Elements of Object-Oriented Program Design

Robert "Corky" Cartwright

Please send corrections and comments to cork@rice.edu

January 10, 2010

Contents

Chapter 1

From Scheme to Java

1.1 Introduction

At first glance, Java and Scheme appear to have little in common. Java is written in notation similar to the widely-used C programming language, while Scheme is written in *parenthesized prefix* notation. In addition, Java expresses computations primarily in terms of operations attached to *objects* while Scheme expresses computations primarily in terms of functions applied to values—mechanizing the familiar world of algebra. In short, Java is a data-centered language while Scheme is a function-centered language.

Nevertheless, Java and Scheme are surprisingly similar beneath the surface. In this short monograph, we will discover how easy it is for a Scheme programmer to learn to write good Java code. The only obstacles are learning Java notation and learning how to organize programs in *object-oriented* (data-centered) form.

1.1.1 What is an Object?

Before discussing the specifics of Java's object system, let's define what an *object* is. Within a computer program, an *object* consists of

- a collection of variables called *fields* representing the properties of a specific physical or conceptual object, and

- a collection of designated operations called *methods* for observing and changing the fields of that object.

No code other than the designated operations can access or modify object fields. The fields and methods of an object are often called the *members* of the object. Each member of an object has a unique identifying name.

To make the notion of object more concrete, let us consider a simple example. Assume that we want to maintain a directory containing the office address and phone number for each person in the Rice Computer Science Department. In Java, each entry in such a directory has a natural representation as an object with three fields containing the person's name, address, and phone number represented as character strings. We defer discussion about how to represent the directory itself until Section 1.4.

Each *entry* object must include operations to retrieve the *name*, *address*, and *phone* number fields, respectively.

Let's summarize the form of a directory entry as a table:

```
Fields:
    String name;
    String address;
    String phone;

Methods:
    String name();
    String address();
    String phone();
```

This tabular description is *not* legal Java syntax. We will introduce the actual syntactic details later in this discussion.

The three methods `name()`, `address()`, and `phone()` do not take any *explicit* arguments because they are invoked by *sending* a "method call" to an *entry* object, called the *receiver*, which serves as an *implicit* argument for the method. In Java, the code defining a method can refer to this implicit argument using the *keyword* `this`, which is reserved for this purpose.

The syntax for method invocation in Java is quite different from the syntax for function application in Scheme. Given an object `o` of some class `C` including the 0-ary method `m`, the expression

```
o.m()
```

invokes the `m` method of object `o`. The object `o` in the method call above is called the *receiver* because it "receives" the method call. (In the *Java Language Specification*, the term *target* is used instead of *receiver*.)

In general, method invocations take arguments. The number and type of the argument expressions in an invocation must match the method's signature given in its declaration. The the receiver `o` in a method call `o.m(`$\mathtt{e}_1$`, ..., `$\mathtt{e}_n$`)` can be any legal Java expression. Similarly, each argument $\mathtt{e}_i$ can be any legal of expressions of the type specified for that argument.

Consider the following example: assume that a Java program can access an `Entry` object `e` and needs to get the value of the `name` field of this object. The method invocation

```
e.name()
```

returns the desired result. For each field declared in a class, Drjava automatically generates a corresponding *accessor* (or *getter*) method with the same name as the field. Accessor methods are 0-ary; they do not take any explicit arguments beyond the *receiver* written to the left of the method application.

Of course, the Scheme expression equivalent to

```
e.name()
```

is written

```
(Entry-name e)
```

Accessor methods in Scheme must be unary because there is no implicit receiver object in a function application.

Finger Exercise: In the DrJava programming environment, set the Language Level to Elementary. Then type the text

```
class Entry {
  String name;
  String address;
  String phone;
}
```

in the Definitions (top) pane, save it as the file `Entry.dj0`, compile it (using the "Compile" button), and type the following statements in the Interactions pane:

```
Entry e = new Entry("Corky","DH 3104","x 6042");
e.name()
e.phone()
```

The first line defines a variable `e` as an `Entry` object with `name` field `"Corky"`, `address` field `"DH 3104"`, and `phone` field `"x 6042"`. The second line is an expression that computes the `name` field of `e`. What value does the Java evaluator return for the `name` field of `e`? The `phone` field of `e`? ■

If we want to perform more interesting computations than constructing objects and accessing the values of their fields, we need to introduce the primitive operations for computing with familiar forms of data like numbers and strings.

Finger Exercise: In the DrJava Interactions pane, try evaluating the following expressions:

```
-5 + 3
-(5 + 3)
5 % 3
5./3.
5 / 0
5./0.
3 + .1 * .1 - 3.
5 < 6
5. > 6.
"Cor" + "ky"
"DH" + " 3104"
```

directly in the Interactions pane. Did you get the answers that you expected? ■

Java has the same precedence rules for expressions built from primitive operations as C/C++, which *roughly* follow standard mathematical conventions. The precedence of Java operations is explained in detail in Section 1.2.3.

Finger Exercise: In the DrJava Interactions pane, try evaluating the following expressions:

```
72. - 32. * 1.8
(72. - 32.) * 1.8
72. - 30. - 12.
72. - (30. - 12.)
```

Did you get the answers that you expected? ■

Program statements have essentially the same syntax in Java as in the widely used C language. The most common form of statement in Java is an assignment statement that introduces a new variable:

type *var* = *expr* ;

In the preceding syntax template, *type* is a Java type name, *var* is a Java variable name, and *expr* is an expression of type compatible with the type of *var*. The assignment statement

```
int x = 5;
```

introduces the variable `x` and gives it the value `5`. We used this form of statement in an earlier **Finger Exercise** to define an `Entry` object.

Finger Exercise: In the DrJava Interactions pane, try evaluating the following statements and expressions:

```
int x = 5;
x*x
double d = .000001;
double dd = d*d;
dd
dd*dd
1. + dd
1. + dd*dd
```

Did you get the answers that you expected? ■

Java includes all of the basic statement forms found in the C/C++ programming language expressed in essentially the same syntax. In the remainder of this monograph, we will introduce these statement forms as they are needed. Although Java accepts most C/C++ syntax, many common C/C++ constructions are considered bad style in Java.

Note that Java treats `boolean` as a distinct type from `int`, eliminating some common sources of errors in C/C++. For example, the test expression in a conditional statement (explained in Section 1.4.4 must be of type `boolean`.

Finger Exercise: In the DrJava Interactions pane, try evaluating the following sequence of statements and expressions:

```
int x = 7;
if (x = 5) y = 0; else y = 10;
y
```

Did you get the behavior that you expected? Repeat the exercise with corrected syntax, replacinng "=" in the test expression `(x = 5)` by "==". ■

Finger Exercise: In the DrJava Interactions pane, try evaluating the following sequence of statements and expressions:

```
boolean flag = (x = 7);
flag
```

Did you get the behavior that you expected? Repeat the exercise with corrected syntax (replacinng "=" in the right hand side of the definition of `flag` by "=="). ■

Finger Exercise: In the DrJava Interactions pane, try evaluating the following sequence of statements and expressions:

```
String first = "George";
String last = "Washington";
first + last
first + " " + last
first.concat(last)
first.concat(rest)
first.concat(" ").concat(rest)
```

Did you get the behavior that you expected? ■

Note that DrJava does not print the value of a Java expression if you type a semicolon following it. Java classifies variable bindings as expressions and they technically can be embedded within an expression, but this usage is generally considered bad style.

1.1.2 Classes: Templates for Creating Objects

Every Java program consists of a collection of *classes*—nothing else. A class is a template for creating a particular form of *object*. A Java `class` definition corresponds to a Scheme `define-struct` definition generalized to include all of procedures that process objects of the defined class. In Java, all program code must be part of some class.

Each *object* created by a `class` template contains the same *members*, each of which is either a *field* or a *method*. A *field* is a "container" that holds a value. A *method* is an operation on the fields of the object and any values that are passed as arguments to the method. The objects created by a particular class template are called the *instances* or *objects* of that class. Each instance contains the members specified in the class template.

Each member of a class has a *name* consisting of a Java *identifier*, any sequence of "alphanumeric characters" (letters, digits, and _) beginning with a letter or _. A few identifiers are reserved by Java for use as *keywords*. A complete list of the keywords of Java is given in Chapter 1.2 which is an Appendix on the syntax of Java. For now, the only keywords that concern us are:

```
abstract    class       float       new         void
boolean     double      int         return
case        else        if          then
char        extends     long        this
```

But beware: you cannot use any keyword for a regular identifier. So you will need to learn the list of keywords in Chapter 1.2 soon or you may trip over an inexpliciable error involving using a keyword as an ordinary identifier.

For now, we will require all the identifiers used within a class to be unique, *i.e.* you will not be allowed to use the same name for both a field and a method in a class. We will gradually relax this restriction as we learn more of the Java language, *e.g.*, in Section 1.10.6.

Let's re-examine the Java class `Entry` defined in the first Finger Exercise above. It consists of only three fields `name`, `address` and `phone` which must contain `String` values.

The DrJava Elementary language level automatically creates four *methods* of the class `Entry` including:

- a special *constructor* method

  ```
  Entry(String name, String address, String phone)
  ```

  Figure 1: The `Entry` class

  that creates and initializes `Entry` objects; and
- three *accessor* methods `name()`, `address()`, and `phone()` that extract from an `Entry` object;

An instance (object) of the class `Entry` is created by an expression of the form

```
new Entry("SomeName", "SomeAddress", "SomePhone")
```

The three accessor methods generated by the Elementary language level are very simple yet they illustrate the most important characteristic of object-oriented programming: *operations are attached to the data objects that they process.* The methods `name()`, `address()`, and `phone()` take no arguments yet they have access to the fields of the `Entry` object to which they are attached.

Finger Exercise: In DrJava, open the file `Entry.dj0` that you created in the very first **Finger Exercise**. The following Java program text

```
class Entry {
  String name;
  String address;
  String phone;
}
```

defining the `Entry` class should appear in the Definitions pane.

Compile the class by hitting the Compile button.

In the DrJava Interactions pane, try evaluating the following program text:

```
Entry e1 =  new Entry("Corky", "DH 3104", "x 6042");
e1.phone()
Entry e2 = new Entry("Stephen", "DH 3102", "x 3814");
```

```
class Entry {
  String name;
  String address;
  String phone;

  /* methods */

  /** Returns true if name matches keyName. */
  boolean match(String keyName) {
    return name.equals(keyName);
  }
}
```

Figure 1.1: The expanded `Entry` class

```
e2.phone()
e1.equals(e2)
Entry e3 = new Entry("Corky", "DH 3104", "x 6042");
e1.equals(e3)
```

■
We will explain the syntax of Java class definitions in more detail in Section 1.1.2.

1.1.3 Defining Instance Methods

The three primitive instance methods `name`, `address` and `phone` in class `Entry` all simply return the value of a field from the object that received the method call.

Let us define another method for class `Entry` that does more computation. Assume that we want to define an operation `match` for the class `Entry` that takes a string argument `keyName` and determines whether it matches the `name` field of the specified `Entry` object. We could include such a method definition within the definition of class `Entry` as shown in Figure 1.1.3:

The `match` method is implemented using the `equals` method on the `String` field `name`. Recall that the `String` class is built-in to Java. The `equals` method from the `String` class takes an argument and returns `true` if (and only if) it is a `String` with exactly the same contents as the receiver `String` object. Hence,

```
(new Entry("Corky","DH 3104","x 6042")) . match("Corky")
```

returns

```
true,
```

while

```
(new Entry("Corky","DH 3104","x 6042")) . match("Matthias")
```

returns

```
false.
```

The code for the `match` method has access to the fields of the Entry object to which they are attached even though this object is not explicitly passed as an argument to the method. The `Entry` object `e1` in the method call

```
e1.match(e2)
```

is called the *receiver* because it "receives" the method call. (In the *Java Language Specification*, the term *target* is used instead of *receiver*.) The receiver is as an *implicit* argument for the method. In Java, the code defining the method can refer to this implicit argument using the *keyword* `this`, which is reserved for this purpose.

Warning The Java infix operator `==` can be used to compare *objects*, but the results of such a comparison are problematic for many applications. On objects, the `==` operator returns `true` if (and only if) the both arguments are *exactly the same* object. Hence, if `x` is a variable of some object type T, the expression

```
x == x
```

returns

```
true
```

For distinct object arguments, the `==` operator returns `false`. Hence,

```
new Entry("Corky","DH 3104","x 6042") ==
  new Entry("Corky","DH 3104","x 6042")
```

returns

```
false
```

because each occurrence of `new` creates a distinct object. For most Java object types including `String`, the `==` operator is not a reliable mechanism for testing equality! For example, Java does not guarantee that it creates only one copy of a `String` constant.

Finger Exercises

1. Add the code for the `match` method to your `Entry` class. Test your code in the Interactions pane by evaluating some examples. You can archive test code entered in Interactions pane by saving your Interactions history; right-click on your mouse while it is over the Interactions pane. DrJava gives you an opportunity to edit this history (to eliminate erroneous entries) before saving it as a file. Save your history in the file `Entry.hist` (DrJava automatically adds the file extension `.hist` to the file name that you provide). ■

2. Modify your `match` method to use the `==` operator instead of the `equals` method. Try running you archived tests stored in `Entry.hist`. If none of these tests fail, devise some test cases where using `==` makes the test fail! Hint: the Java compiler only constructs one copy of a `String` constant, so repeated uses of a given String all refer to same object. The DrJava interpreter, however, constructs a new copy of a `String` constant for every occurrence. Hence a `String` constant that appears in the DrJava Interactions pane is distinct from the same `String` constant defined in a class in the Definitions pane. Moreover, different occurrences of the same `String` constant in the Interactions pane refer to different objects. Save your updated test history in the file `Entry.hist`. ■

Java Design Rule: There are only two valid uses of the `==` operator:

- to compare values of primitive type; and
- to test object *identity* (not equality!).

The second use is relatively uncommon.

In the code for the `Entry` class, the constructor and accessor methods all refer to fields of `this`, the hidden parameter bound to the object that is the receiver of the method invocation. For example, the expression

```
this.name
```

returns the value of the `name` field of the object `this`. In constructor invocations, `this` is bound to the newly allocated object.

One attractive feature of Java is that the method syntax mandates a contract (declaration of input types and output type) as part of the method header. For example, the method header

boolean match(String keyName)

indicates that the `match` method takes a single `String` argument and returns a `boolean` value. In contrast to Scheme, the type of each method argument is declared in the method header.

1.1.4 Writing and Maintaining Tests

In Scheme we practiced test-driven development by writing tests for a program using the `check-expect` operation *before* we wrote the code for the program. In Java, software developers use testing frameworks to perform the same task. In fact, OO languages like Java are particularly well-suited to test-driven development because programs are expressed as a collection of small largely independent units of code, namely *classes*. In writing and maintaining a large program, the software developers need to devise and maintain tests for every non-trivial method and run *all* tests every time the code base (the files containing program text) is changed to ensure that the changes have not "broken" the program. Test are better expressed as classes where test methods silently return (assuming no failures) than they are as interactions histories where the output must be carefully expected to determine where or not the tests passed. JUnit only updates a progress bar as it runs as long as all tests pass.

When running all of the archived tests for a large program, the software developers only want to know when a test *fails*. If a program modification has been performed correctly, all of the archived tests will succeed. Hence, we want a test manager application that runs all the tests for a program and only reports the errors (failed tests) that it finds. The most widely used test manager (also called a *testing framwork*) for Java is called JUnit. JUnit is a conventional Java program that can be run from the command line but DrJava has integrated JUnit so that it can be conveniently run inside DrJava. Moreover, DrJava helps the programmer write unit test classes by generating templates for such classes.

Let's use DrJava to help us write a test class for the `Entry` class. The second command in the File menu is called New JUnit Test Case .... When we depress this menu item, DrJava creates a new class in a new file (not yet written to disk) after popping up a small window asking for the name of the class. Let's chose the name `EntryTest`. The new file initially contains the following text:

```
import junit.framework.TestCase;

/**
 * A JUnit test case class.
 * Every method starting with the word "test" will be called when running
 * the test with JUnit.
 */
public class EntryTest extends TestCase {
```

```
import junit.framework.TestCase;

/** A JUnit test case class for the Entry class. */
public class EntryTest extends TestCase {

  /** Tests the match method. */
  public void testMatch() {
    Entry e = new Entry{"Corky", "DH 3104", "x 6042");
    assertTrue("match should succeed", e.match("Corky"));
    assertTrue("match should fail", ! e.match("Henry");
  }
}
```

Figure 1.2: A JUnit test class for the `Entry` class

```
  /**
   * A test method.
   * (Replace "X" with a name describing the test.  You may write as
   * many "testSomething" methods in this class as you wish, and each
   * one will be called when running JUnit over this class.)
   */
  public void testX() {
  }
}
```

Most of the generated text is comments instructing us what to do. Since we have only defined the equivalent of a Scheme `define-struct` statement augmented by one explicitly defined method `match`, there is little to test. We create two test cases for `match`, including one where the match should succeed and one where it should fail.

The `assertTrue` method takes two arguments: a `String` name used by JUnit to identify the test when reporting a failure and a boolean expression that evaluates to `true` iff the test succeeds.

If we run this test class using the unit test runner provided by DrJava it will report that all test passed successfully. Of course, our only test method, named `testX` at this point, does nothing, but it succeeds because it does not contain any failing tests. As soon as we define some methods to process `IntLists`, we will define some corresponding test methods, renaming our vacuous `testX` method in the process.

The first line of the `EntryTest` file contains an `import` statement that tells Java to include the class `junit.framework.TestCase` in the compiled program. DrJava includes all of the libraries packaged with the `Java JDK` plus the JUnit library, which is developed, maintained, and distributed by `junit.org`, much as `DrJava` is developed, maintained, and distributed by `drjava.org`, which is hosted on the Rice University Computer Science Department web server.

Finger Exercise: In the DrJava programming environment at Elementary Language Level, open the file `Entry.dj0` and create a test class `EntryTest` using the DrJava New JUnit Test Case command that looks like the class in Figure 1.1.4. Run this test class, make sure that it works, and save is as file `EntryTest.dj0`. ■

The `assertTrue` method is very well-suited to testing operations that return a *boolean* value. But in many cases, we need to test method invocations that return values that are not `boolean`. Such tests can clearly be expressed by `assertTrue` invocations (using `equals`) but the `assertEquals` method is more convenient. `assertEquals` takes three arguments: a `String` name, the expected answer, and the expression being evaluated as a test. Since Java is a staticallly typed language, there are nine different versions of `assertEquals` so the method (family) can be to check a result of any type (any of the eight primitive types plus `Object`.

1.2 Java Mechanics

1.2.1 Notation and Syntax

In Scheme, programs are constructed from expressions. In contrast, Java programs are constructed from *statements*. Statements are program phrases that *do not have values*, similar to Scheme expressions like `(define ...)` that return the dummy value `(void)`. Nevertheless, many Java statements contain embedded expressions, so let us look briefly at Java expressions.

1.2.2 Java Expressions

In Java, arithmetic, boolean, and `String` expressions are written in conventional mathematical *infix* notation, adapted to the standard computer character set (called ASCII). For example, the Scheme expression

```
(and (< (+ (* x x) (* y y)) 25) (> x 0))
```

is written in Java as

```
(x*x + y*y < 25) && (x > 0)
```

The syntax of Java expressions is patterned after the C programming language. Like C, Java uses the symbol `&&` for the "and" operation on boolean values (`true` and `false`) and the symbol `==` for the equality operation on numbers. (Warning: the symbols `&` and `=` are used in `C` and Java for other purposes. You have already seen the primary use of the of `=` symbol, which is defining the values of variables.)

The following table lists the major infix operators provided by Java:

`+`	addition and `String` concatenation
`-`	subtraction
`*`	multiplication
`/`	division
`%`	mod (remainder from integer division)
`<`	less than
`<=`	less than or equal
`>`	greater than
`>=`	greater than or equal
`==`	equal
`!=`	not equal
`&&`	and
`||`	or

The Java arithmetic operators all perform the indicated operations using *computer arithmetic* instead of genuine arithmetic. Computer arithmetic does not exactly conform to the standard mathematical conventions. Calculations involving real numbers (Java types `float` and `double`) are approximate; the computer rounds the true result to the nearest real number expressible using the number of digits provided in the standard machine representation (scientific notation with a fixed number of digits for the fraction and exponent). Integer calculations are done exactly provided that the answer is an integer and that it can be represented using 31 binary digits plus a sign.[1] Note that integer division always produces integer answers (unless you try to divide by zero which is an error). For example, the expression

```
5/3
```

produces the result

```
1
```

which is the quotient of 5 divided by 3. Integer division truncates the true rational result, dropping the digits to the right of the decimal point. Similarly, The expression

```
5%3
```

produces the result

[1]As we will explain shortly, Java supports several different sizes of integer representation; 31 binary digits plus sign is the default for integer constants.

```
2
```

which is the remainder of 5 divided by 3. In Java program text, spaces between symbols are ignored; the expression

```
5 / 3
```

is equivalent to the expression

```
5/3
```

Finger Exercise: In the DrJava programming environment, try evaluating the following expressions in the Interactions pane:

```
5/3
5 % 3
5./3.
5 / 0
5./0.
5 < 6
5. < 6.
3 + .1 * .1 - 3.
```

Did you get the answers that you expected? ■

All of the binary infix operators in Java are either arithmetic {`+`, `-`, ...} relational {`<` `==`, ...}, or boolean {`&&`, `||`, ...} except for `+` when it is used to concatenate strings. If either argument to `+` is of `String` type, then Java converts the other argument to a `String`. Object values are coerced to type `String` using their `toString()` methods. As we explain in Section 1.13.1, *every* object has a `toString()` method. The concatenation operator converts primitive values to strings using built-in conversion routines that we will discuss later.

The order in which arguments appear and the use of parentheses in mixed integer and string expressions critically affects the conversion process. For example, the expression

```
9 + 5 + 1 + "S"
```

evaluates to the `String` `"15S"` while the expression

```
9 + (5 + (1 + "S"))
```

evaluates to the `String` `"951S"`. The association rules for Java expressions are explained in Section 1.2.3.

Java also supports the unary prefix operators `-` (arithmetic negation) and `!` (boolean "not") used in conventional mathematical notation. Parentheses are used to indicate how expressions should be decomposed into subexpressions.

Finger Exercise: If the DrJava Interactions pane, try evaluating the following expressions:

```
-5 + 3
-(5 + 3)
! (5 < 6)
```

■

Conditional Expressions

The only pure expression form in Java that deviates from conventional mathematical notation is the conditional expression notation

test `?` *consequent* `:` *alternative*

borrowed from C. This expression returns the value of *consequent* if *test* is `true` and the value of *alternative* if *test* is `false`. It corresponds to the Scheme `cond` expression

`(cond [`*test consequent*`] [else `*alternative*`])`

or, alternatively,

`(if `*test consequent* `[`*alternative*`])`

Note that when *test* is `true`, *alternative* is not evaluated. Similarly, when *test* is `false`, *consequent* is not evaluated. Hence, the expression

```
(2 < 0) ?  2/(1 - 1) :  0
```

does not divide 2 by 0. The *test* expression must be a boolean value, `true` or `false`.

Finger Exercise: In the DrJava Interactions pane, try evaluating the following expressions:

```
(2 < 0) ? 2/(1 - 1) : 0
(0 < 1) ? "foo" : "bar"
17 ? true : false
```

The last example produces a syntax error because `17` is not a boolean value. ■

Conditional expressions are hard to read unless they are short and cleanly formatted. Moreover, in many cases the code can be written more clearly using conventional `if` statements, which are explained in Section 1.4.4. In DrJava program text, conditional statements are forbidden at the Elementary and Intermedate levels.

1.2.3 Precedence of Operations

Since Java uses conventional infix notation for expressions it relies on the notion of *precedence* to determine how expressions like

```
12 * 5 + 10
```

should be interpreted. The Java operations given in the preceding subsection are divided into the following precedence groups:

prefix operators	`- !`
multiplicative	`* / %`
additive	`+ -`
relational	`< > >= <=`
equality	`== !=`
logical and	`&&`
logical or	`||`
conditional	`? ... :`

from highest to lowest. A higher precedence operator has greater "binding power". For example, the expression

```
72.  - 32.  * 1.8
```

is equivalent to

```
72.  - (32.  * 1.8)
```

because `*` has higher precedence than infix `-`.

Finger Exercise: In the DrJava Interactions pane, try evaluating the following expressions:

```
72. - 32. * 1.8
(72. - 32.) * 1.8
```

■

All of infix operators listed above are *left-associative*: when infix operators of equal precedence are chained together, the leftmost operator has precedence. For example,

```
72.  - 30.  - 12.
```

is equivalent to

```
(72.  - 30.)  - 12.
```

Parentheses can be used to override the built-in precedence and associativity of operators. Hence,

```
    (72.  - 32.)  * 1.8
```

equals `40*1.8`. Similarly,

```
    72.  - (30.  - 12.)
```

equals

```
    72.  - 18.
```

It is a good idea to use parentheses if you have any doubts about the precedence relationship between consecutive operators. The judicious use of parentheses can make complex expressions easier to read.

Finger Exercise: In the DrJava Interactions pane, try evaluating the following expressions:

```
    72. - 30. - 12.
    72. - (30. - 12.)
```

■

1.2.4 Java Statements

Since the Java expression language is not very rich, Java programs express computations as sequences of statements that bind variables and fields rather than as compound expressions. The most common form of Java statement is an assignment statement

type *var* = *expr* ;

where *type* is a Java type name, *var* is a Java variable name, and *expr* is an expression of type compatible with the type of *var*. The assignment statement

```
    int x = 5;
```

asserts that "the variable x has value 5".

Finger Exercise: In the DrJava Interactions pane, try evaluating the following statements and expressions:

```
int x = 5;
x*x
double d = .000001;
double dd = d*d;
dd
dd*dd
1. + dd
1. + dd*dd
```

Did you get the answers that you expected? ■

Java variable names and type names must be *identifiers*. An *identifier* is any sequence of "alphanumeric characters" (letters, digits, and _) beginning with a letter or _—except for the following *keywords*, which are reserved and may not be used as variable names or type names:

```
abstract    default     if          private     throw
boolean     do          implements  protected   throws
break       double      import      public      transient
byte        else        instanceof  return      try
case        extends     int         short       void
catch       final       interface   static      volatile
char        finally     long        super       while
class       float       native      switch
const       for         new         synchronized
continue    goto        package     this
```

Java is *case-sensitive*; the variable `X` is distinct from the variable `x`. There are three kinds of variables in Java: class *fields*, *method parameters*, and *local variables*. Fields and method parameters are discussed in detail in the next subsection. We will defer a discussion of local variables until Section 1.10.4.

Java includes all of the basic statement forms found in the C programming language expressed in essentially the same syntax. In the remainder of this monograph, we will introduce these statement forms as they are needed. Although Java accepts most C syntax, many common C constructions (such as embedding an assignment inside in an expression) are considered bad style in Java.

1.2.5 Capitalization and Commenting Conventions

By convention, Java programs are written entirely in lower case characters with three exceptions.

- The first letter of class names are capitalized to distinguish class names from member names.

- The first letter in each word of a *multi-word* identifier after the first is capitalized. For example, the built-in Java class Object includes a method called `toString()` that we will discuss later. The capital `S` signifies the beginning of a word within the multi-word name `toString()`.

- The names of constant `static final` fields (discussed in Section 1.6) are written entirely capital letters. We have not yet discussed `static` fields or the `final` attribute because neither feature is supported in the Elementary language level of DrJava and `final` is not supported even in the Intermediate language level. We will introduce the `static` attribute in Section 1.6 and the `final` attribute in Section 1.13.5.

These conventions are not enforced by Java compilers, but it is considered bad style to violate them. A related convention is to never use the special character `$` in a name; this character is reserved for the use of the Java compiler. Unfortunately, most Java compilers do not enforce this convention.

Java relies on same commenting conventions as those in C/C++. A comment that is confined to a single line begins with the character sequence `//` and ends at the end of the line. Longer comments must be enclosed between the opening "bracket" `/*` and "closing" bracket `*/`. Examples of both form of comments appear in Section 1.4. Note that a bracketed comment can appear in the *middle* of a line of code.

Finger Exercise: add both forms of comment to the `Entry` class from the preceding exercise. ■

1.3 Java Data Types

Java programs manipulate two fundamentally different kinds of values: *primitive* values and *object* values.

1.3.1 Primitive Types

All primitive values belong to one of eight primitive types: `int`, `float`, `boolean`, `char`, `byte`, `short`, `long`, and `double`. Four of these types designate different sizes of bounded integers:

- `byte` contains the integers ranging from -128 to 127;

- `short` contains the integers ranging from -32768 to 32767;

- `int` contains the integers ranging from -2147483648 to 2147483647; and

- `long` contains the integers ranging from

  `-9223372036854775808`

  to

  `9223372036854775807`.

In practice, only three of these primitive types are widely used: `int`, `boolean`, and `double`.

In many programming contexts, it is convenient to treat primitive values and object values uniformly. Java accommodates this practice by providing a built-in wrapper class for each primitive type. A wrapper class consists of single field containing a corresponding primitive value plus a variety of methods for manipulating wrapped objects and the corresponding primitive values. The wrapper class corresponding to each primitive type is given in the following table:

Primitive Type	Wrapper Type
`int`	`Integer`
`float`	`Float`
`boolean`	`Boolean`
`char`	`Character`
`byte`	`Byte`
`short`	`Short`
`long`	`Long`
`double`	`Double`

Note that the wrapper class names follow a uniform naming convention (the primtive name with a capitalized first letter), *except* for `Integer` and `Character`. In most contexts, Java automatically converts between corresponding primitive and wrapped values. A program can explicitly create a wrapped value by using the constructor invocation

`new` *wrapperClassName*`(` *primitiveValue* `)`

where *wrapperClassName* is any wrapper class name and *primitiveValue* is an expression denoting a corresponding primitive value. For example,

```
new Integer(0)
```

creates an `Integer` object corresponding to the `int` value `0`.

The `boolean` type has two values `true` and `false`. The `char` type supports the Unicode character set which includes all conventional ASCII characters plus almost any foreign character imaginable. The `char` type is rarely used in Java programs because a flexible `String` object type is built-in to the language. The remaining two types `float` and `double` are used for approximate computations involving real numbers; they denote standard IEEE 32-bit and 64-bit formats, respectively.

Numeric Constants Java interprets unadorned integer constants as values of type `int`. Long integer constants are indicated by attaching the suffix `L` to the number. For example, the constant `9223372036854775807L` can be used in Java program text, while the same constant without the `L` is an error because it is too big to be an `int`. The `double` type is the default type for any floating point constant. On modern machines, there is little reason to use the less precise `float`.

Conversions Between Types Java will automatically convert any numeric type to a more "general" numeric type demanded by context. The following list gives the primitive numeric types in increasing order of generality:

$$\texttt{byte} \rightarrow \texttt{short} \rightarrow \texttt{int} \rightarrow \texttt{long} \rightarrow \texttt{float} \rightarrow \texttt{double}$$

Note that the notion of generality here is imperfect in some situations. The conversion of a `long` to a `float`, for example, will often lose precision. In fact, even the conversion of a really large `long` value to a `double` can lose precision.

Java provides explicit conversion operators called *casts* to convert a numeric type to a less general type. A cast is simply a type name enclosed in parentheses used a prefix operator. For example, the following expression casts the `int` constant `127` to the type `byte`

```
(byte)127
```

When converting from one bounded integer type to another, Java silently truncates leading digits if the output type is shorter than the input type. Watch out!

Finger Exercise: In Java, the maximum `long` value is denoted `Long.MAX_VALUE`. (This notation will be explained in detail in Section 1.6.1. In DrJava Interactions pane, cast the value of `Long.MAX_VALUE` to `double`. What do you get? Cast this value back to type `Long`. Did the you get back your original result? What is going on?

Try casting `Long.MAX_VALUE - 511` to `double` and back to `long`. Do you get the same result? How does this result compare to `(long) (double) Long.MAX_VALUE`. Try casting the maximum `Long.MAX_VALUE - 511` to `double` and back to `long`..

Cast `Long.MAX_VALUE` to type `float` and back to `long`. Do you see a pattern emerging? Try casting `Long.MAX_VALUE - Integer.MAX_VALUE` to `float` and back to `long`. A `float` has a fraction consisting of 23 binary digits plus a sign bit while a `long` has 63 binary digits plus a sign bit. Subtracting `Integer.MAX_VALUE` from `Long.MAX_VALUE` simply clears (zeroes) the rightmost 31 binary digits of `Long.MAX_VALUE` and hence has no impact on its value when converted to a `float`.

Java accepts integers expressed in hexadecimal (base-16) notation if they are preceded by the prefix `0x`. Hence, `Integer.MAX_VALUE == 0x7FFFFFFF` and { `Long.MAX_VALUE - Integer.MAX_VALUE == (Long.MAX_VALUE >> 31) << 31.` } Check this cacula-tion in the Interactions pane. The expression `(x >> 31) << 31` for any `long` value `x` clears the rightmost 31 bits of `x`. How many digits of `Long.MAX_VALUE` can you clear (using shift operations) before casting to `float` affects its value? ■

1.3.2 Object Types

Object values are created by instantiating classes, which may either be built-in or program-defined. Classes are organized in a strict hierarchy with the special *built-in* class `Object` at the top. Every class C except `Object` has a unique parent in the hierarchy called the *superclass* of C. `Object` has no parent because it is the *root* of the object type hierarchy.

A descendant in the class hierarchy is called a *subclass*. Each subclass of a class C includes all of the members (fields and methods) of C and possibly additional members. For this reason, we say that each immediate subclass of (child of) C *extends* C. Note that *subclass* relation is *transitive* and *reflexive*. If class A is a subclass of class B and class B is a subclass of class C then A is a subclass of C (transitivity). In addition, every class A is a subclass of itself (reflexivity).

We will reserve the term *extends* to describe the immediate subclass (child) relation: A extends B iff A is an immediate subclass of B. Hence, the *extends* relation is neither transitive nor reflexive. Since `Object` is the top class of the hierarchy, all classes are subclasses of `Object`.

For example, the built-in classes `Integer` and `Float` extend the built-in class `Number` which extends `Object`. Hence, the superclass of `Integer` is `Number`, the superclass of `Float` is `Number`, and the superclass of `Number` is `Object`.

Object values are actually *references* to objects. For this reason, two different fields can be bound to *exactly the same object*. In Java, objects are never implicitly copied. When a field or method parameter v is bound to an object o, the value of v is a reference to the object o, not a copy of o! Scheme follows *exactly* the same conventions with regard to copying program data.

Every Java class C has an associated type C consisting of all instances of class C and all of its subclasses. Hence, the type `Object` contains all object values. The built-in class `String` has the class `Object` as its superclass. Since the class `String` has no subclasses, the only values of type `String` are instances of the class `String`. In contrast, the built-in class `Number` is a child of class `Object` and has several subclasses including `Integer` and `Float`. Hence, all instances of the classes `Integer` and `Float` are values of type `Number`.

In Java, every field and method has a declared type given as part of its definition. For a method, the declared type includes the type of the result and the types of the parameters.

Java determines the type of every program expression using a simple set of rules and confirms that

- the value assigned to a field is consistent with the field's declared type;
- the values passed as arguments to a method are consistent with the corresponding parameter types;
- the value returned by a method is consistent with the declared return type of the method; and
- the member name in a field reference or method invocation is compatible with the declared type of the receiver.

We will discuss these "type-checking" rules in more detail in Section 1.10.3.

1.4 The Union and Composite Patterns

In our department directory example, an object of type `Entry` only has one form, namely an instance of class `Entry`. If we were designing the data for a more comprehensive directory such as a city phone directory, we would need more than one form of entry. At a minimum, we would need entry formats suitable for business listings, government listings, and residential listings. For such a phone directory, we might define an entry as follows.

A `CityEntry` is either:

- a `BusinessEntry(name,addr,phone,city,state)`,
- a `GovernmentEntry(name,addr,phone,city,state,gov)`, or
- a `ResidentialEntry(name,addr,phone)`,

where `name` is a string specifying the name for the listing, `addr` is a string specifying the street address for the listing, `phone` is a string specifying the phone number (with area code) for the listing, `city` and `state` are strings specifying the city and state for the listing, and `gov` is a string specifying the government entity for that the listing, *e.g.* the "City of Houston".

The `BusinessEntry` and `GovernmentEntry` forms include city and state information because businesses and government agencies that serve clients in cities outside their local calling area often elect to have their phone numbers included in the directories of other cities (in addition to the cities where they are located). In addition, government listings include a string specifying the government entity to which they belong. For example, a listing for the Federal Bureau of Investigation would specify the "U.S. Government" as the `gov` field.

In Scheme, we would represent such an entry data type by defining three different structures:

```
;; a CityEntry is either:
;;   a BusinessEntry
;;     (make-BusinessEntry name addr phone city state),
;;   or a GovernmentEntry
;;     (make-GovernmentEntry name addr phone city state gov),
;;   or a ResidentialEntry
;;     (make-ResidentialEntry name addr phone).

(define-struct BusinessEntry (name addr phone city state))
(define-struct GovernmentEntry (name addr phone city state gov))
(define-struct ResidentialEntry (name addr phone))
```

Note that the type `CityEntry` consisting of the union of the types `BusinessEntry`, `GovernmentEntry`, and `ResidentialEntry` is *not* defined in program text because all union types in Scheme are implicit.

In Java, we can define the `CityEntry` type by introducing a "dummy" `CityEntry` class that we extend by "concrete" classes[2] for each different form of `Entry` data. The `CityEntry` class is `abstract` because there are no objects in our data model that belong to `City Entry` but not to one of the subtypes .... If `CityEntry` is concrete, then it is *not* the union of `BusinessEntry`, `GovernmentEntry`, and `ResidentialEntry`. This technique, which is widely used in object-oriented programming, is called the *union pattern.* In this pattern, an abstract class serves as the root of a hierarchy of subclasses called *variants*, which are the component types of the union. In this example, there are three variant classes: `BusinessEntry`, `GovernmentEntry`, `ResidentialEntry`. The following Java code defines the city-entry type:

Note that each concrete class includes *exactly* the same fields as the corresponding Scheme structure definition. The pivotal difference between the Java code and the corresponding Scheme code is the presence of the abstract class `CityEntry` in the Java Code identifying the union type which is left implicit in Scheme.

The Java code in the `CityEntry` example above involves two concepts that we have not discussed before.

- The attribute `abstract` attached to the class `CityEntry` indicates that `CityEntry` is a "dummy" class *that cannot be instantiated.* The class `CityEntry` is defined solely to group the concrete classes `BusinessEntry`, `GovernmentEntry`, and `ResidentialEntry` as a type that includes all three kinds of data.

- The concrete classes `BusinessEntry`, `GovernmentEntry`, and `ResidentialEntry` are the only *subclasses* of the class `CityEntry`. Hence, the only values of type `CityEntry` are the instances of the classes `BusinessEntry`, `GovernmentEntry`, and `ResidentialEntry`.

[2]Any class that is not declared as `abstract` is "concrete".

```
/**  a CityEntry is either:
 *   (i)   a BusinessEntry
 *         new BusinessEntry(name,addr,phone,city,state),
 *   (ii)  a GovernmentEntry
 *         new GovernmentEntry(name,addr,phone,city,state,gov), or
 *   (iii) a ResidentialEntry
 *         new ResidentialEntry(name,addr,phone).
 */
abstract class CityEntry { }

class BusinessEntry extends CityEntry {

  /* fields */
  String name;
  String address;
  String phone;
  String city;
  String state;
}

class GovernmentEntry extends CityEntry {

  /* fields */
  String name;
  String address;
  String phone;
  String city;
  String state;
  String government;
}

class ResidentialEntry extends CityEntry {

  /* fields */
  String name;
  String address;
  String phone;
}
```

Figure 1.3: The `CityEntry` union class hierarchy

The following expression creates a `BusinessEntry` for Rice University

```
new BusinessEntry("Rice University", "6100 Main Street",
"713-348-8101", "Houston", "Texas")
```

This syntax is wordy but straightforward. Note that Java requires the keyword `new` in front on each constructor invocation.

Finger Exercise Enter the four preceding class definitions into the Definitions pane of DrJava and save this program in the file `CityEntry.dj0` Compile this program and evaluate the following expressions in the Interactions pane:

```
BusinessEntry e1 = new BusinessEntry("Rice University", "6100 Main St.",
  "713-527-8101", "Houston", "TX");
ResidentialEntry e2 = new ResidentialEntry("Robert Cartwright",
  "3310 Underwood St.", "713-660-0967");
e1.name()
e2.name()
```

Did you get the results that you expected? ■

1.4.1 Defining Instance Methods on a Union Type

Let us define a method for class `CityEntry` that makes sense for all instances of the type `CityEntry`. Specifically, let us define a method `nameStartsWith(String prefix)` that determines whether a putative `prefix` is actually a prefix (matches an intial segment) of `name`. Fortunately, the `String` class is very rich and includes a method called `startsWith(String prefix)` that determines if a putative `prefix` is actually a prefix of the receiver. Hence, we can define such a method in each concrete subclass of `CityEntry` by adding the following text to the body of each concrete class:

```
/** Returns true if prefix is actually a prefix of the field name. */
boolean nameStartsWith(String prefix) { return name.startsWith(prefix); }
```

The complete code for the expanded `ResidentialEntry` class appears in Figure 1.4.

If we add this code to each of the three subclasses, then `nameStartsWith` is defined on receivers with types that match one of the three subclasses, but is *not* defined on type `CityEntry`. Why? Because `nameStartsWith` is not a member of the class `CityEntry`. Trying to execute `o.nameStartsWith(‘‘Cor’’)` for some variable/expression `o` of type `CityEntry` is a static type error. To rectify this problem, we must define `startsWith(String prefix)` as an `abstract` method of `CityEntry`. An `abstract` method is a method header without a body. The code for the expanded

```
class ResidentialEntry extends CityEntry {

  /* fields */
  String name;
  String address;
  String phone;

  boolean nameStartsWith(String prefix) { return name.startsWith(prefix); }
}
```

Figure 1.4: The expanded `ResidentialEntry` class with method `nameStartsWith`
.

```
abstract class CityEntry {

  abstract boolean nameStartsWith(String prefix);
}
```

Figure 1.5: The `CityEntry` class with method `nameStartsWith`

`CityEntry` class is given in figure 1.5. Note that the definition of `nameStartsWith` could *not* have been hoisted into class `CityEntry` unless the `name` field was hoisted as well. The latter was a prerequisite to the former.

Finger Exercises

1. Open your file `CityEntry.dj0` in DrJava. Add a definition for `nameStartsWith` in each of three concrete subclasses and save your expanded file as `CityEntry.dj0`. Create a JUnit test class `CityEntryTest` to test the four classes forming the `CityEntry` union, make sure that it works, and save it as `CityEntryTest.dj0`. ■

2. Make sure that your expanded file (from the preceding exercise) `CityEntry.dj0` is open in the DrJava Definitions pane. Add the declaration for the `abstract` method `nameStartsWith` to the `abstract` class `CityEntry`. Add additional tests to the class to confirm that the `nameStartWith` method for type `CityEntry` works and save the expanded test file as `CityEntryTest.dj0`. ■

1.4.2 Member Hoisting

The preceding Java program can be improved by eliminating duplicated code in the subclasses extending `CityEntry`. The concrete classes forming a union are called *variants*. Note that the fields `name`, `address`, and `phone` and the same method definition for `nameStartsWith` appear in all three variants of the abstract class `CityEntry`. These repeated member definitions can be *hoisted* into the abstract class `CityEntry` yielding the Java code shown in Figure 1.6

Note that we cannot hoist the `nameStartsWith` method definition without hoisting the field `name`. Otherwise, the definition of `nameStartsWith` would be rejected as ill-formed by the compiler because `name` is not defined. **Finger Exercise** Open your saved file `CityEntry.dj0` and revise the code to hoist the members `name`, `address`, `phone`, and `nameStartsWith`. Replay the tests stored in `CityEntryTest.dj0` to confirm that your refactoring was successful. Note that your refactoring changed the implementation of `CityEntry` without changing its visible behavior. Save the revised program file as `CityEntry.dj0`.

Evaluate the following statements/expressions in the Interactions pane.

```
CityEntry e1 =
  new ResidentialEntry("John Doe", "3310 Underwood", "713-664-4873");
CityEntry e2 =
  new ResidentialEntry("Jane Smoth", "3210 Drummond", "713-666-8809");
```

```
abstract class CityEntry {

  /* fields */
  String name;
  String address;
  String phone;

  boolean nameStartsWith(String prefix) { return name.startsWith(prefix); }
}

class BusinessEntry extends CityEntry {

  /* fields */
  String city;
  String state;
}

class GovernmentEntry extends CityEntry {

  /* fields */
  String city;
  String state;
  String government;
}

class ResidentialEntry extends CityEntry { }
```

Figure 1.6: The `CityEntry` class with hoisted members

```
    GovernmentEntry e3 =
      new GovernmentEntry("City Hall", "901 Bagby", "713-837-0311", "Houston",
      "Texas", "Houston City");
    e3.government()
    CityEntry e4 = e3;
    e4.government()
    e1.address().equals(e2.address())
    e3.nameStartsWith("City");
    e2.nameStartsWith("Jane");
    e1.nameStartsWith("John");
```

Did you get the results that you expected (including errors)? Add tests corresponding to the code that worked to the class `CityEntryTest` and save the revised file as `CityEntryTest.dj0`. ■

Finger Exercise In the preceding code, the abstract class `CityEntry` has three concrete subclasses: `ResidentialEntry`, `BusinessEntry`, and `GovernmentEntry`. By applying some very simple program transformations, you can eliminate the duplication of fields by inserting a new abstract class `NonResidentialEntry` between `CityEntry` and the concrete subclasses `BusinessEntry` and `GovernmentEntry` hoisting the common members of these concrete classes. Open the file `RevCityEntry.dj0` in DrJava and revise it by adding the abstract class *NonResidential Entry*. Save the revised program in the file `RevCityEntry.dj0`. After this revision, the class `CityEntry` still has only three *concrete* subclasses but only one of them is an *immediate* subclass. The other immediate subclass is `NonResidentialEntry`. Test your code using DrJava. Run your test cases in `CityEntryTest` to ensure that the revised code works. ■

Member hoisting is a special form of code *code factoring*. Code factoring is any transformation that eliminates repeated code. In functional languages like Scheme, the standard code factoring is typically accomplished by introducing a new λ-abstraction with a repeated code pattern as its body. Each instance of the repeated pattern is replaced by an appropriate call on the new abstraction. In some cases, the arguments to the pattern are procedures. This form of code factoring can be implemented in several different ways in Java. If all of the code repetitions appear within a class hierarchy for which the programmer has control of the source, then a method can be introduced in the most restrictive subclass that includes all of the occurrences of the repeated pattern. Each occurrence can be replaced by an invocation of the introduced method. In some cases, the arguments to the method are command or strategy objects (discussed in Section 1.9) representing procedures.

Java Design Rule: never repeat code in the variant classes in a union type. Hoist any repeated code into methods defined in an abstract superclass.[3]

1.4.3 The Composite Pattern

Let's return to our department directory example and show how to use the union pattern to represent department directory data.

A `DeptDirectory` is either:

- an `Empty` directory, or
- a composite directory `Cons(e, d)` where `e` is the `first Entry` of the directory and `d` is the `rest` of the directory.

In Scheme, we could implement this definition using the following collection of structures:

```
;; a DeptDirectory is either:
;;   the empty directory (make-Empty), or
;;   the non-empty directory (make-Cons Entry DeptDirectory)
(define-struct Empty ())
(define-struct Cons (first rest))

;; an Entry is (make-Entry String String String)
(define-struct Entry (name address phone))
```

Note that the preceding Scheme code leaves most of the corresponding data definition unstated! It never mentions the new type `DeptDirectory`. It does not express the fact that instances of `Empty` and `Cons` are elements of the type `DeptDirectory`, nor does it state that the `first` and `rest` fields of a `Cons` must be of type `Entry` and `DeptDirectory`, respectively. Similarly, it fails to state that the fields `name`, `address` and `phone` are all strings. In Scheme, we must compose comments to communicate this information. Unfortunately, since these comments are not code they are not checked for consistency.

In Java, each new type of data is represented by a class. Since the `DeptDirectory` type has two variants, we must use the union pattern to represent this type. The following collection of class definitions relies on the union pattern to define the

[3]In pathological cases, some repeated code may not be subject to factoring because of conflicts among possible factorings. For example, variants `A` and `B` may share code for method `m` while variants `B` and `C` share code for method `n`. In this case, either the code for method `m` or the code for method `n` can be hoisted but not both. More complex factoring methods are possible (using, for example, the strategy pattern discussed in Section 1.9), but they are typically not worth the complication. Some leading edge OO languages like Scala incorporate a restrictive form of multiple inheritance called traits that largely overcomes this problem.

DeptDirectory type. Since the DeptDirectory type is implemented by an abstract class, we will prepend the name DeptDirectory with the letter A to indicate that the class is abstract.

```
/**  A DeptDirectory is either:
 *   (i)  the empty directory new Empty(), or
 *   (ii) the non-empty directory new Cons(Entry,DeptDirectory)
 */
abstract class DeptDirectory {}

class Empty extends DeptDirectory {}

class Cons extends DeptDirectory {
  Entry first;
  DeptDirectory rest;
}
```

The Java code is slightly wordier, but it captures *all* of the information in the data definition. Moreover, the extra text in the Java code simply provides the type information that is missing in the Scheme code (and can only be specified in comments). The data definition comment in the preceding Java code helpful but technically it is redundant.

The class `Empty` contains no fields, just like the corresponding Scheme `struct`. The class `Cons` contains two fields `first` and `rest` akin to the two fields in the corresponding Scheme `struct`. Similarly, the `Entry` class contains three fields `name`, `address`, and `phone` just like the Scheme `struct Entry` given above. The abstract class `DeptDirectory` is extended by only two classes: `Empty` and `Cons`. Hence, `DeptDirectory` is the union of `Empty` and `Cons`.

When Unions are Composite The use of the `union pattern` in the `DeptDirectory` example has an extra feature not present in the preceding `CityEntry` example. One of the variants of the union class `DeptDirectory` includes a field of type `DeptDirectory` which makes the structure of the union class *self-referential*. Since self-referential structures are ubiquitous in Scheme (and other functional languages), this feature is not at all surprising to programmers familiar with functional programming. In the OOP ("object-oriented programming") community, which has strong historical ties to imperative programming, this feature is viewed as distinctive because it implies that methods that process the union class are naturally recursive. For this reason, the OOP community assigns a distinct pattern name, namely *composite pattern*, to the the special case of the *union pattern* where self-reference is present in the data definition. We will use this terminology in the remainder of the monograph.

The following expression creates a `DeptDirectory` containing the address and phone information for Corky and Zung:

```
new Cons(new Entry("Corky","DH 3104","x 6042"),
  new Cons(new Entry("Zung","DH 3098","x 3835"), new Empty()))
```

This syntax is wordy but straightforward. Don't forget to include the keyword `new` at the front of each constructor invocation!

1.4.4 Defining Instance Methods for a Composite Class

In Section 1.1.3, we showed how to define simple (instance) methods for the individual class `Entry`. But we did not show how to express operations that process all of the different forms of data defined by a composite hierarchy. Since each different form of data in a composite hierarchy is represented by a distinct concrete class, we can write a separate method definition for each kind of data.

Consider the following example. Assume that we want to define a method

```
String firstAddress(String name)
```

on `DeptDirectory` that returns the `address` for the first person in the directory if the directory is non-empty and the null reference `null` if it is empty. We can write separate definitions for the method `firstAddress` in the concrete classes `Empty` and `Cons` as follows:

```
class Empty {
  ...
  String firstAddress() { return null; }
}

class Cons {
  ...
  String firstAddress() { return this.first.address(); }
}
```

Now assume that `x` is a variable of type `DeptDirectory`. If we try to invoke the method `firstAddress` on `x`, Java will reject the code as erroneous because the class `DeptDirectory` does not contain a method named `firstAddress`. How can we enlarge the definition of `firstAddress` so that it applies to the class `DeptDirectory`?

The answer is that we declare the method `firstAddress` in the class `DeptDirectory` as an *abstract* method:

```
abstract class DeptDirectory {
  ...
  /* firstAddress() returns the first address in a DeptDirectory;
       it returns null if the DeptDirectory is empty */
  abstract String firstAddress();
}
```

An abstract method is a method *without* a body. Abstract methods can *only* appear in abstract classes. Any class containing an abstract method must be declared `abstract` because it cannot be instantiated. Every concrete class extending an abstract class must provide concrete definitions for the abstract methods it inherits. This rule guarantees that abstract methods are never attached to objects.

Let us illustrate the process of defining a method over a composite class hierarchy in more detail by defining a method

```
String findAddress(String name)
```

on `DeptDirectory` that finds the `address` for the person identified by `name`, assuming that `name` is in the directory.

Explicit If Statements

To write the `findAddress` method, we need to introduce a new form of statement which supports executing different code blocks *within a method* depending on the value of a boolean expression. Up to now, we have managed to support different execution paths through a program solely by *dynamic* dispatch: selecting the method body to execute based on the receiver of a method call. In principle, dynamic dispatch is the only *control* construct that we need in an object-oriented language, but expressing all branches in control flow (what code is excuted next) using dynamic dispatch is awkward and inconvenient.

In functional languages like Scheme and procedural languages like C, we express conditional code execution is expressed by an `if` construct that incorporates least embedded code fragments: a boolean *test* expression and two alternate code blocks.[4].

In Java, the syntax for an `if` statement is:

if *test*
consequent code block
else
alternative code block

In many cases, the code blocks are compound statments enclosed in braces.

In Java code that does not mutate variables each code block ends with a `return` statement.[5]

Let's return to defining the `findAddress` method to show how `if` statements can be used.

[4]The Schene cond construct takes additional boolean expressions and additional alternative code blocks, but it can be expanded into a sequence of `if` constructs

[5]Or a `throw` statement that raises an error condition. We discuss `throw` statments in Section 1.10.5.

Instance Methods with If Statements

To define the findAddress method for the DeptDirectory type, we must insert the following member somewhere in the class DeptDirectory

```
/** findAddress(s) returns the address of the person with name s or null if
  * no matching entry is found */
abstract String findAddress(String name);
```

The abstract modifier in the definition indicates that the definition only describes the input and output types for the method, not how it is implemented. Each concrete class extending DeptDirectory must provide a definition for findAddress that includes a code body describing how it is implemented.

The ordering of members within a class typically does not affect program behavior. Nevertheless, it is good programming practice to list class members in a consistent order. We recommend listing members in the following order: fields, constructors, methods. According to this convention, the abstract method findAddress should be the last member in the DeptDirectory class.

Second, we must provide a concrete definition of the findAddress method in each concrete subclass of DeptDirectory, namely Empty and Cons. Note that the composite pattern *guarantees* that a program includes code specifically to process each data variant. Moreover, in any variant containing a field *f* of parent type, the method typically invokes itself recursively on *f*. This approach to defining methods is the direct analog of *natural recursion* template for defining functions in Scheme. It is so common and so important that OOP community has labeled this recursive approach to defining methods on composite classes as a separate pattern, called the *interpreter* pattern, enriching the composite pattern.

Let us return to process of coding the findAddress method. By definition, there is no Entry in an Empty directory matching the name passed as an argument to findAddress. Hence, findAddress must return a value signaling failure. In Java, the most convenient choice for such a value is null, the reference to no object. All object values in Java are actually references, so the same object can simultaneously appear as the value of many different variables. Scheme follows exactly the same convention regarding structures. Java also provides the special value null, which is the reference to *no* object. The closest analog to null in Scheme is the special value (void). Java null should *only* be used to represent a special failure value. It should never be used to represent one of the alternatives in a data definition, *e.g.*, the empty DeptDirectory. The reason for this prohibition is simple: null is *not* an object. Any attempt to invoke a method on null will generate a run-time error aborting program execution.

The following code is an appropriate definition of the findAddress method in the Empty class.

```
String findAddress(String name) { return null; }
```

The definition of `findAddress` for `Cons` objects is the only interesting chunk of code in this entire example. Since a `Cons` object always contains an `Entry first` and a `DeptDirectory rest`, the `findAddress` method must check to see if the passed `name` matches the `name` field of `first` and, depending on the outcome, either return the value of the `address` or recur on `rest`.

The object-oriented method has exactly the same recursive structure as the corresponding function definition.

The method can simply be coded as follows:

```
String findAddress(String name) {
  if (name.equals(first.name())) return first.address();
  else return rest.findAddress(name);
}
```

Every class contains the instance method `equals` which takes a single argument of type `Object` and the instance method `toString` which takes no arguments./footnote The `Object` class at the top of the class hierarchy provides default definitions for these methods. In DrJava, these default definitions are automatically overridden in class definitions defined in either the Elementary or Intermediate level programs so that the Java `equals` method behaves exactly like the Scheme `equal?` function and the `toString` method generates a structural description of `this`. We discuss the default definition of `equals` in detail in Section 1.13.2. For a `String` object `name`, the `equals` method returns `true` if and only if the argument object contains exactly the same sequence of characters as `name`.

In the code above, the expression

```
name.equals(first.name())
```

invokes the `equals` method of object in field `name` on the argument

```
first.name().
```

The expression c

```
first.name()
```

invokes the `name` method of the `Entry` object in the field `first` to get its `name` field. Similarly, the expression

```
first.address()
```

invokes thSe `address` method of the `Entry` object in field `first` to get its `address` field; the expression

```
rest.findAddress(name)
```

invokes the `findAddress` method of the `DeptDirectory` object in the `rest` field on the object `name`.

Finger Exercises.

1. Enter the text for the `DeptDirectory` example in the DrJava Definitions pane. Create a test class `DeptDirectoryTest` for your program to make sure that your coding of `DeptDirectory` is correct. Save your `DeptDirectory` program and your test class `DeptDirectoryTest` in file named `DeptDirectory.dj0` and `DeptDirectoryTest.dj0`, respectively. ■

2. Add a method `findPhone` analogous to `findOffice` to the `DeptDirectoryClass` class and add new tests to `DeptDirectoryTest` to ensure that `findPhone` has been correctly coded. Remember to use the same design recipe that you learned in the context of Scheme. Save your program class and your test class in the files `DeptDirectory.dj0` and `DeptDirectoryTest.dj0`, respectively. ■

1.4.5 Conditional Statements

In the definition of the `findAddress` method, we used an *conditional statement* of the form:

```
if (test) then consequent else alternative
```

where *test* is an expression of boolean type and *consequent* and *alternative* are statements. Conditional statements are used to classify program values into disjoint sets or regions using logical tests that we call *claims*. In simple example given above, we distinguished between two claims:

```
name.equals(first.name)
```

and

```
!(name.equals(first.name))
```

1.4.6 Blocks

In Java, braces are to aggregate sequences of statements into individual statements. A sequence of statements

```
{
  s1;
  s2;
  ...
  sn;
}
```

enclosed in braces is called a *block*. A *block* is a form of Java *statement*. The other forms of statements that we have seen so far are *variable definitions*, *assignment statements*, *conditional statements*, and *method calls*.

Suppose that we wanted to print a message every time the findAddress method failed to match a name in a `DeptDirectory`. We need to add a statement to the `else` clause of our conditional statement in the body of `findAddress` in class `Cons`. We can accomplish this task by surrounding the `return` statement in the `else` clause with braces and inserting our print statement before the `return` statement as shown below:

```
String findAddress(String name) {
  if (name.equals(first.name()))
    return first.address();
  else {
    System.out.println(first.name() + " does not match");
    return rest.findAddress(name);
  }
}
```

Why not insert the print statement after the `return` statement instead of before?

1.5 Basic Program Design

In the preceding sections of this monograph, we studied a Java subset suitable for explicating the basic principles of Java program design. As you recall from your Scheme background, the process of program design can be broken down into six steps:

- Data Analysis and Design
- Contract and Header
- Examples
- Template
- Body
- Test

which we collectively call the *design recipe*. Let us examine each of these six steps in the context of writing object-oriented programs in Java.

1.5.1 The Design Recipe

Data Analysis and Design

Our first task in writing a program is to understand and define the data that the program will process. We must compile an inventory of the various forms of data that can appear as program inputs and outputs and determine how to represent their "relevant" properties as Java data objects. In scientific problem solving, this process is often called "data modeling" or simply "modeling". For each distinct form of data, we must define a Java class with fields that represent the relevant properties of each object of that form. We use the composite pattern to specify which different forms of data belong to the same more general category of data. In the preceding section, for example, we grouped the `Empty` department directory and non-empty `Cons` department directories together using the composite pattern to form the more general category `DeptDirectory`.

After we have decided on the form of our data model and the principal methods of each type, we write a skeleton program consisting of all the classes in our model. In each class, we provide a dummy definition specifying the header, type signature (contract in HTDP terminology), and contract (purpose statement in HTDP terminology) for each principal method in that class.

Class definitions in the "full" Java language are more general and flexible than Scheme `struct` definitions because they enable the programmer to determine (*i*) exactly which primitive operations, including constructors, the new form of data will support, (*ii*) how objects will be printed as strings, (*iii*) and the types of object fields and methods. In Scheme, the set of operations generated by a `struct` definition is rigidly determined and Scheme does not include any provision for specifying the types of `struct` fields and operations. The Elementary and Intermediate language levels in DrJava imitate Scheme by automatically generating constructors and accessors and appropriately overriding the `toString` and `equals` methods.

The extra generality provided by "full" Java comes at a price. In the absence of the Elementary and Intermediate language levels, a Java programmer must write far more text to define a class than a Scheme programmer does to define a comparable struct. Of course, some of the classes in a Java data definition, *e.g.*, the abstract class at the top of a composite class hierarchy, have no analog in Scheme, forcing the Scheme programmer to write explanatory comments instead.

It is a good idea to define a collection of examples for each concrete class in a data definition. These examples are best defined in an accompanying test class as explained in Section 1.1.4.

Contract and Header

Since Scheme does not accept type declarations for function parameters, a well-written Scheme program must include a header for each function containing this information. In contrast, Java *requires* type declarations for all method parameters. Hence, every syntactically correct Java program includes headers for all methods. Half of this programming step is mandated by the Java language!

On the other hand, Java does not mandate the inclusion of a contract stating (*i*) what if any additional *preconditions* (beyond the the types of the arguments) must hold for the method to be called, and (*ii*) what relationship exists between the inputs and output of a method. The latter is often called a *postcondition* or output condition for the method. Well-written Java programs include preconditions and postconditions for all methods other than the trivial methods forming the data definition, *e.g.* stock constructors and accessors.

For methods defined over composite types (abstract classes at the top of composite class hierarchies), the contract information should be attached to the abstract method definition in the top class.

Examples

Each class should include a collection of sample inputs and corresponding outputs for each method. If the body of a method contains more than one control path (*e.g.* a conditional statement), the collection of examples should include at least one example for each control path. The examples for each method should be included in an accompanying test class written using a standard testing framework such as `JUnit`.

Template

In Java, much of the template selection process is mandated by the object-oriented programming model. When a data type T consisting of several different forms of data is represented by a composite class hierarchy, each method m defined on T must be defined in the class `T` corresponding to `T`. With a few rare exceptions, the method definition for m in `T` must be `abstract`, forcing each concrete subclass of `T` to provide a definition for m. This decomposition corresponds to the `cond` template used to process (non-recursive) mixed data in Scheme. Moreover, the relevant data for each definition of m is simply the set of object fields in the class containing m!

The only features in the template for a Java method definition that are *not* dictated by the object-oriented programming model are the recursive method calls corresponding to circular references in the data definition. For any method m in a class C containing a object field f of type T where T is a supertype of C (*e.g.* C is a concrete class in a composite class hierarchy with T at the top), the body of m will usually invoke m recursively on each such field f. These recursive calls appear in the template. For a concrete example, see Section 1.5.2.

Body

The coding part of the design process consists of filling in the body of each method m in each concrete class C using the available object fields and the results of recursive calls from the template. In some cases, writing this code requires ingenuity. But in the vast majority of cases, the coding process is very easy given decomposition provided by the template.

Test

Each class `C` representing a data type definition[6] should be tested by evaluating each primitive operation for the data type on the sample data values defined for its input domain and other representative input values. This process is described in detail in Section 1.1.4. In the interim, you can perform this testing in the Interactions pane and archive your interactions history to save your tests.

1.5.2 An Extended Example: Lists

Let us study how the design recipe described above applies to the process of writing some simple programs involving lists of integers. Lists are ubiquitous in programming. The `DeptDirectory` type introduced in Section 1.4 is simply a specialized form of *list*. For reasons that will become clear later in the monograph, we will use the name `IntList` rather than `List` for our list type. As you should recall from your Scheme background, a `IntList` is either:

- an `Empty` list, or
- a composite list `Cons(e,l)` where `e` is the `first Object` of the `IntList`, and `l` is the `rest` of the `IntList`.

Compare this definition with the definition of `DeptDirectory` given earlier.

We can abbreviate the preceding definition using the following mathematical notation:

```
IntList := Empty + Cons
```

which states that the set `IntList` is the union of the sets `Empty` and `Cons`. The set `Empty` contains only empty objects objects created by `new Empty()` operations while

[6]For a composite class hierarchy, use the top class.

the set `Cons` contains all objects `Cons(o,l)` where `o` is any `int` and `l` is any element of type `IntList`.

Assume that we are given the task of writing a program to perform some standard operations on lists of integers such as computing the sum of the numbers in a list, computing the product of the numbers in a list, and sorting the members of a list into ascending order.

The following collection of Java classes provides a minimalist definition for the `IntList` type:

```
/** Composite Data Definition:
 *  IntList := Empty + Cons
 */
abstract class IntList { }

class Empty extends IntList { }

class Cons extends IntList {

  /* fields */
  int first;
  IntList rest;
}
```

These three classes form a conventional composite class hierarchy with `IntList` at the top.

Finger Exercise: Copy the code given above for `IntList` and save it in a file called `IntList.dj0`. Experiment with creating some `IntLists` and applying the accessors `first()` and `rest` to `Cons` lists. Note that variables have to be declared to type `Cons` rather than `Empty` if you want to apply the accessor operations to them. ■

Let us compare the `IntList` defined above with the built-in lists of Scheme. Scheme provides the primitive operations `empty`, `cons`, `first`, and `rest` which correspond to the Java operations `new Empty()`, `new Cons(...)`, `first`, and `rest`. The DrJava Elementary and Intermediate language levels automatically generate the primitive operations for `IntList` from the class definitions defining the IntList data type. For inductively defined types that are not built-in to Scheme, we can write `(define-struct ...)` statements, which direct Scheme to automatically generate the appropriate operations.

Scheme also provides recognizers for empty and compound lists called `empty?` and `cons?` which have no visible analog in the preceding Java program. We could try to define analogous operations in our Java data definition by including `abstract` boolean operations `isEmpty` and `isCons` in the abstract class `IntList` and define them appropriately in the concrete subclasses `Empty` and `Cons`. But this approach does not achieve our objective of defining operations equivalent to the Scheme recognizers, because the `isEmpty` and `isCons` methods are applicable *only* to objects of type `IntList`. In Section 1.6.3, we will discuss how operations equivalent to `empty?` and `cons?` can be expressed in Java, but these operations are seldom used in well-written Java code.

Scheme and DrJava appear very similar with regard to their support for inductive data definitions, but there are a few important differences. Most notably, the data constructors generated by Scheme do not restrict the types of their arguments; any value whatsover can be passed as the component of a constructed object. In inductive data definitions, we usually restrict the arguments of a contructor to a particular form. But Scheme does not provide any way to capture this information in `define-struct` statements. In contrast, Java *requires* type declarations for every field of an object and the Java compiler enforces the correct use of types.

We also re-iterate another important difference between Scheme and Java data definitions outside the scope of the DrJava Elementary and Intermediate languages. Java does not automatically generate any accessors, an overriding of the `equals` method, an overriding of the `toString` method, or any data constructors other than a degenerate 0-ary constructor if no other constructors are provided. The *programmer* is responsible for explicitly coding these operations, which provides added flexibilty at the cost of writing lots of boring code.

1.6 Static Fields and the Singleton Pattern

1.6.1 Static Fields

In addition to (*instance*) fields, a Java class can include `static` fields that are attached to the class rather than instances of the class. The `static` fields of a class are *not* included in the template used to create class instances. There is only one copy of a static field for an entire class—regardless of how many instances of the class are created (possibly none).

Since a `static` field *f* is attached to a class *C* rather than an instance *o* of *C*, it is usually referenced in program text by the expression:

```
    C.f
```

instead of

```
    o.f
```

but Java supports both forms of notation for static field references.

The primary use of static fields is to define constants relevant to a particular class. For example, the minimum and maximum values for each of the primitive numeric types are attached to the corresponding wrapper classes as the values of the static fields `MIN_VALUE` and `MAX_VALUE`. A program should always refer to these numeric values using the names of the appropriate `static` fields rather than raw numeric constants. For example, the maximum `int` value should be denoted by the field reference `Integer.MIN_VALUE` rather than `-2147483648`. Every wrapper class except `Boolean` includes the fields `MIN_VALUE` and `MAX_VALUE` bound to the minimum and maximum values for the type, respectively. For more information on the static fields included in the primitive wrapper classes, read the web page on-line documentation from Sun Microsystems and follow the links for each of the wrapper classes.

Finger Exercise: In DrJava set the language level to Intermediate and try typing the following expressions in the Interactions pane:

```
Integer.MIN_VALUE
Character.MAX_VALUE
Long.MAX_VALUE
```

■

1.6.2 Singleton Pattern

One of the most important uses of `static` fields is storing the canonical instance of a class that only needs a single instance. For example, the `Empty` subclass of `IntList` only needs one instance because the class has no (dynamic) fields.

```
class Empty extends IntList {
  ...
  static Empty ONLY = new Empty();
}
```

Instead of repeatedly allocating new instances of `Empty`, program text can simply refer to the canonical instance `Empty.ONLY`. This code pattern is called the singleton pattern because it constructs a single instance of the class.

The implementation of the singleton pattern shown above suffers from an annoying defect: the class definition for `Empty` does not prevent code in another class from creating additional instances of the `Empty` class. We can solve this problem by explicitly declaring the consructor for the `Empty` class making it `private`.

```
class Empty extends IntList {
  ...
  private Empty() {}
  static Empty ONLY = new Empty();
}
```

Then code outside of class `Empty` cannot perform the operation

```
new Empty();
```

A private member of a class *C* is only visible inside class *C*. We will discuss visibility modifiers in more detail in Section 1.13.3.

Finger Exercise: In DrJava, select the Intermedidate language level on the Language Level menu. Open your save file `IntList.dj0` in the Definitions pane. Modify the `Empty` class to use the singleton pattern and save your revised program in the file `IntList.dj1`. (You will have to change the extension in the popup window for the save command.) Open your test class `DeptDirectoryTest` saved in the file `DeptDirectoryTest` class and replace all occurrences of `new Empty()` by `Empty.ONLY`. Run the tests to ensure that your refactoring of `IntList.java` to use the singleton pattern is correct. ■

A Sample Program

We are now ready to define a simple program to sum the integers in a lists. We will write test cases (examples for the method) and add a definition of the method

```
int sum();
```

to each class in our composite class hierarchy for `IntList`.

Let's begin by writing the contract and header for `sum` in the abstract class `IntList`:

```
// IntList := Empty() + Cons(Object, IntList)

abstract class IntList {
  /** returns the sum of the numbers in this. */
  abstract int sum();
}
```

At this point, we could also insert dummy definitions of the `sum` method in the classes `Empty` and `Cons`. A good convention to follow is the return a simple "default" value of the return type, such as `0` for primitive numeric types, `false` for the `boolean` type, and a trivial object (of the specified type) for an object type.

Next we need to generate examples showing the expected behavior of the method. To accomplish this task, we create the following test class:

```
import junit.framework.TestCase;

/** The JUnit test case class for IntList. */
```

```
public class IntListTest extends TestCase {

  static oneElt  = new Cons(1, Empty.ONLY);
  static twoElts = new Cons(5, oneElt);
  static threeElts = new Cons(-10, twoElts);

  /** Tests the sum method on some simple IntLists. */
  public void testSum() {

    assertEquals("empty list", 0, Empty.ONLY.sum());
    assertEquals("(1)", 1, oneElt.sum());
    assertEquals("(5 1)", 6, twoElts.sum());
    assertEquals("(-10 5 1)", -4, threeElts.sum());
  }
}
```

If you provided dummy definitions for `sum` in each of the subclasses of `IntList`, then you can run the test at this point and confirm that they fail.

As the next step, we select and instantiate a template for writing the `sum` method:

```
class Empty {
  ...
  int sum() { ... }
}

class Cons extends IntList {

  int first;
  IntList rest;
  ...
  int sum() { ... first  ... rest ... rest.sum() ... ; }
}
```

which, in this case, is structural recursion on the inductively defined (composite) type `IntList`

Finally, we complete the coding process by filling in the bodies of the methods in the template:

```
// IntList := Empty() + Cons(Object, IntList)

abstract class IntList {
  /** returns the sum of the numbers in this. */
  abstract int sum();
}
```

```
class Empty {
  int sum() { return 0; }
}

class Cons extends IntList {

  int first;
  IntList rest;
  int sum() { return first + rest.sum(); }
}
```

To finish the design recipe, we run the tests in the `IntListTest` class using the JUnit test runner provided by `DrJava`.

Finger Exercises

1. In DrJava (at Intermediate Language Level), create the classes `IntList` and `IntListTest` described above and ensure that the tests work. Save your classes in the files in `IntListTest.dj1` and `IntListTest.dj1`, respectively. ■

2. Using the same design recipe, add a definition for the method `prod` that computes the product of a list of numbers, and test it. Note that all you have do is develop the method `prod` (including an appropriate contract [purpose statement in HTDP terminology]) and write and run tests for it. ■

1.6.3 Type Predicates and Type Casts in Java

Most programming languages do not have an analog of Scheme predicates like `empty?` because they do not have a universal type that contains all other types. Java is almost identical to Scheme in this regard. All object types are subtypes of the universal type `Object`. If we ignore the eight primitive types (which all have corresponding wrapper types in the `Object` type hierarchy), then the data models of Scheme and Java are essentially identical.

To test membership in any object type, Java provides a collection of postfix operators of the form

```
instanceof T
```

where `T` is any defined object type. Hence, given the preceding program defining type `IntList`, Java interprets the program expressions below as follows:

```
new Empty() instanceof Empty ⇒ true
new Cons(0, new Empty()) instanceof Empty ⇒ false
"A" instanceof Empty ⇒ false
```

The `instanceof` operator has the same precedence as the relational operators. (Although the second "argument" to `instanceof` must be a type name, the Java parser initially recognizes this argument as an expression.)

Finger exercise: Load your program saved in the file `IntList.dj1` into the DrJava Definitions pane. Add definitions for the methods `boolean isEmpty()` and `boolean isCons()` to class `IntList` *without* using the `instanceof` operator. In the Interactions pane try evaluating the following sequence of interactive computations:

```
IntList empty = EmptyIntList.ONLY;
empty
ConsIntList oneElt = new ConsIntList(1, empty);
oneElt
empty.isEmpty()
empty instanceof EmptyIntList
empty.isCons()
empty instanceof ConsIntList
oneElt.first()
oneElt.isEmpty()
oneElt instanceof EmptyIntList
oneElt.isCons()
oneElt instanceof ConsIntList
ConsIntList twoElts = new ConsIntList(0, oneElt);
twoElts.first()
twoElts.rest()
twoElts.rest().isCons()
twoElts.rest() instanceof ConsIntList
empty.first()
empty.rest()
"A".isEmpty()
"A" intanceof EmptyIntList
"A".isCons()
"A" intanceof ConsIntList
```

To accomodate static type checking, Java includes a second form of type predicate not present in Scheme called a *cast*. You may recall that Java includes operations for *casting* one *primitive* type to another. These primitive type casts convert values of one type to "corresponding" values of another type. The casting operations for object types have a *completely different* meaning; casting a value v to an object type T peforms an `instanceof` check on v! If the check returns `false`, then Java throws a

`ClassCastException` indicating that the cast failed. If this exception is not caught (see Section 1.12.3), Java aborts execution and prints an error message indicating which cast failed. In contrast, primitive type casts never fail!

If object type casts can only cause a program to abort execution, what good are they? Since the cast prevents execution from continuing if the `instanceof` test fails, the compiler knows that the result of object casting expression

(*T*) *e*

has type *T*. Consequently, the static type checker in the compiler assigns the static type *T* to this casting expression. By inserting object casting operations in a program, you can tell the static type checker that a particular expression has a narrower (more precise) type that the type that would otherwise be assigned by the static type checking rules.

Finger exercise: Load your saved file `IntList.dj1` into the DrJava Definitions pane. In the Interactions pane try evaluating the following sequence of interactive computations:

```
IntList empty = EmptyIntList.ONLY;
IntList oneElt = new ConsIntList("B", empty);
oneElt
oneElt.first()
((ConsIntList) oneElt).first()
oneElt.rest()
((ConsIntList) oneElt).rest()
```

Perform the equivalent sequence of membership tests as in the previous exercise using `instanceof` operators instead of the operations `isEmpty` and `isCons`.

Casting and `instanceof` Tests

Unfortunately, the Java type checker does *not* recognize the fact that uses of a variable `v` in the control flow path following the test `v instanceof T` must have type `T`. The type checker assumes all uses of a variable `v` have the declared type of `v` which is correct but excessively conservative.

1.6.4 Inheritance and the Composite Pattern

Up to this point, we have used methods in Java essentially like functions in Scheme. What makes object-oriented programming truly powerful is the ability to add new forms of data to our program *without modifying any old code.* For example, if we later decide to insert links to other directories as an additional form of `DeptDirectory` data, we can simply define a new subclass `Link` of `DeptDirectory` with a `subDir` field referring to the embedded directory (which can be searched using the `findAddress` method. The new class must define `findAddress` and `findPhone` for the new form of data that it introduces, but none of the existing classes requires any change whatsoever.

In defining a program extension, the added data does not have to be a new subclass of an abstract class like `DeptDirectory`. The new data can be a subclass of an existing concrete class. For example, we could extend our directory program by defining a class EntryWithPosition extending `Entry` with a `String` field `title` specifying the person's title (`"Graduate student"`, `"Instructor"`, `"Professor"`, `"Chair"`, *etc.*). No revision of the `Entry` class would be required. Unfortunately, to extract this information from an *Entry* using the programming techniques discussed so far, we would have to add a new method `findTitle` to the composite class hierarchy `DeptDirectory`—modifying existing code. We will introduce a design pattern, called the *visitor pattern* near the end of this chapter that eliminates this problem.

When a class *C* extends another class *D*, every instance (non-static) member *m* of *D* is automatically an instance member of *C* with one exception: if *C* redefines the method *m*[7] We say that the instance members of class *D* are *inherited* by the subclass *C*. The linguistic convention of automatically incorporating member definitions of a class in each of its subclasses is called *inheritance*. If an inherited method *m* is redefined in class *C*, we say that the new definition *overrides* the inherited definition.

Inherited fields cannot be overidden. If a class *C* defines a field with the same name as an inherited field, the inherited field is merely "shadowed" exactly as it would be by a local variable with the same name. The inherited field still exists in each instance of *C* and can be accessed by code in the body of *C* by casting the `C` object to its superclass type *D* before extracting the field.

Let's consider an example of how field shadowing works. If we define a field named `address` in a subclass `C` of `Entry` as defined in Section 1.1.1 above, the new `address` field *shadows* the inherited one within `C`. Instances of `C` contain two `address` fields, but only the one local to `C` is accessible using conventional field reference notation. The other can be accessed by casting `this` to type `Entry`. Field shadowing is bad programming practice and should be avoided if possible.

We have already made extensive use of a limited form of *inheritance* in the *composite pattern*. All of the variant classes of in a composite hierarhcy provide definitions for the abstract methods inherited from the abstract superclass. Recall the `DeptDirectory` and `IntList` programs. When an abstract method from an abstract class is overridden in a concrete subclass (*e.g.* `findAddress` from `DeptDirectory` in the classes `Empty` and `Cons`), the "missing" definition inherited from the abstract class is overridden. This special form of overriding is sometimes called *method exten-*

[7]The redefined method must have exactly the same name and input and output types as the method *m* that would have been inherited.

sion because the inherited meaning of the method (nothing) is *extended* rather than *modified*.

In well-written OO code, the overriding of abstract methods is far more common than the overriding of concrete methods. The latter is a perilous practice because it may break the contract for the overridden method. Whenever you override a concrete method, make sure that the new method obeys *all* of its inherited contracts. Typically the overriding of a concrete method add new functionality while preserving the old: it satisfies both a new contract, specific to the class containing the overriding, and all of the inherited contracts.

When a class C overrides a method m in its superclass D, code in the body of C can still invoke the overridden method m from D using the special notation

```
    super.m( ... )
```

The feature can be used to add pre-processing and post-processing code to an inherited method m. The overriding definition of m can check or transform its inputs, invoke `super.`m on the (possibly transformed) inputs in some cases, and possibly check or transform the results produced by `super.`m calls.

It is important to remember that all unqualified member references in inherited code *implicitly refer* to the implicit method argument `this` which is bound to an instance of the *inheriting* class! Hence, the meaning of a method call appearing in inherited code *changes* if the specified method has been overridden! In practice, this semantic convention gives the behavior that programmers expect for instances of subclasses, but it can occasionally produce surprises—particulary if the method overriding is the result of an accidental rather than an intentional name match.

If you follow the programming guidelines recommended in this monograph you can generally avoid the pathologies discussed in this subsection. According to our stylistic rules, field references should never span class boundaries (with the exception of visitor objects discussed below in Section 1.11). All fields should be `private`. The DrJava langauage levels facility automatically makes all fields `private`. Hence, shadowed members are only accessed through *accessor* methods, which makes them accessible via `super` calls if accessors with the same names are written for the shadowing fields. No casts are involved. In general, we strongly recommend that fields not be shadowed.

1.7 Using Classes to Enforce Invariants (Optional)

Some data objects have an associated invariant (boolean condition) which must be maintained for the object to be well-formed. For example, the elements in a sorted list must appear in increasing order (non-decreasing if duplicates are allowed). Similarly, the nodes of ordered binary tree must appear in ascending order in an *in-order* traversal of the tree. In many cases, a class can ensure that such an invariant always holds by enforcing a well-chosen interface.

Consider the example that we already cited: a sorted list. Can we define a class `OrdList` similar to `IntList` class that guarantees that all instances are sorted? The answer is yes, but we have to change the visible interface (members) supported by the `IntList` class. In particular, we cannot allow clients of the `OrdList` class to perform `new` operations on `OrdList`. To add an element to an `OrdList`, clients must use a method

```
OrdList insert(int f)
```

that inserts `f` in proper position in `this`.

The `Ordlist` class includes a binary constructor just like `IntList` except for the fact that it is `private`, implying that no code outside of class `Cons` can use it. This visibility restriction raises a minor problem: how can we write the `insert` method for the `Empty` subclass? The binary `Cons` constructor is not accessible! The answer is to define a second constructor for the `Cons` class that takes a single `int` argument and initializes the `rest` field to `Empty.ONLY`.

Finger Exercise Write a definition for the `OrdList` composite class hierarchy as described above. Test your code. ■

Exercise Load the sample program `IntList` into the `Definitions` pane. Define a subclass `OrdCons` extending `Cons` that guarantees that `first` precedes `rest.first` (assuming it exists). The `rest` field of a `OrdCons` node must be either `Empty` or `OrdCons`. Define a `sort` method for the class `IntList` that sorts a list converting all `Cons` nodes to `OrdCons` nodes. Test your code.

1.8 Interfaces

We have not yet discussed one of most important object-oriented features of Java, namely the notion of an `interface`. In essence, an interface is a special *lightweight* form of `abstract class`. We use the term *lightweight* to describe an abstract class with no fields and no concrete methods; the only members are `abstract` methods. The key difference between an interface and the corresponding abstract class is that a class or interface can have unlimited number of immediate superinterfaces but it can only have one superclass.

An interface definition has almost the same syntax as a class definition:

```
interface name {

... member declarations ...
}
```

The only significant difference is the keyword `interface` instead of `class`.

Some Java software developers follow a convention that interface names begin with "I" followed by a capitalized name. Hence, "IList" is the name of an interface, while "List" is the name of a class. Others follow a convention that interface names end with "I", but there is no generally accepted convention for distinguishing interface names from class names in the Java programming culture. In the Java libraries, for example, there is no way to tell a class from an interface based on its name.

Since most uses of a type defined by a third party (*e.g.*, any program library) do not depend on whether the type is a class or an interface, an "I" naming convention strikes is problematic. Moreover, the distinction between an interface and an abstract class provided by a third party is unimportant except when defining a new class that inherits from the interface or abstract class. The best argument for using an "I" naming convention for program-defined interfaces is that it provides a convenient naming scheme when interfaces are heavily used. In this regard, we prefix using "I" as a suffix rather than a prefix because English words commonly begin with "i" but almost never end with "i". For example, "IntegerI" is much easier to read and pronounce that "IIinteger". We will provide an example in Section 2.1.9 where an "I" naming convention for program-defined interfaces is useful later in this monograph.

A class extends exactly one superclass, but it can "implement" (inherit from) an arbitrary number of interfaces. A class definition can optionally include an `implements` clause immediately following the `extends` clause (assuming one is present) in the class header. Some sample programs using interfaces appear in 1.9.

The Elementary language level of DrJava does not include interfaces. Programs involving interfaces must be written at the Intermediate, Advanced of Full Java language levels.

1.8.1 Multiple Inheritance

Some object-oriented programming languages like C++ permit a class to have multiple superclasses. This form of inheritance, which is called *multiple inheritance*, is very controversial. While clever uses of multiple inheritance abound, the semantics of code and field inheritance with multiple superclasses is problematic because of name clashes between members and, worse, the possibility of inheriting the same class in more than one way. Recent programming language research suggests that there are better approaches to inheritance, such as *traits* and *mixins*, that combine the flexibility of multiple inheritance with the simplicity of single inheritance, but they are currently the subject ongoing research and experimentation.

Java supports *multiple interface inheritance*, which avoids the pathologies of general multiple inheritance. In multiple interface inheritance, a class can extend multiple lightweight abstract classes, but only one class that includes method code and fields. Since no method code or fields appears in the lightweight abstract classes, there is no problem with name clashes (since all interface methods are publicly visible) or inheriting the same interface in multiple ways. We will illustrate the utility of Java interfaces in the next section. At this point, all we can say is that the abstract classes `DeptDirectory` and `IntList` could be declared as interfaces instead if we were willing to make all of their methods `public` and delete the `test` method from `IntList`.

Since the abstract class `DeptDirectory` above does not contain any members, it could be trivially rewritten as follows:

```
interface DeptDirectory {}
```

This change forces a small change in the definition of any immediate subclass of `DeptDirectory`: the word `implements` must be used in the header instead of `extends`:

```
class Empty implements DeptDirectory  ...

class Cons implements DeptDirectory  ...
```

In short, a class `implements` an interface but `extends` a class.

Note that using an `interface` instead of a class at the root of a union or composite hierarchy has a *price*. Members that are repeated in all variants cannot be hoisted into an interface which cannot contain any dynamic fields or concrete methods. On the other hand, a variant of `DeptDirectory` can inherit fields and methods from another class if `DeptDirectory` is an `interface`. We generally recommend using abstract classes as the root classes in composite hierarchies because they frequently need to include concrete methods (hoisted code) and occasionally need to include fields (hoisted fields). On the other hand, some programs involve multiple composite class hierarchies that *share* concrete subclasses. In this case, you must define the root classes as interfaces or define two copies of the "shared" classes.

1.8.2 Implicit Polymorphism

A cardinal rule in Scheme programming is "never repeat code". If a program repeats essentially the same code in two or more places, then the programmer failed to identify a common abstraction that should only be written once. The repeated code sections can be replaced by calls on a procedure that defines the repeated operation. Any minor differences between the repeating code sections can be accommodated by passing appropriate arguments that "fill in" the differing expressions.

The same rule applies to Java, but the notational details are more cumbersome because (*i*) methods cannot be directly passed as arguments in Java and and (*ii*) Java is statically typed. Passing methods as arguments is such an important programming technique that object-oriented programmers have developed two design patterns, called the *command* and *strategy* patterns, respectivelyy, that enable Java programs to indirectly pass methods as parameters by embedding them in "dummy" objects called *commands*, *strategies* or *policies*. These two patterns is discussed in the next subsection. The complicating effects of Java's static type discipline are illustrated by the following example.

Consider a program that manipulates lists of several different types of elements. One approach is to define a separate composite class hierarchy for each kind of list. But this approach requires replicating essentially the same code in the definition of each class. To avoid this code replication, we can define a single composite class hierarchy for lists of type `Object`. Since all Java object types are subtypes of `Object`, such a list type can be used in place of any specific list type. However, when we extract an element from such a list, we will generally have to cast it to the specific type required by the context in which it is used. These casting operations clutter the code and reduce the precision of static type checking. Nevertheless, the advantages conferred by avoiding code replication usually outweigh these disadvantages.

We can define the types in polymorphic lists more precisely and avoid the use of ugly casts by using generic (parameterized) types in our type signatures. Generic types look almost exactly like the informal paramterized types such as `(listOf T)` that are used in the functional programming textbook *HTDP*). We defer discussing generics until Section 1.13.4. The DrJava language levels facility does not support generics.

Finger Exercise Load the your `IntList.dj1` program into the DrJava Definitions pane. Remove the `insert` and `sort` methods. Convert your class definition of a class `ObjectList` where the list elements have type `Object` instead of type `int`. Test this program and save it in a file `ObjectList.dj1` for future use. ■
Finger Exercise Load the program in your saved file `ObjectList.dj1` into the

Definitions pane. Define the methods

```
ObjectList concat(ObjectList other);
ObjectList reverse();
```

that concatenate and reverse lists, respectively, using efficient algorithms. Do not hesitate to use help methods if necessary.

that sorts a list with elements of type `Comparable` into non-descending order, akin to the `sort` method on `IntList` that you defined in section 1.7. Your code will need to use the operation

```
int compareTo(Object other);
```

in the `Comparable` interface to compare elements. If you find the description of this interface using generics confusing, look on the web for the documentation for the Java 1.4 API. It describes the `Comparable` without mentioning generics. Write a JUnit test class for your code. ■

1.8.3 Interface Types

An interface identifies a program type independent of any specific class. This mechanism enables Java to express computations in more abstract and flexible terms. Consider the following example. Java includes a built-in interface named `Comparable` with the following definition:

```
interface Comparable {
  int compareTo(Object o);
}
```

All of the methods in an `interface` are automatically `abstract` and `public`. Let us define a class `ComparableList` similar to `IntList` where list elements have the type `Comparable`. An object has type `Comparable` iff it is an instance of a class that implements the `Comparable` interface. Interface implementation is inherited: if a class C implements an interface I then all subclasses of C implement I also.

In this context, we can define the composite class hierarchy with root `ComparableList` as follows:

```
abstract class ComparableList { }

class EmptyComparableList extends ComparableList {
  static EmptyComparableList ONLY = new EmptyComparableList();
  private EmptyComparableList() { }
}

class ConsComparableList extends ComparableList {
  Comparable first;
  ComparableList rest;
}
```

Now assume that we want to modify the `ComparableList` class so that it implements the `Comparable` interface. The `compareTo` method in the `Comparable` interface has the following contract. Any class C that implements the `Comparable` interface must have an associated binary relation that is *totally ordered*: for every pair of objects a and b in C, either (*i*) a is less than b, (*ii*) a `equals` b, or (*iii*) a is greater than b. For any instance `o` of the same class as `this`, `compareTo(o)` returns (*i*) a negative number if `this` is less than `o`, (*ii*) zero if `this equals o`, and (*iii*) a positive number if `this` is greater than `o`. If `o` belongs to a different class than `this`, `compareTo(o)` throws a `ClassCastException` indicating an erroneous use of the `compareTo` method.

In the `ComparableList` class, we can impose a lexicographic total ordering on lists. This ordering is a generalization of the familiar *alphabetic* ordering on strings. In such an ordering, a precedes b iff either

- a is empty and b is not,
- the first element of a precedes the first element of b, or
- the first element of a `equals` the first element of b and the `rest` of a precedes the `rest` of b.

Finger Exercise Load your saved file `ObjectList.java` into the DrJava Definitions pane. Convert it to a definition of the class `ComparableList` given above and define the `sort` and `insert` methods analogously to way they are defined in `IntList`. (Hint: you can simply apply the appropriate textual substitions to `IntList` to generate `ComparableList`. Be careful to use the "whole word" option when you substitute for `int`.

Revise your class `IntListTest` to form an effective test class for `ComparableList`. Save both `ComparableList` and `ComparableListTest` in files `ComparableList.dj1` and `ComparableListTest.dj1`, respectively. ■

Finger Exercise Extend your `ComparableList` class to implement the `Comparable` interface as described above and save the resulting class as `ComparableList.dj1`. Augment `ComparableListTest.dj1` with appropriate test cases and save it. ■

1.9 The Command and Strategy Patterns

In a finger exercise in Section 1.4.4, we extended the `DeptDirectory` program by writing a method `findPhone(String name)` to look up a person's phone number. We implemented `findPhone(String name)` in exactly the same way as `findAddress(String name)`, replicating method code. A better strategy would be to implement a method

```
Entry findEntry(String name)
```

that returns the `Entry` matching a given `name`, and then to define both `findPhone` and `findAddress` in terms of `findEntry`.

In this section, we will explore a far more general technique for eliminating code replication called the *command pattern*. To accommodate returning different `Entry` fields, we will define a method

```
String findField(Operation f, String name)
```

that takes an `Operation` object as an extra argument specifying which field to return. This approach mimics the familiar "code factoring" process in Scheme: repeated code patterns are abstracted into functions that take parameters that "customize" the code appropriately. In many cases, these parameters are functions.

Code factoring involving functions as parameters cannot be directly implemented in Java because methods are not values that can be passed as arguments. Some object oriented languages such as SmallTalk and Self classify methods as data values, permitting code factoring to be implemented directly. Fortunately, it is not difficult to get around this restriction by explicitly representing methods as objects. All we have to do is introduce an appropriate abstract class `Operation` containing a single abstract method `execute( ... )` and *define a separate concrete subclass of* `Operation` *for each method that we want to pass as an argument.* Each concrete subclass defines the abstract method `execute` appropriately. In the general case, the `Operation` subclasses may contain fields that correspond to the free variables appearing in procedural arguments in Scheme. These free variables must be bound when the `Operation` is constructed, exactly as they are in a language supporting procedures as data values.

In the object-oriented design literature, this technique is called the *command* pattern in homage to the dominant role that imperative operations have played in object-oriented computation. Here we are using this pattern in a purely functional fashion.

To illustrate the command pattern, let us continue our `DeptDirectory` example. If we independently write `findPhone` and `findAddress`, they differ only in the field name used in the `return` expression.

```
class Empty extends DeptDirectory {

  ...

  String findAddress(String name) {
    return null;
  }
  String findPhone(String name) {
    return null;
  }
}

class Cons extends DeptDirectory {

  ...

  String findAddress(String name) {
    if (name.equals(first.name))
      return first.address();
    else return rest.findAddress(name);
  }
  String findPhone(String name) {
    if (name.equals(first.name))
      return first.phone();
    else return rest.findPhone(name);
  }
}
```

We can "abstract out" this difference by writing a single `findField` method embodying the common code in the methods `findPhone` and `findAddress`. To accommodate differing choices for the returned `Entry` field, the method takes an `Operation` that performs the appropriate field extraction on the `Entry`. The following code includes a new mechanism for defining concrete subclasses, called *anonymous classes*, that we have not discussed before. We will explain anonymous classes in detail below. In this example, anonymous classes are used to generate instances of new subclasses of the interface `Operation`; the static fields `address` and `phone` are bound to objects of type `Operation` that define the `execute` method as the method extracting the `address` and `phone` fields, respectively, of an `Entry`.

```
interface Operation {
  String execute(Entry e);  // implicity public and abstract

}

abstract class DeptDirectory {
  ...
  abstract String findField(Operation c, String name);
  String findAddress(String name) {
    return findField(opddress, name);
  }
  String findPhone(String name) {
    return findField(opPhone, name);

  static Operation opAddress = new Operation() {
    // ANONYMOUS class
    public String execute(Entry e) { return e.address(); }
  }
  static Operation opPhone = new Operation() {
    // ANONYMOUS class
    public String execute(Entry e) { return e.phone(); }
  }
}

class Empty extends DeptDirectory {
  ...
  String findField(Operation c, String name) {
    return null;
  }
}

class Cons extends DeptDirectory {
  ...
  String findField(Operation c, String name) {
    if (name.equals(first.name)) return c.execute(first);
    else return rest.findField(c,name);
  }
}
```

Each brace construction

```
    {  // ANON CLASS
      public String execute(Entry e) { return e. ...; }
    }
```

following a `new Operation()` expression above defines a unique instance of a new *anonymous* (unnamed) class implementing the interface `Operation`. In Java, anonymous classes are simply an abbreviation mechanism. The `Operation` class could have been written without anonymous classes as follows:

```
  interface Operation {
    String execute(Entry e);
    }
  }
  class AddressOperation implements Operation {
    public String execute(Entry e) { return e.address(); }
  }
  class PhoneOperation implements Operation {
    public String execute(Entry e) { return e.phone(); }
  }
  abstract class DeptDirectory {
    ...
    static Operation opAddress = new AddressOperation();
    static Operation opPhone = new PhoneOperation();
  }
```

at the cost of introducing the new class names `AddressOperation` and `PhoneOperation`.

In general, a single instance of a new class extending class (implementing interface) C can be created using the notation:

```
new C(...)  { ... members ...}
```

where `C(...)` specifies what superclass initialization should be performed on the instance. If C is an interface, then the argument list in `C(...)` must be empty. No constructors can appear in the list of members because the class is nameless and cannot be instantiated again. Any required initialization of fields inside the instance can be specified directly in the code defining the class.

If we ignore the ugly notation, an anonymous class extending the abstract class `Operation` has a direct analog in Scheme that you may have recognized, namely a `lambda`-expression. In any situation in Scheme where it is appropriate to use a `lambda`-expression, you can use an anonymous class in Java! The failure to make such an identification is the single most glaring failure of most expositions on Java. Of course, they are typically written for readers with a background in C/C++ rather than Scheme or other language that support procedures as general data objects.

If an anonymous class appears inside a dynamic method, it can contain references to the fields of the enclosing class instance—akin to the free variables that can appear in Scheme `lambda`-expressions. The only complication is the treatment of the variable `this`. Since an anonymous class defines an instance of a new class, the variable `this` inside an anonymous class refers to the new class instance. It does *not* refer to the enclosing class instance. To refer to the "entire" enclosing class instance, Java uses the notation C`.this` where C is the name of the enclosing class.

Exercises:

1. Add a `map` method to the `IntList` class that takes an `Operation` and applies it to each element of `this` to produce a new `IntList` with exactly the same number of elements as `this`.

2. Assume that a vector
$$< a_0, a_1, a_2, ..., a_n >$$
is represented by the list
$$(a_0\ a_1\ a_2\ ...\ a_n).$$
where the coefficient's a_i are objects of type `Double`. Add a method

```
double norm()
```

to `IntList` computes the *norm* by `this.v` by squaring the elements, adding the squares together and taking the square-root of the sum. You can compute the vector of squares using `map` and the define a method `double sum()` to compute the sum.

3. Assume that a polynomial

   $$a_0x + a_1x + a_2x^2 + ... + a_nx^n$$

   is represented by the list

   $$(a_0\ a_1\ a_2\ ...\ a_n)0$$

   where the coefficient's a_i are objects of type `Double`. Write a method

   ```
   double eval(IntList p, Double x)
   ```

   to evaluate the polynomial at coordinate x. Use Horner' rule asserting that

   $$a_0 + a_1x + a_2x^2 + ... + a_nx^n = a_0 + x \cdot (a_1 + x \cdot (a_2 + ...x \cdot a_n))$$

   Remember to use the structural design pattern for processing lists (and by association polynomials as we have represented them).

1.10 Loose Ends

We have nearly finished covering the core "functional" subset of Java. Only a few loose ends remain. They include:

- static methods;
- local variables;
- casts and static type checking;
- exceptions;
- name and method overloading; and
- complete Java programs.

We will defer discussing the *visitor* pattern until we cover these loose ends because our code examples for the visitor pattern will use some of these language features.

1.10.1 Static Methods

In Section 1.6.1, we briefly discussed static members of classes because static fields are the only sensible way to define the constants relevant to a given class. Up to now, we have avoided discussing static methods.

A static method is a method definition preceded by the modifier `static`. In the body of a static method, there is no implicit argument `this`. Static methods belong to classes rather than objects. In a static methd call, the class name implicitly or explicitly precedes the method name, just as the receiver the method name in an conventional (dynamic) method call. For example, there is a static method `toString(int i)` in the class `Integer` that converts an `int` input `i` to its standard `String` representation. Hence, given any expression *e* of type `int`, `Integer.toString(e)` is the corresponding `String` representation.

There are several important uses of static methods that warrant disucsssion. Since primitive values are not objects, there is no way in Java to define methods that work on primitive values in an OO style. The best way to express such methods is to make them `static`. Of course, the `static` modifier can be dropped from any static method but the resulting dynamic method completely ignores its receiver. Moreover, a dynamic method that makes no reference to its receiver is less useful than its static counterpart because it cannot be invoked without a receiver. In contrast, static methods from a class `C` can be called even if no objects of type `C`, *e.g.*, `C` is `abstract` and has no subclasses.

1.10.2 Other Uses of Static Methods

Static methods are necessary in Java for a variety or reasons including:

- defining operations on primitive types (as discussed above);
- defining operations on instances of library classes that cannot be modified or extended such as `String`; and
- defining operations on arrays (which we will discuss later).

For example, an operation

```
public String squeezeWhiteSpace(String s);
```

that returns a `String` identical to `s` with all spaces and tabs removed must be expressed as a static method because the library `String` class cannot be modified or extended.

Finally, all operations on arrays must be expressed in static (procedural) form because array types do not have conventional class definitions; they are built-in to the Java virtual machine. We will discuss arrays in Chapter 2 when we address *imperative* programming in Java.

In DrJava, static methods are not allowed at the Elementary and Intermediate language levels.

1.10.3 Casts and Static Type Checking

In Scheme every primitive operation dynamically checks that its arguments have the appropriate form (type) as it executes. If an operation is applied to data of the wrong form, Scheme aborts execution and prints an error message, much as Java does when an exception is thrown and not caught.

Java also performs some argument checking during program execution (*run-time*), but most argument checking is done statically by the compiler *before* a Java program executes. A Java compiler enforces a syntactic discipline on program text called *static typing*. The compiler uses a collection of *type-checking* rules to determine whether a program is *well-typed* or not. Programs that are not well-typed are rejected with an explanation of which rules were broken.

The type-checking rules embody simple "common sense" inferences and consistency checks. The rules assign a type to every program expression and subsequently check that these type assignments are consistent. A Java compiler assigns types to program expression as follows:

1. every program constant has a type specified by the language definition;
2. every program variable (field, parameter, or local variable) has the type declared in the program;
3. each method invocation has the declared return type of the method;
4. each application of an arithmetic operator (*e.g.*, `+`, `*`) has the return type stipulated by a table in the language definition;[8]
5. each application of a relational operator and `instanceof` test has type `boolean`;
6. each conditional expression

   test `?` *consequent* `: alternative`

   has the least (smallest) type among all defined types that includes both the *consequent* type and *alternative* type as subtypes; and
7. the type of any cast expression

   T e

   is *T*.

Given these type assignments, a Java compiler checks their consistency by enforcing the following rules:

[8]The return types of arithmetic operators generally depend on the types of their arguments. For some operator applications, the table immediately reports a type error.

1. the type of the receiver of a field selection or method invocation includes the specified field or method in its signature (a list of the member headers for the class or interface);

2. the type assigned to each argument expression in a method invocation is a subtype of the declared type of the corresponding parameter;

3. the type of the right-hand side of an assignment is a subtype of the type of the left-hand-side; and

4. the type of each return expression in a method body is a subtype of the declared return type of the method.

Note that Java type checking rules do not capture the logical consequences of `instanceof` tests. As a result, Java program text often must include apparently redundant casting operations in code following an `instanceof` test.

This phenomenon is illustrated in the following simple example. Consider the following method which could be added to the `IntList` class above.

```
static Object first(IntList l) {
  if (l instanceof Cons) return ((Cons) l).first;
  else throw
    new ClassCastException("first requires a non-Empty IntList");
}
```

In the method, all occurrences of the parameter `l` have the same type, namely `IntList` as declared in the method header. The

```
l instanceof Cons
```

test has no effect on type-checking. As a result, the occurrence of `l` preceding the field extraction operation `.first` must be explicitly *converted* to type `Cons` using the casting operation `(Cons)` written as a prefix in front of `l`. Since the field `.first` is not defined in the abstract class `IntList`, the definition of the method `first` does not type check if the casting operation `(Cons)` is omitted.

Applying a casting operation `(` T `)` to a Java expression e of some static object type U has consequences for both program execution and compilation. First, it inserts a run-time check to confirm that the value of e belongs to the type T as claimed by the casting operation. Second, it converts the static type of the expression e from U to T.

1.10.4 Local variables

Method parameters in Java play exactly the same role as function parameters in Scheme. They are local variables bound to the corresponding argument values. They are destroyed when the method returns.[9] Java also provides a mechanism, akin to Scheme's `local` construct, for introducing local variables to perform a subcomputation without invoking a helper method. In Java, local variable definitions can be inserted as statements in *any* statement sequence (such as a function body or compound statement). Each such variable is accessible only in program text between its definition and the end of the statement sequence. We have already used this language construct without explicitly mentioning it.

For example, consider the problem of adding a method `IntList merge(IntLlist other)` to the `intList` class discussed earlier in the monograph. This method merges `this` and `other` assuming that both are already in (ascending) order. In the following code fragment, comments have been elided to save space:

> abstract class IntList IntList cons(int n) return new ConsIntList(n, this); abstract IntList merge(IntList other); class EmptyIntList extends IntList static EmptyIntList ONLY = new EmptyIntList(); private EmptyIntList() IntList merge(IntList other) ... class ConsIntList extends IntList int first; IntList rest; Intlist merge(IntList other) ...

The definition of `merge` for subclass `EmptyIntList` is trivial (it merely returns `other`), but the definition for `ConsIntList` requires casting `other` to type `Cons` once we have dealt with the empty case for `other`:

```
IntList merge(IntList other) {
  if (other.equals(EmptyIntList.ONLY)) return this;
  else if (first <= ((ConsIntList) other).first()) return rest.merge(other).cons(first
  else return ((ConsIntList) other).rest().merge(this).cons(((ConsIntList)other).first
}
```

We can simplify this method definition by introducing a local variable of type `ConsIntList` bound to `other`:

```
IntList merge(IntList other) {
  if (other.equals(EmptyIntList.ONLY)) return this;
  ConsIntList o = (ConsIntList} other;
  if (first <= (o.first()) return rest.merge(o).cons(first);
  else return o.rest().merge(this).cons(o.first());
}
```

[9] In Scheme, this statement is false in general; local variables continue to exist as long as a "closure" referring to them exists.

1.10.5 Exceptions as Errors

Some operations on data are inherently partial. For example, there is no mathematically sensible definition for the result of integer division by zero. Similarly, there is no sensible definition for the *first* element of an *empty* sequence. Java provides an elegant mechanism for coping with this problem. Program operations can "throw" a "run-time error" condition called an *unchecked exception* that aborts program execution and prints a diagnostic error message and a list, called a traceback, of the stack of pending method calls. For example, attempting to divide any `int` by 0 generates an `ArithmeticException`. Exceptions are ordinary Java data objects descended from the built-in class called `Exception`. Unchecked exceptions are descendants of the class `RuntimeException` extending `Exception`.

In Java, any method can "throw" an exception simply by executing the statement

`throw` *e*;

where *e* is any expression that evaluates to an object of type `Exception`. The classes `Exception` and `RuntimeException` both include a zero-ary constructor and a unary constructor. The latter takes an error message `String` as an argument. The string argument is printed as a diagnostic message if it is provided.

The following Java code implements "functional" lists of objects:

```
abstract class List {

  abstract Object first();
  abstract List rest();

  abstract String toStringHelp();
  // List -> String without enclosing parentheses and leading blanks
}

class Empty extends List {

  Object first() {
    throw new
      ClassCastException("first requires a non-Empty List");
  }
  List rest() {
    throw new
      ClassCastException("rest requires a non-Empty List");
  }
  public String toStringHelp() { return "()"; }
  String toStringHelp() { return ""; }
}
```

```
class Cons extends List {
  Object first;
  List rest;

  Cons(Object f, List r) {
    first = f;
    rest = r;
    }

  Object first() { return first; }

  List rest() { return rest; }

  public String toString() { return "(" + first + rest.toStringHelp() + ")"; }
  String toStringHelp() { return " " + first + rest.toStringHelp(); }
}
```

The class `ClassCastException` is a built-in Java class extending `RuntimeException` that other built-in Java classes use to signal improper method applications where `this` or some other argument belongs to the wrong class.

`Exception` objects that do not belong to the type `RuntimeException` are called *checked* exceptions. We will discuss how they are used later in this monograph.

1.10.6 Name and Method Overloading

In Java, the same name can simultaneously be used for a local variable or method parameter, different fields of the same object, and different methods of the same object. Java uses context and type information to determine the meaning of a name. For example, many Java programmers would write the constructor for the class `Cons` above as follows:

```
  Cons(Object first, List rest) {
    this.first = first;
    this.rest = rest;
    }
```

In this example, the names `first` and `rest` each have two different meanings. Java resolves potentially ambiguous uses of names by using the innermost (most local) definition of the name. Hence, the prefix `this.` is required to distinguish the `first` and `rest` fields of `this` from the constructor parameters of the same name.

While the use of the same name for *different* kinds of program entities is widely accepted as good programming practice, the use of one name for several different fields or methods is more controversial. A Java subclass B can introduce a field with exactly the same name n as a field in its superclass A. The inherited field is *not overridden* (what would overriding mean in this context?); it is merely "shadowed". When the name n appears in a method of B, its meaning depends on the *type* of the receiver. If the receiver is `this`, then the new field n introduced in B is meant. But if the receiver has type A rather than B, then the old field n introduced in A is meant. **Warning:** duplicate field names can be the source of insidious program errors that are very difficult to diagnose. *For this reason, we strongly recommend against using them.*

Duplicate method names are less controversial but can still be dangerous. In a class, Java permits the same method name to be used for different methods as long as their argument lists do not identical the same length and same types. The practice of defining more than one method in a class with same name is called *method overloading*. Java resolves overloaded method names using the types of the argument expressions. When the Java compiler encounters a method invocation involving an overloaded method, it determines the types of the method arguments and uses this information to select the "best" (most specific) match from among the alternatives. If no best method exists, the program is ill-formed and will be rejected by the Java compiler.

We urge restraint in using the same name for different methods involving the same number of arguments. Since static type information is used to resolve which method is meant, program errors may be difficult to find because the programmer may not infer the correct static type when reading the code.

1.10.7 Complete Java Programs

Every complete Java *program* must contain a *root class* where execution can begin. A root class must contain a `main` method defined with the header

```
public static void main(String[] args)
```

The return type of the `main` method is `void`, which is a degenerate, empty type. There are no values of type `void`. A method with `void` return type does not return a value. A method that retursn `void` usually has some *side effect* such as printing an answer, writing a file, or modifying the value of a field of an object. We have scruplulously avoided programming with side effects up to this point in this monograph and will not introduce them until Chapter 2.

To execute a Java program, a user must identify the name of the root class for the program. In most Java computing environments, the command required to execute the Java program with root class C is simply

```
java C
```

Note that a class may contain a `main` method even if it is not intended to be the root class of an actual program. Of course, evaluating the DrJava expression

```
Motto.main(null);
```

is equivalent to executing this conssoled. For the sake of programming convenience, the DrJava Interactions pane accepts the command line syntax as well as the explicit invocation of the `main` method shown above.

In batch programming environments that lack an Interactions pane like DrScheme, it is common practice to use `main` methods in non-root classes as test methods. Such test methods can easily be executed from the command line by typing

```
java C
```

where *C* is the class being executed. We do not recommend this practice in DrJava because DrJava can evaluate any program expression in the Interactions pane, the testing process is even more convenient. For the moment, we recommend defining a static method `test()` in each class that tests all of the methods of the class. Later, we will revise this recommendation to describe a much more comprehensive testing strategy.

When execution begins in the `main` method of a root class, *no* instances of the root class exist. In fact, most Java programs never instantiate the root class because it serves as a framework for organizing program execution rather than a definition of a new form of data. Java classes really serve two distinct purposes: defining data objects and organizing static methods and variables.

The `args` parameter of the `main` method in the root class is used to pass command line options to the Java program. We will not use this feature in this monograph.

A Complete Java Program

The following simple program

```
class Motto {
  public static void main(String[] args) {
    System.out.println("Java rules!");
  }
}
```

prints the `String` output

```
Java rules!
```

and stops when the Java program with root-class *Motto* is executed. Before any Java program can be executed, it must be compiled into a form suitable for machine execution called a *class file* `Motto.class`. The name of the class file generated for a class *C* is simply *C*`.class`.

In DrJava, the Compile button applies the compiler to each open Java files and write out the resulting class files to the file system—assuming that the files did not contain any syntax errors. Each class file is stored in the same directory as the corresponding source file.

Finger Exercise: Using DrJava, define the `Motto` class given above and execute it as program. ■

1.11 The Visitor Pattern

The composite and interpreter patterns enforce an elegant discipline for writing object-oriented functional programs. But this elegance comes at a price: every time we need to define a new operation we must modify every class in the composite class hierarchy to add a new method. As a result, program modification involving new methods is painful. Moreover, we cannot use these patterns to define new operations if we do not control the source code for the composite class. Without access to the source code, the only way we can define new methods is to write them in procedural style as static methods in another class, losing all the benefits of object-oriented organization.

In this section, we introduce a design pattern, called the *visitor pattern*, that completely and elegantly solves the problem of adding new operations to composite class hierarchies without modifying the text of either the composite class or its variants.

Before delving into the technical definition of the visitor pattern, we present a motivating example: an interpreter for arithmetic expressions.

1.11.1 Interpreting Arithmetic Expressions

An `ArithExpr` is either:

- `Const(`*c*`)`
- `Sum(`*left right*`)`,
- `Prod(`*left*,*right*`)`, or
- `Neg(` *left*`)`

where *c* is an `int` and *left* and *right* are `ArithExprs`.

As before, to represent this data type in Java, we employ the composite pattern. We define an abstract class to represent the union of these types, and four concrete subclasses, one for each of the different variants.

```
abstract class ArithExpr { }

class Const extends ArithExpr {
  private int value;
}

class Sum extends ArithExpr {
  ArithExpr left, right;
}

class Prod extends ArithExpr {
  ArithExpr left, right;
}

class Neg extends ArithExpr {
  ArithExpr arg;
}
```

Next we need a way to evaluate our expressions. First, let's try defining an `eval` method that returns a constant for any `ArithExpr`.

We add the abstract method

```
abstract Const eval();
```

to the `ArithExpr` class. We could easily define `eval` to return an `int`, but we've chosen to return a `Const` instead because it buys us a little flexibility later.

In the `Const` class, we add a concrete version of the abstract `eval` method:

```
  Const eval() { return this;  }
```

But now we encounter a minor problem: to evaluate products and sums, we need to be able to multiply or add two instances of `Const`. Multiplication and addition are not defined for instances of `Const`, but they are defined for the `int` values embedded inside instances of `Const`. Thus, we can use the accessor `value()` to retrieve the value of a `Const`. To the class `Prod`, we add the method

```
  Const eval() {
    return new Const((left.eval().value()) * (right.eval().value()));
  }
```

The `eval` methods for `Sum` and `Neg` are defined similarly.

Let us amplify this example by adding variables as a syntactic category to `ArithExpr`. Writing down our revised data type in a shorthand form, we have

```
ArithExpr ::= Const(int)
            | Sum(ArithExpr, ArithExpr)
            | Prod(ArithExpr, ArithExpr)
            | Neg(ArithExpr)
            | Var(String)
```

where the variant `Var` represents variables. The `String` field in a `Var` is the variable's name.

This notation is merely shorthand for the follow ing prose. An `ArithExpr` is either:

- `Const(`*c* `)`,
- `Sum(`*left*,*right* `)`,
- `Prod(`*left*,*right*) `)`,
- `Neg(`*left*`)`, or
- `Var(`*s*`)`

where *c* is an `int`, *left* and *right* are `ArithExpr`s, and *s* is a `String`.

In order to evaluate expressions including variables, we introduce *environments*, which store collections of *bindings*. A *binding* is simply a pair containing a variable name and a value. We will also have to modify `eval` so that its signature becomes

```
Const eval(Environment env)
```

We can implement `Environment`s using functional lists of string-value pairs. The details of the implementation are left as an exercise. (Hint: look back at the departmental directory example.)

The definition of `eval` will have to change accordingly for each of the existing concrete subclasses, but only the `Var` subclass will actually make use of the environment parameter. For example, `Sum.eval` will become

```
Const eval(Environment env) {
  return new Const(left.eval(env).value() + right.eval(env).value());
}
```

The parameter `env` is not used directly in the `eval` code for `Sum`s, but it is passed to the recursive calls to `eval` in case there is a `Var` further down in the expression. It is only in class `Var` that we need to use the environment parameter to look up the value of a variable:

```
class Var extends ArithExpr {
  /* fields */
  private String name;

  /* constructor */
  Var(String n) { name = n; }

  /* accessors */
  public String name() { return name; }

  /* toString */
  public String toString() { return name; }

  Const eval(Environment env) { return env.lookup(name); }
}
```

Here `env.lookup(name)` fetches the `Const` value associated with `name` in the environment `env` (if there is no entry for `name`, `lookup` should raise some kind of exception).

Having to pass the environment as a parameter to all of the `eval` methods, when it is directly used in only one of them, is clumsy. As we shall see, there is a different way to implement expression evaluation that avoids this problem.

1.11.2 Openness in Data Design

Recall our definition for an arithmetic expression *without* variables:

```
ArithExpr := Const(int) | Sum(ArithExpr, ArithExpr) ....
```

Our implementation of this data definition using the composite pattern would be more robust and more flexible if we could define new operations on `ArithExpr`s *without modifying any existing code*. Fortunately, there is a clever design pattern called the *visitor pattern* that lets us do this. The idea underlying the visitor pattern is to bundle the methods defining the new operation for each concrete subclass together in a new class called a *visitor class*. An instance of such a class is called a *visitor*.

First, we will define a new interface `Visitor` that specifies what methods must be included in every visitor class for `ArithExpr`:

```
interface Visitor {
  int forConst(Const c);
  int forSum(Sum s);
  ...
}
```

Notice that each method takes an instance of the class that it processes. This argument, called the *host*, is needed to give it access to all the information that would be available through `this` if the method were defined inside that class, *e.g.*, the values of the object's fields returned by accessors.

Now we will create a new concrete class `EvalVisitor` to hold all the methods for evaluation of an `ArithExpr`:

```
class EvalVisitor implements Visitor {

  int forConst(Const c) { return c.value(); }
  int forSum(Sum s) {
    return s.left().accept(this) + s.right().accept(this);
  }
  ...
}
```

We need to install a hook in each subclass of `ArithExpr` to execute the corresponding visitor method. The hook is a new method, `accept`, which takes a visitor as an argument and calls the appropriate method in that visitor.

```
abstract class ArithExpr {}
  abstract int accept(Visitor v);
}

class Const {
  ...
  int accept(Visitor v) {
    return v.forConst(this);
  }
}

class Sum {
  ...
  int accept(Visitor v) {
    return v.forSum(this);
}

...
```

To evaluate an arithmetic expression, we simply call

```
a.accept(new EvalVisitor())
```

If we wish to add more operations to arithmetic expressions, we can define new visitor classes to hold the methods, *but there is no need to modify the existing subclasses of* `ArithExpr`.

Notice that, since a visitor has no fields, all instances of a particular visitor class are identical. So it is wasteful to create new instances of the visitor every time we wish to pass it to an `accept` method. We can eliminate this waste by using the *singleton* design pattern which places a static field in the visitor class bound to an instance of that class.

```
class EvalVisitor {
  static ONLY = new EvalVisitor();
  ...
}
```

Then, instead of

```
accept(new EvalVisitor()),
```

we may simply write

```
accept(EvalVisitor.ONLY).
```

Another elegant way to define visitors is to define each visitor as an anonymous class. Since an anonymous class definition defines only one instance of the new class, it produces results similar to the singleton pattern. The principal difference is that the new class has no name; the unique instance must be bound to a local variable or field declared in the enclosing program text.

Recall that an anonymous class has the following syntax:

`new` *className*($arg_1, ..., arg_m$) { $member_1; ...; member_n$ }

In most cases, the class *className* is either an `abstract` class or an `interface`, but it can be any class. The argument list $arg_1, ..., arg_m$ is used to call the constructor for the class *className*; if *className* is an interface, the argument list must be empty. The member list $member_1; ...; member_n$ is a list of the member definitions for the new class separated by semicolons.

For example, to create an instance of a visitor that evaluates an arithmetic expression, we write:

```
new Visitor() {
  int forConst(Const c) {...}
  int forSum(Sum s) {...}
  ...
}
```

Since we generally want to use a visitor more than once, we usually bind the anonymous class instance to a variable, so we can access it again! The statement:

```
visitor ev = new Visitor() {
               int forConst(Const c) {...};
               int forSum(Sum s) {...};
               ...
               };
```

binds the variable `ev` to our anonymous class instance.

1.11.3 Polymorphic Visitors

In our application of the visitor pattern above, the `for` methods of the visitor classes and the `accept` methods for the `ArithExpr` classes returned values of type `int`. This convention is acceptable as long as all the computations we ever want to perform over `ArithExpr`s have integer results. But if not, we are forced to declare a new interface visitor type and new `accept` methods for each distinct result type. Since the whole point of the visitor pattern is to avoid having to modify a data type every time we wish to perform some new computation over it, we have a potential problem.

We can address this problem by redefining the `for` and `accept` methods so that they return a more general type. Before getting into the details, let's step back and give visitors a more general definition.

A *visitor* is an object containing

- a concrete description of an operation on composite data, with a separate method for processing each alternate form of the data; and
- the argument values required by the operation.

The properties common to all visitors for a particular composite type are collected in corresponding the visitor interface. Any arguments that are processed by a particular visitor are typically stored in fields declared in the corresponding visitor subclass to avoid compromising the generality of the interface. A typical visitor interface has the form

```
interface Visitor {
  // a "for" method for each concrete
  // subclass of the visited class
  Object forC1(...);
  .
  .
  .
  Object forCn(...);
}
```

We use `Object` as the return type of the `for` methods so that we can accommodate concrete visitor classes that produce almost any type of result. This convention exploits the *polymorphism* inherent in class inheritance: every object belongs to all of the types associated with its superclass, super-superclass, *etc.* Since the class `Object` is perched at the root of the class hierarchy, all objects belong to the type `Object`, and support the operations defined in class `Object`. The only types that do not belong to `Object` are the primitive types like `int`. Fortunately, we can get around this by using the corresponding wrapper classes (*e.g.* `Integer` instead of `int`). In the event we want a visitor operation to have a `void` return type, we can either return the `null` reference or the only instance of a singleton class `VoidType`.[10]

Let us return to our `ArithExpr` example. We can generalize our visitor class as follows:

```
abstract class AE { // AE is short for ArithExpr
  abstract Object accept(Visitor v);
}

class Const extends AE {
  /* fields */
  int value;

  /* mehods */
  Object accept(Visitor v) { return v.forConst(this); }
}

. . .
```

The code for the `Sum` and `Prod` ares nearly identical except for the fact that the `accept` methods in those classes respectively invoke the `forSum` and `forProd`, methods of the `Visitor` argument `v`. They all have `Object` as their return type.

1.11.4 Polymorphic Visitors with Arguments

To make our visitor example more interesting and realistic, let us include variables in the type `ArithExpr`.

```
class Var extends ArithExpr {
  String name;

  Var(String n) { name = n; }
```

[10]The Java libraries include a `Void` class, but it cannot be instantiated, so we must define our own `VoidType`.

```
    public String toString() { return name; }
    Object accept(Visitor v) { return v.forVar(this); }
  }
```

Then the visitor interface for `ArithExpr`s has the form:

```
  interface Visitor {
    Object forConst(Const c);
    Object forSum(Sum c);
    Object forProd(Prod p);
    Object forVar(Var v);
  }
```

The concrete visitor class that implements expression evaluation is:

```
  class EvalVisitor implements Visitor {
    Env env;  // an environment for looking up variables
    EvalVisitor(Env e) { env = e; }
    Object forConst(Const c) { return c; }
    Object forSum(Sum s) {
      return new Const( ((Const)s.left().accept(this)).value() +
                        ((Const)s.right().accept(this)).value() );
    }
    Object forProd(Prod p) {
      return new Const( ((Const)p.left().accept(this)).value()) *
                        ((Const)p.right().accept(this)).value()));
    }
    Object forVar(Var v) { return env.lookup(v.name()); }
  }
```

The environment `env`, which was an explicit parameter of the `eval` method in our method-based implementation for evaluation, is now a field of the visitor. As before, it is directly used only for evaluating instances of `Var`, but now we don't need to explicitly pass the environment through method argument lists.

Since we are programming in a functional style, the `forConst` method need only return its argument as the result, rather than allocating a copy. The `forSum` and `forProd` methods are mostly straightforward, evaluating the subexpressions first and combining the results. The only subtlety is that since `accept` now returns an instance of `Object` rather than an `int`, we need to perform an explicit type cast to get the values for the left and right subexpressions. For example, to obtain the value for the left subexpression in a `Sum`, we have

```
        ((Const)s.left.accept(this)).value()
```

The expression

```
s.left.accept(this)
```

computes a `Const` whose `value` field is the value of the expression `s.left`. But the declared return type for `accept` is `Object`, and since an `Object` has no field named `value`, we cannot extract the value directly. Since we know that the `Object` is in fact a `Const`, we can insert an explicit *type cast* to `Const`, and then extract the value.

The `forVar` method looks up the value of the variable in the current environment. The environment is passed in when an `EvalVisitor` is created, and is presumably given bindings for the existing variables beforehand.

Finger Exercise Finish the visitor-based implementation of expression evaluation, including the definition of the `Environment` class, by yourself. Can you think of other operations on expressions that the visitor pattern might help you implement? ■

1.12 Exceptions and Runtime Errors

In this section, we study Java exceptions, a language construct to process unusual situations and run-time errors. In the process, we will identify the problems associated with using *checked* exceptions.

Many production programs have to detect erroneous input, report it to the user, and in some cases recover to handle more input. Since handling erroneous input deviates from the expected flow of program control, Java provides a mechanism called exception handling that is tailored for this purpose. To be more specific, let us consider an example based on the arithmetic expression evaluator from the previous section.

1.12.1 A Motivating Example

Recall the visitor class for evaluating arithmetic expressions with variables:

```
class evalVisitor implements Visitor {
  Env e;
  Object forConst(Const c) {return c; }
  Object forSum(Sum s) {
    Const l = (Const)(s.left.accept(this));
    Const r = (Const)(s.right.accept(this));
    return new Const(l.value + r.value);
  }
```

```
    Object forVar(Var v) { return env.lookup(v.name); }
  }
```

The casting operations `(Const) ...` in the body of the `forSum` method are required by the Java type checker. The Java type system is too imprecise to determine that the recursive invocations of the visitor in `forSum` will never return any value other than a `Const`. The Java type system simply uses the declared return type for a method as the type of invocation of that method.

You might wonder why the designers of Java adopted such a simple, imprecise type system. Precise type systems have two crippling disadvantages. First, they perform a complex inference process that is difficult to understand. If a programmer makes a type error, it is difficult to determine what program revisions are required to satisfy the type checker. Second, precise type inference is expensive. The time complexity of very precise type checking (depending on the specific algorithm) may grow with the square or cube of program size or worse.

Now let us augment Arithmetic Expressions with a variable binding operator `let`. For example, we might want to evaluate the expression:

```
let x = 17 in x + y
```

in an environment where y is bound to `10` to produce the value `17`. This extension is reflected in the definition for data type `ArithExpr` by adding a `Let` form to the list of variants:

```
ArithExpr ::= Const(int)
            | Sum(ArithExpr, ArithExpr)
            | ...
            | Let(Var, ArithExpr, ArithExpr)
```

Similarly, in the object-oriented implementation of `ArithExpr`, we must add the variant class

```
class Let extends ArithExpr {
  Var bindVar;
  ArithExpr bindExpr;
  ArithExpr body;
  ...
  Object accept(Visitor v) { v.forLet(this); }
}
```

To define operations for our generalized Arithmetic Expressions using visitors, we need to define a new visitor interface with `for` methods for all of the variants:

```
interface Visitor {
  Object forConst(Const c);
```

```
    Object forSum(Sum c);
    Object forProd(Prod p);
    Object forVar(Var v);
    Object forLet(Let l);
  }
```

Since the evaluation of a `Let` form involves extending the environment, let us write the code for manipulating environments:

```
  class Env {
    abstract Const lookup(String name);
  }

  class Empty extends Env {
    Const lookup(String name) {
      return null;
    }
  }
  class Cons extends Env {
    String firstName;
    Const firstVal;
    Env rest;

    Cons(String name, Const val, Env env) {
      firstName = name;
      firstVal = val;
      rest = env;
    }

    Const lookup(String name) {
      if (name.equals(firstName)) return firstVal;
      else return rest.lookup(name);
    }
  }
```

To evaluate generalized Arithmetic Expressions, we must define an appropriate concrete class extending `Visitor`:

```
  class EvalVisitor implements Visitor {
    Env env;
    Object forConst(Const c) { return c; }
    ...
    Object forLet(Let l) {
      Const bv = (Const) l.bindExpr.accept(this);
      return l.body.accept(
        new evalVisitor(new Cons(l.bindingVar.name, bv, env));
    }
  }
```

Notice the cast to Const in the definition of `bv`. Java requires this cast operation because the declared return type of `accept` is `Object`. But the value we bind to a variable to in a let expression must be a `Const`. What happens if we try to evaluate a `Var` that is not bound in the environment? To explain how Java will behave in a such a situation, we need to discuss Java exceptions in more depth.

1.12.2 Using Java Exceptions

A Java exception is an object of type `Exception`, which is a built-in Java class. There are two basic forms of exceptions that can occur during Java program execution:

1. *Unchecked* exceptions, which extend the class `RuntimeException`, usually signal a program *coding* error.

2. *Checked* exceptions, which extend the class `Exception` but not the class `RuntimeException`, are intended to signal unusual but legal conditions that require deviation from the normal flow of control.

When an `EvalVisitor` encounters a `Var` *not* bound in `Env`, it has detected an error in the input expression. If the Arithmetic Expression evaluator is being used in a larger program that can prompt the user for corrected input, then such an input error should be handled as part of valid program execution. It does *not* indicate a coding error in the Arithmetic Expression evaluator. Hence, when an `EvalVisitor` encounters an unbound exception, it could throw a *checked* exception, which the larger program can intercept and interpret, printing an error message such as

```
I'm sorry, that's not a valid expression
```

and prompt the user for corrected input with a message like

```
Please enter a valid expression:
```

Java requires an explicit `throws` clause in the header for any method that can generate a checked exception, directly or indirectly by invoking another method. The `EvalVisitor` class defined above will return `null` or generate a

```
NullPointerException
```

if it encounters an unbound variable. The `lookup` method will return `null` as the value of an unbound variable. Any subsequent attempt to use such a value as a `Const` (*e.g.*, in computing a `Sum`) will generate a

```
NullPointerException
```

. Since this exception is *unchecked*, it does not need to be declared in `throw` clauses.

If we rewrite `lookup` to throw a checked `UnboundException` instead of returning `null`, the change has a dramatic impact on the rest of the program. The revised code appears below:

```
class UnboundException extends Exception {
  UnboundException(String name) {
    super("Variable " + name + " is unbound");
    String varName = name;
  }
}

class Env {
  abstract Const lookup(String name) throws UnboundException};
}

class Empty extends Env {
  Const lookup(String name) throws UnboundException} {
    throw UnboundException(name);
  }
}

class Cons extends Env {
  String firstName;
  Const firstVal;
  Env rest;

  Cons(String name, Const val, Env env) {
    firstName = name;
    firstVal = val;
    rest = env;
  }

  Const lookup(String name) throws UnboundException} {
    if (name.equals(firstName)) return firstVal;
    else return rest.lookup(name);
  }
}

class EvalVisitor implements Visitor {
  Env env;
  Object forConst(Const c) { return c; }
```

```
    Object forSum(Sum s) throws UnboundException { ... };
    Object forProd(Prod p) throws UnboundException { ... };
    Object forVar(Var v) throws UnboundException { ... };
    Object forLet(Let l) throws UnboundException { ... };
    }
  }

  interface Visitor {
    Object forConst(Const c) { return c; }
    Object forSum(Sum s) throws UnboundException;
    Object forProd(Prod p) throws UnboundException;
    Object forVar(Var v) throws UnboundException;
    Object forLet(Let l) throws UnboundException;
  }
```

The preceding code cleanly handles input errors, but it pollutes the signatures of nearly all of the `for` methods in the class `Visitor` and its descendants. In this case, an unchecked exception is preferable. The code for this variation is identical the code above except for the `extends` clause in the definition of class `UnboundException` and the elimination of all `throws` clauses in `for` methods.

Checked exceptions and polymorphic programming do not mix well. Consequently, it should not be surprising that the Java libraries use unchecked exceptions far more than they use checked exceptions. Our advice is use checked exceptions to signal unusual conditions in code that does not involve polymorphism. If polymorphism is present, use unchecked exceptions instead.

1.12.3 Exception Handling

If the evaluation of a Java statement generates an exception, that exception can be caught (consumed) and processed by an appropriate handler associated with the statement. The handler typically restores the program to a recoverable state.

The Java construct for associating exception handlers with program statements is called a `try-catch` block. It has the following syntax

```
try {
statement_1;
...
statement_m;
}
catch(ExceptionType_1 e) { handler_1 }
... catch(ExceptionType_n e) { handler_n }
```

This statement associates the handlers described by *handler*$_1$, ..., *handler*$_n$ with the statements in the sequence

statement$_1$`;` ...*statement*$_m$`;`

If any statement in this sequence generates an exception e, it is matched against the types *ExceptionType*$_1$, ..., *ExceptionType*$_n$ in order. The handler code for the first type containing e is executed and the exception e is consumed. Then execution resumes at the statement immediately following the `try-catch` block. If no type in the catch clauses matches the generated exception e, program execution searches back up the chain of pending method calls until it finds the next "youngest" pending `try-catch` block with a matching `catch` clause. If this search exhausts the chain of pending method calls, the Java Virtual Machine prints an error message, a traceback of method calls from the point of exception generation, and aborts program execution.

Since the exception is consumed by the matching `catch` clause, the program text surrounding the `try-catch` block does not see the exception. More generally, if a `try-catch` block includes a catch clause for exception type E, the surrounding program text can never see an exception of type E emerge from the `try` block. As a result, if the exception type E is checked, the containing method need not declare a `throws` clause for type E unless other code, unshielded by an enclosing `try-catch` block, can generate exceptions of type E.

A `try/catch` block can optionally be followed by a `finally` clause

`finally` {
cleanUp
}

This clause is always executed when the `try` block terminates (including any required handler code). It typically contains code to perform any clean up that might be necessary after executing the statement sequence enclosed by the `try/catch` block, *regardless of whether the statement sequence generates an exception.* In other words, the `finally` clause is *always* executed (assuming the statement sequence does not loop infinitely). The most common usage of the finally `clause` is to release a explicitly allocated resource such as an opened file.

We illustrate the use of the `try-catch-finally` block by the following example. Suppose the `main` method of the `ArithExpr` class contains test code for expression evaluation. The `main` method can protect itself from the exceptions `eval` might throw by enclosing its invocation in a `try-catch` block as follows:

```
ArithExpr a = ...;            // set the expression
Env.e  = ...;                 // set up the environment

...
```

```
try {                         // try the operation that might fail
  ...
  result = a.eval(e);
  ...
}
catch (UnboundException u) { // handle any error that occurred
  System.out.println ("Unbound exception thrown: " + u.varName +
                      " is undefined.");
}
finally {                     // optional clean-up
  ...
}
...                           // continue with processing
```

The operation that might throw the exception, `eval`, is placed inside a `try` block, and the handling code for the exception appears in a following `catch` block. There may be more than one `catch` block, each for a different type of exception. The type and a name for the caught exception is declared at the top of the `catch` block, and the code in the block can use the exception's fields to perform some kind of error recovery. Here this recovery is simply printing a message that describes the problem.

The `finally` clause would be useful if the `try` block read each expression `a` from a separate file and `eval` could throw other exceptions besides `UnboundException`. In this case, the `finally` clause could close the file regardless of how execution of the `try` block terminated.

If the code in the `try` block raises no exceptions, or raises an exception that matches a `catch` clause, execution continues immediately after the `try-catch` block and optional `finally` clause. Of course, the `finally` clause is executed as part of the `try-catch` block.

1.12.4 Recommendations Based on Software Engineering Experience

Prior to building DrJava, I thought the concept of checked exceptions might be intellectually defensible in some cases. But after man years of experience in the software development trenches, I now believe that checked exceptions are a terrible idea and should be avoided whenever possible. As the example in the preceding section shows, checked exceptions pollute any code that uses them including code that incidentally calls a method that in some bizarre erroneous scenario throws a checked exception. As a result, checked exceptions make code refactoring much more tedious and difficult, impeding agile approaches to software develoment like Extreme Programming. In addition, checked exceptions make it painful to write any transient debugging code involving operations that throw checked exceptions. These exceptions must be caught in the context immediately surrounding the transient code. Similarly, `assert` statements cannot mention methods that potentially (based on conservative control flow analysis) throw checked exceptions. In short, checked exceptions were a superficially interesting idea that turned out to be ill-considered.

In DrJava, checked exceptions only appear is in code that directly interacts with Java library code that uses checked exceptions. DrJava also includes an exception class called `UnexpectedException` that we use to wrap checked exceptions in places where they cannot be appropriately handled. They typically bubble up to the top level where an UncaughtExceptionHandler catches and logs them.

1.13 Full Java

The DrJava language levels facility enforces all of the invariants required to support programming with immutable data in Java by banning language constructs that mutate data and automatically generating the standard operations for algebraic data types. In Scheme, all of our data types with the exception of functions were algebraic.

Given a class declaration `C` with instance fields $f_1, \ldots, f_n$, the Elementary and Intermediate langauge levels in DrJava automatically generate:

- a constructor for `C` that takes arguments $v_1, \ldots, v_n$ corresponding to the fields and initializes each field f_i to the value v_i;
- 0-ary accessor methods named $f_1, \ldots, f_n$ that extract the fields $f_1, \ldots, f_n$ from receivers of type `C`;
- overrides the `equals(...)` method to perform structural equality testing;
- overrides the `int hashCode()` method to generate hash codes consists with the overridden defintion of `equals`[11]
- overrides the `toString()` method to generate a readable, structural description of the object.

These language levels also declare all instance fields as `private` and `final`, all variables as `final`, and all methods as `public`. The `final` attribute is discussed in Section 1.13.5.

To enforce the immutability of data, the Elementary and Intermediate langauge levels prohibit assignments to variables except for initialization when the variable is defined. In addition, the Elementary language level prohibits static members,

[11]Java stipulates that `equals` and `hashCode` should obey the invariant: `x.equals(y)` implies x.hashCode() == y.hastCode(). The default definition for `hashCode` typically returns the initial address of the object in memory.

explicit visiblity modifiers, interfaces, and anonymous classes. The Intermediate level supports static fields, explicit visiblity modifiers, interface, and anonymous classes, but prohibits static methods.

1.13.1 Overriding `toString`

Every Java class includes a method `toString()` that constructs a `String` representation of `this`. But the default implementation of `toString()` provided the `Object` is really ugly: it consists of the class name for `this` followed by an ”@” sign and the `hashCode` of `this` expressed in hexadecimal notation. DrJava automatically generates an overriding of to `toString()` that constructs a “structural” description of `this`, literally how to construct `this` using `new` operations and primitive constants. (The `new` keywords are elided to improve readability.)

1.13.2 Overriding `equals`

As an example of inheritance, consider the method

```
public boolean equals(Object o);
```

which is defined in the class `Object`, the superclass of all Java classes. Any Java class that is defined without a designated superclass is an immediate subclass of the `Object` class. In the class `Object`, `equals` is defined to mean `Object` identity. Two object references are identical if and only if they refer to exactly the same object (produced by a particular `new` operation).

For some classes, identity is the appropriate definition for equality, but for many others it is not. In fact, for all functional algebraic types, the appropriate definition of equality is *structural*: two objects are equal iff they belong to the saem class and all of their fields are equal. Hence, the DrJava langauge levels facility always overrides the `equals` method to define structural equality.

Many of the classes in the Java libraries also override `equals` to suppor structural equality. Some prominent examples include the `String` class and all of the wrapper classes corresponding to the primitve types.

The overriding of the `equals` method is particularly delicate because the Java libraries all assume that `equals` defines an equivalence relation–except on the argument `null`, which is treated as a special case. In particular, for all non-`null` `x` and `y`, `x.equals(y)` iff `y.equals(x)`. If the argument passed to `equals` is `null`, then the Java API specification stipulates that `equals` must return `false`.[12] If the class

[12]This part of the `equals` specification is poorly designed because it unnecessarily complicates the behavior of `equals` without providing any useful benefit. A better specification would have stipulated that the behavior of `equals` on `null` was unspecified because `null` is outside the intended domain of the method.

containing the overriding definition of `equals` can be extended (subclassed) then the coding of `equals` is quite subtle.

In particular, the overriding definition must confirm that the argument `o` belongs to *exactly the same class* as `this`. Assume that we are overriding the definition of `equals` in the composite class hierarchy `IntList` given in Section 1.5.2 above. The following code for the definition of `equals` in the `Cons` does not work in general!

```
public boolean equals(Object o) {
  return (o != null) && (o instanceof Cons) &&
    (first == ((Cons)o).first) && rest.equals(((Cons)o).rest);
}
```

This code can fail if `Cons` has a subclass `ExtCons` because `equals` can report that an instance of `ExtCons` is equal to an instance of `Cons`. Even worse, if `equals` is overridden in `ExtCons` using the same `instanceof` pattern,

```
public boolean equals(Object o) {
  return (o != null) && (o instanceof ExtCons) &&
    (first == ((ExtCons)o).first()) && rest.equals(((ExtCons)o).rest());
}
```

`a.equals(b)` does not imply `b.equals(a)` For example, if `a` is an instance of `Cons` and `b` is an an instance of `ExtCons` with exactly the same `first` and `rest` fields as `a`, `a.equals(b)` will be `true` while `b.equals(a)` will be `false` (because `a instanceof ExtCons` is `false`.

The problem with the `instanceof` pattern for writing `equals` is that `instanceof` does not test for an exact class match. We can compare the classes of objects by using the method `getClass()` which is inherited by all classes from `Object`. This method returns an instance of the class `Class` representing the class of the receiver object. In addition, we can get the `Class` object for any specific class *C*, simply by typing `*C*.class`. Every class has exactly one `Class` object representing it. Hence, the `equals` method for `Cons` above can be rewritten:

```
public boolean equals(Object o) {
  return (o != null) && (o.getClass() == Cons.class) &&
    (first == ((Cons)o).first) && rest.equals(((Cons)o).rest);
}
```

Finger Exercise Load the sample program `intList` into the DrJava Definitions pane. Override the definition of `equals` for both `Empty` and `Cons` to match the definition of the `equal?` function in Scheme on lists of integers. The Scheme `equal?` function compares two lists to determine if they contain the same sequence of elements. Try evaluating a substantial set of test cases in the Interaction pane of DrJava. ■

Finger Exercise Load your saved program `IntList.java` into the DrJava Definitions pane. Override the definition of `equals` for both `Empty` and `Cons` to match the definition of the `equal?` function in Scheme on lists of integers. The Scheme `equal?` function compares two lists to determine if they contain the same sequence of elements. Try evaluating a substantial set of test cases in the Interaction pane of DrJava. ■

1.13.3 Visibility

The coding of non-trivial methods often involves the use of auxiliary methods called "help" methods. The specified operation may be easily derivable from another operation that has a simpler definition. For example, in the preceding definition of the composite class hierarchy `IntList`, we introduced a helper method `toStringHelp` to help support the printing of lists in the same format that Scheme uses. The `toStringHelp` prints the `rest` of a non-empty list with a leading blank before each element but no trailing blanks or closing parenthesis.

Since helper methods are defined strictly for the use of code within the class, we would like to prevent the definition from "leaking" outside the class. Java provides a mechanism for preventing such leaks. Class members can be assigned one of four visibility levels `private`, default, `protected` or `public`. `private` members are visible only within the class in which they are defined. Default members are visible only within the package in which they are defined. `protected` members are visible in the package in which they are defined and in subclasses of the defining class. `public` members are visible everywhere.

In section 1.1.1, we stated that the only way to access the fields of an object is through accessor ("getter") methods provided by the class definition. If we always declare the instance (non-static) fields of a class as `private`, then this statement is completely accurate. We *strongly recommend* following this convention; it supports to the object-oriented principle of separating the implementation of a class from its interface.

We have avoided mentioning the Java package system until now because it is not helpful in writing programs of modest size. Large Java programs typically are partitioned into packages analogous to the file directories in a tree-structure file system. Each package, except for the "default" package discussed below, has a unique name consisting of one or more Java identifiers separated by periods. Hence `java`, `java.lang`, and `java.awt.event` are are all valid package names.

Every Java class belongs to some package. If a source file does not mention a package name, then it is considered part of a special "default" package with no name. In this monograph, we will use the default package for all of the code that we write. On the other hand, all of the Java core library code that we will use resides in named packages. The Java libraries are partitioned into packages like `java.util`, `java.awt`, `java.awt.event` and `javax.swing`. Packages are not nestable. There is no connection between `java.awt` and `java.awt.event` other than a common name prefix.

The `private` attribute is well-suited to hiding helper methods that aren't required in subclasses. The `protected` attribute is useful when helper methods are referenced in subclasses residing in other packages. In our example above, `toStringHelp` is accessed by all of the subclasses of `IntList`. Hence, the appropriate protection mechanism for our `toStringHelp` is either default or `protected`. Since all our program classes reside in the same package, it doesn't matter. However, if we wanted to define subclasses of `IntList` in another package, we would need to declare the `toStringHelp` method as `protected` to make it visible within these subclasses.

When an inherited method is overridden, it cannot be made less visible. Hence, an overridden `public` method must be declared as `public`. On the other hand, an overridden `protected` method may be declared as `public`.

Finger Exercise Load the sample `IntList` program into the DrJava Definitions pane. Convert the `test` method to a `private` method. Confirm that the `main` method for the `IntList` class still executes the test suite for the `sum` method. In the Interactions pane, try evaluating the following sequence of statements:

```
IntList l = new Cons(17, new Cons(13, Empty.ONLY));
l.test(30);
```

■

1.13.4 Generics

In programming applications, we encounter many different forms of lists: list of integers, lists of double, lists of employees, *etc.*. To express these different forms of lists as conventional Java classes, we are forced either to write essentially the same class definition over and over again for each different form of list, or, alternatively, to write a single weakly typed class defintion where list elements are simply declared to be of type `Object` or `Comparable`. Neither approach is satisfactory. The first approach produces significant code duplication while the second sacrifices typing precision and forces the frequent use of type casts, which may fail at run-time if the programmer makes a clerical error.

To eliminate this dilemma, Java 5.0 introduced the concept of type variables and the parameterization of classes and method by type. A class/interface that is parameterized by one or more type variables is called a *generic* class/interface, while a method that is parameterized by type variables is called a *polymorphic* method.

The technical details of generic typing are rather arcane but nearly all of these details can be inferred from a few simple ideas:[13]

- The conventional types that we have previously discussed are considered type *constants* in the context of generic typing.
- Type variables are introduced at the beginning of generic class definitions and polymorphic method definitions just like ordinary variables are introduced at the beginning of a Scheme λ-expresssion. The scope of a type variable introduced at the beginning of a generic class is the remainder of the class definition; the scope of a type variablle introduced at the beginning of a polymorphic method the remainder of the method definition.
- In the body of a generic class, only type variables that are in scope can be accessed.
- Type variables can appear anywhere that conventional type constants can appear with four exceptions that will be explained later.
- Every type variable has a bounding type which can be either a class or an interface. if no bound is specified, the default bound is `Object`.
- In type-checking, type variables are *equal* to themselves and to their bindings if they are bound; otherwise no equality relationships exist on type variables. In addition, a type variable is a sub-type of its bound. No subtyping relationships other than these (and their transitive, reflexive consequences) hold.

The best way to explain the details of Java generics is to scrutinize some examples. We begin with a generically typed definition of functional lists including `map` as a primitive operation:

```
import java.util.NoSuchElementException;

/* A functional list containing elements of type E. */
abstract class List<E> {
  abstract <R> R accept(ListVisitor<E,R> v); // R is return type of v
  ConsList<E> cons(E elt) { return new ConsList<E>(elt, this); }
  abstract E get(int i);
  abstract <R> List<R> map(ILambda<E,R> f);   // R is return type of f
  abstract String toStringHelp();
}

/* The visitor interface for List<E>. R is the return type. */
```

[13]Except for a few glitches in the design of Java generics caused by *type erasure*.

```
interface ListVisitor<E, R> {
  R forEmptyList(EmptyList<E> host);
  R forConsList(ConsList<E> host);
}

/* A function in A->B represented as an object. */
interface ILambda<A,B> {
  B apply(A a);
}

/* An empty list.  Singletons are incompatible with generics. */
class EmptyList<E> extends List<E> {
  <R> R accept(ListVisitor<E, R> v) { return v.forEmptyList(this); }
  E get(int i) {
    return new NoSuchElementException("Attempted to get element " +
      i + " from empty list");
  }
  <R> List<R> map(ILambda<E,R> f) { return new EmptyList<R>(); }
  String toStringHelp() { return ""; }
  public String toString() { return "()"; }
  public boolean equals(Object other) {
    return getClass() == other.getClass();
  }
}

/* A non-empty list. */
class ConsList<E> extends List<E> {
  E first;
  List<E> rest;
  ConsList(E f, List<E> r) {
    first = f;
    rest = r;
  }
  <R> R accept(ListVisitor<E, R> v) { return v.forConsList(this); }
  E get(int i) { // a while loop would be more efficient
    if (i == 0) return first;
    else return rest.get(i-1);
  }
  <R> List<R> map(ILambda<E,R> f) {
   return rest.map(f).cons(f.apply(first));
  }
  String toStringHelp() { return " " + first + rest.toStringHelp(); }
  public String toString() {
    return "(" + first + rest.toStringHelp() + ")";
  }
  public boolean equals(Object o) {
    if (getClass() != o.getClass()) return false;
    ConsList c = (ConsList) o;
    return first.equals.c.first) && rest.equals(c.rest);
  }
}

class InsertVisitor<E extends Number>
  implements ListVisitor<E, List<E>> {

  E arg;
  InsertVisitor(E n) { arg = n; }
  public List<E> forEmptyList(EmptyList<E> host) {
    return new EmptyList<E>().cons(arg);
  }
  public List<E> forConsList(ConsList<E> host) {
    if (arg.doubleValue() <= host.first.doubleValue())
      return host.cons(arg);
    return host.rest.accept(this).cons(host.first);
  }
}

class MergeVisitor<E extends Number>
  implements ListVisitor<E, List<E>> {

  List<E> other;
  MergeVisitor(List<E> o) { other = o; }
  public List<E> forEmptyList(EmptyList<E> host) { return other; }
  public List<E> forConsList(ConsList<E> host) {
    return other.accept(new MergeHelpVisitor<E>(host));
  }
}

class MergeHelpVisitor<E extends Number>
  implements ListVisitor<E, List<E>> {

  ConsList<E> other;
  MergeHelpVisitor(ConsList<E> o) { other = o; }
  public List<E> forEmptyList(EmptyList<E> host) { return other; }
  public List<E> forConsList(ConsList<E> host) {
    E o1 = other.first;
```

```
        E h1 = host.first;
        List<E> oRest = other.rest;
        List<E> hRest = host.rest;
        if (o1.doubleValue() <= h1.doubleValue())
          return oRest.accept(new MergeHelpVisitor<E>(host)).cons(o1);
        else return hRest.accept(this).cons(h1);
      }
    }
```

Type Erasure and Raw Types Java does not include parametric type information the byte code generated by the compiler. Hence, a `List<Integer>` simply becomes a `List`. In effect, the Java compiler translates generic source code to non-generic byte code with weak typing and automatically inserts casts wherever necessary. For every generic type `Foo<A,B>` declared in a Java source program, the Java compiler produces byte code for a non-generic class `Foo`. In generated code for the body of the class `Foo`, the compiler replaces all references to the "naked" type variables `A` and `B` by their bounds. Similarly all references to parameterized types are replaced by the corresponding non-generic types (erasing the parameterization). To produce type-correct non-generic byte code, the compiler inserts casts wherever necessary.

For example, assume that a Java source program constructs a `List<Integer> ints`. In the byte code generated by the compiler, `ints` has type `List`. Assume that the compiler subsequenty computes `ints.get(0) + 1`. The static type of `ints.get(0)` is `Integer` which can be unboxed to an `int` and added to `1`. But the type of `ints.get(0)` in the byte code is `Object` because the byte code uses erased types. So the compiler generates code for `(Integer)(ints.get(0))` instead of `ints.get(0)`.

The compiler initially performs static type checking on the generic source code. If the code is type-correct, the compiler erases all generic type information from the source code and type checks the erased source code which typically generates some failures. At each failure point, the compiler inserts the cast required to restore type-correctness. *The generated casts are guaranteed never to fail provided that the original source program compiled without generating any "unchecked" type warnings (which are described below).* Unchecked type warnings are technically static type errors but they are sometimes necessary in well-written code because of the restrictions on program expressiveness imposed by type erasure.

The erased type (*e.g.*, `List` corresponding to a generic type *e.g.*, `List<Integer>`) is called a *raw* type. Java source code can contain raw types, although their use is considered bad form unless it is necessary to interface to ungenerified legacy code (*e.g.*, some Java core libraries) or it is forced by type erasure. From the perspectively of the Java type checker, the raw type corresponding to a generic type is a proper superset of each instantiation of the generic type.

Restrictions on the Use of Type Parameters There are some sensible operations at Java source code level that cannot be performed by the corresponding type-erased byte code because the execution of those operations depends on the erased information. The forbidden/restricted operations fall into the following five categories:

- Casts to generic (parameterized) types cannot be correctly implemented in Java because the requisite parametric information has been erased by the compiler. Nevertheless, such casts are often necessary in Java source code, particularly when interfacing old non-generic code with new generic code. As a result, Java permits such casts to appear in source code but trusts that the parametric information in such casts is correct. Java "implements" a cast to a generic type as a cast to the corresponding erased (non-generic) type and generates an "unchecked" type "warning". Programs that generate such warnings may fault at run-time on automatically generated casts in situations where the generic cast (properly implemented) would have failed. Consider the following example. Assume some legacy code creates a raw `List nums` and that the code for `List` has subsequently been generified so that its erased signature (the types of all its members) matches its original signature. Then new generic code can use `nums` but should cast it to the appropriate instance of the generic type `List<E>`. Assume that the elements of `nums` are supposed to have type `Integer`. Then the new generic code should cast `nums` to type `List<Integer>`. If some element of `nums`, *e.g.*, `nums.get(0)` turns out not to be an `Integer` and the new generic code attempts to use it as an `Integer`, the generated byte code will generate a `ClassCastException` because the byte tries to cast `nums.get(0)` to an `Integer`.

- Naked type variables (like `T`) cannot be used in `new` operations. In addition, Java prohibits `new` operations that create arrays where the element type is a nake type variable. For example, both `new T()` and `new T[0]` are illegal. Programs that attempt to use such `new` operations are rejected by the Java compiler as ill-formed.

- The types used in `instanceof` tests may not be generic (parameterized) because the queried parametric information is not available at run-time.

- The exception types used in `catch` clauses may not be generic (parameterized) because the parametric information required to match the clause is not available at run-time.

- Naked type variables may not appear as supertypes in either `extends` or `implements` clauses in the declaration of classes and interfaces.

Wild-card Types and Other Pathologies The generic type system has many dark, ugly corners that are best avoided. In particular, wild-card types (which we have not even mentioned) are troublesome. We suggest avoiding wild-card types in your own code until you are very comfortable with generic types. You cannot completely avoid wild-card types because some Java libraries include methods with wild-card types in their signatures. Fortunately, it is much easier to use wild-card types as a client than it is to include wild-card types in your code. A wild card type has the form `Foo<? extends E>` or `Foo<? super E>` where `Foo` is a generic type and `E` is some (perhaps parameterized) type expression. The type `Foo<? extends E>` means any type of the form `Foo<E'>` where `E'` extends `E`. Similarly, the type `Foo<? super E>` means any type of the form `Foo<E'>` where `E` extends `E'`.

The most common form of wild-card type is simply `Foo<?>` which means any instantiation of `Foo<T>`. `Foo<?>` is an abbreviation for `Foo<? extends Object>`.

Wild-cards allow methods to take a broader, more diverse collection of inputs. Assume that we want to add a `catenate` operation to the class `List<E>`. The obvious signature for `catenate` is:

```
/** Returns the concatenatoin of other and this. */
List<E> catenate(List<E> other>)
```

But the usual code for `catenate` works for any argument `other` of type `List<E'>` where `E'` extends `E` because the code copies `other`, rebuilding the `ConsList` nodes for `other`. If `other` consists of `ConsList` nodes of narrower type than `E`, it does not matter. Hence, the more generous type for `catenate` is:

```
List<E> catenate(List<? extends E> other>)
```

Narrowing Return Types In conjuction with supporting generic types, Java 5.0 also liberalizes the form of the return type in an overriding method. This return type can be any subtype of the return type of the overridden method. As a result methods that return the type of `this` in a superclass can often be overridden to return the type of `this` in subclasses.

Section 2.1.3 contains a vivid example of such narrowing in the context of sequence interfaces. In the interface `FinalSeq<E>`, which is a subtype of `Seq<E>`, the return type of the `cons` is narrowed from `Seq<E>` to `FinalSeq<E>`.

1.13.5 Mutability

In unrestricted Java programs, methods calls can modify the values of accessible fields. In addition, code blocks within method bodies can change the values of method parameters and local variables. These mutations can break critical invariants, such as the fact that Objects used as keys in a hash table must be immutable for hashed lookups to work correctly. For this reason, Java includes a variable modifier called `final` that can be used to mark variables that do not change. All of the fields, method parameters, and local variables in the Elementary and Intermediate levels of DrJava are implicitly `final` but in full Java, this property must be specified for each such variable at the point where it is declared.

In most cases, the `final` modifier is simply a clean way for programmers to declare that a variable is never modified and instruct the compiler to enforce this restriction, but there is one important usage of `final` variables discussed below where the omission of the `final` modifier can convert a well-formed program into a syntactically incorrect one.

Local Variables in Anonymous Classes

Java imposes an important restriction on the use of local variables in anonymous classes. Any local variable mentioned in an anonymous class definition must be declared `final`. In practice, this is not a significant restriction. In fact, most attempted uses of non-`final` local variables in anonymous classes correspond to errors in program logic. In any case where you need to mention a non-final local variable in an anonymous class, you can simply introduce an extra `final` variable with the required value. Ironically, this transformation often converts a logically incorrect program in to a correct one! We will revisit this issue in later in the monograph in connection with programming graphical user interfaces.

1.13.6 Other Java Language Features

We have covered all of the constructs in the core of Java. We will introduce a few more language constructs in the sequel, namely nested classes and concurrency features, as they are needed.

Chapter 2

Object-Oriented Data Structures

The traditional programming curriculum focuses far more attention on efficient algorithms than on clean program design. Nearly all existing books on data structures including those that use C++ and Java fail to apply good object-oriented design principles in presenting interesting data structures. In this chapter, we will show how to formulate some of the most common data structures in an object-oriented design framework. Enjoy!

2.1 Sequences

The first data structures that we will discuss are the common representations for sequences of elements. A sequence is an ordered collection $s_0, s_1, s_2, ..., s_{n-1}$, $n >= 0$ of data objects drawn from some base data type D. Sequence representations can be classified based on the set of operations that they support. In Java, it is convenient to describe these various sets of operations as Java `interface`s.

We have already used a common representation for sequences, namely functional lists, in many of our examples of functional programming in the preceding chapter. Before we explore general representations for sequences, we need to discuss a particular formulation of sequences called arrays, which are built-in to Java as they are in essentially all practical programming languages.

2.1.1 Arrays

An array is an indexed sequence $s_0, s_1, s_2, ..., s_{n-1}$, $n >= 0$ of data values of *fixed* length. In contrast to more general representations for sequences an array object cannot grow or shrink. It has a specific length $n >= 0$ when it is created and retains this length during its entire lifetime. Java arrays are almost identical to Scheme vectors; the only difference is that every array in Java has a declared type T`[]` asserting that all elements of the array have type T. All of the primitive operations on arrays in Java preserve the declared types attached to arrays. Given an array `a` of type T`[]`, the expresssion `a[`i`]` extracts the ith element of the array, which must be a value of type T. Array indexing begins with the integer value 0 as in Scheme, C, and C++. Hence, an array of length n has valid indices of $0, 1, ..., n - 1$.

The assignment operator `=` is used to update array elements. Given an array `a` of type T`[]`, the statement

```
a[i] = e;
```

updates the value of the ith element to the value of the expression e. The values of all other elements of `a` remain unchanged.

In Java, arrays of every type are built into the language. If you define a class or interface C, then the array type C`[]` is automatically supported by Java. Every array type is a subtype of type `Object`, but array types cannot be extended. In essence, Java arrays are identical Scheme vectors augmented by an enforced type constraint. If a Java variable has type `T[]` for some primitive or object type `T` then it can only be bound to arrays containing elements of type `T`. Every Java array has a `int` field `length` that contains the length of the array.

Since Java arrays are objects, a Java array value is actually a reference to an array. Hence, a variable of type T`[]` in Java can have the value `null`. A variable of type T`[]` can appear on the left-hand side of an assignment

```
a = e
```

where e is any expression of type T.

Since array values are references to arrays, two variables can be bound to exactly the same array. If array variables `a` and `b` are bound to the same array object, then updating an array of a

```
a[i] = e;
```

changes the value of `b[`i`]`. Scheme variables bound to vectors have precisely the same property.

Arrays are allocated in Java using the `new` statement just like other objects. The array form is

```
new T[length]
```

where T is any type and *length* is an expression that evaluates to a non-negative `int`. Arrays can have length 0. Each element of the array is set to the ‘zero” value of the declared type T. If T is an object type, the initial value of each element is `null`. Hence, the expression

```
new String[1000]
```

allocates an array of 1000 `String` elements all bound to `null`.

Java has alternate form for constructing an array called an anonymous array. The expression

`new` T`[]` $\{v_0, ...v_{n-1}\}$

allocates an array of length n of type T containing the specified sequence of elements.

Java uses two mechanisms to enforce the declared type T`[]` of an array object. First, when an array T`[]` is allocated, all of the initial element values must be of type T. Second, when an array object `a` of type T`[]` is updated by an assignment

`a[`i`] =` e

Java confirms that the the new value `e` for the ith element belongs to the type T. During program execution, the Java virtual machine checks that the value e belongs to type T by casting e to type T each time that the assignment is executed. If the check fails, the virtual machine throws a a `ClassCast` exception.

The array-assignment check must be performed at run-time because an array object of type $S[]$ can be stored in a local variable or field of type $T[]$ where S is a subtype of T. Hence, in the preceding array assignment, the declared (static) type T of `e` could match the declared type $T[]$ of the array variable `a`, yet the assignment could fail at run-time because the value of `a` is an array of type S where S is a proper subtype of T.

The Java type system permits an array variable of type T`[]` to be bound to an array A of type S`[]` provided $S \subseteq T$. This property of the Java type system is called *covariant subtyping.* Note that covariant subtyping implies that the array type `Object[]` is a supertype for *all* object array types, which permits arrays to be treated polymorhically in many situations.

Recipes for Processing Arrays

Arrays do not have the same internal structure as lists: an array of length n does not contain an array of length $n-1$ as a component. Hence, the structural design recipes for processing lists do not *directly* apply to lists. But it is easy to mimic the structural decomposition of lists using arrays as follows. Given an array

$$A = a_0, ..., a_{n-1}$$

we define the *slice*

$$A[j : k]$$

where $j \geq 0$ and $k \leq n$ as the sequence of elements

$$aj, ..., a_{k-1}$$

Since array slices are not arrays, we cannot pass them to helper functions as array values. Instead we must pass three values in the general case: the array A, and the two `int` bounds j and k. Fortunately, in most list processing recipes one of two bounds is always fixed (either $j = 0$ or $k = n$), enabling us to only pass two values.

Assume that we want write a static method `int sum` that takes an argument `a` of type `int[]` and returns the sum of the elements of `a`. If we were summing the elements of a list instead of an array, we could use the natural recursion scheme on lists to reduce summing a compound list (a `Cons`) to summing its tail (the `rest` component of the list). (See Section 1.5.2.) We can use *exactly* the same scheme to sum an array provided that we use array *slices* instead of array values. To process slices, we must write a helper method `sumHelp` that takes arguments `a` and `j` of type `int[]` and `int` respectively and returns the sum of the elements in the array slice `a[j:`n`>` where n is the length of the array a. An empty slice corresponds to the case where $j \geq n$. A compound slice corresponds to the case where $j < n$.

The following Java code implements the `sum` method

```
class ArrayUtil {
  /** returns a[0] + ... + a[a.length-1] */
  public static int sum(int[] a) { return sumHelp(a,0); }
  /** Requires: 0 <= j < a.length
    * @return a[j] +  ... +  a[a.length-1] */
  public static int sumHelp(int[] a, int j) {
    if (j >= a.length) /* array slice is empty */ return 0;
    else return a[j] + sumHelp(a, j+1);
  }
}
```

From the standpoint of computational efficiency, neither the structurally recursive program or the equivalent program on array slices written above is optimal because neither one is *tail-recursive*. A method definition is tail-recursive if recursive calls only appear in *tail-position*, the last operation before returning from the method. In Scheme, the standard recipe for converting such a computation to tail recursive form involves writing a help function with an *accumulator* and summing the elements in the opposite order (left-to-right instead of right-to-left). We can convert our array slice solution to tail-recursive form using the same transformation.

The following Java code implements a tail-recursive solution using array slices:

```
class ArrayUtil {
  /** returns a[0] + ... + a[a.length-1] */
  public static int sum(int[] a) { return tailSumHelp(a,0,0); }
  /** Requires: 0 <= j < a.length; accum = a[0] + ... + a[j-1]
    * @return accum + a[j] +  ... +  a[a.length-1] */
  public static int tailSumHelp(int[] a, int j, int accum) {
    if (j >= a.length) /* empty array slice */ return accum;
    else return tailSumHelp(a, j+1, accum+a[j]);
  }
}
```

In languages that do not support the efficient translation of tail-recursive procedures to machine code, tail recursive (also called *iterative*) computations must be expressed in the more restrictive framework of `for` and `while` loops to produce efficient code. A tail-recursive procedure is a more general framework for expressing iterative computations than structured loops! In contrast to structured loops, tail-recursive procedures gracefully accommodate iterations with complex exit conditions; each procedure `return` clause that is not a tail-recursive call is an exit. To translate the an iterative program expressed using tail recursion to one expressed using a loop, the corresponding loop construction must have multiple exit jumps (implemented as `break` or `go to`). Note: mutual tail recursion is much more expressive than loops with `break` or `go to` statements.

Java has three familiar looping constructs: `while` loops, `do ... while` loops, C-style `for` loops. The first two constructs are completely standard. A `while` loop has syntax:

```
while (test) do statement
```

where *statement* is usually a *block*. A *block* is simply a sequence of local variable declarations and statements enclosed in braces. The *test* expression must have boolean type. A `do while` loop has syntax:

```
do statement while (test);
```

The only difference between the two looping constructs is the obvious one. In a `while` loop the `test` is executed before the loop body is executed. In a `do while` loop the loop body is executed once before the *test* expression is evaluated.

The Java `for` loop is borrowed from C. It has the form

```
for (init-expr; test; incr-expr) statement
```

which simply *abbreviates* the following code fragment containing a `while` loop:[1]

```
init-expr;
while (test) { statement;
incr-expr; }
```

[1]With two minor exceptions when *`init-expr`* is a variable declaration and when `continue` appears in the loop body. If *`init-expr`* is a declaration, the scope the declaration is the loop (*`test`*, *`incr-exper`*, and *`statement`* above), not the remainder of the block containing the `for` loop. If the loop body contains `continue`, which means "jump back to the beginning of the loop", the *`incr-expr`* is executed before jumping back to the beginning of the loop.

Let us return to our tail-recursive Java program that sums the elements of an array. Fortunately, we can translate this tail-recursive procedure[2] directly to a simple `while` lop with `true` as the *`test`*. All that we have to do is embed the tail-recursive procedure body in a

```
while (true)  ...
```

wrapper, converting each tail-recursive call to a block of code that updates the procedure parameters to reflect values passed in the call [3] and then executes a `break` statement to jump back to the beginning of the loop. In the special case where the `break` statement occurs at the end of the loop body, it should be omitted. The procedure parameters must be declared in a prelude preceding the loop that properly initializes them in accord with the top level call on the tail-recursive function. The loop form for our summation example is:

```
class ArrayUtil {
  /** returns a[0] + ... + a[a.length-1] */
  public static int sum(int[] a) { return tailSumHelp(a,0,0); }
  /** Requires: 0 <= j < a.length; accum = a[0] + ... + a[j-1]
    * @return accum + a[j] +  ... +  a[a.length-1] */
  public static int sumHelp(int[] a, int j, int accum) {
    while (true) {
      if (j >= a.length) return accum;  // return result
      accum = accum + a[j];
      j = j+1;            // j must be modified last!
    } // tail recursive call
  }
}
```

Since the loop-based help function is not recursive, it can be "inlined" back into the top-level function. In our summation example, the top-level function becomes:

```
class ArrayUtil {
  /** returns a[0] + ... + a[a.length-1] */
  public static int sum(int[] a) {
    int j = 0;
    int accum = 0;

    while (true) {
      /** Invariant: 0 <= j < a.length; accum = a[0] + ... + a[j-1] */
      if (j >= a.length) return accum;
      accum = accum + a[j];
```

[2]And any other singly recursive tail recursive procedure.

[3]Avoiding interference from side effects!

```
          j = j+1;
        }
      }
    }
```

The single exit loop in our example can be rewritten as a conventional `for` loop as follows:

```
    class ArrayUtil {
      /** returns a[0] + ... + a[a.length-1] */
      public static int sum(int[] a)
         int accum = 0;
         for (int j = 0; j < a.length; ++j) {
          /* Invariant: accum == a[0] + ... + a[j-1] */
          accum = accum + a[j];
        }
        return accum;
      }
    }
```

The expression `++j` is an abbreviation for

```
    j = j+1
```

The resulting program uses the most common idiom in imperative programming: the `for` loop. This form of processing forms the basis for the most commonly used imperative design pattern: the *iterator* pattern. We will discuss this pattern in detail in Section 2.1.8.

We now turn our attention to more general data representations for sequences that accommodate operations that change sequence length.

Binary Search

Even though most operations that mutate arrays–other than assigning a new value to an existing element–are comparitively slow, (*e.g.*, inserting a new element at index i (which requires moving many array elements and perhaps copying the array contents to a larger array), most operations on arrays involving no mutation can be done very efficiently. One the most important functional operations on arrays is *binary search*. This operation presumes the array holds a collection of uniform entries (each occupying one element of the array) in sorted order. Given an array containing a collection of n items, binary search determines where a given key k appears in the collection and returns its index in $O(logn)$ time.

The idea behind binary search is simple: divide the current array slice (initially the entire array) into two nearly equal slices and conquer. Given a key k and a slice consisting of n elements starting at index i, binary search compares k with the value of the key at index $i + (n \; div \; 2)$ (where $n \; div \; 2$ truncates $n/2$ if n is odd). If k is less than this key, then a entry with key value k must lie in array slice $[i \; : \; i+(n \; div \; 2)-1]$; otherwise it must lie in the slice $[i + (n \; div \; 2) - 1 \; : \; i + n - 1]$.

2.1.2 Lists

By convention, linked representations of sequences are called Lists. This representation of sequences is so pervasive that the terms sequence and list are often used interchangeably (conflating the abstraction and the implementation). A particularly important distinction between sequence interfaces is whether or not an interface includes operations that *mutate* (destructively modify) the object `this`. For example, if a sequence object `x` contains the `Strings` `"Corky"` and `"Matthias"`, does any operation on the object `x` permit the *contents* of `x` to be modified, *i.e.,* changing, adding, or subtracting elements? Operations on `x` that construct new sequences that incorporate the contents of `x` are not *mutators* because the object `x` is left unchanged.

Immutable data types are easier to define, to use, and to implement than mutable data types, but they they have two important limitations. First, they do not support some computations efficiently. Second, object mutation plays a critical role in the natural modeling of some computational problems. The functional model of computation that we studied in Chapter 1 is exclusively concerned with immutable data. The term "functional list" is synonymous with "immutable list".

We will focus first on immutable sequences and their representations. Then we will investigate what adjustments must be made to support mutation.

2.1.3 Immutable Sequences

All of the sequence classes presented in this monograph—immutable and mutable—support the operations in the following `Seq` interface

```
/** Represents a sequence s[0], ..., s[n-1] of elements of type E */
interface Seq<E> {
  /** @return an empty sequence. */
  Seq<E> empty();

  /** @return the Seq with elts newElt, s[0], ..., s[n-1]
  Seq<E> cons(E newElt);

  E first();          // returns the element s[0]
  Seq<E> rest();      // returns an object representing s[1], ..., s[n-1]
  E eltAt(int i);     // returns the element s[i]
  boolean isEmpty();  // returns n == 0
```

```
    public R accept(SeqVisitor<E,R> host);  // applies the visitor code host
  }

  /** Visitor interface for an operation on Seq<E> returning type R. */
  interface SeqVisitor<E,R> {
    R forEmpty(Seq<E> host);
    R forCons(Seq<E> host);
  }
```

The contracts for all of these operations stipulate that they do not modify the observable state of a sequence object.

Immutable sequence classes also support the two additional functional operations in the following interface:

```
  /** Represents a sequence s[0], ..., s[n-1] of elements of type E. */
  interface FinalSeq<E> extends Seq<E> {
    /** New signature narrows return type. */
    FinalSeq<E> empty();

    /** New signature narrows return type. */
    FinalSeq<E> cons(E newElt);

    /** @return val,s[1], ..., s[n-1] */
    FinalSeq<E> updateFirst(E val);
    /** @return s[0],r[0], ..., r[m-1]
    FinalSeq<E> updateRest(Seq<E> r);
  }
```

The two new operations return new sequence values; they do *not* modify `this`.

There are two widely used representations for immutable sequences: *linked* and *contiguous*.

Linked Representation

In the *linked* representation, a sequence is a (reference to) an object, which is either an *empty node*, representing the empty sequence, or a *cons node* with a field of type `E` containing the first element of the sequence and a field of type *Seq¡E¿* containing a pointer to the first node in the rest of the sequence. This data representation, which is often called a *linked* list, directly corresponds to the standard inductive definition of sequences. We defined this sequence representation in Section 1.10.5 as the class `List`, but that definition did not support all of operations listed above. The following modification of the `List` composite class hierarchy from Section 1.10.5 defines a linked representation for lists of objects; it includes all of the `FinalSeq` operations:

```
abstract class List<E> implements FinalSeq<E> {

  /* function methods */
  public FinalSeq<E> empty() { return new Empty<E>(); }
  public FinalSeq<E> cons(E first)  { return new Cons(first, this); }
  public abstract E first();
  public abstract Seq<E> rest();
  public abstract E eltAt(int i);
  abstract public boolean isEmpty();

  /* updaters (no mutation!) */
  public abstract FinalSeq<E> updateFirst(E f);
  public abstract FinalSeq<E> updateRest(E r);

  /** @return String rep for this w/o parentheses and leading blanks */
  abstract String toStringHelp();

  private static class Empty<E> extends List<E> {

    /* constructor */
    private Empty() {}

    /* methods */
    public E first() {
      throw new IllegalArgumentException("first() applied to empty list");
    }
    public Seq<E> rest() {
      throw new IllegalArgumentException("rest() applied to empty list");
    }
    public int isEmpty() { return true; }

    public Seq<E> updateFirst(E o) {
      throw new IllegalArgumentException("updateFirst() applied to empty list");
    }
    public Seq<E> updateRest(Seq<E> s) {
      throw new IllegalArgumentException("updateFirst() applied to empty list");
    }
    public E eltAt(int i) {
      throw new IllegalArgumentException("out-of-bounds index in List.eltAt");
    }
    public E accept(SeqVisitor<E,R> v) { return v.forEmpty(this); }
    public String toString() { return "()"; }
    public String toStringHelp() { return ""; }
  }

  class Cons<E> extends List<E> {

    /* fields */
    private final E first;
    private final List<E> rest;

    /* constructor */
    Cons(E f, List<E> r) {
      first = f;
      rest = r;
    }

    /* methods */
    public E first() { return first; }
    public Seq<E> rest() { return rest; }
    public int isEmpty() { return false; }

    public E eltAt(int i) {
      if (0 == i) return first;
      else return rest.eltAt(i-1);
    }
    public E accept(SeqVisitor<E,R> v) { return v.forCons(this); }

    public Seq updateFirst(E o) { return rest.cons(o); }
    public Seq updateRest(Seq<E> r) { return r.cons(first); }

    public String toString() { return "(" + first + rest.toStringHelp() + ")"; }
    String toStringHelp() { return " " + first + rest.toStringHelp(); }
  }
}
```

The definition of the `List` class contains `static` *nested* class definitions for the classes `Empty` and `Cons`. The `static` attribute identifies these classes as simple *nested* classes rather than `inner` classes. Recall that static nested classes are identical to conventional "top-level" classes except for two minor differences.

- First, static nested classes have qualified names (of the form *containing-class-name.nested-class-name*).[4] The full names for the classes `Empty` and `Cons` are `List.Empty` and `List.Cons`, respectively. Within the body of the `List` class, the unqualified names `Empty` and `Cons` are synonymous with the qualified names.

[4]Since classes may be nested to any depth, multiply nested classes have multiple qualifiers, one for each level of nesting.

- Second, static nested classes can be declared as `private`, making their names invisible outside the body of the containing class. Hence, the class names `List.Empty` and `List.Cons` are not defined outside of the body of the `List` class. If we removed the `private` attribute for the `Empty` class above, then the classes `List.Empty` and `List.Cons` are visible outside of class `List`.

In contrast to instances of inner classes, instances of static nested classes *do not have enclosing instances*. Section 2.1.7 discusses nested and inner classes in more detail.

The preceding collection of classes does not exploit the singleton pattern (creating a single shared instance of `Empty`) because generic typing in Java does not support assigning multiple types to a shared object. By breaching the type system, we can exploit the singleton pattern but the resulting code will always generate a *unchecked* type warning.

Finger Exercise Revise the code for `List` class to use the singleton pattern by declaring a static final field `EMPTY` of type `Empty`. The method `empty()` returns `EMPTY` cast to the requiste generic return type; this cast generates a *unchecked* warning.

The `for..` methods in the `SeqVisitor` interface all take a host argument of type `Seq` because the implementation is not constrained to use the composite pattern to represent immutable sequences. The following visitor class implements sequence concatenation:

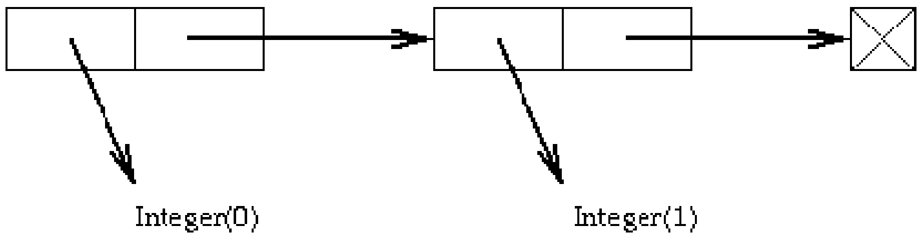
```
/** A purely functional visitor method that returns host || that. */
class Append implements SeqVisitor<E,R> {

  /* fields */
  Seq<E> that;

  /* constructor */
  Append(Seq<E> t) { that = t; }

  /* methods */
  public R forEmpty(Seq<E> host) { return that; }
  public R forCons(Seq<E> host) {
    return host.updateRest(host.rest().accept(this));
  }
}
```

The following figure shows a picture of linked list of integers.



In the figure, the nodes with two fields are `Cons` instances, and the crossed-box is an `Empty` instance. References (pointers) are represented by the heavy arrows. The reference fields in the cells are in fact memory addresses. In Java, these addresses are always interpreted as references to objects. Java only supports operations on references that are consistent with this abstraction, *e.g.* you cannot perform arithmetic on a reference. In lower-level languages like C and C++, references (pointers) can be manipulated as ordinary integers.

Finger Exercise Write a `SeqVisitor` class to reverse a `Seq`. Test your code using DrJava. Hint: introduce a helper visitor with an extra parameter to accumulate the result. ■

Contiguous Representation

In the *contiguous* representation, a sequence of type `T` is represented by a reference to an immutable array of fields of type `T`. An immutable array is an array that, once initialized, is never modified. Java doesn't directly support immutable arrays, but a Java program can enforce immutability by defining a *wrapper* class for arrays (akin to the `Integer` class for wrapping `int`s in objects) with a single `private` field holding the embedded array and a collection of `public` methods that do not mutate this field. A lighter weight but less robust protocol for supporting immutable arrays is to use comments to indicate which arrays are immutable and to follow the discipline of never modifying arrays documented as immutable. In either case, a new array object generally must be created whenever an element of the represented sequence is changed, added, or removed. Creating an array object is a costly operation proportional in time and space to the number of elements in the new array. Inserting an deleting elements of an array are generally expensive as well because elements must be moved (and in some cases a new array must be constructed).

In the linked representation of sequences, every operation in the collection listed above except `eltAt` can be performed in constant time. On the other hand, the `eltAt` operation takes time proportional to the length of the sequence in both the worst case and the typical case. The `List` implementation of sequences given in chapter 1 has this property.

The performance trade-offs embodied in the immutable array implementation are very different. In this implementation, the operations `empty`, `first`, `length`, `eltAt`, can be performed in constant time. (In the Java array implementation, `length` is stored in a separate "field" in the block of storage holding the array.) With the exception of the `rest` operation, the remaining operations *all* take time proportional to the length of the sequence. The running time of the `rest` operation depends on an interesting implementation detail. If immutable arrays are implemented as instances of a "wrapper" class, then the `rest` operation can be performed in constant time at the cost of making an extra field reference in the implementation of `eltAt`. A wrapper object can store an integer `offset` that is added to the index passed as an argument to `eltAt`. In this scheme, the `rest` operation constructs a new wrapper object containing a pointer to the same array object as `this` but an increased `offset` (by one). If immutable arrays are implemented directly by Java arrays, then `rest` operation must construct a completely new array one element shorter than the original.

Finger Exercise Construct two implementations of an `ImmutArray` wrapper class that represents sequences as arrays. Do not include an offset field in the first implemenation. Include a offset field in the second implementation. Test the `Append` and `Reverse` visitors written in the context of the linked representation above and your contiguous implementations. Conduct some experiments to measure the performance impact of including the offset pointer. For each implementation, can you devise a test program that favors it? ■

In practice, array representations of immutable sequences are generally not used in computations that make extensive use of the `cons` and `empty` operations to construct new sequences. The repeated copying of arrays required to support these operations is very inefficient (proportional to the square of the length of the constructed array!)

Finger Exercise Let n be the length of a sequence `host` represented either as a linked list or an array. Prove that the computation

```
host.accept(new Append(host.empty()))
```

runs in time $O(n)$ in the linked representation, $O(n^2)$ in the contiguous representation (with or without an offset field). ■

The usual way to avoid this source of inefficiency is to include an operation in the immutable array class that constructs an array representation for a sequence given either a corresponding linked representation or mutable array representation. The Java Foundation classes include both the immutable string (sequence of `char`) class `String` and the mutable string class `StringBuffer` for this reason.

The array implementation of immutable sequences is a good choice when new sequences are generally built from scratch rather than constructed by applied operations to existing sequences. For this reason, many computations involving immutable sequences of characters (strings) rely on the array representation. The Java `String` class implements the array representation for immutable sequences of characters (the primitive type `char` in Java). Note that Java includes an operation for converting mutable strings (represented by the Java class `StringBuffer`) to immutable strings (represented by the Java class `String`). Strings can be incrementally constructed from characters far more efficiently using the `StringBuffer` class that they can be using the `String` class.

2.1.4 Mutable Sequences

A sequence implementation is mutable if it includes operations that modify the value of `this`. A class representing mutable sequences typically a set of operations like the following:

```
interface SeqObject extends Seq {
  /** Changes the contents of this from s[0],...,s[n] to val,...,s[n]. */
  void setFirst(T f);          // this = this.updateFirst(f)
  /** Changes the contents of this from s[0] * rest to s[0] * r. */
  void setRest(Seq r);         // this = this.updateRest(r)
  /** Changes the contents of this sequence s to v. */
  void set(Seq v);
  /** Changes contents of s[i] in this sequence s[0],...,s[n] to val. */
  void setEltAt(int i, T val);
  /** Changes contents of this from s[0],...,s[n] to elt,s[0],...,s[n]. */
  void insert(Object elt);
  /** Removes s[0] (the first elt) from this sequence s. */
  void remove();               // removes s[0] from this
}
```

As with immutable sequences, there are two basic implementation schemes for mutable sequences: linked and contiguous. Mutation complicates the implementation of linked representations, which we examine in detail below.

Singly-linked Mutable List Representation

The various linked mutable list representations that we will study are all derived from the standard linked representations for immutable sequences. A particularly simple approach to sequence mutation is to represent a mutable sequence as a *variable* `x` of immutable list type and implement mutation by assigning a new value to the variable, *e.g.*

```
List x = empty();
.
.
.
x = cons(0,x);
```

But this approach fails to represent mutable lists as *objects* and to encapulate list mutation as an ordinary method. This representation cannot implement the `insert` method given above. In the list container representation, the `insert` operation modifies the program *variable* representing the mutable sequence, but variables are not objects! When we pass an immutable lists represented by an assignable variable as a method arguments, we can only pass the immutable list value to which the variable is bound.

Finger Exercise Try to write an insert method for mutable sequences represented by variables bound to immutable sequences. What goes wrong? ■

2.1.5 List Containers

A better approach is to define a *container class* with a single field that holds an immutable sequence. Then we can update the mutable sequence by modifying the contents of the field in the container object. For example, suppose we have a class `List` that defines a list representation for immutable sequences. The following container class works for any implementation of the `Seq` interface:

```
/** A mutable cell containing a Seq. */
class ListBox implements SeqObject {

  /** An instance of the class implementing Seq. */
  private static Seq empty = List.EMPTY;

  /** The contents of ListBox: s[0],s[1],...,s[n-1] */
  private Seq value;

  /* constructors */
  ListBox() { value = empty; }
  ListBox(Seq v) { value = v; }

  /* visible accessor methods */
  public Seq empty() { return new ListBox(); }
  public Seq cons(Object newElt) { return new ListBox(value.cons(newElt)); }
  public Object first() { return value.first(); }
  public Seq rest() { return value.rest(); }
```

```
    public Object eltAt(int i) { return value.eltAt(i); }
    public boolean isEmpty() { return value.isEmpty(); }

    /* visible mutator methods */
    public void setFirst(Object o) { value = value.updateFirst(o); }
    public void setRest(Seq r) { value = value.updateRest(r); }
    public void set(Seq v) { value = v; }  // set contents of box to v;
    public void setEltAt(int i, final Object val) {  // changes s[i] to val
      return accept(new UpdateEltAt(i), val);
    }
    public void insert(Object o) { value = value.cons(o); }
    public void remove() { value = value.rest; }

    /** Applies visitor v to value and return result; value is UNCHANGED. */
    public Object accept(SeqVisitor v) { return value.accept(v); }

    /* inner classes */
    private class UpdateEltAt implements SeqVisitor {
      /* fields */
      int index;          // index of element to be updated
      Object eltValue;    // new value for updated element

      /* constructor */
      UpdateEltAt(int i, Object e) { index = i; eltValue = e; }

      /* visit methods */
      Object forEmpty(Seq host) { throw
        new IllegalArgumentException("out-of-bounds index in UpdateEltAt");
      }
      Object forCons(Seq host) {
        if (index == 0) return new Cons(val, host.rest());
        else return host.rest().accept(new UpdateEltAt(i-1));
    }
  }
```

The variable holding the `Seq` instance is now wrapped inside an instance of a class (`ListBox` above) implementing the `SeqObject` interface. A method that accepts a `SeqObject` as an argument can modify it.

In the preceding example, the use of the inner class `UpdateEltAt` warrants careful study. This inner class is a private member of the `ListBox` class and hence inaccessible outside of the class body. Since the inner class referenced only within the `setEltAt` method, we could have placed the definition of `UpdateEltAt` as a declaration inside the body of this method! In this case, the `UpdateEltAt` class would have been visible only within the method `setEltAt` But such an embedded inner class definition can be hard to read, so we elected to make a private member of the `ListBox` instead.

We have already seen a special case of inner classes, namely anonymous classes defined inside dynamic methods. The only difference between an inner class and a conventional class is that every instance of an inner class has an enclosing instance of the class C that textually contains its definition. The free variables in the inner class refer to this class. In addition, the notation C`.this` refers to the "entire" enclosing instance. Inner classes impose exactly the same restrictions on references to local variables of the enclosing instance as anonymous classes do: any such local variable must be declared `final`. Inner classes are discussed in more detail in Section 2.1.7.

In `ListBox` class, the methods `insert`, `remove`, and `set` modify the receiver of the method invocation, so there is no need for them to return values. Consequently they have return type `void`. Mutator methods typically have the return type `void` because they embody *commands* that modify objects rather than *functions* that compute new values based on the value of `this`.

The "lists as containers" representation of mutable lists is a very simple example of the `state` pattern. In the state pattern, a mutable object contains a field of union type (denoted by an abstract class or interface) representing the state of the object. The object can easily change "shape" by updating the field to contain an instance of a different class in the union. In the `ListBox` class, an empty list object can mutate to a non-empty list object (or vice-versa) by modifying the contents of the `value` field containing a `Seq`, which is a union type.

Since the `SeqObject` interface extends the `Seq` interface, it inherits the *visitor* interface from the immutable `Seq` interface. As a result, no visitor class implementing the `SeqVisitor` interface can mutate a `SeqObject`! In particular, to mutate fields of a `ListBox` object, we must use explicit assignment. Given a `ListBox l` and `SeqVisitor v` that returns a `ListBox`, the assignment

```
l = (ListBox) l.accept(v);
```

updates `l` to the new value returned by the visitor operation.

The efficient operations on `ListBox` objects are precisely the efficient operations on the underlying functional `List` class, namely, adding and removing elements at the front of the list. Mutable lists in which elements can only added or removed at the front are called *stacks* or *LIFO* ("last in, first out") *lists*. Representing mutable lists as containers holding immutable list values is well-suited to this form of usage. The operations `first`, `insert`, and `pop` precisely match the usual operations `push`, `top`, and `pop` on stacks.

The "container" representation for mutable lists is simple and easy-to-use but it is poorly suited to many applications because it fails to support certain list operation efficiently. This representation forces list nodes (Cons objects) to be recopied whenever the list is changed. To modify the list element with index i or insert and element in front of the list element with index i, a computation must construct a new `List`, copying the elements from the old `List` with indices less than i. We can avoid this recopying process and avoid the potentially confusing distinction between immutable list values and mutable list objects by using a mutable variant of functional lists developed by Dung Nguyen and Steve Wong.

2.1.6 Quasi-Functional Lists

From the perspective of the public interfaces, quasi-functional lists differ from lists as containers in two respects. First, quasi-functional lists require the list arguments for `setRest` and `set` to be mutable list objects rather than immutable list values. Second, quasi-functional lists support visitor operations that mutate list structure in addition to the "purely functional" visitor operations corresponding to the `SeqVisitor` interface. To capture these differences in the Java type system, we introduce two new interfaces: a new mutable sequence interface called `MutSeq`:and a mutable visitor interface called `MutSeqVisitor`:

```
interface MutSeq extends Seq {
  void setFirst(Object f);         // changes this.first = f
  void setRest(MutSeq r);          // changes this.rest = r
  void set(MutSeq m);              // changes this = m
  void setEltAt(int i, Object val);   // changes this[i] = val
  void insert(Object o);           // changes this.first,this.rest = o,this
  void remove();                   // changes this = this.rest
  Object accept(MutSeqVisitor m);  // applies visitor operation m to this
}

interface MutSeqVisitor {
  Object forEmpty(MutSeq host);
  Object forCons(MutSeq host);
}
```

The `MutSeq` interface stipulates that the arguments to the operations `setRest` and `set` must be list objects, (objects of type `MutSeq`) rather than the list values (objects of type `Seq`) given in the `SeqObject` interface. The `MutSeq` interface also introduces an `accept` operation to support visitor operations (objects of type `MutSeqVisitor`) that mutate list structure. The `MutSeqVisitor` interface differs from the `SeqVisitor` interface in one key respect: the host object must be a mutable list (object of type `MutSeq`) rather than a list value (object of type `Seq`) enabling a visitor to destructively modify the host. This `MutSeqVisitor` interface is not applicable to lists as containers because the component `rest` fields embedded in the immutable list value are not mutable! The pivotal difference between the `QuasiList` and `ListBox` classes is the type of the `rest` field. A `MutSeqVisitor` can destructively modify both the `first` and `rest` fields of the host by using `MutSeq` mutator methods `setFirst` and `setRest`.

```
class QuasiList implements MutSeq {

  /* fields */
  public static final Empty EMPTY = new Empty();

  private List value;

  /* constructor */
  QuasiList() { value = new Empty(); }
  private QuasiList(List v) { value = v; }

  /* visible methods */
  Seq empty() { return new QuasiList(); }
  Seq cons(Object newElt) { return new QuasiList(value.cons(newElt)); }
  Object first() { return value.first(); }
  Seq rest() { return value.rest(); }
  // rest returns a MutSeq (QuasiList) but the Seq interface mandates
  //   the weaker type!

  Object eltAt(int i) { return value.eltAt(i); }
  boolean isEmpty() { return value.isEmpty(); }
  void insert(Object o) { value = new Cons(o, new QuasiList(value)); }

  public String toString() { return value.toString(); }

  void setFirst(Seq v) { value = value.updateFirst(v); }

  void setRest(MutSeq m) { value = value.updateRest(m); }

  void set(MutSeq m) { value = m.value; }

  void setEltAt(int i, final Object val) {
    /* inner class */
    class UpdateEltAt implements MutSeqVisitor {
      /* fields */
      final int index;       // index of element to be updated

      /* constructor */
      UpdateEltAt(int i) { index = i; }

      /* visit methods */
      Object forEmpty(MutSeq host) { throw
        new IllegalArgumentException("out-of-bounds index in UpdateEltAt");
      }
      Object forCons(MutSeq host) {
        if (index == 0) return host.setFirst(val);
        else return host.rest().accept(new UpdateEltAt(index-1));
      }
    }
    accept(new UpdateEltAt(i));
  }

  void remove() { value = value.rest(); }

  Object accept(SeqVisitor v) { return value.accept(v); }
  // apply visitor v to value and return result; value is UNCHANGED

  Object accept(MutSeqVisitor v) { return value.accept(v,this); }
  // apply visitor v to value and return result; value may be CHANGED

  /* inner classes */
  private interface List {
    abstract String toStringHelp();
    // List -> String without any parentheses and leading blanks
  }

  private class Empty extends List {

    /* constructor */
    private Empty() {}

    /* methods */
    Object first() {
    throw new IllegalArgumentException("first() applied to empty list");
    }
    MutSeq rest() {
    throw new IllegalArgumentException("rest() applied to empty list");
    }
    Object eltAt(int i) {
      throw new IllegalArgumentException("out-of-bounds index in List.eltAt");
    }
    Object accept(SeqVisitor v) { return v.forEmpty(this); }
    Object accept(MutSeqVisitor v) { return v.forEmpty(QuasiList.this); }
    public String toString() { return "()"; }
    public String toStringHelp() { return ""; }
  }
```

```
    private class Cons extends List {

      /* fields */
      private final Object first;
      private final MutSeq rest;

      /* constructor */
      Cons(Object f, List r) {
        first = f;
        rest = r;
      }

      /* functional methods */
      Object first() { return first; }
      Seq rest() { return rest; }
      /* MutSeq is the correct output type but Java does not support it */

      Object eltAt(int i) {
        if (0 == i) return first;
        else return rest.eltAt(i-1);
      }

      /* mutator methods */
      void setFirst(Object o) { first = o; }

      Object accept(SeqVisitor v) { v.forCons(this); }
      Object accept(MutSeqVisitor v) { v.forCons(QuasiList.this); }

      public String toString() {
        return "(" + first + rest.toStringHelp() + ")"; }
      String toStringHelp() { return " " + first + rest.toStringHelp(); }
    }

  }
```

The QuasiList implementation given above uses the state pattern to represent each tail (rest component) of the list. Each tail is a quasilist object that can mutate between two forms: empty and non-empty. Since each tail is a mutable object supporting the state pattern, a MutSeqVisitor can modify the state of any tail in the process of traversing a list.

The QuasiList code uses inner classes to hide the classes implementing the state of a QuasiList and to eliminate passing the QuasiList host as an extra parameter to the accept(MutSeqVisitor v) methods in the Cons and Emtpy subclasses of List.

If the `List` class is moved outside of `QuasiList`, the `QuasiList` object containing a given `List` object is not accessible to the `List` object.[5]

Mutating Visitors

Quasi-functional lists are more flexible than lists as containers because the *MutSeq* interface includes support for visitor operations that mutate the structure of a list. The following visitor implements the operation of destructively inserting an element at the rear of the sequence:

```
/** MutSeqVisitor class that inserts an element at the rear of the host.
  * Given the embedded Object elt and a host with elements s[0], ..., s[n],
  * invoking host.accept(this) destructively updates host so that host = s[0],
  * ..., s[n],elt.
  */
class InsertRear implements MutSeqVisitor {

  /* field */
  private Object elt;

  /* constructor */
  InsertRear(Object e) { elt = e; }

  Object forEmpty(MutSeq host) {
    host.insert(elt);
    return null;  /* dummy return value; the return type is Void */
  }

  Object forCons(MutSeq host) {
    ((MutSeq) host.rest()).accept(this);
    return null;  /* dummy return value; the return type is Void */
  }
}

/** MutSeqVisitor class that inserts an element at the rear of the host.
  * Given the embedded MutSeq tail with elements t[0], ..., t[n] and a host
  * with elements s[0], ... s[n], host.accept(this) destructively
  * updates host so that host = s[0],..., s[n],tail[0],...tail[m]
  */
class MutAppend implements MutSeqVisitor {
```

[5]Another alternative is add an `owner` field to the abstract class `List` that refers to the containing `QuasiList` object but this approach complicates the form of the constructors for `Cons` and `Empty`, which must take an additional argument to initialize the `owner` field.

```
    /* field */
    private MutSeq tail;

    /* constructor */
    MutAppend(Object t) { tail = t; }

    Object forEmpty(MutSeq host) {
      host.set(tail);
      return host;  /* dummy return value; the return type is void!
    }

    Object forCons(MutSeq host) {
      return ((MutSeq) host.rest()).accept(this);
    }
  }
```

The primary disadvantage of quasi-functional lists is that sharing list tails between two list objects can produce unexpected results when list objects are mutated. Mutating a shared list tail changes all of the list objects that share that tail! In the programming literature, the sharing of mutable data objects is often called "aliasing".

2.1.7 Nested Classes vs. Inner Classes

A *nested* class is a class whose definition appears inside the definition of another class, as if it were a member of the other class. For example, if a program contains

```
class A {
  class B {
    // fields, methods of class B...
  }
  // fields, methods of class A...
}
```

then class `B` is a nested class of class `A`. Code outside of the methods of class `A` can refer to class `B` by calling it `A.B`, using the same dot notation as for field and method references. Within the methods of class `A` class `B` can be used without qualifying the name. `B` could be hidden from code outside of class `A` by declaring it `private`, just as with fields and methods.

A nested class like `B` is known as an *inner* class. An inner class has access to the fields of an *instance* of its enclosing class. For this reason, an instance of `B` can only be created in association with an instance of `A`, using the expression

```
  new B(...)
```

inside of `A`'s methods, and the ugly notation

```
instanceA.new A.B(...)
```

outside of `A`'s methods, where *instanceA* is some instance of `A`. We strongly recommend avoiding the more complex notation. If a particular usage of inner classes requires this notation, then you should seriously consider refactoring your design to eliminate it.

An instance of `B` has access to the full enclosing instance of `A` and can refer to it using the expression

```
A.this
```

We can think of an instance of an inner class as having two `this` references, one for itself and one for its enclosing instance.

Note that an inner class is implicitly generic if the containing class is generic. An inner class falls within the scope of any enclosing generic type parameters.

An inner class may be nested within another inner class, so an inner class can even have multiple levels of `this` references. Nesting inner classes more deeply than one level is quite uncommon, however, and should generally be avoided.

As discussed earlier, a nested class can be declared `static`, in which case there is no enclosing instance. A static nested class can only access `static` members of its enclosing class. It is semantically equivalent to a top-level class. For this reason, static nested classes are also known as *nested top-level* classes. To repeat: static nested classes are exactly like classes declared outside any other class, except for the way they are named (with the enclosing class name as a qualifier) and for the fact that their visibility can be restricted.

Note that static nested class do *not* fall within the scope of enclosing generic type parameters. Why? Because static nested classes are top level classes. A static nested class merely has a *name* that is only in scope within the enclosing class. It has exactly the same semantics as the corresponding top-level class.

We will see good uses for both static nested classes and inner classes when we present the full implementation of imperative lists.

2.1.8 Extended Mutable Lists

Both of the preceding representations of mutable sequences—lists as containers and quasi-functional lists—are inefficient at inserting elements at the rear of a sequence. In each of these representations, the code for the operation must scan the entire sequence to reach the end. The container representation is particularly inefficient in this regard because the entire sequence must be reconstructed starting with a singleton `List` containing the new element.

Mutable sequence implementations that efficiently support adding elements at rear of the list and removing them from the front are important because this access protocol, called a *queue* or a *FIFO (first in, first out) list*, frequently arises in practice. Procedural formulations of linked lists discussed in traditional textbooks on data structures provide constant-time access to the end of the list by maintaining a “pointer” to the last node of the list. This strategy is conceptually simple but prone to coding errors because the empty list state requires special treatment. Moreover, the traditional procedural approach to representing lists exposes the concrete data structures (nodes and links) used in the implementation. Hence, every client of an imperative queue must cope with the empty state as a special case./footnoteUnless the current queue is provably non-empty. Mistakes in reasoning about the emptiness of queues is common source of coding errors.n

We can exploit the same representation strategy in an object-oriented implementation of lists that hides the concrete data structures in private object fields—provided that we deal carefully with the potentially troublesome “boundary” cases in the definition of list operations that involve the empty list.

Formulating Traditional Linked Lists as Objects

The quasi-list representation of mutable sequences includes an extra level of object nesting in the representation of list tails beyond what is present in the conventional “singly-linked list” representations that are widely used in procedural programming. A major disadvantage of this data representation is the extra memory required to hold the extra object allocated for each node. The basic singly-linked list representation avoids this extra overhead; it relies on the exactly same data representation as the “lists as containers” representation given in Section 2.1.5 with one critical modification: the `first` and `rest` fields of `Cons` objects are mutable. The following Java code implements the `MutSeq` interface using conventional singly-linked lists rather than quasi-lists.

```
class MutList implements MutSeq {

  /* fields */
  static final Empty EMPTY = new Empty();  // singleton empty list

  List value;

  /* constructors */
  MutList() { value = EMPTY; }
  private MutList(List v) { value = v; }

  /* visible methods */
  Seq empty() { return new MutList(); }
  Seq cons(Object newElt) { return new MutList(value.cons(newElt)); }
  Object first() { return value.first(); }
  Object rest() { return MutList(value.rest()); }
  Object eltAt(int i) { return value.eltAt(i); }
  boolean isEmpty() { return value.isEmpty(); }
  void insert(Object o) { value = value.cons(o); }
  void setFirst(Object o) { value = value.setFirst(o); }

  void setEltAt(int i, final Object val) {

    class UpdateEltAt implements MutSeqVisitor {
      /* fields */
      int index;       // index of element to be updated

      /* constructor */
      UpdateEltAt(int i) { index = i; }

      /* visit methods */
      Object forEmpty(MutSeq host) { throw
        new IllegalArgumentException("out-of-bounds index in UpdateEltAt");
      }
      Object forCons(MutSeq host) {
        if (index == 0) {
  host.setFirst(val);
  return null;
        }
        else host.rest().accept(new UpdateEltAt(i-1));
    }
    value = accept(new UpdateEltAt(i));
  }

  void remove() { value = value.rest; }
  Object accept(SeqVisitor v) { return value.accept(v); }
  Object accept(MutSeqVisitor v) {
    if (value == EMPTY) then return v.forEmpty(this)
    else return v.forCons(this);
  }

  private static abstract class List implements Seq {
    abstract void setFirst(Object o};
    abstract List cons(Object o};
    abstract Object first();
    abstract Seq rest();
```

```
      abstract boolean isEmpty();
    }
    private class Empty extends List {
      public void setFirst(Object o) {
        throw new IllegalArgumentException(setFirst() applied to empty list");
      }
      public List cons(Object o} { new Cons(o,MutList.this); }
      public Object first() {
        throw new IllegalArgumentException("first() applied to empty list");
      }
      public Seq rest() {
        throw new IllegalArgumentException("rest() applied to empty list");
      }
      public int length() { return 0; }
    }
    private static class Cons extends List { ... }
  }
```

Note that we have defined the `List` classes as inner classes to hide them from clients of `MutList`. This feature distinguishes our representation of basic linked lists from the traditional representation used in procedural languages. By embedding the `Node` class hierarchy inside the definition of the `MutList` class, we have completely hidden the fact that we are using a conventional linked list representation! To client code, `MutList` is semantically indistinguishable from `QuasiList`!

What have we gained? First, the `MutList` class is a more efficient implementation of the `MutSeq` interface corresponding to quasi-lists because it allocates only one object for each list node instead of two. Second, we can expand the `MutList` class to include constant-time methods for (*i*) adding an element to the end of a list and (*ii*) appending to lists. These methods rely on an extra field (a "pointer") that refers to the last element of the list. But this expansion comes at a steep price because the mutators `set` and `setRest` either need to be abandoned or implemented in `O(n)` time because the reference to the last node must be updated. In addition, efficient implementations of the functional operations in the `Seq` interface are ugly because these operations cannot mutate `this` but they still must return `ExtMutList` objects or degrade their asymptotic complexity. Returning equivalent `List` or `MutList` objects requires copying the contents of `this`.

Since extending `MutList` to support efficient operations on the rear of the list is incompatible with efficiently supporting the `MutSeq` interface, we defer discussing the implementation of such an extension until we discuss *collection* data structures which do not support the `MutSeq` interface.

Spaghetti References (akin to spaghetti code)

We strongly endorse the slogan that "the love of pointers is the root of all evil." In the 1970s, some pioneering computer scientists (notably Edsger Dijstra, Tony Hoare, and Olaf Dahl) developed structured programming, which imposed a discipline on the control flow of programs. Up until that point, languages (and programmers) had been very liberal in their use of the notorious `goto` statement or *unconditional branch*. The undisciplined use of `goto` statements leads to programs whose control flow is almost impossible to trace by hand (*a.k.a. spaghetti* code), and which are hence almost impossible to debug. In his Turing Award Lecture in the early 1970s, Tony Hoare asserted that multiple references to the same data in a program can introduce a similar form of complexity if that data can be mutated through those references. With multiple references to mutable data in a program, we can stumble into serious trouble when one part of our program modifies an object and another part is unaware of or unprepared for the modification.

Even in programs written and maintained by a single programmer, the unrestricted use of pointers is foolhardy and invites disaster. A programmer almost certainly does not remember the precise reasoning involved in writing every section of a significant application. As a result, mistakes are inevitable. Just as structured programming placed constraints on the control flow in programs, *object-oriente*d programming places constraints on when and where mutable data can be changed. This is why we hide mutable nodes inside a list container.

The Iterator Pattern

Before we present a complete implementation of singly-linked imperative lists (essentially `ExtMutList` less support for most functional operations), we describe a new pattern which allows us to process lists similarly to arrays. The pattern is called the *Iterator* pattern, and consists of two interfaces, an abstract mutable list `ItSeq`, with methods for building and modifying lists

```
interface ItSeq {
  /** Mutates this by inserting o at the front. */
  void insertFront(Object o);
  /** Mutates this by inserting o at the rear. */
  void insertRear(Object o);
  boolean isEmpty();
  /** Removes the first element; unspecified if this is empty. */
  void remFront();
  /** Returns an iterator for traversing this. */
  IteratorI<T> newIterator();
}
```

and an abstract iterator `iteratorI`, with methods for traversing a list and examining its contents

```
interface IteratorI<T> {
  void front();             // move to first element
  void next();
  boolean atEnd();          // test whether past the last element
  T currentItem();     // contents of current element
}
```

The iterator's `atEnd` method returns `true` if and only if the iterator has moved *past* the last element of the list. When the iterator is in this state, the `currentItem` method will throw an exception.

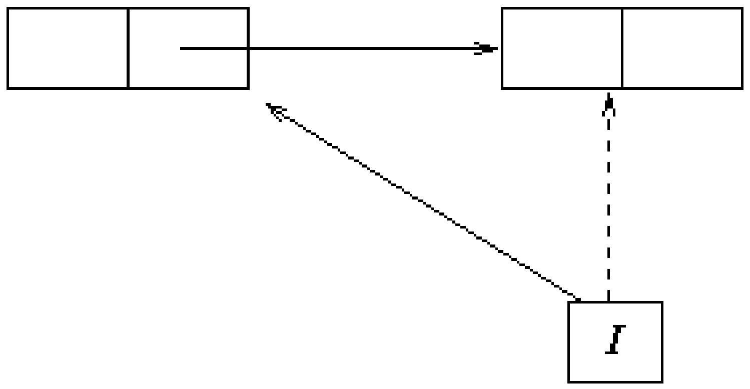
With such a list and such an iterator we could easily implement a queue, since we can remove from the front and add at the back.

It would be nice if the list were more flexible however. For example, we may wish to sort a list. We can already do this in a functional style, by building a new list using an insertion sort, but since this is a mutable list we should ideally be able to sort the list without creating any new nodes, changing it from an unsorted list into a sorted one. We can implement such a sort if we add two more methods to the `IteratorI<T>` interface:

```
  void insertBefore(T o);   // add new element before current
  void remove();            // remove current
```

A given list may have more that one iterator active on it at one time, so the `remove` and `insertBefore` methods must be used with some care.

The definition of an iterator class implementing the `IteratorI` involves two subtle issues: First, for an iterator to remove the current element, it must have a reference to the element immediately *before* the current element.



Second, we can treat the empty list like any other list if we include a dummy node which is always at the head of the list. This dummy node simplifes the implementation of element removal when that element is the last element of the list. When the list is empty, the `last` field refers to dummy node.

2.1.9 An Implementation

All of our implementation sketches of the last few sections now culminate in the following real list implementation. Notice that we make use of both static nested and inner classes in an effort to hide imperative details.

```
// (Singly-Linked) Mutable Lists
import java.util.StringBuilder;

// The ListI interface includes a method newIterator that creates
// a new iterator for a list.  The List Class implementing ListI
// hides the Iterator class implementing ListI.  As a result,
// invoking newIterator is the only way to create new iterators.

interface ListI<T> {

  ListI<T> newList();
  int length();
  void insertFront(T o);
  void insertRear(T o);
  void remFront();
  boolean isEmpty();
  /** @return iterator pointing to first item or end (if list is empty). */
  ReadIteratorI<T> newIterator();
}

/* Iterator for processing lists without mutation. */
interface ReadIteratorI<T> {

  void first();
  void next();
  /** @return true if point to end (just beyond last item. */
  boolean atEnd();
  /** @return true if cursor is pointing to first item in list. */
  boolaan atStart();
  T currentItem();
}

/* Iterator for processing lists with mutation. */
interface IteratorI<T> extends ReadIteratorI<T> {
  /** Inserts o before current item. */
  void insert(T o);
  /** Removes current item, and set cursor to point to next item. */
  void remove();
}

// Exception classes for Lists and Iterators

class ListException extends RuntimeException {
  ListException(String s) { super(s); }
}

class IteratorException extends RuntimeException {
  IteratorException(String s) { super(s); }
}

/** A singly-linked implementation of ListI. */
class List<T> implements ListI<T> {

  // ** fields **
  private Node<T> head = new Node(); // header node
  private Node<T> last = head;  // points to node for last elt or header
  private int size = 0;         // number of elements in list

  /** Relying on default constructor. */

  /** String output for this according in Scheme notation. */
  public String toString(){
    StringBuilder s = new StringBuilder("(");
    for (IteratorI i = newIterator(); ! i.atEnd(); i.next()) {
      if (i.atStart()) s.append(i.currentItem().toString());
      else s.append(" " + i.currentItem().toString());
    }
    return s.append(")").toString();
  }

  public ListI<T> newList() { return new List(); }

  public int size() { return size; }

  public void insertFront(T o) {
    Node<T> oldSucc = head.succ;
    Node<T> newNode = new Node<T>(o,oldSucc);
    head.succ = newNode;
    if (last == head) last = newNode;
    ++size;
  }

  public void insertRear(T o)  {
    Node<T> newNode = new Node<T>(o,null);
    last.succ = newNode;
    last = newNode;
    ++size;
  }

  public void remFront() {
    if (isEmpty()) throw new
      ListException("remFront() applied to EmptyList");
    else {
      Node<T> newSucc = head.succ.succ;
      head.succ = newSucc;
      if (newSucc == null) last = head;
      --size;
    }
  }

  public boolean isEmpty() { return head == last; }

  public IteratorI<T> newIterator() { return new Iterator<T>(); }

  // ** hidden classes Node and Iterator **/

  private static class Node<T> {

    /* fields */
    T item;
    Node<T> succ;

    /* constructors */
    Node(Object i, Node<T> s) {
      item = i;
      succ = s;
    }

    Node() {  // allocate header
      item = null;
      succ = null;
    }

    // fields are accessed directly by code in List class
  }

  private class Iterator<T> implements IteratorI<T> {

    // NOTE: Iterator points to predecessor of current item.
    // Hence, current item is pred.succ

    /* fields */
    Node<T> pred;

    /* Constructors */
    Iterator() { pred = head; }  // refers to enclosing instance

    /* methods in IteratorI interface */
    /** Moves cursor to point to first item (if one exists)
    public void first() { pred = head; }

    public void next() {
     if (atEnd()) throw new
          IteratorException("No next element in Iteration");
      pred = pred.succ;
    }

    public T currentItem() {
      if (atEnd()) throw new
        IteratorException("No current element in " + List.this);
      return pred.succ.item;
    }

    /** @return true if cursor points beyond last elt. */
    public boolean atEnd() { return pred == last; }

    /** Inserts o before current elt.  Handles empty list. */
    public void insert(T o) {
      Node<T> oldSucc = pred.succ;
      Node<T> newNode = new Node(o, oldSucc); // allocate new node
      pred.succ = newNode;                    // insert it
      pred = newNode;                         // update current
      if (oldSucc == null) last = newNode;    // update last if needed
```

```
        ++size;
      }

      /** Removes current element.
        * @throws an IteratorException if atEnd() */
      public void remove() {
        if (atEnd()) // no element available to remove!
          throw new IteratorException(
            "Iterator.remove() applied at end of List");
        Node<T> deadNode = pred.succ;
        pred.succ = deadNode.succ;
        if (last == deadNode) last = pred;
        size--;
      }
    }
  }
```

BiLists and Their Iterators

The lists of the last few sections can be efficiently scanned in only one direction, starting at the front and proceeding element by element to the end. We would now like to develop a more general form of list that supports both forward and backward traversal. The new list implementation will use *doubly-linked* lists, and we will call the new lists `BiLists` and their iterators `BiIterators`.

As with our previous lists, we define a pair of interfaces, `BiListI` for lists, and `BiIteratorI` for iterators. Since `BiLists` and `BiIterators` will support all the same operations as `Lists` and `Iterators`, we will make these interfaces subinterfaces of `BiListI` and `BiIteratorI`.

`BiListI` supports an additional operation for removing nodes at the rear of the list, and provides an additional factory method for producing `BiIteratorI`s.

```
interface BiListI extends ListI {
  void remRear();
  BiIteratorI newIterator();
}
```

`BiIteratorI` supports backward traversal of lists, and so requires methods for moving to the end of a list, moving back an element, and a test for whether the iterator is at the front of the list.

```
interface BiIteratorI extends IteratorI {
  void last();
  void prev();
  boolean atBeginning();
}
```

Since a `BiIteratorI` is also necessarily an `IteratorI`, a `BiIteratorI` instance can be substituted for an `IteratorI` instance. Thus the `newIterator` and `newBiIterator` methods can share the same implementation.

An implementation of `BiList` and `BiIterator` is given below. In contrast to the `List` implementation of the last section, all the classes are top-level, and so imperative operations are not as well hidden. The `BiIterator` must now have a field to record the `BiList` it operates on, and this must be initialized at construction time. As an exercise, try converting the implementation so it uses nested and inner classes as in the `List` case.

The underlying list structure is doubly-linked and *circular*. The dummy node acts as both a marker for the beginning and the end of the list. Since nodes have pointers to both next and previous nodes, the insertion and deletion methods are a little more tricky and require more elaborate pointer juggling. When implementing doubly-linked lists yourself, it helps to draw diagrams that show what points to what at each stage of one of these operations.

In the following code defining the interface `BiListI` class `BiList` the interfaces `ListI`, `ReadIterator`, and `IteratorI` and classes `ListException` and `IteratorException` are identical to those in the code for the (singly-linked) class `MutList`.

```
/** A mutable list supporting bi-directional iterators. */
interface BiListI extends ListI {
  /** Mutates this by deleting the last element; undefined if list is empty. */
  void remRear();
  /** Returns a new bi-directional iterator (enhanced inherited method). */
  BiIteratorI newIterator();
}

/** A bi-directional iterator. */
interface BiIteratorI extends IteratorI {
  void last();
  void prev();
}

/* A doubly-linked, circular implementation of the BiListI interface. */
class BiList implements BiListI {

  // ** fields **
  Node head = new Node();
  int size = 0;

  // ** constructors **
  // relying on default constructor
```

```
    // ** toString
    public String toString() {
      BiIteratorI i = new BiIterator(this);
      String result = "(";

      for (i.first() ; ! i.atEnd(); i.next())
        result = result + " " + i.currentItem();
      return result + " )";
    }

    // ** methods in Interface BiListI

    public ListI newList() { return new BiList(); }

    public BiListI newBiList() { return new BiList(); }

    public int size() { return size; }

    public void insertFront(Object o) {
      Node oldSucc = head.succ;
      Node newNode = new Node(o,head,oldSucc);  // allocate new Node
      // insert new Node
      head.succ = newNode;
      oldSucc.pred = newNode;
      ++size;
    }

    public void insertRear(Object o)  {
      Node oldPred = head.pred;
      Node newNode = new Node(o,oldPred,head);  // allocate new Node
      // insert new Node
      head.pred = newNode;
      oldPred.succ = newNode;
      ++size;
    }

    public void remFront() {
      if (isEmpty())
        throw new ListException("remFront() applied to EmptyList");
      else {
        Node newSucc = head.succ.succ;
        head.succ = newSucc;
        newSucc.pred = head;
        size--;
      }
    }

    public void remRear() {
      if (isEmpty())
        throw new ListException("remRear() applied to EmptyList");
      else {
        Node newPred = head.pred.pred;
        head.pred = newPred;
        newPred.succ = head;
        size--;
      }
    }

    public boolean isEmpty() { return head == head.succ; }

    public IteratorI newIterator() {
      // weaker typing for BiIterator when viewed as Iterator
      return new BiIterator(this);
    }

    public BiIteratorI newBiIterator() {
      return new BiIterator(this);
    }
  }

  // Implementation classes (not hidden!)

  class Node {
    // ** fields **
    Object item;
    Node pred,succ;

    // ** constructors
    Node(Object i, Node p, Node s) {
      item = i;
      pred = p;
      succ = s;
    }

    Node() {  // allocate header
      item = null;
```

```
      pred = this;
      succ = this;
    }
  }

  class BiIterator implements BiIteratorI {

    // ** fields **
    BiList listThis;
    Node current;

    // ** constructors **
    BiIterator(BiList l) {
      listThis = l;                  // associated List instance
      current = listThis.head.succ;  // current is first item (if one exists)
    }

    // ** methods in BiIterator interface **
    public void first() {
      current = listThis.head.succ;  // current is first item (if one exists)
    }

    public void last() {
      current = listThis.head.pred;  // current is last item (if one exists)
    }

    public void next() {
      current = current.succ;        // wraps around end
    }

    public void prev() {
      current = current.pred;        // wraps around end
    }

    public Object currentItem() {
      if (current == listThis.head) throw
        new IteratorException("No current element in " + listThis);
      return current.item;
    }

    public boolean atEnd() { return current == listThis.head; }

    public void insert(Object o) {
      // pre: true
      // post: Node containing o is inserted before current item,
      //       current is unchanged

      Node oldPred = current.pred;
      Node newNode = new Node(o, oldPred, current);  // allocate new node
      current.pred = newNode;                        // insert it
      oldPred.succ = newNode;
      ++(listThis.size);
    }

    public void remove() {

      // pre:  current is valid
      // post: current becomes current.succ

      if (current == listThis.head) throw
        new IteratorException(
          "BiIterator.remove() applied at end of BiList " + listThis);
      Node cPred = current.pred;
      Node cSucc = current.succ;
      cPred.succ = cSucc;
      cSucc.pred = cPred;
      current = cSucc;
      listThis.size--;
    }
  }
```

2.1.10 Collection Libraries

In the preceding code examples, that the interfaces for different formulations of sequences are closely related; they typically contain common operations. This is not surprising since some formulations are extensions of others. It is important in practice because it enables programmers to change data representations, provided the new representation supports the requisite set of operations. Consequently, in designing a library of data classes for assembling aggregations (collections) of elements, it is important to define a hierarchy of interfaces striving for as much compatibilty as possible among interfaces.

In the preceding code examples, we created a rich hierarchy of interfaces where for example, any implementation of BiListI can be substituted for an implementation of ListI since the former interface extends the later (including all method contracts and class invariants). Nevertheless, we could have done even better at the cost of making the type hierarchy more complex. For example, the `Seq` interface could extend a more primitive interface `EnumI` that provides only two operations:

```
boolean isEmpty();
ReadIteratorI newIterator();
```

Any class designed to hold a finite collection of objects, including immutable classes, can be written so that it implements the `EnumI` interface. Moreover, this interface makes it is easy to implement methods return collections. The returned collection can be of type `EnumI`, which accommodates essentially any conceivable implementation. Moreover, it enables many collection returning methods to be implemented simply as adapters superimposed on `this`. In such cases, the host collection (`this)` backs the returned collection. If the host collection is immutable, nothing further needs to be said. But if the host is mutable, the derived collection typically reveals any relevant changes made to the backing collection. In subsequent examples, we will continue to strike a balance between simplicity and generality, producing data structures with few methods and interfaces than we would if were writing a collections library.

2.1.11 Alternate Representations of Lists

So far we have focused on linked lists, but these are not the only possible implementations.

Arrays

Here is a sketch of how we might implement the `BiList` interface using arrays.

1. `newList`: allocate a new array and initialize the elements in order (O(n))
2. `isEmpty`: trivial if along with the array we maintain a count of how many elements are actually in use (O(1))
3. `insertFront`: expensive. If the front is already occupied, we have to shuffle all the contents one place further down the array (O(n);
4. `insertRear`: cheap (O(1));
5. `remRear`: cheap (O(1));
6. `newIterator`: we don't present an implementation, but the details are not hard. The iterator need only keep the index of the current element. But note that inserting or deleting from the middle now requires shuffling elements (O(n) average case).

If we run out of room we can resize the array used to store the list elements. If we double the size of the array at each resizing, then the average number of times an element is copied due to resizing is approximately 1. To prove this, let the initial size of the array be I, and suppose that the final size of the array is N, and there were k resizes. Then

$$N = I \cdot 2^k$$

and we observe that

1. the first I elements move k times;
2. the next I elements move k-1 times;
3. the next $2I$ elements move k-2 times;
4. ...
5. the last $N/2 = 2^{k-1} \cdot I$ elements move $0 = k - k$ times.

Using some summation facts we can show that the total number of array element copy operations is exactly $N - I$. Thus the average number of copy operations per element in the final array is $(N - I)/N$ which is always less than 1, and approaches 1 in the limit as N gets much larger than I (*i.e.* as the number of resizings gets large). We say that the *amortized* cost of copying array elements is (bounded by a) constant. The strategy of doubling the size of the array on each resize operation appears to be an efficient one.

Exercise: Suppose that instead of doubling the array size, we increased it by some constant amount. That is, after k resizings, the size of the array is $I + k \cdot J$ for some constant J. What would the amortized cost of element copying be then?

2.1.12 Hybrid Representations of Sequences

For some application such as a text editor the best representation of sequences may be a hybrid of linked and sequential allocation, sometimes called a *rope* implementation. Such a hybrid links together sequentially allocated blocks of elements (arrays). If the size of blocks is bounded by a constant, then the asymptotic complexity of sequence operations in the hybrid implementation is identical to the corresponding singly or doubly linked list implementation. But the leading constant in the polynomial approximating the running time may be much lower.

2.2 Trees

We have already discussed how to represent algebraic types, inductively defined trees, in an OO language like Java. But there is a fundamentally different form of tree that plays an important role in representing totally ordered collections of data objects, such as ordered sets and maps on an ordered domain. We will focus on Binary Search Trees, which are commonly used to represent both ordered sets and ordered maps. From the perspective of their abstract interfaces, ordered sets and ordered maps are quite different and have only a very coarse common interface at the level of enumerations (specified by the `EnumI` interface discussed in Section 2.1.10.

In a Binary Search Tree, the data items stored in the tree are ordered and every node of the tree contains exactly one item. Moreover, for every node `n` of the tree, all of the items in the left subtree of `n` precede (in the ordering on data items) the item in `n` and all of the items in the right subtree follow (succeed) the item in `n`. In our formulation of Binary Search Trees, no two nodes of the same tree can contain equal items. Generalizations allowing equal items are certainly possible but this form of generalization is rarely useful and adds significant complication.

The fundamental difference between Binary Search Trees representing sets and those representing maps is the nature of the items stored in the tree. In a set, the tree simply contains elements take from some base type `E`, while in a map, the elements are pairs `(a,b)` where `a` belongs to a domain type `A`, `b` belongs to a range type `B`, and the pairs are partially ordered by a total ordering on the domain type `A`. To ensure a total ordering on the pairs in any tree, no two pairs in a tree may contain the same *abscissa* (first element), mirroring the restriction on functions from type `A` to type `B`. In fact, every such tree represents a finite partial function from `A` into `B`. In essence, such functions can be conceptualized as finite tables where each line is a pair txt (a,b) and the lines are sorted by the value of `a`.

We will initially focus on Binary Search Trees representing maps since they are more interesting. Moreover, given an understanding of Binary Search Trees representing maps, it is easy to design Binary Search Trees representing sets. In fact if we restrict the range `B` to a single element, we get a cluttered formulation of Binary Search Trees representing sets.

2.2.1 Procedural Binary Tree Implementations

Binary Search Trees are among the most widely used non-trivial data structures in computer software. Essentially every textbook on data structures and algorithms presents a procedural implementation of Binary Search Trees. As a result, the procedural code for Binary Search Trees has been extensively studied and some references contain extremely tight code for implementing the standard operations. Unfortunately, this code is convoluted and difficult to follow. Moreover, the critical invariant assertions (predicates on the program state) required to establish the correctness of the code (and fully understand it) are almost never included with the code.

The reason why the code is so convoluted is that the standard procedural representation uses a single `Node` type consisting of three pointers: a pointer to a **data** item, a pointer to the **left** subtree, and a pointer to the **right** subtree. But this representation makes no explicit provision for representing empty trees; they are represented by **null** pointers. But null pointers aren't mutable data structures, so any mutation where an empty tree is replaced by a non-empty one typically must be treated as a *special case*. The parent of the empty tree must be updated and if that parent is a Binary Search Tree node, the code must include different cases for updating the **left** and **right** pointers of the node.

In principle, this lack of uniformity can be overcome in procedural languages such as Pascal that explicitly support call-by-reference on variables. But C does not support call-by-reference and C++ only supports it as an afterthought. Furthermore, call-by-reference is treacherous in procedural languages because there are typically no syntactic markets at a call site indicating that a reference (the address of a cell rather than its contents) rather than a value is being passes. Hence, textbook examples of procedural code for Binary Search Trees generally do not employ call-by-reference and plod through ugly case analyses.

OO languages generally do not support call-by-reference because a reference cell is really a simple form of object that is easily and efficiently modeled using objects. The ugly case analyses that appear in procedural code mutating Binary Search Trees epitomize the spaghetti-like procedural flow-of-control that OOP is intended to avoid. On the other hand, it is very easy for OO programmers to get carried away with conceptual elegance at the cost of major performance overhead.

The following OO formulation of Binary Search Trees minimizes this performance overhead. In fact, if the instances of our OO class `RefNode` modeling references (as in call-by-reference) are inlined (absorbed into the client classes in the native code executed by the JVM), it is roughly equivalent to well-written procedural code.

2.2.2 An OO Binary Search Tree Implementation for Maps

Before we present a Binary Search Tree implementation of (finite) maps, we need to introduce the interfaces corresponding to maps. The code is generically typed because the use of generics eliminates many explicit casts which interfere with the clarity of the code.

```
/** A mapping from K to V where K has a natural ordering.  Null keys are
  * not allowed. The pairs (k,v) in a map are called entries. */
interface SortedMapI<K extends Comparable<K,V>> {

  /** @return a new empty map of the same type. */
  SortedMapI<K,V> newMap();

  /** @return true if the map is empty (contains no entries). */
  boolean isEmpty();
```

```
    /** Set this map to the empty map. */
    void clear();

    /** @return the value for key assuming it exists; null otherwise. */
    V get(K key);
    /** Puts entry (key, value) in the map, replacing old entry if extant.
      * @return old value for key or null if no old entry exists. */
    V put(K key, V value);

    /* @return true iff an entry for key exists in the map. */
    public boolean containsKey(K key);
    /** @return the number of entries (keys) in this map. */
    int size();

    /* Deletes the entry for key.
     * @return the value of the removed entry if extant, null otherwise. */
    V remove(K key);

    /** @return the smallest key in this map. */
    K firstKey();
    /** @return the last key in this map. */
    K lastKey();
  }
```

These interfaces are much less comprehensive that what a collections library would provide. For example, a more robust `SortedMap` interface would include a method `newIterator()` that returns an iterator over the entries in the map. You are encouraged to extend the code fragments provide here into a more comprehensive collections library.

The following class implements the preceding interface. Empty nodes are represented by instance of the class `RefNode<K,V>` where the `node` field is `null`. In essence, a `RefNode<K,V>` is a mutable cell containing a reference to an object of type `Node<K,V>` (a node record).

```
  /** A mutable binary search tree of entires of type <K,V>. */
  class TreeMap<K extends Comparable<K>, V> implements SortedMapI<K,V> {

    /** Factory method for creating empty nodes. */
    static <K extends Comparable<K>, V> RefNode<K,V> empty() {
      return new RefNode<K,V>(null);
    }

    /** Number of entries in the map. */
```

```
private int size = 0;

/** Root of binary search tree.  Weak form of state pattern:
  * either empty() or new Ref(new Node(...)). */
RefNode<K,V> root = empty();

/* Relying on default constructor. */

public MapI<K,V> newMap() { return new TreeMap<K,V>(); }
public boolean isEmpty() { return root.isEmpty(); }

public void clear() {
  root = empty();
  size = 0;
}

public V get(K k) { return find(k).value(); }
public V put(K k, V v) {
  RefNode<K,V> location = find(k);  // may be empty
  V oldValue = location.value();    // may return null
  if (location.isEmpty()) {  // key k does not exist in tree
    location.set(k,v);
    ++size;
  }
  else location.setValue(v); // key k exists in tree
  return oldValue;
}

public V remove(K k) {
  RefNode<K,V> location = find(k);     // may return root
  if (location.isEmpty()) return null; // no match found
  size--;
  return location.remove();  // returns old value in location
}

/** Finds the RefNode matching key
  * @returns matching RefNode, which may be empty */
private RefNode<K,V> find(K k) {
  RefNode<K,V> current = root;
  while (! current.isEmpty()) {
    int compareFlag = k.compareTo(current.key());
    if (compareFlag == 0) break;
    else if (compareFlag < 0) current = current.left();
```

```
      else current = current.right();
    }
    return current;
  }

  /** Requires: ! root.isEmpty().
    * @returns the first non-empty RefNode in this. */
  private RefNode<K,V> findFirst() {
    if (isEmpty()) throw
      new MapException("Attempted to find first node of empty TreeMap");
    RefNode<K,V> current = root;
    RefNode<K,V> next = root.left();
    while (true) {
      if (next.isEmpty()) return current;
      current = next; next = next.left();
    }
  }

  /** Requires: ! root.isEmpty()
    * @returns the last non-empty RefNode. */
  private RefNode<K,V> findLast() {
    if (isEmpty()) throw
      new MapException("Attempted to find first node of empty TreeMap");
    RefNode<K,V> current = root;
    RefNode<K,V> next = root.right();
    while (true)
      if (next.isEmpty()) return current;
      current = next; next = next.right();
    }
  }

  public K firstKey() { return findFirst().key(); }
  public K lastKey() { return findLast().key(); }
  public boolean containsKey(K key) { return get(key) != null; }
  public int size() { return size; }

  public String toString() {
    if (isEmpty()) { return "{}"; }

    StringBuilder sb = new StringBuilder();
    toStringHelp(sb, root);
    sb.setCharAt(0, '{');
    sb.append('}');
    return sb.toString();  // sb has a leading space
  }

  private void toStringHelp(StringBuilder sb, RefNode<K,V> n) {
    if (! n.isEmpty()) {
      toStringHelp(sb, n.left());
      sb.append(" [" + n.key() + "," + n.value() + "]");
      toStringHelp(sb, n.right());
    }
  }

  /** An ref class representing updatable cell holding a Node. */
  private static class RefNode<K extends Comparable<K>,V> {
    Node<K,V> node;
    RefNode(Node<K,V> n) { node = n; }

    boolean isEmpty() { return node == null; }
    K key() { return node.key; }
    V value() {
      if (isEmpty()) return null;
      else return node.value;
    }

    /** Requires: node != null.
      * @return left subtree */
    RefNode<K,V> left() { return node.left; }

    /** Requires: node != null.
      * @return left subtree */
    RefNode<K,V> right() { return node.right; }

    /** Sets node to n. */
    void set(Node<K,V> n) { node = n; }

    /** Requires: node != null.
      * Sets the key field of node to k. */
    void setKey(K k) { node.key = k; }

    /** Requires: node != null.
      * Sets the value field of node to v. */
    void setValue(V v) { node.value = v; }

    /** Sets entry in this to (k,v), creating new node if necessary. */
    void set(K k, V v) {
      if (isEmpty()) node = new Node<K,V>(k,v);
      else {
        node.key = k;
        node.value = v;
      }
    }

    /** If this is non-empty, deletes the entry in this. */
    V remove() {
      if (isEmpty()) throw
        new MapException("Attempted to remove a non-existent key");
      V oldVal = node.value;
      if (left().isEmpty()) node = right().node;
      else if (right().isEmpty()) node = left().node;
      else {
        assert !right().isEmpty();
        Node<K,V> minNode = right().removeMin();
        node.key = minNode.key;
        node.value = minNode.value;
      }
      return oldVal;  // size in enclosing TreeMap not updated!
    }

    /** Deletes the Node in this tree with the minimum key.
      * @return the deleted Node. */
    Node<K,V> removeMin() {
      assert node != null;
      RefNode<K,V> current = this;
      V returnVal = node.value;
      do {
        RefNode<K,V> next = current.left();
        if (next.isEmpty()) break; // current refers to minimum node
        current = next;
      } while (true);
      Node<K,V> minNode = current.node;
      current.set(minNode.right.node);  // argument may be null
      return minNode;
    }
  }

  private static class Node<K extends Comparable<K>, V> {
    K key;
    V value;
    RefNode<K,V> left = empty();
    RefNode<K,V> right = empty();

    Node(K k, V v) {
      key = k;
      value = v;
    }
  }
}
```

2.2.3 Performance of Binary Search Trees

2.2.4 Balanced Binary Trees

2.2.5 Other Techniques for Efficiently Searching a Collection

2.2.6 Hashing

2.3 Sorting

2.4 Simple Graph Algorithms

Chapter 3

Graphical User Interfaces

Nearly all contemporary software applications have a graphical user interface. A well-designed graphical interface is far more expressive and easier to use than a text based interface. In this section, we will show how to write simple graphical user interfaces in Java.

3.1 GUI Programming

Graphical user interface programming is inherently more complex than ordinary applications programming because the graphical interface computation is driven by a stream of graphical input actions. All of the input actions performed by a program user including moving the mouse, clicking a mouse button, and typing a keystroke are processed by code in the computer operating system. This code determines when an input action of potential interest to the application occurs. Such an input action is called an "event". Typically mouse movement alone does not constitute an event; the operating system updates the position of the cursor on the screen as the mouse is moved. When a mouse button is clicked or a key is typed, the operating system interrupts the application program and informs it that the specified event has occurred. The Java virtual machine includes an event monitor that processes these interruptions. This event processing code filters input events just as the operating system code filters inputs. For some events such as typing a key (other than return), the Java event monitor simply echoes the character on the screen in the appropriate place. For other events such as a mouse click on a button, the Java event monitor generates a program `Event` object that it places on a queue of pending Events for processing by the running Java program.

Every Java program that creates graphical components has an extra thread of execution that processes the program `Event` objects in the event queue. For each program `Event` object, the thread calls the "listener" method that has been registered by the Java program for handling this kind of program event.

3.1.1 Model-View-controller Pattern

A well-organized graphical application has three components:

- a *model* consisting of the application with no external interface;
- a *view* consisting of one or more graphical frames that interact with the user and the application; and
- a *controller* consisting of the "main" program that constructs the model and the view and links them together.

A model is a "raw" program module with a programming interface consisting a collection of publicly visible methods or procedures. In Java, the application is typically a single object (containing references to many other objects) and the programming interface is the collection of methods supported by that object.

When a program with a graphical interface starts, the controller

1. creates the model (application),
2. creates the view consisting of one or more graphical frames and attaches *commands* to the graphical input controls (buttons, text boxes, etc.) of the view,
3. activates the graphical components in the view, and
4. terminates.

Of course, program execution continues after the controller terminates because the extra thread of execution that processes program events is still running. After the controller terminates, all program execution is triggered by user input actions.

The commands attached to the graphical input controls are operations on the model implemented using the model's programming interface. Recall the *command pattern* from Section 1.9. In Java, each of the graphical input controls in the graphics (AWT/Swing) library has an associated command `interface` that the installed commands implement. In the Java graphics library, these commands are called "listeners" because the are dormant until a graphical input event occurs (*e.g.*, a button is "pressed"). In the programming literature, these commands are often called "callbacks" because they call methods "back" in the model which is logically disjoint from the code running in the view.

To explain how to write programs using the model-view-controller pattern, we will explore a simple example, namely a click-counter application that maintains and displays a simple integer counter ranging from `0` to `999`. The graphical display will show the current value of the counter and include three buttons: an increment button, a decrement button, and reset button.

We will start with the problem of writing the *view* components of the application.

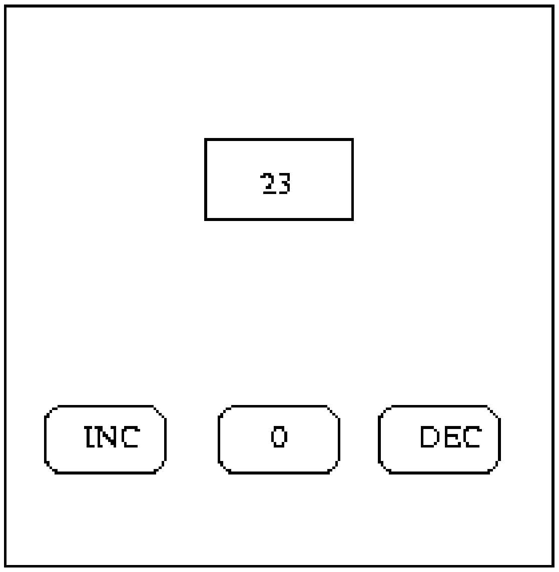
3.1.2 How to Write a View

Most view components have a small number of distinct states that determine how the view is configured and how it will respond to the next program event. As a result, view component programs typically consist of:

- a constructor than initializes the view,
- a registration method for each program event source (`e.g.`, a button) that takes a callback (command) argument and registers this callback as the listener for this event source, and
- a setter method for each distinct view state that sets the fields of the view to the appropriate values.

The controller uses the registration methods to attach callbacks to program event sources in the view. The callbacks use the setter methods to change the state of the view in response to program events.

For our click counter example, the view will have the following format:



which we decompose into three possible states:

1. the *Min* state where the DEC and 0 buttons are deactivated because the counter has its minimum value of 0.
2. the *Counting* state where are three buttons are activated, and

3. the *Max* state where the INC button is deactivated because the counter has reached its maximum value.

The listener must take into account the state of the model to update view.

Warning Most introductory books are written by authors who do not understand the model-view-controller pattern and the proper use of callbacks. Callbacks are not supported in C or C++ because there is no concept of procedures as data objects (simply passing a pointer to a procedure does *not* work!). As a result, textbook authors with a C/C++ background are accustomed to using ugly alternatives to callbacks which they continue to use in the context of Java. A common and particularly onerous abuse of the Java callback interface is implementing the requisite listener interfaces by methods in the main viewer class, which is typically a frame or an applet. This approach limits each event category to a *single* callback, *e.g*, one callback method for all buttons which is coded in the main viewer class.

This approach has four serious disadvantages.

- First, to determine which component produced a given event, the viewer class must uniquely label each event source (or maintain a table of event references).
- Second, when the listener receives an event, it must classify the event source using a sequence of tests or look it up in a table to determine what block of code should be used to process the event.
- Third, the code to process the event embedded in the view relies on the interface provided by the application, corrupting the model-view-controller decomposition.
- Fourth, if another graphical component in the same category is added to the view (*e.g*, a new button) then the code for the callback method for that event *category* must be modified.

Since the `command` pattern (procedures as data objects) completely eliminates this mess, the "view class as listener" approach to event processing is indefensible. Nevertheless, it is widely taught even by some reputed "experts" on Java programming. In fact, I am familiar with only one popular Java book that teaches good programming practice in conjunction with GUI programming, namely *Thinking in Java* by Bruce Eckel.

Coding the View Class The following code defines a view class that supports the schematic display given above:

```
import java.awt.*;
import java.applet.*;
import java.awt.event.*;
import javax.swing.*;

class ClickCounterView {

  // ** fields **
  private JButton incButton;
  private JButton resetButton;
  private JButton decButton;
  private JLabel valueDisplay;

  // ** constructors **
  public ClickCounterView(JApplet itsApplet) {
    JPanel controlPanel = new JPanel();

    itsApplet.getContentPane().setLayout(new BorderLayout());
    valueDisplay = new JLabel("000", JLabel.CENTER);

    itsApplet.getContentPane().add(valueDisplay, "Center");

    incButton = new JButton("+");
    resetButton = new JButton("0");
    decButton = new JButton("-");

    controlPanel.add(incButton);
    controlPanel.add(resetButton);
    controlPanel.add(decButton);
    itsApplet.getContentPane().add(controlPanel, "South");
  }

  // ** methods **
  public void setValueDisplay(String setTo) {
    valueDisplay.setText(setTo);
  }

  public void addIncListener(ActionListener a) {
    incButton.addActionListener(a);
  }

  public void addDecListener(ActionListener a) {
    decButton.addActionListener(a);
  }

  public void addResetListener(ActionListener a) {
    resetButton.addActionListener(a);
  }

  public void setMinimumState() {
    incButton.setEnabled(true);
    resetButton.setEnabled(false);
    decButton.setEnabled(false);
  }

  public void setCountingState() {
    incButton.setEnabled(true);
    resetButton.setEnabled(true);
    decButton.setEnabled(true);
  }

  public void setMaximumState() {
    incButton.setEnabled(false);
    resetButton.setEnabled(true);
    decButton.setEnabled(true);
  }
}
```

The structure of this program is *very simple*. Most of the length is due to Java's wordy syntax and long variable names. What does it do?

The Java AWT/Swing library includes a large number of classes for defining graphical components that can be displayed on the screen. The AWT library relies on the window manager of the underlying operating system to implement common graphical components like windows, menus, and buttons. The Swing extension to the AWT library provides "pure Java" equivalents of these graphical elements, eliminating the vagaries in graphical style among window systems. For every graphical component class *C* in the AWT library, the Swing extension includes an equivalent "pure Java" component class J*C*. For example, the AWT library includes a component class `Button` to represent a button in a graphical window. Hence, the Swing extension includes the corresponding class `JButton`. With a few exceptions, each Swing component class can be used in place of the corresponding AWT class.

All of the component classes in AWT/Swing are *all* descendants of the AWT abstract class `Component` (surprise!). The `ClickCounterView` class mentions three of these component classes, namely `JPanel`, `JButton`, and `JLabel` which are all subclasses of the Swing abstract class `JComponent` (which is a subclass of `Component`). A `JPanel` object is simply a rectangular region that can be incorporated in a graphical container (such as the `JFrame` class in the Swing library) which is subsequently displayed on the screen. A panel typically contains other graphical elements (*e.g.* buttons, drawing canvases, text, pictures) which are displayed as part of the panel. (A blank panel is not very interesting!) A `JButton` is a graphical button and a `JLabel` is a single line of text that can be used as a graphical component.

In AWT/Swing library, graphical components that can contain other graphical components are called containers and belong to type `Container`. Within a container, the layout of the graphical components inside it is determined by a layout manager, a Java object of type `LayoutManager`. One of the important tasks in programming a user interface is determining which layout manager and combination of parameter values to use. A good layout policy will produce an attractive logical layout for a variety of different frame shapes and sizes.

The layout manager `BorderLayout` used in `ClickCounterView` uses compass points to constrain the relative position of graphical components. The four compass points, `"North", "East","South", and "West"` plus the `"Center"` position are supported by the layout manager as directives when graphical components are installed within the panel that it manages.

In the program text above, the `ClickCounterView` object constructor creates panels within the frame provided by the `JApplet` object that is passed to it. An applet is a "top level" container; the browser that is executing the applet provides it with a blank area of the screen in which it can paint its various graphical elements. The constructor for `ClickCounterView` creates a `JLabel displayValue` text line to hold the current click count and a `JPanel controlPanel` containing three buttons, `incButton`, `resetButton`, and `decButton` with adorning `String` labels `+`, `0`, and `-`, respectively. The `displayValue` is placed in the center of the `Applet` and the three buttons are placed in the `controlPanel` using the default layout manager `FlowLayout`. This layout manager places graphical components in rows from left-to-right just like a text editor places characters when you type text into a buffer.

The `ClickCounterView` class defines seven public methods to access and update its components:

- the `setValueDisplay` method update the `valueDisplay` to the specified `String`;
- the methods `addIncListener`, `addDecListener`, and `addResetListener` attach their command arguments to the buttons `incButton`, `resetButton`, and `decButton`, respectively; and
- the methods `setMinimumState`, `setCountingState`, and `setMaximumState` which enable and disable the buttons appropriately for each of the three states described above.

3.1.3 How to Write a Simple Model

From the perspective of GUI design, the critical issue in developing a model is defining the interface for manipulating the model. This interface should be as transparent as possible, without making a commitment to a particular user interface.

In our click counter example program, the model class is utterly trivial. In accordance with the model-view-controller pattern, it is does not presume any particular user interface. The only feature of the counter targeted at supporting a user interface is the `toString` method which pads the output `String` with leading zeroes to produce the specified display width of 3 digits.

```
class ClickCounter {

  // ** fields **
  private static final int MAXIMUM = 999;
  private static final int MINIMUM = 0;
  private static final int STRING_WIDTH = 3;
  private static int count = MINIMUM;

  // ** constructor
  public ClickCounter() {}

  // ** methods
  public boolean isAtMinimum() { return count == MINIMUM; }

  public boolean isAtMaximum() { return count == MAXIMUM; }

  public int inc() {
    if (! this.isAtMaximum()) ++count;
    return count;
  }

  public int dec() {
    if (! this.isAtMinimum()) count--;
    return count;
  }

  public void reset() { count = MINIMUM; }

  public int getCount() { return count; }

  // ** toString() **
  public String toString() {
    StringBuffer buffer =
      new StringBuffer(Integer.toString(count));
    while (buffer.length() < STRING_WIDTH) buffer.insert(0,0);
    return buffer.toString();
  }
}
```

3.1.4 How to Write a Controller

From a programming perspective, the controller is the most interesting part of this example. It glues together the model and view using callbacks and then terminates. Of course, whenever the view receives an event, it invokes callbacks, code defined in the controller, to process them. The controller's callback code performs whatever updates are required to the model an to the view.

```
public class ClickCounterControl extends JApplet {

  // ** fields **
  private ClickCounter counter;
  private ClickCounterView view;

  // ** constructors **
  // relying on default constructor

  // ** methods **
  // relying on inheritance from JApplet

  public void init() {
    counter = new ClickCounter();
    view = new ClickCounterView(this);
    view.setMinimumState();
    view.setValueDisplay(counter.toString());


    view.addIncListener(new ActionListener(){
      public void actionPerformed(ActionEvent event) {
if (counter.isAtMaximum()) return;
if (counter.isAtMinimum()) view.setCountingState();
counter.inc();
view.setValueDisplay(counter.toString());
if (counter.isAtMaximum()) view.setMaximumState();
      }
    });
    view.addDecListener(new ActionListener(){
      public void actionPerformed(ActionEvent event) {
if (counter.isAtMinimum()) return;
if (counter.isAtMaximum()) view.setCountingState();
counter.dec();
view.setValueDisplay(counter.toString());
if (counter.isAtMinimum()) view.setMinimumState();
      }
    });

    view.addResetListener(new ActionListener(){
      public void actionPerformed(ActionEvent event) {
counter.reset();
        view.setMinimumState();
        view.setValueDisplay(counter.toString());
      }
    });
  }
}
```

3.2 What is Concurrent Programming?

Until now, we have been exclusively concerned with sequential programs that execute a single stream of operations. Even the GUI programming in the previous section avoided concurrent execution by terminating the controller as soon as it finished setting up the model and view. Concurrent computation makes programming much more complex. In this section, we will explore the extra problems posed by concurrency and outline some strategies for managing them.

In a concurrent program, several streams of operations may execute concurrently. Each stream of operations executes as it would in a sequential program *except for the fact that streams can communicate and interfere with one another*. Each such sequence of instructions is called a *thread*. For this reason, sequential programs are often called *single-threaded* programs. When a multi-threaded program executes, the operations in its various threads are interleaved in an unpredictable order subject to the constraints imposed by explicit synchronization operations that may be embedded in the code. The operations for each stream are strictly ordered, but the interleaving of operations from a collection of streams is undetermined and depends on the vagaries of a particular execution of the program. One stream may run very fast while another does not run at all. In the absence of fairness guarantees (discussed below), a given thread can starve unless it is the only "runnable" thread.

A thread is *runnable* unless it executes a special operation requiring synchronization that waits until a particular condition occurs. If more than one thread is runnable, all but one thread may starve (make no progress because none of its operations are being executed) unless the thread system makes a *fairness* guarantee. A fairness guarantee states that the next operation in a runnable thread eventually will execute. The Java language specification currently makes no fairness guarantees but most Java Virtual Machines guarantee fairness.

Threads can communicate with each other in a variety of ways that we will discuss in detail later in this section. The Java programming language relies primarily on shared variables to support communication between processes, but it also supports an explicit signaling mechanism.

In general, writing concurrent programs is extremely difficult because the multiplicity of possible interleavings of operations among threads means that program execution is non-deterministic. For this reason, program bugs may be difficult to reproduce. Furthermore, the complexity introduced by multiple threads and their potential interactions makes programs much more difficult to analyze and reason about. Fortunately, many concurrent programs including most GUI applications follow stylized design patterns that control the underlying complexity.

To demonstrate some of the subtle problems that arise with this sort of programming, consider the following example. We have two threads, A and B, that both have access to a variable `ct`. Suppose that, initially, ct is 0, but there are places in both A and B where `ct` is incremented.

```
A        B
...      ...
++ct;    ++ct;
```

To increment a variable `x`, (*i*) the value *v* of `x` must be fetched from memory, (*ii*) a new value *v'* based on *v*, and (*iii*) *v'* must be stored in the memory location allocated to variable `x`. These are three separate actions, and there is no guarantee that no other thread will access the variable until all three are done. So it's possible, for instance, that the order of operations from these two threads occurs as follows:

A fetches `ct` = 0
B fetches `ct` = 0
A computes the value `++ct` = 1
A stores the value 1 in `ct`
B computes new value `++ct` = 1
B stores the value 1 in `ct`

With this order of the operations, the final value for `ct` is 1. But in other possible orderings (*e.g.*, if A performs all of its actions first), the final value would be 2.

A simple strategy for preventing this form of interference (often called a *race condition*) is to make the entire access/modify/store cycle for updating shared variables *atomic*, despite the fact that the cycle is performed using several machine instructions. Atomic operations appear to execute as a single machine instruction because all other threads are forced to pause executing while the atomic operation executes. As a result, it is impossible for another thread to observe the value of the updated variables while the operation is in progress. A block of code that requires atomic execution is called a *critical section.* Some programming languages that support concurrency include **begin/end** brackets for enclosing critical sections.

The critical section mechanism works well in the context of running multi-threaded programs on a computer with a single processor (a *uniprocessor*) since it reduces to ensuring that a critical section is not interruptible (permitting another thread to run). But it is clumsy and inefficient on a multiprocessor because it forces all processors but one to stop execution for the duration of a critical section. (Existing virtual machines treat `new` operations that force garbage collection as *critical sections.*

A much better mechanism for preventing interference in concurrent programs that may be executed on multiprocessors is *locking* data objects. When a data object is locked by a thread, no other thread can access or modify the data object until the locking thread releases it. In essence, locking relaxes the concept of atomic execution so that it is relative to a particular object. Threads can continue executing until they try to access a locked object.

Java relies on object locking to prevent interference. An object can be locked for the duration of a method invocation simply by prefixing the method declaration with the work *synchronized.* For instance, to define a synchronized increment method, we would write:

```
    synchronized void inc() { ++ct; }
```

We can also declare `static` methods to be `synchronized`, which locks the class object (which contain all of the `static` variables of the class) rather than an instance object.

An unusual feature of Java's lock mechanism is the fact that locking an object only inhibits the execution of operations that are declared as `synchronized`. Methods that are not declared as synchronized will be executed even when an object is locked! There is a strong argument for this capability: it supports the definition of classes that partition operations in two groups: those that require synchronization and those that do not. But it also invites subtle synchronization bugs if the `synchronized` modifier is inadvertently omitted from one method definition.

Of course, concurrency only arises in Java when a program uses more than one thread. To support the explicit creation of new threads, Java includes a built-in abstract class `Thread`, that has an abstract method `run()`. We can make a new thread by (*i*) defining a class extending `Thread` that defines the method `run()`, (*ii*) constructing a new instance of this class, and (*iii*) calling the `start()` method on this new instance. The `start()` method actually creates a new thread corresponding to the receiver object (a `Thread`) and invokes the `run()` method of that thread, much as the `main()` method is invoked in the root class when you run a Java Virtual Machine. For example,

```
class Foo extends Thread {
  // must have
  public void run() {
  ...
  }
}
```

When a constructor for `Foo` is called, all of computation for the object allocation and constructor invocation is performed in the current thread; a new thread is not created until the `start()` method is invoked for a Thread() object. To create and start a new `Foo` thread, the current thread can simply execute the code

```
Thread t = new Foo();
t.start();
```

Alternatively, the current thread can execute the `run()` method of the `Thread` object `t` simply by performing the operation

```
t.run()
```

instead of

```
t.start()
```

Assume that a new `Foo` thread `t` has been created and started. At some point in the execution of the original thread (now running concurrently with thread `t`) can wait for thread `t` to terminate by executing the method invocation:

```
t.join();
// waits for the thread object to terminate.
```

So we can view the relationship of the two threads of control as follows:

```
main
  |
t.start
  |\
  | \
  | |
  | /
  |/
t.join
  |
  |
```

Synchronizing multiple threads does incur some overhead. For example, consider the following Java code:

```
class PCount extends Thread {

  // ** fields ***
  static int sharedCtr = 0;
  static final int cntPerIteration = 100000;
  static final int noOfIterations = 10;

  int id;

  // ** constructors **
  PCount(int i) { id = i; }

  // ** methods **
  void inc() {
    ++sharedCtr;
  }
  public void run() {
    for (int i = 0; i < cntPerIteration; ++i) inc();
    System.out.println("Iteration #" + id +
      " has completed; sharedCtr = " + sharedCtr);
  }

  public static void main(String[] args)
      throws InterruptedException {
    Thread[] tPool = new Thread[noOfIterations];
    for (int j = 0; j < noOfIterations; ++j) {
      tPool[j] = new PCount(j);
    }
    for (int j = 0; j < noOfIterations; ++j) {
      tPool[j].start();
    }
    for (int j = 0; j < noOfIterations; ++j) {
      tPool[j].join();
    }
    System.out.println("Computation complete.  sharedCtr = "
      + sharedCtr);
  }
}
```

In each iteration, main creates a new thread. Afterwards, all are synchronized and a final value is determined.

The counter is not locked in this example, and so updates may be lost because of the problems described above. The likelihood with which update losses may occur varies depending on the number of threads. For example, in a test that I ran a few months ago

- for 1 million iterations, the program 65
- for 100,000 iterations, the program lost none.

Apparently, even with 100,000 threads, each iteration occurred within a single time slice.

Synchronizing the threads fixes the problem of lost updates, but it really slows the program down; even for 100,000 iterations.

In modern event-handling models such as those in Java and DrScheme, we have a single event handler that executes events serially. This protocol saves the overhead of synchronization and eliminates potential deadlocks (which we will discuss later).

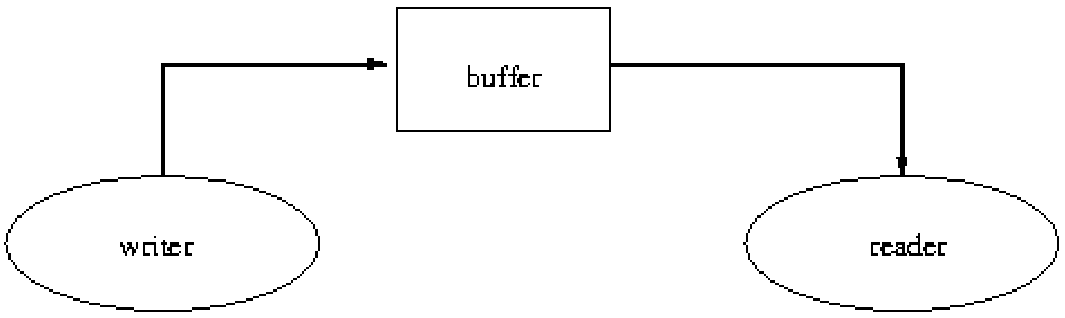
Synchronized methods and statements

We've already discussed synchronized methods above. We can likewise declare synchronized blocks of statements using the following syntax:

```
synchronized(expr) {
   ...
}
```

where ¡tt¿expr¡/tt¿ must evaluate to a reference type (i.e. for most purposes, an object reference). The code between the braces needn't have any connection to ¡tt¿expr¡/tt¿, although this would be perverse in most situations.

For an example, consider implementing a bounded buffer protocol. Bounded buffers are used in Unix to allow interprocess communication via ¡em¿pipes¡/em¿. When the output of one process is piped to the input of another, a bounded buffer is set up between the processes. The first process writes into the buffer, and the second process reads from it.



When the second process attempts to read data but there is none for it to read, it must wait until data is available. There are two general schemes that can be used to implement this waiting:

- *busy-waiting*: the reader process executes a loop in which it tests whether there is any data waiting for it. This approach works but is undesirable because the reader is allocated CPU time just to test if it can do useful computation;

- *blocking*: the operating system provides the routines for reading from and writing to the buffer. When the reader process attempts to read from an empty buffer, it is ¡em¿blocked¡/em¿. The operating system marks the reader as unable to run until data is available in the buffer, and removes it from the collection of currently runnable processes. When the writer process writes data, the system "wakes up" the reader process and makes it one of the runnable processes again. In this way, very little CPU time is expended on coordinating the writer and reader processes.

Naturally, Unix uses the second approach. Note that a symmetric situation occurs when the writer process attempts to write to the buffer and it is full: the writer is blocked pending the availability of space in the buffer. The writer may be reawakened when the reader reads some data and the routine for reading from the buffer determines that there is available space for more data to be written.

We can program a Java thread so that it busy waits on a locked object, but this is almost always a bad programming practice. Fortunately, Java provides a facility that enables us to avoid busy-waiting. There are two primitives,

- `wait()`

- `notify()`

which can be used inside synchronized methods and blocks. `wait()` puts the calling thread to sleep on a queue of threads that are waiting for some change in the status of a locked object. The thread can only be awakened if some other running thread calls `notify()` on the same locked object. When `notify()` is invoked for the locked object, a check is done to see whether any change has been made to it, and then some waiting thread from the associated queue is allowed to run. (Notify arbitrarily picks one thread).

The awakened process will, if prudently written, check whether the condition it is waiting for actually holds. If it does, the thread proceeds, and otherwise it should suspend itself again with another call to `wait()`. Usually, this is implemented in a loop. This may look a bit like busy-waiting, but because `wait()` suspends the calling thread until something of interest happens, most of the time the thread is idle.

There is also a `notifyAll()` method, which works just like `notify()` except that all waiting threads are allowed to run.

3.2.1 Deadlock

In the preceding subsection, we showed how object locking can be used to prevent interference. Unfortunately, locking is not a panacea. The excessive use of locking can severely degrade system performance or, even worse, lock up the system so that all computational progress halts until the program is terminated. The essence of concurrent programming is organizing computations so that neither interference or deadlock can occur.

To illustrate deadlock, let us consider a classical problem called *The Dining Philosophers* in the theory of concurrent programming. It is unrealistic and fanciful, but the synchronization behavior that it models can happen in real systems.

A collection of N philosophers sits at a round table, where $N > 1$. N forks are placed on the table, one between each pair of adjacent philosophers. No philosopher can eat unless he has two forks and he can only use the two forks separating him from his two neighbors. Obviously, adjacent philosophers cannot eat at the same time. Each philosopher alternately eats and sleeps, waiting when necessary for the requisite forks before eating.

Our task is to write code simulating the dining philosophers so that no philosopher starves. An obvious protocol that each philosopher might follow is:

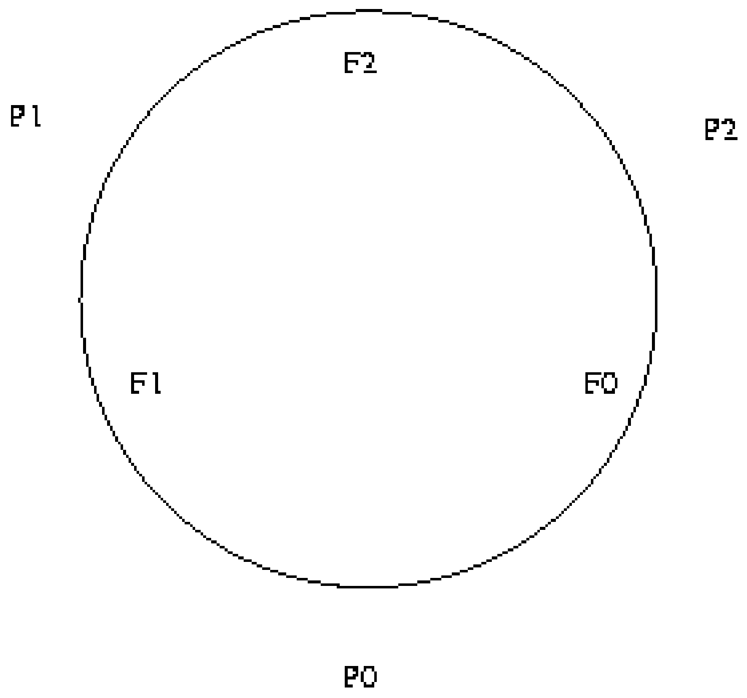
```
while (true) {
  grab left fork;
  grab right fork;
  eat;
  release left fork;
  release right fork
  sleep;
}
```

Now assume that actions of the philosophers are perfectly interleaved: the first philosopher grabs his left fork, then the second philosopher grabs his left fork, and so on until the Nth philosopher grabs his left fork. Then the first philosopher tries to grab his right fork, the second philosopher tries to grab his right fork, and so on. They all have to wait because no right fork is available and they all starve.

Theoretical computer scientists have proven that there is no deterministic uniform solution to this problem. (By uniform, we mean that every philosopher executes exactly the same code with no access to identifying state information such as the name of the “current” philosopher.) But many non-uniform solutions exist. For example, we could number the philosophers around the table from 0 to N. Even numbered philosophers ask for the left fork first, odd numbered ones ask for the right fork first.

Another common solution to this sort of deadlock is to order the resources (in this case forks) and force the processes (philosophers) to grab forks in ascending order. This solution is very general and is widely used in practice.

Consider the case where we have three philosophers: P1, P2, P3 and three forks F1, F2, F3 where P1 sits between F1 and F2, P2 sits between F2 and F3, and P3 sits between F3 and F1. We can order the forks in the obvious ascending order F1, F2, F3.



Now, no matter what, all the philosophers will be able to eat because the linear ordering prevents cycles in the "is waiting for" relation among philosophers. (Finger exercise: prove it!) For instance, if P0 gets F0, and P1 gets F1, P2 must wait until F0 is free. So P1 will get F2 (since there will be no contention), and finish eating. This will release F1 and F2, allowing P0 to get F1 and finish eating. Finally, this will release F0, allowing P2 to get F0 and F2 (since there will be no further contention) and finish eating.