We want the object created to be an independent copy of `original`. That would not happen if we had used the following instead:

```
public Person(Person original) //Unsafe
{
    if (original == null )
    {
        System.out.println("Fatal error.");
        System.exit(0);
    }
    name = original.name;
    born = original.born; //Not good.
    died = original.died; //Not good.
}
```

Although this alternate definition looks innocent enough and may work fine in many situations, it does have serious problems.

Assume we had used the unsafe version of the copy constructor instead of the one in Display 5.19. The "Not good." code simply copies references from `original.born` and `original.died` to the corresponding arguments of the object being created by the constructor. So, the object created is not an independent copy of the `original` object. For example, consider the code

```
Person original =
        new Person("Natalie Dressed",
        new Date("April", 1, 1984), null);
Person copy = new Person(original);
copy.setBirthYear(1800);
System.out.println(original);
```

The output would be

```
Natalie Dressed, April 1, 1800-
```

When we changed the birth year in the object `copy`, we also changed the birth year in the object `original`. This is because we are using our unsafe version of the copy constructor. Both `original.born` and `copy.born` contain the same reference to the same `Date` object.

This all happens because we used the unsafe version of the copy constructor. Fortunately, here we use a safer version of the copy constructor that sets the born instance variables as follows:

```
born = new Date(original.born);
```

which is equivalent to

```
this.born = new Date(original.born);
```

This version, which we did use, makes the instance variable `this.born` an independent `Date` object that represents the same date as `original.born`. So if you change a date in the `Person` object created by the copy constructor, you will not change that date in the `original Person` object.

Note that if a class, such as `Person`, has instance variables of a class type, such as the instance variables `born` and `died`, then to define a correct copy constructor for the class `Person`, you must already have copy constructors for the class `Date` of the instance variables. The easiest way to ensure this for all your classes is to always include a copy constructor in every class you define.

Copy Constructor

A **copy constructor** is a constructor with one parameter of the same type as the class. A copy constructor should be designed so the object it creates is intuitively an exact copy of its parameter, but a completely independent copy. See Displays 5.19 and 5.20 for examples of copy constructors.

clone

The Java documentation says to use a method named `clone` instead of a copy constructor, and, as you will see later in this book, there are situations where the copy constructor will not work as desired and you need the `clone` method. However, we do not yet have enough background to delve into this method.(It is discussed later in this book in Chapters 8 and 13.) Despite the Java documentation, many excellent programmers prefer to sometimes use copy constructors. In this book, we will use both copy constructors and the clone method.

PITFALL: Privacy Leaks

leaking accessor methods

Consider the accessor method `getBirthDate` for the class `Person` (Display 5.19), which we reproduce in what follows:

```
public Date getBirthDate()
{
    return new Date(born);
}
```

Do not make the mistake of defining the accessor method as follows:

```
public Date getBirthDate() //Unsafe
{
    return born; //Not good
}
```

(continued)

PITFALL: (continued)

Assume we had used the unsafe version of `getBirthDate` instead of the one in Display 5.19. It would then be possible for a program that uses the class `Person` to change the private instance variable `born` to any date whatsoever and bypass the checks in constructor and mutator methods of the class `Person`. For example, consider the following code, which might appear in some program that uses the class `Person`:

```
Person citizen = new Person(
"Joe Citizen", new Date("January", 1, 1900), new Date("January", 1,
     1990));
Date dateName = citizen.getBirthDate();
dateName.setDate("April", 1, 3000);
```

privacy leak

This code changes the date of birth so it is after the date of death (an impossibility in the universe as we know it). This citizen was not born until after he or she died! This sort of situation is known as a **privacy leak**, because it allows a programmer to circumvent the `private` modifier before an instance variable such as `born`, and to change the private instance variable to anything whatsoever.

The following code would be illegal in our program:

```
citizen.born.setDate("April", 1, 3000); //Illegal
```

This is illegal because `born` is a private instance variable. However, with the unsafe version of `getBirthDate` (and we are now assuming that we did use the unsafe version), the variable `dateName` contains the same reference as `citizen.born` and so the following is legal and equivalent to the illegal statement:

```
dateName.setDate("April", 1, 3000); //Legal and equivalent to
   //illegal statement.
```

It is as if you have a friend named Robert who is also known as Bob. Some bully wants to beat up Robert, so you say "You cannot beat up Robert." The bully says "OK, I will not beat up Robert, but I will beat up Bob." Bob and Robert are two names for the same person. So, if you protect Robert but do not protect Bob, you have really accomplished nothing.

This is all if we used the unsafe version of `getBirthDate`, which simply returns the reference in the private instance variable born. Fortunately, here we use a safer version of `getBirthDate`, which has the following `return` statement:

```
return new Date(born);
```

PITFALL: (continued)

This `return` statement does not return the reference in the private instance variable `born`. Instead, it uses the copy constructor to return a reference to a new object that is an exact copy of the object named by `born`. If the copy is changed, that has no effect on the date whose reference is in the instance variable `born`. Thus, a privacy leak is avoided.

leaking mutator methods

Note that returning a reference is not the only possible source of privacy leaks. A privacy leak can also arise from an incorrectly defined constructor or mutator method. Notice the definition for the method `setBirthDate` in Display 5.19 and reproduced as follows:

```
public void setBirthDate(Date newDate)
{
    if (consistent(newDate, died))
        born = new Date(newDate);
    else
    {
        System.out.println("Inconsistent dates. Aborting.");
        System.exit(0);
    }
}
```

Note that the instance variable `born` is set to a copy of the parameter `newDate`. Suppose that instead of

```
born = new Date(newDate);
```

we simply use

```
born = newDate;
```

And suppose we use the following code in some program:

```
Person personObject = new Person(
  "Josephine", new Date("January", 1, 2000), null);
Date dateName = new Date("February", 2, 2002);
personObject.setBirthDate(dateName);
```

where `personObject` names an object of the class `Person`. The following will change the year part of the `Date` object named by the `born` instance variable of the object `personObject` and will do so without going through the checks in the mutator methods for `Person`:

```
dateName.setYear(1000);
```

(continued)

PITFALL: (continued)

Because `dateName` contains the same reference as the private instance variable born of the object `personObject`, changing the year part of `dateName` changes the year part of the private instance variable born of `personObject`. Not only does this bypass the consistency checks in the mutator method `setBirthDate`, but it also is a likely source of an inadvertent change to the `born` instance variable.

If we define `setBirthDate` as we did in Display 5.19 and as shown in the following, this problem does not happen. (If you do not see this, go through the code step by step and trace what happens.)

```
public void setBirthDate(Date newDate)
{
    if (consistent(newDate, died))
        born = new Date(newDate);
    . . .
```

clone

One final word of warning: Using copy constructors in this manner is not the officially sanctioned way to make copies of an object in Java. The authorized way to is to define a method named `clone`. We will discuss `clone` methods in Chapters 8 and 13. In Chapter 8, we show you that, in some situations, there are advantages to using a `clone` method instead of a copy constructor. In Chapter 13, we describe the official way to define the `clone` method. For what we will be doing until then, a copy constructor will be a very adequate way of creating copies of an object.

MyProgrammingLab

Self Test Exercises

38. What is a copy constructor?

39. What output is produced by the following code?

```
Date date1 = new Date("January", 1, 2006);
Date date2;
date2 = date1;
date2.setDate("July", 4, 1776);
System.out.println(date1);
```

What output is produced by the following code? Only the third line is different from the previous case.

```
Date date1 = new Date("January", 1, 2006);
Date date2;
date2 = new Date(date1);
date2.setDate("July", 4, 1776);
System.out.println(date1);
```

MyProgrammingLab

Self Test Exercises

40. What output is produced by the following code?

```
Person original =
             new Person("Natalie Dressed",
                             new Date("April", 1, 1984), null);
Person copy = new Person(original);
copy.setBirthDate(new Date("April", 1, 1800));
System.out.println(original)
```

Mutable and Immutable Classes

Contrast the accessor methods `getName` and `getBirthDate` of the class `Person` (Display 5.19). We reproduce the two methods in what follows:

```
public String getName()
{
    return name;
}


public Date getBirthDate()
{
    return new Date(born);
}
```

Notice that the method `getBirthDate` does not simply return the reference in the instance variable `born`, but instead uses the copy constructor to return a reference to a copy of the birth date object. We already explained why we do this. If we return the reference in the instance variable `born`, then we can place this reference in a variable of type `Date`, and that variable could serve as another name for the private instance variable `born`, which would allow us to violate the privacy of the instance variable `born` by changing it using a mutator method of the class `Date`. This is exactly what we discussed in the previous subsection. So why not do something similar in the method `getName`?

The method `getName` simply returns the reference in the private instance variable `name`. So, if we do the following in a program, then the variable `nameAlias` will be another name for the `String` object of the private instance variable `name`:

```
Person citizen = new Person(
"Joe Citizen", new Date("January", 1, 1900), new Date("January",
1, 1990));
String nameAlias = citizen.getName();
```

It looks as though we could use a mutator method from the class `String` to change the name referenced by `nameAlias` and so violate the privacy of the instance variable `name`. Is something wrong? Do we have to rewrite the method `getName` to use the copy constructor for the class `String`? No, everything is fine. We cannot use a mutator method with `nameAlias` because the class `String` has no mutator methods! The class `String` contains no methods that change any of the data in a `String` object.

At first, it may seem as though you can change the data in an object of the class `String`. What about the string processing we have seen, such as the following?

```
String greeting = "Hello";
greeting = greeting + "friend.";
```

Have we not changed the data in the `String` object from `"Hello"` to `"Hello friend."`? No, we have not. The expression `greeting + "friend."` does not change the object `"Hello"`; it creates a new object, so the assignment statement

```
greeting = greeting + "friend.";
```

replaces the reference to `"Hello"` with a reference to the different `String` object `"Hello friend."` The object `"Hello"` is unchanged. To see that this is true, consider the following code:

```
String greeting = "Hello";
String helloVariable = greeting;
greeting = greeting + "friend.";
System.out.println(helloVariable);
```

This produces the output `"Hello"`. If the object `"Hello"` had been changed, the output would have been `"Hello friend."`

immutable

A class that contains no methods (other than constructors) that change any of the data in an object of the class is called an **immutable class**, and objects of the class are called **immutable objects**. The class `String` is an immutable class. It is perfectly safe to return a reference to an immutable object, because the object cannot be changed in any undesirable way; in fact, it cannot be changed in any way whatsoever.

mutable

A class that contains public mutator methods or other public methods, such as input methods, that can change the data in an object of the class is called a **mutable class**, and objects of the class are called **mutable objects**. The class `Date` is an example of a mutable class; many, perhaps most, of the classes you define will be mutable classes. As we noted in the Pitfall entitled "Privacy Leaks" (but using other words): You should never write a method that returns a mutable object, but should instead use a copy constructor (or other means) to return a reference to a completely independent copy of the mutable object.

TIP: Deep Copy versus Shallow Copy

In the previous two subsections, we contrasted the following two ways of defining the method `getBirthDate` (Display 5.19):

```
public Date getBirthDate()
{
    return new Date(born);
}
public Date getBirthDate() //Unsafe
{
    return born; //Not good
}
```

VideoNote Deep Copy vs. Shallow Copy Example

As we noted, the first definition is the better one (and the one used in Display 5.19). The first definition returns what is known as a *deep copy* of the object born. The second definition returns what is known as a *shallow copy* of the object `born`.

deep copy

A **deep copy** of an object is a copy that, with one exception, has no references in common with the original. The one exception is that references to immutable objects are allowed to be shared (because immutable objects cannot change in any way and so cannot be changed in any undesirable way). For example, the first definition of `getBirthDate` returns a deep copy of the date stored by the instance variable `born`. So, if you change the object returned by `getBirthDate`, this does not change the `Date` object named by the instance variable `born`. The reason is that we defined the copy constructor for the class `Date` to create a deep copy (Display 5.20). Normally, copy constructors and accessor methods should return a deep copy.

shallow copy

Any copy that is not a deep copy is called a **shallow copy**. For example, the second definition of `getBirthDate` returns a shallow copy of the date stored by the instance variable `born`.

We will have more to say about deep and shallow copies in later chapters. ■

Never Return a Reference to a Mutable Private Object

A class that contains mutator methods or other methods, such as input methods, that can change the data in an object of the class is called a **mutable class**, and objects of the class are called **mutable objects**. When defining accessor methods (or almost any methods), your method should not return a reference to a mutable object. Instead, use a copy constructor (or other means) to return a reference to a completely independent copy of the mutable object.

TIP: Assume Your Coworkers Are Malicious

Our discussion of privacy leaks in the previous subsections was concerned with the effect of somebody trying to defeat the privacy of an instance variable. You might object that your coworkers are nice people and would not knowingly sabotage your software. That is probably true, and we do not mean to accuse your coworkers of malicious intent. However, the same action that can be performed intentionally by a malicious enemy can also be performed inadvertently by your friends or even by you yourself. The best way to guard against such honest mistakes is to pretend that you are defending against a malicious enemy. ■

MyProgrammingLab

Self-Test Exercises

41. Complete the definition of the method `set` for the class `Person` (Display 5.19).

42. Classify each of the following classes as either mutable or immutable: `Date` (Display 4.11), `Person` (Display 5.19), and `String`.

43. Normally, it is dangerous to return a reference to a private instance variable of class type, but it is OK if the class type is `String`. What is special about the class `String` that makes this true?

5.4 Packages and `javadoc`

... he furnished me,
From mine own library with volumes that
I prize above my dukedom.

WILLIAM SHAKESPEARE, *The Tempest*

In this section, we cover packages, which are Java libraries, and then cover the `javadoc` program, which automatically extracts documentation from packages and classes. Although these are important topics, they are not used in the rest of this book. You can study this section at any time you wish; you need not cover this section before studying any other topic in this book.

This section does not use any of the material in Section 5.3, and so can be covered before Section 5.3.

This section assumes that you know about directories (which are called folders in some operating systems), that you know about path names for directories (folders), and that you know about PATH (environment) variables. These are not Java topics. They are part of your operating system, and the details depend on your particular

operating system. If you can find out how to set the PATH variable on your operating system, you will know enough about these topics to understand this section.

Packages and `import` Statements

package

A **package** is Java's way of forming a library of classes. You can make a package from a group of classes and then use the package of classes in any other class or program you write without the need to move the classes to the directory (folder) in which you are working. All you need to do is include an **`import` statement** that names the package. We have already used `import` statements with some predefined standard Java packages. For example, the following, which we have used before, makes available the class `Scanner` of the package `java.util`:

`import` statement

```
import java.util.Scanner;
```

You can make all the classes in the package available by using the following instead:

```
Import java.util.*;
```

There is no overhead cost for importing the entire package as opposed to just a few classes.

The `import` statements should be at the beginning of the file. Only blank lines, comments, and `package` statements may precede the list of `import` statements. We discuss `package` statements next.

`import` Statement

You can use a class from a package in any program or class definition by placing an **`import` statement** that names the package and the class from the package at the start of the file containing the program (or class definition). The program (or class definition) need not be in the same directory as the classes in the package.

SYNTAX

```
import Package_Name.Class;
```

EXAMPLE

```
import java.util.Scanner;
```

You can import all the classes in a package by using an asterisk in place of the class's name.

SYNTAX

```
import Package_Name.*;
```

EXAMPLE

```
import java.util.*;
```