
CHAPTER 1

# Getting Organized

## Knowledge Goals

*You should be able to*

- describe some benefits of object-oriented programming
- describe the genesis of the Unified Method
- explain the relationships among classes, objects, and applications
- explain how method calls are bound to method implementations with respect to inheritance
- describe, at an abstract level, the following structures: array, linked list, stack, queue, list, tree, map, and graph
- identify which structures are implementation dependent and which are implementation independent
- describe the difference between direct addressing and indirect addressing
- explain the subtle ramifications of using references/pointers
- explain the use of O notation to describe the amount of work done by an algorithm
- describe the sequential search, binary search, and selection sort algorithms

## Skill Goals

*You should be able to*

- interpret a basic UML class diagram
- design and implement a Java class
- create a Java application that uses the Java class
- use packages to organize Java compilation units
- create a Java exception class
- throw Java exceptions from within a class and catch them within an application that uses the class
- predict the output of short segments of Java code that exhibit aliasing
- declare, initialize, and use one- and two-dimensional arrays in Java, including both arrays of a primitive type and arrays of objects
- given an algorithm, identify an appropriate size representation and determine its order of growth
- given a section of code determine its order of growth

Before embarking on any new project, it is a good idea to prepare carefully—to "get organized." In this first chapter that is exactly what we do. A careful study of the topics of this chapter will prepare us for the material on data structures and algorithms, using the object-oriented approach, covered in the remainder of the book.

## 1.1 Classes, Objects, and Applications

Software design is an interesting, challenging, and rewarding task. As a beginning student of computer science, you wrote programs that solved relatively simple problems. Much of your effort went into learning the syntax of a programming language such as Java: the language's reserved words, its data types, its constructs for selection and looping, and its input/output mechanisms.

As your programs and the problems they solve become more complex it is important to follow a software design approach that modularizes your solutions—breaks them into coherent manageable subunits. Software design was originally driven by an emphasis on actions. Programs were modularized by breaking them into subprograms or procedures/functions. A subprogram performs some calculations and returns information to the calling program, but it does not "remember" anything. In the late 1960s, researchers argued that this approach was too limiting and did not allow us to successfully represent the constructs needed to build complex systems.

Two Norwegians, Kristen Nygaard and Ole-Johan Dahl, created Simula 67 in 1967. It was the first language to support object-oriented programming. Object-oriented languages promote the object as the prime modularization mechanism. Objects represent both information and behavior and can "remember" internal information from one use to the next. This crucial difference allows them to be used in many versatile ways. In 2001, Nygaard and Dahl received the Turing Award, sometimes referred to as the "Nobel Prize of Computing," for their work.

The capability of objects to represent both information (the objects have *attributes*) and behavior (the objects have *responsibilities*) allows them to be used to represent "real-world" entities as varied as bank accounts, genomes, and hobbits. The self-contained nature of objects makes them easy to implement, modify, and test for correctness.

Object orientation is centered on classes and objects. Objects are the basic run-time entities used by applications. An object is an instantiation of a class; alternatively, a class defines the structure of its objects. In this section we review these object-oriented programming constructs that we use to organize our programs.

### Classes

A class defines the structure of an object or a set of objects. A class definition includes variables (data) and methods (actions) that determine the behavior of an object. The following Java code defines a `Date` class that can be used to create and manipulate `Date` objects—for example, within a school course-scheduling application. The `Date` class can be used to create `Date` objects and to learn about the year, month, or day of any particular

Date object.[1] The class also provides methods that return the Lilian Day Number of the date (the code details have been omitted—see the feature section on Lilian Day Numbers for more information) and return a string representation of the date.

**Authors' Convention**

Java-reserved words (when used as such), user-defined identifiers, class and file names, and so on, appear in `this font` throughout the entire text.

```
//--------------------------------------------------------------------------
// Date.java               by Dale/Joyce/Weems                Chapter 1
//
// Defines date objects with year, month, and day attributes.
//--------------------------------------------------------------------------
package ch01.dates;
public class Date
{
    protected int year, month, day;
    public static final int MINYEAR = 1583;

    // Constructor
    public Date(int newMonth, int newDay, int newYear)
    {
       month = newMonth; day = newDay; year = newYear;
    }

    // Observers
    public int getYear() { return year; }
    public int getMonth() { return month; }
    public int getDay(){ return day; }

    public int lilian()
    {
       // Returns the Lilian Day Number of this date.
       // Algorithm goes here. Code is included with the program files.
       // See Lilian Day Numbers feature section for details.
    }

    @Override[2]
    public String toString()
```

[1] The Java library includes a `Date` class, `java.util.Date`. However, the familiar properties of dates make them a natural example to use in explaining object-oriented concepts. Here we ignore the existence of the library class, as if we must design our own `Date` class.

[2] The purpose of `@Override` is discussed in Section 1.2 "Organizing Classes."

```
    // Returns this date as a String.
    {
       return(month + "/" + day + "/" + year);
    }
}
```

The `Date` class demonstrates two kinds of variables: instance variables and class variables. The instance variables of this class are `year`, `month`, and `day` declared as

```
protected int year, month, day;
```

Their values vary for each "instance" of an object of the class. Instance variables provide the internal representation of an object's attributes.

The variable `MINYEAR` is declared as

```
public static final int MINYEAR = 1583;
```

`MINYEAR` is defined as being `static`, and thus it is a class variable. It is associated directly with the `Date` class, instead of with objects of the class. A single copy of a class variable is maintained for all objects of the class.

Remember that the `final` modifier states that a variable is in its final form and cannot be modified; thus `MINYEAR` is a constant. By convention, we use only capital letters when naming constants. It is standard procedure to declare constants as class variables. Because the value of the variable cannot change, there is no need to force every object of a class to carry around its own version of the value. In addition to holding shared constants, class variables can be used to maintain information that is common to an entire class. For example, a `BankAccount` class may have a class variable that holds the number of current accounts.

**Authors' Convention**

We highlight important terms that might be unfamiliar to the student in **green**, the first time they are featured, to indicate that their definition can be found in the glossary in Appendix E.

In the `Date` class example, the `MINYEAR` constant represents the first full year that the widely used Gregorian calendar was in effect. The idea here is that programmers should not use the class to represent dates that predate that year. We look at ways to enforce this rule in Section 1.3 "Exceptional Situations," where we discuss handling exceptional situations.

The methods of the class are `Date`, `getYear`, `getMonth`, `getDay`, `lilian`, and `toString`. Note that the `Date` method has the same name as the class. Recall that this means it is a special type of method, called a class **constructor**. Constructors are used to create new instances of a class—that is, to instantiate objects of a class. The other methods are classified as **observer** methods, because they "observe" and return information based on the instance variable values. Other names for observer methods are "accessor" methods and "getters," as in accessing or getting information. Methods that simply return the value of an instance variable, such as `getYear()` in our `Date` class, are very common and always follow the same code pattern consisting of a single `return` statement. For this reason we will format such methods as a single line of code. In addition to constructors

**Table 1.1** Java Access Control Modifiers

| | Access Is Allowed | | | |
|---|---|---|---|---|
| | Within the Class | Within the Package | Within Subclasses | Everywhere |
| `public` | X | X | X | X |
| `protected` | X | X | X | |
| `package` | X | X | | |
| `private` | X | | | |

and observers, there is another general category of method, called a **transformer**. As you probably recall, transformers change the object in some way; for example, a method that changes the year of a `Date` object would be classified as a transformer.

You have undoubtedly noticed the use of the access modifiers `protected` and `public` within the `Date` class. Let us review the purpose and use of **access modifiers**. This discussion assumes you recall the basic ideas behind inheritance and packages. Inheritance supports the extension of one class, called the superclass, by another class, called the subclass. The subclass "inherits" properties (data and actions) from the superclass. We say that the subclass is derived from the superclass. Packages let us group related classes together into a single unit. Inheritance and packages are both discussed more extensively in the next section.

Java allows a wide spectrum of access control, as summarized in **Table 1.1**. The `public` access modifier used with the methods of `Date` makes them "publicly" available; any code that can "see" an object of the class can use its public methods. We say that these methods are "exported" from the class. Additionally, any class that is derived from the `Date` class using inheritance inherits its public methods and variables.

Public access sits at one end of the access spectrum, allowing open access. At the other end of the spectrum is private access. When you declare a class's variables and methods as `private`, they can be used only inside the class itself and are not inherited by subclasses. You should routinely use private (or protected) access within your classes to hide their data. You do not want the data values to be changed by code that is outside the class. For example, if the `month` instance variable in our `Date` class was declared to be `public`, then the application code could directly set the value of a `Date` object's `month` to strange numbers such as −12 or 27.

An exception to this guideline of hiding data within a class is shown in the `Date` example. Notice that the `MINYEAR` constant is publicly accessible. It can be accessed directly by the application code. For example, an application could include the statement

```
if (myYear < Date.MINYEAR) ...
```

Because `MINYEAR` is a final constant, its value cannot be changed by the application. Thus, even though it is publicly accessible, no other code can change its value. It is not necessary

to hide it. The application code above also shows how to access a public class variable from outside the class. Because `MINYEAR` is a class variable, it is accessed through the class name, `Date`, rather than through an object of the class.

Private access affords the strongest protection. Access is allowed only within the class. However, if you plan to extend your classes using inheritance, you may want to use protected access instead.

**Coding Convention**

We use protected access extensively for instance variables within our classes in this text.

The `protected` access modifier used in `Date` provides visibility similar to private access, only slightly less rigid. It "protects" its data from outside access, but allows the data to be accessed from within its own package *or* from any class derived from its class. Therefore, the methods within the `Date` class can access `year`, `month`, and `day`, and if, as we will show in Section 1.2 "Organizing Classes," the `Date` class is extended, the methods in the extended class can also access those variables.

The remaining type of access is called package access. A variable or method of a class defaults to package access if none of the other three modifiers are used. Package access means that the variable or method is accessible to any other class in the same package.

## Lilian Day Numbers

Various approaches to numbering days have been proposed. Most choose a particular day in history as day 1, and then number the actual sequence of days from that day forward with the numbers 2, 3, and so on. The Lilian Day Number (LDN) system uses October 15, 1582, as day 1, or LDN 1.

Our current calendar is called the Gregorian calendar. It was established in 1582 by Pope Gregory XIII. At that time 10 days were dropped from the month of October, to make up for small errors that had accumulated throughout the years. Thus, the day following October 4, 1582, in the Gregorian calendar is October 15, 1582, also known as LDN 1 in the Lilian day numbering scheme. The scheme is named after Aloysius Lilius, an advisor to Pope Gregory and one of the principal instigators of the calendar reform.

Originally, Catholic European countries adopted the Gregorian calendar. Many Protestant nations, such as England and its colonies, did not adopt the Gregorian calendar until 1752, at which

| 1582 | | OCTOBER | | | | 1582 |
|---|---|---|---|---|---|---|
| SUN | MON | TUE | WED | THU | FRI | SAT |
| | 1 | 2 | 3 | 4 | 15 | 16 |
| 17 | 18 | 19 | 20 | 21 | 22 | 23 |
| 24 | 25 | 26 | 27 | 28 | 29 | 30 |
| 31 | | | | | | |

time they also "lost" 11 days. Today, most countries use the Gregorian calendar, at least for official international business. When comparing historical dates, one must be careful about which calendars are being used.

In our `Date` class implementation, `MINYEAR` is 1583, representing the first full year during which the Gregorian calendar was in operation. We assume that programmers will not use the `Date` class to represent dates before that time, although this rule is not enforced by the class. This assumption simplifies calculation of day numbers, as we do not have to worry about the phantom 10 days of October 1582.

To calculate LDNs, one must understand how the Gregorian calendar works. Years are usually 365 days long. However, every year evenly divisible by 4 is a leap year, 366 days long. This aligns the calendar closer to astronomical reality. To fine-tune the adjustment, if a year is evenly divisible by 100, it is not a leap year but, if it is also evenly divisible by 400, it is a leap year. Thus 2000 was a leap year, but 1900 was not.

Given a date, the `lilian` method of the `Date` class counts the number of days between that date and the hypothetical date 1/1/0—that is, January 1 of the year 0. This count is made under the assumption that the Gregorian reforms were in place during that entire time period. In other words, it uses the rules described in the previous paragraph. Let us call this number the Relative Day Number (RDN). To transform a given RDN to its corresponding LDN, we just need to subtract the RDN of October 14, 1582, from it. For example, to calculate the LDN of July 4, 1776, the method first calculates its RDN (648,856) and then subtracts from it the RDN of October 14, 1582 (578,100), giving the result of 70,756.

Code for the `lilian` method is included with the program code files.

## The Unified Method

The object-oriented approach to programming is based on implementing models of reality. But how do you go about this? Where do you start? How do you proceed? The best plan is to follow an organized approach called a **methodology**.

In the late 1980s, many people proposed object-oriented methodologies. By the mid-1990s, three proposals stood out: the Object Modeling Technique, the Objectory Process, and the Booch Method. Between 1994 and 1997, the primary authors of these proposals got together and consolidated their ideas. The resulting methodology was dubbed the Unified Method. It is now, by far, the most popular organized approach to creating object-oriented systems.

The Unified Method features three key elements:

1. It is use-case driven. A use-case is a description of a sequence of actions performed by a user within the system to accomplish some task. The term "user" here should be interpreted in a broad sense and could represent another system.
2. It is architecture-centric. The word "architecture" refers to the overall structure of the target system, the way in which its components interact.

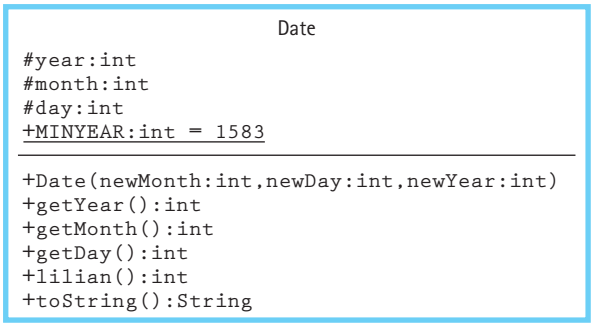



**Figure 1.1** UML class diagram for the `Date` class

3. It is iterative and incremental. The Unified Method involves a series of development cycles, with each one building upon the foundation established by its predecessors.

One of the main benefits of the Unified Method is improved communication among the people involved in the project. The Unified Method includes a set of diagrams for this purpose, called the **Unified Modeling Language (UML)**.[3] UML diagrams have become a de facto industry standard for modeling software. They are used to specify, visualize, construct, and document the components of a software system. We use UML class diagrams throughout this text to model our classes and their interrelationships.

A diagram representing the `Date` class is shown in **Figure 1.1**. The diagram follows the standard UML class notation approach. The name of the class appears in the top section of the diagram, the variables (attributes) appear in the next section, and the methods (operations) appear in the final section. The diagram includes information about the nature of the variables and method parameters; for example, we can see at a glance that `year`, `month`, and `day` are all of type `int`. Note that the variable `MINYEAR` is underlined; this indicates that it is a class variable rather than an instance variable. The diagram also indicates the visibility or protection associated with each part of the class (+ = public, # = protected).

## Objects

Objects are created from classes at run time. They can contain and manipulate data. Multiple objects can be created from the same class definition. Once a class such as `Date` has been defined, a program can create and use objects of that class. The effect is similar to expanding the language's set of standard types to include a `Date` type. To create an object in Java we use the `new` operator, along with the class constructor, as follows:

```
Date myDate = new Date(6, 24, 1951);
Date yourDate = new Date(10, 11, 1953);
Date ourDate = new Date(6, 15, 1985);
```

[3] The official definition of the UML is maintained by the Object Management Group. Detailed information can be found at http://www.uml.org/.

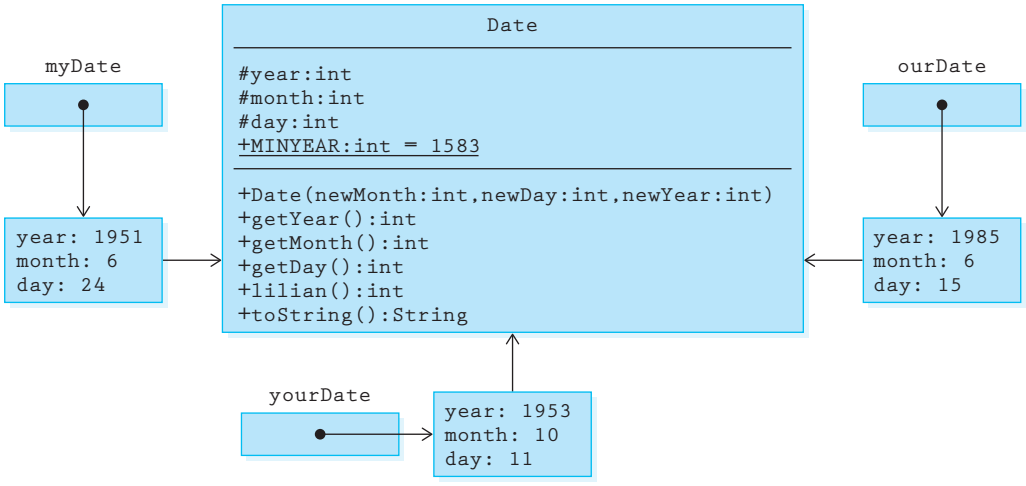



**Figure 1.2** Class diagram showing `Date` objects

We say that the variables `myDate`, `yourDate`, and `ourDate` reference "objects of the class `Date`" or simply "objects of type `Date`." We could also refer to them as "`Date` objects."

**Figure 1.2** extends our previous diagram (shown in Figure 1.1) to show the relationship between the instantiated `Date` objects and the `Date` class. As you can see, the objects are associated with the class, as represented by arrows from the objects to the class in the diagram. Notice that the `myDate`, `yourDate`, and `ourDate` variables are not objects, but actually hold references to the objects. The references are shown by the arrows from the variable boxes to the objects. In reality, references are memory addresses. The memory address of the instantiated object is stored in the memory location assigned to the variable. If no object has been instantiated for a particular variable, then its memory location holds a `null` reference.

Methods are invoked through the object upon which they are to act. For example, to assign the return value of the `getYear` method of the `ourDate` object to the integer variable `theYear`, a programmer would code

```
theYear = ourDate.getYear();
```

Recall that the `toString` method is invoked in a special way. Just as Java automatically changes an integer value, such as that returned by `getDay`, to a string in the statement

```
System.out.println("The big day is " + ourDate.getDay());
```

it automatically changes an object, such as `ourDate`, to a string in the statement

```
System.out.println("The party will be on " + ourDate);
```

The output from these statements would be

```
The big day is 15
The party will be on 6/15/1985
```

To determine how to change the object to a string, the Java compiler looks for a `toString` method for that object, such as the `toString` method we defined for `Date` objects in our `Date` class.

## Applications

You should view an object-oriented program as a set of objects working together, by sending one another messages, to solve a problem. But where does it all begin? How are the objects created in the first place?

A Java program typically begins running when the user executes the **Java Virtual Machine** and passes it the program. How you begin executing the Java Virtual Machine depends on your environment. You may simply use the command "java" if you are working in a command line environment. Or, you may click a "run" icon if you are working within an integrated development environment. In any case, you indicate the name of a class that contains a `main` method. The Java Virtual Machine loads that class and starts executing that method. The class that contains the `main` method is called a **Java application**.

Suppose we want to write a program named `DaysBetween` that provides information about the number of days between two dates. The idea is for the program to prompt the user for two dates, calculate the number of days between them, and report this information back to the user.

In object-oriented programming a key step is identifying classes that can be used to help solve a problem. Our `Date` class is a perfect fit for the days-between problem. It allows us to create and access `Date` objects. Plus, its `lilian` method returns a value that can help us determine the number of days between two dates. We simply subtract the two Lilian Day Numbers. The design of our application code is straightforward—prompt for and read in the two dates, check that valid years are provided, and then display the difference between the Lilian Day Numbers.

**Design Convention**

Our application code usually consists of a class with a single method—`main`. Modularization is provided by using externally defined classes and objects.

The application code is shown below. Some items to note:

- The application imports the `util` package from the Java Class Library. The `util` package contains Java's `Scanner` class, which the application uses for input.
- The `DaysBetween` class contains just a single method, the `main` method. It is possible to define other methods within the class and to invoke them from the `main` method. Such functional modularization can be used if the `main` method becomes long and complicated. However, because we are emphasizing an object-oriented approach, our application code rarely subdivides a solution in that manner. Classes and objects are our primary modularization mechanisms, not application methods.
- Although the program checks to ensure the entered years of the dates are "modern," it does not do any other input correctness checking. In general, throughout the text, we assume the users of our applications are "friendly," that is, they enter input correctly.

```
//-------------------------------------------------------------------------
// DaysBetween.java          by Dale/Joyce/Weems               Chapter 1
//
// Asks the user to enter two "modern" dates and then reports
// the number of days between the two dates.
//-------------------------------------------------------------------------
package ch01.apps;

import java.util.Scanner; import ch01.dates.*;

public class DaysBetween
{
  public static void main(String[] args)
  {
    Scanner scan = new Scanner(System.in);
    int day, month, year;

    System.out.println("Enter two 'modern' dates: month day year");
    System.out.println("For example, January 21, 1939, would be: 1 21 1939");
    System.out.println();
    System.out.println("Modern dates are not before " + Date.MINYEAR + ".");
    System.out.println();

    System.out.println("Enter the first date:");
    month = scan.nextInt(); day = scan.nextInt(); year = scan.nextInt();
    Date d1 = new Date(month, day, year);

    System.out.println("Enter the second date:");
    month = scan.nextInt(); day = scan.nextInt(); year = scan.nextInt();
    Date d2 = new Date(month, day, year);

    if ((d1.getYear() <= Date.MINYEAR) || (d2.getYear() <= Date.MINYEAR))
      System.out.println("You entered a 'pre-modern' date.");
    else
    {
      System.out.println("The number of days between");
      System.out.print(d1 + " and " + d2 + " is ");
      System.out.println(Math.abs(d1.lilian() - d2.lilian()));
    }
  }
}
```

Here is the result of a sample run of the application. User input is shown in `this color`.

```
Enter two 'modern' dates: month day year
For example, January 21, 1939, would be: 1 21 1939
Modern dates are not before 1583.
Enter the first date:
1 1 1900
Enter the second date:
1 1 2000
The number of days between
1/1/1900 and 1/1/2000 is 36524
```

# 1.2 Organizing Classes

During object-oriented development, dozens—even hundreds—of classes can be generated or reused to help build a system. The task of keeping track of all of these classes would be impossible without some type of organizational structure. In this section we review two of the most important ways of organizing Java classes: inheritance and packages. As you will see, both of these approaches are used "simultaneously" for most projects.

## Inheritance

**Inheritance** is much more than just an organizational mechanism. It is, in fact, a powerful reuse mechanism. Inheritance allows programmers to create a new class that is a specialization of an existing class. The new class is a **subclass** of the existing class that in turn is the **superclass** of the new class.

A subclass "inherits" features from its superclass. It adds new features, as needed, related to its specialization. It can also redefine inherited features as necessary by overriding them. "Super" and "sub" refer to the relative positions of the classes in a hierarchy. A subclass is below its superclass and a superclass is above its subclasses.

Suppose we already have a `Date` class as defined previously, and we are creating a new application to manipulate `Date` objects. Suppose also that the new application is often required to "increment" a `Date` object—that is, to change a `Date` object so that it represents the next day. For example, if the `Date` object represents 7/31/2001, it would represent 8/1/2001 after being incremented. The algorithm for incrementing the date is not trivial, especially when you consider leap year rules. But in addition to developing the algorithm, another question that must be addressed is where to put the code that implements the algorithm. There are several options:

- Implement the algorithm within the application. The application code would need to obtain the month, day, and year from the `Date` object using the observer

methods; calculate the new month, day, and year; instantiate a new `Date` object to hold the updated month, day, and year; and if required, assign all the variables that previously referenced the original `Date` to the new object. This might be a complex task so this is probably not the best approach. Besides, if future applications also need this functionality, their programmers would have to reimplement the solution for themselves. This approach does not promote reusability and possibly requires complex tracking of object aliases.

- Add a new method, called `increment`, to the `Date` class. This method would update the value of the current object. Such an approach allows future programs to use the new functionality. However, in some cases, a programmer may want a `Date` class with protection against any changes to its objects. Such objects are said to be **immutable**. Adding `increment` to the `Date` class undermines this protection.
- Add a new method, called `nextDay`, to the `Date` class. Rather than updating the value of the "current" object, `nextDay` would return a new `Date` object that represents the day after the `Date` object upon which it is invoked. An application could then reassign a `Date` variable to its next day, perhaps like this:

  ```
  d1 = d1.nextDay();
  ```

  This approach resolves the drawbacks of the previous approach in that the `Date` objects remain immutable, although if one wants all variables that referenced the original object to also reflect the updated information it is lacking. Aliases of the `d1` object will not be updated.
- Use inheritance. Create a new class, called `IncDate`, that inherits all the features of the current `Date` class, but that also provides the `increment` method. This approach allows `Date` objects to remain immutable but at the same time provides a mutable `Date-like` class that can be used by the new application.

**Important**

Inheritance is a powerful reuse mechanism that allows us to define a new class as an extension of a current class. The new class is a specialization of the current class. New features can be added and inherited features can be redefined.

We now look at how to implement the final option, that is, to use inheritance to solve our problem. The inheritance relationship is often called an *is-a* relationship. An object of the class `IncDate` is also a `Date` object, because it can do anything that a `Date` object can do—and more. This idea can be clarified by remembering that inheritance typically means specialization. `IncDate` *is-a* special case of `Date`, but not the other way around. Here is the code for `IncDate`:

```
package ch01.dates;
public class IncDate extends Date
{
   public IncDate(int newMonth, int newDay, int newYear)
```

```
    {
       super(newMonth, newDay, newYear);
    }

    public void increment()
    // Increments this IncDate to represent the next day.
    // For example, if this = 6/30/2005, then this becomes 7/1/2005.
    {
       // Increment algorithm goes here.
    }
}
```

**Authors' Convention**

Note that sometimes in our code listings we emphasize the sections of code most pertinent to the current discussion by underlining them.

Inheritance is indicated by the keyword `extends`, that shows that `IncDate` inherits from `Date`. It is not possible in Java to inherit constructors, so `IncDate` must supply its own. In this case, the `IncDate` constructor simply takes the month, day, and year arguments and passes them to the constructor of its superclass (that is, to the `Date` class constructor) using the `super` reserved word.

The other part of the `IncDate` class is the new `increment` method, which is classified as a transformer because it changes the internal state of the object. The `increment` method changes the object's `day` and possibly the `month` and `year` values. The method is invoked through the object that it is to transform. For example, if `aDate` is an object of type `IncDate` then the statement

```
aDate.increment();
```

transforms the `aDate` object.

Although we have left out the details of the `increment` method because they are not crucial to our current discussion, note that it would require access to the `year`, `month`, and `day` instance variables of its superclass. Therefore, using `protected` rather than `private` access for those variables within the `Date` class, as we did, is crucial for our approach to be viable.

A program with access to each of the date classes can now declare and use both Date and `IncDate` objects. Consider the following program segment:

```
Date myDate = new Date(6, 24, 1951);
IncDate aDate = new IncDate(1, 11, 2001);

System.out.println("myDate day is:   " + myDate.getDay());
System.out.println("aDate day is:    " + aDate.getDay());

aDate.increment();
System.out.println("the day after is: " + aDate.getDay());
```

This program segment **instantiates** and initializes `myDate` and `aDate`, outputs the values of their days, increments `aDate`, and finally outputs the new day value of `aDate`. You might ask, "How does the system resolve the use of the `getDay` method by an `IncDate` object when `getDay` is defined in the `Date` class?" Understanding how inheritance is supported by Java provides the answer to this question. The extended class diagram in **Figure 1.3**, that shows the inheritance relationships and captures the state of the system after the `aDate` object has been incremented, helps us investigate the situation. As is standard with UML class diagrams, inheritance is indicated by a solid arrow with an open arrow head (a triangle). Note that the arrow points from the subclass to the superclass.

The compiler has available to it all the declaration information captured in the extended class diagram. Consider the `getDay` method call in the statement

```
System.out.println("the day after is: " + aDate.getDay());
```

To resolve this method call, the compiler follows the reference from the `aDate` variable to the `IncDate` class. It does not find a definition for a `getDay` method in the `IncDate`

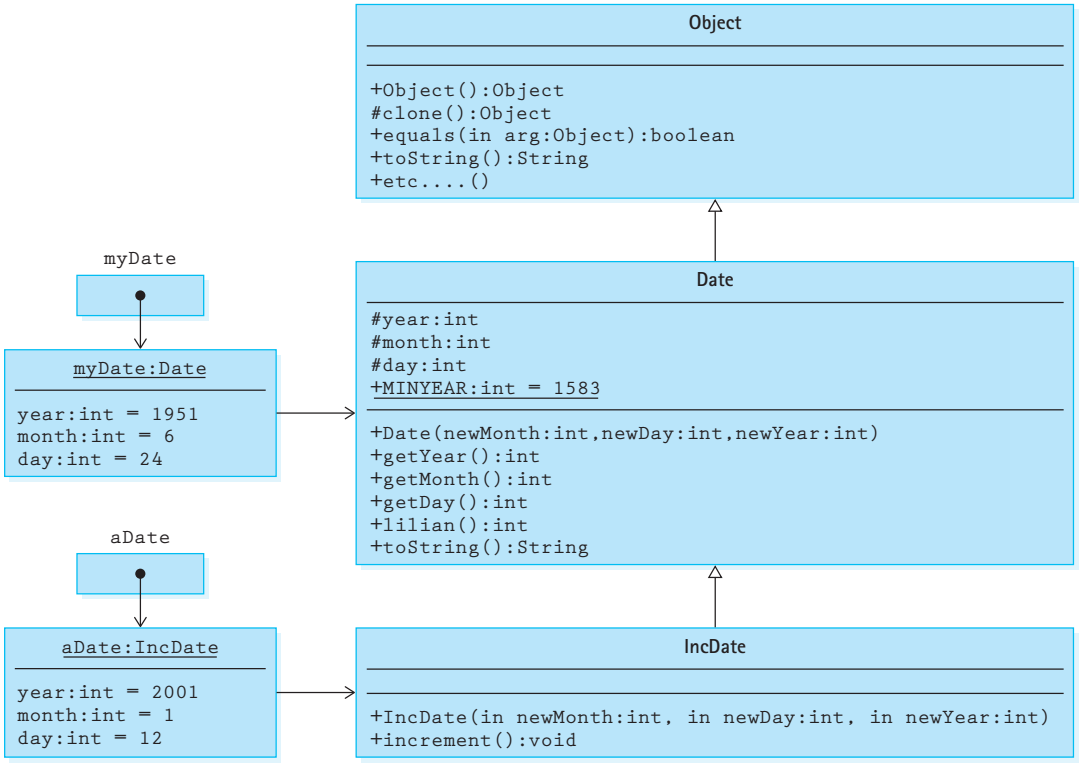



**Figure 1.3** Extended class diagram showing inheritance

class, so it follows the inheritance link to the superclass `Date`. There it finds, and uses, the `getDay` method. In this case, the `getDay` method returns an `int` value that represents the `day` value of the `aDate` object. During execution, the system changes the `int` value to a `String`, concatenates it to the string "the day after is: ", and prints it to `System.out`.

## The Inheritance Tree

Java supports single inheritance only. This means that a class can extend only one other class. Therefore, in Java, the inheritance relationships define an **inheritance tree**.

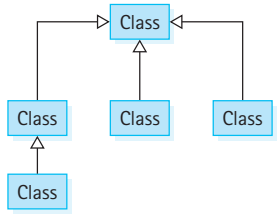



Figure 1.3 shows one branch of the overall system inheritance tree. Note that because of the way method calls are resolved, by searching *up* the inheritance tree, only objects of the class `IncDate` can use the `increment` method—if you try to use the `increment` method on an object of the class `Date`, such as the `myDate` object, no definition is available in either the `Date` class or any of the classes above `Date` in the inheritance tree. The compiler would report a syntax error in this situation.

**Important**

Association of method names with method code is accomplished by moving up the inheritance tree. If a matching method is not found in the named class, then its superclass is searched. And if not found there, then the superclass above that and so on.

**Java Note**

In Java, the `Object` class is the root of the inheritance tree—all classes inherit from `Object`. Therefore, for example, all objects support `equals` and `toString`, although unless their class overrides the `Object` class definitions of those methods, they may not support those operations well.

Notice the `Object` class in Figure 1.3. Where did it come from? In Java, any class that does not explicitly extend another class implicitly extends the predefined `Object` class. Because `Date` does not explicitly extend any other class, it inherits directly from `Object`. The `Date` class is a direct subclass of `Object`.

All Java classes can be traced up to the `Object` class. We say that the `Object` class is the root of the inheritance tree. The `Object` class defines several basic methods: comparison for equality (`equals`), conversion to a string (`toString`), and so on. Therefore, for example, any object in any Java program supports the method `toString` because it is inherited from the `Object` class. Let us consider the `toString` example more carefully.

As discussed previously, just as Java automatically changes an integer value to a string in the statement

```
System.out.println("aDate day is:    " + aDate.getDay());
```

so it automatically changes an object to a string in the statement

```
System.out.println("tomorrow: " + aDate);
```

To accomplish this, the Java compiler looks for a `toString` method for that object. In this case, the `toString` method is not found in the `IncDate` class, but it is found in its superclass, the `Date` class. However, if it was not defined in the `Date` class, the compiler would continue looking up the inheritance hierarchy and would find the `toString` method in the `Object` class. Given that all classes trace their roots back to `Object`, the compiler is always guaranteed to find a `toString` method eventually.

But wait a minute. What does it mean to "change an object to a string"? Well, that depends on the definition of the `toString` method that is associated with the object. The `toString` method of the `Object` class returns a string representing some of the internal system implementation details about the object. This information is somewhat cryptic and generally not useful to us. This situation is an example of where it is useful to redefine an inherited method by overriding it. We generally **override** the default `toString` method when creating our own classes so as to return a more relevant string, as we did with the `Date` class. This is why we use the `@Override` notation with the `toString` method as shown on page 3. By annotating our `toString` method as overriding an ancestor's `toString` method, we allow the compiler to double-check our syntax. If it cannot find an associated ancestor method with the same **signature**, it will generate an error. Additionally, some development environments will use the information to inform how they display the code.

**Table 1.2** shows the output from the following program segment:

```
Date myDate = new Date(6, 24, 1951);
IncDate currDate = new IncDate(1, 11, 2001);
System.out.println("mydate:   " + myDate);
System.out.println("today:    " + currDate);

currDate.increment();
System.out.println("tomorrow: " + currDate);
```

The results on the left show an example of the output generated if the `toString` method of the `Object` class is used by default; the results on the right show the outcome if the `toString` method of our `Date` class is used.

**Table 1.2** Output from Program Segment

| `Object` Class `toString` Used | | `Date` Class `toString` Used | |
|---|---|---|---|
| `mydate:` | `Date@256a7c` | `mydate:` | `6/24/1951` |
| `today:` | `IncDate@720eeb` | `today:` | `1/11/2001` |
| `tomorrow:` | `IncDate@720eeb` | `tomorrow:` | `1/12/2001` |

## Inheritance-Based Polymorphism

This is a good place to introduce an important object-oriented concept. The word **polymorphism** has Greek roots and literally means "many forms." Object-oriented languages that support polymorphism allow an object variable to reference objects of different classes at different times during the execution of a program—the variable can have "many types" and is called a polymorphic variable or polymorphic reference.

There are two ways to create polymorphic references with Java. Here we look at inheritance-based polymorphism. In Section 2.1, "Abstraction," we will look at interface-based polymorphism.

Typically in our programs we can tell exactly what method will be executed when a method is invoked through an object variable. For example, in the following code section the third and fourth lines respectively invoke the `toString` method of the `String` class and the `toString` method of the `Date` class.

```
String s = new String("Hello");
Date d = new Date(1,1,2015);
System.out.println(s.toString());
System.out.println(d.toString());
```

It is easy to see that this code will print "Hello" followed by "1/1/2015".

Remember that both `String` and `Date` inherit from the `Object` class. In terms of inheritance we say that a `String` "is-an" `Object` and that a `Date` also "is-an" `Object`. Due to the polymorphism built into the Java language this means that we can declare a variable to be of type `Object`, and then instantiate it as a `String` or as a `Date`. In fact, since the `Object` class is at the root of the Java inheritance tree, an `Object` reference can refer to an object of any class.

In the following code section assume that `cutoff` was assigned a random value between 1 and 100, perhaps through the `Random` class's `nextInt` method. Can you predict what method is invoked by the `obj.toString()` method invocation? Can you predict what will be printed? Do not forget that both the `String` class and the `Date` class override the `toString` method of the `Object` class.

```
Object obj;
if (cutoff <= 50)
   obj = new String("Hello");
else
  obj = new Date(1,1,2015);
System.out.println(obj.toString());
```

We cannot infer from the code whether the `obj` variable references a `String` or a `Date`. We can only infer that it references one or the other. The binding of the `obj` variable to a class occurs dynamically, at run time. As is implied by the arrows connecting objects to classes in Figure 1.3, each object carries information indicating the class to which it belongs. This can also be noticed in the output of the `Object` class's `toString` method,

displayed on the left side of Table 1.2. **Run-time** (also called **dynamic**) **binding** and polymorphism go hand in hand. We can only predict that half of the time the `toString` method of the `String` class is invoked and the other half of the time the `toString` method of the `Date` class is invoked.

**Important Concept**

Inheritance, overriding of methods, and dynamic binding all interact to support polymorphic references. Because objects carry with them information about their class, that information can vary dynamically, as long as it satisfies the is-a relationship established by the inheritance tree.

You might ask how the compiler can parse a method invocation to ensure syntactical correctness when run-time binding is used. The key is that the `Object` class itself defines a `toString` method. The compiler is able to verify that the `obj.toString()` invocation correctly matches a defined method in the `Object` class, and after all, `obj` was declared to be of type `Object`. The Java Virtual Machine, however, when executing the method invocation, follows the dynamically created reference from `obj` to either the `String` class definition or the `Date` class definition and uses the `toString` method defined there.

Although the preceding example does demonstrate polymorphism, it does not really do justice to the power of inheritance-based polymorphism or demonstrate how it should be used. The example was selected due to its simplicity and conciseness. We will see another example of how polymorphism can be used in the next chapter, and although we will not make extensive use of it throughout the text, it is an important object-oriented concept, useful for creating easily maintained, versatile, adaptable systems of classes. Its true power becomes apparent when constructing large enterprise-level systems and their interfaces. If you continue to study object orientation, you will find it a powerful and crucial tool.

## Packages

Java lets us group related classes together into a unit called a package. Packages provide several advantages:

- They let us organize our files.
- They can be compiled separately and imported into our programs.
- They make it easier for programs to use common class files.
- They help us avoid naming conflicts (two classes can have the same name if they are in different packages).

### Package Syntax

The syntax for a package is extremely simple. All one has to do is to specify the package name at the start of the file containing the class. The first noncomment, nonblank line of the file must contain the keyword `package` followed by an identifier and a semicolon. By convention, Java programmers start a package identifier with a lowercase letter to distinguish package names from class names:

```
package someName;
```

Following the package name specification in the file, the programmer can write import declarations, so as to make the contents of other packages available to the classes inside the package being defined, and then one or more declarations of classes. Java calls this file a *compilation unit*. The classes defined in the file are members of the package. The imported classes are not members of the package.

The name of the file containing the compilation unit must match the name of the public class within the unit. Therefore, although a programmer can declare multiple classes in a compilation unit, only one of them can be declared public. All nonpublic classes in the file are hidden from the world outside the package. If a compilation unit can hold at most one public class, how do we create packages with multiple public classes? We have to use multiple compilation units, as described next.

## Packages with Multiple Compilation Units

Each Java compilation unit is stored in its own file. The Java system identifies the file using a combination of the package name and the name of the public class in the compilation unit. Java restricts us to having a single public class per file so that it can use file names to locate public classes. Thus a package with multiple public classes is implemented as multiple compilation units, each in a separate file.

Using multiple compilation units has the further advantage of providing us with greater flexibility in developing the classes of a package. Team programming projects would be more cumbersome if Java made multiple programmers share a single package file.

We split a package among multiple files simply by placing its members into separate compilation units with the same package name. For example, we can create one file containing the following code (the . . . between the braces represents the code for each class):

```
package gamma;
public class One{ ... }
class Two{ ... }
```

A second file could contain this code:

```
package gamma;
class Three{ ... }
public class Four{ ... }
```

The result: The package `gamma` contains four classes. Two of the classes, `One` and `Four`, are public, so they are available to be imported by application code. The two file names must match the two public class names; that is, the files must be named `One.java` and `Four.java`, respectively.

How does the Java compiler manage to find these pieces and put them together? The answer is that it requires that all compilation unit files for a package be kept in a single directory or folder that matches the name of the package. For our preceding example, a programmer would store the source code in files called `One.java` and `Four.java`, both in a directory called `gamma`.

### The Import Statement

To access the contents of a package from within a program, you must import it into your program. You can use either of the following forms of import statements:

```
import packagename.*;
import packagename.Classname;
```

An import declaration begins with the keyword `import`, the name of a package, and a dot (period). Following the dot you can write either the name of a class in the package or an asterisk (*). The declaration ends with a semicolon. If you want to access exactly one class in a particular package, then you can simply use its name in the import declaration. If you want to use more than one of the classes in a package, the asterisk is a shorthand notation to the compiler that says, "Import whatever classes from this package that this program uses."

### Packages and Subdirectories

**Java Note**

The Java package construct is designed to work seamlessly with the commonly used hierarchical file system.

Many computer platforms use a hierarchical file system. The Java package rules are defined to work seamlessly with such systems. Java package names may also be hierarchical; they may contain "periods" separating different parts of the name—for example, `ch01.dates`. In such a case, the package files must be placed underneath a set of subdirectories that match the separate parts of the package name. Continuing the same example, the package files should be placed in a directory named `dates` that is a subdirectory of a directory named `ch01`. You can then import the entire package into your program with the following statement:

```
import ch01.dates.*;
```

As long as the directory that contains the `ch01` directory is on the `ClassPath` of your system, the compiler will be able to find the package you requested. The compiler automatically looks in all directories listed in `ClassPath`. Most programming environments provide a command to specify the directories to be included in the `ClassPath`. You will need to consult the documentation for your particular system to see how to do this. In our example, the compiler will search all `ClassPath` directories for a subdirectory named `ch01` that contains a subdirectory named `dates`; upon finding such a subdirectory, it will import all of the members of the `ch01.dates` package that it finds there.

### The Program Files

The files created to support this text are organized into packages. They are organized exactly as we have described and are available at the book's website, go.jblearning.com/oods4ecatalog/9781449613549/. All of the files are found in a directory named `bookFiles`. It contains a separate subdirectory for each chapter of the book: `ch01`, `ch02`, etc. You will find the corresponding subdirectories underneath the chapter subdirectories.

For example, the `ch01` subdirectory does, indeed, contain a subdirectory named `dates`, that in turn contains files that define Java classes related to dates. Each of the class files begins with the statement

```
package ch01.dates;
```

Thus they are all in the `ch01.dates` package. If you write a program that needs to use these files, you can simply import the package into your program and make sure the parent directory of the `ch01` directory (that is, the `bookFiles` directory), is included in your computer's `ClassPath`.

We suggest that you copy the entire `bookFiles` directory to your computer's hard drive, ensuring easy access to all of the book's files and maintaining the crucial subdirectory structure required by the packages. Also, make sure you extend your computer's `ClassPath` to include your new `bookFiles` directory.

# 1.3 Exceptional Situations

In this section we take a look at various methods of handling exceptional situations that might arise when running a program.

## Handling Exceptional Situations

Many different types of exceptional situations can occur when a program is running. Exceptional situations alter the flow of control of the program, sometimes resulting in a crash. Some examples follow:

- A user enters an input value of the wrong type.
- While reading information from a file, the end of the file is reached.
- A user presses a control key combination.
- A program attempts to invoke a method on a null object.
- An out-of-bounds value is passed to a method, for example, passing 25 as the month value to the `Date` constructor.

Java (along with some other languages) provides built-in mechanisms to manage exceptional situations. In Java an exceptional situation is referred to simply as an **exception**. The Java exception mechanism has three major parts:

- *Defining the exception.* Usually as a subclass of Java's `Exception` class
- *Generating (raising) the exception.* By recognizing the exceptional situation and then using Java's `throw` statement to "announce" that the exception has occurred.
- *Handling the exception.* Using Java's `try-catch` statement to discover that an exception has been thrown and then take the appropriate action.

Java also includes numerous predefined built-in exceptions that are raised automatically under certain situations.

From this point on we use the Java term "exception" instead of the more general phrase "exceptional situation." Here are some general guidelines for using exceptions:

**Java Note**

In Java, exceptions are objects. They can be defined, instantiated, raised, thrown, caught, and handled. They allow us to control the flow of execution of a program to handle exceptional situations.

- An exception may be handled anywhere in the software hierarchy—from the place in the program module where it is first detected through the top level of the program.
- Unhandled built-in exceptions carry the penalty of program termination.
- Where in an application an exception is handled is a design decision; however, exceptions should be handled at a level that knows what the exception means.
- An exception need not be fatal.
- For nonfatal exceptions, the thread of execution should continue from the lowest level that can recover from the exception.

## Exceptions and Classes: An Example

When creating our own classes we identify exceptions that require special processing. If the special processing is application dependent, we use the Java exception mechanism to throw the problem out of the class and force the application programmers to handle it. Conversely, if the exception handling can be hidden within the class, then there is no need to burden the application programmers with the task.

For an example of an exception created to support a programmer-defined class, we return to our `Date` class example. As currently defined, an application could invoke the `Date` constructor with an invalid month—for example, 25/15/2000. We can avoid the creation of such dates by checking the legality of the month argument passed to the constructor. But what should our constructor do if it discovers an illegal argument? Here are some options:

- Write a warning message to the output stream. This is not a good option because within the `Date` class we do not really know which output stream, if any, is used by the application.
- Instantiate the new `Date` object to some default date, perhaps 0/0/0. The problem with this approach is that the application program may just continue processing as if nothing is wrong and produce erroneous results. In general, it is better for a program to "bomb" than to produce erroneous results that may be used to make bad decisions.
- Throw an exception. This way, normal processing is interrupted and the constructor does not have to return a new object; instead, the application program is forced to acknowledge the problem (catch the exception) and either handle it or throw it to the next level.

Once we have decided to handle the situation with an exception, we must decide whether to use one of the Java library's predefined exceptions or to create one of our own. A study of the library in this case reveals a candidate exception called `DataFormatException`, to be used to signal data format errors. We could use that exception but decide it does not really fit; it is not the format of the data that is the problem in this case, it is the value of the data.

We decide to create our own exception, `DateOutOfBounds`. It could be called `MonthOutOfBounds`, but we decide that we want to use the exception to indicate other potential problems with dates, not just problems with the month value. Our exception class is placed in a file named `DateOutOfBounds.java`.

Our `DateOutOfBounds` exception extends the library's `Exception` class. It is customary when creating your own exceptions to define two constructors, mirroring the two constructors of the `Exception` class. In fact, the easiest thing to do is define the constructors so that they just call the corresponding constructors of the superclass:

```
package ch01.dates;
public class DateOutOfBoundsException extends Exception
{
   public DateOutOfBoundsException()
   {
      super();
   }
   public DateOutOfBoundsException(String message)
   {
      super(message);
   }
}
```

The first constructor creates an exception without an associated message. The second constructor creates an exception with a message equal to the string argument passed to the constructor.

Let us create a new class `SafeDate`. We could simply upgrade our previous `Date` class but do not want to invalidate our previous examples. So we will use the new class, `SafeDate`, to demonstrate the use of exceptions. Where, within our `SafeDate` class, should we throw the exception? All places within our class where a date value is created or changed should be examined to see if the resultant value could be an illegal date. If so, we should create an object of our exception class with an appropriate message and throw the exception.

Here is a `SafeDate` constructor that checks for legal months and years:

```
public SafeDate(int newMonth, int newDay, int newYear)
                throws DateOutOfBoundsException
{
   if ((newMonth <= 0) || (newMonth > 12))
      throw new DateOutOfBoundsException("Month " + newMonth + " illegal.");
```

```
   else
      month = newMonth;

   day = newDay;

   if (newYear < MINYEAR)
      throw new DateOutOfBoundsException("Year " + newYear + " too early.");
   else
      year = newYear;
}
```

Notice that the message defined for each `throw` statement pertains to the problem discovered at that point in the code. This should help the application program that is handling the exception, or at least provide pertinent information to the user of the program if the exception is propagated all the way to the user level.

Finally, we see how an application program might use the `SafeDate` class. Consider a program called `UseSafeDate` that prompts the user for a month, day, and year and creates a `SafeDate` object based on the user's responses. In the following code we hide the details of how the prompt and response are handled by replacing those statements with comments. This way we can *emphasize* the code related to our current discussion:

```
//-------------------------------------------------------------------------
// UseSafeDate.java          by Dale/Joyce/Weems               Chapter 1
//
// Example of re-throwing exceptions thrown by SafeDate class
//-------------------------------------------------------------------------

package ch01.apps;
public class UseSafeDate
{
  public static void main(String[] args) throws DateOutOfBoundsException
  {
    SafeDate theDate;

    // Program prompts user for a date.
    // M is set equal to user's month.
    // D is set equal to user's day.
    // Y is set equal to user's year.

    theDate = new SafeDate(M, D, Y);

    // Program continues ...
  }
}
```

When this program runs, if the user responds with an illegal value—for example, a year of 1051—the `DateOutOfBoundsException` is thrown by the `SafeDate` constructor; because it is not caught and handled within the program, it is thrown to the interpreter as indicated by the emphasized `throws` clause. The interpreter stops the program and displays a message like this:

```
Exception in thread "main" DateOutOfBoundsException: Year 1051 too early.
at SafeDate.<init>(SafeDate.java:18)
at UseSafeDate.main(UseSafeDate.java:57)
```

The interpreter's message includes the name and message string of the exception as well as a trace of calls leading up to the exception.

Alternatively, the `UseSafeDate` class could catch and handle the exception itself, rather than throw it to the interpreter. The application could ask for a new date when the exception occurs. Here is how `UseSafeDate` can be written to do this:

```
//--------------------------------------------------------------------------
// UseSafeDate.java          by Dale/Joyce/Weems                Chapter 1
//
// Example of catching exceptions thrown by SafeDate class
//--------------------------------------------------------------------------
package ch01.apps;

import java.util.Scanner; import ch01.dates.*;

public class UseSafeDate
{
  public static void main(String[] args)
  {
    int month, day, year;
    SafeDate theDate;
    boolean DateOK = false;
    Scanner scan = new Scanner(System.in);

    while (!DateOK)
    {
      System.out.println("Enter a date (month day and year):");
      month = scan.nextInt(); day = scan.nextInt(); year = scan.nextInt();
      try
      {
        theDate = new SafeDate(month, day, year);
        DateOK = true;
        System.out.println(theDate + " is a safe date.");
      }
```

```
      catch(DateOutOfBoundsException DateOBExcept)
      {
        System.out.println(DateOBExcept.getMessage() + "\n");
      }
    }
  // Program continues . . .
  }
}
```

If the `new` statement executes without any trouble, meaning the `SafeDate` constructor did not throw an exception, then the `DateOK` variable is set to `true`, the date is output, and the *while* loop terminates. However, if the `DateOutOfBounds` exception is thrown by the `Date` constructor, the latter two statements in the *try* clause are skipped and the exception is caught by the `catch` statement. This, in turn, prints the message from the exception and the *while* loop is executed, again prompting the user for a date. The program repeatedly prompts for date information until it is given a legal date. Notice that the `main` method no longer throws `DateOutOfBoundsException`, as it handles the exception itself.

One last important note about exceptions. The `java.lang.Run-TimeException` class is treated uniquely by the Java environment. Exceptions of this class are thrown when a standard run-time program error occurs. Examples of run-time errors include null-pointer-exception and array-index-out-of-bounds. Because run-time exceptions can happen in virtually any method or segment of code, we are not required to explicitly handle these exceptions. Otherwise, our programs would become unreadable because of so many `try`, `catch`, and `throw` statements. These errors are classified as **unchecked exceptions**.

**Java Note**

Java "Run-Time Exceptions" do not need to be handled explicitly. If we elect not to handle them and they are raised, they will eventually be thrown out to the Java Interpreter and our program will "bomb."

## 1.4 Data Structures

You are already familiar with various ways of organizing data. When you look up a course description in a catalog or a word in a dictionary, you are using an ordered list of words. When you take a number at a delicatessen or barbershop, you become part of a line/queue of people awaiting service. When you study the pairings in a sports tournament and try to predict which team or player will advance through all the rounds and become champion, you create a treelike list of predicted results.

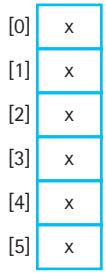
Just as we use many approaches to organize data to deal with everyday problems, programmers use a wide variety of approaches to organize data when solving problems using computers. When programming, the way you view and structure the data that your programs manipulate greatly influences your success. A language's set of primitive types (Java's are `byte`, `char`, `short`, `int`, `long`, `float`, `double`, and `boolean`) can be very useful if we need a counter, a sum, or an index in a program. Generally, however, we must also deal with large amounts of data that have complex interrelationships.

Computer scientists have devised many organizational structures to represent data relationships. These structures act as a unifying theme for this text. In this section we introduce the topic in an informal way, by briefly describing some of the classic approaches.

## Implementation-Dependent Structures

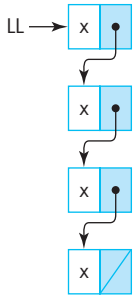
The internal representation of the first two structures is an inherent part of their definition. These structures act as building blocks for many of the other structures.

### Array




You have studied and used arrays in your previous work. An array's components are accessed by using their positions in the structure. Arrays are one of the most important organizational structures. They are available as a basic language construct in most high-level programming languages. Additionally, they are one of the basic building blocks for implementing other structures. We look at arrays more closely in Section 1.5, "Basic Structuring Mechanisms."

### Linked List




A linked list is a collection of separate elements, with each element linked to the one that follows it in the list. We can think of a linked list as a chain of elements. The linked list is a versatile, powerful, basic implementation structure and, like the array, it is one of the primary building blocks for the more complicated structures. Teaching you how to work with links and linked lists is one of the important goals of this text. We look at Java's link

mechanism, the reference, in Section 1.5, "Basic Structuring Mechanisms." Additionally, throughout the rest of the text we study how to use links and linked lists to implement other structures.

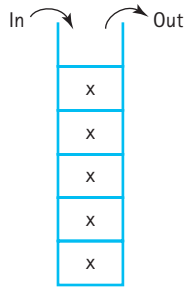
## Implementation-Independent Structures

Unlike the array and the linked list, the organizational structures presented in this subsection are not tied to a particular implementation approach. They are more abstract.

The structures presented here display different kinds of relationships among their constituent elements. For stacks and queues, the organization is based on when the elements were placed into the structure; for sorted lists, maps, and priority queues it is related to the values of the elements; and for trees and graphs, it reflects some feature of the problem domain that is captured in the relative positions of the elements.

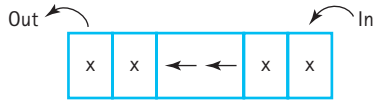
These structures (and others) are treated separately later in the text, when we describe them in more detail, investigate ways of using them, and look at several possible implementations.

### Stack




The defining feature of a stack is that whenever you access or remove an element, you work with the element that was most recently inserted. Stacks are "last in, first out" (LIFO) structures. To see how they work, think about a stack of dishes or trays. Note that the concept of a stack is completely defined by the relationship between its accessing operations, the operations for inserting something into it or removing something from it. No matter what the internal representation is, as long as the LIFO relationship holds, it is a stack.

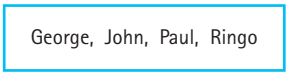
### Queue




Queues are, in one sense, the opposite of stacks. They are "first in, first out" (FIFO) structures. The defining feature of a queue is that whenever you access or remove an element

from a queue, you work with the element that was in the queue for the longest time. Think about an orderly line of people waiting to board a bus or a group of people, holding onto their service numbers, at a delicatessen. In both cases, the people will be served in the order in which they arrived. In fact, this is a good example of how the abstract organizational construct, the queue, can have more than one implementation approach—an orderly line or service numbers.

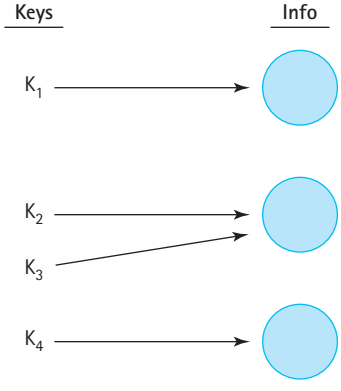
## Sorted List




The elements of a sorted list display a linear relationship. Each element (except the first) has a predecessor, and each element (except the last) has a successor. In a sorted list, the relationship also reflects an ordering of the elements, from "smallest" to "largest," or vice versa.

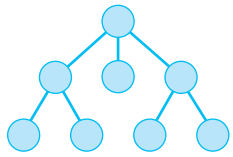
You might be thinking that an array whose elements are sorted is a sorted list—and you would be correct! As we said earlier, arrays are one of the basic building blocks for constructing other structures. But that is not the only way to implement a sorted list. We will cover several other approaches.

## Map




Maps, also known as dictionaries, tables, or associative arrays, are used to store "key"-"info" ordered pairs. Maps provide quick access to desired information when you provide an appropriate key. Consider, for example, when you enter a bank and provide a teller with your account number—within a few seconds (hopefully) the teller has access to your account information. Your account number is the "key"—it "maps" onto your account information. Although there are many ways to implement a map structure, they all must follow the same simple rules: keys are unique and a key maps onto a single information node.

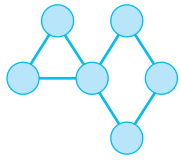
### Tree



Trees and graphs are nonlinear. Each element of a tree is capable of having many successor elements, called its *children*. A child element can have only one *parent*. Thus, a tree is a branching structure. Every tree has a special beginning element called the *root*. The root is the only element that does not have a parent.

Trees are useful for representing hierarchical relationships among data elements. For example, they can be used to classify the members of the animal kingdom or to organize a set of tasks into subtasks. Trees can even be used to reflect the *is-a* relationship among Java classes, as defined by the Java inheritance mechanism.

### Graph



A graph is made up of a set of elements, usually called *nodes* or *vertices*, and a set of *edges* that connect the vertices. Unlike with trees, there are no restrictions on the connections between the elements. Typically, the connections, or edges, describe relationships among the vertices. In some cases, values, also called weights, are associated with the edges to represent some feature of the relationship. For example, the vertices may represent cities and the edges may represent pairs of cities that are connected by airplane routes. Values of the edges could represent the distances or travel times between cities.

## What Is a Data Structure?

We divided our examples of structures into implementation-dependent and implementation-independent categories. Originally, in the infancy of computing, such a distinction was not made. Most of the emphasis on the study of structures at that time dealt with their implementation. The term "data structure" was associated with the details of coding lists, stacks, trees, and so on. As our approaches to problem solving have evolved, we have recognized the importance of separating our study of such structures into both abstract and implementation levels.

As is true for many terms in the discipline of computing, you can find varied uses of the term "data structure" throughout the literature. One approach is to say that a data structure is the implementation of organized data. With this approach, of the structures described

in this section, only the implementation-dependent structures, the array and the linked list, are considered data structures. Another approach is to consider any view of organizing data as a data structure. With this second approach, the implementation-independent structures, such as the stack and the graph, are also considered data structures.

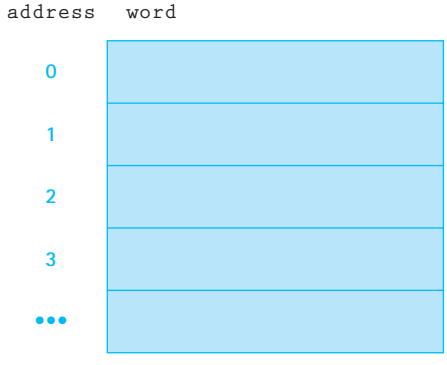
No matter how you label them, all of the structures described here are important tools for solving problems with programs. In this text we will explore all of these data structures, plus many additional structures, from several perspectives. When you are presented with a problem and are devising a computational solution, it is important to decide how you will store, access, and manipulate the information associated with the problem at an early stage of the solution design process. Knowledge of data structures allows you to successfully make and carry out this decision.

# 1.5 Basic Structuring Mechanisms

All of the structures described in Section 1.4 "Data Structures" can be implemented using some combination of two basic mechanisms, the reference and the array. Most general-purpose high-level languages provide these two mechanisms. In this section we review Java's versions of them. In Chapter 2 we will begin to use references and arrays to build structures.

## Memory

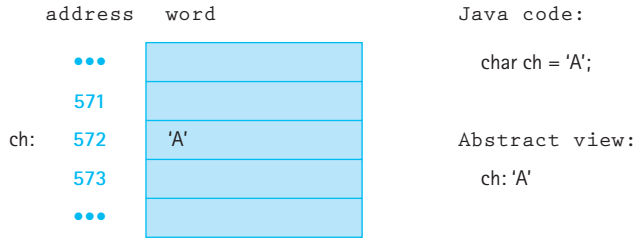
All programs and data are held in memory. Although memory is buried under layers of system software that hides it from us and manages it for us, at its most basic level memory consists of a contiguous sequence of addressable words:




A variable in our program corresponds to a memory location. The compiler handles the translation so that every time the code references the same variable, the system uses the same memory location.

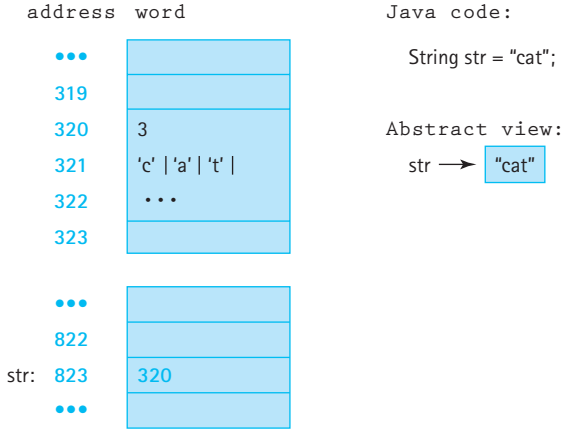
When doing low-level programming, assembly level or lower, there are typically many different addressing "modes" that can be used. However, the two most basic approaches are direct addressing and indirect addressing.

With *direct addressing* the memory location associated with the variable holds the value of the variable. This corresponds to how **primitive variables** are used in Java. For example, if the `char` variable `ch` holds the value 'A' and is associated with memory location **572**, it can be pictured as:




On the left, we show how things are implemented in memory—to clarify the figure we include the variable name `ch` beside its associated memory location. On the right, we show the Java code that declares and instantiates the variable, as well as how we model the variable and its contents in our abstract view of memory.

With *indirect addressing* the memory location associated with the variable holds the *address* of the location that holds the value of the variable. This corresponds to how **reference variables** are used in Java. For example, if the `String` object `str` holds the value "cat" and is associated with memory location **823**, with the actual object stored beginning at memory location **320**, it can be pictured as:




The variable `str` corresponds to location **823**, which holds the address of the location where the information about the `String` object begins—that location, location **320**, is where the system stores information about the string including the string length, the characters, and more—for example, a link to the `String` class. Note that the `String` variable, like all reference variables, is held in a single word (at address **823**) whereas the string itself requires several words. In our abstract view we represent the former location with

the variable name "str" and the latter location with the arrow. Throughout the text we will use arrows to represent references—in actuality they represent memory locations.

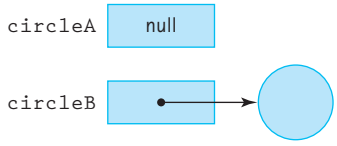
## References

To help present the concepts of this section, we assume access to a `Circle` class. The `Circle` class defines circular objects of different diameters. It provides a constructor that accepts an integer value that represents the diameter of the circle. The `Circle` class provides a convenient example, allowing us to graphically represent objects in our figures—we simply use actual circles of various diameters to represent the `Circle` objects.

Variables of an object class hold references to objects—they use indirect addressing. Consider the effects of the following Java statements:

```
Circle circleA;
Circle circleB = new Circle(8);
```

The first statement reserves memory space for a variable of class `Circle`. The second statement does the same thing, but also creates an object of class `Circle` and places a reference to that object in the `circleB` variable.




The reference is indicated by an arrow, but the reference is actually a memory address, as discussed in the previous subsection. References are sometimes referred to as *links*, *addresses*, or *pointers*. The memory address of the `Circle` object is stored in the memory location assigned to the `circleB` variable. Note how we are representing the `Circle` object with an actual circle. In reality, it would consist of a section of memory allocated to the object.

Because no object has been instantiated or assigned to the `circleA` variable, its memory location holds a `null` reference. Java uses the reserved word `null` to indicate an "absence of reference." If a reference variable is declared without being assigned an instantiated object, it is automatically initialized to whatever the system uses to represent the value `null`. You can also explicitly assign `null` to a variable:

```
circleB = null;
```

In addition, you can use `null` in a comparison:

```
if (circleA == null)
   System.out.println("The Circle does not exist");
```

### Reference Types Versus Primitive Types

It is important to understand the differences in how primitive and nonprimitive types are handled in Java. Primitive types, such as the `int` type, are handled "by value."

Nonprimitive types, such as arrays and classes, are handled "by reference." Whereas the variable of a primitive type holds the value of the variable, the variable of a nonprimitive type holds a *reference* to the value of the variable. That is, the variable holds the address where the system can find the value associated with the variable.

**Java Note**

In Java, variables of a primitive type such as `int` or `char` are stored using direct addressing. We say they are stored "by value." Variables of a reference type, such as type `Circle`, are stored using indirect addressing. We say they are stored "by reference."

The difference in how "by value" and "by reference" variables are handled is seen dramatically in the result of a simple assignment statement. **Figure 1.4** shows the result of the assignment of one `int` variable to another `int` variable, and the result of the assignment of one `Circle` variable to another `Circle` variable.

## Aliases

When we assign a variable of a primitive type to another variable of the same type, the latter becomes a copy of the former. After the integer assignment statement in Figure 1.4 both `intA` and `intB` contain the value 10.

Although the same occurs for reference variables, that is, a value is copied, when we assign a variable of a reference type to another variable of the same type, the effect is quite different. Because the value being copied is a reference in this case, the result is that both variables now point to the same object. Thus we have two "names" for the same object. In this case, we have an **alias** of the object. Good programmers avoid aliases when possible because they make programs difficult to understand. An object's state can change, even though it appears that the program did not access the object, when the object is accessed through the alias. For example, consider the `IncDate` class that was defined in Section 1.3 "Exceptional Situations." If `date1` and `date2` are aliases for the same `IncDate` object, then the code

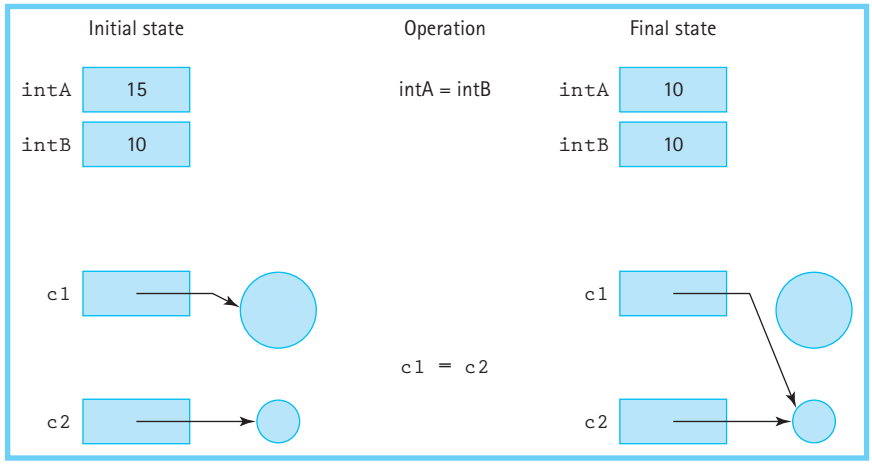



**Figure 1.4** Results of assignment statements

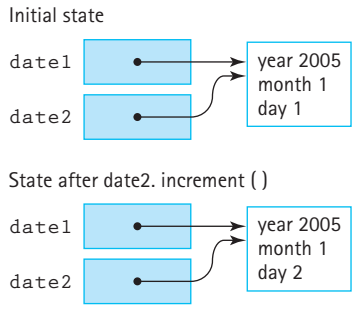



**Figure 1.5** Aliases can be confusing

```
System.out.println(date1);
date2.increment();
System.out.println(date1);
```

would print out two different dates, even though at first glance it would appear that it should print out the same date twice (see **Figure 1.5**). This behavior can be very confusing for a maintenance programmer and lead to hours of frustrating testing and debugging.

## Garbage

It would be fair to ask in the situation depicted in the lower half of Figure 1.4, "What happens to the space being used by the larger circle?" After the assignment statement the program has lost its reference to the large circle, so it can no longer be accessed. This kind of memory space, that has been allocated to a program but can no longer be accessed by a program, is called **garbage**. Garbage can be created in several other ways in a Java program. For example, the following code would create 100 objects of class `Circle`, but only one of them can be accessed through the `c1` variable after the loop finishes executing:

```
Circle c1;
for (n = 1; n <= 100; n++)
{
   Circle c1 = new Circle(n);
   // Code to initialize and use c1 goes here.
}
```

The other 99 objects cannot be reached by the program. They are garbage.

When an object is unreachable, the Java run-time system marks it as garbage. The system regularly performs an operation known as **garbage collection**, in which it identifies unreachable objects and **deallocates** their storage space, returning the space to the free pool for the creation of new objects.

This approach—creating and destroying objects at different points in the application by allocating and deallocating space in the free pool—is called **dynamic memory management**. Without it, the computer would be much more likely to run out of storage space for data.

## Comparing Objects

The fact that nonprimitive types are handled by reference affects the results returned by the == comparison operator. Two variables of a nonprimitive type are considered identical, in terms of the == operator, only if they are aliases for each other. This makes sense when you consider that the system compares the contents of the two variables; that is, it compares the two references that those variables contain. So even if two variables of type `Circle` reference circles with the same diameter, they are not considered equal in terms of the comparison operator. **Figure 1.6** shows the results of using the comparison operator in various situations.

## Parameters

When methods are invoked, they are often passed information (arguments) through parameters. Some programming languages allow the programmer to control whether arguments are passed by value (a copy of the argument's value is used) or by reference (a copy of the argument's address is used). Java does not allow such control. Whenever a variable is passed as an argument, the value stored in that variable is copied into the method's corresponding parameter variable. In other words, all Java arguments are passed by value.

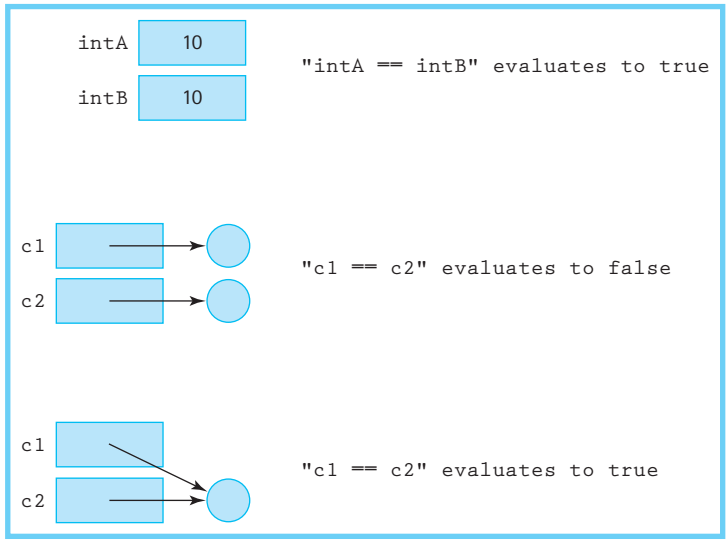



**Figure 1.6** Comparing primitive and nonprimitive variables

Therefore, if the argument is of a primitive type, the actual value (`int`, `double`, etc.) is passed to the method. However, if the argument is a reference type, an object, or an array, then the value passed to the method is the value of the reference—it is the address of the object or the array.

**Java Note**

All java arguments are "passed by value." If the argument is of a primitive type, it represents the value of the primitive. If the argument is of a reference type, then it represents the address of the object.

As a consequence, passing an object variable as an argument causes the receiving method to create an alias of the object. If the method uses the alias to make changes to the object, then when the method finishes, an access via the original variable finds the object in its modified state.

## Arrays

The second basic structuring construct is the array. An array allows the programmer to access a sequence of locations using an indexed approach. We assume you are already familiar with the basic use of arrays from your previous work. In this subsection we review some of the subtle aspects of using arrays in Java.

Arrays in Java are a nonprimitive type and, therefore, are handled by reference, just like objects. Thus they need to be treated carefully, just like objects, in terms of aliases, comparison, and their use as arguments. And like objects, in addition to being declared, arrays must be instantiated. At instantiation you specify how large the array will be:

```
numbers = new int[10];
```

As with objects, you can both declare and instantiate arrays with a single command:

```
int[] numbers = new int[10];
```

Let us discuss a few questions you may have about arrays in Java:

- What are the initial values in an array instantiated by using `new`? If the array components are primitive types, they are set to their default value. If the array components are reference types, such as arrays or classes, the components are set to `null`.
- Can you provide initial values for an array? Yes. An alternative way to create an array is with an initializer list. For example, the following line of code declares, instantiates, and initializes the array `numbers`:

  ```
  int numbers[] = {5, 32, -23, 57, 1, 0, 27, 13, 32, 32};
  ```

- What happens if we try to execute the statement

  ```
  numbers[n] = value;
  ```

  when *n* is less than 0 or when *n* is greater than 9? A memory location outside the array would be indicated, which causes an out-of-bounds exception. Some languages—C++, for instance—do not check for this error, but Java does. If your program attempts to use an index that is not within the bounds of the array, an `ArrayIndexOutOfBoundsException` is thrown.

In addition to component selection, one other "operation" is available for our arrays. In Java, each array that is instantiated has a public instance variable of type `int`, called `length`, associated with it that contains the number of components in the array. You access this variable using the same syntax you use to invoke object methods—you use the name of the object followed by a period, followed by the name of the instance variable. For the `numbers` example, the expression "`numbers.length`" would have the value 10.

## Arrays of Objects

Although arrays with primitive-type components are very common, many applications require a collection of objects. In such a case we can simply define an array whose components are objects.

Here we define an array of `Circle` objects. Declaring and creating the array of objects is exactly like declaring and creating an array where the components are primitive types:

```
Circle[] allCircles = new Circle[10];
```

This means `allCircles` is an array that can hold 10 references to `Circle` objects. What are the diameters of the circles? We do not know yet. The array of circles has been instantiated, but the `Circle` objects themselves have not. Another way of saying this is that `allCircles` is an array of references to `Circle` objects, that are set to `null` when the array is instantiated. The objects must be instantiated separately. The following code segment initializes the first and second circles. We will assume that a `Circle` object `myCircle` has already been instantiated and initialized to have a diameter of 8.

```
Circle[] allCircles = new Circle[10];
allCircles[0] = myCircle;
allCircles[1] = new Circle(4);
```

**Figure 1.7** provides a visual representation of the array.

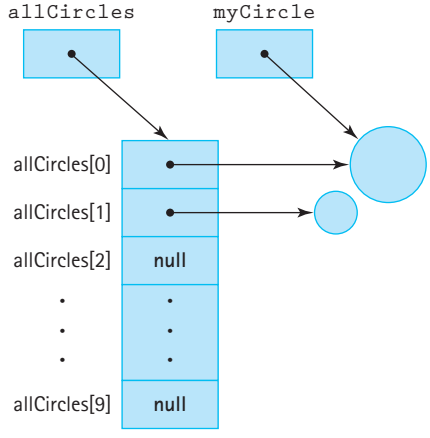



**Figure 1.7** The `allCircles` array

## Generating Images

The `BufferedImage` class in the Java Library allows us to create and manipulate images using a two-dimensional model. It supports most of the popular image types. In this feature we see how to generate JPEG images using this class. Consider the following program:

```
//********************************************************************
//
//  ImageGen01.java          By Dale/Joyce/Weems                Chapter 1
//
//  Demonstrates image generation
//
//********************************************************************
package ch01.apps;

import java.awt.image.*;
import java.awt.Color;
import java.io.*;
import javax.imageio.*;

public class ImageGen01
{
   public static void main (String[] args) throws IOException
   {
      String fileOut = args[0];   // destination file

      // create BufferedImage of SIZE and TYPE
      final int SIDE = 1024;
      final int TYPE = BufferedImage.TYPE_INT_RGB;
      BufferedImage image = new BufferedImage(SIDE, SIDE, TYPE);

      final int LIMIT = 255; // limit of RGB values
      int c;                 // specific value for R G and B
      Color color;

      for (int i = 0; i < SIDE; i++)
        for (int j = 0; j < SIDE; j++)
        {
           c = (i + j) % LIMIT;
           color =  new Color(c, c, c);  // creates 'gray' values
           image.setRGB(i, j, color.getRGB());  // saves pixel
        }

      File outputfile = new File(fileOut);
      ImageIO.write(image, "jpg", outputfile);
   }
}
```

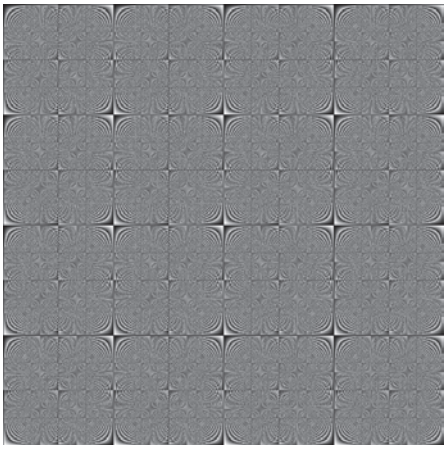
The ImageGen01 application is in the ch01.apps package. It uses a run-time argument as the name of its output file. It is best to use a standard JPEG file extension within this name, for example test.jpg. The program instantiates a BufferedImage object image of size 1024 × 1024 and of type RGB. Images of this type consist of pixels (picture elements) that use a red-green-blue model. The values for red, green, and blue can range from 0 to 255. Individual pixels can be set using the setRGB method, for example

```
color = new Color(200, 20, 125);
image.setRGB(10, 20, color.getRGB);
```

sets the pixel in the 10th row and 20th column to a pinkish purplish color. In the above snippet of code we first create a Color object with a red value of 200, a green value of 20, and a blue value of 125. The method getRGB invoked on that Color object returns a single int value that represents the corresponding color. It is that value that is used by the setRGB method to set the value of the pixel.

To create "black and white" images for our textbook we use the fact that within the RGB color model, colors with identical red, green, and blue values are "gray". For example (0, 0, 0) represents black, (255, 255, 255) represents white and (127, 127, 127) represents a medium gray. The double for-loop in the ImageGen01 program walks through the entire image, from top left to bottom right. The loop body generates an int value c based on the expression (i + j) % LIMIT. The corresponding Color object, which is set to an RGB value of (c, c, c) will cycle through grey values from black to white. The resulting image is shown in **Figure 1.8(a)** below. By varying the expression used for the value of c, alternate images can be generated. It is not difficult to generate interesting images using this approach. For example Figure 1.8 (b) shows the image resulting from the expression (i * j) % LIMIT.

(a) Using (i + j) % LIMIT



(b) Using (i * j) % LIMIT

**Figure 1.8** Generated images

## Two-Dimensional Arrays

A one-dimensional array is used to represent elements in a list or a sequence of values. A two-dimensional array is used to represent elements in a table with rows and columns. Two dimensional arrays are useful when we need to store multiple pieces of information about multiple elements. They can also be used to represent images (see the Feature: Generating Images).

**Figure 1.9** shows a two-dimensional array with 100 rows and 9 columns. The rows are accessed by an integer ranging from 0 through 99; the columns are accessed by an integer ranging from 0 through 8. Each component is accessed by a row—column pair—for example, [0][5].

A two-dimensional array variable is declared in exactly the same way as a one-dimensional array variable, except that there are two pairs of brackets. A two-dimensional array object is instantiated in exactly the same way, except that sizes must be specified for two dimensions.

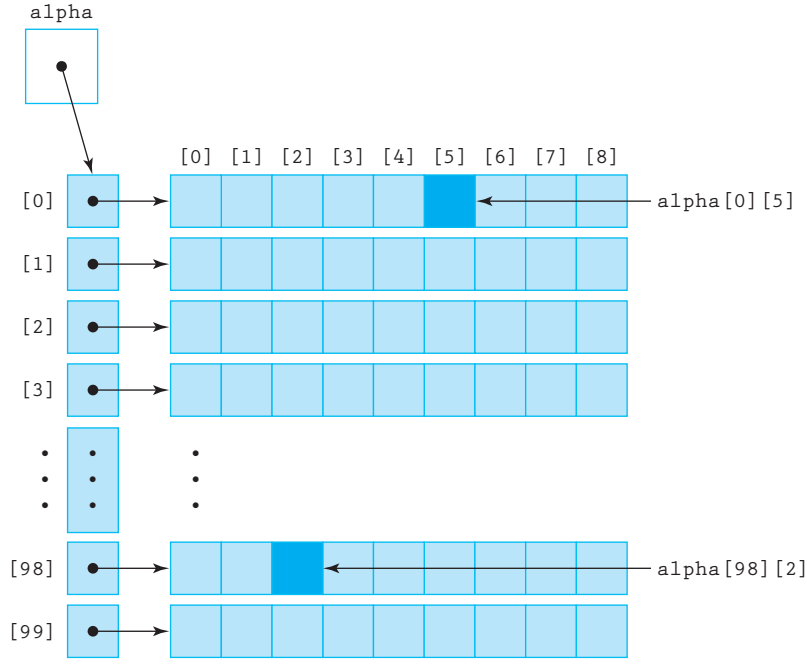


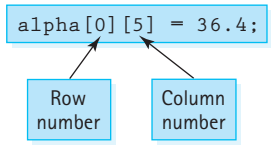

**Figure 1.9** Java implementation of the `alpha` array

The following code fragment would create the array shown in Figure 1.8, where the data in the table are of type `double`.

```
double[][] alpha;
alpha = new double[100][9];
```

The first dimension specifies the number of rows, and the second dimension specifies the number of columns.

To access an individual component of the `alpha` array, two expressions (one for each dimension) are used to specify its position. We place each expression in its own pair of brackets next to the name of the array:




Note that `alpha.length` would give the number of rows in the array. To obtain the number of columns in a row of an array, we access the `length` field for the specific row. For example, the statement

```
rowLength = alpha[30].length;
```

stores the length of row 30 of the array `alpha`, which is 9, into the `int` variable `rowLength`.

It is not difficult to imagine many ways that a two-dimensional array can be used—rows could represent students and columns could be test grades, rows could represent employees and columns the hours they work each day, and so on.

Remember that in Java each row of a two-dimensional array is itself a one-dimensional array. Many programming languages directly support two-dimensional arrays; Java doesn't. In Java, a two-dimensional array is an array of references to array objects. If higher dimension arrays are required we simply extend the number of levels of arrays used, so for example, a three-dimensional array can be created as an two-dimensional array whose elements are arrays.

## 1.6 Comparing Algorithms: Order of Growth Analysis

Alice: "I'm thinking of a number between 1 and 1,000."

Bob: "Is it 1?"

Alice: "No . . . it's higher."

Bob: "Is it 2?"

Alice: "No . . . it's higher."

Bob: "Is it 3?"

Alice: rolls her eyes . . .

Eventually, Bob will guess the secret number by incrementing his guess by 1 each time. Despite Alice's obvious frustration with him, he is following a valid **algorithm** known as **sequential search**.

The analysis of algorithms is an important area of theoretical computer science. In this section we introduce you to this topic to an extent that will allow you to determine which of two algorithms requires fewer resources to accomplish a particular task. The efficiency of algorithms and the code that implements them can be studied in terms of both time (how fast it runs) and space (the amount of memory required). When appropriate throughout this text we point out space considerations, but usually we concentrate on the time aspect—how fast the algorithm is, as opposed to how much space it uses.

Before continuing with a discussion of the time efficiency of algorithms we should point out that quite often time efficiency and space efficiency are interrelated, and trade-offs between time and space efficiency can be made. Consider, for example, the problem of sorting a deck of cards numbered 1–300. Suppose you are sitting on a bus with these cards and have to sort them while holding them in your hands. You will spend a lot of time shuffling through the cards, and you will most likely need to look at each card many times. Alternately, imagine trying to sort the same set of cards if you are standing in front of a table large enough to hold all 300 of them. In this situation you can look at each card just once and place it in its correct spot on the table. The extra space afforded by the table allows for a more time-efficient sorting algorithm.

## Measuring an Algorithm's Time Efficiency

How do programmers compare the time efficiency of two algorithms? The first approach that comes to mind is simply to code the algorithms and then compare the execution times after running the two programs. The one with the shorter execution time is clearly the better algorithm. Or is it? Using this technique, we really can determine only that program A is more efficient than program B on a particular computer at a particular time using a particular set of input data. Execution times are specific to a particular computer, because different computers run at different speeds. Sometimes they are dependent on what else the computer is doing in the background. For example, if the Java run-time engine is performing garbage collection, it can affect the execution time of the program. Coding style and input conditions can also effect the time of a running program. We need a better approach.

A standard technique, and the one we use in this text, is to isolate a particular operation fundamental to the algorithm and count the number of times that this operation is performed. When selecting which operation to count, we want to be sure to select an operation that is executed at least as many times as any other operation during the course of the algorithm.

Consider, for example, Bob's use of the sequential search algorithm to guess Amy's secret number for the Hi-Lo game.

**Hi-Lo Sequential Search**

```
Set guess to 0
do
  Increment guess by 1
  Announce guess
  while (guess is not correct)
```

It is clear that "Announce guess" is a fundamental operation for the Hi-Lo Sequential Search algorithm. It is found inside the loop so it executes over and over again, and it is directly related to the goal of discovering the hidden number.

So, how many times is "Announce guess" executed? How many guesses does Bob make?

## Complexity Cases

If Bob is lucky, Alice is thinking of a low number and he will not need to make many guesses. On the other hand, if he is unlucky he will be guessing for a long time, for example, if Alice is thinking of the number 998.

Clearly, the number of "guesses" required by the Hi-Lo Sequential Search algorithm depends upon the input conditions. This is not unusual. To handle this situation, analysts define three complexity cases:

- **Best case complexity** tells us the complexity when we are very lucky. It represents the fewest number of steps that an algorithm can take. For Alice's guessing game, the best case occurs when she is thinking of the number 1 and Bob only needs to make one guess. In general, best case complexity is not very useful as a complexity measure. We would not want to choose an algorithm due to its best case complexity and then hope we get lucky in terms of the input conditions.
- **Average case complexity** represents the average number of steps required, considering all possible inputs. In the guessing game case this is not difficult to determine: if all of the numbers between 1 and 1,000 are equally likely to occur, then on average it will require $(1 + 1{,}000)\ /2 = 500.5$ guesses to guess a number. Average case complexity analysis can be useful but it is often difficult to define for a specific algorithm.
- **Worst case complexity** represents the highest number of steps that an algorithm would require. If Alice is thinking of the number 1,000 then Bob will need to make 1,000 guesses. With his approach he would never need to make

more than 1,000 guesses. For our purposes we will usually use worst case analysis. It is typically easier to define and calculate than the average case and it gives us useful information. If we know that we can afford the amount of work required in the worst case then we can confidently use the algorithm under review.

We conclude that in the worst case the Hi-Lo Sequential Search algorithm requires 1,000 guesses. But wait—what if the game is changed slightly?

## Size of Input

Bob: "Is it 366?"

Alice, patiently: "No . . . it's higher."

Bob: "Is it 367?"

Alice: "Yes!"

Bob: "Ha—that was easy."

Alice: "Want to play again?"

Bob: "Sure."

Alice: "OK. I'm thinking of a number between 1 and 1,000,000."

Bob: blinks

If we perform worst case analysis of the Hi-Lo Sequential Search algorithm for this new version of the game, we arrive at a different answer—1,000,000 steps. Clearly, the number of steps required by the algorithm depends on the range of possible numbers. Rather than saying the algorithm requires 1,000 steps under this condition and 1,000,000 steps under that condition we can describe the complexity of the algorithm as a function of the input size. If the game is to guess a number between 1 and $N$, the size of the input is $N$, and for the sequential search algorithm, the worst case number of guesses required is also $N$.

Most algorithms require more work to solve larger problems. For example, clearly it is more difficult to sort a list of 500 numbers than it is to sort a list of 10 numbers. Therefore, it makes sense to speak of an algorithm's efficiency in terms of the input size, and to use that size as a parameter when describing the efficiency of the algorithm. For the problems we address in this text it is usually obvious how to identify the required size parameter although for some interesting complex algorithms this is not the case. Most of the problems in this book involve data structures—stacks, queues, lists, maps, trees, and graphs. Each structure is composed of elements. We develop algorithms to add an element to the structure and to modify or delete an element from the structure. We can describe the work done by these operations in terms of $N$, where $N$ is the number of elements in the structure.

## Comparing Algorithms

Carlos: "What's up?"

Bob: "Alice wants me to guess a number between 1 and 1,000,000. No way."

Carlos: "Hmmm. I'll try. Is it 500,000?"

Alice: "No, it's lower."

Carlos: "Is it 250,000?"

Alice: "No, it's higher."

Carlos: "Is it 375,000?"

. . .

As you can see, Carlos is using a different algorithm than Bob. It is called **binary search** and leverages the fact that Carlos can eliminate half the remaining numbers each time by cleverly choosing a number in the middle of the range. Carlos will need, in the worst case, only 20 guesses to guess a number between 1 and 1,000,000!

**Hi-Lo Binary Search(N)**[4]

```
Set range to 1 . . . N
do
  Set guess to middle of range
  Announce guess
  if (guess was too high)
    Set range to first half of range
  if (guess was too low)
    Set range to second half of range
  while (guess is not correct)
```

What is the worst case complexity of the Hi-Lo Binary Search algorithm? Let us again count how many times, in the worst case, the statement "Announce guess" is executed. Each time an incorrect guess is made, the remaining range of possible numbers is cut in half. So, another way of asking this is "How many times can you reduce $N$ by half, before you get down to 1?" The answer is $\log_2 N$.[5] After $\log_2 N$ guesses all of the numbers except

[4] Code that implements this algorithm is found in the `SelSortAndBinSearch.java` file of the `ch01.apps` package.

[5] Recall that $\log_2 N$ is the power that you raise 2 to, in order to get $N$. For example, $\log_2 8 = 3$ because $2^3 = 8$. But another way of looking at this is to consider that $\log_2 N$ is the number of times you can cut $N$ in half before reaching 1. We can cut 8 in half 3 times: $8 \rightarrow 4 \rightarrow 2 \rightarrow 1$.

**Clever Algorithms**

Devising clever algorithms that efficiently solve problems is an exciting part of computer science. Proving such algorithms are correct, analyzing space/time trade-offs, devising heuristics for special cases, and determining optimal bounds are all key steps in the evolution of our understanding of computation. Such work also has important practical benefits as evidenced by advancements in areas such as genome sequencing, modeling, signal processing, encryption, data compression, and network analysis.

one would have been eliminated so with one last guess you will be correct. Therefore, in the worst case Hi-Lo Binary Search requires $\log_2 N + 1$ guesses as compared to the $N$ guesses of Hi-Lo Sequential Search. Of course, $\log_2 N$ is not always an integer—we can "round down" in the case of a nonintegral result. For an input size of 1,000,000, this equates to 20 guesses for binary search as opposed to 1,000,000 guesses for sequential search.

Obviously, the binary search approach is faster than the sequential search approach. Or is it? Each time a guess is made using the binary search approach, the algorithm must do more calculations than when using sequential search. Determining the middle of the remaining range is more time consuming than just adding 1 to the previous guess. Just by looking at the descriptions of the two algorithms we can see that the sequential search is simpler than the binary search. So which is better?

Let us try, in this one case, to count the operations more carefully. For sequential search the initial guess is set to 0, and then each time a guess is made the value must be both incremented and announced. In the worst case this will require two steps (increment, announce) for each guess plus the one initial step resulting in a total of $2N + 1$ steps. For binary search the algorithm must set the low value and high value of the range, and then each time a guess is made it must add together the low value and high value, divide, round, announce the guess, and adjust the range. It must also make the final guess. In the worst case, this will require five steps for each guess (add, divide, round, announce, and adjust) plus the two initial steps and one final step, resulting in a total of $5 \log_2 N + 3$ steps. The accompanying table compares the counts of our two algorithms for various values of $N$.

| Size | Sequential Search | Binary Search |
|---|---|---|
| $N$ | $2N + 1$ steps | $5 \log_2 N + 3$ steps |
| 2 | 5 | 8 |
| 4 | 9 | 13 |
| 8 | 17 | 18 |
| 16 | 33 | 23 |
| 32 | 65 | 28 |
| 1,024 | 2,049 | 53 |
| 1,000,000 | 2,000,001 | 98 |
| 1,000,000,000 | 2,000,000,001 | 148 |

A study of the table shows that if the size of the problem is 8 or less, fewer steps are required by sequential search than by binary search. And for problem sizes like 16 and 32, the difference in number of steps needed by the two algorithms is not much. On the other hand, as the size of the problem grows, the difference in the number of steps required becomes dramatic, in favor of binary search.

Our example is typical. For many problems we can devise simple "brute force" algorithms that are easy to understand and that perform adequately when the size of the problem is small but as the problem size increases they become prohibitively expensive. If you play the Hi-Lo guessing game where the range of possible numbers is small, go ahead and use Bob's brute force approach—but as that range increases you will be much better off emulating the cleverer Carlos.

In general, we are interested in finding solutions to large problems. If you want to sort a list of three names into alphabetical order, you most likely would not need to consider an automated solution right? But what if it was a list of a million names? The study of algorithms focuses on large problem sizes.

## Order of Growth

We must point out that the counting steps exercise of the previous subsection, although enlightening, is also somewhat futile. It is difficult to count accurately the number of steps required by an algorithm. At what level should you count? The pseudo-code description, the high-level language encoding, the machine language translation? And how to handle the issue that all steps are not created equal, for example, "increment a number" and "divide two numbers" will require different amounts of time.

Besides, the detailed counts do not really give us extra information in terms of comparing algorithms. Consider the following table, identical to the previous one, except it does not use any of the detailed step counting information. Here we simply use an estimate of how many times the fundamental operation occurs, $N$ steps for sequential search and $\log_2 N$ (rounded to closest integer) steps for binary search.

| Size | Sequential Search | Binary Search |
|---|---|---|
| $N$ | $N$ steps | $\log_2 N$ steps |
| 2 | 2 | 1 |
| 4 | 4 | 2 |
| 8 | 8 | 3 |
| 16 | 16 | 4 |
| 32 | 32 | 5 |
| 1,024 | 1,024 | 10 |
| 1,000,000 | 1,000,000 | 20 |
| 1,000,000,000 | 1,000,000,000 | 30 |

From this table we can still conclude that as the size of the problem increases, the binary search vastly outperforms the sequential search. That is the focus of our analysis—determining which algorithm is better for large problems.

Computer scientists take advantage of the fact that what really matters when comparing algorithms is the highest order of the polynomial that represents the number of steps needed. We simply report that order to describe the efficiency. Perhaps the number of steps needed for the sequential search is $f(N) = 2N + 1$, but we say that it is "**Order of Growth** $N$" or even $O(N)$ read as "Oh of $N$" or "Order $N$."[6] Perhaps the number of steps needed for binary search is $f(N) = 5 \log_2 N + 3$, but we say it is $O(\log_2 N)$. Here are some more examples:

$2N^5 + N^2 + 37$ is $O(N^5)$ $\quad 2N^2 \log_2 N + 3N^2$ is $O(N^2 \log_2 N)$ $\quad 1 + N^2 + N^3 + N^4$ is $O(N^4)$

By focusing only on the critical information provided by the order of growth, our analysis is simplified. We are able to simply look at an algorithm like Hi-Lo Sequential Search, recognize that in the worst case the loop will account for every number in the range, and state confidently that the efficiency of the algorithm is $O(N)$. Similarly, with the Hi-Lo Binary Search algorithm, we recognize that half the range is removed from consideration each time through the loop and that therefore the algorithm is $O(\log_2 N)$. There is no need to count operations in detail.

## Selection Sort

Let us analyze one more example using the techniques we developed in this section. Putting an unsorted list of data elements into order—*sorting*—is a very common and useful operation. Entire books have been written about sorting algorithms. Here we look at a relatively simple brute-force algorithm that is somewhat similar to the approach many people use to sort a hand of randomly dealt cards in games such as bridge or poker.

Given an unsorted list of elements, the algorithm scans through the list and finds the smallest element. It then *selects* that element and swaps it with the first element. Next it scans the list again to find the second smallest element, again *selecting* it and swapping it with the second element. As the algorithm repeatedly *selects* the next smallest element and swaps it into its "correct" position, the sorted section of the list grows larger and the unsorted section of the list grows smaller, until eventually the entire list is sorted. For reasons which should be obvious, this algorithm is called the *Selection Sort*.

Before we can analyze the Selection Sort algorithm we need to identify the size of the input. It is easy to see that the larger the list, the more work is required to sort it. So the number of elements in the list is the natural choice for the size of the input for the sorting problem. We will use $N$ to indicate this size. Here is a more formal description of the

[6] Many people read the notation as "Big Oh of N". There is a specific mathematical definition of the concept of "Big Oh" that is related to the order of growth and is used in the analysis of algorithms; however, the way we pursue analysis in this text is more properly called "order of growth" and we will use that term.

algorithm, where our goal is to sort an array `values` of size $N$ (the indices of the array go from 0 to $N - 1$):

**Selection Sort(values[0 . . . N – 1])**

for current going from 0 to $N - 2$
  Set minIndex to index of smallest unsorted element
  Swap the elements at indices current and minIndex

**Figure 1.10** shows the steps taken by the algorithm to sort a five-element array. Each section of the figure represents one iteration of the *for* loop. The first part of a section represents the "find the smallest unsorted element" step. To do that it repeatedly examines the unsorted elements asking if each one is the smallest seen so far. The second part of a

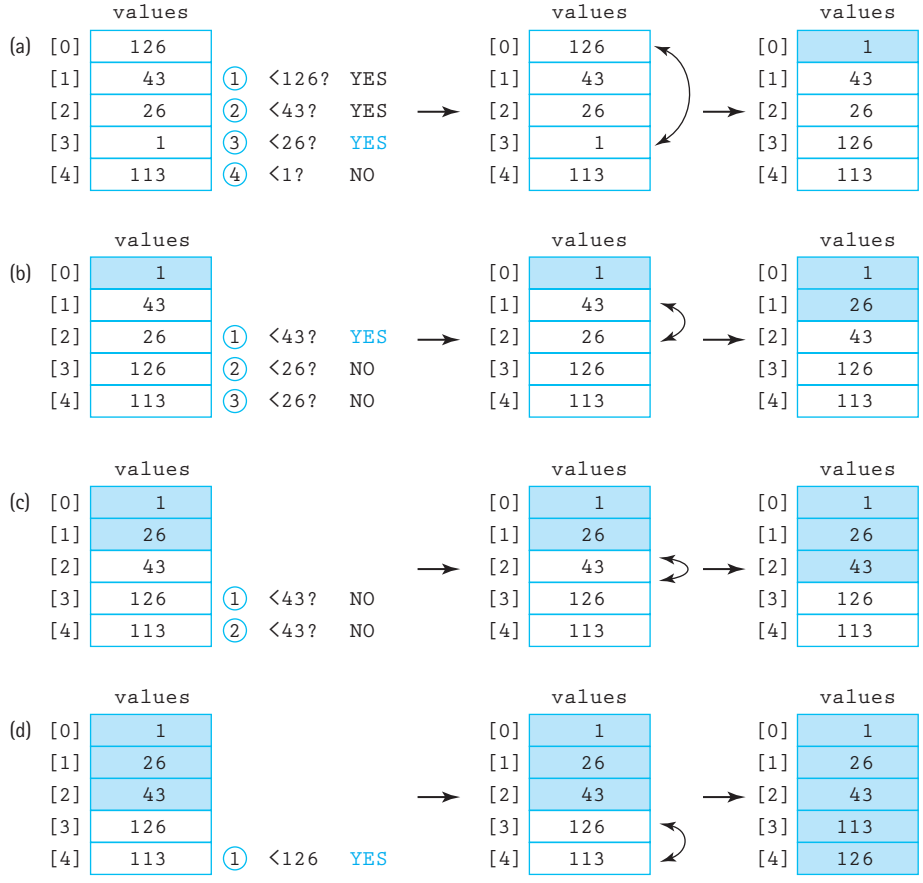



**Figure 1.10** Example of a Selection Sort (sorted elements are shaded)

section shows the two-array elements to be swapped and the final part shows the result of the swap.

During the progression, we can view the array as being divided into a sorted part and an unsorted part. Each time it performs the body of the *for* loop, the sorted part grows by one element and the unsorted part shrinks by one element. Except at the very last step the sorted part grows by two elements—do you see why? When all the array elements except the last one are in their correct locations, the last one is in its correct location also, by default. This is why our *for* loop can stop at index $N - 2$, instead of at the end of the array, index $N - 1$.

We must be careful in identifying the "operation fundamental to the algorithm" to use in our analysis. Can we use "Swap the elements"? Although it appears to be a fundamental operation and is executed once for each iteration of the *for* loop, it is not the operation that is executed the most. Consider that in the act of finding the smallest element in the remaining part of the array each time through the loop we must "look at" all the remaining elements. If we add more detail to our algorithm, we see that we actually have a loop inside a loop:

**Selection Sort(values[0 . . . *N* – 1])**[7]

```
for current going from 0 to N – 2
  Set minIndex to current
  for check going from (current + 1) to (N – 1)
    if (values[check] < values[minIndex)
      Set minIndex to check
  Swap the elements at indices current and minIndex
```

Clearly, the innermost operation, the comparison of the two-array elements, is the fundamental operation that occurs most frequently. This is also evident from a study of Figure 1.10, where we can count 10 comparisons that occur in contrast to only four swaps. We describe the number of comparisons as a function of the number of elements in the array, that is, $N$.

The comparison operation is in the inner loop. We know that this loop is executed $N - 1$ times because the outer loop goes from 0 to $N - 2$. Within the inner loop, the number of comparisons varies, depending on the value of current. The first time the inner loop is executed, current is 0 so the algorithm checks locations 1 to $N - 1$, so there are $N - 1$ comparisons; the next time the current is 1 so there are $N - 2$ comparisons, and so on, until in the last call, there is only one comparison. The total number of comparisons is

$$(N - 1) + (N - 2) + (N - 3) + \ldots + 2 + 1$$

Applying a well-known summation formula tells us this sum is equal to $N(N - 1)/2$. To accomplish our goal of sorting an array of $N$ elements, the selection sort requires

[7] Code that implements this algorithm is found in the `SelSortAndBinSearch.java` file of the `ch01.apps` package.

$N(N-1)/2$ comparisons. The particular arrangement of values in the array does not affect the amount of work done at all. Even if the array is in sorted order before using Selection Sort, the algorithm still makes $N(N-1)/2$ comparisons. Best case, average case, and worst case all require $N(N-1)/2$ comparisons.

How do we describe this algorithm in terms of order of growth? If we expand $N(N-1)/2$ as $\frac{1}{2}N^2 - \frac{1}{2}N$, it is easy to see. In order of growth notation we only consider the term "$\frac{1}{2}N^2$, " because it increases fastest relative to $N$. Further, we ignore the constant coefficient, ½, making this algorithm $O(N^2)$. This means that, for large values of $N$, the computation time is approximately proportional to $N^2$.

Computer scientists who study and analyze many algorithms reach the point where they can often quickly determine the order of growth of an algorithm. For example, they could look at the Selection Sort algorithm described above and they would immediately know it is $O(N^2)$ because they have seen that pattern—a loop inside a loop with conditions interrelated in the same way—many times before. The exercises for this section hopefully will help you reach that level of expertise!

## Common Orders of Growth

In this subsection we discuss some common orders of growth, listed from most efficient to least efficient.

$O(1)$ is called "bounded time." The amount of work is bounded by a constant and is not dependent on the size of the problem. Initializing a sum to 0 is $O(1)$. Although bounded time is often called constant time, the amount of work is not necessarily constant. It is, however, bounded by a constant.

$O(\log_2 N)$ is called "logarithmic time." The amount of work depends on the logarithm, in base 2, of the size of the problem. Algorithms that successively cut the amount of data to be processed in half at each step, like the binary search algorithm, typically fall into this category. Note that in the world of computing we often just say "log $N$" when we mean $\log_2 N$. The base 2 is assumed.

$O(N)$ is called "linear time." The amount of work is some constant times the size of the problem. Algorithms that work through all the data one time to arrive at a conclusion, like the sequential search algorithm, typically fall into this category.

$O(N \log_2 N)$ is called (for lack of a better term) "$N$ log $N$ time". Algorithms of this type typically involve applying a logarithmic algorithm $N$ times. The better sorting algorithms, such as Quicksort presented in Chapter 11, have $N$ log $N$ complexity.

$O(N^2)$ is called "quadratic time." Algorithms of this type typically involve applying a linear algorithm $N$ times. Most simple sorting algorithms, such as the Selection Sort algorithm, are $O(N^2)$ algorithms.

This pattern of increasingly time complex algorithms continues with $O(N^2 \log_2 N)$, $O(N^3)$, $O(N^3 \log_2 N)$, and so on.

$O(2^N)$ is called "exponential time." These algorithms are extremely costly and require more time for large problems than any of the polynomial time algorithms previously listed. An example of a problem for which the best known solution is exponential is the

**Table 1.3** Comparison of Rates of Growth

| $N$ | $\log_2 N$ | $N \log_2 N$ | $N^2$ | $N^3$ | $2^N$ |
|---|---|---|---|---|---|
| 1 | 0 | 1 | 1 | 1 | 2 |
| 2 | 1 | 2 | 4 | 8 | 4 |
| 4 | 2 | 8 | 16 | 64 | 16 |
| 8 | 3 | 24 | 64 | 512 | 256 |
| 16 | 4 | 64 | 256 | 4,096 | 65,536 |
| 32 | 5 | 160 | 1,024 | 32,768 | 4,294,967,296 |
| 64 | 6 | 384 | 4,096 | 262,144 | approximately 20 billion billion |
| 128 | 7 | 896 | 16,384 | 2,097,152 | It would take a fast computer a trillion billion years to execute this many instructions |
| 256 | 8 | 2,048 | 65,536 | 16,777,216 | Do not ask! |

traveling salesman problem—given a set of cities and a set of roads that connect some of them, plus the lengths of the roads, find a route that visits every city exactly once and minimizes total travel distance.

**Table 1.3** presents the values of various common orders of growth functions for several different values of $N$. As you can see in the table, the differences in the function values become quite dramatic as the size of $N$ increases.

## Summary

This chapter is all about organization.

Object orientation allows developers to organize their solutions around models of reality, accruing benefits of understandability, reusability, and maintainability. The primary construct for creating systems using this approach is the class. Classes are used to create objects that work together to provide solutions to problems. Java's inheritance mechanism and package construct help us organize our classes.

Java's exception handling mechanisms provide a powerful way to organize our system's responses to special situations. We can choose to handle exceptional situations where they are first encountered or to throw the responsibility out to another level. A good understanding of this mechanism is a crucial ingredient for creating safe, reliable systems.

Programs operate on data, so how the data are organized is of prime importance. Data structures deal with this organization. Several classic organizational structures have been identified through the years to help programmers create correct and efficient solutions to problems. The Java language provides basic structuring mechanisms for creating these

structures—namely, the array and the reference mechanisms. Order of growth notation is an approach for classifying the efficiency of the algorithms that we will employ when implementing and using our data structures.

Programmers are problem solvers. Object orientation allows seamless integration of problem analysis and design, resulting in problem solutions that are maintainable and reusable. Data structures provide ways of organizing the data of the problem domain so that solutions are correct and efficient. Staying organized is the key to solving difficult problems!

## Exercises

### 1.1 Classes, Objects, and Applications

1. *Research Question*: The Turing Award has been awarded annually since 1966 to a person or persons for making contributions of lasting and major technical importance to the computer field. Locate information about this award on the Web. Study the list of award winners and their contributions. Identify those winners whose contribution dealt directly with programming. Then identify those winners whose contributions dealt directly with object orientation.
2. *Research Question*: List and briefly describe the UML's 14 main diagramming types.
3. What is the difference between an object and a class? Give some examples.
4. Describe each of the four levels of visibility provided by Java's access modifiers.
5. According to the `DaysBetween` application, how many days are between 1/1/1900 and 1/1/2000? How many leap years are there between those dates? What about between 1/1/2000 and 1/1/2100? Explain the difference in these answers.
6. Use the `DaysBetween` application to answer the following:
   a. How old are you, in days?
   b. How many days has it been since the United States adopted the Declaration of Independence, on July 4, 1776?
   c. How many days between the day that Jean-François Pilâtre de Rozier and François Laurent became the first human pilots, traveling 10 kilometers in a hot air balloon on November 21, 1783, near Paris and the day Neil Armstrong took one small step onto the moon, at the Sea of Tranquility, on July 20, 1969?
7. Think about how you might test the `DaysBetween` application. What type of input should give a result of 0? Of 1? Of 7? Of 365? Of 366? Try out the test cases that you identified.
8. Modify the `Date` class so that it includes a `compareTo` method with signature

```
int compareTo(Date anotherDate)
```

This method should return the value 0 if this date (the date of the object upon which the method is invoked) is equal to the argument date; a value less than 0 if this date is a date earlier than the argument date; and a value greater than 0 if this date is a

date later than the argument date. Create a **test driver** that shows that your method performs correctly.

**9.** A common use of an object is to "keep track" of something. The object is fed data through its transformer methods and returns information through its observer methods. Define a reasonable set of instance variables, class variables, and methods for each of the following classes. Indicate the access level for each construct. Note that each of these class descriptions are somewhat "fuzzy" and allow multiple varied "correct" answers.

**a.** *A time counter*—this will keep track of total time; it will be fed discrete time amounts (in either minutes and seconds or just in seconds); it should provide information about the total time in several "formats," number of discrete time units, and average time per unit. Think of this class as a tool that could be used to keep track of the total time of a collection of music, given the time for each song.

**b.** *Basketball statistics tracker*—this will keep track of the score and shooting statistics for a basketball team (not for each player but for the team as a unit); it should be fed data each time a shot is taken; it should provide information about shooting percentages and total score when requested.

**c.** *Tic-Tac-Toe game tracker*—this will keep track of a tic-tac-toe game; it should be fed moves and return an indication of whether or not a move was legal; it should provide information about the status of the game (is it over? who won?) when requested.

**10.** For one or more of the classes described in the previous exercise

**a.** Implement the class.

**b.** Design and implement an application that uses the class.

**c.** Use your application to help verify the correctness of your class implementation.

**11.** You will create a class that models a standard pair of dice.

**a.** Create a class called `PairOfDice`. Objects of this class represent a single pair of six-sided dice. The only attributes of such an object are the face values of the dice. Provide a constructor. Provide a `roll` method that simulates rolling the dice. Provide a `value` method that returns the sum of the face values of the dice. Provide a `toString` method that returns a nicely formatted string representing the pair of dice, for example "5 : 3 = 8". Finally, create a "test driver" that demonstrates that your `PairOfDice` class performs correctly.

**b.** The game of Craps is played in casinos all over the world. The basic bet made by the "shooter" in this game is the pass-line bet. To start a pass-line round, the shooter makes a "come-out" roll. A come-out roll of 2, 3, or 12 is called "craps" or "crapping out," and the shooter loses. A come-out roll of 7 or 11 is a "natural," and the shooter wins. The other possible numbers are the point numbers: 4, 5, 6, 8, 9, and 10. If the shooter rolls one of these numbers on the come-out roll,

this establishes the "point"—to win, the point number must be rolled again before a seven. So in the case where a "point" is established the shooter rolls over and over until either the point is rolled (a win) or a seven is rolled (a loss). Using your `PairOfDice` class simulate 100,000 pass-line bets and output how many result in a win, and how many result in a loss. *Hint*: Your result should tell you to be wary of casinos.

**12.** You will create a class that keeps track of the total cost, average cost, and number of items in a shopping bag.

**a.** Create a class called `ShoppingBag`. Objects of this class represent a single shopping bag. Attributes of such an object include the number of items in the bag and the total retail cost of those items. Provide a constructor that accepts a tax rate as a `double` argument. Provide a transformer method called `place` that models placing a number of identically priced items into the bag—it accepts an `int` argument indicating the number of items and a `double` argument that indicates the cost of each of the items. For example, `myBag.place(5, 10.5)` represents placing five items that cost $10.50 each into `myBag`. Provide getter methods for both the number of items in the bag and their total retail cost. Provide a `totalCost` method that returns the total cost with tax included. Provide a `toString` method that returns a nicely formatted string that summarizes the current status of the shopping bag. Finally, provide a program, a "test driver," that demonstrates that your `ShoppingBag` class performs correctly.

**b.** Create an application that repeatedly prompts the user for a number of items to put in the bag, followed by a prompt for the cost of those items. Use a 0 for the number of items to indicate that there are no more items. The program then displays a summary of the status of the shopping bag. Assume the tax rate is 6%. A short sample run might look something like this:

```
Enter count (use 0 to stop): 5
Enter cost: 10.50
Enter count (use 0 to stop): 2
Enter cost: 2.07
Enter count (use 0 to stop): 0
The bag contains seven items. The retail cost of the items is $56.64.
The total cost of the items, including tax, is $60.04.
```

**13.** You will create a class that represents a polynomial; for example, it could represent $5x^3 + 2x - 3$ or $x^2 - 1$.

**a.** Create a class called `Polynomial`. Objects of this class represent a single polynomial. Attributes of such an object include its `degree` and the `coefficients` of each of its terms. Provide a constructor that accepts the degree of the polynomial as an `int` argument. Provide a transformer method called `setCoefficient` that accepts as `int` arguments the degree of the term it is setting and the

coefficient to which it should be set. For example, the polynomial $5x^3 + 2x - 3$ could be created by the sequence of statements:

```
Polynomial1 myPoly = new Polynomial1(3);
myPoly.setCoefficient(3,5);
myPoly.setCoefficient(1,2);
myPoly.setCoefficient(0,-3);
```

Provide an `evaluate` method that accepts a `double` argument and returns the value of the polynomial, as a `double`, as evaluated at the argument value. For example, given the previous code the following sequence of code would print –3.0, 4.0, and –1.375.

```
System.out.println(myPoly.evaluate(0.0));
System.out.println(myPoly.evaluate(1.0));
System.out.println(myPoly.evaluate(0.5));
```

Finally, provide a program, a "test driver," that demonstrates that your `Polynomial1` class performs correctly.

**b.** Create an application that accepts the degree of a polynomial and the coefficients of the polynomial, from highest degree to lowest, as a command line argument and then creates the corresponding `Polynomial` object. For example, the polynomial $5x_3 + 2x - 3$ would be represented by the command line argument "3 5 0 2 – 3." The program should then repeatedly prompt the user for a double value at which to evaluate the polynomial and report the result of the evaluation. A sample run, assuming the previously stated command line argument, might look something like this:

```
Enter a value> 0.0
The result is -3.0
Continue?> Yes
Enter a value> 1.0
The result is 4.0
Continue?> Yes
Enter a value> 0.5
The result is -1.375
Continue?> No
```

**c.** Create an application that accepts the degree of a polynomial and the coefficients of the polynomial as a command line argument as in part b. The program should then prompt the user for two `double` values that will represent the end points of an interval on which the polynomial is defined. Your program should then calculate and output the approximation of the definite integral of the polynomial on the indicated interval, using 1,000 bounding rectangles.

## 1.2 Organizing Classes

**14.** Describe the concept of inheritance, and explain how the inheritance tree is traversed to bind method calls with method implementations in an object-oriented system.

**15.** *Research*: Find the Java library description of the `ArrayList` class and answer the following questions:

- **a.** What class does it directly inherit from?
- **b.** How many direct subclasses does it have?
- **c.** How many methods does it implement?
- **d.** How many methods does it inherit?
- **e.** If we invoke the `toString` method on an object of class `ArrayList`, which class's `toString` method will be used?

**16.** Given the definition of the `Date` and `IncDate` classes in this chapter, and the following declarations

```
int temp;
Date date1 = new Date(10,2,1989);
IncDate date2 = new IncDate(12,25,2001);
```

indicate which of the following statements are illegal, and which are legal. Explain your answers.

- **a.** `temp = date1.getDay();`
- **b.** `temp = date2.getYear();`
- **c.** `date1.increment();`
- **d.** `date2.increment();`

**17.** Design a set of at least three classes related by inheritance from the world of

- **a.** *Banking*—for example, account, checking account, savings account
- **b.** *Gaming*—for example, creature, hero, villain, pet
- **c.** *Travel*—for example, vehicle, plane, boat
- **d.** *Whatever*—use your imagination

**18.** Devise a program that demonstrates polymorphism, using the example provided on page 18.

**19.** Explain how packages are used to organize Java files.

**20.** *Research*: Copy the program files to your system and answer the following questions:

- **a.** How many classes are in the `support` package?
- **b.** How many classes are in the `ch01.apps` package?
- **c.** The `CSInfo` class is in the `ch05.apps` package:
  - i. What four packages does it import from?
  - ii. How do the import statements differ?

**d.** The `Dates` class is in the `ch01.dates` class:

i. What happens if you change its package statement from `package ch01.dates` to `package ch01.date` and compile it? Explain.

ii. What happens if you remove its package statement and compile it? Explain.

iii. What happens if you remove its package statement and compile the `DaysBetween` application that is in the `ch01.apps` package? Explain.

**21.** Suppose file 1 contains

```
package media.records;
public class Labels{ . . . }
class Check { . . . }
```

and file 2 contains

```
package media.records;
public class Length{ . . . }
class Review { . . . }
```

**a.** Are the `Check` class and the `Review` class in the same package?

**b.** What is the name of file 1?

**c.** What is the name of file 2?

**d.** What is the name of the directory that contains the two files?

**e.** What is that directory a subdirectory of?

## 1.3 Exceptional Situations

**22.** Explain the difference between a programmer-defined exception that extends the Java `Exception` class and one that extends the Java `RunTimeException` class.

**23.** Create a program that asks users to enter an integer and then thanks them. If they do not enter an integer your program should ask again, until they do. Running your program might result in this sort of console trace:

```
Please enter an integer.
OK
That is not an integer. Please enter an integer.
Twenty-seven
That is not an integer. Please enter an integer.
64
Thank you.
```

**24.** Create a `BankAccount` class that models a typical bank account where you deposit and withdraw money. To keep things simple, assume this bank account deals only with integral amounts.

**a.** You should provide a constructor, a `toString` method, a `getTotal` method that returns an `int`, and both `deposit` and `withdraw` methods that take `int` arguments and return `void`. Also create an application `UseBankAccount` that demonstrates that the `BankAccount` class works correctly.

**b.** Create a `BankAccountException` class. Change your `deposit` method so that it throws an appropriate exception if an attempt is made to deposit a

negative amount. Do the same with the `withdraw` method, but also have it throw an exception if an attempt is made to withdraw more money than is available. In each case include appropriate exception messages. Create three short applications that demonstrate each of the three exceptional situations—it is OK if the applications bombs, as long as it demonstrates that the appropriate exception has been thrown.

**c.** Create a new application, `Banker`, that creates a `BankAccount` object and then interacts with users, allowing them to deposit or withdraw funds, or to request an account total. This application should not bomb in any of the exceptional situations—it should catch the exception, pass the message to the user, and continue processing.

**25.** There are three parts to this exercise:

**a.** Create a "standard" exception class called `ThirteenException`.

**b.** Write a program that repeatedly prompts the user to enter a string. After each string is entered, the program outputs the length of the string, unless the length of the string is 13, in which case the `ThirteenException` is thrown with the message "Use thirteen letter words and stainless steel to protect yourself!" Your `main` method should simply throw the `ThirteenException` exception out to the run-time environment. A sample run of the program might be:

```
Input a string > Villanova University
That string has length 20.
Input a string > Triscadecaphobia
That string has length 16.
Input a string > misprogrammed
```

At this point the program bombs and the system provides some information, including the "Use thirteen letter words and stainless steel to protect yourself!" message.

**c.** Create another program similar to the one you created for part b, except this time, within your code, include a *try-catch* clause so that you catch the exception when it is thrown. If it is thrown, then catch it, print its message, and end the program "normally."

## 1.4 Data Structures

**26.** *Research Question*: On the Web find two distinct definitions of the term "data structure." Compare and contrast them.

**27.** Identify things in the following story that remind you of the various data structures described in the section. Be imaginative. How many can you find? What are they? [*Note*: We can find nine!]

Kaede arrives at the train station with just a few minutes to spare. This weekend is shaping up to be a disaster. She studies the electronic map on the wall

for a few seconds in confusion. She then realizes she just needs to select her destination from the alphabetized list of buttons on the right. When she presses Gloucester a path on the map lights up—so, she should take the Blue train to Birmingham where she can connect to the Red train that will take her to Gloucester. The wait in line to buy her ticket does not take long time. She hurries to the platform and approaches the fourth car of the train. Double-checking that her ticket says "car 4," she boards the train and finds a seat. Whew, just in time, as a few seconds later the train pulls out the station. About an hour into the journey Kaede decides it is time for lunch. She walks through cars 5, 6, and 7, to arrive at car 8, the dining car. She grabs the top tray (it is still warm from the tray dryer) and heads for the candy machine, thinking to herself, "May as well figure out what to have for dessert first, as usual. Hmmm, that's an interesting Pez dispenser in slot F4." She presses the button contentedly, and thinks "Looks like this is going to be a nice weekend after all. Thank goodness for data structures."

**28.** Describe three uses of a tree structure as a way of organizing information.

**29.** Some aspect of each of the following can be modeled with a graph structure. Describe, in each case, what the nodes would represent and what the edges would represent.

- **a.** Trips available using a specific airline
- **b.** Countries and their borders
- **c.** A collection of research articles about data structures
- **d.** Actors (research the "six degrees of Kevin Bacon")
- **e.** The computers at a university
- **f.** A labyrinth
- **g.** The Web

## 1.5 Basic Structuring Mechanisms

**30.** Draw images similar to those shown in the *Memory* subsection of Section 1.5 "Basic Structuring Mechanisms" that represent the contents of memory resulting from the following code segments. Assume that `i` is associated with memory location 123, `j` with 124, the `str1` variable with 135 and its associated object with 100, and the `str2` variable with 136.

**a.**
```
int i = 10;
int j = 20;
String str1 = "cat";
```

**b.**
```
int i = 10;
int j = i;
String str1 = "cat";
String str2 = str1;
```

**31.** What is an alias? Show an example of how it is created by a Java program. Explain the dangers of aliases.

**32.** Assume that `date1` and `date2` are objects of class `IncDate` as defined in Section 1.2 "Organizing Classes." What would be the output of the following code?

```
date1 = new IncDate(5, 5, 2000);
date2 = date1;
System.out.println(date1);
System.out.println(date2);
date1.increment();
System.out.println(date1);
System.out.println(date2);
```

**33.** What is garbage? Show an example of how it is created by a Java program.

**34.** Assume that `date1` and `date2` are objects of class `IncDate` as defined in Section 1.2 "Organizing Classes." What would be the output of the following code?

```
date1 = new IncDate(5, 5, 2000);
date2 = new IncDate(5, 5, 2000);
if (date1 = = date2)
  System.out.println("equal");
else
  System.out.println("not equal");
date1 = date2;
if (date1 = = date2)
  System.out.println("equal");
else
  System.out.println("not equal");
date1.increment();
if (date1 = = date2)
  System.out.println("equal");
else
  System.out.println("not equal");
```

**35.** Write a program that declares a 10-element array of `int`, uses a loop to initialize each element to the value of its index squared, and then uses another loop to print the contents of the array, one integer per line.

**36.** Write a program that declares a 10-element array of `Date`, uses a loop to initialize the elements to December 1 through 10 of 2005, and then uses another loop to print the contents of the array, one date per line.

**37.** Create an application that instantiates a 20 × 20 two-dimensional array of integers, populates it with random integers drawn from the range of 1 to 100, and then outputs the index of the row with the highest sum among all the rows and the index of the column with the highest sum among all the columns.

**38.** Compile and run the `ImageGen01` application. Experiment with alternate formulas for the value of `c`. Add two more `int` variables so that you can separately set the RGB values of `color` and experiment some more. Share your most interesting results with your classmates.

## 1.6 Comparing Algorithms: Order of Growth Analysis

**39.** We examined two approaches to guessing the secret number for the Hi-Lo Guessing Game: Hi-Lo Sequential Search and Hi-Lo Binary Search. What is the best and worst case number of guesses required by each of these approaches if the highest possible number is (a) 10, (b) 1,000, (c) 1,000,000, and (d) 1,000,000,000.

**40.** For each of the following problems briefly describe an algorithm that solves the problem, identify a good "operation fundamental to the algorithm" that could be used to calculate the algorithm's efficiency, succinctly describe the size of the problem, and state the number of times the fundamental operation occurs as a function of the problem size in the best case and worst case. For example, if the problem was "guess the secret number in the Hi-Lo Guessing Game" your answer might be "start at 1 as my first guess and keep adding one to my guess until I guess the number; announce my guess; the highest possible number—call it $N$; best case 1 time, worst case $N$ times".

- **a.** Finding *The Art of Computer Programming* on a shelf of unsorted books.
- **b.** Sorting an array of integers.
- **c.** Finding the cheapest pair of shoes in a shoe catalog.
- **d.** Figuring out how much money is in a piggy bank.
- **e.** Computing $N!$ for a given $N$.
- **f.** Computing the sum of the numbers 1 to $N$, for a given $N$.
- **g.** Multiplying two $N \times N$ matrices.

**41.** Compare each of the following pairs of functions $f(x)$ and $g(x)$ by graphing them on the set of nonnegative numbers (yes, just like in algebra class). Note that in each case the higher ordered function g eventually becomes larger than the lower ordered function f. Identify the x value where this occurs.

- **a.** $f(x) = 3 \log_2 x$ $\quad$ $g(x) = x$
- **b.** $f(x) = 5 \log_2 x + 3$ $\quad$ $g(x) = 2x + 1$
- **c.** $f(x) = 4 x^2$ $\quad$ $g(x) = x^3$
- **d.** $f(x) = 8 x^2$ $\quad$ $g(x) = 2^x$

**42.** Describe the order of growth of each of the following functions using O notation.

- **a.** $N^2 + 3N$
- **b.** $3N^2 + N$
- **c.** $N^5 + 100N^3 + 245$

**d.** $3N\log_2 N + N^2$

**e.** $1 + N + N^2 + N^3 + N^4$

**f.** $(N * (N - 1)) / 2$

**43.** Describe the order of growth of each of the following code sections, using O notation:

**a.**
```
count = 0;
for (i = 1; i <= N; i++)
  count++;
```

**b.**
```
count = 0;
for (i = 1; i <= N; i++)
  for (j = 1; j <= N; j++)
    count++;
```

**c.**
```
value = N;
count = 0;
while (value > 1)
{
  value = value / 2;
  count++;
}
```

**d.**
```
count = 0;
value = N;
value = N * (N - 1);
count = count + value;
```

**e.**
```
count = 0;
for (i = 1; i <= N; i++)
  count++;
for (i = N; i >= 0; i--)
  count++;
```

**f.**
```
count = 0;
for (i = 1; i <=N; i++)
  for (j = 1; j <= 5; j++)
    count++;
```

**44.** The method `Sum` listed below returns the sum of the integers between 1 and n. What is its order of growth? Create a new method that performs the same function that is a lower order of growth.

```
public int Sum (int n)
// Precondition: n is > 0
{
  int total = 0;
  for (int i = 1; i <= n; i++)
    total = total + i;
  return total;
}
```

**45.** Assume that `numbers` is a large array of integers, currently holding *N* values in locations 0 through *N* – 1. Describe the order of growth (worst case) of each of the following operations, using O notation:

- **a.** Set location *N* of `numbers` to 17.
- **b.** Shift all values in the `numbers` array to the "right" one location to make room at location 0 for a new number without disrupting the order of the current values; insert the number 17 into location 0.
- **c.** Randomly choose a location *L* from 0 to *N* – 1; Shift all the values in the `numbers` array, from location *L* to location *N* – 1, to the right one location to make room at location *L* for a new number; insert the number 17 into location *L*.

**46.** Show the sequence of changes the array `values` undergoes while it is sorted using selection sort.

values

| 27 | 15 | 83 | 12 | 104 | 28 | 57 | 30 |
|---|---|---|---|---|---|---|---|

**47.** For this exercise you must implement the selection sort algorithm.

- **a.** Create a program named `SelectionSort` that instantiates an array of `int` of size 100 and initializes it with random numbers between 1 and 1000. The program should display the integers from the array in five columns. Next it sorts the array using selection sort. Finally, it prints the contents of the array again, in columns of 5.
- **b.** Augment your program from part a so that it also counts the number of comparisons and the number of swaps executed during the selection sort. It should report these numbers after printing the sorted array. Based on the analysis of selection sort in this section what are the expected values for the number of comparisons and the number of swaps? How do the values reported by your program compare to the "theoretical" values?
- **c.** Augment your program from part b so that it works first with an array of size 10, then 100, then 1000, then 10,000, and finally 100,000 (remove the code that prints out the array values of course). For each array size your program should display the number of comparisons and swaps. Have the program display these numbers in a nice tabular format.