================================================================
             |                   |                |
             |                   |                |
             |     +---------------------------+  |
             |     | IMenu                     |  |
             |     +---------------------------+  |
             |     +---------------------------+  |
             |     | IMenu select()            |  |
             |     +---------------------------+  |
             |                   |                |
             |                  / \               |
             |                  ---               |
             |                   |          +-----+
             |        -----------------------+    |
             |        |                      |    |         +--------------------+
             |        |                      |    |         | MenuItem           |
    +-----------------+  +------------------------+         +--------------------+
    | MtMenu          |  | ConsMenu               |         +--------------------+
    +-----------------+  +------------------------+         | hasProperty)()     |
    +-----------------+  +------------------------+         +--------------------+
```
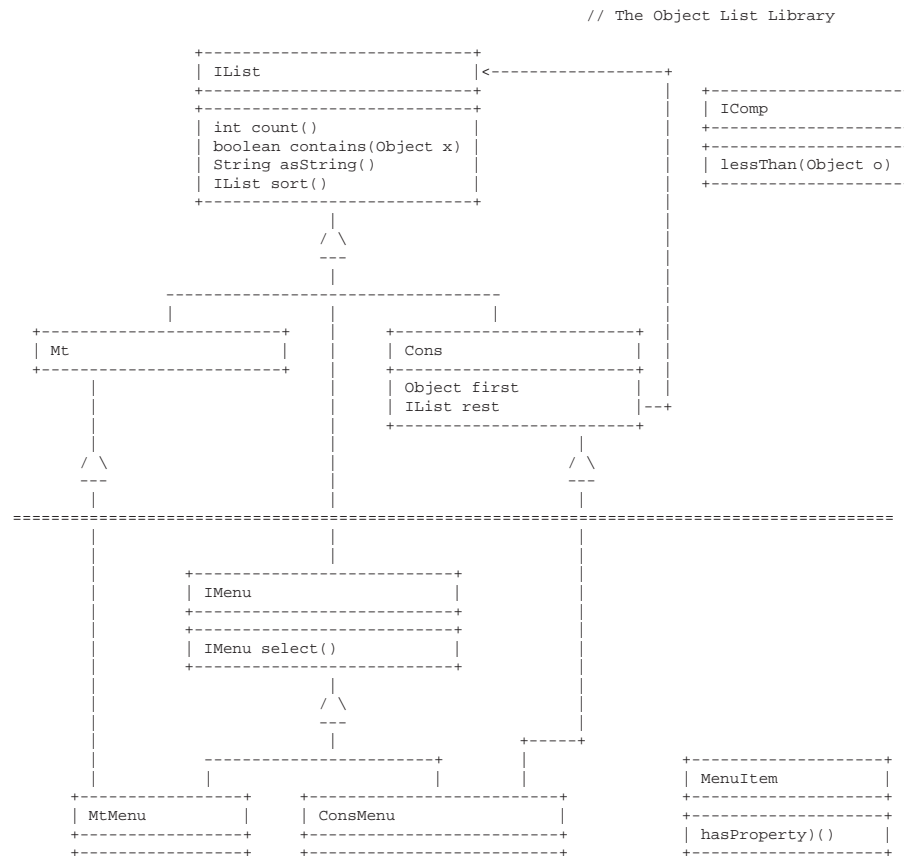
Figure 183: Deriving sortable lists from *Object* lists, the class diagram

of the type checker, *rest* has type *IList*.[92] Thus if you wrote

   rest.select()

the type checker would fail. Instead, the method uses local variable definitions to cast *rest* to *IMenu* and *first* to *MenuItem*. The rest of the method body refers to these local variables instead of the fields.

Exercises

---

[92]Remember from footnote 72 though that some programmer could derive a subclass from *ConsSort* that assigns instances from other, unrelated classes into these fields.

```
interface IMenu extends IList {
  // select that part of this menu whose items satisfy hasProperty
  IMenu select();
}
```

```
class ConsMenu extends Cons implements IMenu {
  ConsMenu(Object first, IMenu rest) {
    super(first,rest);
  }

  public IMenu select() {
    MenuItem fst = (MenuItem)first;
    IMenu rst = (IMenu)rest;
    if (fst.hasProperty()) {
      return new ConsMenu(fst,rst.select()); }
    else {
      return rst.select(); }
  }
}
```

Figure 184: Representing menus with *Object* lists

Exercise 34.3 Define *MtMenu* in analogy to *ConsMenu* for the library in figure 184. Also design a *MenuItem* class that represents item on a menu and supports the method *hasProperty*. Finally, develop examples and tests for *IMenu* in figure 184. ∎

Exercise 34.4 Design a generic counterpart to the extensible framework of figures 183 and 184. ∎

At first glance, everything appears to be working just fine but it isn't. The problem is that the methods for menus aren't collaborating in the expected manner with the methods of the list library. For a specific example, imagine a customer who wishes to sort the menu and to select items that cost between $10 and $20. Here is a formulation of this scenario as a pair of tests:

inside of *MenuExamples* :
MenuItem fish = **new** *MenuItem*("Fish & Chips",1295);
MenuItem steak = **new** *MenuItem*("Steak & Fries",2095);
MenuItem veggie = **new** *MenuItem*("French Veggies",1095);

IMenu mt = **new** *MtMenu*();
IMenu m1 = **new** *ConsMenu*(*fish,mt*);
IMenu m2 = **new** *ConsMenu*(*steak,m1*);
IMenu m3 = **new** *ConsMenu*(*veggie,m2*);

. . . *checkExpect*(*m3.select*().*sort*(),. . . ,"composing select and sort") . . .
. . . *checkExpect*(*m3.sort*().*select*(),. . . ,"composing sort and select") . . .

Selecting first and then sorting should produce the same result as sorting first and then selecting, at least as long as both test cases use the same sorting and selection criteria. Alas, only the first test type-checks; the second one doesn't because *sort* produces an *IList* not an *IMenu* and the former doesn't implement the *select* method.

Right here, you may realize that defining the lower tier in figure 184 corresponds to the testing step in the design recipe of section 32. This step calls for overriding all method definitions that don't match the desired type. More specifically, it suggests using a context-specific signature for the *sort* method and defining it with the help of casts and **super** calls:

inside of *IMenu* :   inside of *ConsMenu* :
IMenu sort();   **public** *IMenu sort*() {
                                              IMenu result = (*IMenu*)**super**.*sort*();
                                              return *result*;
                                            }

Once these changes are in place, both of the above tests type-check.

Unfortunately, evaluating the tests stops with an exception:

```
Exception in thread "main" java.lang.ClassCastException: Cons
  at ConsMenu.sort ...
```

The first line of this text tells you that your program evaluation failed because a cast failed. Specifically, the cast found a *Cons* where it expected something else. The second line pinpoints the cast that failed. Here it is the cast that the *sort* method in *ConsMenu* needs in order to turn the result of the **super**.*sort*() call into a menu:

```
inside of ConsMenu :
public IMenu sort() {
  IMenu result = (IMenu)super.sort();
  return result;
}
```

In short, *result* is not made up of objects that implement *IMenu*, but of instances of *Cons* and *Mt*.

On second thought the exception shouldn't surprise you. An inspection of *sort* in *Mt* and *Cons* shows that it creates lists with *Mt* and *Cons*. Since neither of these classes implements *IMenu*, the cast from the result of *sort* to *IMenu* must fail.

At this point, you might think that the design of *sort* should have used *ConsMenu* instead of *Cons*. Doing so, however, is wrong because the upper tier in figure 183 is a library that is useful for representing many different kinds of data, including menus, electronic phone books, and others. Separating those two tiers is the point of creating a widely useful abstraction.

The conclusion is thus obvious:

WARNING ON FRAMEWORK EXTENSIONS

Extending a framework with an additional layer of functionality fails, if the methods in its classes uses any of its constructors.

If so, any call to the respective methods creates instances of the original library, not the extension.

34.3 . . . and how to Fix it (Mostly)

The warning isn't just negative. It says that designers must prepare libraries for function extensions; it doesn't happen automatically. As a matter of fact, it also suggests a way to do so. To understand why, re-read the warning. In our running example, the *sort* method employs the constructor. Hence, if some other code invokes this inherited method on some list of (sub-)type *ISort*, it returns a plain list.

An apparent solution is to override *insert* in *MtMenu* and *ConsMenu* so that they use the proper constructors:

inside of *MtMenu* :
public *IMenu insert*(*IComp o*) {
  return new *ConsMenu*(*o*,**this**);
}

inside of *ConsMenu* :
public *IMenu insert*(*IComp o*) {
  if (*o.lessThan*(*first*)) {
    return new *ConsMenu*(*o*,**this**); }
  else {
   return
     new *ConsMenu*(*first*,*rest.insert*(*o*)); }
  }
}

While doing so solves the problem, it mostly duplicates a method and creates an obvious candidate for abstraction. After all, the definitions of *sort* in the library and in its extensions are equal up to the constructors. Sadly, though, it is impossible to abstract over constructors or its return types, a problem we have encountered seen several times now.

It is possible, however, to introduce another method that uses the constructor and to use it in place of the constructor:

inside of *MtMenu* :
public *IMenu insert*(*IComp o*) {
  return *factory* (*o*, *factoryMt* ());
}

inside of *ConsMenu* :
public *IMenu insert*(*IComp o*) {
  if (*o.lessThan*(*first*)) {
    return *factory* (*o*,**this**); }
  else {
   return *factory* (*first*,*rest.insert*(*o*)); }
  }
}

Naturally the methods *factory* and *factoryMt* just mimic the constructors:

IList factoryMt() {
  return new *Mt*();
}

IList factory(Object *f* , *IList r*) {
  return new *Cons*(*f* ,*r*);
}

Now there is no need to copy the definitions of *sort* and modify them; it suffices to override them if you also override the two *factory* methods in each class:

IMenu factoryMt() {
  return new *MtMenu*();
}

IMenu factory(Object *f* , *IList r*) {
  return new *ConsMenu*(*f* ,(*IMenu*)*r*);
}

These new definitions create instances of *MtMenu* and *ConsMenu*. And hence, *sort* on *IMenu*s produces menus, too.

```
interface IList {
  . . .
  // sort this list, according to lessThan
  IList sort();

  // insert o into this (sorted) list
  IList insert(IComp o);

  // factory method for Cons
  // extensions must override this method
  IList factory(Object f, IList r);

  // factory method for Mt
  // extensions must override this method
  IList factoryMt();
}
```

```
class Mt implements IList {
  . . .
  public IList sort() {
    return this;
  }


  public IList insert(IComp o) {
    return factory(o,factoryMt());
  }




  public IList factoryMt() {
    return new Mt();
  }

  public IList factory(Object f, IList r) {
    return new Cons(f,r);
  }
}
```

```
class Cons implements IList {
  . . .
  public IList sort() {
    IList rst = (IList)rest;
    return rst.sort().insert((IComp)first);
  }

  public IList insert(IComp o) {
    if (o.lessThan(first)) {
    return factory(o,this); }
    else {
    return factory(first,rest.insert(o)); }
  }



  public IList factoryMt() {
    return new Mt();
  }

  public IList factory(Object f, IList r) {
    return new Cons(f,r);
  }
}
```

Figure 185: An extensible list library

```
interface IMenu {
  …
  // sort this list, according to lessThan
  IMenu sort();

  // factory method for Cons
  // extensions must override this method
  IMenu factory(Object f, IMenu r);

  // factory method for Mt
  // extensions must override this method
  IMenu factoryMt();
}
```

```
class MtMenu implements IMenu {
  …
  public IMenu sort() {
    return factoryMt();
  }




  public IMenu factoryMt() {
    return new MtMenu();
  }


  public IMenu factory(Object f,
                          IMenu r) {
    return new ConsMenu(f,r);
  }
}
```

```
class Cons implements IMenu {
  …
  public IMenu sort() {
    IMenu r = (IMenu)rest;
    IComp f = (IComp)first;
    return r.sort().insert(f);
  }


  public IMenu factoryMt() {
    return new MtMenu();
  }


  public IMenu factory(Object f,
                          IMenu r) {
    return new ConsMenu(f,r);
  }
}
```

Figure 186: … and its extension

Put a bit more abstractly, we have used an arrangement similar to the template-and-hook pattern (page 238) for abstraction. Here the templates are all those methods that use *factory* and *factoryMt* to create lists; the two factory methods are the hooks with which we can extend the meaning of these templates. Of course, there is no single template and there is no single hook. And for this reason, this arrangement has its own name: the "factory method" pattern.

In general, using the "factory method" pattern prepares a framework of interfaces and classes for future extensions with a tier of additional functionality. Our FACTORY METHODS have the same signature as a constructor and perform the same function; all the other methods in the library use factory methods instead of the constructor. If some designer wishes to extend this library framework, the extension must override the factory method so that it creates instances of the extension. Thus, all inherited methods with references to a factory method use the overridden method and, when invoked, create instances of the extension tier, too.

Figures 185 and 186 demonstrate how this pattern works for our concrete example. The first figure displays the portion of the interface and the classes that specify factory methods and their use in *sort* and *insert*. To inform future extension programmers, the comments in the *IList* interface flag the factory methods as such and request an overriding definition in classes that extend *Mt* and *Cons*.

The second figure displays the interface and class definitions of the second tier, i.e., the extension of the list library that represent menus. The *IMenu* interface overrides the *sort* method so that sorting menus produces menus. Its two implementing classes override *sort* with definitions that accommodate the refined type of *IMenu*, using either the appropriate *factory* method or the **super** method plus casts. To ensure that the **super** call produce instances of *IMenu*, both implementing classes also override the *factory* methods as advertised above.

Exercises

Exercise 34.5 Develop examples and tests that demonstrate the proper behavior of both the original list library (figure 185) as well as its extension 186. ∎

Exercise 34.6 The *factory* method definitions in *Cons* are identical to those in *Mt* in figure 185. Abstract them. Start with a revised class diagram then change the class and method definitions. Use the tests of exercise 34.5 to ensure that the abstraction didn't introduce typos.

Question: Is it possible to use an analogous abstraction for the same commonalities in the lower ties, i.e., in the function extension of the library of figure 186? If possible, do so; if not explain why. ∎

Exercise 34.7 Design the method *remove* for the library figure 185. Extend the tests from exercise 34.5 to ensure that *remove* works properly with the library extension. ∎

Exercise 34.8 Since the representation of menus in figure 186 is designed to be extensible, you can add another function extension layer.

Design a function extension of the menu representation that can extract vegetarian, vegan, and gluton-free menus from a given menu. The items on the menu must of course provide appropriate methods. Ensure that sorting a menu, extracting a reasonably priced sub-menu, and then focusing on vegetarian items can be done in any order. ∎

Exercise 34.9 A proper object-oriented design for exercise 34.2 extends the integer list library of figure 181 with a function extension as well as a data extension, along the lines of figure 180:

1. *IListX* is the interface that extends *IList* for integers from figure 181.

2. The function extension equips the library with a method for inserting an interval into a list:

   > inside of *IListX* :
   > // insert the interval [*low*,*high*] into **this** sorted list
   > *IListX insert*(int *low*, int *high*);

3. While you could introduce a *Range* class for the data extension and a *RangeX* class for the function extension, you may instead wish to design a single class that extends the library in both dimensions. ∎

34.4   Function Extension: Take 2

When you first encountered the problem of designing a menu representation that supports certain methods (see page 34.2), you might have wondered why we didn't just use one of the representations from chapters I and II. In particular, we could introduce a menu class that keeps track of all menu items via some field. In case we expect variations of menus, we could also define an interface for menus so that it is easy to add other implementing classes later on.

Figure 187 displays a code-based sketch of such a solution. The interface on the left specifies all five methods; the class on the right shows how to design the *sort* method via a dispatch to a *sort* method for the *items* list. The dispatch follows the containment arrow of a diagram, using the plain design recipe from the first two chapters. Convince yourself that *count*, *contains* and *asString* can be designed in an analogous manner.

The *select* method is left unspecified because it isn't possible to design it in this straightforward manner. The chosen list library doesn't support

```
interface IMenu {
  // how many items are on this menu?
  int count();
  // does this menu contain an item named i?
  boolean contains(String o);
  // render this menu as a String
  String asString();
  // select those items from this menu
  // that cost between $10 and $20
  IMenu select();
  // sort this menu by price
  IMenu sort();
}
```

```
class Menu implements IMenu {
  IList items;
  Menu(IList items) {
    this.items = items;
  }

  . . .

  public IMenu sort() {
    IList sortedItems = items.sort();
    return new Menu(sortedItems);
  }

  public IMenu select() {
    return ???;
  }
}
```

Figure 187: Menus like stacks

a method for selecting items from a list, and the method therefore can't just dispatch as *sort* does. One thing we could try is to design a private, auxiliary method in *Menu* for filtering lists and create a menu from the result:

inside of *Menu* :
```
public IMenu select() {
  return new Menu(selectAux(items));
}
```

This definition assumes of course that we can solve the following programming problem:

inside of *Menu* :
```
// select those items from items that cost between $10 and $20
private IList selectAux(IList items) {
```

The most important point to notice is that the purpose statement cannot (and does not) refer to **this** menu; instead it talks about the list parameter of the method.

As it turns out, we must remember our earliest experiences with lists and the generalization of design recipes to recursive data representations. From the perspective of *selectAux*, a list of menu items is defined like this:

> A *list of menu items* is either
>
> 1. an instance of *Mt*; or
>
> 2. an instance of *Cons*, constructed from a *MenuItem* and a list of menu items.

As always, creating data examples follows the data definition: **new** *Mt*(), **new** *Cons*(*m1*,**new** *Mt*()), etc. for some *MenuItem m1*. From there you can also create functional examples:

1. *selectAux*(**new** *Mt*()) should produce **new** *Mt*();

2. the result of *selectAux*(**new** *Cons*(**new** *MenuItem*(`"Fish & Chips"`,1295), . . .)) should start with **new** *MenuItem*(`"Fish & Chips"`,1295);

3. while evaluating *selectAux*(**new** *Cons*(**new** *MenuItem*(`"Steak"`,2295), . . .)) should produce a list without this first item.

In order to create a template, we need a predicate that distinguishes instances of *Mt* from other objects and selectors that provide access to the fields of a *Cons* instances. Fortunately, Java provides the *instanceof* operator and, because the fields in *Cons* aren't protected, we are free to use field references to get access to their content. Thus the conventional template looks roughly like this:

<u>inside of *Menu* :</u>
```
private IList selectAux(IList items) {
    if (items instanceof Mt) {
        return ...; }
    else {
        ... items.first ... selectAux(items.rest) ...; }
}
```

The flaw of this template—as any attempt to type check this code shows you—is that *items* is of type *IList* not *Cons*, and it is therefore illegal to extract the *first* and *rest* fields.

You should recall at this point that casts fix such problems:

<u>inside of *Menu* :</u>
```
private IList selectAux(IList items) {
    Cons c;
    MenuItem m;
    IList r;
    if (items instanceof Mt) {
       return new Mt(); }
    else {
     c = (Cons)items;
     m = (MenuItem)c.first;
     if (m.hasProperty()) {
        return new Cons(m,selectAux(c.rest)); }
     else {
        return selectAux(c.rest); }
    }
}
```

Using the casts poses no danger here, because the **if** statement has established that *items* is an instance of *Cons* for the **else**-branch. The casts are only used to make the type checker accept the method. Include *selectAux* in the *Menu* class and test it.

Our solution suffers from two problems. First, the design is not based on the class-based representation of the menu, but on a functional interpretation of the data. It is not object-oriented. Second, the design assumes that the fields of *Cons* objects are accessible. If the designer of the list library had followed the recommendations of section 20.4, however, the fields would be private and thus inaccessible to others. In that case, we simply wouldn't be able to design *selectAux*.

What this suggests and what other sections in this chapter have suggested is that a list library such as ours should provide a method for inspecting every item on the list. In *How to Design Programs*, we have encountered several functions: map, filter, foldl, and so on. We also worked out how to design such functions for arbitrary data representations. Clearly, class-based libraries need equivalent methods if others are to design additional functionality for them.

In general then, the designers of data libraries face a choice. Since they cannot anticipate all possible methods that their future users may want, they must either turn the library into an extensible framework (as explained in this section) or they must prepare it for arbitrary traversals. The next chapter shows how to implement this second choice.

Exercise

Exercise 34.10 When you design classes that use a library, you should program to the library's interface. In our running example, you should design the *selectAux* method by using the *IList* interface. As discussed, this isn't possible because *selectAux* uses *instanceof* and accesses fields.

Modify the list library so that you need neither *instanecof* nor field access to define *selectAux*. The former was dealt with in the preceding chapter. Section 33.2 addresses the issue of accessing fields via interfaces. Here we propose this specific change:

> inside of *IList* :
> // is there another item on **this** list?
> boolean *hasNext*();
>
> // retrieve the first item from **this** list
> // ASSUMES: **this** list has next: **this**.*hasNext*()
> Object *next*();
>
> // retrieve the rest of **this** list
> // ASSUMES: **this** list has next: **this**.*hasNext*()
> *IList getRest*();

Once you have modified the list library, re-formulate *selectAux* using these new methods.

Would this kind of extension continue to work if you were to add a *Range*-like variant (see subsection 34.1) to the list library? ▮

34.5   Mini Project: The Towers of Hanoi

Many books on programming use a puzzle known as "Towers of Hanoi" to illustrate recursive programming.[93] While it might be possible to justify such an action, our interest here is not in designing a recursive method for solving a "Towers of Hanoi" puzzle. Instead, our goal is to design a graphical program interface that allows people to play the game.

A "Towers of Hanoi" puzzle consists of three poles; we refer to them as "left pole," "middle pole," and "right pole." Three disks of different sizes

---

[93]In the terminology of *How to Design Programs*, a program that solves an instance of the "Towers of Hanoi" puzzle illustrates *generative recursion*.
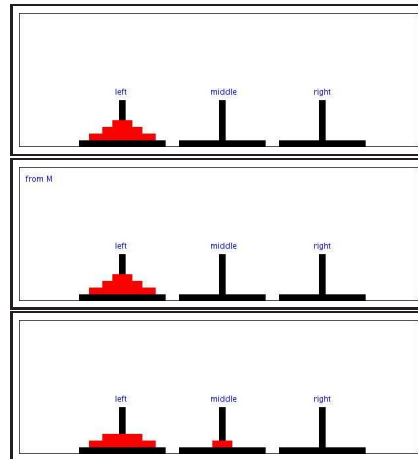
Figure 188: The towers of Hanoi

with a hole in the middle are stacked up on the left pole. The largest disk is at the bottom of the stack; the second largest is in the middle; and the smallest is on top. The player must move the disks from the left pole to the right pole, one disk at a time, using the middle pole as an intermediate resting point. As the player moves one disk, it may not come to rest on top of a disk that is smaller.

Your graphical program should use a simple white canvas to draw the current state of the puzzle. See the top-most screen shot in figure 188 for a drawing of the initial state of the puzzle. A player should specify the movement of disks by pressing single-character keys on the keyboard. Specifically, if the player presses the letters "L", "M," or "R" for the first time, your program should prepare the move of the top-most disk from the left pole, middle pole, or right pole, respectively. Since pressing such an "action key" puts your program into a different state, your canvas should reflect this change; it should indicate what it is about to do. As the second screen shot in figure 188 shows, our solution places a string at the top-left corner of the canvas that specifies from where the player is about to remove the next disk. Finally, when the player presses one of those three letter keys again, the program completes the move. See the bottom most screen shot in figure 188 and also note that the message has disappeared. The program is ready to perform the next disk move.

The objective of this mini project is to use what you have learned in this chapter: the design and re-use of data representation libraries; their adaptation to new situations; and the creation of extensible frameworks from libraries. As always, the following series of exercises begins with a top-down plan and then proceeds with a bottom-up design of classes and methods. The section ends with an integration of the pieces in the "World of Hanoi" class.

You are going to face two design choices as you solve the following exercises. One concerns the notion of generics; the other one the use of a function extension in the spirit of section 34.3 or 34.4. In short, there are four major outcomes possible based on your current design knowledge. We encourage you to design as many of these choices as you have time for.

Exercises

Exercise 34.11 The "Tower of Hanoi" puzzle clearly deals with a world of disks and poles, but also stacks and lists. See section 32.4 if you have forgotten about stacks.

Design a data representation, starting with a class that extends *World* from the **idraw** or **draw** library. Focus on the essence of the objects.

You should start this step with a class and interface diagram, and you should finish it with data examples. ∎

Exercise 34.12 Design a complete data representation for disks. Equip the class with a method for drawing a disk into a canvas at some given x and y coordinates. ∎

Exercise 34.13 Design a generally useful library for representing stateful stacks based on applicative lists:

1. The *Stack* class should support methods for pushing items on top of the stack; retrieving the top item from a stack; popping the first item off the stack; and determining its current depth and whether the stack is empty.

2. The *IList* interface, which specifies that lists support the methods of retrieving the first item, retrieving the rest of the list, determining its length and whether it is empty.

3. The *Mt* class, which implements *IList* and provides a constructor of no arguments.

4. The *Cons* class, which implements *IList* and provides the usual two-argument constructor.

Indicate whether you choose an approach based on subtyping or generics or both. ∎

Exercise 34.14 Design a data representation for poles. Since a pole acts like a stack of disks, you must re-use the stack and list library from exercise 34.13 and the disk representation from exercise 34.12. In doing so, ensure that, for example, the method for retrieving the top-most item actually returns a disk.

Equip the data representation with a method for drawing the pole and the disks on the pole into a canvas at some given x and y coordinates. Doing so you are facing a choice with two alternatives: you may either use the traversal-based design from this section or the design of extending the functionality of the list representation of the library. After all, the list representation doesn't come with a method for rendering itself on a canvas. ∎

Exercise 34.15 Design the class *Hanoi*, which extends *World* and uses the results of exercises 34.11 to 34.14 to implement a game for solving three-disk "Tower of Hanoi" puzzles.

When you have solved the problem and the puzzle to your satisfaction, consider what it takes to generalize the program so that players can specify the number of disks in the initial configuration (with or without limits). ∎

If you are curious about how to solve "Towers of Hanoi" puzzles in general, here is the *eureka!* insight. Suppose the initial tower has n disks. In that case you move $n - 1$ disks to the middle pole. That is, you ignore the bottom disk. Since the bottom disk is the largest, you can act as if it didn't exist while you solve the "Towers of Hanoi" puzzle for $n - 1$ disks. When you have done so, you are free to move the largest disk from the left pole to the right pole. This creates another "Towers of Hanoi" puzzle, namely, move $n - 1$ disks from the middle pole to the right pole using the left pole as an auxiliary. And then you are done.

As you can see, solving such a puzzle means solving two smaller puzzles that are like the original one and smaller than the original one. The former insight tells you that (generative) recursion is the tool of choice for the discriminating "Towers of Hanoi" puzzler, and the latter tells you that at some point you will finish. So what is the base case?

Intermezzo 5: Generic Classes

syntax

typing and subtyping

BUT semantics: compile to the Object based solution with subtyping

error messages are weird:

IList¡I¿; if you drop the ¡I¿, it is not an illegal program but you may get strange messages; ditto for Cons¡I¿ vs Cons

;; —

C#, a language that is similar to Java, deals with generics differently.

instances of *Mt* or *Cons* instantiated at the exact same type. In particular, using Object in place of *String* above violates this rule.

Just because a type *A* is below some other type *B*, Java does not consider *IList<A>* a subtype of *IList<B>*.

Exercise 35.1 Experiment with the specification syntax for type parameters. In particular, omit the constraints from the *Mt* and *Cons* class headers. What kind of error does your Java implementation signal? ∎

Purpose and Background

The purpose of this chapter is to introduce the idea of abstraction over similar traversals in class hierarchies. Studying such cases for abstraction naturally leads to visitors. It also requires the introduction of objects-as-functions and type-parameterized methods.

The visitor for lists at `void` type is `forEach`. Surprisingly, you can perform interesting computations with `forEach` despite the poverty of `void`. This prepares students for the introduction of loops proper in the next chapter.

We assume that students have understood chapter III and that familiar with the idea of loops in *How to Design Programs*(chapter iv) specifically `map`, `filter`, and friends.

After studying this chapter, students should be able to abstract over traversals, design visitors, and to use objects as functions.

TODO

Abstracting Data Traversals

Many methods have the purpose of traversing compound data and "collecting" data from such traversals (all in the form of objects). Some methods search lists or trees or graphs for objects with certain properties; other methods collect information from all of them and combine it; and yet others modify all items in a uniform manner.

If you look closely, these methods share a lot of basic elements with each other. Bluntly put, they all follow a similar pattern and writing this same pattern over and over again must violate your sense of proper abstraction. At the same time, these patterns don't fall into the abstraction categories of the preceding chapters, so even if you recognized the problem, you couldn't have abstracted yet.

The lack of abstract traversals also poses a problem for the design and use of so-called collection libraries. When programmers design libraries for collections, they must anticipate the methods that are most useful. Of course, it is generally impossible to anticipate them all. Hence, the programmers who use the library may have to create extensions to supply the missing functionality. As the preceding chapter has shown, such extensions are difficult to build in general and, once built, are difficult to maintain. If a library offers a general traversal method, however, designing additional functionality is usually straightforward.

Here we discuss how to design a general traversal method for a data structure library and how to design uses of such traversal methods. We also discuss an imperative traversal method and—in preparation of the next chapter—how it is often abused for applicative computations.

36   Patterns in Traversals

The preceding chapter deals with restaurant menus and their data representations. The emphasis there is on creating representations that re-use

```
interface IMenu {
  // select the items from this menu that cost between $10 and $20
  IMenu selectByPrice();
  // select the items from this menu that do not start with "Meat:"
  IMenu selectByMeat();
}
```

```
class Mt implements IMenu {
  // constructor omitted

  public IMenu selectByPrice() {
    return this;
  }

  public IMenu selectByMeat() {
    return this;
  }
}
```

```
class Cons implements IMenu {
  private MenuItem first;
  private IMenu rest;
  // constructor omitted

  public IMenu selectByPrice() {
    IMenu r = rest.selectByPrice();
    if (first.hasGoodPrice()) {
      return new Cons(first,r); }
    else {
      return r; }
  }

  public IMenu selectByMeat() {
    IMenu r = rest.selectByMeat();
    if (first.hasNoMeat()) {
      return new Cons(first,r); }
    else {
      return r; }
  }
}
```

```
class MenuItem {
  private String name;
  private int val; // in dollars

  public MenuItem(String name, int val) {
    this.name = name;
    this.val = val;
  }

  // is this item's cost betw. $10 and $20
  public boolean hasGoodPrice() {
    return 10 <= val && val <= 20;
  }

  // is this item free of "Meat:"
  public boolean hasNoMeat() {
    return !(name.startsWith("Meat:"));
  }
}
```

Figure 189: A menu with two select methods

existing data libraries. Here we are interested in methods for processing menus. Let's use a concrete problem statement to get started:

> ... Your company is selling software for managing restaurants menus like these:

```
              Today's Menu

  Garlic Pasta ....................   9
  Vegetarian Pizza ................  13
  Meat: Pepperoni Pizza ...........  18
  Meat: Steak and Fries ...........  21
  Gnochi ..........................  16
```

> As you can see, this restaurant takes care of its vegetarian clientele with a "`Meat:`" warning at the head of the line.

> Design a menu representation that allows customer to select those items on the menu that cost between \$10 and \$20 (inclusive) and those that are vegetarian. ...

36.1   Example: Menus Designed from Scratch

To keep things simple, we start with a solution that assumes the data representation for menus is designed from scratch. Figure 189 shows the expected collection of classes:

***MenuItem*:** This class represents individual menu items. Given the problem context, it supports two methods: *hasGoodPrice*, which determines whether a menu item costs between \$10 and \$20; and *hasNoMeat*, which ensures that the item's description doesn't start with "`Meat:`".

***IMenu*:** The interface represents menus as a whole. As you can see from the signature, this data representation of menus supports two methods, one for selecting entrees in a certain price range and another one for selecting dishes not marked with "`Meat:`".

***Mt*:** The *Mt* class, for empty menus, implements the interface in the predicted manner. Just note that both selection methods have the same method body.

***Cons*:** The *Cons* class extends an existing menu with another instance of
MenuItem. Pay close attention to the two selection methods; they
are so similar that they are nearly indistinguishable. The first, *select-
ByPrice*, uses the *hasGoodPrice* method from *MenuItem*. In its place,
the second one, *selectByMeat*, uses the *hasNoMeat* method.

Of course, in the real world we would use a field to designate a menu item
as a non-meat item; here we just use the structure of the *MenuItem* that you
know from the preceding section.

   The last clause in the enumeration points out the problem in detail.
Here it is again with the differences high-lighted:

```
public IMenu selectByPrice() {            public IMenu selectByMeat() {
 IMenu r = rest.selectByPrice();            IMenu r = rest.selectByMeat();
 if (first. hasGoodPrice ()) {              if (first. hasNoMeat ()) {
   return new Cons(first,r); }                return new Cons(first,r); }
 else {                                     else {
   return r; }                                return r; }
}                                         }
```

The two method definitions from *Cons* are alike except for the two method
calls in gray. If you had encountered such a problem in the context of *How
to Design Programs*, you would have abstracted over the different methods
with an additional parameter, and the two method definitions would have
been identical after that.

   In the world of Java, abstracting over methods is impossible. Unlike
functions in Scheme, methods are not first-class values. Still, the idea of
abstracting over the method call *as if* methods were first-class objects is
what it takes to avoid this kind of code duplication. Before we do that,
however, let's look at the menu problem from a slightly different angle.

Exercise

Exercise 36.1 Design a data representation for menu items that allows a
proper accommodation of dietary restrictions (e.g., kosher, vegan, etc). At
a minimum, make sure a dish is properly designated as vegetarian or not. ∎

36.2   Example: Menus Designed as Lists

Once you have understood the preceding chapter, you should first look in
your collection of available libraries before you design a data representa-

```
interface IPred {
  // does this item have a property?
  boolean good();
}
```

```
interface IList<I extends IPred> {
  // select the items from this list
  // that satisfy good (in IPred)
  IList<I> select();
}
```

```
class Mt<ITEM extends IPred>
    implements IList<ITEM> {
  // constructor omitted

  public IList<ITEM> select() {
    return this;
  }
}
```

```
class Cons<I extends IPred>
    implements IList<I>  {
  private I first;
  private IList<I> rest;
  // constructor omitted

  public IList<I> select() {
    IList<I> r = rest.select();
    if (first.good()) {
      return new Cons<I>(first,r); }
    else {
      return r; }
  }
}
```

```
class MenuItem implements IPred {
  private String name;
  private int val; // in dollars

  // constructor intentionally omitted

  public boolean good() {
    return 10 <= val && val <= 20;
  }
}
```

```
class MenuItem implements IPred {
  private String name;
  private int val; // in dollars

  // constructor intentionally omitted

  public boolean good() {
    return !(name.startsWith("Meat:"))
  }
}
```

Figure 190: Selecting items from a menu

tion from scratch. Since menu items come and go, it is obvious to identify menus with lists—at least those we have dealt with here.

Figure 190 spells out the details of this approach. Its top-half shows a generics-based list library that supports a single method: *select*, because this is all we need. The method traverses the lists and produces a list whose items produce true when their *good* method is invoked. The latter is specified in the *IPred* interface of the library.

Using this library requires the design of a class *C* that implements *IPred*. Instances of *C* can then be collected in an instance of *IList<C>*. For our specific case, you need to design the *MenuItem* class so that it has a **public** *good* method and then you can use *IList<MenuItem>* as a data representation for menus.

The bottom-half of figure 190 displays not one, but two designs of *MenuItem*, and as you can tell, they are distinct. The one on the left supports a *good* method that returns true if the item costs between $10 and $20; it is useful for cost-conscientious customers. In contrast, the class on the right defines a *good* method that identifies vegetarian menu items, i.e., those suited for a vegetarian visitor.

Unfortunately, it is impossible to combine the two classes in the given context. The list library has settled on the name *good* as the one criteria to be used when objects are selected from a list. If there is a need to use two *distinct* criteria to select sub-lists from one and the same list, you cannot use this library to achieve this goal.

You may recall that we have encountered an instance of this very same problem in the preceding chapters in the shape of the *sort* problem. Just like *select* in the library of figure 190 use a fixed *good* method to pick the appropriate items, the *sort* method uses a fixed *lessThan* method to arrange the items in the desired order. Sorting the same list according to two different criteria is impossible because it is impossible to abstract over methods.

36.3   Methods as Objects

If we had encountered the following two function definitions in the context of *How to Design Programs*,

```
(define (select1 l)                        (define (select2 l)
  (cond                                      (cond
    [(empty? l) empty]                         [(empty? l) empty]
    [else                                      [else
      (if ( hasGoodPrice  (first l))             (if ( hasNoMeat  (first l))
          (cons (first l) (select1 (rest l)))        (cons (first l) (select2 (rest l)))
          (select1 (rest l)))]))                     (select2 (rest l)))]))
```

we would have abstracted over them like this:

```
(define (select-abstract l  predicate )
  (cond
    [(empty? l) empty]
    [else
      (if ( predicate  (first l))
          (cons (first l) (select-abstract (rest l)  predicate ))
          (select-abstract (rest l)  predicate ))]))
```

That is, we would have introduced an additional parameter, passed it along to all recursive function calls, and used it where we used to call specific functions. In order to validate this abstraction, we would eventually have defined the original functions in terms of *select-abstract*:

```
(define (select1 l)                        (define (select2 l)
  (select-abstract l  hasGoodPrice ))         (select-abstract l  hasNoMeat ))
```

In general, we would have applied *select-abstract* to a list and a selection function (*hasGoodPrice*, *hasNoMeat*) instead of designing new functions.

  While methods aren't values or objects over which we can parameterize, let us imagine this possibility for just a moment. Given the pair of methods from figure 189:

```
public IMenu selectByPrice() {            public IMenu selectByMeat() {
 IMenu r = rest.selectByPrice();            IMenu r = rest.selectByMeat();
 if (first. hasGoodPrice ()) {              if (first. hasNoMeat ()) {
  return new Cons(first,r); }                return new Cons(first,r); }
 else {                                     else {
  return r; }                                return r; }
}                                          }
```

we could then equip the *select* method with an additional parameter:

    inside of *Cons* :
    // select a sublist of items for which *predicate* holds
    public *IMenu selectBy*(*IMethod predicate*) { … }

Like above, *predicate* is the name of a "method parameter," i.e., it stands for an object that acts like a method. The name of its type, *IMethod* suggests an interface that describes how to represent methods as objects. In this specific case, the *predicate* object represents methods that determine whether an instance of *MenuItem* has a certain property.

    Our next step must be to translate this requirement on *predicate* into a method signature for the *IMethod* interface. The word "determine" implies that if *IMethod* describes a method, it is a boolean-valued method. In turn, the phrase "some instances has a property" means the method consumes such an instance:

    interface *IMethod* {
      // determine whether *m* has **this** property
      boolean *good*(*MenuItem m*);
    }

Put differently, *IMethod* represents objects that support one method. This method consumes a *MenuItem* and produces a boolean. If you ponder this for a moment, it shouldn't surprise you that such objects represent methods. After all, all you can do with them is invoke exactly one method.

    Equipped with *IMethod* and an understanding how *predicate* represents a method, the next step follows logically, without a choice:

    inside of *Cons* :
    public *IMenu selectBy*(*IMethod predicate*) {
       IMenu r = *rest.selectBy*(*has*);
       if (*predicate.good*(*first*))
       …
    }

The test expression in gray invokes the method in *predicate* on the *first MenuItem*. The **if**-test then uses the result to continue as before.

    If we can actually implement the *IMethod* class, the new *selectBy* method works and works for general selection criteria. Since abstraction demands that we show how to get the old programs back, let's see how we could represent the *hasGoodPrice* method. To do so, we act as if we are designing a method using the design recipe for methods but we do design an implementation of *IMethod* with a specific purpose:

```
interface ISelect {
  // determine whether m has this property
  boolean good(MenuItem m);
}
```

```
interface IMenu {
  // select the items from this menu according to has
  IMenu selectByMeat(ISelect);
}
```

```
class Mt implements IMenu {
  // constructor omitted

  public IMenu selectBy(ISelect has) {
    return this;
  }
}
```

```
class Cons implements IMenu {
  private MenuItem first;
  private IMenu rest;
  // constructor omitted

  public IMenu selectBy(ISelect is) {
    IMenu r = rest.selectBy(is);
    if (is.good(first)) {
      return new Cons(first,r); }
    else {
      return r; }
  }
}
```

```
class HasGoodPrice
    implements ISelect {
  // is m's cost betw. $10 and $20
  public boolean good(MenuItem m) {
    return m.hasGoodPrice();
  }
}
```

```
class HasNoMeat
    implements ISelect {
  // is m item free of "Meat:"
  public boolean good(MenuItem m) {
    return m.hasNoMeat();
  }
}
```

Figure 191: A menu with two select methods

```
class HasGoodPrice implements IMethod {
  // determine whether m costs between $10 and $20
  public boolean good(MenuItem m) {

    ...
  }
}
```

The signature is dictated by the *IMethod* interface. The purpose statement states the obvious, namely, that *m* should have a "good" price. While we could go through examples and the rest of the design recipe, you can see that the method just invokes *hasGoodPrice* on *m*.

Figure 191 sums up the discussion. It displays a data representation of menus with a properly abstracted *select* method. That is, it includes only one *select* method though one that accepts an extra argument. The latter is a placeholder for objects that represent methods. It is this method-represented-as-an-object that determines whether *select* includes an item or not. The figure uses the name *ISelect* for the interface that describes the methods-as-objects representation. The two classes at the bottom show how to implement this interface: the instances of *HasGoodPrice* help select menu items that cost an appropriate amount and the instances of *Has-NoMeat* helps compile a vegetarian menu.

Before we move on, let's take a second look at these two classes and their methods. This look shows that the *good* methods in both *HasGood-Price* and *HasNoMeat* do not depend on the instance of *MenuItem* that they process. This instance is passed in explicitly, implying that the *good* methods really are like functions on *MenuItem*s. While these classes don't have fields, even if they did, these fields would be initialized when the "function" is created not when the *MenuItem* is created. The following exercises show how fields in such classes can be useful and thus reinforce that abstracting over traversals requires function-oriented reasoning in the manner of *How to Design Programs*.

Exercise

Exercise 36.2 For some diners, a price range of $10 to $20 could be too expensive, for others, it may sound cheap. Design a variant of *HasGoodPrice* that allows that customers to select a price range for the selection process. ∎

Exercise 36.3 Re-equip the menu representation in figure 191 with the *selectByPrice* and the *selectByMeat* methods of figure 189. Doing so ensures that the former is as powerful and as useful as the latter. ∎

Exercise 36.4 Use the methods-as-objects idea to abstract the *select* method in the list library of figure 190 (see section 36.2). We suggest you proceed as follows:

1. Design the class *Menu*, which represents a menu as information about today's date, the restaurant's name, and today's menu. Use lists for

the latter. Equip the class with methods for selecting a well priced menu from the existing menu.

2. Design the abstraction of the *select* method.

3. Re-design the selection method from item 1 so that it uses the abstraction from item 2. Also design a method for selecting a vegetarian sub-menu. ∎

36.4 List Traversals and Polymorphic Methods

Now that we know that representing methods as objects gives us "functions as first-class values" back, we can re-visit the general list traversals functions from *How to Design Programs*. If we can reconstruct those traversal functions in an object-oriented setting, we should be able to formulate a general traversal in the world of classes.

Let's start with the traversal function that is extremely useful as far as lists are concerned:

;; map *: [Listof X] (X → Y) [Listof Y]*

It consumes a function f and a list l of values, applies f to each item on l, and collects the results on a list again:

(map f (list $x_1 \ldots x_n$)) = (list (f x_1) $\ldots$ (f x_n))

Adding this function to our list library in Java should give us an idea of how easy it all is.

As always we start with a purpose statement and a method signature:

inside of *IList<ITEM>* :
// apply f to each item on **this** list; collect the results in new list
IList<ITEM> map(IFun<ITEM> f);

The purpose statement translates the terminology from *How to Design Programs* to the one used in this book. The signature clarifies two points:

1. f is of type *IFun<ITEM>*, which is an interface for the methods-as-objects we wish to use with in conjunction with *map*.

2. the result list has the same type as the given list.

Note how the result type simplifies the one from the Scheme version. The overall problem, though, is sufficiently complex and we return to this simplification issue later after solving the basic problem.

```
interface IFun<I> {
  // create new data from theItem
  I process(I theItem);
}
```

```
interface IList<I> {
  // apply f to each item on this list; collect in new list
  IList<I> map(IFun<I> f);
}
```

```
class Mt<I>
    implements IList<I> {
  // constructor omitted

  public IList<I> map(IFun<I> f) {
    return this;
  }
}
```

```
class Cons<I>
    implements IList<I> {
  private I first;
  private IList<I> rest;

  // constructor omitted

  public IList<I> map(IFun<I> f) {
    I fst = f.process(first);
    return new Cons<I>(fst,rest.map(f));
  }
}
```

```
class Add1
    implements IFun<Integer> {
  public Integer process(Integer i) {
    return i + 1;
  }
}
```

```
class Sub1
    implements IFun<Integer> {
  public Integer process(Integer i) {
    return i − 1;
  }
}
```

Figure 192: Mapping over lists

The *IFun* interface is similar to the one for the *select* method in the preceding subsection:

inside of *IFun<I>* :
I process(*I theItem*);

We know from the signature for *map* that its "processing function" consumes one list item at a time and produces another one. This knowledge

is translated here into a method specification: the input type is *I* and so is the output type. Since we can't foresee the kinds of applications that may need to use *map*, this type is a parameter of the interface definition. It is instantiated in *IList<ITEM>* with *ITEM*, the type of list items.

From *IFun<I>*, you can see that *map* invokes the *process* method from the given instance of *IFun<I>* on individual list items:

   f.process(anItem)

We should, however, not jump ahead but design *map* step by step.

The second method design step is to formulate examples that can be turned into tests. For *map* two easy sounding examples are ways to add 1 or to subtract 1 from all items on a list of integers.

For *map*, we first need lists of integers:

inside of *Examples* :
IList<Integer> mt = **new** *Mt<Integer>*();

IList<Integer> l1 =	*IList<Integer> r2* =
new *Cons<Integer>*(1,*mt*);	**new** *Cons<Integer>*(2,*mt*);
IList<Integer> l2 =	*IList<Integer> r3* =
new *Cons<Integer>*(2,*l1*);	**new** *Cons<Integer>*(3,*r2*);
IList<Integer> l3 =	*IList<Integer> r4* =
new *Cons<Integer>*(3,*l2*);	**new** *Cons<Integer>*(4,*r3*);

If you add 1 to all items on the list on the left, you get the list on the right. Conversely, if you subtract 1 from the lists on the right, you get the list on the left. In short, mapping something like the add1 function over the left list produces the right one and mapping a sub1 function over the right list produces the left one.

The other argument to *map* is an object that implements *IFun<ITEM>*. In order to add or subtract 1, the method must consume and produce an integer, which implies the following template:

  class *???* **implements** *IFun<Integer>* {
    // add (subtract, . . .) to the given integer *i*
    public *Integer process(Integer i)* { . . . }
  }

From here, the rest of the design is straightforward. See the bottom of figure 192 for the actual class definitions.

With sample lists and sample instances of *IFun<Integer>* in hand, you can now specify examples and tests for *map*:

inside of *Examples* :
checkExpect(*l3.map*(**new** *Add1*()),*r4*,"add 1 to all")
checkExpect(*r4.map*(**new** *Sub1*()),*l3*,"sub 1 to all")

They express in our testing framework what the preceding paragraphs say. informally.

Putting together the template for *map* follows proven procedures:

inside of *Mt<ITEM>* :	inside of *Cons<ITEM>* :
I map(IFun<ITEM> f) {	*I map(IFun<ITEM> f)* {
...	... *first* ... *rest.map(f)* ...
}	}

Once you have the template, defining the *map* method itself is as easy as designing a method in chapter II. The result of the coding step is on display in figure 192; take a good look, study it in depth, and solve the following exercises before you move on.

Exercises

Exercise 36.5 Design a data representation for the inventory of a grocery store, including at least a name (*String*), a producer (*String*), a price (a double), and an available quantity (int). Then equip the representation with two applicative methods:

1. *inflate*, which multiplies the price of each item with 1.1;

2. *remove*, which sets the available quantity of all items from a specific producer to 0.

Finally abstract over the two methods by designing a *map*-like method. ∎

Exercise 36.6 Abstract the classes *Add1* and *Sub1* in figure 192 into a single class that implements *IFun<Integer>*. ∎

Exercise 36.7 Design the following implementations of *IFun<ITEM>* (for an appropriate type *ITEM*):

1. *Not*, which when applied to an instance of *Boolean* negates it;

2. *Hello*, which when applied to an instance of *String* (referring to a name) prefixes it with "hello ";

3. *Root*, which when applied to a *Double* computes its square root.

Develop a test suite that uses these classes to *map* over lists of *Boolean*s, *String*s, and *Double*s, respectively. ∎

Exercise 36.8 Design the following implementations of *IFun<ITEM>* (for an appropriate type *ITEM*):

1. *And*, which when applied to an instance of *Boolean* computes the conjunction with some other, fixed but specifiable *Boolean*;

2. *Append*, which when applied to an instance of *String* prefixes it with some other, fixed but specifiable *String*;

3. *Multiply*, which when applied to an instance of *Double* multiplies it with some other, fixed but specifiable *Double*.

Develop a test suite that uses these classes to *map* over lists of *Boolean*s, *String*s, and *Double*s, respectively. ∎

The exercises demonstrate how useful the *map* method in figure 192 is. At the same time, they drive home the point that *map*, as currently available, always consumes a list of some type and produces one of the *same* type. Clearly, this form of traversal function is overly restrictive. As you may recall from *How to Design Programs* or from this book, list traversals of the kind that *map* represents are common, but usually they produce a list of a different type than the one given.

Consider the simplistic problem of determining for each integer on a list of integers whether it is positive (or not). That is, given a list of integers *loi*, you want an equally long list of booleans *lob* such that if the *i*th item on *loi* is positive then the *i*th item on *lob* is true. Processing a list of integers in this manner is clearly a *map*-style task but defining an appropriate method as an object fails:

```
class IsPositive implements IFun<Integer> { // ILL-TYPED CODE
  public Boolean process(Integer i) {
    return i > 0;
  }
}
```

As you can easily see, the signature of the *process* method in this class does *not* match the signature in *IFun<Integer>*, and therefore the type checker cannot approve the **implements** clause.

A close look at this example suggests a first and obvious change to the function interface of the list library:

> **interface** *IFun<I,RESULT>* {
>   // create new data from *theItem*
>   *RESULT process(I theItem)*;
> }

While the definition of figure 192 parameterizes functions only over the type of the list items, this revised definition parameterizes functions over both the type of the items as well as the type of the result. Hence, in this context, you can define the *IsPositive* class just fine: see the bottom right class in figure 193. Note, though, that it implements an *IFun* interface at *Integer*, the type of the list items, and *Boolean*, the type of the result.

So now you just need to check whether this definition of *IFun* works with the rest of the list library. Let's start with *IList*:

> inside of *IList<ITEM>* :
> *IList<R> map(IFun<ITEM,R> f)*;

Since the interface for functions requires two types, we provide *I* and *R*. The former, *I*, represents the type of list items; the latter, *R*, is the result type of the function. In other words, ==*IFun<I,R>* is the transliteration of ($X \rightarrow Y$) into Java's type notation==. From here, you can also conclude that the result type of *map* must be *IList<R>*, because *f* creates one instance of *R* from each instance of *I* and *map* collects those in a list.

Problem is that *R* comes out of nowhere, which you must find disturbing. If you were to submit this revised interface definition to Java's type checker as is, it would report two errors:

```
... cannot find symbol ... R
```

one per occurrence of *R* in the code. These error messages mean that *R* is an unbound identifier; since we want it to stand for some concrete type, it is an *unbound or undeclared type parameter*.

Thus far, we know only one place where type parameters are declared: to the immediate right of the name of a class or interface definition. You might therefore wonder whether *R* should be a parameter of *IList*:

> **interface** *IList<I,R>* {
>   *IList<R> map(IFun<I,R> f)*
> }

```
interface IFun<I,R> {
  // create new data from theItem
  R process(I theItem);
}
```

```
interface IList<I> {
  // apply each item to process in f, collect in new list
  <R> IList<R> map(IFun<I,R> f);
}
```

```
class Mt<I>
   implements IList<I> {
 // constructor omitted

 public <R>
  IList<R> map(IFun<I,R> f) {
    return new Mt<R>();
  }
}
```

```
class Cons<I>
   implements IList<I> {
 private I first;
 private IList<I> rest;
 // constructor omitted

 public <R>
  IList<R> map(IFun<I,R> f) {
    R fst = f.process(first);
    return new Cons<I>(fst,rest.map(f));
  }
}
```

```
class Add1
  implements
    IFun<Integer,Integer> {
 public Integer process(Integer i) {
  return i + 1;
 }
}
```

```
class IsPositive
  implements
    IFun<Integer,Boolean> {
  public Boolean process(Integer i) {
   return i > 0;
  }
}
```

Figure 193: Mapping over lists

Obviously this interface definition type checks. Declaring the type parameter in this manner poses a serious obstacle, however, as a quick thought experiment demonstrates. Specifically, consider a *Menu* class that uses *IList<I,R>* to encapsulate a field of menu items:

```
class Menu {
  IList<MenuItem,XYZ> items;
  . . .
}
```

The *XYZ* indicates the missing return type for *map* in *IList*. Putting a type here fixes the signature of *map* to

  IList<XYZ> map(IFun<MenuItem,XYZ> f)

once and for all. In other words, all uses of *map* within *Menu* must produce lists of *XYZ*s. This is, of course, impractical because the *Menu* class may have to use *map* at many different return types. Before you read on, come up with three examples of how *map* could be used with distinct return types for menus.

Our conclusion must be that the result type *R* for *map* functions cannot be bound for the entire interface. Instead, it must be instantiated differently for every use of *map*. In Java, we can express this independence of *map*'s result type from *IList*'s type parameters with the introduction of a map-specific type parameter:

  inside of *IList<ITEM>* :
  <R> IList<R> map(IFun<ITEM,R> f);

Syntactically, you do so by pre-fixing a method signature with a list of type parameters in angle brackets. Here a single type, *R*, suffices. Invocations of *map* implicitly specify *R* from the type of the *IFun* argument, and the Java type checker uses this type for the result list, too. Method signatures such as *map*'s are said to introduce POLYMORPHIC METHODS.

Figure 193 shows the class and interface definitions for a completely functional list library with a versatile abstraction of a *map*-style traversal. Other than the generalizations of *map*'s signature and the *IFun* interface, nothing has changed from figure 192. The bottom part displays two classes that implement the new *IFun* definition; take a good look and get some practice by solving the following exercises.

Exercises

Exercise 36.9 Recall the parametric *Pair* class from chapter V (page 461). In this context, design the following implementations of *IFun<I,R>* (for appropriate types *I* and *R*):

  1. *First*, which when applied to an instance of *Pair<Integer,Boolean>* returns the "left" part of the object;

2. *Distance*, which when applied to an instance of *Pair<Double,Double>* computes the distance of this "point" to the origin;

3. *And*, which when applied to an instance of *Pair<Boolean,Boolean>* computes the conjunction of the two paired-up boolean values.

Develop a test suite that uses these classes to *map* over lists of appropriate *Pair*s, using the list library from figure 193 without change. ∎

```
interface IList<I> {                          interface IMap<KEY,VALUE> {
  // f.process() each item, collect in new list   // add the entry [k,value] to this
  <R> IList<R> map(IFun<I,R> f);                   void add(KEY k, VALUE value);

  // select the items from this are f.good()       // does this contain a pair with k k?
  IList<I> select(IPredicate<I> f);                boolean contains(KEY k);

  // is x a member of this?                        // what value comes with k in this
  boolean member(I x);                             // ASSUMES: contains(k)
                                                   VALUE retrieve(KEY k);
  // the first item on this                      }
  I first();
}
```

Figure 194: Lists and maps

Exercise 36.10 An ASSOCIATION MAP is like a dictionary. Instead of associating words with explanations of their meanings, though, it associates arbitrary keys with values. After creating a map, you *add* key-value pairs; you may then check whether a given map associates a key with any value; and you may retrieved the value that a map associates with a key.

The right side of figure 194 displays an interface that has turned this informal description into a rigorous specification. design an association map library that implements this *IMap* interface. Start with composing a list library with the interface specified on the left in figure 194; use whatever pieces are available, but do develop a test suite. ∎

36.5   Example: *fold*

In addition to *map*-style traversals, you have also encountered traversals that process each item on a list and collect the results in a single number,

```
interface IListInt {
  // the sum of integers on this list
  int sum();
  // the product of integers on this list
  int pro();
}
```

```
class MtInt implements IListInt {
  public MtInt() { }

  public int sum() {
    return  0 ;
  }

  public int pro() {
    return  1 ;
  }
}
```

```
class ConsInt implements IListInt {
  private int first;
  private IListInt rest;

  public ConsInt(int first, IListInt rest) {
    this.first = first;
    this.rest = rest;
  }

  public int sum() {
    return first  +  rest.sum();
  }

  public int pro() {
    return first  *  rest.sum();
  }
}
```

Figure 195: Integer lists with addition and subtraction

boolean, string, etc. This is what *How to Design Programs* and functional programming call a *fold* operation, but it has also found its way into object-oriented programming languages from there.

Figure 195 shows two *fold*-style list traversals in a data representation of lists of integers. The first one is the *sum* method, which adds up all the numbers on the list. The second is the *pro* method, which computes the product of all the numbers. Both methods traverse the entire list and compute a single number by combining the list items with + or *. When *sum* reaches the end of the list, it produces 0; *pro*, in contrast, returns 1. Following the design recipe for abstraction, figure 195 highlights the two pairs of differences in gray.

Figure 196 displays a list-of-ints representation that abstracts over these two traversals, including the introduction of an **abstract** class to abstract

```
abstract
  class AListInt implements IListInt {
    // process each item of this list,
    // collect results with f; start with e
    abstract int arith(IOp f, int e);

    public int sum() {
      return arith(new Sum(), 0 );
    }

    public int pro() {
      return arith(new Pro(), 1 );
    }
  }
```

```
interface IOp {
  // a function that combines x and y
  int combine(int x, int y);
}
```

```
class MtInt extends AListInt {
  public MtInt() { }

  protected int arith(IOp f, int e) {
    return e;
  }
}
```

```
class ConsInt extends AListInt {
  private int first;
  private AListInt rest;

  public ConsInt(int first, IListInt rest) {
    this.first = first;
    this.rest = (AListInt)rest;
  }

  protected int arith(IOp f, int e) {
    return f.combine(first,rest.arith(f,e));
  }
}
```

```
class Sum implements IOp {
  public int combine(int x, int y) {
    return x + y;
  }
}
```

```
class Pro implements IOp {
  public int combine(int x, int y) {
    return x * y;
  }
}
```

Figure 196: Integer lists with *fold*

over the commonalities once the *sum* and *pro* methods are abstracted. Let's take one step at a time, starting from the differences.

Since the two traversals differ in two places, you should expect that the abstracted traversal functions consumes two arguments. Figure 196 confirms this with the gray line in the *AListInt* class at the top:

> inside of *AListInt* :
> **abstract** int *arith*(*IOp f*, int *e*);

The first parameter abstracts over the operation that the traversal uses to collect the results. The second parameter abstracts over the constant that the traversal uses when it encounters an empty list. The second one is also easy to explain. Both constants are ints in the original traversal, meaning that the new *arith* method just consumes the appropriate int constant.

The first parameter of the new *arith* method requires an explanation. Unlike in Scheme, where operations are functions, Java does not recognize operations as first-class values. Hence, it is once again necessary to encode something as objects; *IOp* is the interface that we use to specify the behavior of these objects.

Based on the signature of *arith* and its explanation, the design of the *arith* methods is straightforward. For *MtInt*, the method just returns *e*; for *ConsInt*, the method uses the "function" argument *f* to combine the *first* item on the list and the result of processing the *rest* of the list.

Once you have an abstraction of two traversals, you need to recover the original operations. The first step is to create method invocations of *arith* that simulate the traversals:

// *aList.sum*() is equivalent to:	// *aList.pro*() is equivalent to:
aList.arith(**new** *Sum*(),0)	*aList.arith*(**new** *Pro*(),1)

You can see the definitions of *Sum* and *Pro* at the bottom of figure 196. They implement *IOp* in the expected manner. Of course, the ideal abstraction process ends up restoring the two traversals and offering designers the option of defining more such methods. A moment's thought shows that doing so would leave you with two pairs of identical copies of *sum* and *pro*, respectively. Because you wish to avoid such replications, you would follow the design recipe of chapter III, which yields the abstract class in figure 196.

Exercises

Exercise 36.11 Develop a test suite for the list library of figure 195 and use it to test the library of figure 196, too. ∎

Exercise 36.12 Add the methods *isEmpty* and *first* to figure 196. The latter should assume that the list is not empty.

Use these methods and *arith* to design the method *max* (*min*), which finds the largest (smallest) item on a non-empty list of ints respectively. ∎

```
interface IFun<X,Z> {
  // create new data from i and e
  Z process(X i, Z e);
}
```

```
interface IList<I> {
  // f processes each list item and collects results
  // start with e if there are no items
  <Z> Z fold(IFun<I,Z> f, Z e);
}
```

```
class Mt<I> implements IList<I> {
  public Mt() { }

  public
   <Z> Z fold(IFun<I,Z> f, Z e) {
     return e;
   }
}
```

```
class Cons<I> implements IList<I> {
  private I first;
  private IList<I> rest;

  public Cons(I first, IList<I> rest) {
    this.first = first;
    this.rest = rest;
  }

  public
   <Z> Z fold(IFun<I,Z> f, Z e) {
     return f.process(first,rest.fold(f,e));
   }
}
```

Figure 197: Lists with *fold*

Exercise 36.13 In the context of figure 196, a list is a union of two variants. Add a *Range* variant for representing intervals of integers. See section 33.3 for the idea behind *Range*. Ensure that *sum* and *pro* "just work." ∎

You may realize from reading *How to Design Programs* that *arith* is a "fold" method, i.e., a method that folds together an entire list into a single value. Generalizing *arith* to *fold* is simple. First, you generalize the type of the list items from int to I, a type parameter:

inside of *AListInt<I>* :   inside of *IOp2<I>* :
int *arith(IOp2<I> f*, int *e*)   int *combine(I item*, int *y*)

That is, both *arith* and *combine* (from *IOp*) process *I*s not ints as inputs. The remaining occurrences of int in the code refer to the *result*s of *arith* and *combine*, respectively.

   Naturally, a generalization of *arith* shouldn't stop with a parameterization of the input type. It should also use a type parameter, say *Z*, to abstract over the result:

inside of *IListInt\<I\>* :   inside of *IFun\<I,Z\>* :
\<Z\> Z fold(IFun\<I,Z\> f, Z e)   int *combine*(*I item*, *Z y*)

Of course, the type parameter isn't a parameter of the list interface, but only of the *fold* method itself. Put differently, just like *map* from the preceding section, the *fold* method is parametric. The specification of the *combine* must change in analogous manner.

   This is all there is to the abstract *fold* method. For the full definition, see figure 197. As you can see from the figure, the abstraction of *arith* to *fold* is really just about type abstractions, not an abstraction of the mechanism per se. The method definitions remain the same; that is, the body of *fold* in *Cons\<I\>* is identical to the body of *fold* in *ConsInt* and the body of *fold* in *Mt\<I\>* is the same as the body of fold in *MtInt*.

Exercises

Exercise 36.14 The abstraction of *arith* to *fold* isn't complete without showing that you can redefine some of the motivating examples. Demonstrate that *fold* can be used to define *sum* and *pro* for lists of integers. ∎

Exercise 36.15 Design a data representation for teams. A single team has two attributes: a name and a list of members, represented via first names (*String*). Add a method that computes a collective greeting of the shape

```
"Dear Andy, Britney, Carl, Dolly, Emil:"
```

assuming the team members have the names `"Andy"`, `"Britney"`, `"Carl"`, `"Dolly"`, and `"Emil"`. ∎

Exercise 36.16 Design a class that represents a list of measurements as a *String* (for the source of the measurements) and a list of *Double*s, created with the list library of figure 197. Also design a method that determine whether all measurements are positive. ∎

Exercise 36.17 Design a class that represents a bill as a *String* (for the customer) and a list of pairs of type *Pair<String,Integer>* for the items sold and their cost. (See chapter V (page 461) for *Pair*.) Also design a method that computes the sum total of the billed items. ∎

36.6 Example: Arithmetic Expressions and Traversals

Lists are not the only data representations that can benefit from abstracted traversal methods. Indeed, all data representations can benefit though the cost (of designing the abstraction) is often higher than for the simple case of lists. Still, if the need for repetition shows up, it is almost always best to create the necessary abstraction.

In order to illustrate the principle of traversal abstraction in a general setting, let's look at the problem of representing and processing variable expressions as found in typical algebra books for middle schools: $x + 10$, $2 \cdot (x + 10)$, etc. For concreteness, here is a problem statement:

> ... Design a data representation for manipulating (algebraic) variable expressions. Equip it with methods for determining whether an expression is variable-free and for evaluating expressions that don't contain variables. ... Expect requests for methods that determine the "complexity" of an expression, replace variables with numbers, and so on. ...

While the problem doesn't spell out all the details, it constraints the setting in a reasonable manner.

Following the most basic design recipe, our first step is to focus on an informal description of the information:

A *variable expression* is either

1. a number; e.g., 5, 3.14, or -20;

2. a variable; e.g., x, y, or z;

3. or an addition expression, which combines two expressions via +; e.g., $x + 10$, $10 + y$, or $10 + 10 + 10$.

As you can see, this description narrows down the kinds of expressions that your program must deal with initially. It does, however, include the three

elements that are essential for the original problem statement: numbers, variables, and the combination of two expressions into one.

<mark>The second step is to choose interfaces and classes to represent this kind of information as data and to ensure that you can translate all the informal examples into the chosen data representation.</mark>[94] Figure 198 shows our representation. Not surprisingly, it consists of an interface and three implementing variants. The interface represents the type as a whole and descriptions of the methods that it supports. The classes represent the three data variants.

Exercise

Exercise 36.18 Translate the informal examples of variable expressions into data examples. How would you represent $10 + x + z$? Which of the two representations do you like best?

Interpret the following data in this context: **new** *Pls*(**new** *Var*(`"x"`,**new** *Val*(2))). ∎

Designing the required methods is straightforward from here. Let's use <mark>the examples</mark> from above to agree on behavioral examples:

Example	Variable Free?	Value
55	yes	55
$5 + x$	no	n/a
z	no	n/a
$3 + 2$	yes	5
$1 + 1 + 3$	yes	5

The table illustrates the meaning of "variable free" and "value of expression." It also makes it obvious why looking for the value of expressions with variables makes little sense; until you know the value of the variables, you can't determine the value of the expression. The only line that may surprise you is the third one, which deals with an expression that consists of just one item: a variable (z). Although this example is an unusual one, our methods must deal with it, and you should keep in mind that dealing with such examples at the level of information is the best we can do.

---

[94]You might wonder how a method could turn a textual representation of expressions, say `"x + z + 10"`, into the chosen data representation. This problem is not the one we are dealing with. A typical introductory book might give an ad hoc answer. The proper approach is to study the concept of "parsing" and its well-developed technology.

```
interface IExpression {
  // is this expression free of variables?
  boolean variableFree();
  // compute the value of this expression; ASSUMES: variableFree()
  int valueOf();
}
```

```
class Num implements IExpression {
  private int value;
  public Num(int value) {
    this.value = value;
  }

  public boolean variableFree() {
    return true;
  }

  public int valueOf() {
    return value;
  }
}
```

```
class Var implements IExpression {
  private String name;
  public Var(String name) {
    this.name = name;
  }

  public boolean variableFree() {
    return false;
  }

  public int valueOf() {
    throw new RuntimeException(...);
  }
}
```

```
class Pls implements IExpression {
  private IExpression left;
  private IExpression right;
  public Pls(IExpression left, IExpression right) {
    this.left = left;
    this.right = right;
  }

  public boolean variableFree() {
    return left.variableFree() && right.variableFree();
  }

  public int valueOf() {
    return left.valueOf() + right.valueOf();
  }
}
```

Figure 198: Expressions and traversals

Figure 198 displays the complete solution of the problem, for the restricted set of expressions. The interface contains the two obvious signatures and purpose statements. The methods in the implementing classes are straightforward, except that the "ASSUMES" part is expressed as a single test in the one relevant class instead of a test in each.

Exercises

Exercise 36.19 Use the examples from the above table to develop a test suite for the code in figure 198.

What do you think the results for $10 + z + 55$ and $10 + 20 + 30$ should be? Which of the possible data representation do you like best? Does it make a difference which one you choose? Why or why not? ∎

Exercise 36.20 Design an extension of the data representation of figure 198 that deals with multiplication expressions. ∎

Exercise 36.21 Compare the two methods, *variableFree* and *valueOf*, on a class-by-class basis. Highlight the differences. Can you abstract over the two traversals? ∎

The last exercise exposes the similarities between *variableFree* and *valueOf*. If you invoke either one of them, it traverses the tree of objects that represent the expression and processes one piece at a time. Since the analogous methods are located inside the same class, it is easy to compare them:

1. in *Num*, one method always returns true while the other one's result depends on the *value* field:

   ```
   public boolean variableFree() {              public int valueOf() {
     return true;                                 return value;
   }                                            }
   ```

2. in the *Var* class, the *variableFree* method naturally produces false and the *valueOf* method throws an exception:

   ```
   public boolean variableFree() {    public int valueOf() {
     return false;                      throw new RuntimeException(...);
   }                                  }
   ```

3. and in *Pls* both methods traverse the two sub-expressions and then combine the result:

```
public boolean variableFree() {          public int valueOf() {
  return left.variableFree()                return left.valueOf()
        &&                                        +
        right.variableFree();                     right.valueOf();
}                                         }
```

To do so, one uses logical conjunction and the other one uses addition.

Note: The *variableFree* method does not always traverse the entire sub-expression. Explain why.

Abstracting over these differences is a tall order. Unlike in the cases of *map* and *fold*, the differences between the methods come in many varieties. The difference between *variableFree* and *valueOf* in *Pls* looks like those we dealt with for *map* and *fold*. The difference between *variableFree* and *valueOf* in *Num*, however, looks quite different. While one method produces a constant, the other one actually refers to a local field.

Given these varieties of differences, let's step back and reflect on the nature of traversals for a moment. From *How to Design Programs*, we know that to traverse a data representation means to visit each object and to process it with some function. The *map* method illustrates this statement in a particularly elegant manner. It visits each item on the list and applies a "function," that is, a method as an object, to the item; eventually it collects all the results in a list. For *fold*, the abstraction includes a second argument, a constant for the empty list, and the collection process is specified by the caller of *fold*.

What we are seeing here is that *IExpression* is implemented by three different kinds of classes. Hence, an "expression" consists of many nested instances of *Pls* as well as *Num* and *Var*. Since all three kinds of objects are processed in a different manner, we should consider the idea of using three function parameters to abstract over a traversal:

```
inside of IExpression :
// process all instances of
// – Num in this IExpression with proNum
// – Var in this IExpression with proVar
// – Pls in this IExpression with proPls
??? traverse(INum proNum, IVar proVar, IPls proPls)
```

The names of these "functions" as well as the purpose statement of *traverse*
implies that they consume an instance of *Num*, *Var*, and *Pls* respectively
and produce an appropriate result. In other words, *traverse* is polymorphic
in its return type and each argument produces the same type as *traverse*:

> inside of *IExpression* :
> $\overline{\langle R \rangle\ R\ traverse(INum\langle R \rangle\ proNum,\ IVar\langle R \rangle\ proVar,\ IPls\langle R \rangle\ proPls)}$

Each of the three interfaces *INum*, *IVar*, and *IPls* specifies one function
using the methods-as-objects encoding:

interface *INum$\langle R \rangle$* {	**interface** *IVar$\langle R \rangle$* {	**interface** *IPls$\langle R \rangle$* {
R process(Num n)	*R process(Var n)*	*R process(Pls n)*
}	}	}

The type parameters of these interface definitions represent the respective
return types of course.

A natural alternative to abstracting with three "functions" is to abstract
with one object that provides three different methods, one per case:

inside of *IExpression* :	**interface** *IProc$\langle R \rangle$* {
$\overline{\langle R \rangle\ R\ traverse(IProc\langle R \rangle\ process)}$	*R processNum(Num n)*
	R processVar(Var v)
	R processPls(Pls p)
	}

The three methods in *IProc* exist because *traverse* visits three different kinds
of objects. In the course of doing so, it uses *process.processNum* to process
instances of *Num*, *process.processVar* to process instances of *Var*, and *process.processPls* to process instances of *Pls*.

This second way of abstracting over the three differences has an advantage over the first one, and it is arguably more object-oriented than the
first one. First, the advantage is that the three methods are bundled into
one object. It thus becomes impossible to accidentally invoke *traverse* on a
proNum object that doesn't match the intentions of *prcessVar*. Second, when
things belong together and should always be together, *How to Design Programs* and this book have always argued that they should reside in a single
object (structure). In this example, we are dealing with three "functions"
to which *traverse* should always be applied together. It is thus best to think
of them as one object with three methods. Or, more generally, think of an
object as a multi-entry function.[95]

---

[95]While object-oriented languages could introduce objects as generalizations of first-class
closures in the spirit of functional programming languages, they instead leave it to the programmer to encode the lexical context in an object manually.

```
interface IExpression {
  <R> R traverse(IProc<R> f);
}
```

```
interface IProc<R> {
  R processNum(Num n);
  R processVar(Var x);
  R processPls(Pls x);
}
```

```
class Num implements IExpression {
  private int value;
  public Num(int value) {
    this.value = value;
  }

  public <R> R traverse(IProc<R> f) {
    return f.processNum(this);
  }

  public int valueOf() {
    return value;
  }
}
```

```
class Var implements IExpression {
  private String name;
  public Var(String name) {
    this.name = name;
  }

  public <R> R traverse(IProc<R> f) {
    return f.processVar(this);
  }

  public String nameOf() {
    return name;
  }
}
```

```
class Pls implements IExpression {
  private IExpression left;
  private IExpression right;
  public Pls(IExpression left, IExpression right) {
    this.left = left;
    this.right = right;
  }

  public <R> R traverse(IProc<R> f) {
    return f.processPls(this);
  }

  public IExpression leftOf() { return left; }

  public IExpression rightOf() { return right; }
}
```

Figure 199: Expressions and abstract traversals

Now that we have proper signatures and purposes statements, it is possible to design the methods in a nearly straightforward fashion. We start with *traverse* and then deal with *process* methods. Let's look at the template for *traverse* in *Num*:

inside of *Num* :
<R> R traverse(IProc<R> p) {
  … **this**.*value* …
}

As you can see, the template reminds you that *traverse* can use **this**, the *value* field in **this**, and *p* to compute its result (plus auxiliary methods). Since the description of *p* says that its *processNum* method consumes an instance of *Num* (and processes it), you should consider the following definition obvious:

inside of *Num* :
<R> R traverse(IProc<R> p) {
  return *p.processNum*(**this**);
}

The design of the methods in *Var* and *Pls* proceeds in a similar manner and yields similar method definitions:

inside of *Var* :   inside of *Pls* :
<R> R traverse(IProc<R> p) {   *<R> R traverse(IProc<R> p)* {
  return *p.processVar*(**this**);   **return** *p.processPls*(**this**);
}   }

Indeed, with method overloading, you could make the three methods *identical*. We avoid overloading here so that the design remains useful in languages that don't support it.

Exercise

Exercise 36.22 If the *IProc* interface in figure 199 used overloading, the three *traverse* method definitions in figure 199 would be identical. The design recipe from chapter III would then call for abstracting them via the creation of a common superclass. Why would type checking fail for this abstraction? You may wish to re-read intermezzo 22.3 concerning overloading and its resolution. ∎

   With the design of *traverse* completed, you should turn your attention to
the final step of the abstraction recipe: the re-definition of the existing meth-
ods. Here we design only the implementations of *IProc<R>* that compute
whether an expression is variable free and, if so, what its value is.

   The return type of *variableFree* determines the type that we use to instan-
tiate *IProc<R>*. Here it is *Boolean* because the original method produced
a boolean result. Designing a class—call it *VariableFree*—that implements
IProc<Boolean> requires the design of three methods:

1. The *processNum* method consumes an instance of *Num* and deter-
   mines whether it is free of variables:

   inside of *VariableFree* :
   public *Boolean processNum*(*Num n*) {
     return true;
   }

   The method return true without looking at *n* because an expression
   that represents a number doesn't contain any variables. Of course,
   you could have just looked at the *variableFree* method in *Num* in fig-
   ure 198 and used its method body.

2. The *processVar* method consumes an instance of *Var* and determines
   whether it is free of variables:

   inside of *VariableFree* :
   public *Boolean processVar*(*Var v*) {
     return false;
   }

   The method return false without looking at *v* because an expression
   that represents a variable is guaranteed not to be variable-free.

3. Last but not least, the *processPls* method consumes an instance of *Pls*
   and must determine whether it contains a variable. Since an instance
   of *Pls* contains two expressions in two fields, the answer is clearly
   non-obvious. So we turn to our design recipe and write down the
   template:

   inside of *VariableFree* :
   public *Boolean processPls*(*Pls e*) {
     … **this** … *e.left* … *e.right* …
   }

We have explicitly included **this** in the template to remind you of its existence.

One way to finish the design is to inspect the original method, the one that you abstracted, and to adapt it to the re-definition. In this example, the original *variableFree* method traverses *e.left* and *e.right* to find out whether both of them are variable free. If so, it says that the entire *Pls* expression is variable free.

Adapting this purpose to the new context means to invoke *traverse* on *e.left* and *e.right*. The purpose of doing so is to determine whether the two sub-expressions are variable free. To invoke *traverse* properly, you need a "function" that inspects each possible kind of object for free variables. And that of course is **this**. After all, **this** represents a "function" as an object with three methods, and the function's purpose is to determine whether an expression is variable free.

In summary, the *processPls* method should look like this:

```
inside of VariableFree :
public Boolean processPls(Pls e) {
  return e.left.traverse(this) && e.right.traverse(this);
}
```

Unfortunately, the two fields, *left* and *right*, are **private** in *Pls*, which means that you can't just write *e.left* to get access to the current value of the field. The natural solution to this problem is to introduce a getter method into *Pls* as shown in section 33.2.

For a complete solution of the problem, see the left side of figure 200. The *VariableFree* class is an implementation of the *IProc<R>* interface.

The right side of figure 200 displays an implementation of *IProc<R>* for determining the value of a variable-free expression. Like *VariableFree*, *ValueOf* comes with three methods:

1. *processNum(Num n)* uses the *valueOf* method from *Num* to extract the integer that *n* represents;

2. *processVar(Var v)* throws an exception without inspecting *v*;

3. and *processPls(Pls e)* traverses the two sub-expressions using *traverse* and **this** and adds the results, just like the original *valueOf* method.

```
class VariableFree                          class ValueOf
  implements IProc<Boolean> {                 implements IProc<Integer> {
 public Boolean processNum(Num n) {          public Integer processNum(Num n) {
  return true;                                return n.valueOf();
 }                                           }

 public Boolean processVar(Var x) {          public Integer processVar(Var x) {
  return false;                               throw new RuntimeException("...");
 }                                           }

 public Boolean processPls(Pls x) {          public Integer processPls(Pls x) {
  return x.leftOf().traverse(this)            return x.leftOf().traverse(this)
         &&                                          +
         x.rightOf().traverse(this);                x.rightOf().traverse(this);
 }                                           }
}                                           }
```

Figure 200: Expressions and instantiated traversals

You can also see in the figure that *ValueOf* implements *IProc<Integer>*, meaning all three methods produce integers as a results.

One attention-drawing aspect of the two classes is the lack of any use of *v* in the *processVar* methods. Neither *VariableFree* nor *ValueOf* needs to know the name of the variable in *v*. This suggests that, at least in principle, the signature could be weakened to

inside of *IProc<R>* :
R processVar()

The exercises below demonstrate that while this change in signature is feasible for the two problems that motivate the abstraction of the traversal, other traversals exploit the full functionality of this interface. In addition, the exercises also explore a more conservative approach to the abstraction of *variableFree* and *valueOf*.

Exercises

Exercise 36.23 Adapt the test suite from exercise 36.19 to the data representation in figures 199 and 200. Ensure that the revised code passes the same tests. ∎

Exercise 36.24 The key to the abstraction of *variableFree* and *valueOf* is to hand the entire object from the *traverse* method to a method-as-object that implements *IProc<R>*. Doing so demands the addition of getter methods, which make field values widely accessible.

<mark>A more conservative approach</mark> than this is to hand over those values directly to the visitor. Here is an interface between such a *traverse* method and its *process* methods:

```
interface IProc<R> {
  R processNum(int n);
  R processVar(String x);
  R processPls(IExpression left, IExpression right);
}
```

Design a complete data representation with an abstract *traverse* method based on this interface. Design variants of *variableFree* and *valueOf* to ensure that the abstraction can implement the motivating examples; use the test suite from exercise 36.23 to ensure this point.

Challenge: Yet another approach to abstracting the two methods is to optimistically traverse sub-expressions. That is, the *traverse* method in *Pls* would process the *left* and *right* fields and hand the results to the visitor:

```
interface IProc<R> {
  R processNum(int n);
  R processVar(String x);
  R processPls(R left, R right);
}
```

Design a complete data representation and traversals for this variant.

While this second approach appears to work just like the first one or the on discussed in this section, there is an essential difference in how the *variableFree* method and an instance of (the adapted) *VariableFree* (class) process an expression. Explain the difference. Can this difference affect the workings of a program? ∎

Exercise 36.25 Design a traversal for *IExpression* from figure 199 that collects all the names of the variables that occur in the expression. ∎

Exercise 36.26 Design a traversal for *IExpression* from figure 199 that determines the maximal depth of an expression. The depth is a measure of how many *Pls* objects a traversal has to reach an instance of *Var* or *Num*. ∎

Exercise 36.27 Design a traversal for *IExpression* from figure 199 that substitutes all occurrences of variables with numbers. To this end, the traversal needs an association between variable names (*String*) and values (*Integer*). Hint: Use the *IMap* data representation from exercise 36.10 to represent such variable-value associations. ∎

37 Designing Abstract Traversal Methods

While the preceding section illustrates how to abstract over methods that traverse collections of objects, it also introduces a powerful and important idea: representing methods as objects. Indeed, without a mechanism for turning methods into first-class values, it would simply be impossible to parameterize traversals properly.

Hence this section first revisits the methods-as-objects idea in depth, with a look at how this works in the world of plain subtyping and a brief summary of how it works with generics. It also introduces two additional notational tools for creating methods as objects wherever and whenever needed: inner classes and anonymous implementations of **interface**s. Then we study the design principles for abstracting over method calls and for general data traversals.

37.1 Methods as Objects via Subtyping

Unlike objects, methods are not first-class values. While it is impossible to pass a method to a method as an argument, passing an object to a method is common. Similarly, while it is impossible for a method to produce a method as a result, objects are used as results all the time. Finally, our objects do contain objects in fields; you cannot do this with methods. Methods aren't values.

Indeed, in object-oriented programming languages objects are the only complex values that are first-class, and they come with behavior. After all, invoking a method from a first-class object is *the* way to compute in this world. Hence, it is natural to think of objects when it comes time to represent methods if we need programs to pass them around *as if* they were first-class values. Since first-class methods-as-objects are analogous to functions in *How to Design Programs*, they are called FUNCTION OBJECTS.

Let's start with the one-line problem that motivates it all. Your program contains an expression like this one

> ... *anObject.aMethod* (*anotherObject*, ...) ...

and you have determined that you need to abstract over which methods you want to call on *anObject*. Following our standard practice of using interfaces to specify requirements, here is a simple one and a sketch of its implementation:

```
// methods as objects                 class Function implements IFunction {
interface IFunction {                    public Object invoke() {
  Object invoke();                         return ... // behave like aMethod
}                                        }
                                       }
```

This interface demands that a function object have one method: *invoke*. In its most general form, this method is supposed to compute the same results as the desired method, e.g., *aMethod*.

Because the behavior of methods, such as *aMethod*, depends on the object on which they are invoked—*anObject* in the example—you can see from this sketchy implementation of *IFunction* that something is wrong. Because *Function* doesn't have fields and *invoke* takes no additional arguments, *invoke* has no access to *anObject*. Hence, it must always produce the same result or a randomly chosen result, neither of which is useful.

The obvious solution is to pass *anObject* as the first explicit argument to *invoke*. Put differently, an interface for function objects must demand that *invoke* consume at least one argument:

```
// methods as objects
interface IFunction {
  Object invoke(Object this0);
}
```

Here *this0* is the object on which the abstracted method used to be invoked and which was then passed implicitly. If it were possible to use **this** as a parameter name, it would be the most appropriate one; here we use *this0* on occasion to indicate this relationship.

The interface uses Object for the argument and result types, thus making the signature as general as possible. Of course, an implementing class would have to use casts to use *this0* at the appropriate type. Furthermore, it would have to use a more specific return type than Object or the site of the original method call would have to be modified to use a cast so that the result is typed appropriately for the context.

For example, if you wished to use *IFunction* to abstract in conjunction with the *map* method from figure 193, you would need casts like the following in the implementing classes:

```
// add 1 to an integer                // is the integer positive?
class Add1 implements IFunction {     class Positive implements IFunction {
  public Integer invoke(Object this0) {  public Boolean invoke(Object this0) {
    Integer first = (Integer)this0;        Integer first = (Integer)this0;
    return ... // an integer,              return ... // a boolean,
    // as in figure 193;                   // as in figure 193;
}                                     }
```

While the method representations use proper return types (*Integer*, *Boolean*), the type checker still calculates that *map* produces a list of Objects because it uses *IFunction* as the type for the method representations. Hence, the context that uses *map*'s result must use casts to convert them to lists of *Integer*s or lists of *Boolean*s.

This situation clearly isn't satisfying. In some special cases, it is possible to use specialized interfaces that describe the desired methods with just the right types:

```
interface IFunInt2Int {               interface IFunInt2Bool {
  Integer invoke(Integer this0);        Positive invoke(Integer this0);
}                                     }
```

In a reasonably complex context, such methods for general traversals, this approach doesn't work because the interfaces are too narrow and would immediately lead to code replication again.

Finally, your abstraction may demand the processing of additional arguments, the explicit arguments of method invocations. One way to abstract in this case is to introduce interfaces with the exact number of parameters needed, e.g., an abstraction over a call with two explicit arguments:

```
// methods as objects
interface IFunction3 {
  Object invoke(Object this0, Object arg1, Object arg2);
}
```

Another one is to collect all explicit arguments in a list and to pass along this list of objects:

```
// methods as objects
interface IFunctionN {
  Object invoke(Object this0, IListObject allArguments);
}
```

If you choose the latter, you have the most general interface for abstraction. The abstracted call passes the *i*th argument as the *i*th item on the list. It is

then up to the implementing method to check that the correct number of arguments was passed along and to select the arguments from *allArguments* as needed.

Again, to avoid casts, length checks, and other extraneous computations, it is best to specify interfaces with precise method signatures. For example, if you know that your *traverse* methods always consume and produce int lists, the following interface is the most precise specification:

> **interface** *IFunIntInt2Integer* {
>   *Integer invoke(Integer this0, Integer accumulator)*;
> }

In the ideal case, your methods-as-objects implementations then needs neither casts nor length checks for lists. The disadvantage of precise interfaces is that you narrow down the range of possible instantiations of your abstraction. You make them less useful than they could be. Once again, you can see that representing methods as objects via subtyping is feasible, the preceding section shows that the use of generics is superior.

Exercise

Exercise 37.1 Design the classes *Add3* and *Add3Again*, which implement *IFunctionN* and *IFun3*, respectively. They both are to represent methods as objects that consume exactly three integers and produce their sum.

Which interface would you use to produce the list of sums of a list of lists of integers? How would you use it? ∎

37.2   Methods as Objects via Generics

The first section of this chapter uses generics exclusively for abstracting over method calls. Indeed, this particular use of generics illustrates the power of generics like no other example.

With generics you can set up a single library of interfaces for functions, i.e., methods represented as objects, and use those interfaces whenever you wish to abstract over method calls. Figure 201 shows the first few interfaces in this library. As you can see, these interfaces are type-level abstractions of the Object-typed interfaces from the preceding section. Specifically, *IFun0* is an interface that specifies the existence of the method *invoke* that consumes a single argument; *IFun1* is about methods that consume two arguments; and *IFun2* specifies a method with three arguments. In all cases, the types

```
interface IFun0<RANGE,DOMAIN> {
  RANGE invoke(DOMAIN this0);
}
```

```
interface IFun1<RANGE,DOMAIN,DOMAIN1> {
  RANGE invoke(DOMAIN this0, DOMAIN1 arg1);
}
```

```
interface IFun2<RANGE,DOMAIN,DOMAIN1,DOMAIN2> {
  RANGE invoke(DOMAIN this0, DOMAIN1 arg1, DOMAIN2 arg2);
}
```

Figure 201: Generic function interfaces for all cases

of the arguments and the result type are parameters of the interface specification. Furthermore, if these interfaces are used to abstract over method calls, we assume that the formerly implicit argument, that is, the object on which the method is invoked, is passed as the first argument.

As you can guess from the arrangement in figure 201, the three interfaces represent just the beginning of a (n infinitely) long series. Since you can't predict the kind of functions you will need to represent, you need—at least in principle—signatures for all possible numbers of parameters. In practice, though, programmers do not—and should not—design methods with more than a handful of parameters. Thus, spelling out, say, 10 of these interfaces is enough.[96]

Exercise

Exercise 37.2 Define a generic interface that is analogous to *IFunctionN* at the end of section 37.1. ∎

37.3 Abstracting over Method Calls, Anonymous Inner Classes

The preceding two subsections lay the ground work for abstracting over method calls, i.e., the analog to functional abstraction from *How to Design Programs*. With "function" interfaces like those, it is possible to specify the type of objects that represent methods as first-class values. Although

---

[96]Scala, an object-oriented language derived from Java, provides just such a library. An alternative is to develop an abstraction mechanism that allows the specification of *all* such interfaces as instances of a single parameterized interface.

abstractions over method calls are particularly useful in conjunction with general data traversal methods, they also make sense by themselves.

As always the first step for abstracting method calls is to recognize the need for abstraction. Here we focus on the occurrence of two similar methods in the same class *C*:

inside of *C* :	inside of *C* :
Type method1(...) {	*Type method2*(...) {
...	...
o.methodA(...)	*o.methodB*(...)
...	...
}	}

In other words, we assume that class *C* contains two methods that are alike except for a method call. Even though such a coincidence is unlikely, considering this case is good practice for considerations of complex situations.

Further following the usual design recipe for abstraction, you need to equip *method1* and *method2* with an additional parameter so that the two method bodies become the same:

inside of *C* :	inside of *C* :
Type method1a($\boxed{\textit{IFun}\texttt{<...>}}$ *f*, ...) {	*Type method2a*($\boxed{\textit{IFun}\texttt{<...>}}$ *f*, ...) {
...	...
f.invoke(*o*,...)	*f.invoke*(*o*,...)
...	...
}	}

The key for this step is to pick a type that specifies the demands on the additional parameter and *IFun* from the preceding section serve just this purpose. Since the two methods are now basically identical, you may eliminate them in favor of a single method, dubbed *method*.

You must next show that you can re-create the original methods from this new abstraction. As figure 202 shows, doing so takes three classes:

1. You need to modify class *C* and equip it with a (**private**) *method* that performs the computations. It is abstracted over an object that represents a method. Hence, the original methods may just invoke *method* passing along appropriate methods represented as objects.

   To complete the re-creation of the original methods, we must address the question of how to create appropriate objects that implement the *IFun* interface. From what you know, the most direct way of doing so requires the design of two classes that implement *IFun* and instantiating those classes.

```
class C {                               class FunA implements IFun<...> {
                                          public ... invoke(...) {
  private Type method(IFun<...> f, ...) {    // a computation like methodA
  ...                                       }
    f.invoke(o,...);                      }
  ...
  }

  Type method1(...) {                   class FunB implements IFun<...> {
    return method( new FunA(...) ,...);    public ... invoke(...) {
  }                                         // a computation like methodB
                                          }
  Type method2(...) {                     }
    return method( new FunB(...) ,...);
  }
}
```

Figure 202: Abstracting over methods

2. Class *FunA* introduces a data representation for methods that compute like *methodA*.

3. Class *FunB* is a representation for methods that compute like *methodB*.

Obviously the notational cost of creating extra classes and instantiating them elsewhere is large. First, introducing three classes where there used to be one means you and future readers need to re-establish the connection. Second, it is far less convenient than creating first-class functions with **lambda** in Scheme (see *How to Design Programs*); so you know that there must be a simpler way than that.

Before we discuss these additional linguistic mechanisms, let us introduce the term COMMAND PATTERN, which is what regular programmers call methods-as-objects. The simplest explanation of a command pattern is that a program uses objects to represent computational actions, such as drawing a shape or changing the value of some field. Sophisticated programmers understand, however, that useful actions are parameterized over arguments and often produce values, which is why our objects really represent something akin to functions, also known as closures in functional programming languages such as Scheme.

37.4   Inner Classes, Anonymous Classes

```
class C {

  private Type method(IFun<...> f, ...) {
    ...
    f.invoke(o,...);
    ...
  }

  Type method1(...) {
    return method( new FunA(...) ,...);
  }

  Type method2(...) {
    return method( new FunB(...) ,...);
  }

  private
    class FunA implements IFun<...> {
      public ... invoke(...) {
        // a computation like methodA
      }
    }

  private
    class FunB implements IFun<...> {
      public ... invoke(...) {
        // a computation like methodB
      }
    }
}
```

```
class C {

  private Type method(IFun<...> f, ...) {
    ...
    f.invoke(o,...);
    ...
  }

  Type method1(...) {
    return
      method(
        new IFun<...>() {
          public ... invoke(...) {
            // a computation like methodA
          }
        })
  }

  Type method2(...) {
    return
      method(
        new IFun<...>() {
          public ... invoke(...) {
            // a computation like methodB
          }
        })
  }
}
```

Figure 203: Abstracting over methods

The notational overhead of creating separate classes explains why you will hardly ever encounter Java programs that abstract over a method call (in a simple method as opposed to a traversal method). In order to manage this overhead, the designers of Java added two linguistic mechanisms:

1. inner classes, which are class definitions that occur within a class def-

inition;

2. and anonymous implementations of interfaces.

In this section, we briefly study these mechanisms in general. They are relatively simple and easy to use, which we do in later sections of this chapter.

The left column in figure 203 is a reformulation of the classes in figure 202 as a single class. Nested within the class definition are two more class definitions; see the framed boxes. A comparison of figure 203 with figure 202 shows that these nested class definitions are those we introduced to re-create *method1* and *method2*. Since these NESTED CLASSES are used only for this purpose, they are intimately linked to class *C*, which is what the textual nesting expresses and signals to any future reader. The figure also shows that, just like methods and fields, classes come with privacy attributes; we label classes used for command patterns with **private**.

While the nesting of the two auxiliary classes expresses their intimate connection to class *C*, it does not reduce the textual overhead. For this purpose, Java supports a mechanism for directly instantiating an interface without first defining a class. In general, the direct instantiation of an interface has the following shape:

> **new** *IInterface* () {
> . . .
> }

where the . . . define those methods that *IInterface* specifies. Naturally, if *IInterface* is parameterized over types, you need to apply it to the correct number of types, too: *IIinterface<Type1, ...>*. In either case, it is unnecessary to name a new class, which is why people speak of ANONYMOUS INSTANTIATION.

Using the direct instantiation of interfaces, you can complete the last step of the design recipe in a reasonably compact manner: see the right column of figure 203. It shows that you can formulate the two methods-as-objects exactly where you need them. No future reader of this class must search for auxiliary class definitions—global or nested—just to find out what the *invoke* methods compute. Instead, the code comes right along with the re-definitions of the original methods.

The mechanism for instantiating interfaces directly facilitates the use of this pattern but it is not essential. Indeed, the designers of many object-oriented programming languages have added constructs for creating first-class functions directly because the instantiation of interfaces is still cumbersome. Because this book isn't about Java (alone), we use anonymous

instantiation, mostly in section 38 in preparation of chapter VII. If you end up working with Java, though, read up on the details of anonymous and inner classes and learn to use them properly.

37.5 <mark>Visitor Traversals and Designing Visitors</mark>

When you find a data representation with two similar traversals, you have an opportunity to create a powerful abstraction. If you anticipate any future extensions or revisions involving traversals, or if you anticipate turning the data representation into a library—especially one that others should use and can't modify—you should create a general traversal method.

As the preceding chapter shows, extending a library and adding traversal methods is a complex undertaking. Sometimes you, the library designer, can't even allow that.[97] Once you add an all-powerful traversal method, though, it becomes easy to use your library because your "clients" can always create a specialized traversal from the general traversal method.

The creation and use of a general, all-powerful traversal method combines a fixed set of methods with the very first design recipe you encountered. Professional programmers know the arrangement as the VISITOR PATTERN. The pattern assumes that you have a self-referential data definition that provides a common interface *I* to the world and has several implementing variants: *C1*, *C2*, ... In this context, you can add a general traversal method in three mandatory steps plus one optional one:

1. add a generic signature that specifies a general traversal method to the common interface *I*:

   <u>inside of *I* :</u>
   <RESULT> RESULT traverse(IVisitor<RESULT> v)

   The signature specifies that each implementing variant comes with a parametric method that consumes a visitor object *v*. The result type *RESULT* of the traversal process is declared with the method signature and handed to the generic visitor interface.

   Naturally, if your chosen language is a version of Java that does not support generics or an object-oriented language without type abstractions, you use subtyping to make the visitor as general as possible.

2. add a generic interface *IVisitor* to the library:

---

[97]Remember to look up **final** in a Java description.

```
interface IVisitor<RESULT> {
  RESULT visitC1(C1 x);
  RESULT visitC2(C2 x);
  . . .
}
```

The interface specifies one *visit* method per class that implements *I*.

If your chosen programming language supports overloading, you may wish to specify an appropriate number of overloaded methods:

```
interface IVisitor<RESULT> {
  RESULT visit(C1 x);
  RESULT visit(C2 x);
  . . .
}
```

For clarity, we forgo Java's overloading here.

3. to each variant *C1*, *C2*, . . . , add the following implementation of the visit method:

inside of *C1* :	inside of *C2* :	. . .
public	**public**	. . .
<R>	*<R>*	. . .
R traverse(IVisitor<R> x) {	*R traverse(IVisitor<R> x)* {	. . .
return *x.visitC1(***this***)*;	**return** *x.visitC2(***this***)*;	. . .
}	}	. . .

Note: If you use overloading, the *visit* methods in all variants look identical. Since Java's type checker must resolve overloaded method calls before evaluation, however, it is impossible to apply the design recipe of lifting those methods to a common superclass.

4. to each class *C1*, *C2*, . . . , you may also have to add getters for **private** fields so that the *visit* methods may gain access as appropriate.

Equipping the data representation with such a *traverse* method ensures that it invokes a *visit* method on each object that it reaches during a traversal. If the *visit* method chooses to resume the traversal via a call to *traverse* for fields with self-referential types, the *traverse* method resumes its computation and ensures that *visit* is invoked on the additional objects. In short, the terminology of "visitor pattern" is justified because the *traverse* method

```
interface IList<I> {                    interface IVisitor<I,R> {
  <R> R traverse(IVisitor<I,R> f);        R visitMt(Mt<I> o);
}                                         R visitCons(Cons<I> o);
                                        }
```

```
class Mt<I> implements IList<I> {       class Cons<I> implements IList<I> {
                                          private I first;
                                          private IList<I> rest;
  // constructor omitted                  // constructor omitted

  public                                  public
    <R> R traverse(IVisitor<I,R> f) {       <R> R traverse(IVisitor<I,R> f) {
      return f.visitMt(this);                 return f.visitCons(this);
  }                                       }

                                          public I getFirst() { return first; }
                                          public IList<I> getRest() { return rest; }
}                                       }
```

<div align="center">Figure 204: Lists and visitors</div>

empowers others to specify just how many objects in a collection of inter-connected objects must be visited and processed.

Figure 204 illustrates the result of adding a visitor-based traversal to our conventional list representation. The top left is the list interface equipped with a *traverse* signature; the top right shows the interface for a list visitor. The implementing classes—at the bottom of the figure—contain only those methods that the above items demand.

Although the creation of the general traversal method itself doesn't require a design recipe, using the method does. Specifically, when you design an implementation of the *IVisitor* interface—which we call a VISITOR—you are after all designing a class-based representation of a method, which we also call a function object. Thus, using the visitor abstraction is all about formulating a purpose for methods and functions, illustrating it with examples, designing a template, filling in the gaps, and testing the examples.

Let's enumerate the steps abstractly and examine them in the context of the concrete list example:

1. The data definitions are already implemented; this step is of no concern to you when you are designing visitors.

2. Thus, the real first step is to formulate a contract and a purpose state-

ment. For visitors, you formulate it for the entire class, thinking of it (and its instances) as a function that traverses the data. In the same vein, the contract isn't a method signature; it is an instantiation of the *IVisitor* interface at some type, requiring at a minimum the specification of a result type. Still, instantiating the interface determines the function's contract, what it consumes and what it produces.

Example: Suppose you are to design a list traversal for integer lists that adds 1 to each item on a list of integers. Describing the purpose of such a function is straightforward:

```
// a function that adds 1 to each number on a list of integers
class Add1 implements IListVisitor<Integer,IList<Integer>> {
  public Add1() { }
  public IList<Integer> visitMt(Mt<Integer> o) { … }
  public IList<Integer> visitCons(Cons<Integer> o) { … }
}
```

The signature is the instantiation of *IListVisitor* with a list item type (*Integer*) and a result type (*IList<Integer>*).

Let's generalize the example a bit to "adds or subtracts a fixed integer to the items on the list." In the context of *How to Design Programs*, a purpose statement and header for this problem would look like this:

```
;; addN : Number [Listof Number] → [Listof Number]
;; add n to each number on a list of numbers
(define (addN n alon) …)
```

Note how the purpose statement refers to the parameter *n*, the number that is to be added to each list item.

Since the methods in the visitor interface are about one and only one kind of argument—the pieces of the data structure that is to be traversed—there appears to be no room for specifying any extra parameters, such as *n*. At the same time, you know that of the two arguments for *addN*, *n* is fixed throughout the traversal and *alon* is the piece of data that *addN* must traverse. Hence we can turn *n* into a field of the class:

```
class AddN implements IListVisitor<Integer,IList<Integer>> {
  private int n;
  public AddN(int n) {
    this.n = n;
  }
  ...
}
```

The field is initialized via the constructor, meaning that each instance of *AddN* is a function that adds a (potentially different) number to each integer on the traversed list.

In the world of *How to Design Programs*, this separation of parameters corresponds to the creation of closures via curried, higher-order functions:

```
;; addN-curried : Number → ([Listof Number] → [Listof Number])
;; add n to each number on a list of numbers
(define (addN-curried n)
  (local ((define (addN alon) ...))
    addN))
```

3. Next you determine functional, also known as behavioral, examples:

   Example:

   inside of *Examples* :
   IList<Integer> mt = **new** *Mt<Integer>*();
   IList<Integer> l1 = **new** *Cons<Integer>*(1,*mt*);
   IList<Integer> l2 = **new** *Cons<Integer>*(2,*l1*);
   IList<Integer> l3 = **new** *Cons<Integer>*(3,*l2*);
   IList<Integer> r2 = **new** *Cons<Integer>*(2,*mt*);
   IList<Integer> r3 = **new** *Cons<Integer>*(3,*r2*);
   IList<Integer> r4 = **new** *Cons<Integer>*(4,*r3*);

   checkExpect(*l3.traverse*(**new** *AddN*(1)),*r4*,"add 1 to all");

   This simple example shows how to create a function that adds 1 to each integer on the list; see the gray box. Invoking *traverse* on a list of integers with this function-as-object should yield a list where every integer is increased by 1.

4. The templates for the *visit* methods is about the fields in the objects on which the method is invoked (plus the fields of the visitor). To this end you must know the fields in the classes of the data representation; if fields are **private**, you must use the getter methods to access them. For all fields in the visited object that refer back to the interface *I*, i.e., for all self-referential classes in the data representation, add

  o.getField().*traverse*(**this**)

to the method body, assuming *o* is the parameter of the *visit* method. This expression reminds you that the method may recur through the rest of the data representation if needed.

This last point distinguishes a template for *visit* methods from a regular method template. While the two kinds of template are similar in principle, the emphasis here is on the object that is being visited, not the visitor object. The complete traversal is accomplished via an indirect call to *traverse* (with **this** instance of the visiting class) not via a recursive call to *visit*.

While you will use **this** for the recursive use of a visitor in most cases, you will need different traversals to complete the processing on some occasions. For an example, see below as well as the finger exercises in the next two subsections.

Example:

inside of *AddN* :	inside of *AddN* :
IList<Integer>	*IList<Integer>*
visitMt(Mt<Integer> o) {	*visitCons(Cons<Integer> o)* {
... *n* ...	... *n* ...
}	... *o.getFirst*() ...
	... *o.getRest*().*traverse*(**this**) ...
	}

For lists you have just one class that refers back to the list representation (*IList*), so it is not surprising that only *visitCons* invokes *traverse*.

5. Now it is time to define the *visit* methods. As always, if you are stuck, remind yourself what the various expressions in the template compute—using the purpose statement if needed—and then find the proper way to combine these values.

Example: Let's start with the reminder. The expression *n* in *visitMt* reminds you of the int constant that comes with *AddN*; the parameter *o* is the empty list of integers. The expressions *o.getFirst*() and *o.getRest*() access the first item on the non-empty list *o* and the rest of the list. Then, *o.getRest*().*traverse*(**this**) creates the list that is like the rest with *n* added to each integer, because this is what the purpose statement tells us. With these reminders, defining the two methods is easy:

inside of *AddN* :	inside of *AddN* :
IList<Integer>	*IList<Integer>*
visitMt(*Mt<Integer>* o) {	*visitCons*(*Cons<Integer>* o) {
return *o*;	int *a* = *n* + *o.getFirst*();
}	*IList<Integer>* r =
	o.getRest().*traverse*(**this**);
	return new *Cons<Integer>*(a,r);
	}

6. After you have finished the design of the visitor implementation, it is time to test the examples.

   Example: Turn the above examples into proper tests and run them. Can you use *AddN* to subtract 1 from a list of integers?

Let's look at another example, a visitor that checks whether all integers on some list are positive:

1. The purpose statement is given; the interface instantiation specifies that the "function" consumes a list of *Integer*s and produces a *Boolean*:

   ```
   // are all integers on the list greater than 0?
   class Positive implements IListVisitor<Integer,Boolean> {
     public Positive() {}
     public Boolean visitMt(Mt<Integer> o) { … }
     public Boolean visitCons(Cons<Integer> o) { … }
   }
   ```

2. Here are two examples, one for each kind of outcome:

   inside of *Examples* :
   IList<Integer> mt = **new** *Mt<Integer>*();
   IList<Integer> m1 = **new** *Cons<Integer>*(1,mt);
   IList<Integer> m2 = **new** *Cons<Integer>*(2,m1);
   IList<Integer> m3 = **new** *Cons<Integer>*(−3,m2);
   Positive isPositive = **new** *Positive*();

> *checkExpect*(*m3.traverse*(*isPositive*),false);
> *checkExpect*(*m2.traverse*(*isPositive*),true);

Since *m3* starts with −3, you should expect that the visitor can produce false without looking at the rest of the list, *m2*. In particular, because *m2* contains only positive integers—as the second test shows—traversing it contributes nothing to the final result of checking *m3*.

3. Given that instances of *Positive* consume the same kind of data as *AddN* functions, we just adapt the template from the first example:

<table>
<tr><td>inside of Positive :
Boolean
  visitMt(Mt<Integer> o) {

   . . .

}</td><td>inside of Positive :
Boolean
  visitCons(Cons<Integer> o) {
   . . . o.getFirst() . . .
   . . . o.getRest().traverse(this) . . .
}</td></tr>
</table>

4. The final method definitions look almost identical to the method definitions that you would have designed without *traverse* around:

<table>
<tr><td>inside of Positive :
public Boolean
  visitMt(Mt<Integer> o) {
   return true;
}</td><td>inside of Positive :
public Boolean
  visitCons(Cons<Integer> o) {
   return (o.getFirst() > 0) &&
              (o.getRest().traverse(this));
}</td></tr>
</table>

In particular, the *visitCons* method first checks whether the first item on the list is positive and *traverse*s the rest only after it determines that this is so. Conversely, if the invocation of *getFirst* produces a negative integer, the traversal stops right now and here.

5. Complete the code and run the tests.

What this second example shows is that the design of a visitor according to our design recipe preserves traversal steps. That is, the visitor steps through the collection of objects in the same manner as a directly designed method would. The abstraction works well.

The third example recalls one of the introductory exercises from *How to Design Programs* on accumulator-style programming:

```
interface IList {
  // convert this list of relative
  // distances to a list of absolutes
  IList relativeToAbsolute();
}
```

```
abstract class AList implements IList {
  public IList relativeToAbsolute() {
    return relAux(0);
  }

  abstract
    protected IList relAux(int soFar);
}
```

```
class Mt extends AList {
  public Mt() { }

  protected IList relAux(int soFar) {
    return this;
  }
}
```

```
class Cons extends AList {
  private int first;
  private AList rest;
  public Cons(int first, IList rest) {
    this.first = first;
    this.rest = (AList)rest;
  }

  protected IList relAux(int soFar) {
    int tmp = first+soFar;
    return new Cons(tmp,rest.relAux(tmp));
  }
}
```

Figure 205: Relative to absolute distances

> … Design a data representation for sequences of relative dis-
> tances, i.e., distances measured between a point and its prede-
> cessor, and sequences of absolute distances, i.e., the distance of
> a point to the first one in a series. Then design a method for
> converting a sequence of relative distances into a sequence of
> absolute distances. For simplicity, assume distances are mea-
> sure with integers. …

Figure 205 displays a complete solution for this problem. It uses a plain list
representation for both kinds of lists. The design is properly abstracted, lo-
cating the main method in an abstract class and pairing it with an auxiliary
(and **protected**) method based on accumulators.

While a solution like the above is acceptable after the first three chapters
of this book, a proper solution re-uses an existing list library and provides
the functionality as a visitor:

1. The purpose statement is the one from figure 205 and the class signa-

ture is just like the one from *AddN*:

```
// convert this list of relative distances to a list of absolutes
class RelativeToAbsolute
  implements IListVisitor<Integer,IList<Integer>> {
    public RelativeToAbsolute() { }
    public IList<Integer> visitMt(Mt<Integer> m) { … }
    public IList<Integer> visitCons(Cons<Integer> c) { … }
}
```

2. If $(3, 2, 7)$ is a list of relative distances between three points (and the origin), then $(3, 5, 12)$ is the list of absolute distances of the three points from the origin:

inside of *Examples* :
IList<Integer> mt = **new** *Mt<Integer>*();
IList<Integer> l1 = **new** *Cons<Integer>*(7,*mt*);
IList<Integer> l2 = **new** *Cons<Integer>*(2,*l1*);
IList<Integer> l3 = **new** *Cons<Integer>*(3,*l2*);

IList<Integer> r2 = **new** *Cons<Integer>*(12,*mt*);
IList<Integer> r3 = **new** *Cons<Integer>*(5,*r2*);
IList<Integer> r4 = **new** *Cons<Integer>*(3,*r3*);
RelativeToAbsolute translate = **new** *RelativeToAbsolute*();

checkExpect(*l3.traverse*(*translate*),*r4*,"relative to absolute");

3. Since the signature of *RelativeToAbsolute* is the same as the one for *AddN*, the template remains the same.

4. From there, the definitions of the methods should fall out but we fail, just like in *How to Design Programs*:

inside of *RelativeToAbsolute* :
IList<Integer>
  visitMt(*Mt<Integer> o*) {
    return this;
}

inside of *RelativeToAbsolute* :
IList<Integer>
  visitCons(*Cons<Integer> o*) {
    return
      new *Cons<Integer>*(
        o.getFirst()+ ??? ,
        o.getRest().*traverse*(???));
}

The method in *Cons* should add the first distance to those that pre-
ceded it in the original list, and it should pass along the first distance
to the computations concerning the rest of the list. Put differently, the
function should accumulate the distance computed so far but doesn't.

Note: The method could also use *AddN* repeatedly to add the first dis-
tance of any sublist to the distances in the rest of the list. This solution
is, however, convoluted and needs too much time to evaluate.

Here is a second start:

1. Since the class is to represent an accumulator-style function, it comes
   with an additional, constructor-initialized field:

   ```
   class RelativeToAbsolute
       implements IListVisitor<Integer,IList<Integer>> {
     int dsf; // distance so far
     RelativeToAbsolute(int dsf) {
       this.dsf = dsf;
     }
     public IList<Integer> visitMt(Mt<Integer> m) { … }
     public IList<Integer> visitCons(Cons<Integer> c) { … }
   }
   ```

   Ideally this *dsf* field should be 0 initially but we ignore this detail
   for the moment and assume instances are always created with **new**
   RelativeToAbsolute(0).

2. The revised template is like the one for *AddN* except for the recursion
   in *Cons*:

   ```
   inside of RelativeToAbsolute :
   IList<Integer>
     visitCons(Cons<Integer> o) {
       … o.getFirst() …
       … o.getRest().traverse( new RelativeToAbsolute(…) ) …
   }
   ```

   Instead of just using **this**, we indicate that a new instance of the class,
   distinct from **this**, is possibly needed.

3. With this revision, working definitions are in plain sight:

inside of *RelativeToAbsolute* :
IList<Integer>
  visitCons(Cons<Integer> o) {
    int *tmp* = *dsf* + *c.getFirst*();
    RelativeToAbsolute processRest = **new** *RelativeToAbsolute(tmp)*;
    return new *Cons<Integer>(tmp,c.getRest().traverse(processRest))*;
  }

4. To complete the first draft, it remains to test the visitor with the examples from above.

What also remains is to revise the draft class so that it doesn't expose a constructor that may initialize the field to the wrong value.

Exercise

Exercise 37.3 Use privacy specifications and overloading to revise the first draft of *RelativeToAbsolute* so that it becomes impossible to accidentally misuse it. ∎

Finally, an implementation of *IVisitor* doesn't have to use concrete types; it may use type variables, just like plain methods that traverse a piece of data. Suppose you wish to design an implementation of *IListVisitor* that acts like *map* from figure 192:

inside of *IList<I>* :
<RESULT> IList<RESULT> map(IFun<I,RESULT> f);

To do so, you must design a class whose result type is *IList<RESULT>* and whose *visit* methods process items of type *ITEM*:

class *Map<I,RESULT>* **implements** ...

Using the fresh type parameters, you can describe what kind of visitor the instances of *Map* implement:

// process each item, collect in new list
class *Map<I,RESULT>*
  implements *IListVisitor<ITEM,IList<RESULT>>* {
  ...
    public *IList<RESULT> visitMt(Mt<I> m)* { ... }
    public *IList<RESULT> visitCons(Cons<I> c)* { ... }
  }

Of course, these type choices are also reflected in the signatures for the *visit* methods; they consume lists of *ITEM*s and produce lists of *RESULT*s. The rest of this example is an exercise.

Exercises

Exercise 37.4 Reformulate the design instructions for visitor classes in a world with just subtyping, i.e., without generics. Then formulate a design recipe for visitors. ∎

Exercise 37.5 Demonstrate the workings of your design recipe from exercise 37.4 with the definition of a general visitor pattern for list classes and designs for the two visitor classes from this section: *AddN* and *Positive*. ∎

37.6   Finger Exercises: Visiting Lists, Trees, Sets

Exercise 37.6 Design the following visitors for the list library of figure 204:

1. *Sum*, which computes the sum of a list of *Integer*s;

2. *Contains*, which determines whether a list of *String*s contains an item that is equal to some given *String*;

3. *ContainsAll*, which determines whether a list of *String*s *l1* is contained in some given list *l2*, that is, all *String*s on *l1* are also on *l2*.

4. *Append*, which creates the juxtaposition of two lists. ∎

Exercise 37.7 Design the following generic visitors for the list library of figure 204:

1. *Map*, which consumes a method, represented as an object, plus a list. It produces the list that results from applying the given method to each item on the list and collecting the results.

2. *Fold*, which mimics the behavior of the *fold* method in figure 197.

3. *Filter*, which consumes a predicate *P*, i.e., an object representation of a boolean-producing method . It extracts those items *i* from the list for which *P.invoke*(*i*) returns true.

4. *Sort*, which sorts a list of arbitrary objects. Parameterize the "function" over the comparison used; represent the comparison method as an object that consumes two list items and produces a boolean. Define a single class though add nested classes as needed. ∎

Exercise 37.8 Figures 23 and 24 (see pages 45 and 45) introduce a data representation for the representation of geometric shapes.

Equip this data representation with an abstract traversal method using the visitor pattern.

Design *In*, a visitor that determines whether or not some position is inside some given geometric shape.

Design *Area*, which computes the area of some given geometric shape.

Design *BB*, which constructs a representation of a bounding box for the given shape. See section 15.3 for the definition of a bounding box. ∎
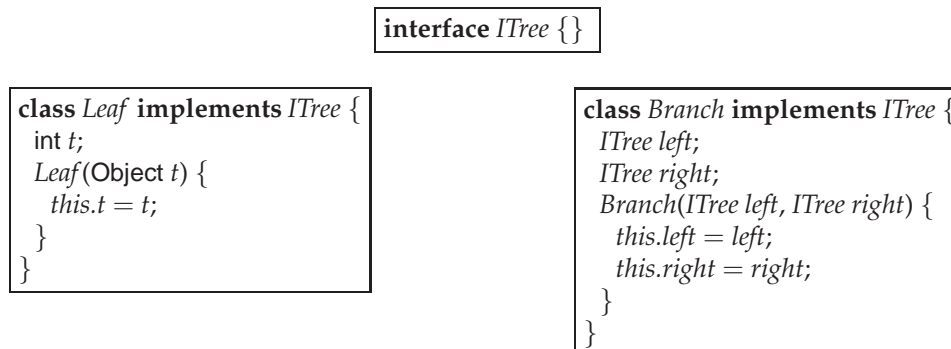
---

interface *ITree* {}

```
class Leaf implements ITree {
  int t;
  Leaf (Object t) {
    this.t = t;
  }
}
```

```
class Branch implements ITree {
  ITree left;
  ITree right;
  Branch(ITree left, ITree right) {
    this.left = left;
    this.right = right;
  }
}
```

Figure 206: Visiting trees

Exercise 37.9 Figure 206 displays the class definitions for representing binary trees of integers. Equip this binary tree representation with an abstract traversal, using the visitor pattern.

Design the visitor *Contains*, which determines whether a tree contains some given integer.

Design *Sum*, which visits a binary tree and determines the sum of integers in the tree. ∎

Exercise 37.10 Generalize the data representation of figure 206 so that it represents binary trees of objects instead of just ints. Equip it with a visitor

traversal and design the visitor class *Contains* for this binary tree representation (see exercise 37.9). Do not use generics for this exercise. ∎

Exercise 37.11 Section 32.4 introduces two distinct data representations of sets, one using lists (exercise 32.8) and another one using binary search trees (exercise 32.14). Furthermore, the two exercises request the design of generalizations using generics as well as subtyping, meaning you have four different ways of representing sets.

Equip your favorite set representation with a visitor-based traversal method so that you can design visitors that process all elements of a set. To test your traversal method, design three visitors. The first one adds 1 to each element of a set of integers, producing a distinct set of integers. The second one uses the **draw** package to draw the elements onto a canvas. The third one maps the elements of a set of integers to a set of corresponding *String*s. ∎

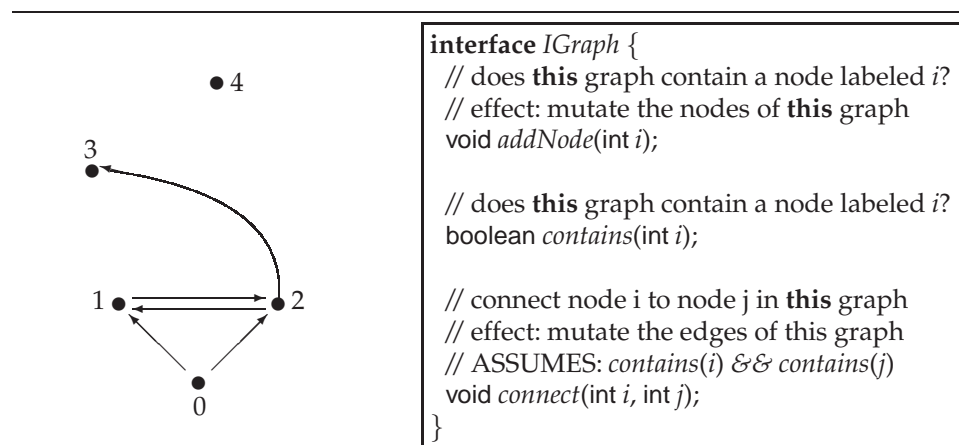## 37.7   Extended Exercise: Graphs and Visitors

Figure 207: Graphs and visitors

Equipping data representations with visitor-based traversal methods and designing visitors is also useful for collections of objects with cyclic connections. Take a look at the left side of figure 207. It displays an example of a graph. Concretely speaking, a graph is a collection of *nodes*, often labeled with distinct markers (ints for us), and *edges*, i.e., connections going from one node to another. A connection from node *n* to node *k* represents

the fact that you can "move" from *n* to *k* in the world of information. If, in this world, you can also move from *k* to *n*, you need to add an edge to the graph that points from *k* to *n*.[98]

General speaking, a graph is an abstract rendering of many forms of real-world information. For example, the nodes could be intersections of streets in a city and the edges would be the one-way lanes between them. Or, a node may stand in for a person and an edge for the fact that one person knows about another person. Or, a node may represent a web page and edges may signal that one web page refers to another.

As for a data representation of graphs, you should expect it to be cyclic. Nothing in our description prevents nodes from referring to each other; indeed, the description encourages bi-directional connections, which automatically create cycles. The *IGraph* interface on the right side of figure 207 follows from the design recipe of chapter IV. It suggests constructing basic data first, which means the collection of nodes for graphs, and providing a method for establishing connections afterwards, which for us means a method for adding edges. Thus the graph library provides the interface and a plain graph constructor:

inside of *Graph* **implements** *IGraph* :
public *Graph*() { }

Once you have a graph, you can specify the collection of nodes with the *addNode* method and the collection of edges with the *connect* method.

Exercises

Exercise 37.12 Turn the information example from figure 207 into a data example, based on the informal description of the library. ∎

Exercise 37.13 Design an implementation of the specified library. Use a generic list library, equipped with a visitor-based traversal, to represent the collection of nodes and the neighbors of each node. ∎

Constraint: Do not modify the list library for the following exercises. Design a visitor for use with the list traversal method instead.

Exercises

[98]We are dealing here with *directed* graphs. In so-called *undirected* graphs, every edge indicates a two-way connection.

Exercise 37.14 Design a method for the graph library from exercise 37.13 that given the (integer) label of some node in a graph, retrieves the labels of its neighbors. ▪

Exercise 37.15 Design the interface *IGraphNode*, which represents nodes for visitors. Then equip the graph library with a traverse method for processing the collection of nodes in a graph. Finally, design a visitor that computes the labels of all the nodes in a graph. ▪

Exercise 37.16 Design an implementation of the graph visitor interface to discover the sinks and sources of a graph.

A node *i* in a graph is a *sink* if there is no edge going from *i* to any other node. In figure 207, the node labeled 3 is a sink.

A node *o* in a graph is a *source* if there is no edge from any other node to *o*. In figure 207, the node labeled 0 is a source.

The node labeled with 4 is both a source and a sink. ▪

As you may recall from *How to Design Programs*, a node *i* in a graph is *reachable* from a node *o* if $i = o$ or if there is a neighbor *r* of *o* from which *i* is reachable. Thus, in figure 207 node 3 is reachable from node 0, because node 2 is a neighbor of node 0 and node 3 is reachable from node 2. The latter is true because node 3 is a neighbor of node 2. Note the recursive nature of the description of the reachability process!

One way to compute the set of reachable nodes for any given node is to start with its neighbors, to add the neighbors of the neighbors, the neighbors of those, and so forth until doing so doesn't add any more nodes. Proceeding in this manner is often called computing the *transitive closure* (of the neighbor relation) for the node.

Exercises

Exercise 37.17 Design the method *reachable*. Given a node *n* (label) in a graph, the method computes the list of nodes reachable from *n*. ▪

Exercise 37.18 Design a graph visitor that computes for every node the list of all reachable nodes. Use the solution of exercise 37.17. ▪

A mathematician's description of graphs doesn't usually use lists but sets. The former emphasizes the existing of an sequential arrangement

among objects and allows for repetitions; the latter assumes no ordering and allows no repetition of elements. Thus, when we suggested the use of lists for the collection of nodes in the graph or for the collection of neighbors per node, we committed to an ordering where none seems needed.

Exercise

Exercise 37.19 Inspect all the method signatures in *IGraph* and *IGraphNode*, including the ones you have added. For all occurrences of *IList<T>* decide whether it makes sense to replace it with *ISet<T>*. In other words, determine whether repetition of items matters and whether the sequencing of the item matters.

Re-design the methods using the interface from section 32.4. First use the set library based on lists, then use the one based on binary trees. Do you have to change any of your code when you switch? ∎

37.8 Where to Use Traversals, or Aggregation

At this point, you know how to add a general traversal to a data representation and how to design a visitor that uses this traversal. You have also practiced this skill with a few problems. For each of those, you defined an *Examples* class and tested the visitor within this example class. The question remains, however, where traversals are used, meaning where visitor classes are located and instantiated.

```
+--------------+         +--------+
| UFOWorld     |-------->| IShots |<----------+
+--------------+         +--------+           |
| IShots shots |         +--------+           |
+--------------+            |                 |            +--------------+
                           / \                |            | UFOWorld     |
                           ---                |            +--------------+    * +------+
                            |                 |            | ?????  shots |----->| Shot |
                  -------------------         |            +--------------+      +------+
                  |                 |         |                                  +------+
             +--------+    +-------------+ |                                       +------+
             | MtShot |    | ConsShot    | |
             +--------+    +-------------+ |      +------+
             +--------+    | Shot first  |----->| Shot |
                          | IShots rest |-+    +------+
                          +-------------+      +------+
```
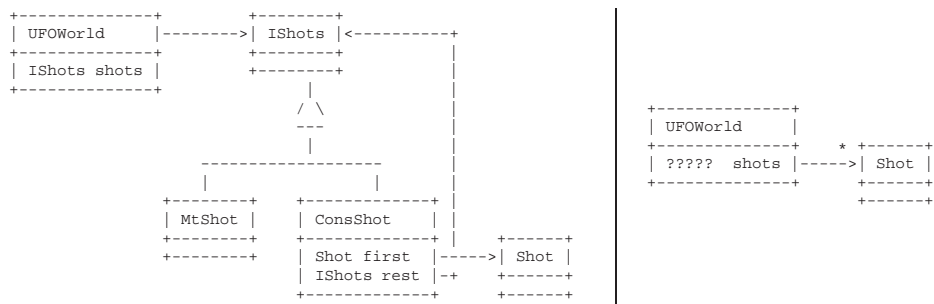
Figure 208: An aggregation diagram

The answer to these question is closely related to a concept known as AGGREGATION. Roughly speaking, an aggregation is a relationship between one class and a collection of others Thus far, when this situation

came up, we spelled out the details of an aggregation, usually as a list of things, sometimes as a tree of items, as a graph of nodes, etc. Examples are *UFOWorld* and its association with many *Shot*s, *Worm* and *WormSegment*s, *Library* and *Book*s, and so on. Since aggregation shows up so often, it has its own special arrow in class diagrams.

Figure 208, left side, reminds you of our usual representation of an association between *UFOWorld* and the many *Shot*s that have been fired. On the right, you see how to represent this situation as an "aggregation scenario." The containment arrow comes with an asterisk, which indicates that the class at *UFOWorld* is associated with a collection of *Shot*s.
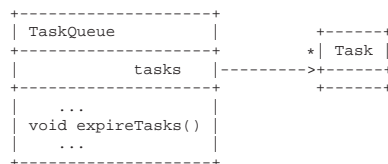
The aggregation arrow is more than just a convenience. First, it expresses that a *UFOWorld* comes with many *Shot*s *but* it doesn't spell out whether it is a list, a search tree, a set, or something else. In this regard, it delays the decision of how to represent the collection until we define classes, and it even allows you to change the representation choice later. Second, it also informs a reader that the methods of *UFOWorld* have to deal with entire collections of *Shot*s at once and thus implies that those methods are formulated as traversals. More generally, classes that refer to aggregates are where traversals are used, and they are also where you place/find visitor classes.

Let's look at some examples, starting with a task queue:

> ... A task queue keeps track of tasks that have been assigned to an employee. Customers or other employees enter tasks into the queue. The employee works on the tasks in a first-come, first-served order. Some of the tasks come with expiration dates, however, and overdue tasks are filtered out on a regular basis.
> ...

Even the first sentence in this problem statement suggests that a task queue comes with a collection of tasks

Here is a translation into a class diagram:

```
+--------------------+
| TaskQueue          |                      +------+
+--------------------+                    *| Task |
|            tasks   |--------->+------+
+--------------------+                      +------+
|    ...             |
| void expireTasks() |
|    ...             |
+--------------------+
```

A second look at the problem suggests that the method for dropping expired tasks must traverse the collection of *Task*s. We have therefore added a signature for such a task to the box for *TaskQueue*.

For now assume that, like in section 32.4, queues contain a list of *Task*s, represented via the list library from figure 204:

```
class TaskQueue {
  public TaskQueue() { }
  private IList<Task> tasks = new Mt<Task>();
  private int howMany = 0;

  ...

  // effect: remove those tasks from this queue that are expired
  public void expireTasks() {
    ... tasks ... howMany ...
    tasks ...
    howMany ...
    return ;
  }
}
```

The class fragment also spells out the first three steps of the design recipe for the *expireTasks* method: its purpose and effect statement, its signature, and its template. The latter reminds you that the method may use the values of the two (**private**) fields and that it can change them.

Skipping the examples step for now, we turn to the method definition step. Given that the *TaskQueue* refers to its collection of tasks via a list, the method should traverse this list, eliminate the expired tasks, and retain the remaining ones. Describing in this manner suggests the design of a visitor for the list library:

```
inside of TaskQueue :
public void expireTasks() {
  tasks = tasks.traverse(new ByDate());
  howMany = ...
}

// select those tasks from the list that haven't expired yet
private class Expired implements IListVisitor<Task,IList<Task>> {
  public Expired() { }
  public IList<Task> visitMt(Mt<Task> this0) { ... }
  public IList<Task> visitCons(Cons<Task> this0) { ... }
}
```

Following the advice from section 37.4, this auxiliary class is nested and hidden with *TaskQueue* because its methods are the only one that use it.

As you can see from the purpose statement of the visitor, it determines for each *Task* object on the list whether it is expired and retains those that aren't. Since "expired" typically means with respect to some time, the *Expired* class should probably grab the current data and/or time:
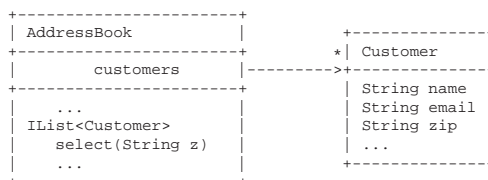
```
// select those tasks from the list that haven't expired yet
class Expired implements IListVisitor<Task,IList<Task>> {
  private Date today = ...
  private Time now = .
  public Expired() { }..
  public IList<Task> visitMt(Mt<Task> this0) { ... }
  public IList<Task> visitCons(Cons<Task> this0) { ... }
}
```

We leave it to you to complete the design from here. The point is that you have seen that the traversal is used in *TaskQueue* and that therefore the visitor class is nested with *TaskQueue*.

Now take a look at this second problem:

> ... You are to design an electronic address book for a small company. On request, the check-out clerks enter a customer's name, email, and zip code into the address book; the cash register adds what the customers have bought. The company uses this address book for customer assistance (returns, information about purchased items) but also for sending coupons for select stores to customers. ...

Here the problem implies that an address book is an aggregation of customer records:

```
+-----------------------+
| AddressBook           |                 +--------------+
+-----------------------+              *| Customer      |
|        customers       |-------->+--------------+
+-----------------------+              | String name   |
|    ...                 |              | String email  |
| IList<Customer>       |              | String zip    |
|   select(String z)    |              | ...           |
|    ...                 |              +--------------+
+-----------------------+
```

How the *AddressBook* class is associated with its collection of *Customer*s remains unspecified. The desire to look up individual customers quickly—while they wait—implies that a binary search tree is a better choice than a plain list. No matter which choice you make, however, you do need a method for *select*ing the customers with some specific zip code so that the company can mail coupons to specific regions. Just as before this *select*

method traverses the entire collection of *Customer*s and retains those with the given zip code attribute. Thus, *select* uses *traverse* and, to this end, you must place a visitor class in *AddressBook*.

Exercises

Exercise 37.20 Design an *AddressBook*, including the *select* method, using the list library from figure 204. Also add a method *addCustomers*, which consumes a list of *Customer*s and adds them to the *AddressBook*. ∎

Exercise 37.21 Design an *AddressBook*, including the *select* method, using the general binary search tree representation that you designed for exercise 32.13 (page 507). Start by adding a *traverse* method to the library. Don't forget to add a method *addCustomers*, which consumes a list of *Customer*s and adds them to the *AddressBook*. ∎

An aggregation doesn't have to involve collections such as lists or trees; it may re-use a class that represents a collection. For example, the preceding section ended with the suggestion to use a *Set* class instead of the *IList* library to associate a graph with a collection of nodes.

37.9 Object-Oriented and Functional Programming

Now is a good time to step back and reflect on the nature of general traversals and the design of visitors. Let's start with the structure of traversal computations and then re-visit the design question.

Imagine a self-referential data representation that consists of an interface *I* and several implementing classes (*Co*, *Cp*, ...). Also assume a regular recursive method for processing objects that implements *I*. In this context every (plain or recursive) invocation of *m* on some instance *o* of *Co*, uses the method definition of *m* in *Co* to compute the result. This process is dubbed polymorphic method dispatch (see section 13.1) and is considered the essence of object-oriented computing.

Contrast this scenario with one where the data representation comes with a visitor-based traversal method. In order to process an object now, your program creates an instance *v* of a visitor—a method represented as an object—and evaluates *o.traverse(v)*. While the invocation of *traverse* proceeds via polymorphic method dispatch, it immediately invokes a *visit*

method in *v*. People tend to use the terminology DOUBLE DISPATCH[99] for this step. Within the chosen *visit* method in *v*, computation proceeds as if *m* had been called on *o* directly (except for invocations of getter methods) until a recursive call is needed to resume the traversal. At this point, another double dispatch moves the evaluation from *v* to *o* and back again. In short, all essential computations take places in the methods of the visitor; the *traverse* and *visit* methods exist only for navigating the maze of objects.

How to Design Programs	Visitors
data definition	pre-existing *traverse*
function purpose & contract	class purpose & interface signature
functional examples	functional examples
template: conditional structure	*visit* methods in interface
template: conditions	*visit* per implementing class
template: layout	access methods & calls to *traverse*
coding: start with base cases	coding: start with base cases
coding: connecting expressions	coding: connecting expressions
testing examples	testing examples

Figure 209: Visitors and functions

Because of this separation of activities, the traversal methods within the data representations—its interface and its classes—are schematic while the methods in the visitor look like those usually found in data representations. Indeed, the differences concern just two parts: recursion and access to fields. The reason is that the design process for visitors is basically the one for functions in *How to Design Programs*. Figure 209 shows is a table that compares the two design recipes. You can see from the table that the visitor design has the advantage over the function design that the visitor interface (*IVisitor*) dictates how many "cond" lines you must consider and the polymorphic dispatch (via *traverse*) to the appropriate *visit* method eliminates the need to formulate conditions. Other than that, designing a visitor is just like designing a function.

What all this really means is that the visitor pattern re-introduces functional programming into the world of class and library design. The very moment you wish to abstract over traversals, which is common, you design almost as if you are designing functions and your visitors compute

---

[99]We also encountered double dispatch in our first discussion of extensional equality (section 21).

almost as if they were functions (though they use polymorphic dispatch heavily). All the case separation and all the action is located in the visitor class and its instances.

38   Traversing with Effects

The preceding sections of this chapter use few stateful and imperative examples, and none of the traversals use an imperative method. Indeed, if you were to try to use *traverse* with imperative methods, you would have to instantiate the *IVisitor* signature with void as the result type and doing so would fail.[100] The illusion that we provide of void being a regular type with one (invisible) value is just that, an illusion. Thus, if you need imperative traversals—and you will!— you must design them specially.

This section starts with an explanation of the *forEach* method, which performs a computation "action" on each item of a list, without any results. It then demonstrates how to use this method and how to abuse it. The latter is important because Java actually forces you to do something like that, as the next chapter explains.

38.1   Abstracting over Imperative Traversals: the *forEach* Method

Following our convention, we create abstractions from concrete examples, and for the design of *forEach*, we look to the "war of the worlds" project for inspiration. It comes with numerous needs for processing entire lists of objects: moving lists of shots, drawing lists of shots, moving lists of charges, drawing lists of charges, and a few more. In the imperative setting of chapter IV (section 27.7), all of these list traversals have the goal of changing the state of the objects.

Recall that the *UFOWorld* class contains a *move* method whose task it is to move all objects for every tick event:

---

[100]You may wonder whether there is *Void* related to void like *Integer* is related to int. While Java comes with a class *Void*, its role is different than *Integer*'s, and it has no use here.

```
// imperative methods as objects          // list with imperative traversals
interface IAction<I> {                    interface IList<I> {
  // invoke the represented method           // invoke f.invoke on each item
  void invoke(I c);                          void forEach(IAction<I> f);
}                                         }
```

```
class Mt<I> implements IList<I> {         class Cons<I> implements IList<I> {
  public Mt() { }                           private I first;
                                            private IList<I> rest;
  public void forEach(IAction<I> f) {
    return ;                                public Cons(I first, IList<I> rest) {
  }                                           this.first = first;
}                                             this.rest = rest;
                                            }

                                            public void forEach(IAction<I> f) {
                                              f.invoke(first);
                                              rest.forEach(f);
                                              return ;
                                            }
                                          }
```

Figure 210: The *forEach* method

```
inside of UFOWorld :
// move all objects in this world
public void move() {
  ufo.move();
  aup.move();
  shots.move();
  charges.move();
  return ;
}
```

For simplicity, this *move* method assumes that the methods it invokes consume no additional arguments.

The method definitions for moving the *Shot*s on a list look like this:

inside of *MtShot* :
void *move*() {
  return ;
}

inside of *ConsShot* :
void *move*() {
  first.move();
  rest.move();
  return ;
}

The method on the left performs no computation at all; the method on the right invokes the *move* method on the *first* instance of *Shot* and recurs on the *rest* of the list.

    If you were to look back at other imperative methods that process lists, you would find that this arrangement is quite common, which is why you want to abstract it. Roughly speaking, the abstraction is just like the *map* method with two exceptions. First, it uses methods-as-objects that are imperative and have return type void. Second, the results of processing the *first* item and traversing the *rest* of the list are combined by sequencing the two effects, throwing away the results of the first computation.

    Figure 210 displays the complete design of a general and imperative traversal method for lists. The method is dubbed *forEach*, implying that it performs some action for each item on the list. Otherwise the design has the expected elements:

1. The *IAction* interface specifies the shape of methods as objects that the *forEach* method consumes.

2. The *IList* interface includes the signature for the *forEach* method, indicating that *forEach* consumes an *IAction<I>* whose second argument has type *I*, the type of a list item.

3. The two implementations of *IList* define *forEach* in the manner discussed. Specifically, the method in *Mt* performs no action; the method in *Cons* invokes the action on *first* and then recurs on *rest*.

In short, *forEach* really is closely related to *map* and less so to *traverse*. Before you proceed with your readings, you may wish to consider in which situations you would rather have a *traverse* style method.

38.2   Using the *forEach* Method

When you have an abstraction, you need to demonstrate that it subsumes the motivating examples. That is, we should design actions for moving and

drawing objects from the *UFOWorld* class, because it aggregates the *Shot*s and other collections of objects.

   Let's start with an action for moving all the shots *Shot*. Keep in mind that the instances of the class represent methods that are invoked on every *Shot* on the list, one at a time:

1. The purpose statement is just that of the *move* method in *Shot*:

   ```
   inside of UFOWorld :
   // move the shots on this list
   private class Move implements IAction<Shot> {
     public void invoke(Shot s) { … }
   }
   ```

2. Just like for the design of any method, you need examples. In the case of imperative methods you need behavioral examples. That is of course also true if you represent methods as objects:

   ```
   inside of Examples :
   IList<Shot> mt = new Mt<Shot>();


   IList<Shot> l1 = new Cons<Shot>(s1,mt);
   IList<Shot> l2 = new Cons<Shot>(s2,l1);
   IList<Shot> l3 = new Cons<Shot>(s3,l2);


   IList<Shot> m1 = new Cons<Shot>(t1,mt);
   IList<Shot> m2 = new Cons<Shot>(t2,m1);
   IList<Shot> m3 = new Cons<Shot>(t3,m2);


   l3.forEach(new Move()) ;
   … checkExpect(l3,m3) …
   ```

   The gray box highlights the invocation of *forEach* on *l3* using an instance of *Move*. Below the box, a *checkExpect* expression formulates our expectation that *forEach* changes *l3*.

3. Like the template for all imperative methods, the one for *invoke* suggests that the method may use its arguments (**this**, *s*) and the fields of its invocation object (none):

```
inside of Move :
public void invoke(Shot s) {
  ... s ...
  return ;
}
```

4. Filling in the template leaves us with the expected invocation of *move* on the given shot:

```
inside of Move :
public void invoke(Shot s) {
  s.move();
  return ;
}
```

5. Finally it's time to run the tests.

With *Move* in place, you can now move the shots in *UFOWorld* via an invocation of *forEach*:

inside of *UFOWorld* :

```
public void move() {
  ufo.move();
  aup.move();
  shots.forEach(new Move());
  charges.move();
  return ;
}
```

inside of *UFOWorld* :
```
private IAction<Shot> mS = new Move();
public void move() {
  ufo.move();
  aup.move();
  shots.forEach(mS);
  charges.move();
  return ;
}
```

On the left you see the most direct and notationally most concise way of doing so. On the right you see the best version from a computationally perspective. Specifically, while the version on the left creates one instance of *Move* per tick event, the version on the right instantiates *Move* only once for the entire world. Although it is unlikely that a player or viewer can tell the difference, it is important for you to begin to appreciate this difference.

For a second design example, consider the case of drawing a list of *Shot*s. Remember that *draw* in *UFOWorld* is called to refresh *theCanvas*, which comes with every *World*:

<u>inside of *UFOWorld*</u> :
```
// draw all objects in this world onto theCanvas
public void draw() {
  drawBackground();
  . . .
  shots.draw(theCanvas);
  . . .
}
```

The code snippet ignores that the *draw* methods actually consume the *World* so that they can find out its dimensions. From here, we proceed as before:

1. The purpose statement and the contract are straightforward again:

    <u>inside of *UFOWorld*</u> :
    ```
    // draw a shot to the given canvas
    private class Draw implements IAction<Shot> {
      public void invoke(Shot s) { ... }
    }
    ```

    Like *Move*, *Draw* implements *IAction<Shot>*. Its *invoke* method processes one shot at a time. We know, however, that *draw* methods always consume a *Canvas* into which they draw a shape. As with *AddN* use a field to store the chosen *Canvas* throughout the entire traversal:

    <u>inside of *UFOWorld*</u> :
    ```
    // draw a shot to the given canvas
    private class Draw implements IAction<Shot> {
      private Canvas can;
      public Draw(Canvas can) {
        this.can = can;
      public void invoke(Shot s) { ... }
      }
    ```

2. The template for *invoke* lists both a parameter and a field:

    <u>inside of *Draw*</u> :
    ```
    public void invoke(Shot s) {
      ... s ... this.can ...
      return ;
    }
    ```

3. Filling in the template leaves us with the expected method body:

<u>inside of *Draw* :</u>
```
public void invoke(Shot s) {
  s.draw(this.can);
  return ;
}
```

The method invokes *draw* on *s* and passes along the *Canvas*.

You can now draw a list of *Shot*s with *forEach* and a *Canvas*:

<u>inside of *UFOWorld* :</u>

```
public void draw() {
 drawBackground();
 ...
 shots.forEach(new Draw(theCanvas));
 ...
}
```

<u>inside of *UFOWorld* :</u>
```
private IAction<Shot> dS =
 new Draw(theCanvas);
public void draw() {
 drawBackground();
 ...
 shots.forEach(dS);
 ...
}
```

We show both solutions again, the notational concise version as well as the one that creates only one object to represent the method.

As you can see, designing actions for imperative traversals is straight-forward. Following the design recipe is too much work for such non-recursive methods, also because you (should) have internalized it all. Even the notational overhead seems high; you add private classes and instantiate them, even if just once. If you recall *How to Design Programs*'s **lambda**-defined functions, you sense that there should be an easier way. Fortunately, there is.

Exercises

Exercise 38.1 Use the list library from figure 210 to design the method *drawAll*. The method consumes a list of *Posn*s and draws them as red dots on a 200 by 200 canvas. Add the method to an *Examples* class. ∎

Exercise 38.2 Use the list library from figure 210 to design the method *swap*. The method consumes a list of *Posn*s and imperatively swaps the *x* with the *y* coordinates in each *Posn*. ∎

Exercise 38.3 Design a simplistic representation for a grocery store. Think of the latter as an object that aggregates a collection of sales items, where each item comes with a name and a price. Use the list library from figure 210 to represent this aggregation here. Then design the method *inflation* for the *Store* class. The method raises the price on all grocery items in the store by some given factor. ∎

38.3 Using *forEach* with Anonymous Classes

In section 37.4 we briefly alluded to the idea of implementing and instantiating interfaces without defining a class explicitly. Let's consider a simple case. Suppose you are using a library that exports an Object-based "method" interface:

> **interface** *IFunII* {
>   int *invoke*(int *i*);
>   }

The interface describes a methods-as-objects representation for methods that consume and produce ints.

Now imagine that you need an instance of a specific implementation of *IFunII* in one—and only one—class *C*. Based on what you know, you would have to define an implementation of *IFunII* and instantiate it in *C*:

class *FunII* **implements** *IFunII* {   **class** *C* {
  public int *invoke*(int *i*) {   . . .
    return *i+1*;   . . . **new** *FunII*() . . .
  }   . . .
}   }

For these cases—when there is a single reference to a class that implements an interface—you're best off implementing the interface *anonymously*, that is, without giving it a name and without even introducing a class definition:

> **class** *C* {
>   . . .
>   . . . **new** *IFunII*() {
>   **public** int *invoke*(int *i*) {
>   **return** *i+1*;
>   }
>   }
>   . . .
>   }

That is, **new** is useful in conjunction with an interface *I* if the **new** *I*() is followed by a "block" that defines the specified methods of *I*.

Notes: Since it is impossible to define a constructor for an anonymous class—we don't even have a name for it—you need to initialize the fields of an anonymous interface implementation directly. As you do so, you may wish to use local fields, which is legal in Java, or local variables, which isn't (immediately) legal in Java, though in other object-oriented languages. Because of the Java specificity of this issue, we recommend that you look up the exact rules and mechanisms when you are to design lasting programs in Java. Until then, keep in mind that it is about design principles not language details. ∎

When you are dealing with generic interfaces, the anonymous implementation must also specify type arguments for the type parameters. Here is, for example, an anonymous instantiation of *IAction*:

```
new IAction<Shot> () {
  public void invoke(Shot s) {
    s.move();
    return ;
  }
}
```

Because the interface is parameterized over the type of list items that its *invoke* method must process, the generic interface is applied to one type, *Shot* in this case. Furthermore, because the interface specifies one method signature in terms of its type parameter, the anonymous implementation consists of a "block" with a single method on *Shot*s.

As you can easily tell, both *Move* (for *Shot*) and *Draw* (also for *Shot*) occur once in *UFOWorld* . Hence these classes are candidates for replacing them with anonymous implementations and instantiations. Indeed, doing so is relatively easy; it is just like the above, abstract example:

inside of *UFOWorld* :
public void *move*() {
 ufo.move();
 aup.move();
 shots.forEach(

  new *IAction<Shot>* () {
   public void *invoke*(*Shot s*) {
    s.move();
   return ;
  }
 }   );
 charges.move();
 return ;
}

inside of *UFOWorld* :
public void *draw*() {
 drawBackground();
 ufo.draw(*theCanvas*);
 aup.draw(*theCanvas*);
 shots.forEach(

  new *IAction<Shot>* () {
   public void *invoke*(*Shot s*) {
    s.draw(*theCanvas*);
   return ;
  }
 }   );
 charges.draw(*theCanvas*);
 return ;
}

The one aspect worth a remark is the reference to *theCanvas* from the anonymous instantiation of *IAction* in the *draw* method. Because *theCanvas* is a field in the surrounding class—indeed, a field in the superclass—this reference is legal.

Let's look at one last example, the creation of a complete card deck. Card games exist in many different cultures and come in many different forms. All of them, though, involve a deck of cards, and cards belong to a suit and have a rank. Figure 211 summarizes the scenario in those terms.

The *Game* class corresponds to the *World* class with which we always start to lay out what we have. It sets up the legal *suits* and *ranks* for the game, both individually and as lists. The next field creates a deck. The rest is left to a *setUp* method, which is presumably responsible for creating the deck, shuffling it, dealing the cards to players, and so on. Its first action is to invoke *createDeck*, which is to add cards of all (specified) *suits* at all (specified) *ranks* to the initially empty *deck*. All of this is captured in the method's purpose and effect statement.

Our goal here is to design this *createDeck* method. With the purpose statement given, we can move straight to the examples, which is best done with a table here:

	Seven	Eight	Nine	...	Ace
Clubs	...	...	...	...	...
Diamonds	...	...	...	...	...
...	...	...	...	...	...

```
class Game {
  Suit d = new Suit("Diamond");
  Suit c = new Suit("Clubs");
  . . .
  Rank seven = new Rank("seven");
  . . .
  Rank ace = new Rank("ace");
  IList<Suit> s0 = new Mt<Suit>();
  IList<Suit> s1 = new Cons<Suit>(d,s0);
  . . .
  IList<Suit> suits = new Cons<Suit>(c,...);
  IList<Rank> r0 = new Mt<Rank>();
  IList<Rank> r1 = new Cons<Rank>(seven,r0);
  . . .
  IList<Rank> ranks = new Cons<Rank>(ace,...);
  Deck deck = new Deck();
...
  public void setUp() {
    this.createDeck();

    . . .
  }

  // effect: add all suits at all ranks to the deck
  private void createDeck() { ... }
}
```

```
class Card {
  Suit s;
  Rank r;
  // constructor omitted
}
```

```
class Rank {
  String v;
  // constructor omitted
}
```

```
class Suit {
  String s;
  // constructor omitted
}
```

```
class Deck {
  private IList<Card> listOfCards = new Mt<Card>();
  public Deck() { }

  // add a card to this deck
  public void addCard(Card c) {
    listOfCards = new Cons<Card>(c,listOfCards);
  }
}
```

Figure 211: Card games and decks of cards

The first row specifies some of the possible ranks, the first column lists the possible suits. For each pairing of *suits* and *ranks*, the *createDeck* method must create a card and add that card to the deck. The problem statement implies that the order appears to be irrelevant.

Both the purpose statement and the examples suggest that the method must traverse *ranks* as well as *suits*. Following the advice in *How to Design Programs*, we should consider three cases: processing one while treating the other as a constant; processing both in parallel; and processing the cross product. The table suggests the last option, meaning the method should traverse one list and then, for each item on that list, the other.

At this point, you guess and choose one of the lists as the primary list for iteration. If the design fails, you choose the other order. Let's start with *suits* for now and lay out the template for *forEach*:

```
void createDeck() {
  suits.forEach(new IAction<Suit> () {
    public void invoke(Suit s) {
      ... s ... ranks ...
      return ;
    }
  });
  return ;
}
```

This template is dictated by the choice to traverse *suits* with *forEach* and the shape of *IAction<Suit>*. Concretely, the two say that you must design an *invoke* method and that *invoke* is applied to one suit at a time.

The template's body tells us that *s* and *ranks* are available. From the example step we know that the method is to traverse the list of *ranks* and pair each with *s*. Put differently, *invoke* must traverse *ranks*. Since a traversal within a traversal sounds complex, the proper decision is to design an auxiliary method or actually class, because methods are represented as objects:

```
void createDeck() {                          // effect: add all ranks at suit s
  suits.forEach(new IAction<Suit> () {       //    to the deck of this game
    public void invoke(Suit s) {             class PerRank
      ranks.forEach(new PerRank(s));           implements IAction<Rank> {
      return ;                                 Suit s;
    }                                          PerRank(Suit s) {
  });                                            this.s = s; }
  return ;                                     public void invoke(Rank r) { ... }
}                                            }
```

The full method definition on the left assumes that we can design the auxiliary "method." For the latter, the partial class definition on the right shows

how much we know: the purpose and effect statement, the outline of the class, and that it needs a field to keep track of the current suit (*s*).

Designing *PerRank* proceeds just like the design of any method. For examples, we can take examples for *createDeck* and formulate examples for *PerRank*. More precisely, *PerRank* holds the suit constant and looks at all ranks, meaning it is one row of the above table. To translate this into a template, we sketch out the body of the *invoke* method:

incPerRank
  public void *invoke*(*Rank r*) {
    . . . *s* . . . *r* . . .
  }

where *s* is the chosen suit and *r* is the rank that the method is currently processing. With all this information laid out, you can translate the purpose and effect statement into a full method definition:

```
void createDeck() {                     // effect: add all ranks at suit s
  suits.forEach(new IAction<Suit> () {  //     to the deck of this game
    public void invoke(Suit s) {        class PerRank
      ranks.forEach(new PerRank(s));      implements IAction<Rank> {
      return ;                            Suit s;
    }                                     PerRank(Suit s) {
  });                                       this.s = s; }
  return ;                                public void invoke(Rank r) {
}                                           deck.addCard(new Card(s,r));
                                            return ;
                                          }
                                        }
```

For completeness, we re-state the definition of both the main method, *createDeck*, as well as the full definition of the auxiliary "method." Figure 212 summarizes the design.

38.4   Mini Projects, including a Final Look at "War of the Worlds"

The following exercises propose small projects that use one and the same list library (exercise 38.4) without modification for representing aggregations or collections of objects. Each exercise involves the use of either *forEach* or *traverse* or both. Try to design anonymous instantiations of the corresponding interfaces where possible; if not, explain why and use inner classes. As you tackle these projects, don't forget that the first draft of any

```
class Game {
  IList<Suit> suits = new Cons<Suit>(c,...);
  ...
  IList<Rank> ranks = new Cons<Rank>(ace,...);
  Deck deck = new Deck();
...
  public void setUp() {
    this.createDeck();
    ...
  }

  // effect: add all suits at all ranks to the deck
  private void createDeck() {
    suits.forEach(new IAction<Suit> () {
        public void invoke(Suit oneSuit) {
          ranks.forEach(new PerRank(oneSuit));
          return ;
        }
      });
    return ;
  }

  // effect: add all ranks of a given suit to the deck of this game
  private class PerRank implements IAction<Rank> {
    Suit s;
    PerRank(Suit s) { this.s = s; }
    public void invoke(Rank r) {
      deck.addCard(new Card(s,r));
      return ;
    }
  }
}
```

Figure 212: Card games and decks of cards, refined

design is just that: a draft. Don't forget to use one method per task, and don't forget to use the recipes for abstraction to *edit* your programs.

Exercises

Exercise 38.4 Equip the list library from figure 210 with a visitor-based traversal method. ∎

Exercise 38.5 Re-design your "War of the World" project. Start from a version of the game that allows the UFO to drop charges on a random basis (see exercise 19.20, page 284). ∎

Exercise 38.6 Re-design the imperative version of your "Worm" project. See section 19.9 for the original case study and exercise 27.16 (page 408) for the development of the imperative version. ∎
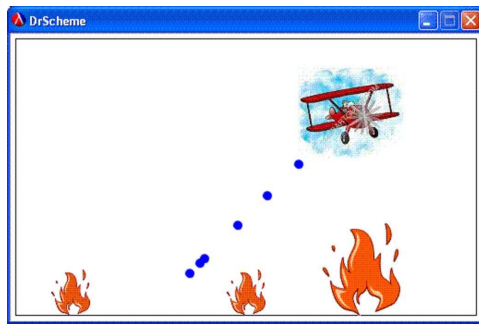

Figure 213: Action in the "Fire Plane" game

Exercise 38.7 The interactive computer game "Fire Plane" is about extinguishing wild fires, a nuisance and occasionally a serious danger in the western states of the US and in many other countries, too.[101] Imagine a prairie with fires flaring up at random places. The player is in control of a fire plane. Such an airplane has water tanks that it can empty over a fire, thus extinguishing it.

Design a minimal "Fire Plane" game:

1. Your game should should display one fire when the game starts.

2. While any fire is burning, your game should add other fires at random places. The number of fires should be unlimited, though, you should be careful not to start too many fires at once.

3. Also, your game should offer the player one fire plane with a fixed number of water loads. The fire plane should move continuously. You are free to choose the means by which the player controls the movements of the plane and which movements it may perform.

---

[101] We thank Dr. Kathi Fisler for the idea of the fire-fighting game.

4. Finally, write a brief introduction for players.

See figure 213 for a screenshot; your program should render the airplane in a simpler manner than our prototype. ∎

Exercise 38.8 Design a data representation for a library. For our purposes, a library owns a collection of books. Naturally it starts with no books and acquires them one at a time. For this exercise, a book has a title, a "check out" status, and a unique identification number. The latter is issued when the book is added to the library.

Imagine for this last part that the "book addition" method makes up a unique number (int) and draws it on a paper label that is then attached to the book's spine. Your data representation should also support a method for checking out individual books. To this end, the librarian enters the unique identification number from the book's label (via a barcode reader), and the software uses this number to locate the book and to mark it as checked out. Finally, books are returned in bunches. Add a method for returning several books by their identification number. ∎

Exercise 38.9 A *web crawler* is a program that explores web pages on the internet and acquires knowledge about how the pages are interconnected. Companies then use this information to provide all kinds of services.

Think of giving a web crawler one web page and asking it to visit all reachable web pages. To reach the first web page, a crawler is given a link— also known as a *URL* or *URI*. It then looks at the page to find links to other pages. When web page *A* contains URLs to web pages *B*, *C*, etc., then the latter are recorded as immediate neighbors of *A*. Once the crawler is done with *A*, it asks its knowledge base whether there are more pages to visit, and if so, requests the link for the next page to visit. In addition to recording the "neighborhood" relationship among web pages, the web crawler also records how many times a web page has been reached. Notice that reaching a web page differs from visiting it.

Design a data representation for the knowledge base of a web crawler. If you have solved the exercises in section 37.7, adapt one of those graph libraries for this exercise; otherwise design one using the hints from that section. At a minimum, the main class of the data representation should support methods for recording that some pages are neighbors of a given page; for marking a page as visited; for determining whether there are more pages to visit; for picking another page to visit; for determining how many times a page has been reached; and for the graph of web pages acquired (thus far). ∎

38.5   Abusing the *forEach* Method

Traditional approaches to programming abuse methods such as *forEach* for computations for which it isn't intended. Worse, some programming languages necessitate such abuses for various reasons. As it turns out, Java is one of those languages. The next chapter explains Java's problem; how this problem forces programmers to abuse a construct similar to *forEach*; and how you must cope with this problem. This subsection prepares the next chapter with an introduction to the idea of using imperative constructions to compute functional results.

Unlike other parts of the book, we use simplistic examples to illustrate the idea. Let's start with a list of integers:

> inside of *Examples* :
>
> $\overline{IList<Integer>\ mt} = $ **new** *Mt<Integer>*();
> *IList<Integer> n1* = **new** *Cons<Integer>*(1,*mt*);
> *IList<Integer> n2* = **new** *Cons<Integer>*(2,*n1*);
> *IList<Integer> n3* = **new** *Cons<Integer>*(3,*n2*);

In the first part of this chapter we have seen how to use abstract traversals, such as *map* or *traverse* to compute values from such lists. Here is a method that uses *traverse* to compute the sum of such lists:

> inside of *Examples* :
>
> // determine the sum of *l*
> **public** int *sum(IList<Integer> l)* {
>   **return** *l.traverse*(**new** *IListVisitor<Integer,Integer>* () {
>   **public** *Integer visitMt(Mt<Integer> o)* {
>   **return** 0;
>   }
>   **public** *Integer visitCons(Cons<Integer> o)* {
>   **return** *o.getFirst*() + *o.getRest*().*traverse*(**this**);
>   }
>   });
> }

It is like a Scheme function, using a visitor to distinguish between the two cases and to recursively add up the numbers.

Now imagine that someone evil had whimsically decided to equip the list library only with the *forEach* method and to make it impossible to modify or extend the library. At this point you have two choices: design your own list representation, duplicating some of the work, or use the existing

library. If you try the latter, you must figure out how to define *sum* with *forEach* instead of *traverse*:

> inside of *Examples* :
> _____
> // determine the sum of *l*
> **public** int *sumFE(IList<Integer> l)* {
>   *l.forEach*(**new** *IAction<Integer>* () {
>   **public** void *invoke(Integer i)* {
>   ... *i* ...
>   **return** ;
>   }
>   });
>   **return** ... ;
> }

Given that *forEach* produces no result, the invocation of the method on *l* can't be the last part of the method. Similarly, *invoke*'s return type is also void, meaning it too can't produce a value via a plain return. Thus the first conclusion is that we need an stateful field[102] and an assignment to this field in *invoke* to communicate the sum from inside of *invoke* to its surroundings:

> inside of *Examples* :
> _____
> **private** int *sumAux* = 0;
> **public** int *sumFE(IList<Integer> l)* {
>   *l.forEach*(**new** *IAction<Integer>* () {
>   **public** void *invoke(Integer i)* {
>   ... *i* ...
>   *sumAux* = ...
>   **return** ;
>   }
>   });
>   **return** ... ;
> }

The new field is called *sumAux* because it is associated with *sumFE*. Also note that, following the design recipe for imperative methods, *invoke* now comes with a partial assignment statement to the new field.

The template is suggestive enough to make progress. If *invoke* adds *i* to *sumAux* for each item on the list, *sumAux* should be the sum of all integers

---

[102]We can't use a local variable that is hidden inside of *sumFE* due to Java's restrictions. See the note on this issue in the preceding section.

after the invocation of *forEach* is evaluated:

```
inside of Examples :
private int sumAux = 0;
public int sumFE(IList<Integer> l) {
  l.forEach(new IAction<Integer> () {
    public void invoke(Integer i) {
      sumAux = sumAux + i;
      return ;
    }
  });
  return sumAux ;
}
```

The result looks as succinct as the original *sum* method, though it does use effects to compute a value. All that remains at this point is testing.

While we usually leave testing to you, running the tests is highly instructive here. Thus add the following tests to *Examples*:

```
inside of Examples :
... checkExpect(sumFE(n1),1,"one element list") ...
... checkExpect(sumFE(n3),6,"three element list") ...
```

and run them to experience the following surprise:

```
  Ran 2 tests.
  1 test failed.
...
  actual: 7
  expected: 6
```

The second test fails, because *sumFE* returns 7 when 6 is expected.

Stateful classes and fields are tricky, and we have just been bitten. While the *sumAux* field is initialized to 0, the *sumFE* method just keeps adding to *sumAux* never re-setting it when it is done with a list. There are two obvious solutions: one is to set *sumAux* to 0 *before* the list is traversed and another is to set it to 0 afterwards. The first solution is simpler to write down than the second one and easier to comprehend:

```
inside of Examples :
private int sumSoFar;
public int sumFE(IList<Integer> l) {
  sumSoFar = 0;
  l.forEach(new IAction<Integer> () {
    public void invoke(Integer i) {
      sumSoFar = sumSoFar + i;
      return ;
    }
  });
  return sumSoFar ;
}
```

A close look at this definition shows that initializing the field to 0 at the very beginning has two advantages. First, the initialization and the assignment in the *forEach* traversal show the reader that the purpose of the field is to represents the sum of the integers encountered so far. We have therefore renamed the field to *sumSoFar*. Second, the initialization avoids accidental interferences, just in case some other method uses the field.

Let's consider a second example, selecting all positive numbers from a list. Since the problem statement contains a perfectly phrased purpose statement, we just add a contract to get started:

```
inside of Examples :
// select all positive integers from the given list
public IList<Integer> allPositiveV(IList<Integer> l) { … }
```

and move straight to the example step of the design recipe:

```
inside of Examples :
IList<Integer> mt = new Mt<Integer>();
IList<Integer> i1 = new Cons<Integer>(0,mt);
IList<Integer> i2 = new Cons<Integer>(1,i1);

IList<Integer> o1 = new Cons<Integer>(1,mt);

… checkExpect(allPositive(i1),mt) …
… checkExpect(allPositive(i2),o1) …
```

The examples remind you of two points. First, *allPositive* isn't a method (of the integer list representation), but a function-like construction. Second, because 0 isn't positive, the function drops it from both *i1* and *i2*.

Furthermore, the two examples fail to explore the problem statement properly. Although the design of proper examples and tests is a topic for a

book of its own, this sample problem demands a third test, one that can tell whether the function-method preserves the order of the numbers:

inside of *Examples* :
$\overline{IList{<}Integer{>}\ i3 =}$ **new** *Cons<Integer>(2,i2)*;

IList<Integer> o2 = **new** *Cons<Integer>(2,o1)*;

... *checkExpect(allPositive(i3),o2)* ...

While the preservation of order is not mentioned in the problem statement, you—the problem solver—should wonder about it and writing down an example shows that you did.

At this stage in your development, you can skip the template and write down the definition:

inside of *Examples* :
```
// select all positive integers from the given list
public IList<Integer> allPositiveV(IList<Integer> l) {
  return l.traverse(
    new IListVisitor<Integer,IList<Integer>> () {
      public IList<Integer> visitMt(Mt<Integer> o) {
        return new Mt<Integer>();
      }
      public IList<Integer> visitCons(Cons<Integer> o) {
        int f = o.getFirst();
        if (f > 0) {
          return new Cons<Integer>(f,o.getRest().traverse(this)); }
        else {
          return o.getRest().traverse(this); }
      }
    });
}
```

Given that the purpose of the method is to traverse the given list and to make a decision for each integer on the list, it is natural to invoke the *traverse* method and to use an anonymous implementation and instantiation of the *IListVisitor* interface. This *visitor*'s first method returns an empty list for a given empty list, and its second method inspects the first integer before it decides whether to add it to the result of traversing the rest.

Following the first example, we can obviously abuse *forEach* in a similar way, using a (private) field to keep track of the positive integers so far:

inside of *Examples* :

// select all positive integers from the given list
private *IList<Integer> posSoFar*;
public *IList<Integer> allPositiveFE(IList<Integer> l)* {
  posSoFar = **new** *Mt<Integer>*(); ₁

  l.forEach(**new** *IAction<Integer>*() {
          public void *invoke(Integer f)* {
            if (*f* > 0) {
                  posSoFar = **new** *Cons<Integer>*(*f*,*posSoFar*);
                return ; }
              else {
              return ;}
                }
          });
                                                                               ₂

  return *posSoFar*; ₃
}

Like the definition of *sumFE*, the one for *allPositiveFE* consists of three parts, each highlighted in gray and labeled with a subscript:

1. The first part initializes the auxiliary **private** field to the proper value. Here this means an empty list, because no positive integer has been encountered so far.

2. The second part is an imperative traversal based on *forEach*. For each positive integer encountered, the *invoke* method updates the private field via an assignment statement. It thus adds the most recently encountered positive number to the list of positives seen so far before *forEach* continues to process the list.

3. The third part of the method returns the current value of the field.

Our description says that *allPositiveFE* proceeds like *sumFE*. While the latter starts with 0 as the sum seen so far and adds integers as it encounters them, *allPositiveFE* starts with an empty list and adds positive integers.

   This last thought suggests that the results of *allPositiveFE* and *allPositive* differ in the order in which the numbers appear on the result list. And indeed, adding appropriate examples and tests spells out and confirms this difference in plain view:

    ... *checkExpect*(*allPositive*(*i3*),
                    new *Cons<Integer>*(2,**new** *Cons<Integer>*(1,*mt*)) ...
    ... *checkExpect*(*allPositiveFE*(*i3*),
                    new *Cons<Integer>*(1,**new** *Cons<Integer>*(2,*mt*)) ...

The two examples suggest that methods based on *forEach* traversals are *accumulator* versions of their *traverse*-based counterparts. You may wish to re-visit *How to Design Programs*, chapter VI, to refresh your memory of the differences between naturally recursive functions and their accumulator-style counterparts. We continue to explore the topic, as well as the abuse of *forEach*, in the exercises.

Exercises

Exercise 38.10 Design accumulator-based variants of *sum* and *allPositive*. In other words, design applicative visitor classes and use instances of these visitors in conjunction with *sum* and *allPositive*. ∎

Exercise 38.11 While *allPositiveFE* and *allPositive* obviously produce different results, this doesn't appear to be true for *sumFE* and *sum*. Can you think of a list of integers for which the two methods would produce different results? How about doubles? ∎

Exercise 38.12 Design two variants of *juxtapose*, which consumes a list of *String*s and computes their juxtaposition. The first uses *traverse* to accomplish its purpose, the second uses *forEach*. Do their results differ for the same input? ∎

Exercise 38.13 Design two variants of the method *min*, one using *traverse* and the other using *forEach*. The method consumes a non-empty list of ints and determines the minimum. ∎

Exercise 38.14 Design two variants of the method *closeTo*, one using *traverse* and the other using *forEach*. The method consumes a list of *Posn*s and determines whether any one of them is close to the origin. For the purpose of this exercise, "close" means a distance of less or equal to 5 (units). Does one variant have an performance advantage over the other? ∎

Exercise 38.15 Use *forEach* to design the method *sort*, which consumes a list of ints and produces one sorted in ascending order. ∎

```
LIBRARY
              +-----------+                    +----------+          *
              | IIfc      |                    | IVisitor |          *
              +-----------+                    +----------+          *
                   |                                |                *
                  / \                              / \               *
                  ---                              ---               *
                   |                   ******************|****************    FUNCTIONS
       +---------------+---------------+-------+ *    +--------+--------+------------+
       |               |               |       | *    |        |        |            |
  +-----------+  +-----------+  +-----------+ | *  +--------+      +--------+   +----------+
  | ACls      |  | BCls      |  | CCls      | | *  | Visit1 |  ... | Visitn |   | Visitn+1 |
  +-----------+  +-----------+  +-----------+ | *  +--------+      +--------+   +----------+
                                             | *
  * * * * * * * * * * * * * * * * * * * * * * | *
    DATA EXTENSION                            |
                                              |
                               +-----------+
                               | DCls      |
                               +-----------+
```
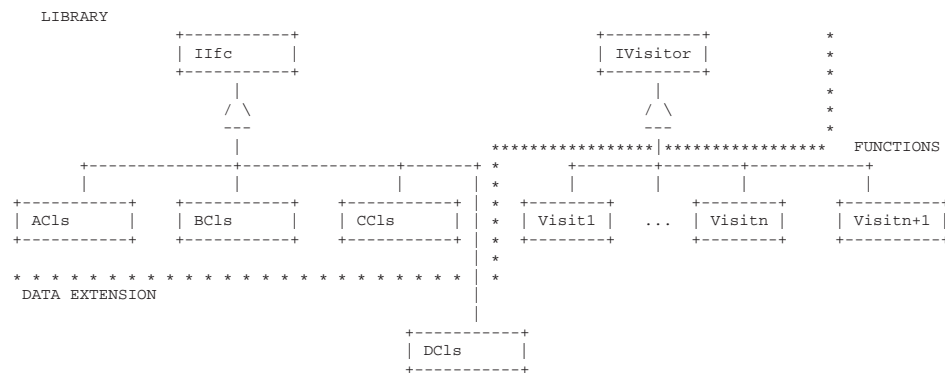
Figure 214: Data and functional extensions of frameworks

39 Extensible Frameworks with Visitors

the core library comes in the same old shape: an interface and a bunch of implementing classes, plus a visitor interface

functions can be defined inside or outside the library by just implementing the visitor interface. great

how about data extensions? if you design a class *DCls* that implements *IIfc* and you create instances, what will your visitors do? they can't cope, because they are intimately tied to the organization of the union in the core library.

research problem: how can you add extensions and simultaneously revises your visitors, too? does it always work?[103]

keep this small; it isn't as relevant as the for loop stuff

---

[103]See SK, DF, MF, plus MF/DF

Intermezzo 6: Generic Methods, Inner Classes

syntax
    typing and subtyping
    BUT semantics: compile to the Object based solution with subtyping
    error messages are weird:

TODO

purpose: to apply programming via refinement in the Java context

Loops and Arrays

Java suffers from a serious design flaw that impinges on proper object-oriented design.[104]

it is not just tail-recursion, there is something else too (i wish i could recall)

and

void for(...)

is the only loop that Java provides. this forces a programmer to write imperative code almost all the time, even though Java by design is an object-oriented language otherwise.

41   The Design Flaw

42   Loops

42.1   Designing Loops

42.2   Designing Nested Loops

```
deck of cards

IList<Suit> suits = ...
IList<Rank> ranks = ...

suits.forEach(new IAction<Suit> () {
  public void invoke(Suit suit) {
   ranks.forEach(new IAction<Rank> () {
     public void invoke(Rank rank) {
      deck.add(new Card(suit,rank));
```

---

[104]Guy Steele, a co-specifier of the language, has repeatedly agreed with this statement in public, the last time during a talk in Northeastern University's ACM Curriculum Series.

```
    }
  })
  }
})

for (Suit suit : suits)
    for (Rank rank : ranks)
        sortedDeck.add(new Card(suit, rank));
```

42.3   Why Loops are Bad

43   From Design to Loops

44   ArrayLists

relative to absolute distances

Intermezzo 7: Loops

syntax

    typing and subtyping

    BUT semantics: compile to the Object based solution with subtyping

    error messages are weird:

TODO

purpose: some more Java linguistic (static, packages); some Java lib stuff (collections, stacks); some Java GUI stuff (world, if you don't have our libraries)

VIII   Java

46   Some Java Linguistics

packages
    final
    + is overloaded for strings
    static inner classes

47   Some Java Libraries

collections, maps, stacks, queues, ...

48   Java User Interfaces

48.1   Scanning and parsing

48.2   Graphical User Interfaces

some minor swing and friends

49   Java doc

To CHECK

Developing Large Programs
   Iterative Refinement
   Layers of Data Abstraction

1. AM I INTRODUCING OVERLOADED METHODS PROPERLY?

2. AM I INTRODUCING OBJECT as SUPER before V?

3. WHEN do I drop "this"?

   ———————————————

4. cross-check all figures: captions, code arrangements

5. cross-check all code displays: left margin, top margin

   (a) frame classes, interfaces separately

   (b) arrange boxes similarly

   (c) format Java comments properly // $\\mbox{ ... }$
       use **this**
       the purpose statement of every boolean typed method should be
       a question

   (d) diagrams: classes that implements interfaces shouldn't repeat
       methods
       From David: Diagram conventions for classes with no fields ver-
       sus interfaces are inconsistent.
       Sometimes (eg. Fig 16) an interface is denoted with double un-
       derline:

       ```
       +--------------+
       | ITaxiVehicle |
       +--------------+
       +--------------+
       ```

       Sometimes (eg. Fig 18) it is denoted with single underline:

       ```
       +------+
       | ILog |
       +------+
       ```

Most of the time (eg. Fig 30), a class with no fields is drawn with single underline:

```
+---------+
| MTShots |
+---------+
```

But sometimes they are drawn with a double underline, as in Fig 31:

```
+------+
| Blue |
+------+
+------+
```

(e) code: add(?) information about the package imports to world-ish figures

6. cross check all Designing subsections for style/content

7. check on examples clases, should they have 0-ary constructors?

To Do

1. make sure to say how to create instances of Object at the end of chapter 4 when Object is introduced

2. part should be chapter

3. should stub definitions replace headers in the design recipe?

4. else { should be on a line by itself. The whole if { } else { } type setting requires a close look.

5. templates shouldn't have comments

6. CHANGE RECIPE SO THAT STUDENTS USE THE TYPES OF THE INVENTORY ITEMS

7. ?? replace "template" with "inventory", plus footnote ??

8. use 'natural recursion' a lot in chapters 2 and forth

9. drop this. from iv on up

10. add exercises to the section on method dispatch and type checking

11. ProfessorJ Companion Guide (appendix? on-line?)

12. Eclispe transition as a separate guide (appendix? on-line?)

Unallocated Goals

File allocations

Robby: 1.tex 1-2.tex 2.tex 1.tex 2-3.tex 3-4.tex 3.tex
Matthias: 4-5.tex 4.tex 5.tex 6.tex 7.tex