Chapter 1

The Object-Oriented Method

Concepts:
- ▷ Data structures
- ▷ Abstract data types
- ▷ Objects
- ▷ Classes
- ▷ Interfaces

I will pick up the hook.
You will see something new.
Two things. And I call them
Thing One and Thing Two.
These Things will not bite you.
They want to have fun.
—Theodor Seuss Geisel

COMPUTER SCIENCE DOES NOT SUFFER the great history of many other disciplines. While other subjects have well-founded paradigms and methods, computer science still struggles with one important question: *What is the best method to write programs?* To date, we have no best answer. The focus of language designers is to develop programming languages that are simple to use but provide the power to accurately and efficiently describe the details of large programs and applications. The development of Java is one such effort.

Throughout this text we focus on developing data structures using *object-oriented programming*. Using this paradigm the programmer spends time developing templates for structures called *classes*. The templates are then used to construct *instances* or *objects*. A majority of the statements in object-oriented programs involve *sending messages* to objects to have them report or change their state. Running a program involves, then, the construction and coordination of objects. In this way languages like Java are *object-oriented*.

OOP: Object-oriented programming.

In all but the smallest programming projects, *abstraction* is a useful tool for writing working programs. In programming languages including Pascal, Scheme, and C, the details of a program's implementation are hidden away in its procedures or functions. This approach involves *procedural abstraction*. In object-oriented programming the details of the implementation of data structures are hidden away within its objects. This approach involves *data abstraction*. Many modern programming languages use object orientation to support basic abstractions of data. We review the details of data abstraction and the design of formal *interfaces* for objects in this chapter.

1.1 Data Abstraction and Encapsulation

If you purchase a donut from Morningside Bakery in Pittsfield, Massachusetts, you can identify it as a donut without knowing its ingredients. Donuts are circular, breadlike, and sweet. The particular ingredients in a donut are of little concern to you. Of course, Morningside is free to switch from one sweetener to another, as long as the taste is preserved.[1] The donut's ingredients list and its construction are details that probably do not interest you.

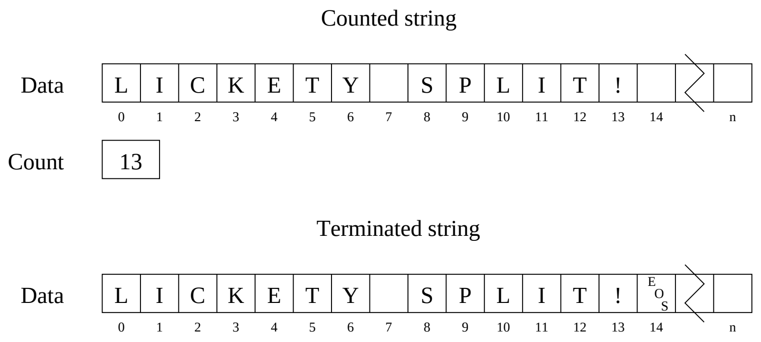
Likewise, it is often unimportant to know how data structures are *implemented* in order to appreciate their *use*. For example, most of us are familiar with the workings or *semantics* of strings or arrays, but, if pressed, we might find it difficult to describe their *mechanics*: *Do all consecutive locations in the array appear close together in memory in your computer, or are they far apart?* The answer is: *it is unimportant*. As long as the array behaves like an array or the string behaves like a string we are happy. The less one knows about how arrays or strings are implemented, the less one becomes dependent on a particular implementation. Another way to think about this abstractly is that the data structure lives up to an implicit "contract": *a string is an ordered list of characters*, or *elements of an array may be accessed in any order*. The implementor of the data structure is free to construct it in any reasonable way, as long as all the terms of the contract are met. Since different implementors are in the habit of making very different implementation decisions, anything that helps to hide the implementation details—any means of using *abstraction*—serves to make the world a better place to program.

Macintosh and UNIX store strings differently.

When used correctly, object-oriented programming allows the programmer to separate the details that are important to the user from the details that are only important to the implementation. Later in this book we shall consider very general behavior of data structures; for example, in Section 10.1 we will study structures that allow the user only to remove the most recently added item. Such behavior is inherent to our most abstract understanding of how the data structure works. We can appreciate the unique behavior of this structure even though we haven't yet discussed how these structures might be implemented. Those abstract details that are important to the user of the structure—including abstract semantics of the methods—make up its *contract* or *interface*. The interface describes the abstract behavior of the structure. Most of us would agree that while strings and arrays are very similar structures, they behave differently: you can shrink or expand a string, while you cannot directly do the same with an array; you can print a string directly, while printing an array involves explicitly printing each of its elements. These distinctions suggest they have distinct abstract behaviors; there are distinctions in the design of their interfaces.

The unimportant details hidden from the user are part of what makes up the *implementation*. We might decide (see Figure 1.1) that a string is to be

[1] Apple cider is often used to flavor donuts in New England, but that decision decidedly *changes* the flavor of the donut for the better. Some of the best apple cider donuts can be found at Atkin's apple farm in Amherst, Massachusetts.



Counted string
Data
L I C K E T Y S P L I T !
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 n
Count
13
Terminated string
Data
L I C K E T Y S P L I T ! EOS
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 n

Figure 1.1 Two methods of implementing a string. A counted string explicitly records its length. The terminated string's length is determined by an end-of-string mark.

constructed from a large array of characters with an attendant character count. Alternatively, we might specify the length implicitly by terminating the string with a special *end-of-string mark* that is not used for any other purpose. Both of these approaches are perfectly satisfactory, but there are trade-offs. The first implementation (called a *counted string*) has its length stored explicitly, while the length of the second implementation (called a *terminated string*) is implied. It takes longer to determine the length of a terminated string because we have to search for the end-of-string mark. On the other hand, the size of a terminated string is limited only by the amount of available memory, while the longest counted string is determined by the range of integers that can be stored in its length field (often this is only several hundred characters). If implementors can hide these details, users do not have to be distracted from their own important design work. As applications mature, a fixed interface to underlying objects allows alternative implementations of the object to be considered.

Data abstraction in languages like Java allows a structure to take responsibility for its own state. The structure knows how to maintain its own state without bothering the programmer. For example, if two strings have to be concatenated into a single string structure, a request might have to be made for a new allotment of memory. Thankfully, because strings know how to perform operations on themselves, the user doesn't have to worry about managing memory.

1.2 The Object Model

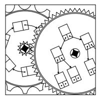
To facilitate the construction of well-designed objects, it is useful to have a design method in mind. As alluded to earlier, we will often visualize the data for our program as being managed by its objects. Each object manages its own data that determine its state. A point on a screen, for example, has two coordinates.

A medical record maintains a name, a list of dependents, a medical history, and a reference to an insurance company. A strand of genetic material has a sequence of base pairs. To maintain a consistent state we imagine the program manipulates the data within its objects only through messages or *method calls* to the objects. A string might receive a message "tell me your length," while a medical record might receive a "change insurance" message. The string message simply accesses information, while the medical record method may involve changing several pieces of information in this and other objects in a consistent manner. If we directly modify the reference to the insurance company, we may forget to modify similar references in each of the dependents. For large applications with complex data structures, it can be extremely difficult to remember to coordinate all the operations that are necessary to move a single complex object from one consistent state to another. We opt, instead, to have the designer of the data structure provide us a method for carefully moving between states; this method is activated in response to a high-level message sent to the object.

This text, then, focuses on two important topics: (1) how we implement and evaluate objects with methods that are logically complex and (2) how we might use the objects we create. These objects typically represent *data structures*, our primary interest. Occasionally we will develop *control structures*—structures whose purpose is to control the manipulation of other objects. Control structures are an important concept and are described in detail in Chapter 8.

1.3 Object-Oriented Terminology

In Java, data abstraction is accomplished through *encapsulation* of data in an *object*—an instance of a *class*. Like a *record* in other languages, an object has *fields*. Unlike records, objects also contain *methods*. Fields and methods of an object may be declared `public`, which means that they are visible to entities outside the class, or `protected`, in which case they may only be accessed by code within methods of the class.[2] A typical class declaration is demonstrated by the following simple class that keeps track of the ratio of two integer values:



`Ratio`

```
public class Ratio
{
    protected int numerator;   // numerator of ratio
    protected int denominator; // denominator of ratio

    public Ratio(int top, int bottom)
    // pre: bottom != 0
    // post: constructs a ratio equivalent to top::bottom
    {
        numerator = top;
        denominator = bottom;
        reduce();
```

[2] This is not quite the truth. For a discussion of the facts, see Appendix B.8.

```
}

public int getNumerator()
// post: return the numerator of the fraction
{
    return numerator;
}

public int getDenominator()
// post: return the denominator of the fraction
{
    return denominator;
}

public double getValue()
// post: return the double equivalent of the ratio
{
    return (double)numerator/(double)denominator;
}

public Ratio add(Ratio other)
// pre: other is nonnull
// post: return new fraction--the sum of this and other
{
    return new Ratio(this.numerator*other.denominator+
                     this.denominator*other.numerator,
                     this.denominator*other.denominator);
}

protected void reduce()
// post: numerator and denominator are set so that
// the greatest common divisor of the numerator and denominator is 1
{
    int divisor = gcd(numerator,denominator);
    if (denominator < 0) divisor = -divisor;
    numerator /= divisor;
    denominator /= divisor;
}

protected static int gcd(int a, int b)
// post: computes the greatest integer value that divides a and b
{
    if (a < 0) return gcd(-a,b);
    if (a == 0) {
        if (b == 0) return 1;
        else return b;
    }
    if (b < a) return gcd(b,a);
    return gcd(b%a,a);
}
```

```
        public String toString()
        // post: returns a string that represents this fraction.
        {
            return getNumerator()+"/"+getDenominator();
        }
    }
```

First, a `Ratio` object maintains the numerator and denominator as protected `int`s that are not directly modifiable by the user. The `Ratio` method is a *constructor*: a method whose name is the same as that of the class. (The formal comments at the top of methods are pre- and postconditions; we discuss these in detail in Chapter 2.) The constructor is called whenever a new `Ratio` object is formed. Constructors initialize all the fields of the associated object, placing the object into a predictable and consistent initial state. We declare the constructors for a class `public`. To construct a new `Ratio` object, users will have to call these methods. The `value` method returns a `double` that represents the ratio, while the `getNumerator` and `getDenominator` methods fetch the current values of the numerator and denominator of the fraction. The `add` method is useful for adding one `Ratio` to another; the result is a newly constructed `Ratio` object. Finally, the `toString` method generates the preferred printable representation of the object; we have chosen to represent it in fractional form.

Two methods, `reduce` and `gcd`, are utility methods. The `gcd` method computes the greatest common divisor of two values using *Euclid's method*, one of the oldest numerical algorithms used today. It is used by the `reduce` method to reduce the numerator and denominator to lowest terms by removing any common factors. Both are declared `protected` because computing the reduction is not a necessary (or obvious) operation to be performed on ratios of integers; it's part of the implementation. The `gcd` method is declared `static` because the algorithm can be used at any time—its utility is independent of the number of `Ratio` objects that exist in our program. The `reduce` method, on the other hand, works only with a `Ratio` object.

Exercise 1.1 *Nearly everything can be improved. Are there improvements that might be made to the* `gcd` *method? Can you write the method iteratively? Is iteration an improvement?*

As with the `Ratio` class, data fields are usually declared `protected`. To manipulate protected fields the user must invoke `public` methods. The following example demonstrates the manipulation of the `Ratio` class:

```
    public static void main(String[] args)
    {
        Ratio r = new Ratio(1,1);      // r == 1.0
        r = new Ratio(1,2);            // r == 0.5
        r.add(new Ratio(1,3));         // sum computed, but r still 0.5
        r = r.add(new Ratio(2,8));     // r == 0.75
        System.out.println(r.getValue()); // 0.75 printed
        System.out.println(r.toString());  // calls toString()
        System.out.println(r);  // calls toString()
    }
```

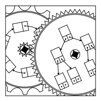
To understand the merit of this technique of class design, we might draw an analogy between a well-designed object and a lightbulb for your back porch. The protected fields and methods of an object are analogous to the internal design of the bulb. The observable features, including the voltage and the size of the socket, are provided without giving any details about the implementation of the object. If light socket manufacturers depended on a particular implementation of lightbulbs—for example the socket only supported bright xenon bulbs—it might ultimately restrict the variety of suppliers of lightbulbs in the future. Likewise, manufacturers of lightbulbs should be able to have a certain freedom in their implementation: as long as they only draw current in an agreed-upon way and as long as their bulb fits the socket, they should be free to use whatever design they want. Today, most lamps take either incandescent or fluorescent bulbs.

In the same way that fields are encapsulated by a class, classes may be encapsulated by a *package*. A package is a collection of related classes that implement some set of structures with a common theme. The classes of this text, for example, are members of the `structure` package. In the same way that there are users of classes, there are users of packages, and much of the analogy holds. In particular, classes may be declared `public`, in which case they may be used by anyone who *imports* the package into their program. If a class is not `public`, it is automatically considered `protected`. These `protected` classes may only be constructed and used by other classes within the same package.

1.4 A Special-Purpose Class: A Bank Account

We now look at the detailed construction of a simplistic class: a `BankAccount`. Many times, it is necessary to provide a tag associated with an instance of a data structure. You might imagine that your bank balance is kept in a database at your bank. When you get money for a trip through the Berkshires, you swipe your card through an automated teller bringing up your account. Your account number, presumably, is unique to your account. Nothing about you or your banking history is actually stored in your account number. Instead, that number is used to find the record linked to your account: the bank searches for a structure associated with the number you provide. Thus a `BankAccount` is a simple, but important, data structure. It has an `account` (an identifier that never changes) and a `balance` (that potentially *does* change). The public methods of such a structure are as follows:

Automated teller: a robotic palm reader.


`BankAccount`

```
public class BankAccount
{
    public BankAccount(String acc, double bal)
    // pre: account is a string identifying the bank account
```

```
        // balance is the starting balance
        // post: constructs a bank account with desired balance

        public boolean equals(Object other)
        // pre: other is a valid bank account
        // post: returns true if this bank account is the same as other

        public String getAccount()
        // post: returns the bank account number of this account

        public double getBalance()
        // post: returns the balance of this bank account

        public void deposit(double amount)
        // post: deposit money in the bank account

        public void withdraw(double amount)
        // pre: there are sufficient funds in the account
        // post: withdraw money from the bank account
    }
```

The substance of these methods has purposefully been removed because, again, it is unimportant for us to know exactly how a `BankAccount` is implemented. We have ways to construct and compare `BankAccounts`, as well as ways to read the account number or balance, or update the balance.

Let's look at the implementation of these methods, individually. To build a new bank account, you must use the `new` operator to call the constructor with two parameters. The account number provided never changes over the life of the `BankAccount`—if it were necessary to change the value of the account number, a new `BankAccount` would have to be made, and the balance would have to be transferred from one to the other. The constructor plays the important role of performing the one-time initialization of the account number field. Here is the code for a `BankAccount` constructor:

```
protected String account;  // the account number
protected double balance; // the balance associated with account

public BankAccount(String acc, double bal)
// pre: account is a string identifying the bank account
// balance is the starting balance
// post: constructs a bank account with desired balance
{
    account = acc;
    balance = bal;
}
```

Two fields—`account` and `balance`—of the `BankAccount` object are responsible for maintaining the object's state. The `account` keeps track of the account number, while the `balance` field maintains the balance.

Since account numbers are unique to BankAccounts, to check to see if two accounts are "the same," we need only compare the account fields. Here's the code:

```
public boolean equals(Object other)
// pre: other is a valid bank account
// post: returns true if this bank account is the same as other
{
    BankAccount that = (BankAccount)other;
    // two accounts are the same if account numbers are the same
    return this.account.equals(that.account);
}
```

Notice that the BankAccount equals method calls the equals method of the key, a String. Both BankAccount and String are nonprimitive types, or examples of Objects. Every object in Java has an equals method. If you don't explicitly provide one, the system will write one for you. Generally speaking, one should assume that the automatically written or *default* equals method is of little use. This notion of "equality" of objects is often based on the complexities of our abstraction; its design must be considered carefully.

One can ask the BankAccount about various aspects of its state by calling its getAccount or getBalance methods:

```
public String getAccount()
// post: returns the bank account number of this account
{
    return account;
}

public double getBalance()
// post: returns the balance of this bank account
{
    return balance;
}
```

These methods do little more than pass along the information found in the account and balance fields, respectively. We call such methods *accessors*. In a different implementation of the BankAccount, the balance would not have to be explicitly stored—the value might be, for example, the difference between two fields, deposits and drafts. Given the interface, it is not much of a concern to the user which implementation is used.

We provide two more methods, deposit and withdraw, that explicitly modify the current balance. These are *mutator* methods:

```
public void deposit(double amount)
// post: deposit money in the bank account
{
    balance = balance + amount;
}
```

```
public void withdraw(double amount)
// pre: there are sufficient funds in the account
// post: withdraw money from the bank account
{
    balance = balance - amount;
}
```

Because we would like to change the balance of the account, it is important to have a method that allows us to modify it. On the other hand, we purposefully don't have a `setAccount` method because we do not want the account number to be changed without a considerable amount of work (work that, by the way, models reality).

Here is a simple application that determines whether it is better to deposit \$100 in an account that bears 5 percent interest for 10 years, or to deposit \$100 in an account that bears $2\frac{1}{2}$ percent interest for 20 years. It makes use of the `BankAccount` object just outlined:

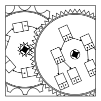
```
public static void main(String[] args)
{
    // Question: is it better to invest $100 over 10 years at 5%
    //           or to invest $100 over 20 years at 2.5% interest?
    BankAccount jd = new BankAccount("Jain Dough",100.00);
    BankAccount js = new BankAccount("Jon Smythe",100.00);

    for (int years = 0; years < 10; years++)
    {
        jd.deposit(jd.getBalance() * 0.05);
    }
    for (int years = 0; years < 20; years++)
    {
        js.deposit(js.getBalance() * 0.025);
    }
    System.out.println("Jain invests $100 over 10 years at 5%.");
    System.out.println("After 10 years " + jd.getAccount() +
                       " has $" + jd.getBalance());
    System.out.println("Jon invests $100 over 20 years at 2.5%.");
    System.out.println("After 20 years " + js.getAccount() +
                       " has $" + js.getBalance());
}
```

Exercise 1.2 *Which method of investment would you pick?*

1.5 A General-Purpose Class: An Association

At least Dr. Seuss started with 50 words!

The following small application implements a Pig Latin translator based on a dictionary of nine words. The code makes use of an array of `Associations`, each of which establishes a relation between an English word and its Pig Latin translation. For each string passed as the argument to the main method, the dictionary is searched to determine the appropriate translation.


atinlay

```
public class atinLay {
    // a pig latin translator for nine words
    public static void main(String args[])
    {
        // build and fill out an array of nine translations
        Association dict[] = new Association[9];
        dict[0] = new Association("a","aay");
        dict[1] = new Association("bad","adbay");
        dict[2] = new Association("had","adhay");
        dict[3] = new Association("dad","adday");
        dict[4] = new Association("day","ayday");
        dict[5] = new Association("hop","ophay");
        dict[6] = new Association("on","onay");
        dict[7] = new Association("pop","oppay");
        dict[8] = new Association("sad","adsay");

        for (int argn = 0; argn < args.length; argn++)
        {   // for each argument
            for (int dictn = 0; dictn < dict.length; dictn++)
            {   // check each dictionary entry
                if (dict[dictn].getKey().equals(args[argn]))
                    System.out.println(dict[dictn].getValue());
            }
        }
    }
}
```

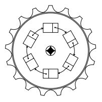
When this application is run with the arguments hop on pop, the results are

```
ophay
onay
oppay
```

While this application may seem rather trivial, it is easy to imagine a large-scale application with similar needs.[3]

We now consider the design of the Association. Notice that while the *type* of data maintained is different, the *purpose* of the Association is very similar to that of the BankAccount class we discussed in Section 1.4. An Association is a key-value pair such that the key cannot be modified. Here is the interface for the Association class:

```
import java.util.Map;
```


Association

[3] Pig Latin has played an important role in undermining court-ordered restrictions placed on music piracy. When Napster—the rebel music trading firm—put in checks to recognize copyrighted music by title, traders used Pig Latin translators to foil the recognition software!

```
public class Association implements Map.Entry
{
    public Association(Object key, Object value)
    // pre: key is non-null
    // post: constructs a key-value pair

    public Association(Object key)
    // pre: key is non-null
    // post: constructs a key-value pair; value is null

    public boolean equals(Object other)
    // pre: other is non-null Association
    // post: returns true iff the keys are equal

    public Object getValue()
    // post: returns value from association

    public Object getKey()
    // post: returns key from association

    public Object setValue(Object value)
    // post: sets association's value to value
}
```

For the moment, we will ignore the references to `Map` and `Map.entry`; these will be explained later, in Chapter 15. What distinguishes an `Association` from a more specialized class, like `BankAccount`, is that the fields of an `Association` are type `Object`. The use of the word `Object` in the definition of an `Association` makes the definition very general: any value that is of type `Object`—any non-primitive data type in Java—can be used for the `key` and `value` fields.

Unlike the `BankAccount` class, this class has two different constructors:

```
protected Object theKey; // the key of the key-value pair
protected Object theValue; // the value of the key-value pair

public Association(Object key, Object value)
// pre: key is non-null
// post: constructs a key-value pair
{
    Assert.pre(key != null, "Key must not be null.");
    theKey = key;
    theValue = value;
}

public Association(Object key)
// pre: key is non-null
// post: constructs a key-value pair; value is null
{
    this(key,null);
}
```

The first constructor—the constructor distinguished by having two parameters—allows the user to construct a new `Association` by initializing both fields. On occasion, however, we may wish to have an `Association` whose `key` field is set, but whose `value` field is left referencing nothing. (An example might be a medical record: initially the medical history is incomplete, perhaps waiting to be forwarded from a previous physician.) For this purpose, we provide a single parameter constructor that sets the `value` field to `null`. Note that we use `this(key,null)` as the body. The one-parameter constructor calls this object's two-parameter constructor with `null` as the second parameter. We write the constructors in this dependent manner so that if the underlying implementation of the `Association` had to be changed, only the two-parameter method would have to be updated. It also reduces the complexity of the code and saves your fingerprints!

Now, given a particular `Association`, it is useful to be able to retrieve the key or value. Since the implementation is hidden, no one outside the class is able to see it. Users must depend on the accessor methods to observe the data.

```
public Object getValue()
// post: returns value from association
{
    return theValue;
}

public Object getKey()
// post: returns key from association
{
    return theKey;
}
```

When necessary, the method `setValue` can be used to change the value associated with the key. Thus, the `setValue` method simply takes its parameter and assigns it to the `value` field:

```
public Object setValue(Object value)
// post: sets association's value to value
{
    Object oldValue = theValue;
    theValue = value;
    return oldValue;
}
```

There are other methods that are made available to users of the `Association` class, but we will not discuss the details of that code until later. Some of the methods are required, some are useful, and some are just nice to have around. While the code may look complicated, we take the time to implement it correctly, so that *we will not have to reimplement it in the future*.

Principle 2 *Free the future: reuse code.*

It is difficult to fight the temptation to design data structures from scratch. We shall see, however, that many of the more complex structures would be very difficult to construct if we could not base our implementations on the results of previous work.

1.6 Sketching an Example: A Word List

Suppose we're interested in building a game of Hangman. The computer selects random words and we try to guess them. Over several games, the computer should pick a variety of words and, as each word is used, it should be removed from the word list. Using an object-oriented approach, we'll determine the essential features of a `WordList`, the Java object that maintains our list of words.

Our approach to designing the data structures has the following five informal steps:

1. Identify the types of operations you expect to perform on your object. What operations *access* your object only by reading its data? What operations might modify or *mutate* your objects?

2. Identify, given your operations, those data that support the *state* of your object. Information about an object's state is carried within the object between operations that modify the state. Since there may be many ways to encode the state of your object, your description of the state may be very general.

3. Identify any rules of consistency. In the `Ratio` class, for example, it would not be good to have a zero denominator. Also, the numerator and denominator should be in lowest terms.

4. Determine the number and form of the constructors. Constructors are synthetic: their sole responsibility is to get a new object into a good initial and consistent state. Don't forget to consider the best state for an object constructed using the parameterless *default constructor*.

5. Identify the types and kinds of information that, though declared `protected`, can *efficiently* provide the information needed by the `public` methods. Important choices about the internals of a data structure are usually made at this time. Sometimes, competing approaches are developed until a comparative evaluation can be made. That is the subject of much of this book.

The operations necessary to support a list of words can be sketched out easily, even if we don't know the intimate details of constructing the Hangman game itself. Once we see how the data structure is used, we have a handle on the design of the interface. Thinking about the overall design of Hangman, we can identify the following general use of the `WordList` object:

WordList

```
WordList list;                          // declaration
String targetWord;

list = new WordList(10);                // construction
list.add("disambiguate");  // is this a word? how about ambiguate?
list.add("inputted");      // really? what verbification!
list.add("subbookkeeper"); // now that's coollooking!
while (!list.isEmpty())                 // game loop
{
    targetWord = list.selectAny();      // selection
    // ...play the game using target word...
    list.remove(targetWord);            // update
}
```

Let's consider these lines. One of the first lines (labeled `declaration`) declares a *reference* to a `WordList`. For a reference to refer to an object, the object must be constructed. We require, therefore, a constructor for a `WordList`. The `construction` line allocates an initially empty list of words ultimately containing as many as 10 words. We provide an upper limit on the number of words that are potentially stored in the list. (We'll see later that providing such information can be useful in designing efficient data structures.) On the next three lines, three (dubious) words are added to the list.

The `while` loop accomplishes the task of playing Hangman with the user. This is possible as long as the list of words is not empty. We use the `isEmpty` method to test this fact. At the beginning of each round of Hangman, a random word is selected (`selectAny`), setting the `targetWord` reference. To make things interesting, we presume that the `selectAny` method selects a random word each time. Once the round is finished, we use the `remove` method to remove the word from the word list, eliminating it as a choice in future rounds.

There are insights here. First, we have said very little about the Hangman game other than its interaction with our rather abstract list of words. The details of the screen's appearance, for example, do not play much of a role in understanding how the `WordList` structure works. We knew that a list was necessary for our program, and we considered the program *from the point of view of the object*. Second, we don't really know how the `WordList` is implemented. The words may be stored in an array, or in a file on disk, or they may use some technology that we don't currently understand. It is only important that we have *faith* that the structure can be implemented. We have sketched out the method headers, or *signatures*, of the `WordList` *interface*, and we have faith that an *implementation* supporting the interface can be built. Finally we note that what we have written is not a complete program. Still, from the viewpoint of the `WordList` structure, there are few details of the interface that are in question. A reasoned individual should be able to look at this design and say "this will work—provided it is implemented correctly." If a reviewer of the code were to ask a question about how the structure works, it would lead to a refinement of our understanding of the interface.

We have, then, the following required interface for the `WordList` class:

```
public class WordList
{
    public WordList(int size)
    // pre: size >= 0
    // post: construct a word list capable of holding "size" words

    public boolean isEmpty()
    // post: return true iff the word list contains no words

    public void add(String s)
    // post: add a word to the word list, if it is not already there

    public String selectAny()
    // pre: the word list is not empty
    // post: return a random word from the list

    public void remove(String word)
    // pre: word is not null
    // post: remove the word from the word list
}
```

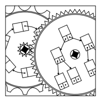
We will leave the implementation details of this example until later. You might consider various ways that the `WordList` might be implemented. As long as the methods of the interface can be supported by your data structure, your implementation is valid.

Exercise 1.3 *Finish the sketch of the* `WordList` *class to include details about the state variables.*

1.7 Sketching an Example: A Rectangle Class

Suppose we are developing a graphics system that allows the programmer to draw on a `DrawingWindow`. This window has, associated with it, a Cartesian coordinate system that allows us to uniquely address each of the points within the window. Suppose, also, that we have methods for drawing line segments, say, using the `Line` object. How might we implement a rectangle—called a `Rect`—to be drawn in the drawing window?

One obvious goal would be to draw a `Rect` on the `DrawingWindow`. This might be accomplished by drawing four line segments. It would be useful to be able to draw a filled rectangle, or to erase a rectangle (think: draw a filled rectangle in the background color). We're not sure how to do this efficiently, but these latter methods seem plausible and consistent with the notion of drawing. (We should check to see if it is possible to draw in the background color.) This leads to the following methods:



Rect

```
public void fillOn(DrawingTarget d)
// pre: d is a valid drawing window
// post: the rectangle is filled on the drawing window d
```

```
public void clearOn(DrawingTarget d)
// pre: d is a valid drawing window
// post: the rectangle is erased from the drawing window

public void drawOn(DrawingTarget d)
// pre: d is a valid drawing window
// post: the rectangle is drawn on the drawing window
```

It might be useful to provide some methods to allow us to perform basic calculations—for example, we might want to find out if the mouse arrow is located within the `Rect`. These require accessors for all the obvious data. In the hope that we might use a `Rect` multiple times in multiple locations, we also provide methods for moving and reshaping the `Rect`.

```
public boolean contains(Pt p)
// pre: p is a valid point
// post: true iff p is within the rectangle

public int left()
// post: returns left coordinate of the rectangle

public void left(int x)
// post: sets left to x; dimensions remain unchanged

public int width()
// post: returns the width of the rectangle

public void width(int w)
// post: sets width of rectangle, center and height unchanged

public void center(Pt p)
// post: sets center of rect to p; dimensions remain unchanged

public void move(int dx, int dy)
// post: moves rectangle to left by dx and down by dy

public void moveTo(int left, int top)
// post: moves left top of rectangle to (left,top);
//       dimensions are unchanged

public void extend(int dx, int dy)
// post: moves sides of rectangle outward by dx and dy
```

Again, other approaches might be equally valid. No matter how we might represent a `Rect`, however, it seems that all rectangular regions with horizontal and vertical sides can be specified with four integers. We can, then, construct a `Rect` by specifying, say, the left and top coordinates and the width and height.

For consistency's sake, it seems appropriate to allow rectangles to be drawn anywhere (even off the screen), but the width and height should be non-negative values. We should make sure that these constraints appear in the documentation associated with the appropriate constructors and methods. (See Section 2.2 for more details on how to write these comments.)

Given our thinking, we have some obvious `Rect` constructors:

```
public Rect()
// post: constructs a trivial rectangle at origin

public Rect(Pt p1, Pt p2)
// post: constructs a rectangle between p1 and p2

public Rect(int x, int y, int w, int h)
// pre: w >= 0, h >= 0
// post: constructs a rectangle with upper left (x,y),
//       width w, height h
```

We should feel pleased with the progress we have made. We have developed the signatures for the rectangle interface, even though we have no immediate application. We also have some emerging answers on approaches to implementing the `Rect` internally. If we declare our `Rect` data protected, we can insulate ourselves from changes suggested by inefficiencies we may yet discover.

Exercise 1.4 *Given this sketch of the* `Rect` *interface, how would you declare the private data associated with the* `Rect` *object? Given your approach, describe how you might implement the* `center(int x, int y)` *method.*

1.8 Interfaces

Sometimes it is useful to describe the interface for a number of different classes, without committing to an implementation. For example, in later sections of this text we will implement a number of data structures that are able to be modified by adding or removing values. We can, for all of these classes, specify a few of their fundamental methods by using the Java `interface` declaration:

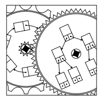
`Structure`

```
public interface Structure
{
    public int size();
    // post: computes number of elements contained in structure

    public boolean isEmpty();
    // post: return true iff the structure is empty

    public void clear();
    // post: the structure is empty

    public boolean contains(Object value);
    // pre: value is non-null
    // post: returns true iff value.equals some value in structure
```

```
        public void add(Object value);
        // pre: value is non-null
        // post: value has been added to the structure
        //       replacement policy is not specified

        public Object remove(Object value);
        // pre: value is non-null
        // post: an object equal to value is removed and returned, if found

        public java.util.Enumeration elements();
        // post: returns an enumeration for traversing structure;
        //       all structure package implementations return
        //       an AbstractIterator

        public Iterator iterator();
        // post: returns an iterator for traversing structure;
        //       all structure package implementations return
        //       an AbstractIterator

        public Collection values();
        // post: returns a Collection that may be used with
        //       Java's Collection Framework
    }
```

Notice that the body of each method has been replaced by a semicolon. It is, in fact, illegal to specify any code in a Java interface. Specifying just the method signatures in an interface is like writing boilerplate for a contract without committing to any implementation. When we decide that we are interested in constructing a new class, we can choose to have it *implement* the Structure interface. For example, our WordList structure of Section 1.6 might have made use of our Structure interface by beginning its declaration as follows:

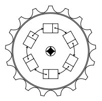
```
    public class WordList implements Structure
```



WordList

When the WordList class is compiled by the Java compiler, it checks to see that each of the methods mentioned in the Structure interface—add, remove, size, and the others—is actually implemented. In this case, only isEmpty is part of the WordList specification, so we must either (1) not have WordList implement the Structure interface or (2) add the methods demanded by Structure.

Interfaces may be extended. Here, we have a possible definition of what it means to be a Set:



Set

```
    public interface Set extends Structure
    {
        public void addAll(Structure other);
        // pre: other is non-null
        // post: values from other are added into this set
        public boolean containsAll(Structure other);
        // pre: other is non-null
        // post: returns true if every value in set is in other

        public void removeAll(Structure other);
        // pre: other is non-null
        // post: values of this set contained in other are removed

        public void retainAll(Structure other);
        // pre: other is non-null
        // post: values not appearing in the other structure are removed
    }
```

A `Set` requires several set-manipulation methods—`addAll` (i.e., set union) `retainAll` (set intersection), and `removeAll` (set difference)—as well as the methods demanded by being a `Structure`. If we implement these methods for the `WordList` class and indicate that `WordList implements Set`, the `WordList` class could be used wherever either a `Structure` or `Set` is required. Currently, our `WordList` is close to, but not quite, a `Structure`. Applications that demand the functionality of a `Structure` will not be satisfied with a `WordList`. Having the class implement an interface increases the flexibility of its use. Still, it may require considerable work for us to upgrade the `WordList` class to the level of a `Structure`. It may even work against the design of the `WordList` to provide the missing methods. The choices we make are part of an ongoing design process that attempts to provide the best implementations of structures to meet the demands of the user.

1.9 Who Is the User?

When implementing data structures using classes and interfaces, it is sometimes hard to understand *why* we might be interested in hiding the implementation. After all, perhaps we know that ultimately we will be the only programmers making use of these structures. That might be a good point, except that if you are really a successful programmer, you will implement the data structure flawlessly this week, use it next week, and not return to look at the code for a long time. When you *do* return, your view is effectively that of a user of the code, with little or no memory of the implementation.

One side effect of this relationship is that we have all been reminded of the need to write comments. If you do not write comments, you will not be able to read the code. If, however, you design, document, and implement your interface carefully, you might not ever have to look at the implementation! That's good news because, for most of us, in a couple of months our code is as foreign to us as if someone else had implemented it. The end result: consider yourself a user and design and abide by your interface wherever possible. If you know of some public field that gives a hint of the implementation, do not make use of it. Instead, access the data through appropriate methods. You will be happy you did later, when you optimize your implementation.

Principle 3 *Design and abide by interfaces as though you were the user.*

A quick corollary to this statement is the following:

Principle 4 *Declare data fields* `protected`.

If the data are protected, you cannot access them from outside the class, and you are forced to abide by the restricted access of the interface.

1.10 Conclusions

The construction of substantial applications involves the development of complex and interacting structures. In object-oriented languages, we think of these structures as objects that communicate through the passing of messages or, more formally, the invocation of methods.

We use object orientation in Java to write the structures found in this book. It is possible, of course, to design data structures without object orientation, but any effective data structuring model ultimately depends on the use of some form of abstraction that allows the programmer to avoid considering the complexities of particular implementations.

In many languages, including Java, data abstraction is supported by separating the interface from the implementation of the data structure. To ensure that users cannot get past the interface to manipulate the structure in an uncontrolled fashion, the system controls access to fields, methods, and classes. The implementor plays an important role in making sure that the structure is usable, given the interface. This role is so important that we think of implementation as supporting the interface—sometimes usefully considered a *contract* between the implementor and the user. This analogy is useful because, as in the real world, if contracts are violated, someone gets upset!

Initial design of the interfaces for data structures arises from considering how they are used in simple applications. Those method calls that are required by the application determine the interface for the new structure and constrain, in various ways, the choices we make in implementing the object.

In our implementation of an `Association`, we can use the `Object` class—that class inherited by all other Java classes—to write very general data structures. The actual type of value that is stored in the `Association` is determined by the values passed to the constructors and mutators of the class. This ability to pass a subtype to any object that requires a super type is a strength of object-oriented languages—and helps to reduce the complexity of code.

Self Check Problems

Solutions to these problems begin on page 441.

1.1 What is meant by abstraction?

1.2 What is procedural abstraction?

1.3 What is data abstraction?

1.4 How does Java support the concept of a *message*?

1.5 What is the difference between an object and a class?

1.6 What makes up a method's signature?

1.7 What is the difference between an interface and an implementation?

1.8 What is the difference between an accessor and a mutator?

1.9 A general purpose class, such as an `Association`, often makes use of parameters of type `Object`. Why?

1.10 What is the difference between a reference and an object?

1.11 Who uses a class?

Problems

Solutions to the odd-numbered problems begin on page 451.

1.1 Which of the following are primitive Java types: `int`, `Integer`, `double`, `Double`, `String`, `char`, `Association`, `BankAccount`, `boolean`, `Boolean`?

1.2 Which of the following variables are associated with valid constructor calls?

```
BankAccount a,b,c,d,e,f;
Association g,h;
a = new BankAccount("Bob",300.0);
b = new BankAccount(300.0,"Bob");
c = new BankAccount(033414,300.0);
d = new BankAccount("Bob",300);
e = new BankAccount("Bob",new Double(300));
f = new BankAccount("Bob",(double)300);
g = new Association("Alice",300.0);
h = new Association("Alice",new Double(300));
```

1.3 For each pair of classes, indicate which class extends the other:

a. `java.lang.Number`, `java.lang.Double`

b. `java.lang.Number`, `java.lang.Integer`

c. `java.lang.Number`, `java.lang.Object`

d. `java.util.Stack`, `java.util.Vector`

e. `java.util.Hashtable`, `java.util.Dictionary`

1.4 Rewrite the compound interest program (discussed when considering `BankAccounts` in Section 1.4) so that it uses `Associations`.

1.5 Write a program that attempts to modify one of the private fields of an `Association`. When does your environment detect the violation? What happens?

1.6 Finish the design of a `Ratio` class that implements a ratio between two integers. The class should support standard math operations: addition, subtraction, multiplication, and division. You should also be able to construct `Ratios` from either a numerator-denominator pair, or a single integer, or with no parameter at all (what is a reasonable default value?).

1.7 Amazing fact: If you construct a `Ratio` from two random integers, $0 < a, b$, the probability that $\frac{a}{b}$ is already in reduced terms is $\frac{6}{\pi^2}$. Use this fact to write a program to compute an approximation to π.

1.8 Design a class to represent a U.S. telephone number. It should support three types of constructors—one that accepts three numbers, representing area code, exchange, and extension; another that accepts two integers, representing a number within your local area code; and a third constructor that accepts a string of letters and numbers that represent the number (e.g., `"900-410-TIME"`). Provide a method that determines if the number is provided toll-free (such numbers have area codes of 800, 866, 877, 880, 881, 882, or 888).

1.9 Sometimes it is useful to measure the length of time it takes for a piece of code to run. (For example, it may help determine where optimizations of your code would be most effective.) Design a `Stopwatch` class to support timing of events. You should consider use of the nanosecond clock in the Java environment, `System.nanoTime()`. Like many stopwatches, it should support starting, temporary stopping, and a reset. The design of the protected section of the stopwatch should hide the implementation details.

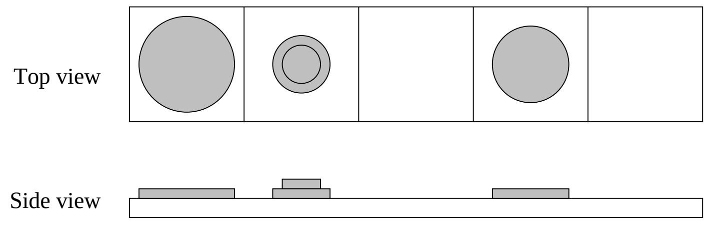
1.10 Design a data structure in Java that represents a musical tone. A `tone` can be completely specified as a number of cycles per second (labeled Hz for hertz), or the number of half steps above a commonly agreed upon tone, such as A (in modern times, in the United States, considered to be 440 Hz). Higher tones have higher frequencies. Two tones are an octave (12 semitones) apart if one has a frequency twice the other. A half step or semitone increase in tone is $\sqrt[12]{2} \approx 1.06$ times higher. Your `tone` constructors should accept a frequency (a `double`) or a number of half steps (an `int`) above A. Imperfect frequencies should be tuned to the nearest half step. Once constructed, a tone should be able to provide its frequency in either cycles per second or half-steps above A.

1.11 Extend Problem 1.10 to allow a second parameter to each constructor to specify the definition of A upon which the `tone`'s definition is based. What modern tone most closely resembles that of modern middle C (9 semitones below A) if A is defined to be 415 Hz?

1.12 Design a data structure to represent a combination lock. When the lock is constructed, it is provided with an arbitrary length array of integers between 0 and 25 specifying a combination (if no combination is provided, $9 - 0 - 21 - 0$ is the default). Initially, it is locked. Two methods—`press` and `reset`—provide a means of entering a combination: `press` enters the next integer to be used toward matching the combination, while `reset` re-readies the lock for accepting the first integer of the combination. Only when `press` is used to match the last integer of the combination does the lock silently unlock. Mismatched integers require a call to the `reset` method before the combination can again be entered. The `isLocked` method returns true if and only if the lock is locked. The `lock` method locks and resets the lock. In the unlocked state only the `isLocked` and `lock` methods have effect. (Aside: Because of the physical construction of many combination locks, it is often the case that combinations have patterns. For example, a certain popular lock is constructed with a three-number combination. The first and last numbers result in the same remainder x when divided by 4. The middle number has remainder $(x + 2)\%4$ when divided by 4!)

1.13 Design a data structure to simulate the workings of a car radio. The state of the radio is on or off, and it may be used to listen to an AM or FM station. A dozen modifiable push buttons (identified by integers 1 through 12) allow the listener to store and recall AM or FM frequencies. AM frequencies can be represented by multiples of 10 in the range 530 to 1610. FM frequencies are found at multiples of 0.2 in the range 87.9 to 107.9.

1.14 Design a data structure to maintain the position of m coins of radius 1 through m on a board with $n \geq m$ squares numbered 0 through $n - 1$. You may provide whatever interface you find useful to allow your structure to represent any placement of coins, including stacks of coins in a single cell. A configuration is *valid* only if large coins are not stacked on small coins. Your structure should have an `isValid` method that returns true if the coins are in a valid position. (A problem related to this is discussed in Section 10.2.1.)



1.11 Laboratory: The Day of the Week Calculator

Objective. To (re)establish ties with Java: to write a program that reminds us of the particulars of numeric calculations and array manipulation in Java.

Discussion. In this lab we learn to compute the day of the week for any date between January 1, 1900, and December 31, 2099.[4] During this period of time, the only calendar adjustment is a leap-year correction every 4 years. (Years divisible by 100 are normally not leap years, but years divisible by 400 always are.) Knowing this, the method essentially computes the number of days since the beginning of the twentieth century in modulo 7 arithmetic. The computed remainder tells us the day of the week, where 0 is Saturday.

An essential feature of this algorithm involves remembering a short table of monthly adjustments. Each entry in the table corresponds to a month, where January is month 1 and December is month 12.

Month	1	2	3	4	5	6	7	8	9	10	11	12
Adjustment	1	4	4	0	2	5	0	3	6	1	4	6

If the year is divisible by 4 (it's a leap year) and the date is January or February, you must subtract 1 from the adjustment.

Remembering this table is equivalent to remembering how many days are in each month. Notice that 144 is 12^2, 025 is 5^2, 036 is 6^2, and 146 is a bit more than 12^2. Given this, the algorithm is fairly simple:

1. Write down the date numerically. The date consists of a month between 1 and 12, a day of the month between 1 and 31, and the number of years since 1900. Grace Hopper, computer language pioneer, was born December 9, 1906. That would be represented as year 6. Jana the Giraffe, of the National Zoo, was born on January 18, 2001. That year would be represented as year 101.

2. Compute the sum of the following quantities:

   - the month adjustment from the given table (e.g., 6 for Admiral Hopper)
   - the day of the month
   - the year

[4] This particular technique is due to John Conway, of Princeton University. Professor Conway answers 10 day of the week problems before gaining access to his computer. His record is at the time of this writing well under 15 seconds for 10 correctly answered questions. See "Scientist at Work: John H. Conway; At Home in the Elusive World of Mathematics," *The New York Times*, October 12, 1993.

- the whole number of times 4 divides the year (e.g., 25 for Jana the Giraffe)

3. Compute the remainder of the sum of step 2, when divided by 7. The remainder gives the day of the week, where Saturday is 0, Sunday is 1, etc. Notice that we can compute the remainders *before* we compute the sum. You may also have to compute the remainder after the sum as well, but if you're doing this in your head, this considerably simplifies the arithmetic.

What day of the week was Tiger Woods born?

1. Tiger's birth date is 12-30-75.
2. Remembering that $18 \times 4 = 72$, we write the sum as follows:

   $$6 + 30 + 75 + 18$$

   which is equivalent to the following sum, modulo 7:

   $$6 + 2 + 5 + 4 = 17 \equiv 3 \bmod 7$$

3. He was born on day 3, a Tuesday.

Now you practice: Which of Grace and Jana was born on a Thursday? (The other was born on a Sunday.)

Procedure. Write a Java program that performs Conway's day of the week challenge:

1. Develop an object that can hold a date.
2. Write a method to compute a random date between 1900 and 2099. How will you limit the range of days potentially generated for any particular month?
3. Write a method of your date class to compute the day of the week associated with a date. Be careful: the table given in the discussion has January as month 1, but Java would prefer it to be month 0! Don't forget to handle the birthday of Jimmy Dorsey (famous jazzman), February 29, 1904.

Jimmy was a Monday's child.

4. Your `main` method should repeatedly (1) print a random date, (2) read a predicted day of the week (as an integer/remainder), and (3) check the correctness of the guess. The program should stop when 10 dates have been guessed correctly and print the elapsed time. (You may wish to set this threshold lower while you're testing the program.)

Helpful Hints. You may find the following Java useful:

1. Random integers may be selected using the `java.util.Random` class:

```
Random r = new Random();
int month = (Math.abs(r.nextInt()) % 12) + 1;
```

You will need to `import java.util.Random;` at the top of your program to make use of this class. Be aware that you need to only construct one random number generator per program run. Also, the random number generator potentially returns negative numbers. If `Math.abs` is not used, these values generate negative remainders.

2. You can find out how many thousandths of seconds have elapsed since the 1960s, by calling the Java method, `System.currentTimeMillis()`. It returns a value of type `long`. We can use this to measure the duration of an experiment, with code similar to the following:

In 2001, 1 trillion millis since the '60s. Dig that!

```
long start = System.currentTimeMillis();
//
// place experiment to be timed here
//
long duration = System.currentTimeMillis()-start;
System.out.println("time: "+(duration/1000.0)+" seconds.");
```

The granularity of this timer isn't any better than a thousandth of a second. Still, we're probably not in Conway's league yet.

After you finish your program, you will find you can quickly learn to answer 10 of these day of the week challenges in less than a minute.

Thought Questions. Consider the following questions as you complete the lab:

1. True or not: In Java is it true that `(a % 7) == (a - a/7*7)` for `a >= 0`?
2. It's rough to start a week on Saturday. What adjustments would be necessary to have a remainder of 0 associated with Sunday? (This might allow a mnemonic of Nun-day, One-day, Twos-day, Wednesday, Fours-day, Fives-day, Saturday.)
3. Why do you *subtract* 1 in a leap year if the date falls before March?
4. It might be useful to compute the portion of any calculation associated with this year, modulo 7. Remembering that value will allow you to optimize your most frequent date calculations. What is the remainder associated with this year?

For years divisible by 28: think zero!

Notes: