Chapter 8 Collaborative Classes

After studying this chapter, you should be able to:

- Write a class that uses references to an object by storing them in instance variables, passing them to parameter variables, and using them with temporary variables
- Draw class diagrams depicting collaborating classes
- Explain how reference variables are different from primitive variables
- Explain what an alias is and what dangers arise from aliasing
- Write code that compares two objects for equivalence
- Throw and catch exceptions
- Use a Java collection object to collaborate with many objects, all having the same type

So far, our programs have usually required writing only a single class plus the `main` method. Almost any program of consequence, however, involves at least several classes that work together—or collaborate—to solve the problem. In fact, most of our programs already have this property of collaboration. For example, the `Robot` class collaborates with `City` and `Intersection` objects, and the `Meter` class collaborates with the `GasPumpGUI` that displays it. However, the mechanics of these collaborations have usually been hidden.

In this chapter, we become more intentional about a particular kind of collaboration: when one object has a reference to another object as an instance variable or is passed a reference to another object via a parameter variable. We will also begin to investigate exceptions, and how a class can collaborate with many instances of another class.

Now that we have many programming tools at our disposal, we will move away from the robot examples. The rest of the book uses examples involving a `Person` class, a program for a charitable organization, games, and others.

8.1 Example: Modeling a Person

A `Person` class might be useful in many kinds of programs. Payroll systems, student information systems, airline reservation systems, tax preparation programs, and programs to track genealogies all maintain information about people and might use a `Person` class.

In this section, we will develop a simple `Person` class, first as a single class using the techniques we've seen so far, but then using collaborating classes. Complex concepts can be modeled more easily using collaborating classes because they can divide the work.

Our simple `Person` class will be oriented toward registering births and deaths, perhaps within a government, an insurance company, or a genealogical program. It will model a person's name, mother, father, birth date, and death date. Of course, it will need a constructor and some accessor methods. We'll also be interested in a `daysLived` method. If the person has died, `daysLived` returns the number of days between his birth and death dates. If the person is still alive, it returns the number of days between his birth and the current date.

8.1.1 Using a Single Class

Building on what we have already learned, it's not hard to imagine how a `Person` class could be constructed. A suggested class diagram is shown in Figure 8-1, and an initial test harness is shown in Listing 8-1.

Listing 8-1: *The beginnings of a test harness for the* `Person` *class*

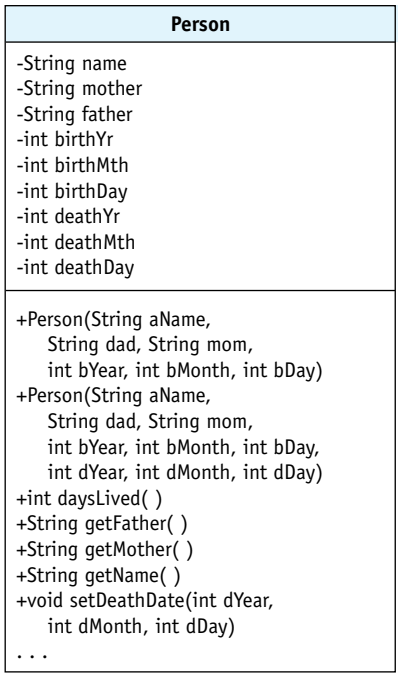
```
1  import becker.util.Test;
2
3  public class Person extends Object
4  {
5     // instance variables and methods omitted
6
7     // Test the class.
8     public static void main(String[] args)
9     { Person p = new Person("Joseph Becker",
10                    "Jacob B. Becker", "Elizabeth Unruh", 1900, 6, 14);
11
12       p.setDeathDate(1901, 6, 14);
13       Test.ckEquals("exactly 1 year", 365, p.daysLived());
14       p.setDeathDate(1901, 6, 13);
15       Test.ckEquals("1 year less a day", 364, p.daysLived());
```

Listing 8-1: *The beginnings of a test harness for the* `Person` *class* (continued)

```
16      p.setDeathDate(1902, 6, 15);
17      Test.ckEquals("2 years plus a day", 365*2+1, p.daysLived());
18    }
19  }
```

(figure 8-1)

Suggested class diagram for the `Person` *class*



All of the methods shown in the class diagram should be easy to write and test with the exception of `daysLived`. The test harness chooses several easy ages to calculate—exactly one year old, a year less one day, and two years plus a day. Many other combinations would be worth testing, but these three make a good start.

After considerable thought, we might come up with pseudocode for `daysLived` that appears to solve the problem:

```
declare variables for end date
if (not dead)
{ set end date to today's date
} else
{ set end date to death date
```

```
}

days = 0
for each full year lived
{ days = days + days in the year (remember leap years!)
}

daysLivedInFirstYear = # days between birth date and Dec 31
daysLivedInLastYear = # days between Jan 1 and end date
return days + daysLivedInFirstYear + daysLivedInLastYear
```

This is a complicated algorithm and some problems haven't been solved yet (finding the number of days between January 1 and a given date, getting today's date, and determining if a year is a leap year). Furthermore, these details are not part of the main purpose of the class: maintaining information about a person. The `Person` class would be easier to write and maintain if the details related to dates were in a separate class.

KEY IDEA

Delegate peripheral details to a separate class.

Using a separate class for dates is also a good idea because working with dates is a common activity. Having a separate class allows us to write and debug the class once but use it in many classes. For these reasons, we should either write our own `Date` class or find one that has already been written. For both of these scenerios, we need to learn to write the `Person` class to make effective use of a date class; this is the primary focus of this chapter.

8.1.2 Using Multiple Classes

In fact, Java provides classes to deal with dates. One is `GregorianCalendar` in the package `java.util`. It is rather complex to use, however. A simpler class is found in `becker.util` and is called `DateTime`. We'll use this class to simplify our implementation of `Person`.

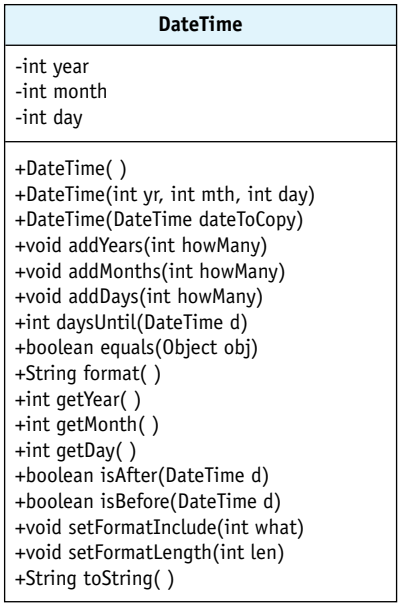
The `DateTime` Class

One possible class diagram for `DateTime` is shown in Figure 8-2. The diagram is abbreviated because, as the name implies, the class also handles time. This aspect has been omitted from the class diagram.

The first constructor in this class creates an object corresponding to the current date, the second constructor allows you to create an object for a specific date, and the third creates a copy of the specified `DateTime` object. The `add` methods allow the date to be adjusted, either forward or backward in time. The `daysUntil` method calculates the number of days between two dates.

(figure 8-2)

Class diagram for `DateTime` *(methods related to time are not shown)*



Listing 8-2 shows a simple program to calculate and print Luke's age, in days. It uses two of the constructors and the query `daysUntil` to calculate the number of days from Luke's birthday until the current date.

Running this program on the day this paragraph was written gives an answer of 5,009 days.

FIND THE CODE

cho8/lukesAge/

Listing 8-2: *A simple program to calculate and print someone's age, in days*

```
1  import becker.util.DateTime;
2
3  public class Main
4  { public static void main(String[] args)
5    {
6      DateTime lukesBD = new DateTime(1990, 10, 1);
7      DateTime today = new DateTime();
8
9      int daysOld = lukesBD.daysUntil(today);
10     System.out.println("Luke is " + daysOld + " days old.");
11   }
12 }
```

Reimplementing the `Person` Class

Using the `DateTime` class, we can replace six instance variables in our original class with only two—one to represent the birth date and another to represent the death date. Besides eliminating instance variables, some of the code from the `Person` class can now be delegated to the `DateTime` class. This is like a high-level manager delegating work to one of her employees. Delegation can make more effective use of the resources available.

KEY IDEA

Collaborative classes are all about getting someone else to do the work.

In Listing 8-3, this delegation of work occurs at line 45. The `daysLived` method uses the `daysUntil` method in `DateTime` by calling `this.birth.daysUntil`, which is just like calling `lukesBD.daysUntil` (line 9, Listing 8-2) except that `luke` was a temporary variable within the `main` method. Here, we use `this` to access the instance variable referring to the `DateTime` object. In both cases, we are asking a `DateTime` object to perform a service on our behalf—and if `DateTime` can do it for us, we don't have to do it ourselves.

But we're getting ahead of ourselves. Lines 12 and 13 of Listing 8-3 show the declaration of the two `DateTime` objects to store the birth and death dates. These declarations are like other instance variable declarations except that instead of a primitive type such as `int`, they use the name of a class or interface.

Listing 8-3: *An implementation of* `Person` *that collaborates with the* `DateTime` *class*

```
1  import becker.util.Test;
2  import becker.util.DateTime;
3
4  /** Represent a person.
5  *
6  *  @author Byron Weber Becker */
7  public class Person extends Object
8  {
9     private String name;                    // person's name
10    private String mother;                  // person's mother's name
11    private String father;                  // person's father's name
12    private DateTime birth;                 // birth date
13    private DateTime death = null;          // death date (null if still alive)
14
15    /** Represent a person who is still alive. */
16    public Person(String aName, String mom, String dad,
17                  int bYear, int bMonth, int bDay)
18    { this(aName, mom, dad, bYear, bMonth, bDay, 0, 0, 0);
19    }
20
```

FIND THE CODE

cho8/collabPerson/

PATTERN

Has-a (Composition)

Listing 8-3: *An implementation of* `Person` *that collaborates with the* `DateTime` *class* (continued)

```
21    /** Represent a person who has died. */
22    public Person(String aName, String mom, String dad,
23                  int bYear, int bMonth, int bDay,
24                  int dYear, int dMonth, int dDay)
25    { super();
26      this.name = aName;
27      this.mother = mom;
28      this.father = dad;
29
30      this.birth = new DateTime(bYear, bMonth, bDay);
31      if (dYear > 0)
32      { this.death = new DateTime(dYear, dMonth, dDay);
33      }
34    }
35
36    /** Return the number of days this person has lived. */
37    public int daysLived()
38    { DateTime endDate = this.death;
39      if (this.death == null)
40      { endDate = new DateTime();
41      }
42      return this.birth.daysUntil(endDate);
43    }
44
45    /** Set the death date to a new value. */
46    public void setDeathDate(int dYear, int dMonth, int dDay)
47    { this.death = new DateTime(dYear, dMonth, dDay);
48    }
49
50    // Accessor methods omitted.
51    // main method omitted. It's the same as Listing 8-1 but with a few additional tests.
52  }
```

The instance variable `birth` is initialized in line 30 to refer to a new `DateTime` object. The form of its initialization is like all the objects we've constructed except that we use `this` to access the instance variable assigned the new value. The birth date is always dependent on information passed to the constructor's parameters and is therefore always performed in the constructor.

KEY IDEA

Variables refer to objects rather than containing them.

We say `birth` "refers" to an object rather than "contains" an object. This is a subtlety that we'll explore in detail in Section 8.2. Until then, we'll use the appropriate language for accuracy even though it hasn't been fully explained.

`null` Values

Unlike `birth`, `death` may or may not refer to an object, depending on whether the person has already died. Lines 13 and 32 address the issue of what to do with a person who hasn't died. The declaration in line 13 assumes that the person has not died and initializes `death` to the special value **null**. `null` can be assigned to any reference variable and means that the variable does not refer to any object at all. If it turns out that the person has died, the variable is reinitialized in line 32 with a `DateTime` object.

KEY IDEA

Use `null` *when there is no object to which the variable can refer.*

This example represents a common situation: A reference variable is needed but sometimes no object is appropriate to store there. At those times, use `null`. In this case, storing `null` means that the person has not yet died. We can determine if the person has died by comparing `death` with `null` using the `==` and `!=` operators. This is shown in line 39. If the death date is `null`, the person is still alive and the temporary variable `endDate` is assigned the current date. Otherwise, `endDate` is assigned the date the person died.

Null values can lead to trouble for beginning and experienced programmers alike. The problem stems from assuming the variable refers to an object when it does not. For example, suppose you want to know how many days have passed since a person died. A natural approach is to add the following method to `Person`:

```
public int daysSinceDeath()
{ DateTime today = new DateTime();
  return this.death.daysUntil(today);
}
```

If `death` refers to a `DateTime` object, this works as desired. However, if `death` contains `null`, executing this code will result in a `NullPointerException`. An exception stops the program and prints a message that contains helpful information for finding the problem. Adding a line that calls `daysSinceDeath` to the `main` method in Listing 8-3 results in the following error message:

KEY IDEA

Variables containing `null` *can't be used to call methods.*

```
Exception in thread "main" java.lang.NullPointerException
        at Person.daysSinceDeath(Person.java:53)
        at Person.main(Person.java:75)
```

This message says that the problem was a `NullPointerException` (which means we tried to use a `null` value as if it referred to an object). Furthermore, it tells us that it occurred in the method we added (`daysSinceDeath`), which would appear in Listing 8-3 at line 53. Note that the error message tells us the filename and line number. If we're curious about why the program was executing `daysSinceDeath` in the first place, the subsequent line(s) trace the execution all the way back to the `main` method.

KEY IDEA

Exceptions give useful information to help find the error.

8.1.3 Diagramming Collaborating Classes

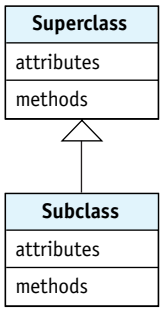
KEY IDEA

Class diagrams show the relationships between collaborating classes.

We have used class diagrams regularly to give an overview of an individual class. These diagrams can also be used to show the relationships between collaborating classes. In fact, we've already seen class diagrams showing such collaborating classes: when we extended one class to form a new one with additional capabilities (see Sections 2.2 and 3.5.3). In that situation, we generally place the superclass above the subclass and connect the two with a closed arrow pointing to the superclass. A generic example is shown in Figure 8-3.

(figure 8-3)

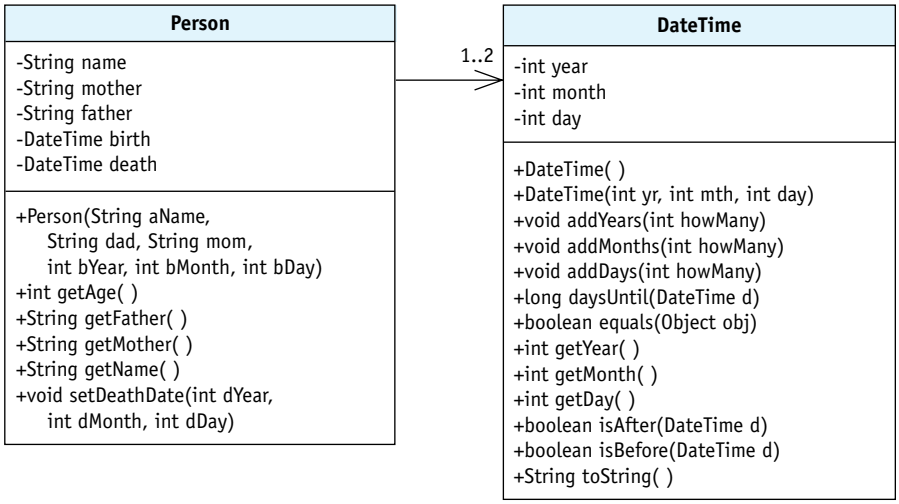
Class diagram showing two classes collaborating via inheritance



However, the `Person` class does not extend `DateTime` (nor is the reverse true), and so we use a different diagramming convention. This convention uses an open-headed arrow from one class to the other. The tail of the arrow is the class containing the instance variable and the head of the arrow is the class representing the variable's type. Usually the classes are drawn side by side, if possible. A class diagram for the `Person` class in Figure 8-4 serves as an example.

(figure 8-4)

Class diagram for the `Person` *class showing its collaboration with* `DateTime`



Another feature of the diagram is the **multiplicity** near the arrowhead. The `1..2` in the diagram shows that each `Person` object uses at least one but no more than two `DateTime` objects. A class diagram will show each class only once, no matter how many objects are actually created using the classes. In general, the first number is the minimum number of objects that will be used, and the second number is the maximum number that will be used in the running program.

PATTERN

Has-a (Composition)

Other multiplicities are common. `1` is an abbreviation for `1..1` and means that exactly one object is used. An asterisk (`*`) is used to mean "many." An asterisk by itself is an abbreviation for `0..*` meaning "anywhere from none to many." If there will always be at least one but possibly many, use `1..*`. An arrow without an explicit multiplicity is assumed to be `1`.

The inheritance relationship, as shown in Figure 8-3, never includes a multiplicity.

Clients and Servers

In Section 1.1.2, we briefly discussed the terms **client** and **server**. Here we see those roles depicted graphically. The arrow goes from the client to the server. The client, `Person`, requests a service such as finding the days until another date. The server, `DateTime`, is the class or object that performs the service.

"Is-a" versus "Has-a"

How do you know which diagramming convention to use? If you already have the Java code, you examine the code. If the code says `public class X extends Y`, use the "**is-a**" relationship shown in Figure 8-3. If the class has an instance variable referring to an object, use the "**has-a**" relationship shown in Figure 8-4.

"Is-a" comes from the sentence "An `X` is a kind of `Y`." For example, "a `Harvester` robot is a kind of `Robot`" (see Listing 3-3) or "a `Lamp` is a kind of `Thing`" (see Listing 2-6). Other examples include "a `Circle` is a kind of `Shape`," "an `Employee` is a kind of `Person`," and "an `Automobile` is a kind of `Vehicle`." Given two classes, if a sentence like any one of these makes sense, then using `extends` and a diagram like Figure 8-3 is often the right thing to do.

LOOKING AHEAD

We'll examine is-a relationships more carefully in Chapter 12.

On the other hand, it's more often the case that "an `X` has a `Y`." In that case, we use the "has-a" relationship, also called **composition**. "A `Person` has a `birth date`" or "a `GasPump` has a `Meter`" or "an `Automobile` has an `Engine`." Has-a relationships are implemented by adding an instance variable in the class that "has" something and is diagrammed similar to Figure 8-4.

8.1.4 Passing Arguments

LOOKING BACK

Overloading involves two or more methods with the same name but different signatures. See Section 6.2.2.

Passing object references as arguments is like passing an integer: declare a parameter variable in the method's declaration and pass a reference to an object when the method is called. For example, the `setDeathDate` method (lines 46–48 in Listing 8-3) could be overloaded with another version of `setDeathDate` that takes an object reference as an argument:

```
public void setDeathDate(DateTime deathDate)
{ this.death = deathDate;
}
```

Both this method and the original accomplish the same purpose: assigning a new `DateTime` object to the `death` instance variable. The difference is in where the object is constructed. In the original version, the method received the year, month, and day, and then constructed the object itself. In this version, the client constructs the object.

8.1.5 Temporary Variables

We have been using temporary variables to refer to objects since our first program. In our first program, we wrote the following lines:

```
 8 City prague = new City();
 9 Thing parcel = new Thing(prague, 1, 2);
10 Robot karel = new Robot(prague, 1, 0, Direction.EAST);
```

We didn't mention that `prague`, `karel`, and `parcel` are all temporary variables referring to objects, but they are. They can be similarly used in any method, not just `main`. However, remember that temporary variables only exist while the method containing them is executing. As soon as the method is finished, so are the temporary variables.

8.1.6 Returning Object References

Finally, a query may return an object reference as easily as it can return an integer. For example, we could add a query to our `Person` class to get the person's birth date. Listing 8-4 shows an abbreviated version of the class.

Listing 8-4: *An abbreviated version of the* `Person` *class showing* `getBirthDate`

```
 7  public class Person extends Object
 8  { ... // instance variables omitted
12    private DateTime birth;              // birth date
13    private DateTime death;              // death date (null if still alive)
      ...
51
52    public DateTime getBirthDate()
53    { return this.birth;
54    }
55  }
```

A client could use this query to compare the ages of two persons, as in the following example. Assume that `luke` and `caleb` both refer to `Person` objects.

```
1  DateTime lukesBD = luke.getBirthDate();
2  if (lukesBD.isBefore(caleb.getBirthDate())
3  { System.out.println("Luke is older.");
4  } else if (caleb.getBirthDate().isBefore(lukesBD))
5  { System.out.println("Caleb is older.");
6  } else
7  { System.out.println("Luke and Caleb are the same age.");
8  }
```

In line 1, the `getBirthDate` query is used to assign a value to the temporary variable `lukesBD`.

The `isBefore` query is used in line 2 to compare two dates—Luke's birth date and Caleb's birth date. In this case, Luke's birth date is held in a temporary variable, but the value to use for Caleb's birth date is obtained directly from the relevant `Person` object via our new query.

Line 4 shows that the object reference returned by `getBirthDate` does not even have to be saved in a variable before it can be used to call a method. Read the statement left to right. The first part, `caleb`, is a reference to a `Person` object. Any such reference can be used to call the methods in the object, including `getBirthDate`. This call returns a reference to a `DateTime` object. Any such reference, whether it is stored in a variable or returned by a query, can be used to call methods in the `DateTime` class, including `isBefore`. This query returns a Boolean value, so no further method calls can be chained to the end of this expression.

KEY IDEA

Methods that return references can be chained together, eliminating the need for some temporary variables.

8.1.7 Section Summary

In this section, we've seen how to implement a class, `Person`, that collaborates with another class, `DateTime`. This particular relationship is sometimes called the "has-a" relationship because a person has a birth date and a death date. This relationship is also called composition.

We have also seen that references to objects such as the birth date and death date can be used much like integers and other primitive types. They can be used as instance variables, temporary variables, and parameter variables, and can be returned by queries.

8.2 Reference Variables

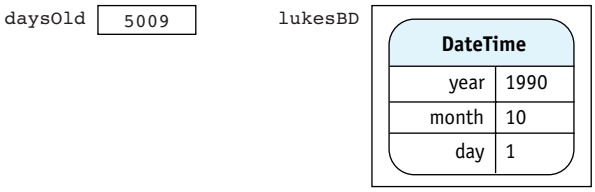
Throughout the previous section, we used phrases like "references to objects" and "object references." What do those phrases really mean?

Consider again the program to calculate Luke's age in days, which appeared in Listing 8-2 and is reproduced in Listing 8-5. We'll focus on two variables, `lukesBD` and `daysOld`. We know that a variable stores a value; this was one of the basic concepts introduced in Chapter 6, where variables were described as being like a box that has a name. Inside the box is a value, such as 5009, that can be retrieved by giving the name of the variable.

Listing 8-5: *A simple program reproduced from Listing 8-2*

```
1  import becker.util.DateTime;
2
3  public class Main extends Object
4  { public static void main(String[] args)
5    {
6      DateTime lukesBD = new DateTime(1990, 10, 1);
7      DateTime today = new DateTime();
8
9      int daysOld = lukesBD.daysUntil(today);
10     System.out.println("Luke is " + daysOld + " days old.");
11   }
12 }
```

At this point you might imagine `daysOld` and `lukesBD` as something like the illustrations in Figure 8-5. The "box" for `daysOld` holds the value `5009` and the "box" for `lukesBD` holds an object, represented with an object diagram.



(figure 8-5)

Simplistic visualization of two variables

This is an accurate enough description for `daysOld`, but not for `lukesBD`. `lukesBD` is a **reference variable**, a variable that refers to an object rather than actually holding the object. To understand what this means, we need to better understand the computer's memory.

8.2.1 Memory

Every computer has **memory**, where it stores information. This information includes values stored in variables such as `daysOld` and `lukesBD`, objects, text, images, and audio clips. Even the programs themselves constitute information stored in the computer's memory.

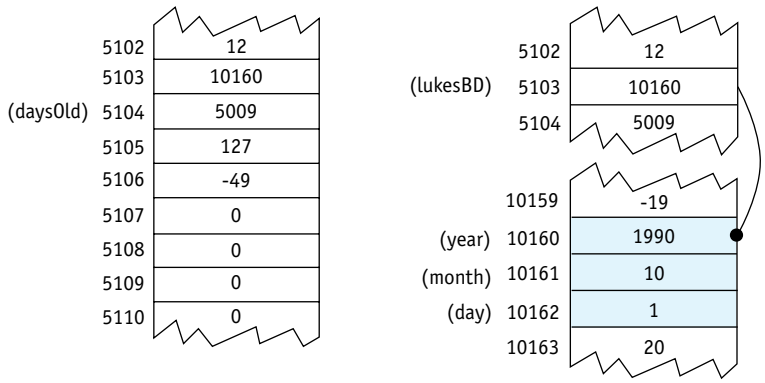
Memory is composed of many storage locations; these are the "boxes" we've described that hold the information. Each location has its own **address**, numbered consecutively beginning with 0. The address is how the computer program identifies which memory location it should access. Each variable name in the program is associated by the Java compiler with a specific memory address, as shown in Figure 8-6a. It shows the variable `daysOld` associated with the memory address 5104. The current value of `daysOld`, 5009, is in that location. Notice that every location has a value, even if it's 0.

The point of this discussion is that objects are handled differently from primitive types, such as integers. The variable `lukesBD`, for example, is associated with an address, and its value is stored in a memory location just like `daysOld`. However, that memory location does not store the object itself but the address of the object; that is, it *refers* to the object, as shown in Figure 8-6b. Notice that the object takes up several memory locations—one for each of the three instance variables.[1]

[1] We are glossing over the fact that one location is only big enough to store a value between –128 and 127. A larger number, such as occupied by an `int` or an address, requires four locations. Every `int` requires four locations, even if the actual value is between –128 and 127.

(figure 8-6)

Illustrating a variable storing a primitive type and a reference variable



a. Variable storing a primitive type

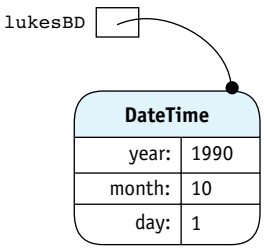
b. Reference variable

Why not store the object at the address associated with `lukesBD`, as illustrated in Figure 8-5? Why do we store the address of the object in `lukesBD` instead? The answer involves efficiency—making the program run faster. If you need to pass an object as an argument, for example, it is faster to pass a reference than to pass the entire object. A reference is always the same size and does not occupy very much memory. Objects, on the other hand, vary in length and can occupy a large amount of memory.

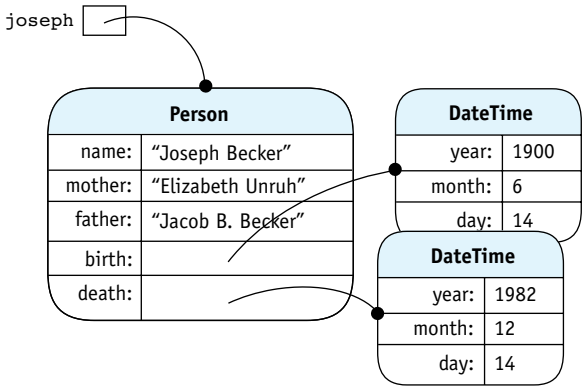
Fortunately, we can usually ignore addresses and memory locations, and let the computer manage them. We only need to keep in mind that reference variables refer to an object instead of hold the object directly. A simplified diagram, as shown in Figure 8-7 will be sufficient to do this.

(figure 8-7)

Simplified diagram showing a reference variable



References are often held in an object, as with the birth and death dates in a `Person` object. In these cases, we can diagram the objects as shown in Figure 8-8.



(figure 8-8)

Object with two instance variables referring to other objects

8.2.2 Aliases

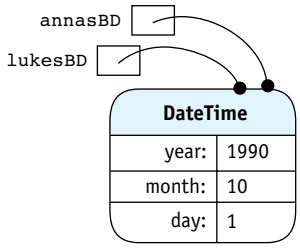
One way that reference variables are different from primitive variables is that it is possible to have several variables refer to the same object. For example, consider the following statements:

```
DateTime lukesBD = new DateTime(1990, 10, 1);
DateTime annasBD = lukesBD;
```

KEY IDEA

Assigning reference variables copies the address from one to the other. The object itself is not copied.

The results of these statements are shown in Figure 8-9. In the second line, it's the address of the date object that is copied from `lukesBD` to `annasBD`. Now both variables refer to the same object.



(figure 8-9)

Assigning one reference variable to another

We can use either reference variable to invoke the object's methods, as in the following statements:

```
lukesBD.addYear(1);
annasBD.addYear(2);
```

Executing these statements changes the date for this object from 1990 to 1993.

Having two or more variables refer to the same object is called **aliasing** and is similar to people with aliases. For example, the Beatles drummer would presumably answer to either Ringo Starr or the name his parents gave him, Richard Starkey.

The question is, why would you want two variables that refer to the same object? The example involving Luke's and Anna's birthdays is clear but rarely used. A closely related example, however, occurs frequently. That is when a reference variable is passed as an argument to a method. Consider the following method:

```
public void adjustDate(DateTime d)
{ d.addYear(2);
}
```

This method could be called as follows:

```
DateTime lukesBD = new DateTime(1990, 10, 1);

lukesBD.addYear(1);
this.adjustDate(lukesBD);
```

While the method `adjustDate` is executing, both `lukesBD` and the parameter variable `d` refer to the same object. When `adjustDate` is called, the value in the argument, `lukesBD`, is copied into the parameter variable, `d`. Once again, two variables contain the address of the same object. Either one can be used to invoke the object's methods, and the net result of this three-line fragment is that the object's year, 1990, is changed to 1993.

The Dangers of Aliases (advanced)

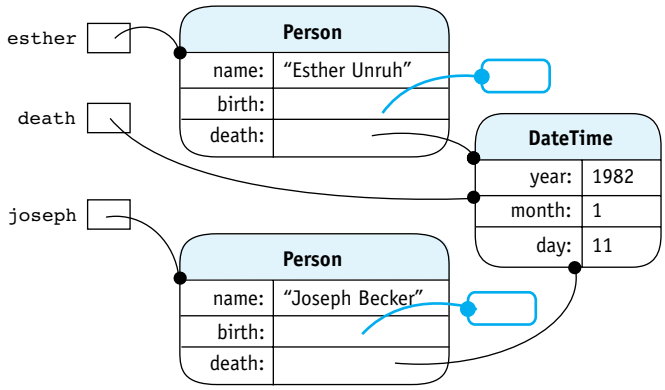
Aliases can lead to dangerous situations. Consider the following code, where `joseph` and `esther` are both instances of `Person`. They died eight years apart.

```
1 DateTime death = new DateTime(1974, 1, 11);
2 esther.setDeathDate(death);
3 death.addYears(8);
4 joseph.setDeathDate(death);
```

KEY IDEA

Aliases can be used to change objects unintentionally or maliciously.

Here, the programmer avoids constructing a new `DateTime` object. What is the effect of this code? Because both `esther` and `joseph` refer to the same `DateTime` object, one of their death dates will be wrong. In lines 1 and 2, `esther`'s death date is set correctly. However, when `death` is changed in line 3, `esther`'s death date inadvertently changes as well because they both refer to the same object. Finally, the date is set for `joseph`, resulting in the situation shown in Figure 8-10—a single `DateTime` object that has three references to it and is shared by both `esther` and `joseph`.



(figure 8-10)

Two `Person` *objects inadvertently sharing the same* `DateTime` *object*

A similar danger can result from an accessor method that returns a reference. The `getBirthDate` method (Section 8.1.6) returns a reference to the relevant `DateTime` object. Once the client has that reference, it could use it to reset the birth date—perhaps to a year that has not yet occurred.

```
DateTime birth = joseph.getBirthDate();
birth.addYears(291);
```

A two-line example makes the error obvious, but such an error can also be separated by many lines of code and be much more difficult to identify.

There are measures you can take to protect your code from aliasing errors. First, you could verify that the referenced object is immutable, meaning it has no methods to change its state. If the state can't change, it doesn't matter if the object is shared. Unfortunately, `DateTime` is not immutable, so this approach won't work here. `String`, a commonly used class, is immutable.

LOOKING BACK

Immutable classes were discussed in Section 7.3.3.

Second, the methods could avoid accepting or returning references in the first place. The first version of `setDeathDate`, which takes integer values for the year, month, and day, avoids this problem. Instead of having `getBirthDate` return a reference, determine why the client wants the reference. For example, if the purpose is to change the birth date, provide an `updateBirthDate` method that performs integrity checks to ensure the new date is reasonable.

LOOKING AHEAD

Listing 11-4 shows how to use `DateTime` *to make an immutable* `Date` *class.*

A third approach, and probably the most common, is to hope that the object's clients won't cause problems with the references. This is good enough in many situations, particularly if the program is well tested. However, in safety-critical applications or an application that may be the target of fraud, this approach is not sufficient.

The fourth, and safest, approach when using a mutable class is to make a copy of the object. For example, `setDeathDate` could be implemented as follows:

```
public void setDeathDate(DateTime deathDate)
{ DateTime copy = new DateTime(deathDate.getYear(),
                  deathDate.getMonth(), deathDate.getDay());
  this.death = copy;
}
```

Another `DateTime` constructor returns a copy of a `DateTime` object it is passed. The following `getBirthDate` method uses it to return a copy of the birth date.

```
public DateTime getBirthDate()
{ DateTime copy = new DateTime(this.birth);
  return copy;
}
```

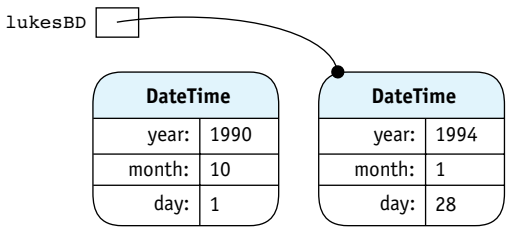
8.2.3 Garbage Collection

Not only can an object have several variables referencing it, but it might have none. Consider the following situation, illustrated in Figure 8-11. An object is created, but then its reference is assigned a new value. The result is that the first object is **garbage**; there is no way to access the object because there are no references to it.

```
DateTime lukesBD = new DateTime(1990, 10, 1);
...
lukesBD = new DateTime(1994, 1, 28);
```

(figure 8-11)

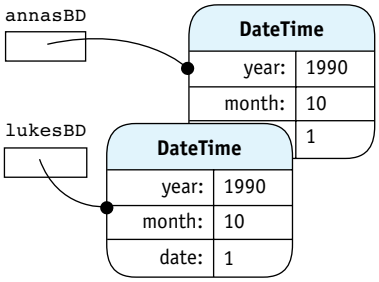
Object with no references



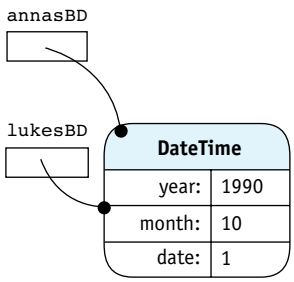
As in the rest of life, garbage is undesirable. It consumes computer memory but cannot affect the running of the program because there is no way to access it. To address this situation, the Java system periodically performs **garbage collection**. It scans the computer's memory for unreferenced objects, enabling the memory they consume to be reused again when new objects are allocated. Because the memory can be reused, "memory recycling" might be a better name than "garbage collection."

8.2.4 Testing for Equality

Testing two objects for equality is a bit tricky. Suppose you have the situation shown in Figure 8-12a.



a. `annasBD == lukesBD` returns `false`



b. `annasBD == lukesBD` returns `true`

(figure 8-12)

Testing to determine if Anna and Luke have the same birthday

If you want to check whether Anna and Luke were born on the same day, you might write the following statement:

```
if (annasBD == lukesBD)
{   // what to do if they have the same birthday
```

This is, after all, what you would write to compare two integer variables. For example, if `annasAge` and `lukesAge` are two integer variables containing the ages of Anna and Luke, then the following code tests whether both variables contain the same value.

```
if (annasAge == lukesAge)
{   // what to do if they are the same age
```

If they both contain 18, for example, the == operator returns `true`.

The statement `if (annasBD == lukesBD)` also tests whether both variables contain the same value. In this case, however, the values being compared are object references, not the objects themselves. In other words, the test will be true if `annasBD` and `lukesBD` both contain the same address in memory and thus refer to exactly the same object. A situation where this is `true` is shown in Figure 8-12b.

KEY IDEA

Comparing object references with == returns `true` if they refer to exactly the same object.

Sometimes this behavior is exactly what is needed. For example, in Chapter 10, we will search lists of objects. We may want to know if a specific object is in the list or not, and a test containing == is the tool to use. This approach to equality is called **object identity**.

A Method to Test Equivalence

In the case of comparing birth dates, what we really need is **object equality**, or **equivalence**. We want to compare two date objects and determine if they have the same meaning. In the

case of `DateTime` objects, they are equivalent if both objects have the same values for year, month, and day.

Testing for equivalence is done with a method such as the following in the `DateTime` class:

Equivalence Test

```
public boolean isEquivalent(DateTime other)
{ return other != null &&    // Make sure other actually refers to an object!
         this.year == other.getYear() &&
         this.month == other.getMonth() &&
         this.day == other.getDay();
}
```

The test for `null` protects against a `NullPointerException` occurring later in the method.

After the test for `null` comes a series of tests to ensure that all the relevant fields in the two objects are equivalent. If the relevant fields are primitive types, as shown here, use `==` for the test. If they are reference fields, use either an `isEquivalent` method that you've written or `equals` for provided classes.

This method could be used to test whether `annasBD` and `lukesBD` refer to objects with equivalent dates by writing one of the following statements:

```
if (annasBD.isEquivalent(lukesBD)) ...
```

or

```
if (lukesBD.isEquivalent(annasBD)) ...
```

KEY IDEA

Code in a given class can use the private instance variables of any instance of that class.

This version of `isEquivalent` is more verbose than necessary. So far we have only accessed private instance variables using `this`. However, Java allows us to access the private members of any object belonging to the same class. That is, inside the `DateTime` class, we can also access the instance variables for `other`—the `DateTime` object passed as an argument. Using this fact, the method can be rewritten as follows:

```
public boolean isEquivalent(DateTime other)
{ return other != null &&    // make sure other actually refers to an object!
         this.year == other.year &&
         this.month == other.month &&
         this.day == other.day;
}
```

Overriding `equals`

LOOKING AHEAD

`equals` is discussed again in Section 12.4.2.

The `Object` class has a method named `equals` that is meant to test for equivalence. Most classes should provide a method named `equals` that overrides the one in `Object`. Unfortunately, technicalities in doing so are difficult to explain without knowing about polymorphism, the topic of Chapter 12.

8.3 Case Study: An Alarm Clock

Suppose you are one of those people who lose all track of time when you're working at your computer. What you need is a computer-based alarm clock that rings an alarm to remind you when it's time to take a break, call a friend, attend a meeting, or quit for the day. You set the alarms for the day when you begin work and let the program run. When one of the alarms is due, it will print a message on the console and play a sound to get your attention. Our first version will be limited to four alarms.

KEY IDEA

A design methodology can help us figure out how to get started on a complex problem.

Now that our problems are getting more complex and will often involve several classes, it may not be obvious which classes we need and how they work together. A design methodology is helpful. The methodology shown in Figure 8-13 is a set of steps to help us get started.

1. Read the description of what the program is supposed to do, highlighting the nouns and noun phrases. These are the objects your program must declare. If there are any objects that cannot be directly represented using existing types, define classes to represent such objects.
2. Highlight the verbs and verb phrases in the description. These are the services. If a service is not predefined:
   a. Define a method to perform the service.
   b. Place it in the class responsible for providing the service.
3. Apply the services from Step 2 to the objects from Step 1 in a way that solves the problem.

(figure 8-13)

Object-based design methodology

Program design is as much art as science. The methodology leaves room for interpretation, and programming experience helps with recognizing and implementing common design patterns. Nevertheless, these basic steps have proven helpful to object-oriented programmers of all experience levels and on all sizes of projects. In fact, the larger the project, the more help these steps are in getting started.

The opening paragraph of Section 8.3 is our description of what the program is supposed to do.

8.3.1 Step 1: Identifying Objects and Classes

The first step in the methodology is to use nouns and noun phrases to identify the relevant classes to solve the problem. A noun is a person, a place, a thing, or an idea. The most important nouns in the description are *alarm clock*, *alarm*, and *time*. Other nouns include *program*, *message*, *console*, and *sound*.

KEY IDEA

Nouns in the specification often identify classes needed in the program.

Some of these can be represented with objects from existing classes. For example, time can be represented with the `DateTime` class, and a message with the `String` class; the console is where strings are printed by `System.out.println`. Exploring the online *Java Tutorial*[2] reveals the `AudioClip` class as one way to work with sound.

This leaves only the nouns *alarm clock* and *alarm* to develop into classes. We'll call them, appropriately, `AlarmClock` and `Alarm`.

Class Relationships

KEY IDEA

Sometimes a noun represents an attribute, not a class.

Sometimes the less important nouns go with another noun. For example, *message* and *sound* go with `Alarm` ("when an alarm is due, it will print a message…and play a sound"). They will appear as instance variables in the `Alarm` class. The "has-a" test from Section 8.1.3 also applies here: "An `Alarm` has-a message to display" and "an `Alarm` has-a sound to play."

The noun *time* applies in two ways. First, time is linked to `Alarm` in the statement "rings an alarm…when it's time," and "when one of the alarms is due" implies time. The "has-a" test makes sense, too: "An `Alarm` has-a time when it rings."

KEY IDEA

A solid arrow in a class diagram indicates an instance variable. A dotted line indicates a temporary variable or parameter.

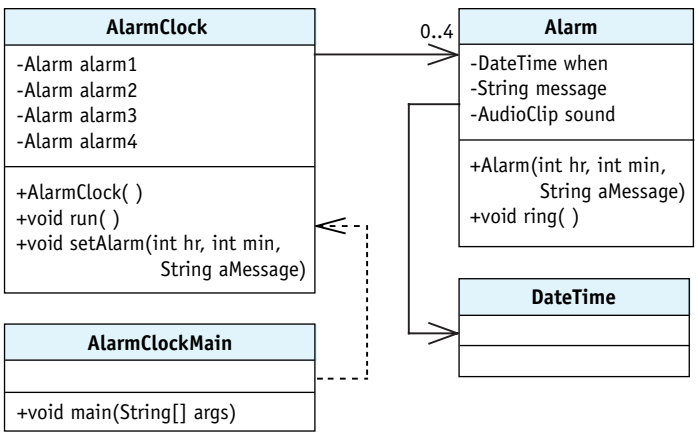
Second, *time* is used by the `AlarmClock` class to keep track of the current time. The instance of `DateTime` will be a temporary variable, not an instance variable.

In addition, the alarm clock has up to four alarms. Again, the appearance of the word *has* indicates the presence of instance variables in the `AlarmClock` class.

Putting these observations together results in the classes, attributes, and class relationships shown in Figure 8-14. The class diagram also includes a class holding the `main` method where execution begins.

(figure 8-14)

First class diagram of the alarm clock program



[2] See *http://java.sun.com/docs/books/tutorial/sound/index.html*

8.3.2 Step 2: Identifying Services

Step 2 in the object-based design methodology is to identify the services required in the classes by analyzing the verbs and verb phrases. Verbs are action words such as *ring*, *run*, *set*, and *print*. Verbs are used in the program description as "ring an alarm," "remind you when," "set the alarm," "run (the program)," "print a message," and "play a sound."

KEY IDEA

Verbs in the specification often identify services in the program's classes.

Some of these verbs are different descriptions of the same thing. For example, an alarm rings to remind you of something. It does so by displaying a message and playing a sound to get your attention. All of that could be collapsed into a single `ring` service in the `Alarm` class.

That still leaves setting the alarms, which sounds like it might be a service of the `AlarmClock` class, and running the program. This phrase is often a generic way of saying we should execute the program. In this case, however, we actually need a method that keeps the time for the clock. We'll name it `run`.

These services and the classes to which they are assigned are also shown in Figure 8-14.

Implementing Methods in `Alarm`

Now let's turn to implementing these methods, beginning with the `Alarm` class. We will defer the sound until later; our first version will "ring" the alarm by only printing a message.

KEY IDEA

Defer nonessential features until after the core features are working.

The constructor is passed the hour and minute that the alarm should ring and the message that should print. We'll use a `DateTime` object internally to represent the time the alarm should ring. The time and message must be remembered until they are needed by the `ring` method and are therefore saved in instance variables.

Has-a (Composition)

```
public class Alarm extends Object
{
  private DateTime when;
  private boolean hasRung = false;
  private String msg = "";

  /** Construct a new Alarm for today at the given time.
  *  @param hr        the hour the alarm should "ring"
  *  @param min       the minute of the hour that the alarm should "ring"
  *  @param msg       the message the alarm gives */
  public Alarm(int hr, int min, String msg)
  { super();
    this.when = new DateTime();
    this.when.setTime(hr, min, 0);
    this.msg = msg;
  }
}
```

The `ring` method is shown in the following code. It prints the time and the alarm's message on the console, using the `format` method in line 5 to format the alarm's time as a `String`. Two method calls at lines 3 and 4 determine how much information is presented.

```
1    /** Alert the user. */
2    public void ring()
3    { this.when.setFormatInclude(DateTime.TIME_ONLY);
4      this.when.setFormatLength(DateTime.SHORT);
5      String time = this.when.format();
6      System.out.println(time + ": " + this.msg);
7    }
```

Implementing Methods in `AlarmClock`

KEY IDEA

Asking and answering questions is a useful technique, even if you are programming by yourself.

The `AlarmClock` class has three fundamental things to do: keep the current time, ring the alarms at the correct times, and provide a way to set the alarms. We'll start with the `run` method, which keeps the current time. It will also call a helper method to ring the alarms, if appropriate. This method is not trivial, so we'll return to the expert-and-novice format of earlier chapters.

Expert What does `run` method need to do?

Novice Keep track of the current time. And if it's time for one of the alarms to ring, it needs to ring it.

Expert Is this something it does just once?

Novice Not really. As time passes, it will need to check again and again whether it is time to ring an alarm.

Expert So it sounds like a loop would be appropriate. What needs to be repeated inside the loop?

Novice It needs to figure out the current time and check if the alarms should be rung.

Expert So when should that loop stop?

Novice When there are no more alarms to ring.

Expert What's the negation of that condition? That tells us whether the loop should continue.

Novice Hey. It sounds like you're leading me through the four-step process for building a loop. Step 1 is to identify the actions that must be repeated to solve the problem; Step 2 is to identify the test that must be true when the loop stops and to negate it; Step 3 is to assemble the loop; and Step 4 is determining what comes before or after the loop to complete the solution.

LOOKING BACK

The four-step process for constructing a loop is discussed in Section 5.1.2.

Expert You're absolutely right. Now, what about negating the test in Step 2?

Novice The loop continues as long as there is at least one alarm left to ring.

As for Step 3, I'd like to start with pseudocode. It's easier than thinking in Java right away. Something like this, perhaps?

KEY IDEA

Pseudocode helps you think about the algorithm without distracting Java details.

```
while (number of alarms left > 0)
{ get the current time
  check alarms and ring if it's the right time
}
```

Expert Excellent. The fourth step was to think through what needs to come before or after the loop. What do you think?

Novice I don't think we need to do anything after the loop. Before the loop, we'll need to initialize some variables or something to control the loop.

Expert Yes. We could use an instance variable to count the number of alarms that have not been rung. When we set an alarm, we'll increment the counter; when we ring an alarm, we'll decrement it. Given that, can you code a solution in Java?

Novice I think so. I'm going to assume a helper method to check and ring the alarms for me. That will keep this method simpler.

KEY IDEA

Keep methods short. Use helper methods to reduce complexity.

```
public void run()
{ DateTime currTime = new DateTime();
  while (this.numAlarmsLeft > 0)
  { currTime = new DateTime();
    this.checkAndRingAlarms(currTime);
  }
}
```

Expert How would you evaluate your efforts so far?

Novice Pretty good. With the help of the four-step process for building loops and the pseudocode, I'm pretty confident `run` will do what it is supposed to do.

Expert I agree. I do have one suggestion, however. Let's insert a call to the `sleep` method inside the loop. Your loop probably runs thousands of times per second. We could slow it down with a `sleep` command, giving the computer

more time to do other things. If we insert `Utilities.sleep(1000)` at the end of the loop, it will still check about once per second.

KEY IDEA

Think about testing from the beginning.

Novice Great idea. One thing is bothering me, though. Testing this method is going to be really hard because it runs in real time. If we set an alarm for 3:30 in the afternoon and it's only 10 in the morning now, we'll have to wait 5½ hours to see if the program works!

Expert That *is* a problem. Normally we want to test the same code that makes up the finished solution. Here, however, we may need to make a slight change to make testing easier.

Here's my suggestion: Let's add an instance variable to indicate whether or not we are testing. When we're testing, we'll calculate the current time slightly differently to make time pass more quickly. When the `run` method sleeps for one second, we'll add two seconds to the current time. That makes time pass twice as fast. If we want the virtual time to pass even more quickly, add four or even more seconds to the current time in each iteration of the loop.

If we're *not* testing, we'll continue to calculate the current time as you suggested earlier. Creating a new instance of `DateTime` will keep the time accurate because that constructor actually uses the computer's clock.

Novice If we use a parameter, the method calling `run` can decide how fast the time should pass. Then our new method would look like this:

```
public void run(int secPerSec)
{ DateTime currTime = new DateTime();

  while (this.numAlarmsLeft > 0)
  { if (this.TESTING)
    { currTime.addSeconds(secPerSec);
    } else
    { currTime = new DateTime();
    }

    this.checkAndRingAlarms(currTime);
    Utilities.sleep(1000); // sleep one second real time
  }
}
```

Expert Good. Now, what does your helper method, `checkAndRingAlarms`, need to do?

Novice It will check each alarm's time against the current time. If it's time for the alarm to ring, it will call its `ring` method. Or, in pseudocode (because I know you're going to ask):

```
if (alarm1's time matches current time)
{ ring alarm1
} else if (alarm2's time matches current time)
{ ring alarm2
}
```

We'll need a couple of more tests for the other alarms. I'm assuming the alarms are stored in four instance variables. Seems pretty simple to me.

Expert Actually, I think I see two problems. The first problem is when there is no alarm set. How can you check whether its time matches? If you tried, I think you would get a `NullPointerException`.

The second problem is that you are assuming that only one alarm becomes due at any given time. Remember the Cascading-`if` pattern? It says that only one of the groups of statements will be executed. If two alarms happen to be set for the same time, only the first will ring.

LOOKING BACK

The Cascading-`if` pattern was discussed in Section 5.3.3.

Novice So we could have four separate groups of statements, each one like this:

```
if (alarm is not null)
{ if (alarm's time matches current time)
  { ring the alarm
    decrement the number of alarms left to ring
  }
}
```

Expert Can you improve this? Is the nested `if` statement really necessary? Do you really need to repeat almost the same code four times?

Novice Aha. We can use short-circuit evaluation. If the first part of the "and" is `false`, Java won't even bother to check the second part. And we can put the whole thing in a method to avoid the code duplication. Like this:

LOOKING BACK

Short-circuit evaluation was discussed in Section 5.4.3.

```
private void checkOneAlarm(Alarm alarm, DateTime currTime)
{ if (alarm ! = null && alarm.isTimeToRing(currTime))
  { alarm.ring();
    this.numAlarmsLeft -= 1;
  }
}
```

Expert Good. I see you'll need to add a method, `isTimeToRing`, to the `Alarm` class. I like the way you're asking that class to figure out the answer for you. It's the one with the needed data. Asking `Alarm` for the answer seems better than asking it for its time and then doing the computation yourself.

KEY IDEA

Put methods in the same class as the data they use.

With this helper method, the `checkAndRingAlarms` helper method becomes:

```
private void checkAndRingAlarms(DateTime currTime)
{ this.checkOneAlarm(this.alarm1, currTime);
  this.checkOneAlarm(this.alarm2, currTime);
  this.checkOneAlarm(this.alarm3, currTime);
  this.checkOneAlarm(this.alarm4, currTime);
}
```

The last big step is to set the alarms. Ideas?

Novice We already know we'll have four instance variables. I think we need to just check each one in turn to see if it's `null`. If it is, we can save the alarm in that variable. A cascading-`if` should work. Of course, we also need to construct the `Alarm` itself.

```
public void setAlarm(int hr, int min, String msg)
{ Alarm theAlarm = new Alarm(hr, min, msg);
  if (this.alarm1 == null)
  { this.alarm1 = theAlarm;
  } else if (this.alarm2 == null)
  { this.alarm2 = theAlarm;
  } else if (this.alarm3 == null)
  { this.alarm3 = theAlarm;
  } else if (this.alarm4 == null)
  { this.alarm4 = theAlarm;
  }
}
```

Expert Looks good. But aren't you forgetting something? We made an assumption earlier that we had a count of the number of alarms yet to ring. This seems like the place to include it.

Novice Oops. Add the following to the end of the method:

```
this.numAlarmsLeft++;
```

Expert One more detail to consider for `setAlarm`. What happens if we try to set five alarms?

Novice Right now, absolutely nothing happens. The cascading-`if` statement doesn't have any tests that match and there is no `else` clause. I think the user should know about the error, so I'll add a warning in an `else` clause, as follows:

```
  ...
  } else if (this.alarm4 == null)
  { this.alarm4 = theAlarm;
  } else
  { System.out.println("Too many alarms.");
  }
```

Expert This is a fine solution for now, but throwing an exception would be better. I'm sure you'll learn how soon.

Excellent job. I think we're about done!

All these ideas come together in Listing 8-6 and Listing 8-7.

The `isTimeToRing` method in the `Alarm` class is mentioned in the dialogue but not discussed thoroughly. In this application, we dare not compare two times for equality to see if the alarm should ring because it's possible that the time might be skipped over—particularly given the time acceleration that we built into the `run` method. Instead, we need to check if the time for the alarm has passed and the alarm has not yet been rung. This requires an extra instance variable at line 10 in the `Alarm` class that is checked in the `isTimeToRing` method (line 28) and changed in the `ring` method (line 37).

Listing 8-6: *The* `Alarm` *class*

FIND THE CODE

cho8/alarmClock/

PATTERN

Has-a (Composition)

```
1  import becker.util.DateTime;
2  import becker.util.Utilities;
3
4  /** An Alarm represents a time when someone or something needs to be interrupted.
5  *
6  *  @author Byron Weber Becker */
7  public class Alarm extends Object
8  {
9    private DateTime when;
10   private boolean hasRung = false;
11   private String msg = "";
12
13   /** Construct a new Alarm for today at the given time.
14   *  @param hr        the hour the alarm should "ring"
15   *  @param min       the minute of the hour that the alarm should "ring"
16   *  @param msg       the message the alarm gives */
17   public Alarm(int hr, int min, String msg)
18   { super();
19     this.when = new DateTime();
20     this.when.setTime(min, hr, 0); // Deliberate bug
21     this.msg = msg;
22   }
23
24   /** Is it time for this alarm to ring?
25   *  @param currTime the current time, as determined by the calling clock
26   *  @return true if time for the alarm; false otherwise. */
27   public boolean isTimeToRing(DateTime currTime)
28   { return !this.hasRung && this.when.isBefore(currTime);
29   }
```

Listing 8-6: *The* `Alarm` *class* (continued)

```
30
31    /** Alert the user. */
32    public void ring()
33    { this.when.setFormatInclude(DateTime.TIME_ONLY);
34      this.when.setFormatLength(DateTime.SHORT);
35      String time = this.when.format();
36      System.out.println(time + ":" + this.msg);
37      this.hasRung = true;
38    }
39  }
```

FIND THE CODE

cho8/alarmClock/

Listing 8-7: *The* `AlarmClock` *class*

```
1  import becker.util.DateTime;
2  import becker.util.Utilities;
3
4  /** Maintain a set of up to four alarms. Keep time and ring alarms at the appropriate times.
5  *
6  *  @author Byron Weber Becker */
7  public class AlarmClock extends Object
8  {
9     // Allow up to four alarms.
10    private Alarm alarm1 = null;
11    private Alarm alarm2 = null;
12    private Alarm alarm3 = null;
13    private Alarm alarm4 = null;
14
15    // Count the alarms left to be rung.
16    private int numAlarmsLeft = 0;
17    // Make time pass more quickly when testing.
18    private final boolean TESTING;
19
20    /** Construct a new alarm clock.
21    *  @param test When true, the run method makes time pass more quickly for testing. */
22    public AlarmClock(boolean test)
23    { super();
24      this.TESTING = test;
25    }
26
27    /** Run the clock for one day, ringing any alarms at the appropriate times.
28    *  @param secPerSec The speed with which the clock should run (for testing purposes).
```

PATTERN

Has-a (Composition)

Listing 8-7: *The* `AlarmClock` *class* (continued)

```
29    *  Each second of real time advances this clock the given number of seconds. With
30    *  a value of 3600 one "day" takes about 24 seconds of elapsed time. */
31    public void run(int secPerSec)
32    { DateTime currTime = new DateTime();
33
34      while (this.numAlarmsLeft > 0)
35      { if (this.TESTING)
36        { currTime.addSeconds(secPerSec);
37        } else
38        { currTime = new DateTime();
39        }
40
41        this.checkAndRingAlarms(currTime);
42        Utilities.sleep(1000); // sleep one second real time
43      }
44    }
45
46    // Check each alarm. Ring it if it's time.
47    private void checkAndRingAlarms(DateTime currTime)
48    { this.checkOneAlarm(this.alarm1, currTime);
49      this.checkOneAlarm(this.alarm2, currTime);
50      this.checkOneAlarm(this.alarm3, currTime);
51      this.checkOneAlarm(this.alarm4, currTime);
52    }
53
54    // Check one alarm. Ring it if it's time.
55    private void checkOneAlarm(Alarm alarm, DateTime currTime)
56    { if (alarm != null && alarm.isTimeToRing(currTime))
57      { alarm.ring();
58        this.numAlarmsLeft1-=1;
59      }
60    }
61
62    /** Set an alarm to ring at the given time today. A maximum of four alarms may be set.
63    *  @param hr        The hour the alarm should ring.
64    *  @param min       The minute of the hour the alarm should ring.
65    *  @param msg       Why the alarm is being set */
66    public void setAlarm(int hr, int min, String msg)
67    { Alarm theAlarm = new Alarm(hr, min, msg);
68      if (this.alarm1 == null)
69      { this.alarm1 = theAlarm;
70      } else if (this.alarm2 == null)
71      { this.alarm2 = theAlarm;
```

Listing 8-7: *The* `AlarmClock` *class* (continued)

```
72      } else if (this.alarm3 == null)
73      { this.alarm3 = theAlarm;
74      } else if (this.alarm4 == null)
75      { this.alarm4 = theAlarm;
76      } else
77      { System.out.println("Too many alarms.");
78      }
79
80      this.numAlarmsLeft++;
81    }
82
83    // For testing
84    public static void main(String[] args)
85    { AlarmClock clock = new AlarmClock(true);
86
87      clock.setAlarm(10, 30, "Coffee break");
88      clock.setAlarm(11, 00, "Call Amy");
89      clock.setAlarm(17, 30, "Turn off the computer and get a life!");
90
91      clock.run(3600);
92    }
93  }
```

8.3.3 Step 3: Solving the Problem

The hard part is over. The last step in the methodology is to solve the problem using the methods we created for the various classes. For the alarm clock problem, we can use a `main` method that constructs an `AlarmClock` object, sets alarms, and then calls the `run` method. A sample is shown in Listing 8-8.

FIND THE CODE

cho8/alarmClock/

Listing 8-8: *A* `main` *method to run the alarm clock program*

```
1  import becker.util.DateTime;
2
3  /** Run the alarm clock with today's alarms.
4  *
5  *  @author Byron Weber Becker */
6  public class AlarmClockMain extends Object
7  {
8    public static void main(String[] args)
```

Listing 8-8: *A* `main` *method to run the alarm clock program* (continued)

```
 9    { AlarmClock clock = new AlarmClock(false);
10
11      clock.setAlarm(10, 30, "Coffee break");
12      clock.setAlarm(11, 00, "Call Amy");
13      clock.setAlarm(17, 30, "Turn off the computer and get a life!");
14
15      clock.run(1);
16    }
17  }
```

8.4 Introducing Exceptions

In writing the `setAlarm` method in the `AlarmClock` class (lines 66–81 of Listing 8-7) we noted an error that could occur. An `AlarmClock` object can only store four alarms. If someone tries to add a fifth alarm, he or she should be warned that the maximum has been exceeded. We added a warning `print` statement to address this issue; we can do better.

8.4.1 Throwing Exceptions

Java provides **exceptions** for handling exceptional circumstances—like adding too many alarms to an alarm clock. An `Exception` is an object that, when it is thrown, interrupts the program's normal flow of control. **Throwing an exception** immediately stops the currently executing method, and if nothing is done to intervene, the program will stop with an error message displayed on the console.

There are various subclasses of `Exception` that are more specific about the exceptional circumstance. For example, adding a fifth alarm when our alarm clock can only handle four is attempting to put the object into an illegal state. In such a circumstance, the `IllegalStateException` is applicable.

The original `setAlarm` method used a cascading-`if` statement that concluded with the following code:

```
74      } else if (this.alarm4 == null)
75      { this.alarm4 = theAlarm;
76      } else
77      { System.out.println("Too many alarms.");
78      }
```

Replacing the print statement in line 77 with the following line will throw an `IllegalStateException`.

```
throw new IllegalStateException("Too many alarms.");
```

The constructor's argument is a string describing in more detail what caused the problem. The result of throwing this exception is shown in Figure 8-15.

(figure 8-15)

Exception message printed after attempting to add a fifth alarm

One of the most common exceptions to throw is `IllegalArgumentException`. A good defensive programming strategy is to check the arguments passed to your methods to ensure that they are appropriate. For example, the `setAlarm` method is passed an hour and a minute. The following check, and a similar one for minutes, would be appropriate:

```
if (hr < 0 || hr > 23)
{ throw new IllegalArgumentException(
                              "Hour = " + hr +   "; should be 0–23, inclusive.");
}
```

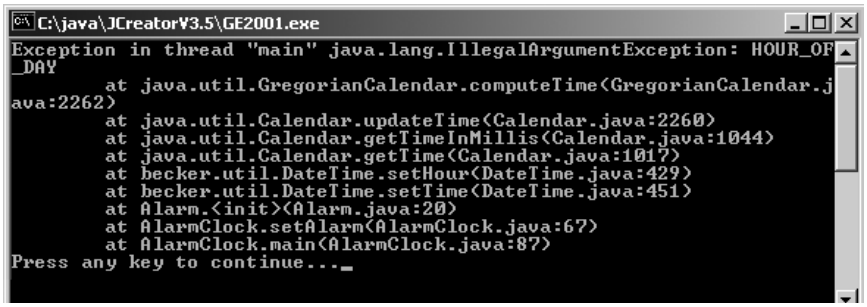
These checks are especially important in constructors where the arguments are often used to initialize instance variables.

8.4.2 Reading a Stack Trace

The information printed when an exception is thrown is very useful for debugging. For example, one run of the alarm clock program produced the exception message shown in Figure 8-16.

(figure 8-16)

Stack trace printed as part of an exception message



The first item of useful information is the name of the exception, `IllegalArgumentException`. The string passed to the exception when it was thrown is "`HOUR_OF_DAY`". Its relevance isn't known yet.

The nine lines following it, each beginning with "at," make up a **stack trace**. A stack trace follows the execution from the exception back to `main`, listing all of the methods that have not yet completed executing. Each line has the following form:

```
at «packageName».«className».«methodName»(«fileName»:«line»)
```

The alarm clock program's classes are not in a package, so that part is blank for the last three lines.

The last line of the stack trace tells us that the `main` method in the `AlarmClock` class called a method at line 87. The method it called is shown on the line above it, `setAlarm`. If we look at line 87 in Listing 8-7, we can verify that `main` calls the `setAlarm` method.

The second-to-last line of the stack trace tells us that `setAlarm` called a method at line 67 in `AlarmClock.java`. The third-to-last line tells us that method was `Alarm.<init>`. This refers to the initialization that occurs when an instance of `Alarm` is constructed, including the initialization of instance variables. In this case, it occurred at line 20 in `Alarm.java`. That line calls the `setTime` method in the `DateTime` class. The rest of the method calls shown in the stack trace are for code in libraries we used.

It's usually most fruitful to debug our code beginning with the line closest to the exception—that is, `Alarm.java` at line 20. It reads as follows:

```
20      this.when.setTime(min, hr, 0);
```

The variable `this.when` is an instance of `DateTime`. Because the exception was `IllegalArgumentException`, we can guess that something was wrong with the arguments passed to the method. In this case, the order looks wrong and a quick check of the documentation confirms that the order of `min` and `hr` is reversed.

8.4.3 Handling Exceptions

Java has two types of exception—checked and unchecked. **Checked exceptions** are exceptions from which the program may be able to recover; in addition, programmers are required to include code to check for them. **Unchecked exceptions** should be thrown only when they result from a program bug. Programmers are not required to check for them. `IllegalArgumentException` and `IllegalStateException` are two examples of unchecked exceptions. Unchecked exceptions include `Error`, `RuntimeException`, and their subclasses. All other exceptions are checked.

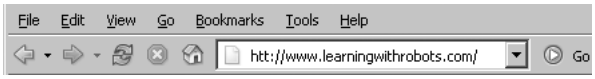
LOOKING AHEAD

We will need to construct a URL to make `Alarm` *play a sound.*

`MalformedURLException` is an example of a checked exception. It might arise from a user typing a Uniform Resource Locator (URL) into the address bar of a Web browser, as shown in Figure 8-17. In this browser, a dialog box is shown stating that "htt is not a registered protocol" (it should be "http" rather "htt").

(figure 8-17)

Address bar of a typical Web browser



In a Java program, such an error would likely be discovered when it constructs a `URL` object. The `URL` constructor takes a string, such as the one typed by the user in the previous figure. If an error is found, the `URL` constructor throws a `MalformedURLException`. This fact is included in the online documentation.

Programmers can check for an exception and handle it with code derived from the following template:

```
try
{ «statements that may throw an exception»
} catch («ExceptionType1» «name1»)
{ «statements to handle exceptions of type ExceptionType1»
} catch («ExceptionType2» «name2»)
{ «statements to handle exceptions of type ExceptionType2»
...
} catch («ExceptionTypeN» «nameN»)
{ «statements to handle exceptions of type ExceptionTypeN»
} finally
{ «statements that are always executed»
}
```

The `try` block contains the statements that the program must try to execute and that may throw an exception. There is a `catch` block for each exception to handle. The `catch` blocks are formatted and executed similar to a cascading-`if` statement. When an exception is thrown, Java starts with the first `catch` block and works its way downward. It executes the statements in the first `catch` block where *«ExceptionType»* matches the exception thrown or is a superclass of the thrown exception.

For example, the mixture of pseudocode and Java in Listing 8-9 shows how to handle the `MalformedURLException` thrown by the `URL` constructor.

Listing 8-9: *A mixture of pseudocode and Java showing how an exception can be caught*

```
1  private void loadPage()
2  { String urlString = get the url typed by the user
3    try
4    { URL url = new URL(urlString);     // can throw MalformedURLException
5      use url to load the page          // can throw IOException
6    } catch (MalformedURLException ex)
7    { display a dialog box describing the error and asking the user to try again
8    } catch (IOException ex)
9    { display a dialog box describing the error
10   }
11 }
```

If the `URL` constructor in this example throws an exception, the statements following it in the `try` block (using the URL) are *not* executed. When an exception is thrown, execution resumes with the nearest `catch` block.

KEY IDEA

An exception skips over code between the line throwing it and a matching `catch` *statement.*

Because `malformedURLexception` extends `IOException`, the order of the `catch` clauses is important. If `IOException` is listed first, it will handle both kinds of exceptions. When listing multiple `catch` clauses, always list the subclasses (most specific exceptions) first and the superclasses (most general exceptions) last.

The names in the `catch`'s parentheses are much like a parameter variable declaration. In the previous example, `ex` is a variable that can be used within the `catch` clause. Recall that an exception is an object, and `ex` can be used to access its methods. For example, the `getMessage` method returns the string that was passed to the exception's constructor. The `printStackTrace` method prints the stack trace. It is often followed with the statement `System.exit(1)`, which causes the program to terminate immediately. Without the call to `exit`, the program would resume after the `try-catch` statement.

The `finally` clause shown in the template is optional. If included, the code it contains will always be executed if any of the code in the `try` block is executed. The `finally` clause is executed even if an exception is thrown, whether or not it is handled in a `catch` clause. It's also executed if a `return`, `break`, or `continue` statement is executed within the `try` block to end it early.

KEY IDEA

Code in the `finally` *clause is always executed if code in the* `try` *block has executed.*

8.4.4 Propogating Exceptions

Methods often can't handle the exceptions thrown by the methods they call. They could catch the exceptions, but can't do anything constructive to respond to the error. In these cases, the exceptions should be propogated up the call stack. This is exactly what happened in Figure 8-16. The `computeTime` method threw an exception. It's

caller, `updateTime`, couldn't handle it constructively and so allowed it to propogate to its caller, `getTimeInMillis`. Likewise, this method could not handle the exeception constructively and allowed it to propogate to its caller. This pattern continued a number of times.

When a checked exception is allowed to propogate like this, the method must declare that fact with the `throws` keyword. For example, suppose the `loadPage` method in Listing 8-9 is not an appropriate place to display a dialog box. The method can be rewritten as follows:

```
private void loadPage()
                throws MalformedURLException, IOException
{ String urlString = get the url typed by the user
  URL url = new URL(urlString);     // can throw MalformedURLException
  use url to load the page          // can throw IOException
}
```

The `throws` clause alerts everyone who might use this method that it can throw the listed exceptions. The clause is required for checked exceptions. If it is omitted, the compiler will issue an error message with the following format:

```
«className»:«lineNum»: unreported exception «exceptionName»;
must be caught or declared to be thrown
```

The reference to "must be caught" means to include the code in a `try-catch` statement. The alternative, "declared to be thrown," means to change the method signature to include the keyword `throws`, as shown earlier.

8.4.5 Enhancing the Alarm Clock with Sound (optional)

We can use our new expertise with exceptions to add sound to the alarm clock program. One way that Java works with sound is via the `AudioClip` class. An `AudioClip` can be loaded from a file using the .wav, .au, or .midi formats (but not .mp3, unfortunately). There may be appropriate sound files already on your computer, or you can create your own with a program such as Audacity, a free sound editor found at *http://audacity.sourceforge.net/*.

The location of the sound file is specified with a URL and can be either on the Web or on your disk.

Listing 8-10 shows the additions to the `Alarm` class to accommodate sound. Four changes are required:

- Lines 3 and 4 import the `Applet`, `AudioClip`, `URL`, and `MalformedURLException` classes.
- Line 8 declares a class variable, `sound`. It's a class variable so that all the `Alarm` instances can share the same sound.

- Lines 13–24 load the sound from a location on the Web. A `URL` is required, which may throw a `MalformedURLException`,1 and so a `try-catch` statement is required. If the exception is thrown, lines 21–22 print a stack trace to aid debugging and exit the program. Because the sound is shared among all instances of `Alarm`, it only needs to be loaded once. The `if` statement at line 14 prevents it from loading more than once.
- Line 31 actually plays the sound. An `AudioClip` has three methods: `play` to play a sound, `stop` to stop a sound currently playing, and `loop` to play a sound repeatedly. Sounds play in their own thread. Line 31 starts that thread, but then execution of the program continues while the sound plays.

The part of this code most likely to cause a problem is specifying the URL for the sound file. If the form of the URL is correct but there is no sound file actually at that location, nothing will notify you; the program just won't play a sound. The best way to avoid this problem is to first locate the file using a Web browser. Then cut and paste the URL from the browser's address bar to the program.

The sound file may also be loaded from your disk drive using a URL similar to the following:

```
URL url = new URL("file:///D:/Robots/examples/ch08/alarmSound/ringin.wav");
```

Listing 8-10: *Modifying the* `Alarm` *class to play a sound*

FIND THE CODE

cho8/alarmSound/

```
1  import becker.util.DateTime;
2  import becker.util.Utilities;
3  import java.applet.*;
4  import java.net.*;
5
6  public class Alarm extends Object
7  { // Same as Listing 8-6.
8    private static AudioClip sound = null;
9
10   public Alarm(int hr, int min, String msg)
11   { // Same as Listing 8-6.
12
13     // Load the sound if it hasn't already been loaded.
14     if (Alarm.sound == null)
15     { try
16       { URL url = new URL(
17             "http://www.learningwithrobots.com/downloads/WakeupEverybody.wav");
18         Alarm.sound = Applet.newAudioClip(url);
19       }
20       catch (MalformedURLException ex)
21       { ex.printStackTrace();
```

Listing 8-10: *Modifying the* `Alarm` *class to play a sound* (continued)

```
            System.exit(1);
          }
        }
      }

      public void ring()
      { // Same as Listing 8-6.

        // Play the sound.
        Alarm.sound.play();
      }
}
```

8.5 Java's Collection Classes

Programs often need to have collections of similar objects. The alarm clock program we developed in the previous section is a prime example. Even with a collection of only four alarms, code such as `setAlarm` and `checkAndRingAlarms` got tedious. Furthermore, why should there be only four alarms? Why not 40 or 400 or even 4 million?

Four million alarms seems excessive, but other programs could easily have a collection of 4 million or more objects. Consider an inventory program for a large chain of stores, for example. When our collections of similar objects grow beyond four or five, we need better techniques than we used in `AlarmClock`.

Fortunately, Java provides a set of classes for maintaining collections of objects. These classes are used when objects in a collection need to be treated in a similar way: a collection of `Alarm` objects that need to be checked and perhaps rung, a collection of `Student` objects that need to be enrolled in a course, or a collection of `Image` objects that need to display on a computer monitor. The objects maintained by these collections are usually called the **elements** of the collection.

Java has three kinds of collections:

- A **list** is an ordered collection of elements, perhaps with duplicates. Because the list is ordered, you can ask for the element in position 5, for example.
- A **set** is an unordered collection of unique elements; duplicates are not allowed.
- A **map** is an unordered collection of associated **keys** and **values**. A key is used to find the associated value in the collection. For example, your student number is a key that is often used to look up an associated value, such as your address or grades.

Collection objects cannot hold primitive types, only objects. We'll discuss a way around that limitation in Section 8.5.4.

KEY IDEA

Collections hold objects, not primitives.

These collection classes are sophisticated, and covering all the details would require several chapters. Therefore, we will focus on constructing the objects; adding and removing elements, plus a few other useful methods; and processing all the elements (for example, checking all the `Alarm` objects to see if one should be rung). We'll look at one example of each kind of collection. We'll look at a list class first in some detail. We will go faster when we examine sets and maps because much of what we learn with lists will also apply to them.

The approach taken in this textbook assumes that you are using Java 5.0 or higher. Previous versions of Java have these classes, but they are more difficult to use without the advances made in Java 5.0

KEY IDEA

This section assumes you are using Java 5.0 or higher.

8.5.1 A List Class: `ArrayList`

A list is probably the most natural collection class to use for our `AlarmClock` program. It can hold any kind of object (sets and maps have some restrictions) and allows us to easily process all of the elements or to get just one.

There are two distinct ways to write a list class—`ArrayList` and `LinkedList`. Both are in the `java.util` package, meaning that you'll need to import from that package if you want to use the classes. `ArrayList` is the one we'll study here. By the end of Chapter 10, you will be able to write a simple version of `ArrayList`. By the end of your second computer science course, you should be able to write your own version of `LinkedList`.

Lists such as `ArrayList` keep its elements in order. It makes sense to speak of the first element or the last element. Like a `String`, an individual element is identified by its index—a number greater than or equal to zero and less than the number of elements in the list. The number of elements in the list can be obtained with the `size` query.

KEY IDEA

The `size` query returns the number of elements in the list.

Construction

The type of a collection specifies the collection's class and the class of object it holds. For example, one type that could hold a collection of `Alarm` objects is `ArrayList<Alarm>`. The type of objects held in the collection is placed between angle brackets. This type can be used to declare and initialize a variable, as follows:

```
ArrayList<Alarm> alarms = new ArrayList<Alarm>();
```

KEY IDEA

The type of the collection includes the type of objects to be stored in it.

A list of `Robot` objects and a list of `Person` objects would be created similarly:

```
ArrayList<Robot> workers = new ArrayList<Robot>();
ArrayList<Person> friends = new ArrayList<Person>();
```

Of course, if we're declaring instance variables, we would include the keyword `private` at the beginning of each line.

KEY IDEA

A collection allows you to access many objects using only one variable.

In the `AlarmClock` class shown in Listing 8-7, the declaration of the four `Alarm` instance variables in lines 10–13 can be replaced with the following line:

```
private ArrayList<Alarm> alarms = new ArrayList<Alarm>();
```

Furthermore, we are no longer limited to just four alarms.

FIND THE CODE

cho8/alarmsWithLists/

Adding Elements

The power of using a collection class becomes evident in the `setAlarm` method. In Listing 8-7, we devote lines 68–78 to assigning an alarm to one of the four instance variables—11 lines. Even so, we're limited to only four alarms. For each additional alarm, we need to add an instance variable and two more lines in the `setAlarm` method.

Using an `ArrayList` to store the alarms reduces lines 68–78 to a single line:

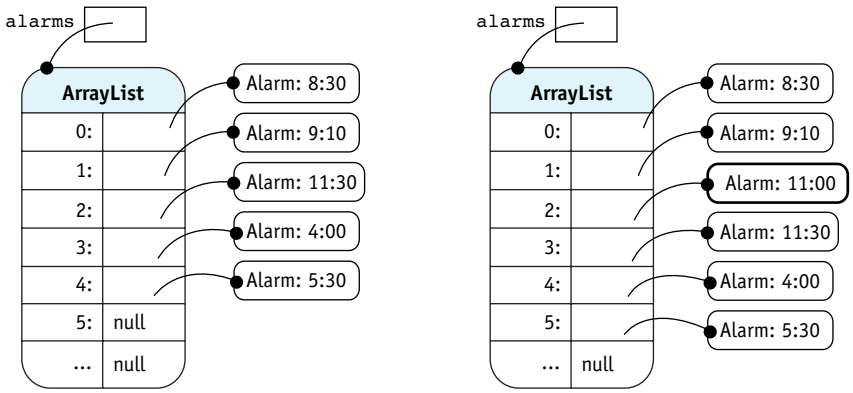
```
this.alarms.add(theAlarm);
```

Furthermore, we can now have an almost unlimited number of alarms.

The `add` method just shown adds the new alarm to the end of the list. An overloaded version of `add` allows you to state the index in the list where the alarm should be added. Like `String`s, an `ArrayList` numbers the positions in its list starting with 0. Therefore, the following line adds a new alarm in the third position:

```
this.alarms.add(2, theAlarm);
```

The alarms at indices 0 and 1 come before it. Objects at indices 2 and larger are moved over by one position to make room for the new object. Figure 8-18 illustrates inserting a new `Alarm` for 11:00 at index 2.



Before inserting an `Alarm` at `index 2`

After inserting an `Alarm` at `index 2`

(figure 8-18)

Inserting an `Alarm` *into an* `ArrayList` *at index 2*

The index for `add` must be in the range `0..size()`. Positions can't be skipped when adding objects. For example, you can't add an object at index 2 before there is data at indices 0 and 1. Doing so results in an `IndexOutOfBoundsException`.

Getting, Setting, and Removing Elements

A single element of the collection can be accessed using the `get` method and specifying the object's index. For example, to get a reference to the third alarm (which is at index 2 because numbering starts at 0), write the following statements:

```
Alarm anAlarm = this.alarms.get(2);
anAlarm.ring();              // do something with the alarm
```

As with any other method that returns a reference, you aren't required to assign the reference to a variable before calling a method. We could condense the previous two statements to a single line:

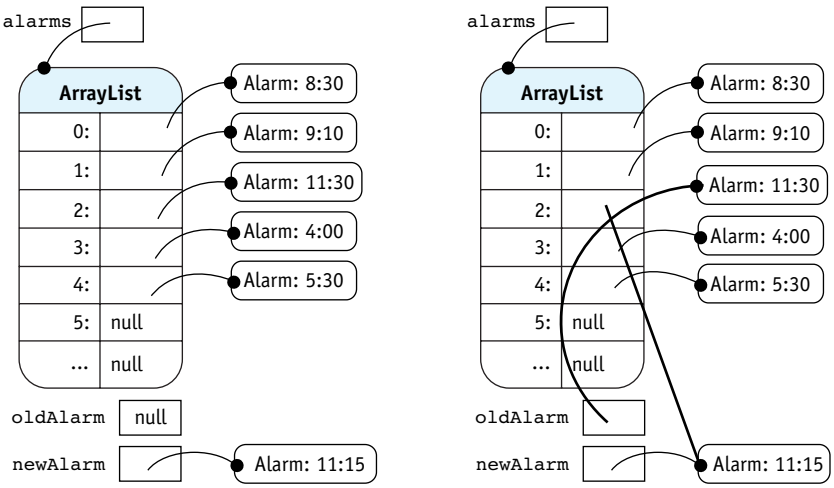
```
this.alarms.get(2).ring();
```

An element can be replaced using the `set` method. Its parameters are the index of the element to replace and the object to put there. For example, Figure 8-19 illustrates the change made by the following code fragment:

```
Alarm oldAlarm = null;
Alarm newAlarm = new Alarm(11, 15, "Meeting with Mohamed");
oldAlarm = this.alarms.set(2, newAlarm);
```

(figure 8-19)

Effects of the `set` *method*



Notice that the element at index 2 now refers to the new alarm. The `set` method returns a reference to the element that is replaced, which is assigned to `oldAlarm`.

An element can be removed from the `ArrayList` with the `remove` method. Its only argument is the index of the element to remove. After removing the element, any elements in subsequent positions are moved up to occupy the now open position—the opposite of what `add` does. Like `set`, `remove` returns a reference to the removed element.

Other Useful Methods

There are many other methods in the `ArrayList` class and its superclasses. Table 8-1 lists the name and purpose of some of the most useful methods. `E` represents the type of elements stored in this particular collection.

The `contains` and `indexOf` methods depend on the element's class overriding the equals method to test for equivalence. As noted in Section 8.2.4, we don't have the tools to do this yet for the classes we write. Provided classes such as `String`, `DateTime`, and others should meet this requirement.

Method	Purpose
`boolean add(E elem)`	Add the specified element to the end of this list. Return `true`.
`void add(int index, E elem)`	Insert the specified element at the specified index in this list. `0 ≤ index < size()`.
`void clear()`	Remove all of the elements from this list.
`boolean contains(Object elem)`	Return `true` if this list contains the specified element.
`E get(int index)`	Return the element at the specified index. `0 ≤ index < size()`.
`int indexOf(Object elem)`	Search for the first element in this list that is equal to `elem`, and return its index or `-1` if there is no such element in this list.
`boolean isEmpty()`	Return `true` if this list contains no elements.
`E remove(int index)`	Remove and return the element at the given index. `0 ≤ index < size()`.
`E set(int index, E elem)`	Replace the element at the given position in this list with `elem`. Return the old element. `0 ≤ index < size()`.
`int size()`	Return the number of elements in this list.

(table 8-1)

Some of the most useful methods in the `ArrayList` *class.* `E` *is the type of the elements*

Processing All Elements

The last detail needed to replace the four `Alarm` variables with a list is checking each alarm to see if it's time to ring it. In Listing 8-7, we did this in lines 47–52. Each line calls a helper method to check one of the alarms. That means 4 alarms, 4 lines of code; 400 alarms, 400 lines of code.

There are three distinct ways[3] to process all of the elements in an `ArrayList`. We've already seen the basic tools for one of them: the `get` and `size` methods. We can use them in a `for` loop to get each element in turn:

```
1 private void checkAndRingAlarms(DateTime currTime)
2 { for (int index = 0; index < this.alarms.size(); index++)
3   { Alarm anAlarm = this.alarms.get(index);
4     this.checkOneAlarm(anAlarm, currTime);
5   }
6 }
```

PATTERN

Process All Elements

[3] The third way uses iterators, a topic we won't be covering in this textbook.

These six lines of code completely replace `checkAndRingAlarms` in lines 47–52 of Listing 8-7. Furthermore, this code will work for almost[4] any number of alarms—from zero on up.

A loop to process all of the elements in a collection is so common that Java 5.0 introduced a special version of the `for` loop just to make these situations easier. It is sometimes called a **foreach** loop—the body of the loop executes once for each element in the collection.

Using a *foreach* loop to process each alarm results in the following method:

Process All Elements

```
private void checkAndRingAlarms(DateTime currTime)
{ for(Alarm anAlarm : this.alarms)
  { this.checkOneAlarm(anAlarm, currTime);
  }
}
```

A template for the *foreach* loop is as follows:

```
for(«elementType» «varName» : «collection»)
{«statements using varName»
}
```

The statement includes the keyword `for`, but instead of specifying a loop index, the `for each` loop declares a variable, *«varName»*, of the same type as the objects contained in *«collection»*. The variable name is followed with a colon and the collection that we want to process. *«varName»* can only be used within the body of the loop.

A version of `AlarmClock` that uses an `ArrayList` is shown in Listing 8-11. Note that changes are shown in bold. Documentation is identical to Listing 8-7, so it is omitted.

FIND THE CODE

cho8/alarmsWithLists/

Listing 8-11: *The* `AlarmClock` *class implemented with an* `ArrayList`

```
1  import becker.util.DateTime;
2  import becker.util.Utilities;
3  import java.util.ArrayList;
4
5  public class AlarmClock extends Object
6  {
7    // A list of alarms.
8    private ArrayList<Alarm> alarms = new ArrayList<Alarm>();
9
10   private int numAlarmsLeft = 0;
```

[4] We don't say `ArrayList` will handle any number because eventually your computer would run out of memory to store them all.

Listing 8-11: *The* `AlarmClock` *class implemented with an* `ArrayList` (continued)

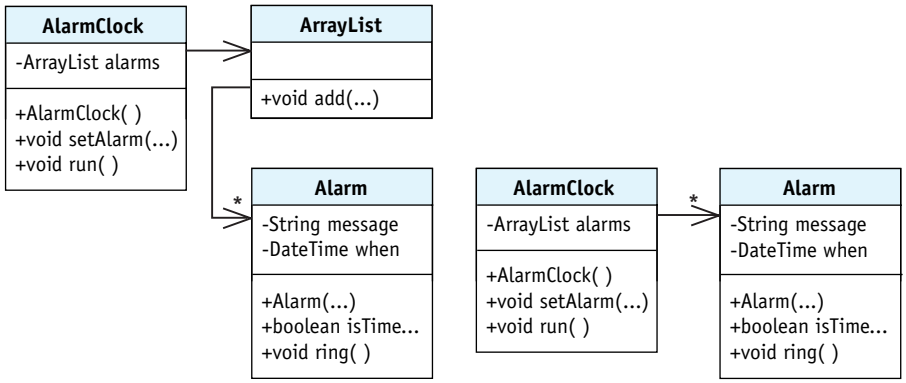
```
11    private final boolean TESTING;
12
13    public AlarmClock(boolean test)
14    { // Same as Listing 8-7.
15    }
16
17    public void run(int secPerSec)
18    { // Same as Listing 8-7.
19    }
20
21    private void checkAndRingAlarms(DateTime currTime)
22    { for(Alarm anAlarm : this.alarms)
23      { this.checkOneAlarm(anAlarm, currTime);
24      }
25    }
26
27    private void checkOneAlarm(Alarm alarm, DateTime currTime)
28    { // Same as Listing 8-7.
29    }
30
31    public void setAlarm(int hr, int min, String msg)
32    { Alarm theAlarm = new Alarm(hr, min, msg);
33      this.alarms.add(theAlarm);
34      this.numAlarmsLeft++;
35    }
36
37    public static void main(String[] args)
38    { // Same as Listing 8-7.
39    }
40  }
```

Class Diagrams

Someone drawing a class diagram for `AlarmClock`, as shown in Listing 8-11, would probably draw a diagram as shown in Figure 8-20a. However, collection classes like `ArrayList` appear so often in Java programs and their function is so well known that most programmers prefer to draw the abbreviated class diagram shown in Figure 8-20b.

(figure 8-20)

Class diagrams



a. Full class diagram

b. Abbreviated class diagram

8.5.2 A Set Class: `HashSet`

Like a list, a set also manages a collection of objects. There are two important differences:

- A set does not allow duplicate elements. Sets ignore attempts to add an element that is already in the set.
- The elements are not ordered. None of the methods in `HashSet` take an index as an argument.

KEY IDEA

Sets do not allow duplicates.

These restrictions don't affect the `AlarmClock` class—each alarm is unique and individual alarms are not important; they are all processed as a group. In fact, changing `ArrayList` to `HashSet` in line 8 of Listing 8-11 is all that is needed to convert that program to use a set.

LOOKING AHEAD

Processing files is a major topic of Chapter 9.

So how might we exploit the specific properties of a set? We could use it, for example, to count the number of unique strings in a file. About two dozen lines of code are enough to discover that William Shakespeare's play *Hamlet* contains 7,467 unique "words." (Words is quoted because the program doesn't remove punctuation or numbers, meaning that "merry" and "merry?" are considered different words.)

Construction

We'll use an instance of the `HashSet` class to count the words. An instance of `HashSet` is constructed just like `ArrayList`—specify the type of the elements you want it to manage in angle brackets. In this case, we'll store our words as strings.

```
HashSet<String> words = new HashSet<String>();
```

Useful Methods

Words can be added to this set with the `add` method. If the word is already there, it will be ignored.

To add many words, we should read them from a file—the topic of Section 9.1. Until then, we can add some words from *Hamlet* manually:

```
words.add("to");
words.add("be");
words.add("or");
words.add("not");
words.add("to");
words.add("be");
```

FIND THE CODE

cho8/collections/

The `size` method returns the number of elements in the set. Given the previous six calls to `add`, `size` would return 4.

The word "not" could be removed with the statement `words.remove("not")`. In general, an object is removed from the set by passing the object to the `remove` method.

The `contains` method will return `true` if the set contains the given object and `false` otherwise. Other useful methods are summarized in Table 8-2.

(table 8-2)

Some of the most useful methods in the `HashSet` *class (*`E` *is the type of the elements)*

Method	Purpose
`boolean add(E elem)`	Add the specified element to this set. Return `true` if the element was already present.
`void clear()`	Remove all of the elements from this set.
`boolean contains(Object elem)`	Return `true` if this set contains the specified element.
`boolean isEmpty()`	Return `true` if this set contains no elements.
`boolean remove(Object elem)`	Remove the specified element from this set, if present. Return `true` if the element was present.
`int size()`	Return the number of elements in this set.

Processing All Elements

We can print all of the words in the set using a `for each` loop, just as we processed all of the elements in the `ArrayList` earlier.

KEY IDEA

A set's `for each` *loop works the same way as for a list.*

```
for (String w : words)
{ System.out.print(w + "  ");
}
```

PATTERN

Process All Elements

Executing this loop after adding the first six words of Hamlet's speech would yield "to," "be," "or," and "not." The order in which they are printed is *not* specified.

Limitations

`HashSet` uses a technique known as **hashing**, in which elements are stored in an order defined by the element's `hashCode` method. The hash code is carefully constructed to make operations such as `contains` and `remove` faster than for an `ArrayList`. When the elements are printed, however, they appear in an order that seems random.

`hashCode` is inherited from the `Object` class. As defined there, no two objects are considered equal or equivalent. If two elements in your set should be considered equivalent (for example, two different date objects both representing the same date), the `equals` and `hashCode` methods must both be overridden. Unfortunately, overriding `hashCode` is beyond the scope of this textbook. However, you should have no problem using `HashSet` if you either use it with a set of unique objects or use it with provided classes, such as `String` or `DateTime`.

8.5.3 A Map Class: `TreeMap`

KEY IDEA

A map associates a key with a value. Use the key to look up the value.

A map is a collection of associated keys and values. A key is used to find the associated value in the collection. For example, we could associate the names of our friends (the keys) with their phone numbers (the values), as shown in Figure 8-21.

(figure 8-21)

Key-value pairs

Key	Value
Sue	578-3948
Fazila	886-4957
Jo	1-604-329-1023
Don	578-3948
Rama	886-9521

With these associations between keys and values, we can ask questions such as "What's the phone number for Don?" We use the key, "Don," to look up the associated value, "578-3948."

KEY IDEA

The keys in any given map must be unique.

Notice that all the keys are unique; that's a fundamental requirement of a map. If we have two friends named "Don" we must distinguish between them, perhaps by adding initials or last names. However, the associated values do not need to be unique. In this example, Don and Sue both appear in the mapping even though they have the same phone number.

Java provides two classes implementing a map, `TreeMap` and `HashMap`. Each one has different advantages and disadvantages. `HashMap`s have the advantage of being somewhat faster but require a correct implementation of the `hashCode` method. On the other hand, `TreeMap`s keep the keys in sorted order but require a way to order the elements. We'll use a `TreeMap` to build a simple phone book.

Construction

When declaring and constructing a `TreeMap` object, the types for both the keys and the values must be specified. For our simple phone book, we'll use `String`s for both the keys and the values:

```
TreeMap<String, String> phoneBook =
                                new TreeMap<String, String>();
```

Although this example happens to use strings for both keys and values, that need not be the case. The types of the keys and values are often different and must be a reference type—not a primitive type like `int`, `double`, or `char`.

How can you figure out that `TreeMap` needs two types to define it but that `ArrayList` and `HashSet` require only one? Look at the class documentation. Figure 8-22 shows the beginning of the online documentation for `TreeMap`, which includes `TreeMap<K, V>` in large type. The two capital letters between the angle brackets indicate that two types are needed when a `TreeMap` is constructed. Finding out that `K` stands for the type of the key and `V` stands for the type of the value is, unfortunately, not as easy to figure out from the documentation.

There is one restriction on the type of the key. Because `TreeMap` keeps the keys in sorted order, it needs a way to compare them. It relies on the key's class to implement the `Comparable` interface. The keys are then known to have a `compareTo` method. `String` and `DateTime` both implement the interface and can be used as keys.

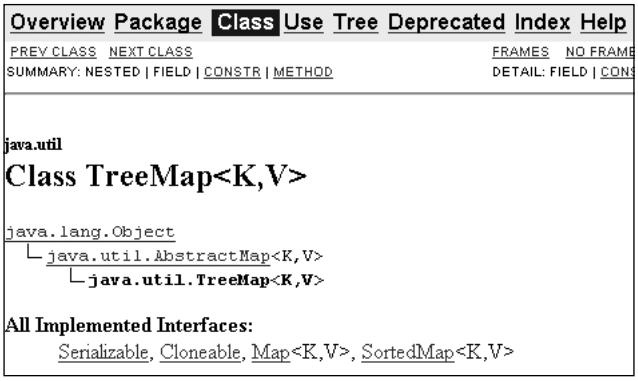
You can tell if a class implements `Comparable` by looking at the "All Implemented Interfaces" line in the documentation. You can see an example of this line in Figure 8-22. Also, if you look at the documentation for `Comparable`, it will list the classes in the Java library that implement it.

LOOKING AHEAD

Writing your own classes that implement the `Comparable` *interface will be discussed in Section 12.5.1.*

(figure 8-22)

Part of the online documentation for `TreeMap`



Useful Methods

Pairs are added to a map with the `put` method. It takes a key and a value as arguments:

```
phoneBook.put("Sue", "578-3948");
phoneBook.put("Fazila", "886-4957");
```

If the key already exists in the map, the value associated with that key will be replaced by the new value.

A value can be retrieved with the `get` method. The key of the desired value is passed as an argument. For example, after executing the following line:

```
String number = phoneBook.get("Sue");
```

the variable `number` will contain "578-3948" (assuming the associations shown in Figure 8-21). It's similar to accessing an element in a list except that instead of specifying the element's index, you specify the element's key.

The `remove` method takes a key as its only argument and removes both the key and its associated value.

Like a list and a set, a map has `isEmpty`, `clear`, and `size` methods. Instead of `contains`, it has two methods: `containsKey` and `containsValue`, which both return a Boolean result. These methods are summarized in Table 8-3.

Method	Purpose
`void clear()`	Remove all of the key-value pairs from this mapping.
`boolean containsKey (Object elem)`	Return `true` if this mapping contains the specified key.
`boolean containsValue (Object elem)`	Return `true` if this mapping contains the specified value.
`V get(Object key)`	Return the value associated with the specified key.
`boolean isEmpty()`	Return `true` if this mapping contains no elements.
`Set<K> keySet()`	Return a set containing the keys in this mapping.
`V put(K key, V value)`	Associate the specified key with the specified value in this mapping. Return the value previously associated with the key or `null` if there wasn't one.
`V remove(Object key)`	Remove and return the value associated with the specified key, if it exists. Return `null` if there was no mapping for the key.
`int size()`	Return the number of key-value pairs in this mapping.

(table 8-3)

Some of the most useful methods in the `TreeMap` *class (*`K` *is the type of the keys;* `V` *is the type of the values)*

Processing All Elements

Processing all the elements in a map is more complicated than a list or a set because each element is a pair of objects rather than just one thing.

One approach is to use the `keySet` method to get all of the keys in the map as a set. We can then loop through all of the keys using the `for each` loop. As part of the processing, we can also get the associated value, as shown in the following example:

```
// print the phoneBook
for (String key : phoneBook.keySet())
{ System.out.println(key + "=" + phoneBook.get(key));
}
```

Process All Elements

Completed Program

The completed telephone book program is shown in Listing 8-12. It uses a `Scanner` object in 27 and 30 to obtain a name from the program's user. Using `Scanner` effectively is one of the primary topics of the next chapter.

FIND THE CODE

cho8/collections/

Listing 8-12: *An electronic telephone book*

```
1  import java.util.*;
2
3  /** An electronic  telephone book.
4  *
5  *  @author Byron Weber Becker */
6  public class MapExample extends Object
7  {
8    public static void main(String[] args)
9    { // Create the mapping between names and phone numbers.
10     TreeMap<String, String> phoneBook =
11                               new TreeMap<String, String>();
12
13     // Insert the phone numbers.
14     phoneBook.put("Sue", "578-3948");
15     phoneBook.put("Fazila", "886-4957");
16     phoneBook.put("Jo", "1-604-329-1023");
17     phoneBook.put("Don", "578-3948");
18     phoneBook.put("Rama", "886-9521");
19
20     // Print the phonebook.
21     for (String k : phoneBook.keySet())
22     { System.out.println(k + "=" + phoneBook.get(k));
23     }
24
25     // Repeatedly ask the user for a name until "done" is entered.
26     // Scanner is discussed in detail in Chapter 9.
27     Scanner in = new Scanner(System.in);
28     while (true)
29     { System.out.print("Enter a name or 'done': ");
30       String name = in.next();
31
32       if (name.equalsIgnoreCase("done"))
33       { break;   // Break out of the loop.
34       }
35
36       System.out.println(name + ":" + phoneBook.get(name));
37     }
38   }
39 }
```

8.5.4 Wrapper Classes

What if we want to store integers or characters or some other primitive type in one of the collection classes? For example, we might need a set of the prime numbers (integers that can only be divided evenly by 1 and itself). If we write

```
HashSet<int> primeNumbers = new HashSet<int>();
```

the Java compiler will give us a compile-time error, perhaps with the cryptic message "unexpected type." The problem is that the compiler is expecting a reference type—the name of a class—between the angle brackets. `int`, of course, is a primitive type.

We can get around this by using a **wrapper class**. It "wraps" a primitive value in a class. A simplified wrapper class for `int` is as follows:

```
public class IntWrapper extends Object
{ private int value;

  public IntWrapper(int aValue)
  { super();
    this.value = aValue;
  }

  public int intValue()
  { return this.value;
  }
}
```

Fortunately, Java provides a wrapper class for each of the primitive types: `Integer`, `Double`, `Boolean`, `Character`, and so on. These are in the `java.lang` package, which is automatically imported into every class.

We can use these built-in wrapper classes to construct a set of integers:

```
HashSet<Integer> primes = new HashSet<Integer>();
```

The Java compiler will automatically convert between an `int` and an instance of `Integer` when using `primes`. For example, consider the program in Listing 8-13. In lines 12–17, the `add` method takes an `int`, not an instance of `Integer`. The `contains` method in line 25 is the same. Before Java 5.0 the programmer needed to manually include code to convert between primitives and wrapper objects.

KEY IDEA

Java 5.0 automatically converts between primitive values and wrapper classes.

FIND THE CODE

cho8/collections/

Listing 8-13: *A program to help classify prime numbers*

```
1  import java.util.*;
2
3  /** Help the user find out if a number is prime.
4  *
5  *  @author Byron Weber Becker */
6  public class WrapperExample extends Object
7  {
8    public static void main(String[] args)
9    { HashSet<Integer> primes = new HashSet<Integer>();
10
11     // The prime numbers we know.
12     primes.add(2);
13     primes.add(3);
14     primes.add(5);
15     primes.add(7);
16     primes.add(11);
17     primes.add(13);
18
19     // Help the user classify numbers.
20     // Scanner is discussed in detail in Chapter 9.
21     Scanner in = new Scanner(System.in);
22     System.out.print("Enter a number: ");
23     int num = in.nextInt();
24
25     if (primes.contains(num))
26     { System.out.println(num + " is prime.");
27     } else if (num <= 13)
28     { System.out.println(num + " is not prime.");
29     } else
30     { System.out.println(
31             num + " might be prime; it's too big for me to know.");
32     }
33   }
34 }
```

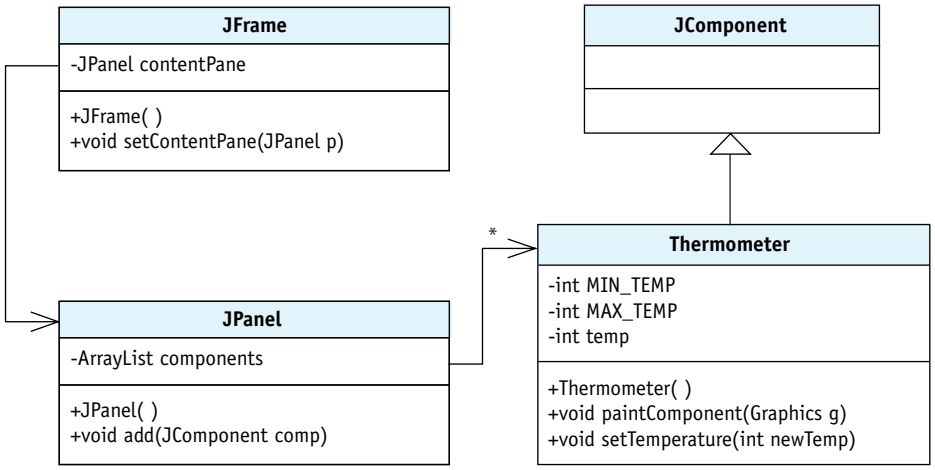
8.6 GUIs and Collaborating Classes

Programs with graphical user interfaces almost always use collaborating classes in two ways. Collaborating classes makes these programs easier to understand, write, debug, and maintain.

8.6.1 Using Libraries of Components

First, GUIs are constructed from a library of components. You've already used a number of these: `JFrame`, `JPanel`, `JComponent`, `JButton`, and so on. `JFrame` typically collaborates with `JPanel` to organize a number of components to display. `JPanel` collaborates with one or more components such as `JButton` to display information in the right format.

For example, consider the program written in Section 6.7. It displays three temperatures using a custom thermometer component. The class diagram in Figure 8-23 shows that the `JFrame` has-a `JPanel` to help organize the components it displays. The `JPanel` has-a `Thermometer` to actually display a temperature. In fact, the `JPanel` has a number of `Thermometer` objects. Finally, the `Thermometer` class is-a `JComponent`. The two classes collaborate to provide a standard set of services with the customized appearance provided by `paintComponent`.



(figure 8-23)

Simplified class diagram of the thermometer program from Section 6.7

The collaboration between these classes allows each to have a specific focus. Focused classes are easier to understand, write, debug, and maintain.

8.6.2 Introducing the Model-View-Controller Pattern

Collaborating classes are also used with modern graphical user interfaces via the Model-View-Controller pattern. This pattern splits a program into three collaborating classes or groups of classes.

- The **model** is responsible for modeling the current problem. For example, the `AlarmClock` class we wrote earlier models the problem of keeping the current time and determining when to ring the alarms, but has little to do with displaying anything to the user.

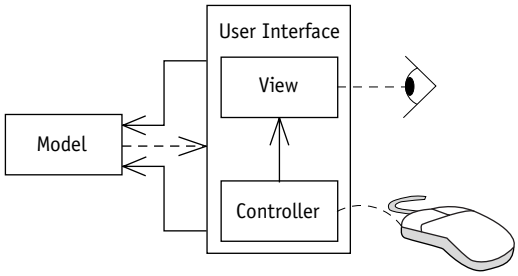
- The **view** shows relevant information in the model to the user. In an alarm clock program, the view is the class (or group of classes) that show the user what time it is and when the alarms are due to ring. This is information that the view obtains from the model.
- The **controller** is responsible for gathering input from the user and using it to modify the model, for example, by changing the current time or the time when an alarm is due to ring. When the controller changes the model, the view should also change to show the new information.

The view and the controller work together closely and are known as the user interface.

The relationships between these three groups of classes are shown in Figure 8-24. The eye represents the user observing the model via the view. The mouse represents the user changing the model via the controller. The arrow between the controller and the view indicates that the controller may call methods in the view, but the view has no need to interact with the controller. The two arrows from the user interface to the model indicate that both the view and the controller will have reason to call methods in the model. The last arrow is dotted to indicate that the model will call methods in the user interface, but in a limited and controlled way.

(figure 8-24)

The view and controller interact with the user and the model



The Model-View-Controller pattern will be explored fully in Chapter 13, Graphical User Interfaces.

8.7 Patterns

8.7.1 The Has-a (Composition) Pattern

Name: Has-a (Composition)

Context: A class is getting overly complex.

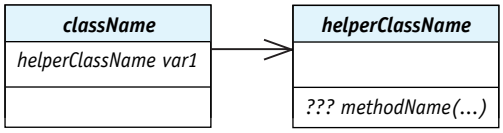
Solution: Identify one or more subsets of methods and instance variables that form a cohesive concept. Make each subset into a separate helper class that the original class can use to solve the overall problem. The original class will likely have one or more instance variables referring to instances of the helper classes. A general pattern is shown in the following code:

```
public class «className»...
{ private «helperClassName» «var1»;

  public «className»(...)
  { // initialize the helper class
    this.«var1» = ...
  }

  ... «methodName»(...)
  { // use the helper class
    ... this.«var1».«methodName»...
  }
}
```

This pattern results in the class diagram shown in Figure 8-25.



(figure 8-25)

Class diagram resulting from Has-a (composition) pattern

Consequences: The individual classes will become smaller and more focused on a particular task, making them easier to write, test, debug, and modify.

Related Pattern: The Has-a pattern is a special case of the Instance Variable pattern, where the instance variable is an object reference.

8.7.2 The Equivalence Test Pattern

Name: Equivalence Test

Context: A method is required to test whether two objects are equivalent to each other in value.

Solution: Write a method, `isEquivalent`, that takes one of the objects as an argument and tests all the relevant fields for equivalence. In general,

```
public class «className» ...
{ private «primitiveType» «relevantField1»
  ...
```

```
    private «referenceType» «relevantField2»
    ...

    public boolean isEquivalent(«className» other)
    { return other != null &&
          this.«relevantField1» == other.«relevantField1» &&
          ...
          this.«relevantField2».isEquivalent(
                                    other.«relevantField2») &&
          ...;
    }
}
```

where == is used for primitive fields and either `isEquivalent` or `equals` is used for objects.

Consequences: The method will determine whether two objects are equivalent by testing all the relevant fields for equivalence. Using `isEquivalent` may give unexpected results with methods such as `contains` in Java's collection classes. Those classes assume that `equals` has been properly overridden, but that requires concepts first discussed in Chapter 12.

Related Patterns:

- The Equivalence Test pattern is a specialization of the Predicate pattern.
- The Equals pattern (Section 12.7.3) is a better choice than this pattern, once the details of implementing `equals` have been mastered.

8.7.3 The Throw an Exception Pattern

Name: Throw an Exception

Context: Your method detects an exceptional event that is most appropriately handled by the method's client.

Solution: Create an exception object to report details of the exceptional event and use Java's `throw` statement, as follows:

```
if («testForErrorCondition»)
{ throw new «exceptionName»(«stringDescription»);
}
```

Consequences: Clients of the called method are informed of the exceptional event and may be able to recover if the exception is handled. In the case of a checked exception such as `FileNotFoundException`, clients must either handle the exception or declare that they throw it.

Related Pattern: Thrown exceptions may be caught and handled with the Catch an Exception pattern.

8.7.4 The Catch an Exception Pattern

Name: Catch an Exception

Context: You are calling a method that can throw an exception. You want to handle the exception to protect the program's users from the consequences of the problem.

Solution: Catch the exception using a `try-catch` statement and the following template:

```
try
{ «statements that may throw an exception»
} catch («exception_1 e»)
{ «statements to handle excepton_1»
} catch («exception_2 e»)
{ «statements to handle excepton_2»
}
```

More `catch` clauses can also be added.

Consequences: Exceptions that are thrown by statements within the `try` clause are handled in the matching `catch` clause, if one exists. If there is no matching `catch` clause, the exception is propagated to the caller. The `catch` clauses are evaluated in order, with the result that the most specific exceptions should appear first and the most general exceptions later.

Related Pattern: Exceptions are thrown with the Throw an Exception pattern.

8.7.5 The Process All Elements Pattern

Name: Process All Elements

Context: The same operation must be performed on all the objects in a collection.

Solution: Store all of the relevant objects in an `ArrayList`, `HashSet`, `TreeMap`, or similar collection object. Use one of the following forms to retrieve all of the objects one at a time to perform the required operation.

The exact form of the pattern depends on the type of collection. For a list, the following code may be used.

```
for (int i = 0; i < «collection».size(); i++)
{ «elementType» element = «collection».get(i);
  «statements to process element»
}
```

The following form, available in Java 5.0 and later, is applicable to both lists and sets:

```
for («elementType» element : «collection»)
{ «statements to process element»
}
```

If the collection is an instance of a mapping, such as `TreeMap`, a slight variant of the preceding template is required:

```
for («keyType» key : «collection».getKeySet())
{ «valueType» value = «collection».get(key);
  «statements to process key and value»
}
```

Consequences: Using a collection to handle multiple objects of the same type can make lots of code much simpler, especially code that processes each of the elements in turn.

Related Patterns:

- The Process All Elements pattern is related to the Process File pattern (Section 9.9.3) and will be recast using arrays in Section 10.8.1.
- Processing all the characters in a `String` is similar to this pattern, although the `for each` loop is not applicable in that setting.

8.8 Summary and Concept Map

Structuring programs so that classes work together to solve a problem is a great idea. By delegating work to other classes, each class can be simpler and more focused on one particular idea. This makes the program easier to understand, write, debug, and maintain.

8.9 Problem Set

Written Exercises

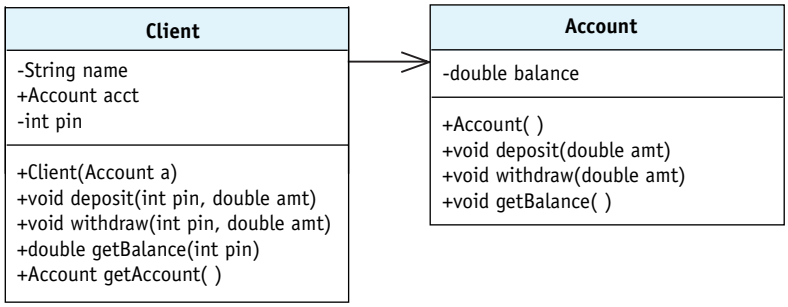
8.1 A book has a title, an author, and a call number. A library patron has a name and an ID number, and may or may not have a book checked out.

a. Draw a class diagram for `Patron` where only a single book may be checked out at any given time. Include the methods necessary to check out a book, and print which book (if any) the patron has.

b. Elaborate the class diagram from part (a) so that a patron may have zero or more books checked out.

c. Draw a class diagram including a `Library` object. A library, of course, has many patrons and many books. Include the methods required to check out a book to a patron, given the patron's ID number and the book's call number. Also include the methods required to list which books a patron has, given the patron's ID number.

8.2 Consider the class diagram shown in Figure 8-26. It shows one possible relationship between a bank's client and the client's account. Each client has a personal identification number (PIN) to use in accessing his account. The methods requiring a PIN do nothing if the PIN doesn't match the one stored for the client.



(figure 8-26)

Partial class diagram for a bank

Suppose you are a programmer working on the `Bank` class, which contains references to objects representing all of the bank's clients. Explain three ways in which you could transfer money from one client's account to your account without knowing the client's PIN. In each case, explain how this security hole could be closed.

Assume the programmer who implemented `Client` and `Account` knows nothing of the dangers of using aliases.

Programming Exercises

8.3 Consider the program in Listing 8-2. According to the surrounding text, it was used to find that Luke was 5,009 days old on the day that paragraph was written. Modify the program to print the date the paragraph was written.

8.4 Consider a `FuelBot` class. It extends `Robot` and uses a `FuelTank`. Each time the robot moves, it will use 0.4 liters of fuel from the tank. The tank holds 3 liters of fuel when it is full. If the robot comes to an intersection with a `Thing` on it, refill its tank. If the robot ever runs out of fuel, it breaks.

a. Draw a class diagram that includes the `Robot`, `FuelBot`, and `FuelTank` classes. Include variables and methods.

b. Implement `FuelBot`. Write a `main` method to test your class.

c. Make a simple game by overriding the `keyTyped` method to allow the user to control the robot (see Listing 7-5). Scatter gas stations around the city. Put a `Thing` with a different color at a random location to serve as the goal. Can the robot reach it before running out of fuel? Use the robot's `setLabel` command to display the amount of fuel remaining as a percentage.

8.5 A normal playing card has a rank (one of Ace, 2, 3, 4, …, 10, Jack, Queen, King) and a suit (one of Diamonds, Clubs, Hearts, or Spades). Players in a card game usually have a "hand" consisting of several cards. For a game, a player will want to know the value of his or her hand. The value is calculated by summing the rank of each card, where Ace is 1, Jack is 11, Queen is 12, and King is 13. The number cards have their number as their value. There is one exception: the Ace of the trump suit is valued at 14. The trump suit is specified when the hand is created.

a. Draw a class diagram of the `Hand` and `Card` classes. Assume that a hand consists of at most four cards.

b. Implement `Hand` and `Card`. Write a `main` method to test the hand's value calculation. Assume the hand consists of at most four cards.

c. Draw a class diagram of the `Hand` and `Card` classes. Use an `ArrayList` or `HashSet`.

d. Implement `Hand` and `Card`. Write a `main` method to test the hand's value calculation. Don't make any assumptions about the number of cards in a hand.

8.6 In a simplified version of the game of Monopoly, a player may have between 0 and 4 properties. Each property has a name, a purchase price, and a rent. The purchase price is typically between $60 and $400, and the rent is typically between 10 percent and 15 percent of the purchase price. A player needs to calculate the total of the purchase price of its properties, return whether it owns a specified property, and return the rent for a property. Properties are identified to these methods by their names.

a. Draw a class diagram showing the `Player` and `Property` classes.

b. Implement the classes without using a collection class. Include a `main` method in `Player` to test the class.

c. Implement the classes, removing the restriction of owning no more than four properties. Include a `main` method in `Player` to test the class.

8.7 The `Person` class in Listing 8-3 uses a `String` to store the person's mother and father. Why not use an instance of `Person`? After all, mothers and fathers are persons.

Revise the class using this idea. Provide a second `Person` constructor for when parents aren't known; it sets `mother` and `father` to `null`. Include a `toString` method in `Person` that returns the person's name.

Write a `main` method that creates objects for seven people—you, your parents, and your grandparents. Make up any data you don't know. Print the results of the `toString` method for each of the seven `Person` objects.

a. Replace `toString` with a method that prints "[*«name»*: m = *«name»* f = *«name»*]", where each *«name»* is filled in with the appropriate name. If either the mother or the father is `null`, print "unknown" for the name.

b. Modify the `toString` method from part (a). Instead of printing the name of the mother and father, call that person's `toString` method. As before, if the mother or father is `null`, print "unknown."

Programming Projects

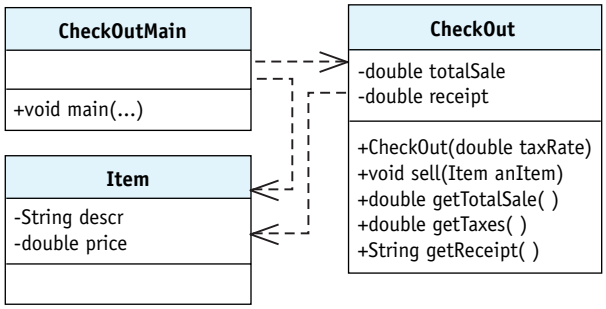
LOOKING BACK

Dotted lines between classes mean a class uses another class but doesn't hold an instance variable to it. See Section 8.2.2.

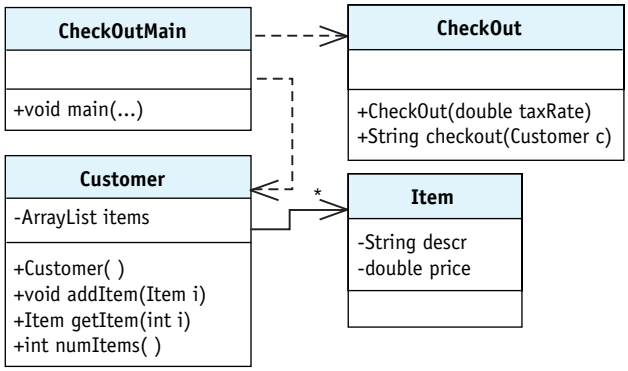
8.8 Implement a program to run at a checkout counter in a store. The `main` method will create the `CheckOut` object and then give it a number of `Items` to check out. The `CheckOut` object will be able to produce an itemized receipt. (*Hint*: You can use an `ArrayList` or a `String` to build the receipt as items are sold.) A partial class diagram is shown in Figure 8-27.

(figure 8-27)

Partial class diagram for checking out items at a store



8.9 A class diagram for another store's checkout counter is shown in Figure 8-28. Write a program where the `main` method creates a `CheckOut` object and a `Customer` object, complete with a number of `Items` to buy. Call the `checkout` method to generate an itemized receipt. Print the receipt.



(figure 8-28)

Another partial class diagram for checking out items at a store

8.10 A checkbook has an opening balance and zero or more checks. Each check has a check number, the name of the person or company who can cash it, an amount, and a memo. A checkbook should be able to return information about a check, given its check number. It should also be able to give the current balance.

a. Would you implement this program using a list, set, or map? Why?

b. Without prejudicing your answer to part (a), draw a class diagram assuming the program uses a map.

c. Implement the classes as shown in part (b). Include a `main` method in the `Checkbook` class to test it.

8.11 Modify the prime number program in Listing 8-13 to include all the prime numbers less than 10,000. Obviously, you want the program to calculate these values. The most straightforward approach is to consider every integer between 2 and 10,000. If the integer is prime, add it to the set. How do you test if the integer *i* is prime? Divide *i* by every number between 2 and *i*-1. If the remainder is 0 for any of them, *i* is *not* prime. *The* `%` operator yields the remainder of an integer division.

An equivalent test that is more efficient is to only divide by the prime numbers less than *i*—that is, the numbers that are already in your set of prime numbers. Use this more efficient approach to calculate the prime numbers.

8.12 Write a main method that repeatedly asks the user for the URL of a sound file, downloads it, and plays it. You will need to use the `newAudioClip` method in the `java.applet.Applet` class along with the `java.applet.AudioClip` and `java.net.URL` classes, among others. Unfortunately, these classes won't play `.mp3` files. There is a `.wav` file you may test your program with at `www.learningwithrobots.com/downloads/WakeupEverybody.wav`. It's a large file but only plays for a few seconds.

The sample solution is less than 30 lines of code. You will need to handle at least one checked exception.