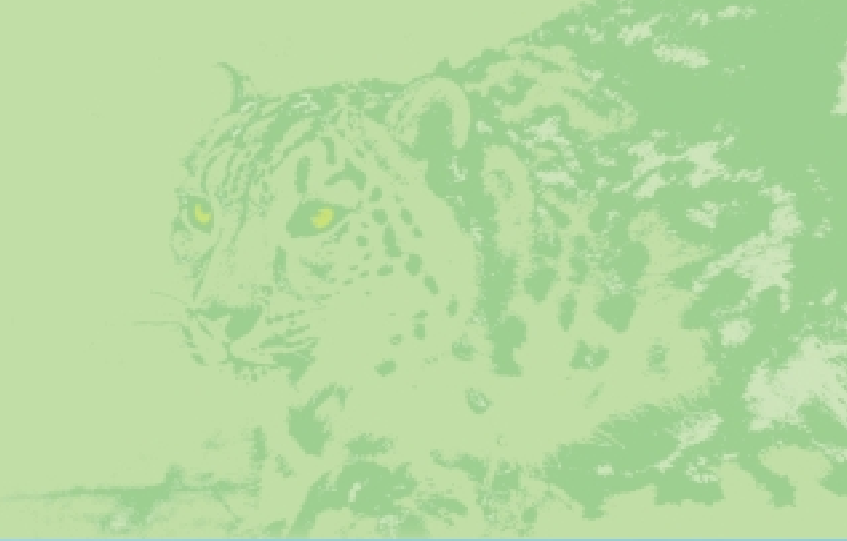
CHAPTER 7

Inheritance



7 Inheritance

Like mother, like daughter.

Common saying

Introduction

Object-oriented programming (OOP) is a popular and powerful programming philosophy. One of the main techniques of OOP is known as *inheritance*. **Inheritance** means that a very general form of a class can be defined and compiled. Later, more specialized versions of that class may be defined by starting with the already defined class and adding more specialized instance variables and methods. The specialized classes are said to *inherit* the methods and instance variables of the previously defined general class. In this chapter we cover inheritance in general and more specifically how it is realized in Java.

Prerequisites

This chapter does not use any material on arrays from Chapter 6. It does require Chapters 1 through 5 with the exception that most of the chapter does not require Section 5.4 on packages and `javadoc`. The subsection "Protected and Package Access" is the only part of this chapter that requires anything from Section 5.4 and it requires only the material on packages and not any material on `javadoc`. The subsection "Protected and Package Access" can be omitted without any loss of continuity in reading this chapter.

7.1 Inheritance Basics

If there is anything that we wish to change in the child, we should first examine it and see whether it is not something that could better be changed in ourselves.

Carl Gustav Jung, *The Integration of the Personality*

Inheritance is the process by which a new class—known as a *derived class*—is created from another class, called the *base class*. A derived class automatically has all the instance variables and all the methods that the base class has, and can have additional methods and/or additional instance variables.

DERIVED CLASSES

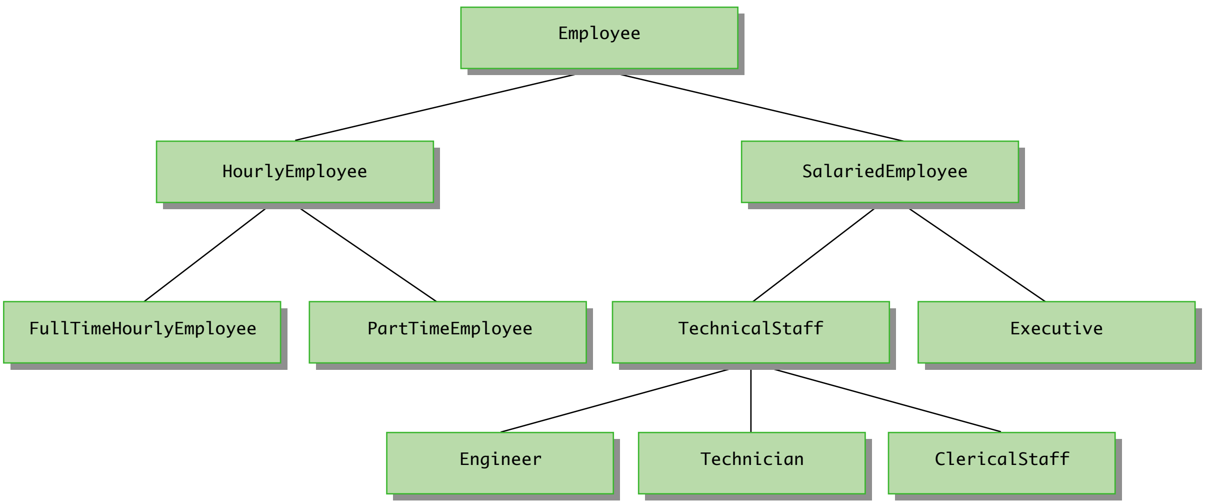
Suppose we are designing a record-keeping program that has records for salaried employees and hourly employees. There is a natural hierarchy for grouping these classes. These are all classes of people who share the property of being employees.

Employees who are paid an hourly wage are one subset of employees. Another subset consists of salaried employees who are paid a fixed wage each month. Although the program may not need any type corresponding to the set of all employees, thinking in terms of the more general concept of employees can be useful. For example, all employees have a name and a hire date (when they started working for the company), and the methods for setting and changing names and hire dates will be the same for salaried and hourly employees. The classes for hourly employees and salaried employees may be further subdivided as diagrammed in Display 7.1.

Within Java you can define a class called `Employee` that includes all employees, whether salaried or hourly, and then use this class to define classes for hourly employees and salaried employees. You can then, in turn, use classes like `HourlyEmployee` to define classes like `PartTimeHourlyEmployee`, and so forth.

Display 7.2 shows our definition for the class `Employee`. The class `Employee` is a pretty ordinary class much like earlier classes we have seen. What will be interesting about the class `Employee` is how we use it to create a class for hourly employees and a

Display 7.1 A Class Hierarchy



Display 7.2 The Base Class Employee *(Part 1 of 2)*

```
/**
 Class Invariant: All objects have a name string and hire date.
 A name string of "No name" indicates no real name specified yet.
 A hire date of Jan 1, 1000 indicates no real hire date specified yet.
*/
public class Employee
{
    private String name;
    private Date hireDate;

    public Employee()
    {
         name = "No name";
         hireDate = new Date("Jan", 1, 1000); //Just a placeholder.
    }

    /**
     Precondition: Neither theName nor theDate is null.
    */
    public Employee(String theName, Date theDate)
    {
        if (theName == null || theDate == null)
        {
             System.out.println("Fatal Error creating employee.");
             System.exit(0);
        }
        name = theName;
        hireDate = new Date(theDate);
    }

    public Employee(Employee originalObject)
    {
         name = originalObject.name;
         hireDate = new Date(originalObject.hireDate);
    }

    public String getName()
    {
        return name;
    }

    public Date getHireDate()
    {
        return new Date(hireDate);
    }
```

The class Date is defined in Display 4.11.

Display 7.2 The Base Class Employee *(Part 2 of 2)*

```
    /**
     Precondition newName is not null.
    */
    public void setName(String newName)
    {
        if (newName == null)
        {
             System.out.println("Fatal Error setting employee name.");
             System.exit(0);
        }
       else
            name = newName;
    }

    /**
     Precondition newDate is not null.
    */
    public void setHireDate(Date newDate)
    {
        if (newDate == null)
        {
             System.out.println("Fatal Error setting employee hire date.");
             System.exit(0);
        }
        else
            hireDate = new Date(newDate);
    }


    public String toString()
    {
        return (name + " " + hireDate.toString());
    }

    public boolean equals(Employee otherEmployee)
    {
        return (name.equals(otherEmployee.name)
                          && hireDate.equals(otherEmployee.hireDate));
    }
}
```

class for salaried employees. It is legal to create an object of the class `Employee`, but our reason for defining the class `Employee` is so that we can define derived classes for different kinds of employees.

Display 7.3 contains the definition of a class for hourly employees. An hourly employee is an employee, so we define the class `HourlyEmployee` to be a *derived* class of the class `Employee`. A **derived class** is a class defined by adding instance variables and methods to an existing class. The existing class that the derived class is built upon is called the **base class**. In our example, `Employee` is the base class and `HourlyEmployee` is the derived class. As you can see in Display 7.3, the way we indicate that `HourlyEmployee` is a derived class of `Employee` is by including the phrase `extends Employee` on the first line of the class definition, like so:

derived class

base class

extends

```
public class HourlyEmployee extends Employee
```

subclass and superclass

A derived class is also called a **subclass**, in which case the base class is usually called a **superclass**. However, we prefer to use the terms *derived class* and *base class.*

When you define a derived class, you give only the added instance variables and the added methods. For example, the class `HourlyEmployee` has all the instance variables and all the methods of the class `Employee`, but you do not mention them in the definition of `HourlyEmployee`. Every object of the class `HourlyEmployee` has instance variables called `name` and `hireDate`, but you do not specify the instance variable `name` or the instance variable `hireDate` in the definition of the class `HourlyEmployee`. The class `HourlyEmployee` (or any other derived class) is said to **inherit** the instance variables and methods of the base class that it extends. For this reason, the topic of derived classes is called **inheritance**.

inheritance

Just as it inherits the instance variables of the class `Employee`, the class `HourlyEmployee` inherits all the methods from the class `Employee`. So, the class `HourlyEmployee` inherits the methods `getName`, `getHireDate`, `setName`, and `setHireDate`, from the class `Employee`.

For example, suppose you create a new object of the class `HourlyEmployee` as follows:

```
HourlyEmployee joe = new HourlyEmployee();
```

Then, the name of the object `joe` can be changed using the method `setName`, which the class `HourlyEmployee` inherited from the class `Employee`. The inherited method `setName` is used just like any other method; for example:

```
joe.setName("Josephine");
```

A small demonstration of this is given in Display 7.4.

Display 7.5 contains the definition of the class `SalariedEmployee`, which is also derived from the class `Employee`. The class `SalariedEmployee` inherits all the instance variables and methods of the class `Employee`. Inheritance allows you to reuse code, such as the code in the class `Employee`, without needing to literally copy the code into the definitions of the derived classes, such as `HourlyEmployee` and `SalariedEmployee`.

Display 7.3 The Derived Class HourlyEmployee *(Part 1 of 3)*

```
/**
 Class Invariant: All objects have a name string, hire date, nonnegative
 wage rate, and nonnegative number of hours worked. A name string of
 "No name" indicates no real name specified yet. A hire date of Jan 1, 1000
 indicates no real hire date specified yet.
*/
public class HourlyEmployee extends Employee
{
    private double wageRate;
    private double hours; //for the month

    public HourlyEmployee()
    {
        super();
        wageRate = 0;
        hours = 0;
    }

    /**
     Precondition: Neither theName nor theDate is null;
     theWageRate and theHours are nonnegative.
    */
    public HourlyEmployee(String theName, Date theDate,
                          double theWageRate, double theHours)
    {
        super(theName, theDate);
        if ((theWageRate >= 0) && (theHours >= 0))
        {
            wageRate = theWageRate;
            hours = theHours;
        }
        else
        {
            System.out.println(
                      "Fatal Error: creating an illegal hourly employee.");
            System.exit(0);
        }
    }

    public HourlyEmployee(HourlyEmployee originalObject)
    {
        super(originalObject);
        wageRate = originalObject.wageRate;
        hours = originalObject.hours;
    }
```

It will take the rest of Section 7.1 to explain this class definition.

If this line is omitted, Java will still invoke the no-argument constructor for the base class.

An object of the class `HourlyEmployee` is also an instance of the class `Employee`.

Display 7.3 The Derived Class HourlyEmployee *(Part 2 of 3)*

```
    public double getRate()
    {
        return wageRate;
    }

    public double getHours()
    {
        return hours;
    }

    /**
     Returns the pay for the month.
    */
    public double getPay()
    {
        return wageRate*hours;
    }

    /**
     Precondition: hoursWorked is nonnegative.
    */
    public void setHours(double hoursWorked)
    {
          if (hoursWorked >= 0)
              hours = hoursWorked;
          else
          {
              System.out.println("Fatal Error: Negative hours worked.");
              System.exit(0);
          }
     }

    /**
     Precondition: newWageRate is nonnegative.
    */
    public void setRate(double newWageRate)
    {
          if (newWageRate >= 0)
              wageRate = newWageRate;
          else
          {
              System.out.println("Fatal Error: Negative wage rate.");
              System.exit(0);
          }
    }
```

Display 7.3 The Derived Class HourlyEmployee *(Part 3 of 3)*

The method toString is overridden so it is different in the derived class HourlyEmployee than it is in the base class Employee.

```
    public String toString()
    {
        return (getName() + " " + getHireDate().toString()
                + "\n$" + wageRate + " per hour for " + hours + " hours");
    }

    public boolean equals(HourlyEmployee other)
    {
       return (getName().equals(other.getName())
                && getHireDate().equals(other.getHireDate())
                && wageRate == other.wageRate
                && hours == other.hours);
    }
}
```

We will show you a better way to define equals later in this chapter.

Derived Class (Subclass)

You define a **derived class** by starting with another already defined class and adding (and/or changing) methods, instance variables, and static variables. The class you start with is called the **base class**. The derived class inherits all the public methods, all the public and private instance variables, and all the public and private static variables from the base class and can add more instance variables, more static variables, and/or more methods. So, of the things we have seen thus far, the only members not inherited are private methods. (As discussed a little later in this chapter, the derived class definition can also change the definition of an inherited method.)

A derived class is also called a **subclass**, in which case the base class is usually called a **superclass**.

Syntax:

```
public class Derived_Class_Name extends Base_Class_Name
{
        Declarations_of_Added_Static_Variables
        Declarations_of_Added_Instance_Variables
        Definitions_of_Added__And_Overridden_Methods
}
```

Example:

See Displays 7.3 and 7.5.

Display 7.4 Inheritance Demonstration

The methods getName and setName are inherited from the base class Employee.

```
public class InheritanceDemo
{
    public static void main(String[] args)
    {
        HourlyEmployee joe = new HourlyEmployee("Joe Worker",
                                    new Date("Jan", 1, 2004), 50.50, 160);

        System.out.println("joe's longer name is " + joe.getName());

        System.out.println("Changing joe's name to Josephine.");
        joe.setName("Josephine");

        System.out.println("joe's record is as follows:");
        System.out.println(joe);
      }
}
```

Sample Dialogue

```
joe's longer name is Joe Worker
Changing joe's name to Josephine.
joe's record is as follows:
Josephine Jan 1, 2004
$50.5 per hour for 160 hours
```

Inherited Members

A derived class automatically has all the instance variables, all the static variables, and all the public methods of the base class. These members from the base class are said to be **inherited**. These inherited methods and inherited instance and static variables are, with one exception, not mentioned in the definition of the derived class, but they are automatically members of the derived class. As explained later in this chapter, you can give a definition for an inherited method in the definition of the derived class; this will redefine the meaning of the method for the derived class.

Display 7.5 The Derived Class SalariedEmployee *(Part 1 of 2)*

```
/**
 Class Invariant: All objects have a name string, hire date,
 and nonnegative salary. A name string of "No name" indicates
 no real name specified yet. A hire date of Jan 1, 1000 indicates
 no real hire date specified yet.
*/
public class SalariedEmployee extends Employee
{
    private double salary; //annual

    public SalariedEmployee()
    {
        super();
        salary = 0;
    }

    /**
     Precondition: Neither theName nor theDate are null;
     theSalary is nonnegative.
    */
    public SalariedEmployee(String theName, Date theDate, double theSalary)
    {
         super(theName, theDate);
         if (theSalary >= 0)
             salary = theSalary;
         else
         {
             System.out.println("Fatal Error: Negative salary.");
             System.exit(0);
         }
    }

    public SalariedEmployee(SalariedEmployee originalObject )
    {
         super(originalObject);
         salary = originalObject.salary;
    }

    public double getSalary()
    {
        return salary;
    }
```

It will take the rest of Section 7.1 to fully explain this class definition.

If this line is omitted, Java will still invoke the no-argument constructor for the base class.

An object of the class `SalariedEmployee` is also an object of the class `Employee`.

Display 7.5 **The Derived Class SalariedEmployee *(Part 2 of 2)***

```
    /**
     Returns the pay for the month.
    */
    public double getPay()
    {
        return salary/12;
    }

    /**
     Precondition: newSalary is nonnegative.
    */
    public void setSalary(double newSalary)
    {
         if (newSalary >= 0)
             salary = newSalary;
         else
         {
             System.out.println("Fatal Error: Negative salary.");
             System.exit(0);
         }
    }

    public String toString()
    {
        return (getName() + " " + getHireDate().toString()
                                  + "\n$" + salary + " per year");
    }

    public boolean equals(SalariedEmployee other)
    {
        return (getName().equals(other.getName())
                && getHireDate().equals(other.getHireDate())
                && salary == other.salary);
    }
}
```

We will show you a better way to define `equals` *later in this chapter.*

Parent and Child Classes

parent class
child class

ancestor class
descendent class

A base class is often called the **parent class**. A derived class is then called a **child class**. This analogy is often carried one step further. A class that is a parent of a parent of a parent of another class (or some other number of "parent of" iterations) is often called an **ancestor class**. If class A is an ancestor of class B, then class B is often called a **descendent** of class A.

OVERRIDING A METHOD DEFINITION

The definition of an inherited method can be changed in the definition of a derived class so that it has a meaning in the derived class that is different from what it is in the base class. This is called **overriding** the definition of the inherited method. For example, the methods `toString` and `equals` are overridden (redefined) in the definition of the derived class `HourlyEmployee`. They are also overridden in the class `SalariedEmployee`. To override a method definition, simply give the new definition of the method in the class definition, just as you would do with a method that is added in the derived class.

overriding

OVERRIDING A METHOD DEFINITION

A derived class inherits methods that belong to the base class. However, if a derived class requires a different definition for an inherited method, the method may be redefined in the derived class. This is called **overriding** the method definition.

THE `final` MODIFIER

If you add the modifier `final` to the definition of a method, that indicates that the method may not be redefined in a derived class. If you add the modifier `final` to the definition of a class, that indicates that the class may not be used as a base class to derive other classes. We will say more about the `final` modifier in Chapter 8.

Pitfall

OVERRIDING VERSUS OVERLOADING

Do not confuse *overriding* (that is, redefining) a method definition in a derived class with *overloading* a method name. When you override a method definition, the new method definition given in the derived class has the exact same number and types of parameters. On the other hand, if the method in the derived class were to have a different number of parameters or a parameter of a different type from the method in the base class, then the derived class would have both methods. That would be overloading. For example, suppose we added the following method to the definition of the class `HourlyEmployee` (Display 7.3):

```
public void setName(String firstName, String lastName)
{
    name = firstName + " " + lastName;
}
```

The class `HourlyEmployee` would then have this two-argument method `setName` and it would also inherit the following one-argument method `setName` from the base class `Employee`:

```
public void setName(String newName)
{
    if (newName == null)
    {
        System.out.println("Fatal Error setting employee name.");
        System.exit(0);
    }
    else
        name = newName;
}
```

The class `HourlyEmployee` would have two methods named `setName`. This would be *overloading* the method name `setName`.

On the other hand, both the class `Employee` and the class `HourlyEmployee` define a method with the following method heading:

```
public String toString()
```

In this case, the class `HourlyEmployee` has only one method named `toString()`, but the definition of the method `toString()` in the class `HourlyEmployee` is different from the definition of `toString()` in the class `Employee`; the method `toString()` has been *overridden* (that is, redefined).

If you get overriding and overloading confused, you do have one consolation. They are both legal.

Self-Test Exercises

1. Suppose the class named `DiscountSale` is a derived class of a class called `Sale`. Suppose the class `Sale` has instance variables named `price` and `numberOfItems`. Will an object of the class `DiscountSale` also have instance variables named `price` and `numberOfItems`?

2. Suppose the class named `DiscountSale` is a derived class of a class called `Sale`, and suppose the class `Sale` has public methods named `getTotal` and `getTax`. Will an object of the class `DiscountSale` have methods named `getTotal` and `getTax`? If so, do these methods have to perform the exact same actions in the class `DiscountSale` as in the class `Sale`?

3. Suppose the class named `DiscountSale` is a derived class of a class called `Sale`, and suppose the class `Sale` has a method with the following heading and no other methods named `getTax`:

   ```
   public double getTax()
   ```

And suppose the definition of the class `DiscountSale` has a method definition with the following heading and no other method definitions for methods named `getTax`:

```
public double getTax(double rate)
```

How many methods named `getTax` will the class `DiscountSale` have and what are their headings?

4. The class `HourlyEmployee` (Display 7.3) has methods named `getName` and `getRate` (among others). Why does the definition of the class `HourlyEmployee` contain a definition of the method `getRate` but no definition of the methods `getName`?

THE super CONSTRUCTOR

You can invoke a constructor of the base class within the definition of a derived class constructor. A constructor for a derived class uses a constructor from the base class in a special way. A constructor for the base class normally initializes all the data inherited from the base class. Thus, a constructor for a derived class begins with an invocation of a constructor for the base class.

There is a special syntax for invoking the base class constructor that is illustrated by the constructor definitions for the class `HourlyEmployee` given in Display 7.3. In what follows we have reproduced the beginning of one of the constructor definitions for the class `HourlyEmployee` taken from that display:

```
public HourlyEmployee(String theName, Date theDate,
                      double theWageRate, double theHours)
{
    super(theName, theDate);
    if ((theWageRate >= 0) && (theHours >= 0))
    {
        wageRate = theWageRate;
        hours = theHours;
    }
    else
   ...
```

The line

super

```
super(theName, theDate);
```

is a call to a constructor for the base class, which in this case is a call to a constructor for the class `Employee`.

There are some restrictions on how you can use the base class constructor call `super`. You cannot use an instance variable as an argument to `super`. Also, the call to the base class constructor (`super`) must always be the first action taken in a constructor definition. You cannot use it later in the definition of a constructor.

Notice that you use the keyword `super` to call the constructor of the base class. You do not use the name of the constructor; you do *not* use

```
Employee(theName, theDate); //ILLEGAL
```

If a constructor definition for a derived class does not include an invocation of a constructor for the base class, then the no-argument constructor of the base class constructor will be invoked automatically as the first action of the derived class constructor. So, the following definition of the no-argument constructor for the class `HourlyEmployee` (with `super` omitted) is equivalent to the version we gave in Display 7.3:

```
public HourlyEmployee()
{
    wageRate = 0;
    hours = 0;
}
```

A derived class object has all the instance variables of the base class. These inherited instance variables should be initialized, and the base class constructor is the most convenient place to initialize these inherited instance variables. That is why you should always include a call to one of the base class constructors when you define a constructor for a derived class. As already noted, if you do not include a call to a base class constructor (using `super`), then the no-argument constructor of the base class is called automatically. (If there is no no-argument constructor for the base class, that is an error condition.)

CALL TO A BASE CLASS CONSTRUCTOR

Within the definition of a constructor for a class, you can use `super` as a name for a constructor of the base class. Any invocation of `super` must be the first action taken by the constructor.

EXAMPLE:

```
public SalariedEmployee(SalariedEmployee originalObject)
{
    super(originalObject); //Invocation of base class constructor.
    salary = originalObject.salary;
}
```

THE this CONSTRUCTOR

`this`

When defining a constructor, it is sometimes convenient to be able to call one of the other constructors in the same class. You can use the keyword `this` as a method name to invoke a constructor in the same class. This use of `this` is similar to the use of `super`, but with `this`, the call is to a constructor of the same class, not to a constructor for the base class. For example, consider the following alternate, and equivalent, definition of the no-argument constructor for the class `HourlyEmployee` (from Display 7.3):

```
public HourlyEmployee()
{
    this("No name", new Date("Jan", 1, 1000), 0, 0);
}
```

The line with `this` is an invocation of the constructor with the following heading:

```
public HourlyEmployee(String theName, Date theDate,
                      double theWageRate, double theHours)
```

The restrictions on how you can use the base class constructor call `super` also apply to the `this` constructor. You cannot use an instance variable as an argument to `this`. Also, any call to the constructor `this` must always be the first action taken in a constructor definition. Thus, a constructor definition cannot contain both an invocation of `super` and an invocation of `this`. If you want to include both a call to `super` and a call to `this`, use a call with `this`, and have the constructor that is called with `this` have `super` as its first action.

Call to Another Constructor in the Same Class

Within the definition of a constructor for a class, you can use `this` as a name for another constructor in the same class. Any invocation of `this` must be the first action taken by the constructor.

Example:

```
public HourlyEmployee()
{
    this("No name", new Date("Jan", 1, 1000), 0, 0);
}
```

Tip

An Object of a Derived Class Has More than One Type

An object of a derived class has the type of the derived class, and it also has the type of the base class, and more generally, has the type of every one of its ancestor classes. For example, consider the following copy constructor definition from the class `HourlyEmployee` (Display 7.3):

```
public HourlyEmployee(HourlyEmployee originalObject)
{
    super(originalObject);
```

```
    wageRate = originalObject.wageRate;
    hours = originalObject.hours;
}
```

The line

```
super(originalObject);
```

is an invocation of a constructor for the base class `Employee`. The class `Employee` has no constructor with a parameter of type `HourlyEmployee`, but `originalObject` is of type `HourlyEmployee`. Fortunately, every object of type `HourlyEmployee` is also of type `Employee`. So, this invocation of `super` is an invocation of the copy constructor for the class `Employee`.

The fact that every object not only is of its own type but is also of the type of its ancestor classes simply reflects what happens in the everyday world. An hourly employee is an employee as well as an hourly employee. This sometimes is referred to as the **"is a" relationship**: For example, an `HourlyEmployee` is an `Employee`.

is a

Display 7.6 contains a program demonstrating that an `HourlyEmployee` and a `SalariedEmployee` are also `Employee` objects. The method `showEmployee` requires an argument of type `Employee`. The objects `joe` and `sam` are of type `Employee` because they are instances of classes derived from the class `Employee` and so they are suitable arguments for `showEmployee`.

An Object of a Derived Class Has More than One Type

An object of a derived class has the type of the derived class, and it also has the type of the base class. More generally, a derived class has the type of every one of its ancestor classes. So, you can assign an object of a derived class to a variable of any ancestor type (but not the other way around). You can plug in a derived class object for a parameter of any of its ancestor types. More generally, you can use a derived class object anyplace you can use an object of any of its ancestor types.

Pitfall

The Terms "Subclass" and "Superclass"

subclass and superclass

Many programmers and authors use the term *subclass* for a derived class and *superclass* for its base class (or any of its ancestor classes). This is logical. For example, the collection of all hourly employees in the world is a subclass of all employees. Similarly, the collection of all objects of type `HourlyEmployee` is a subcollection of the collection of all objects of the class `Employee`. As you add more instance variables and methods, you restrict the number of objects that can satisfy the class definition. Despite this logic, people often reverse the terms *subclass* and *superclass.*

Display 7.6 An Object Belongs to Multiple Classes

```
public class IsADemo
{
    public static void main(String[] args)
    {
        SalariedEmployee joe = new SalariedEmployee("Josephine",
                                   new Date("Jan", 1, 2004), 100000);
        HourlyEmployee sam = new HourlyEmployee("Sam",
                                   new Date("Feb", 1, 2003), 50.50, 40);

        System.out.println("joe's longer name is " + joe.getName());

        System.out.println("showEmployee(joe) invoked:");
        showEmployee(joe);

        System.out.println("showEmployee(sam) invoked:");
        showEmployee(sam);

    }

    public static void showEmployee(Employee employeeObject)
    {
           System.out.println(employeeObject.getName());
           System.out.println(employeeObject.getHireDate());
    }
}
```

A SalariedEmployee is an Employee.

An HourlyEmployee is an Employee.

Sample Dialogue

```
joe's longer name is Josephine
showEmployee(joe) invoked:
Josephine
Jan 1, 2004
showEmployee(sam) invoked:
Sam
Feb 1, 2003
```

Remember that these terms refer to the collections of objects of the derived class and the base class and not to the number of instance variables or methods. A derived class is a *subclass* (*not a superclass*) of its base class. Another way to remember which is a superclass is as follows: Recall that the super constructor invocation is an invocation of the base class and so the base class is the *superclass*.

Self-Test Exercises

5. Is the following program legal? The relevant classes are defined in Displays 7.2, 7.3, and 7.5.

```
public class EmployeeDemo
{
    public static void main( String[] args)
    {
        HourlyEmployee joe =
             new HourlyEmployee("Joe Young",
                         new Date("Feb", 1, 2004), 10.50, 40);
        SalariedEmployee boss =
             new SalariedEmployee("Mr. Big Shot",
                            new Date("Jan", 1, 1999), 100000);
        printName(joe);
        printName(boss);
    }

    public static void printName(Employee object)
    {
        System.out.println(object.getName());
    }
}
```

6. Give a definition for a class `TitledEmployee` that is a derived class of the base class `SalariedEmployee` given in Display 7.5. The class `TitledEmployee` has one additional instance variable of type `String` called `title`. It also has two additional methods: `getTitle`, which takes no arguments and returns a `String`, and `setTitle`, which is a `void` method that takes one argument of type `String`. It also overrides (redefines) the method definition for `getName`, so that the string returned includes the title as well as the name of the employee.

7.2 Encapsulation and Inheritance

Ignorance is bliss.

Proverb

This section is a continuation of Section 7.1 and uses the same example classes we used in Section 7.1. In this section we consider how the information-hiding facilities of Java, primarily the `private` modifier, interact with inheritance.

Pitfall

Use of Private Instance Variables from the Base Class

An object of the class HourlyEmployee (Display 7.3) inherits, among other things, an instance variable called name from the class Employee (Display 7.2). For example, the following would set the value of the instance variable name of the HourlyEmployee object joe to "Josephine":

```
joe.setName("Josephine");
```

But, you must be a bit careful about how you manipulate inherited instance variables such as name. The instance variable name of the class HourlyEmployee was inherited from the class Employee, but the instance variable name is a private instance variable in the definition of the class Employee. That means that name can only be accessed by name within the definition of a method in the class Employee. An instance variable (or method) that is private in a base class is not accessible *by name* in the definition of a method in *any other class, not even in a method definition of a derived class.*

For example, notice the following method definition taken from the definition of the class HourlyEmployee in Display 7.3:

```
public String toString()
{
    return (getName() + " " + getHireDate().toString()
      + "\n$" + wageRate + " per hour for " + hours + " hours");
}
```

You might wonder why we needed to use the methods getName and getHireDate. You might be tempted to rewrite the method definition as follows:

```
public String toString()   //Illegal version
{
    return (name + " " + hireDate.toString()
     + "\n$" + wageRate + " per hour for " + hours + " hours");
}
```

As the comment indicates, this will not work. The instance variables name and hireDate are private instance variables in the class Employee, and although a derived class like HourlyEmployee inherits these instance variables, it cannot access them directly. You must instead use some public methods to access the instance variable name or hireDate, as we did in Display 7.3.

In the definition of a derived class, you cannot mention a private inherited instance variable by name. You must instead use public accessor and mutator methods (such as getName and setName) that were defined in the base class.

The fact that a private instance variable of a base class cannot be accessed in the definition of a method of a derived class often seems wrong to people. After all, if you are an hourly employee and you want to change your name, nobody says, “Sorry, name is a private instance variable of the class Employee.” After all, if you are an hourly employee, you are also an employee. In Java, this is also true; an object of the class `HourlyEmployee` is also an object of the class `Employee`. However, the laws on the use of private instance variables and methods must be as we described, or else they would be compromised. If private instance variables of a class were accessible in method definitions of a derived class, then anytime you wanted to access a private instance variable, you could simply create a derived class and access it in a method of that class, and that would mean that all private instance variables would be accessible to anybody who wants to put in a little extra effort. This scenario illustrates the problem, but the big problem is unintentional errors, not intentional subversion. If private instance variables of a class were accessible in method definitions of a derived class, then the instance variables might be changed by mistake or in inappropriate ways. (Remember, accessor and mutator methods can guard against inappropriate changes to instance variables.)

We will discuss one possible way to get around this restriction on private instance variables of the base class in the upcoming subsection entitled "Protected and Package Access."

Self-Test Exercises

7. Would the following be legal for the definition of a method to add to the class `Employee` (Display 7.2)? (Remember, the question is whether it is legal, not whether it is sensible.)

```
public void crazyMethod()
{
    Employee object = new Employee("Joe",
                                   new Date("Jan", 1, 2005));
    System.out.println("Hello " + object.name);
}
```

   Would it be legal to add this `crazyMethod` to the class `HourlyEmployee`?

8. Suppose you change the modifier before the instance variable `name` from `private` to `public` in the class `Employee`. Would it then be legal to add the method `crazyMethod` (from exercise 7) to the class `HourlyEmployee`?

Pitfall

Private Methods Are Effectively Not Inherited

As we noted in the previous Pitfall section: An instance variable (or method) that is private in a base class is not directly accessible outside of the definition of the base class, *not even in a method definition for a derived class.* The private methods of the base class are just like private variables in terms of not being directly available. But in the case of methods, the restriction is more dramatic. A private variable can be accessed indirectly via an accessor or mutator method. A private method is simply not available. It is just as if the private method were not inherited. (In one sense, private methods in the base class may be indirectly available in the derived class. If a private method is used in the definition of a public method of the base class, then that public method can be invoked in the derived class, or any other class, so the private method can be indirectly invoked.)

This should not be a problem. Private methods should just be used as helping methods, so their use should be limited to the class in which they are defined. If you want a method to be used as a helping method in a number of inherited classes, then it is not *just* a helping method, and you should make the method public.

PROTECTED AND PACKAGE ACCESS

As you have seen, you cannot access (by name) a private instance variable or private method of the base class within the definition of a derived class. There are two classifications of instance variables and methods that allow them to be accessed by name in a derived class. The two classifications are *protected access*, which always gives access, and *package access*, which gives access if the derived class is in the same package as the base class.

If a method or instance variable is modified by `protected` (rather than `public` or `private`), then it can be accessed by name inside its own class definition, it can be accessed by name inside any class derived from it, and it can also be accessed by name in the definition of any class in the same package (even if the class in the same package is not derived from it). However, the `protected` method or instance variable cannot be accessed by name in any other classes. Thus, if an instance variable is marked `protected` in the class `Parent` and the class `Child` is derived from the class `Parent`, then the instance variable can be accessed by name inside any method definition in the class `Child`. However, in a class that is not in the same package as `Parent` and is not derived from `Parent`, it is as if the `protected` instance variable were `private`.

For example, consider the class `HourlyEmployee` that was derived from the base class `Employee`. We were required to use accessor and mutator methods to manipulate the inherited instance variables in the definition of `HourlyEmployee`. For example, consider the definition of the `toString` method of the class `HourlyEmployee`, which we repeat here:

```
public String toString()
{
    return (getName() + " " + getHireDate().toString()
      + "\n$" + wageRate + " per hour for " + hours + " hours");
}
```

If the private instance variables `name` and `hireDate` had been marked `protected` in the class `Employee`, the definition of `toString` in the derived class `HourlyEmployee` could be simplified to the following:

```
public String toString() //Legal if instance variables in
                         // Employee are marked protected
{
```

```
        return (name + " " + hireDate.toString()
         + "\n$" + wageRate + " per hour for " + hours + " hours");
    }
```

The protected Modifier

If a method or instance variable is modified by `protected` (rather than `public` or `private`), then it can be accessed by name inside its own class definition, it can be accessed by name inside any class derived from it, and it can also be accessed by name in the definition of any class in the same package.

The `protected` modifier provides very weak protection compared to the `private` modifier, since it allows direct access to any programmer who is willing to go through the bother of defining a suitable derived class. Many programming authorities discourage the use of the `protected` modifier. Instance variables should normally not be marked `protected`. On rare occasions, you may want to have a method marked `protected`. If you want an access intermediate between `public` and `private`, then the access described in the next paragraph is often a preferable alternative to `protected`.

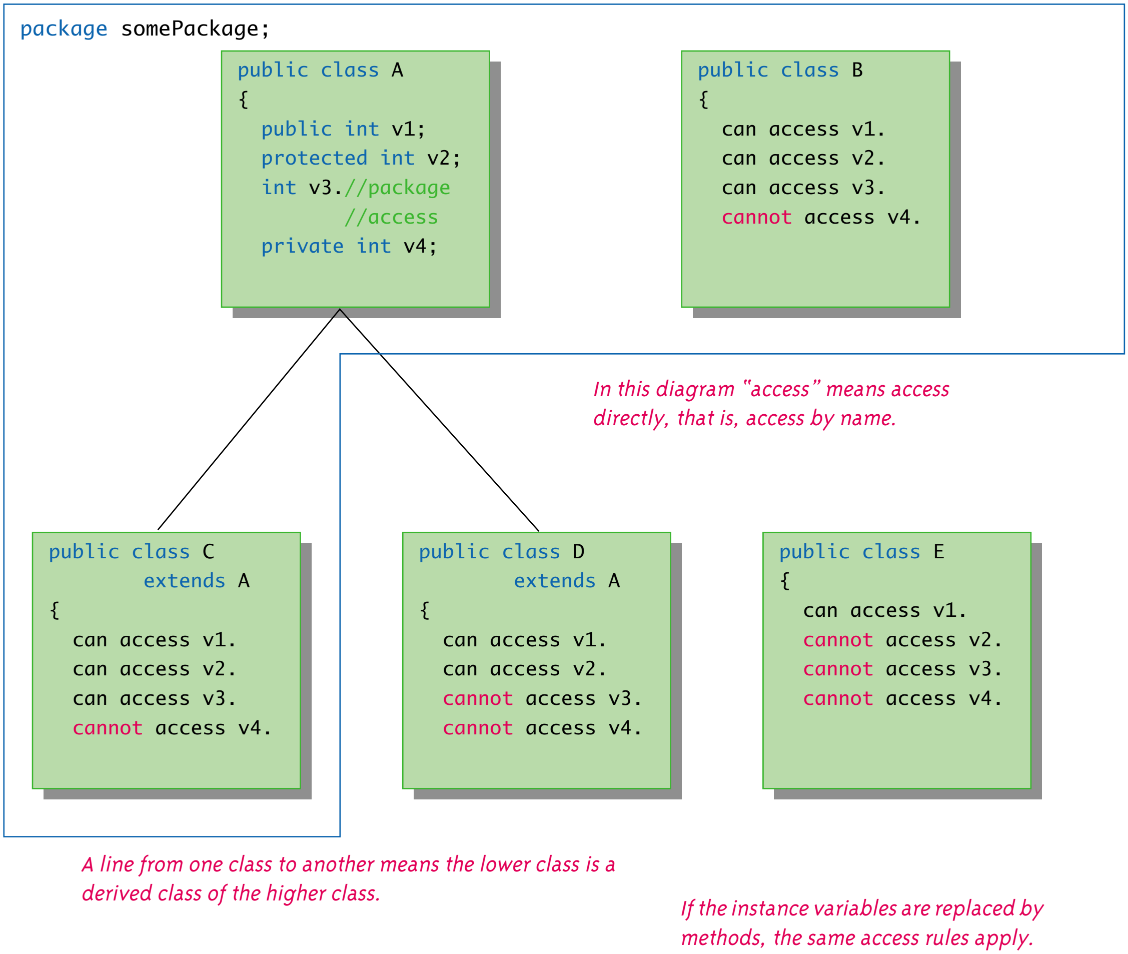
You may have noticed that if you forget to place one of the modifiers `public`, `private`, or `protected` before an instance variable or method definition, then your class definition will still compile. If you do not place any of the modifiers `public`, `private`, or `protected` before an instance variable or method definition, then the instance variable or method can be accessed by name inside the definition of any class in the same package, but not outside of the package. This is called **package access**, **default access**, or **friendly access**. You use package access in situations where you have a package of cooperating classes that act as a single encapsulated unit. Note that package access is more restricted than `protected`, and that package access gives more control to the programmer defining the classes. If you control the package directory (folder), then you control who is allowed package access.

The diagram in Display 7.7 may help you to understand who has access to members with public, private, protected, and package access. The diagram tells who can (directly) access (by name) variables that have public, private, protected, and package access. The same access rules apply to methods that have public, private, protected, and package access.

Package Access

If you do not place any of the modifiers `public`, `private`, or `protected` before an instance variable or method definition, then the instance variable or method is said to have **package access**. Package access is also known as **default access** and as **friendly access**. If an instance variable or method has package access, it can be accessed by name inside the definition of any class in the same package, but not outside of the package.

Display 7.7 Access Modifiers



Pitfall

Forgetting about the Default Package

When considering package access, do not forget the default package. Recall that all the classes in your current directory (that do not belong to some other package) belong to an unnamed package called the *default package*. So, if a class in your current directory is not in any other package, then it is in the default package. So if an instance variable or method has package access, then that instance variable or method can be accessed by name in the definition of any other class in the default package.

Pitfall

A Restriction on Protected Access ✜

The situation described in this pitfall does not occur often, but when it does occur it can be very puzzling if you do not understand what is going on.

Suppose class D is derived from class B, the instance variable n has protected access in class B, and the classes D and B are in different packages, so the class definitions begin as follows:

```
package one;
public class B
{
    protected int n;
     ...
}

package two;
import one.B;
public class D extends B
{
     ...
}
```

Then, the following is a legitimate method that can appear in the definition of class D:

```
public void demo()
{
    n = 42;//n is inherited from B.
}
```

The following is also a legitimate method definition for the derived class D:

```
public void demo2()
{
    D object = new D();
    object.n = 42;//n is inherited from B.
}
```

However, the following is not allowed as a method of D:

```
public void demo3()
{
    B object = new B();
    object.n = 42; //Error
}
```

The compiler will give an error message saying n is protected in B.

Similar remarks apply to protected methods.

A class knows about its own classes' inherited variables and methods, but cannot directly access any instance variables or methods of an ancestor class unless they are public. In the above example, n is an instance variable of B and an instance variable of the derived class D. D can access n whenever n is used as an instance variable of D, but D cannot access n when n is used as an instance variable of B.

If the classes B and D are in the same package, you would not get the error message because, in Java, protected access implies package access. In particular, if the classes B and D are both in the default package, you would not get the error message.

Self-Test Exercises

9. Suppose you change the modifier before the instance variable `name` from `private` to `protected` in the class `Employee` (Display 7.2). Would it then be legal to add the method `crazyMethod` (from Self-Test Exercise 7) to the class `HourlyEmployee` (Display 7.3)?

10. Which is more restricted, protected access or package access?

11. Suppose class `D` is derived from class `B`, the method `doStuff()` has protected access in class `B`, and the classes `D` and `B` are in different packages, so the class definitions begin as follows:

```
package one;
public class B
{
    protected void doStuff()
    {
     ...
}

package two;
import one.B;
public class D extends B
{
     ...
}
```

Is the following a legitimate method that can appear in the definition of the class `D`?

```
public void demo()
{
    doStuff();//doStuff is inherited from B.
}
```

12. Suppose `B` and `D` are as described in exercise 11. Is the following a legitimate method that can appear in the definition of the class `D`?

```
public void demo2()
{
    D object = new D();
    object.doStuff();//doStuff is inherited from B.
}
```

13. Suppose `B` and `D` are as described in exercise 11. Is the following a legitimate method that can appear in the definition of the class `D`?

```
public void demo3()
{
    B object = new B();
    object.doStuff();
}
```

7.3 Programming with Inheritance

The devil is in the details.

Common saying

In the previous section we described the basic idea and basic details about derived classes. In this section we continue that discussion and go on to discuss some more subtle points about derived classes. In the process we also discuss the class `Object`, which is an ancestor class of all Java classes, and we describe a better way to define an `equals` method.

Tip

"Is a" Versus "Has a"

Early in this chapter we defined a derived class called `HourlyEmployee` using the class `Employee` as the base class. In such a case an object of the derived class `HourlyEmployee` is also an instance of the class `Employee`, or, stated more simply, an `HourlyEmployee` *is an*

Employee. This is an example of the "is a" relationship between classes. It is one way to make a more complex class out of a simpler class.

"is a" relationship

Another way to make a more complex class out of a simpler class is known as the "has a" relationship. For example, the class Employee defined earlier has an instance variable of the class type Date. We express this relationship by saying an Employee "has a" Date. Using the "has a" relationship to build a class (such as building the class Employee by using Date as an instance variable) is often called **composition**.

"has a" relationship

composition

Since the class HourlyEmployee inherits the instance variable of type Date from the class Employee, it is also correct to say an HourlyEmployee "has a" Date. Thus, an HourlyEmployee *is an* Employee and *has a* Date.

Tip

Static Variables Are Inherited

Static variables in a base class are inherited by any derived classes. The modifiers public, private, and protected, and package access have the same meaning for static variables as they do for instance variables.

ACCESS TO A REDEFINED BASE METHOD

Suppose you redefine a method so that it has a different definition in the derived class from what it has in the base class. The definition that was given in the base class is not completely lost to the derived class objects. However, if you want to invoke the version of the method given in the base class with an object in the derived class, you need some way to say "use the definition of this method as given in the base class (even though I am an object of the derived class)." The way you say this is to use the keyword super as if it were a calling object.

super relationship

For example, the method toString of the class HourlyEmployee (Display 7.3) was defined as follows:

```
public String toString() //in the derived class HourlyEmployee
{
    return (getName() + " " + getHireDate().toString()
      + "\n$" + wageRate + " per hour for " + hours + " hours");
}
```

This overrides the following definition of toString() that was given in the definition of the base class Employee:

```
public String toString() //in the base class Employee
{
```

```
    return (name + " " + hireDate.toString());
}
```

We can use the version of the method `toString()` defined in the base class `Employee` to simplify the definition of the method `toString()` in the derived class `HourlyEmployee`. The following is an equivalent way to define `toString()` in the derived class `HourlyEmployee`:

```
public String toString() //in the derived class HourlyEmployee
{
   return (super.toString()
      + "\n$" + wageRate + " per hour for " + hours + " hours");
}
```

The expression `super.toString()` is an invocation of the method `toString()` using the definition of `toString()` given in the base class `Employee`.

You can only use `super` in this way within the definition of a method in a derived class. Outside of the definition of the derived class you cannot invoke an overridden method of the base class using an object of the derived class.

INVOKING THE OLD VERSION OF AN OVERRIDDEN METHOD?

Within the definition of a method of a derived class, you can invoke the base class version of an overridden method of the base class by prefacing the method name with `super` and a dot. Outside of the derived class definition, there is no way to invoke the base class version of an overridden method using an object of the derived class.

EXAMPLE:

```
public String toString()
{
   return (super.toString()
       + "\n$" + wageRate + " per hour for " + hours + " hours");
}
```

Pitfall

YOU CANNOT USE MULTIPLE `super`s

As we already noted, within the definition of a method of a derived class, you can call an overridden method of the base class by prefacing the method name with `super` and a dot. However, you cannot repeat the use of `super` to invoke a method from some ancestor class other than a direct parent. For example, suppose that the class `Employee` were derived from the class `Person`, and the class `HourlyEmployee` is derived from the class `Employee`. You might think that you can invoke a method of the class `Person` within the definition of the class `HourlyEmployee`, by using `super.super`, as in

```
super.super.toString()//ILLEGAL!
```

However, as the comment indicates, it is illegal to have such multiple `super`s in Java.

Self-Test Exercises

14. Redefine the `toString` method of the class `SalariedEmployee` (Display 7.5) so that it uses `super.toString()`. This new definition of `toString` will be equivalent to the one given in Display 7.5.

15. Redefine the `equals` method for the class `HourlyEmployee` (Display 7.3) using `super.equals` to invoke the `equals` method of the base class `Employee`.

16. Is the following program legal? The relevant classes are defined in Displays 7.2 and 7.3.

```
public class EmployeeDemo
{
    public static void main( String[] args)
    {
        HourlyEmployee joe =
             new HourlyEmployee("Joe Young",
                         new Date("Feb", 1, 2004), 10.50, 40);
        String nameNDate = joe.super.toString();
        System.out.println(nameNDate);
    }
}
```

17. Suppose you add the following defined constant to the class `Employee` (Display 7.2):

```
public static final int STANDARD_HOURS = 160;//per month
```

Would it then be legal to add the following method to the class `HourlyEmployee` (Display 7.3)?

```
public void setHoursToStandard()
{
    hours = STANDARD_HOURS;
}
```

THE CLASS `Object`

Object class

Java has a class that is an ancestor of every class. In Java, every class is a derived class of a derived class of . . . (for some number of iterations of "a derived class of") of the class `Object`. So, every object of every class is of type `Object`, as well as being of the type of its class (and also of the types of all its ancestor classes). Even classes that you define yourself are descendent classes of the class `Object`. If you do not make your class a derived class of some class, then Java will automatically make it a derived class of the class `Object`.

THE CLASS `Object`

In Java, every class is a descendent of the class `Object`. So, every object of every class is of type `Object`, as well as being of the type of its class.

The class `Object` allows you to write Java code for methods with a parameter of type `Object` that can be replaced by an object of any class whatsoever. You will eventually encounter library methods that accept an argument of type `Object` and hence can be used with an argument that is an object of absolutely any class.

The class `Object` is in the package `java.lang`, which is always imported automatically. So, you do not need any `import` statement to make the class `Object` available to your code.

The class `Object` does have some methods that every Java class inherits. For example, every object inherits the methods `equals` and `toString` from some ancestor class, which either is the class `Object` or a class that itself inherited the methods ultimately from the class `Object`. However, the inherited methods `equals` and `toString` will not work correctly for (almost) any class you define. You need to override the inherited method definitions with new, more appropriate definitions.

toString

It is important to include definitions of the methods `toString` and `equals` in the classes you define, since some Java library classes assume your class has such methods. There are no subtleties involved in defining (actually redefining or overriding) the method `toString`. We have seen good examples of the method `toString` in many of our class definitions. The definition of the overridden method `equals` does have some subtleties and we will discuss them in the next subsection.

equals

clone

Another method inherited from the class `Object` is the method `clone`, which is intended to return a copy of the calling object. We discuss the `clone` method in Chapters 8 and 13.

THE RIGHT WAY TO DEFINE `equals`

Earlier we said that the class `Object` has an `equals` method, and that when you define a class with an `equals` method you should override the definition of the method `equals` given in the class `Object`. However, we did not, strictly speaking, follow our own advice. The heading for the method `equals` in our definition of the class `Employee` (Display 7.2) is as follows:

```
public boolean equals(Employee otherEmployee)
```

On the other hand, the heading for the method `equals` in the class `Object` is as follows:

```
public boolean equals(Object otherObject)
```

The two `equals` methods have different parameter types, so we have not overridden the definition of `equals`. We have merely overloaded the method `equals`. The class `Employee` has both of these methods named `equals`.

In most situations, this will not matter. However, there are situations in which it does matter. Some library methods assume your class's definition of `equals` has the following heading, the same as in the class `Object`:

```
public boolean equals(Object otherObject)
```

We need to change the type of the parameter for the `equals` method in the class `Employee` from type `Employee` to type `Object`. A first try might produce the following:

```
public boolean equals(Object otherObject)
{
    Employee otherEmployee = (Employee)otherObject;
    return (name.equals(otherEmployee.name)
               && hireDate.equals(otherEmployee.hireDate));
}
```

We needed to type cast the parameter `otherObject` from type `Object` to type `Employee`. If we omit the type cast and simply proceed with `otherObject`, the compiler will give an error message when it sees

```
otherObject.name
```

The class `Object` does not have an instance variable named `name`.

This first try at an improved `equals` method does override the definition of `equals` given in the class `Object` and will work well in many cases. However, it still has a shortcoming.

Our definition of `equals` now allows an argument that can be any kind of object whatsoever. What happens if the method `equals` is used with an argument that is not an `Employee`? The answer is that a run-time error will occur when the type cast to `Employee` is executed.

We need to make our definition work for any kind of object. If the object is not an `Employee`, we simply return `false`. The calling object is an `Employee`, so if the argument is not an `Employee`, they should not be considered equal. But how can we tell whether the parameter is or is not of type `Employee`?

Every object inherits the method `getClass()` from the class `Object`. The method `getClass()` is marked `final` in the class `Object`, so it cannot be overridden. For any object `o`, `o.getClass()` returns a representation of the class used to create `o`. For example, after the following is executed:

```
o = new Employee();
```

`o.getClass()` returns a representation `Employee`.

We will not describe the details of this representation except to say that two such representations should be compared with `==` or `!=` if you want to know if two representations are the same. Thus,

```
if (object1.getClass() == object2.getClass())
    System.out.println("Same class.");
else
    System.out.println("Not the same class.");
```

will output `Same class` if `object1` and `object2` were created with the same class when they were created using `new`, and output `Not same class` otherwise.

Our final version of the method `equals` is shown in Display 7.8. Note that we have also taken care of one more possible case. The predefined constant `null` can be plugged in for a parameter of type `Object`. The Java documentation says that an `equals` method should return `false` when comparing an object and the value `null`. So that is what we have done.

extra code on CD

On the accompanying CD, the subdirectory `improvedEquals` (of the directory for this chapter) has a definition of the class `Employee` that includes this definition of `equals`.

Display 7.8 A Better equals Method for the Class `Employee`

```
public boolean equals(Object otherObject)
{
    if (otherObject == null)
        return false;
    else if (getClass() != otherObject.getClass())
        return false;
    else
    {
        Employee otherEmployee = (Employee)otherObject;
        return (name.equals(otherEmployee.name)
           && hireDate.equals(otherEmployee.hireDate));
    }
}
```

Tip

getClass VERSUS instanceof ✜

Many authors suggest that in the definition of `equals` for a class such as `Employee`, given in Display 7.8, you should not use

```
else if (getClass() != otherObject.getClass())
    return false;
```

but should instead use

```
else if (!(otherObject instanceof Employee))
    return false;
```

What is the difference and which should you use? At first glance it seems like you should use `instanceof` in the definition of `equals`. The `instanceof` operator checks to see if an object is of the type given as its second argument. The syntax is

instanceof

Object `instanceof` *Class_Name*

which returns `true` if *Object* is of type *Class_Name*; otherwise it returns `false`. So, the following will return `true` if `otherObject` is of type `Employee`:

```
(otherObject instanceof Employee)
```

Suppose that (contrary to what we really did) we instead used `instanceof` in our definition of `equals` for the class `Employee` and we also used `instanceof` in our definition for the class `HourlyEmployee`, so that the definition of `equals` for `HourlyEmployee` is as follows:

```
public boolean equals(Object otherObject)
//This is NOT the right way to define equals.
{
    if (otherObject == null)
        return false;
    else if (!(otherObject instanceof HourlyEmployee))
        return false;
    else
    {
        HourlyEmployee otherHourlyEmployee =
                              (HourlyEmployee)otherObject;
         return (super.equals(otherHourlyEmployee)
              && (wageRate == otherHourlyEmployee.wageRate)
              && (hours == otherHourlyEmployee.hours));
    }
}
```

Assuming that the `equals` method for both `Employee` and `HourlyEmployee` are defined using `instanceof` (as above), consider the following situation:

```
Employee e =
      new Employee("Joe Worker", new Date("Jan", 1, 2004));
HourlyEmployee hourlyE = new HourlyEmployee("Joe Worker",
                     new Date("Jan", 1, 2004), 50.50, 160);
```

Then, with the definition of `equals` that uses `instanceof`, we get that

```
e.equals(hourlyE)
```

returns `true`, since `hourlyE` is an `Employee` with the same name and hire date as `e`. So far, it sounds reasonable.

However, since we are assuming that we also used `instanceof` in the definition of `equals` for the class `HourlyEmployee`, we also get that

```
hourlyE.equals(e)
```

returns `false` because `e instanceof HourlyEmployee` returns `false`. (`e` is an `Employee` but `e` is not an `HourlyEmployee`.)

So, if we define `equals` in both classes using `instanceof`, then `e` equals `hourlyE`, but `hourlyE` does not equal `e`. And, that's no way for `equals` to behave.

Since `instanceof` does not yield a suitable definition of `equals`, you should instead use `getClass()` as we did in Display 7.8. If we use `getClass()` in a similar way in the definition of `equals` for the class `HourlyEmployee` (see Self-Test Exercise 19), then

```
e.equals(hourlyE)
```

and

```
hourlyE.equals(e)
```

both return `false`.

Self-Test Exercises

18. ✜ Redefine the method `equals` given in Display 7.8 using `instanceof` instead of `getClass()`. Give the complete definition. Remember, we do not want you to define `equals` this way in your class definitions; this is just an exercise.

19. Redefine the `equals` method of the class `HourlyEmployee` (Display 7.3) so that it has a parameter of type `Object` and follows the other guidelines we gave for an `equals` method. Assume the definition of the method `equals` for the class `Employee` has been changed to be as in Display 7.8. (Remember, you should use `getClass()`, not `instanceof`.)

`instanceof` AND `getClass()`

Both the `instanceof` operator and the `getClass()` method can be used to check the class of an object. The `instanceof` operator simply tests the class of an object. The `getClass()` method used in a test with == or != tests if two objects were created with the same class. The details follow.

THE `instanceof` OPERATOR

The `instanceof` operator checks if an object is of the type given as its second argument. The syntax is

```
Object instanceof Class_Name
```

which returns `true` if *Object* is of type *Class_Name*; otherwise it returns `false`. So, the following will return `true` if `otherObject` is of type `Employee`:

```
(otherObject instanceof Employee)
```

Note that this means it returns `true` if `otherObject` is of the type of any descendent class of `Employee`, because in that case `otherObject` is also of type `Employee`.

THE `getClass()` METHOD

Every object inherits the method `getClass()` from the class `Object`. The method `getClass()` is marked `final` in the class `Object`, so it cannot be overridden. For any `object` of any class,

```
object.getClass()
```

returns a representation of the class that was used with new to create `object`. Any two such representations can be compared with == or != to determine whether or not they represent the same class. Thus,

```
if (object1.getClass() == object2.getClass())
    System.out.println("Same class.");
else
    System.out.println("Not the same class.");
```

will output `Same class` if `object1` and `object2` were created with the same class when they were created using new, and output `Not same class` otherwise.

EXAMPLES:

Suppose that `HourlyEmployee` is a derived class of `Employee` and that `employeeObject` and `hourlyEmployeeObject` are created as follows:

```
Employee employeeObject = new Employee();
HourlyEmployee hourlyEmployeeObject = new HourlyEmployee();
```

Then:

```
employeeObject.getClass() == hourlyEmployeeObject.getClass()
```

returns `false`.

```
employeeObject instanceof Employee
```

returns `true`.

```
hourlyEmployeeObject instanceof Employee
```

returns `true`.

```
employeeObject instanceof HourlyEmployee
```

returns `false`.

```
hourlyEmployeeObject instanceof HourlyEmployee
```

returns `true`.

20. Redefine the `equals` method of the class `SalariedEmployee` (Display 7.5) so that it has a parameter of type `Object` and follows the other guidelines we gave for an `equals` method. Assume the definition of the method `equals` for the class `Employee` has been changed to be as in Display 7.8. (Remember, you should use `getClass()`, not `instanceof`.)

21. Redefine the `equals` method of the class `Date` (Display 4.11) so that it has a parameter of type `Object` and follows the other guidelines we gave for an `equals` method. (Remember, you should use `getClass()`, not `instanceof`.)

22. What is the output produced by the following program? (The classes `Employee` and `HourlyEmployee` were defined in this chapter.)

```
public class Test
{
    public static void main(String[] args)
    {
        Employee object1 = new Employee();
        Employee object2 = new HourlyEmployee();

        if (object1.getClass( ) == object2.getClass( ))
            System.out.println("Same class.");
```

```
            else
                System.out.println("Not the same class.");
        }
    }
```

Chapter Summary

- Inheritance provides a tool for code reuse by deriving one class from another. The derived class automatically inherits the features of the old (base) class and may add features as well.
- A derived class object inherits the instance variables, static variables, and public methods of the base class, and may add additional instance variables, static variables, and methods.
- An object of a derived class has the type of the derived class, and it also has the type of the base class, and more generally, has the type of every one of its ancestor classes.
- If an instance variable is marked `private` in a base class, then it cannot be accessed by name in a derived class.
- Private methods are effectively not inherited.
- A method may be redefined in a derived class so that it performs differently from how it performs in the base class. This is called *overriding* the method definition. The definition for an overridden method is given in the class definition of the derived class, in the same way as the definitions of any added methods.
- A constructor of a base class can be used in the definition of a constructor for a derived class. The keyword `super` is used as the name for a constructor of the base class.
- A constructor definition can use the keyword `this`, as if it were a method name, to invoke a constructor of the same class.
- If a constructor does not contain an invocation of either `super` or `this`, then Java automatically inserts an invocation of `super()` as the first action in the body of the constructor definition.
- A `protected` instance variable or method in the base class can be accessed by name in the definition of a method of a derived class and in the definition of any method in the same package.
- If an instance variable or method has none of the modifiers `public`, `private`, or `protected`, then it is said to have *package access*. An instance variable or method with package access can be accessed by name in the definition of any method in the same package.
- The class `Object` is an ancestor class of every class in Java.
- The `equals` method for a class should have `Object` as the type of its one parameter.

ANSWERS TO SELF-TEST EXERCISES

1. Yes, it will have the instance variables. A derived class has all the instance variables that the base class has and can add more instance variables besides.

2. Yes, it will have the methods. A derived class has all the public methods that the base class has and can also add more methods. If the derived class does not override (redefine) a method definition, then it performs exactly the same action in the derived class as it does in the base class. However, the base class can contain an overriding definition of (a new definition of) a method and the new definition will replace the old definition (provided it has the same number and types of parameters).

3. The class `DiscountSale` will have two methods named `getTax` and they will have the following two headings. This is an example of overloading.

```
public double getTax()
public double getTax(double rate)
```

4. The method `getName` is inherited from the class `Employee` and so needs no definition. The method `getRate` is a new method added in the class `HourlyEmployee` and so needs to be defined.

5. Yes. You can plug in an object of a derived class for a parameter of the base class type. An `HourlyEmployee` is an `Employee`. A `SalariedEmployee` is an `Employee`.

6.
```
public class TitledEmployee extends SalariedEmployee
{
    private String title;

    public TitledEmployee()
    {
        super(null, null, 0);
        title = null;
    }

    public TitledEmployee(String theName, String theTitle,
                               Date theDate, double theSalary)
    {
        super(theName, theDate, theSalary);
        title = theTitle;
    }

    public String getTitle()
    {
        return title;
    }

    public void setTitle(String theTitle)
    {
        title = theTitle;
```

```
    }

    public String getName()
    {
        return (title + super.getName());
    }
}
```

7. It would be legal to add `crazyMethod` to the class `Employee`. It would not be legal to add `crazyMethod` to the class `HourlyEmployee` because, although the class `HourlyEmployee` has an instance variable `name`, `name` is private in the base class `Employee` and so cannot be accessed by name in `HourlyEmployee`.

8. Yes, it would be legal as long as `name` is marked `public` in the base class `Employee`.

9. Yes, it would be legal as long as `name` is marked `protected` in the base class `Employee`.

10. Package access is more restricted. Anything allowed by package access is also allowed by protected access, but protected access allows even more.

11. Yes, it is legitimate.

12. Yes, it is legitimate.

13. No, it is not legitimate. The compiler will give an error message saying `doStuff()` is protected in `B`.

14.
```
public String toString()
{
        return (super.toString()
                          + "\n$" + salary + " per year");
}
```

15.
```
public boolean equals(HourlyEmployee other)
{
    return (super.equals(other)
               && wageRate == other.wageRate
               && hours == other.hours);
}
```

We will give an even better definition of `equals` for the class `HourlyEmployee` later in this chapter.

16. It is not legal. You cannot use `super` in this way. `super.toString()` as used here refers to `toString()` in the class `Employee` and can only be used in definitions of classes derived from `Employee`. Moreover, you cannot have a calling object, such as `joe`, before `super`, so this is even illegal if you add `extends Employee` to the first line of the class definition.

17. Yes, all static variables are inherited. Since a defined constant is a form of static variable, it is inherited. So, the class `HourlyEmployee` inherits the constant `STANDARD_HOURS` from the class `Employee`.

18.
```
public boolean equals(Object otherObject)
//This is NOT the right way to define equals.
{
    if (otherObject == null)
        return false;
    else if (!(otherObject instanceof Employee))
        return false;
    else
    {
        Employee otherEmployee = (Employee)otherObject;
        return (name.equals(otherEmployee.name)
           && hireDate.equals(otherEmployee.hireDate));
    }
 }
```

extra code on CD

19. A version of the `HourlyEmployee` class with this definition of `equals` is in the subdirectory `improvedEquals` of the `ch07` directory on the accompanying CD.

```
public boolean equals(Object otherObject)
{
    if (otherObject == null)
        return false;
    else if (getClass() != otherObject.getClass())
        return false;
    else
    {
        HourlyEmployee otherHourlyEmployee =
                               (HourlyEmployee)otherObject;
         return (super.equals(otherHourlyEmployee)
              && (wageRate == otherHourlyEmployee.wageRate)
              && (hours == otherHourlyEmployee.hours));
    }
}
```

extra code on CD

20. A version of the `SalariedEmployee` class with this definition of `equals` is in the subdirectory `improvedEquals` of the `ch07` directory on the accompanying CD.

```
public boolean equals(Object otherObject)
{
    if (otherObject == null)
        return false;
    else if (getClass() != otherObject.getClass())
        return false;
    else
    {
        SalariedEmployee otherSalariedEmployee =
                               (SalariedEmployee)otherObject;
```

```
            return (super.equals(otherSalariedEmployee)
                  && (salary == otherSalariedEmployee.salary));
        }
    }
```

21. A version of the `Date` class with this definition of `equals` is in the subdirectory `improved–Equals` of the `ch07` directory on the accompanying CD. extra code on CD

```
public boolean equals(Object otherObject)
{
  if (otherObject == null)
    return false;
  else if (getClass() != otherObject.getClass())
    return false;
    else
   {
        Date otherDate =
                      (Date)otherObject;
        return ( month.equals(otherDate.month)
                  && (day == otherDate.day)
                  && (year == otherDate.year) );
    }
}
```

22. `Not the same class.`

Programming Projects

1. Define a class called `Administrator`, which is a derived class of the class `SalariedEmployee` in Display 7.5. You are to supply the following additional instance variables and methods:

   An instance variable of type `String` that contains the administrator's title (such as `"Director"` or `"Vice President"`).

   An instance variable of type `String` that contains the administrator's area of responsibility (such as `"Production"`, `"Accounting"`, or `"Personnel"`).

   An instance variable of type `String` that contains the name of this administrator's immediate supervisor.

   Suitable constructors, and suitable accessor and mutator methods.

   A method for reading in an administrator's data from the keyboard.

   Override the definitions for the methods `equals` and `toString` so they are appropriate to the class `Administrator`.

   Also, write a suitable test program.

2. Give the definition of a class named `Doctor` whose objects are records for a clinic's doctors. This class will be a derived class of the class `SalariedEmployee` given in Display 7.5. A `Doctor` record has the doctor's specialty (such as `"Pediatrician"`, `"Obstetrician"`, `"General Practitioner"`, and so forth; so use the type `String`) and office visit fee (use type `double`). Be sure your class has a reasonable complement of constructors, accessor and mutator methods, and suitably defined `equals` and `toString` methods. Write a program to test all your methods.

3. Create a class called `Vehicle` that has the manufacturer's name (type `String`), number of cylinders in the engine (type `int`), and owner (type `Person` given below). Then, create a class called `Truck` that is derived from `Vehicle` and has the following additional properties: the load capacity in tons (type `double` since it may contain a fractional part) and towing capacity in pounds (type `int`). Be sure your class has a reasonable complement of constructors, accessor and mutator methods, and suitably defined `equals` and `toString` methods. Write a program to test all your methods.

   The definition of the class `Person` is below. Completing the definitions of the methods is part of this programming project.

```
public class Person
{
    private String name;

    public Person()
    {...}
    public Person(String theName)
    {...}
    public Person(Person theObject)
    {...}
    public String getName()
    {...}
    public void setName(String theName)
    {...}
    public String toString()
    {...}
    public boolean equals(Object other)
    {...}
}
```

4. Give the definition of two classes, `Patient` and `Billing`, whose objects are records for a clinic. `Patient` will be derived from the class `Person` given in Programming Project 3. A `Patient` record has the patient's name (inherited from the class `Person`) and primary physician, of type `Doctor` defined in Programming Project 2..A `Billing` object will contain a `Patient` object and a `Doctor` object, and an amount due of type `double`. Be sure your classes have a reasonable complement of constructors, accessor and mutator methods, and suitably defined `equals` and `toString` methods. First write a driver program to test all your methods, then write a test program that creates at least two patients, at least two doctors, and at least two `Billing` records, and then prints out the total income from the `Billing` records.