Module 6

Class Design

Objectives

Upon completion of this module, you should be able to:

- Define *inheritance, polymorphism, overloading, overriding,* and *virtual method invocation*
- Use the access modifiers `protected` and the default (*package-friendly)*
- Describe the concepts of constructor and method overloading
- Describe the complete object construction and initialization operation

This module is the second of three modules that describe the object-oriented paradigm and the object-oriented features of the Java programming language.

Relevance

Discussion – The following question is relevant to the material presented in this module:

How does the Java programming language support object inheritance?

_____

_____

_____

Subclassing

In programming, you often create a model of something (for example, an employee), and then need a more specialized version of that original model. For example, you might want a model for a manager. A manager *is an* employee, but an employee with additional features.

Figure 6-1 shows the UML class diagrams that model the `Employee` and `Manager` classes.

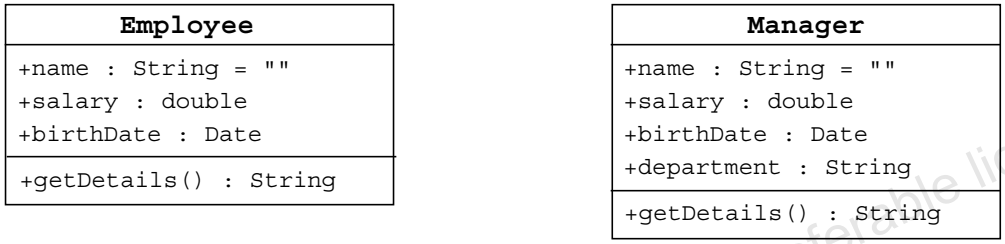

Figure 6-1 Class Diagrams for `Employee` and `Manager`

Code 6-1 and Code 6-2 show possible implementations of the `Employee` and `Manager` classes as they are modeled in Figure 6-1.

Code 6-1 A Possible Implementation of the `Employee` Class

```
public class Employee {
  public String name = "";
  public double salary;
  public Date birthDate;

  public String getDetails() {...}
}
```

Code 6-2 A Possible Implementation of the `Manager` Class

```
public class Manager {
  public String name = "";
  public double salary;
  public Date birthDate;
  public String department;

  public String getDetails() {...}
}
```

This example illustrates the duplication of data between the `Manager` class and the `Employee` class. Additionally, there could be a number of methods applicable to both `Employee` and `Manager`. Therefore, you need a way to create a new class from an existing class; this is called *subclassing*.

In object-oriented languages, special mechanisms are provided that enable you to define a class in terms of a previously defined class. Figure 6-2 shows the UML class diagram in which the `Manager` class is a subclass of the `Employee` class.

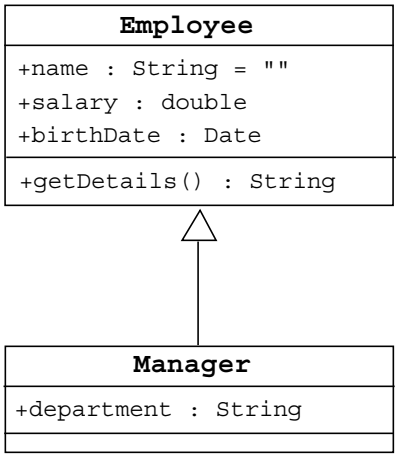

Figure 6-2 Class Diagrams for `Employee` and `Manager` Using Inheritance

Code 6-3 shows an implementation of the `Manager` class that inherits from the `Employee` class as modeled in Figure 6-2.

Code 6-3 Another Implementation of the `Manager` Class

```
public class Manager extends Employee {
  public String department;
}
```

Single Inheritance

The Java programming language permits a class to extend one other class only. This restriction is called *single inheritance*. The relative merits of single and multiple inheritance are the subject of extensive discussions among object-oriented programmers. Module 7, "Advanced Class Features," examines a language feature called *interfaces* that provides most of the benefits of multiple inheritance without suffering from any of its drawbacks.

Figure 6-3 shows the base class `Employee` and three subclasses: `Engineer`, `Manager`, and `Secretary`. The `Manager` is also subclassed by `Director`.

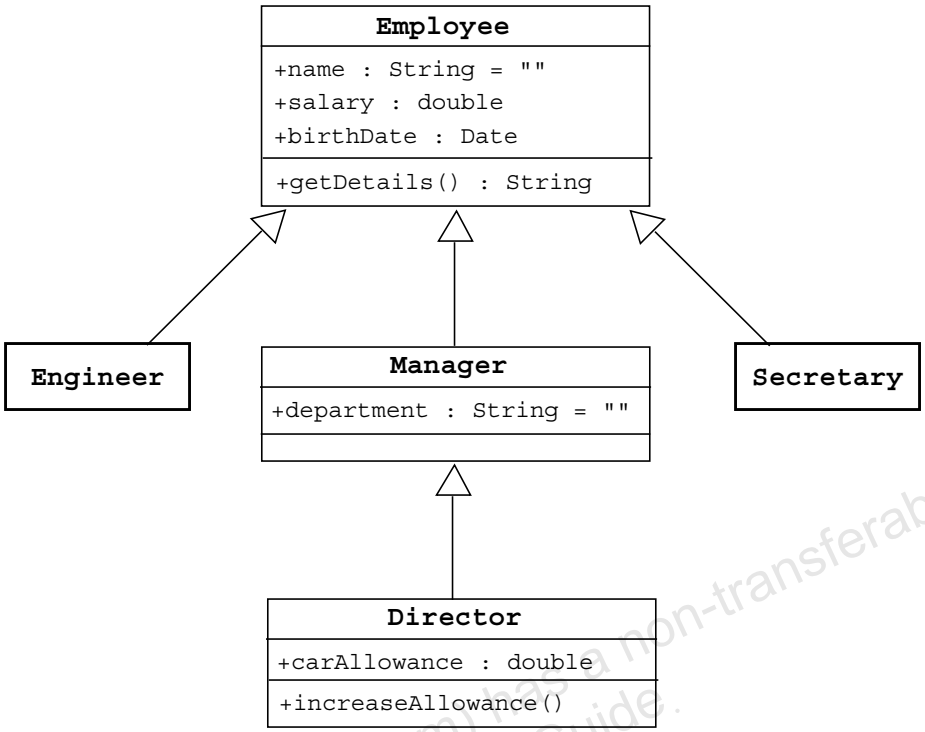

Figure 6-3 An Example Inheritance Tree

The `Employee` class contains three attributes (`name`, `salary`, and `birthDate`), as well as one method (`getDetails`). The `Manager` class inherits all of these members and specifies an additional attribute, `department`, as well as the `getDetails` method. The `Director` class inherits all of the members of `Employee` and `Manager` and specifies a `carAllowance` attribute and a new method, `increaseAllowance`.

Similarly, the `Engineer` and `Secretary` classes inherit the members of the `Employee` class and might specify additional members (not shown).

Access Control

Variables and methods can be at one of four access levels: `public`, `protected`, *default*, or `private`. Classes can be at the `public` or *default* level. Table 6-1 shows the access levels.

Table 6-1 Accessibility Criteria

Modifier	Same Class	Same Package	Subclass	Universe
`private`	Yes			
default	Yes	Yes		
`protected`	Yes	Yes	Yes	
`public`	Yes	Yes	Yes	Yes

A variable or method marked `private` is accessible only by methods that are members of the same class as the `private` variable or method.

A variable, method, or class has default accessibility if it does not have an explicit access modifier as part of its declaration. Such accessibility means that access is permitted from any method in classes that are members of the *same package* as the target. This is often called *package-friendly* or *package-private*.

A variable or method marked with the modifier `protected` is actually more accessible than one with default access control. A `protected` method or variable is accessible from methods in classes that are members of the same package and from any method in any *subclass*. You should use the protected access when it is appropriate for a class's subclass, but not unrelated classes.

A variable or method marked with the modifier `public` is accessible universally.

Note – Protected access is provided to subclasses that reside in a different package from the class that owns the protected feature.

Overriding Methods

In addition to producing a new class based on an old one by adding additional features, you can modify existing behavior of the parent class.

If a method is defined in a subclass so that the name, return type, and argument list match exactly those of a method in the parent class, then the new method is said to *override* the old one.

Note – In J2SE version 5.0, these matching rules have changed slightly. The return type of the overriding method can now be a subclass of the inherited method. A discussion of covariant returns is beyond the scope of this course.

Consider these sample methods in the `Employee` and `Manager` classes:

```
public class Employee {
  protected String name;
  protected double salary;
  protected Date birthDate;

  public String getDetails() {
    return "Name: " + name + "\n"
         + "Salary: " + salary;
  }
}

public class Manager extends Employee {
  protected String department;

  public String getDetails() {
    return "Name: " + name + "\n"
           + "Salary: " + salary + "\n"
           + "Manager of: " + department;
  }
}
```

The `Manager` class has a `getDetails` method by definition because it inherits one from the `Employee` class. However, the original method has been replaced, or overridden, by the child class's version.

Overridden Methods Cannot Be Less Accessible

Remember that the method name and the order of arguments of a child method must be identical to those of the method in the parent class for that method to override the parent's version. Furthermore, an overriding method cannot be less accessible than the method it overrides.

Consider this invalid scenario:

```
public class Parent {
  public void doSomething() {}
}

public class Child extends Parent {
  private void doSomething() {} // illegal
}

public class UseBoth {
  public void doOtherThing() {
    Parent p1 = new Parent();
    Parent p2 = new Child();
    p1.doSomething();
    p2.doSomething();
  }
}
```

The Java programming language semantics dictate that `p2.method()` results in the `Child` version of method being executed, but, because the method is declared `private`, `p2` (declared as `Parent`) cannot access it. Thus, the language semantics are violated.

Invoking Overridden Methods

A subclass method can invoke a superclass method using the `super` keyword.

The `super` keyword refers to the superclass of the class in which the keyword is used. It is used to refer to the member variables or the methods of the superclass.

This can be achieved using the keyword `super` as follows:

```
public class Employee {
  private String name;
  private double salary;
  private Date birthDate;

  public String getDetails() {
    return "Name: " + name + "\nSalary: " + salary;
  }
}

public class Manager extends Employee {
  private String department;

  public String getDetails() {
    // call parent method
    return super.getDetails()
           + "\nDepartment: " + department;
  }
}
```

A call of the form `super.method()` invokes the entire behavior, along with any side effects of the method that would have been invoked if the object had been of the parent class type. The method does not have to be defined in that parent class; it could be inherited from some class even further up the hierarchy.

Note – In the previous example, member variables have been declared as `private`. This is not necessary but is a generally good programming practice.

Polymorphism

Describing a Manager as *is an* Employee is not just a convenient way of describing the relationship between these two classes. Recall that the Manager has all the members, both attributes and methods, of the parent class Employee. This means that any operation that is legitimate on an Employee is also legitimate on a Manager. If the Employee has the method getDetails, then the Manager class does also.

It might seem unrealistic to create a Manager and assign deliberately the reference to it to a variable of type Employee. However, this is possible, and there are reasons why you might want to achieve this effect.

An *object* has only one form (the one that is given to it when constructed). However, a *variable* is polymorphic because it can refer to objects of different forms.

The Java programming language, like most object-oriented languages, actually permits you to refer to an object with a variable that is one of the parent class types. So you can say:

```
Employee e = new Manager(); //legal
```

Using the variable e as is, you can access only the parts of the object that are part of Employee; the Manager-specific parts are hidden. This is because as far as the compiler is concerned, e is an Employee, not a Manager. Therefore, the following is not permitted:

```
// Illegal attempt to assign Manager attribute
e.department = "Sales";
// the variable is declared as an Employee type,
// even though the Manager object has that attribute
```

Virtual Method Invocation

Assume that the following scenario is true:

```
Employee e = new Employee();
Manager m = new Manager();
```

If you ask for `e.getDetails()` and `m.getDetails()`, you invoke different behaviors. The `Employee` object executes the version of `getDetails()` associated with the `Employee` class, and the `Manager` object executes the version of `getDetails()` associated with the `Manager` class.

What is less obvious is what happens if you have:

```
Employee e = new Manager();
e.getDetails();
```

or something similar, such as a general method argument or an item from a heterogeneous collection.

In fact, you get the behavior associated with the object to which the variable refers at runtime. The behavior is not determined by the compile time type of the variable. This is an aspect of polymorphism, and is an important feature of object-oriented languages. This behavior is often referred to as *virtual method invocation*.

In the previous example, the `e.getDetails()` method executed is from the object's real type, a `Manager`. If you are a C++ programmer, there is an important distinction to be drawn between the Java programming language and C++. In C++, you get this behavior only if you mark the method as `virtual` in the source. In *pure* object-oriented languages, however, this is not normal. C++ does this to increase execution speed.

Heterogeneous Collections

You can create collections of objects that have a common class. Such collections are called *homogeneous* collections. For example:

```
MyDate[]  dates = new MyDate[2];
dates[0] = new MyDate(22, 12, 1964);
dates[1] = new MyDate(22, 7, 1964);
```

The Java programming language has an `Object` class, so you can make collections of all kinds of elements because all classes extend class `Object`. These collections are called *heterogeneous* collections.

A *heterogeneous* collection is a collection of dissimilar items. In object-oriented languages, you can create collections of many items. All have a common ancestor class: the `Object` class. For example:

```
Employee [] staff = new Employee[1024];
```

```
staff[0] = new Manager();
staff[1] = new Employee();
staff[2] = new Engineer();
```

You can even write a sort method that puts the employees into age or salary order, regardless of whether some of these employees are managers.

Note – Every class is a subclass of `Object`, so you can use an array of `Object` as a container for any combination of objects. The only items that cannot be added to an array of `Object` are primitive variables. However, you can create objects from primitive data using wrapper classes, as described in "Wrapper Classes" on page 6-26.

Polymorphic Arguments

You can write methods that accept a *generic* object (in this case, the class `Employee`) and work properly on objects of any subclass of this object. You might produce a method in an application class that takes an employee and compares it with a certain threshold salary to determine the tax liability of that employee. Using the polymorphic features you can do this as follows:

```
public class TaxService {
  public TaxRate findTaxRate(Employee e) {
    // do calculations and return a tax rate for e
  }
}

// Meanwhile, elsewhere in the application class
TaxService taxSvc = new TaxService();
Manager m = new Manager();
TaxRate t = taxSvc.findTaxRate(m);
```

This is legal because a `Manager` is an `Employee`. However, the `findTaxRate` method only has access to the members (both variables and methods) that are defined in the `Employee` class.

The instanceof Operator

Given that you can pass objects around using references to their parent classes, sometimes you might want to know what actual objects you have. This is the purpose of the instanceof operator. Suppose the class hierarchy is extended so that you have the following:

```
public class Employee extends Object
public class Manager extends Employee
public class Engineer extends Employee
```

Note – Remember that, while acceptable, extends Object is redundant. It is shown here only as a reminder.

If you receive an object using a reference of type Employee, it might turn out to be a Manager or an Engineer. You can test it by using instanceof as follows:

```
public void doSomething(Employee e) {
  if ( e instanceof Manager ) {
    // Process a Manager
  } else if ( e instanceof Engineer ) {
    // Process a Engineer
  } else {
    // Process any other type of Employee
  }
}
```

Note – In C++, you can do something similar using runtime-type information (RTTI).

Casting Objects

In circumstances where you have received a reference to a parent class, and you have determined that the object is, in fact, a particular subclass by using the instanceof operator, you can access the full functionality of the object by casting the reference.

```
public void doSomething(Employee e) {
  if ( e instanceof Manager ) {
    Manager m = (Manager) e;
    System.out.println("This is the manager of "
                       + m.getDepartment());
  }
  // rest of operation
}
```

If you do not make the cast, an attempt to execute e.getDepartment() would fail because the compiler cannot locate a method called getDepartment in the Employee class.

If you do not make the test using instanceof, you run the risk of the cast failing. Generally, any attempt to cast an object reference is subjected to several checks:

- Casts *upward* in the class hierarchy are always permitted and, in fact, do not require the cast operator. They can be done by simple assignment.
- For *downward* casts, the compiler must be satisfied that the cast is at least possible. For example, any attempt to cast a Manager reference to a Engineer reference is not permitted, because the Engineer is not a Manager. The class to which the cast is taking place must be some subclass of the current reference type.
- If the compiler permits the cast, then the object type is checked at runtime. For example, if it turns out that the instanceof check is omitted from the source, and the object being cast is not in fact an object of the type it is being cast to, then a runtime error (*exception)* occurs. Exceptions are a form of runtime error and are the subject of a later module.

Overloading Methods

In some circumstances, you might want to write several methods in the same class that do the same basic job with different arguments. Consider a simple method that is intended to output a textual representation of its argument. This method could be called `println()`.

Now suppose that you need a different print method for printing each of the `int`, `float`, and `String` types. This is reasonable, because the various data types require different formatting and, probably, varied handling. You could create three methods, called `printInt()`, `printFloat()`, and `printString()`, respectively. However, this is tedious.

The Java programming language, along with several other programming languages, permits you to reuse a method name for more than one method. This works only if there is something in the circumstances under which the call is made that distinguishes the method that is needed. In the case of the three print methods, this distinction is based on the number and type of the arguments.

By reusing the method name, you end up with the following methods:

```
public void println(int i)
public void println(float f)
public void println(String s)
```

When you write code to call one of these methods, the appropriate method is chosen according to the type of argument or arguments you supply.

Two rules apply to overloaded methods:

- Argument lists *must* differ.

  The argument lists of the calling statement must differ enough to allow unambiguous determination of the proper method to call. Normal widening promotions (for example, `float` to `double`) might be applied; this can cause confusion under some conditions.

- Return types *can* be different.

  The return type of the methods can be different, but it is not sufficient for the return type to be the only difference. The argument lists of overloaded methods must differ.

Methods Using Variable Arguments

A variation on overloading is when you need a method that takes any number of arguments of the same type. For example, imagine you need to create a method that calculates the average of a set of integers. You could define the following methods:

```
public class Statistics {
  public float average(int x1, int x2) {}
  public float average(int x1, int x2, int x3) {}
  public float average(int x1, int x2, int x3, int x4) {}
}
```

These methods can be invoked as follows:

```
float gradePointAverage = stats.average(4, 3, 4);
float averageAge = stats.average(24, 32, 27, 18);
```

These three overloaded methods all share the same functionality. It would be nice to collapse these methods into one method. J2SE version 5.0 now provides a feature, called *varargs* or *variable arguments*, to enable you to write a more generic method:

```
public class Statistics {
  public float average(int... nums) {
    int sum = 0;
    for ( int x : nums ) {
      sum += x;
    }
    return ((float) sum) / nums.length;
  }
}
```

This new varargs method can be invoked in the same manner as the suite of overloaded methods. Notice that the `nums` argument is an actual array object of type `int[]`; this permits the method to iterate over the elements and to get the number of elements (that is, the length of the array).

Overloading Constructors

When an object is instantiated, the program might be able to supply multiple constructors based on the data for the object being created. For example, a payroll system might be able to create an Employee object when it knows all of the basic data about the person: name, starting salary, and date of birth. Sometimes the system might not know the starting salary or the date of birth.

Code 6-4 shows four overloaded constructors for the `Employee` class. The first constructor (Lines 7–11) initializes all instance variables. In the second one (Lines 12–14), the date of birth is not provided. The `this` reference is being used as a forwarding call to another constructor (always within the same class); in this case, the first constructor. Likewise, the third constructor (Lines 15–17) calls the first constructor passing in the class constant `BASE_SALARY`. The fourth constructor (Lines 18–20) calls the second constructor passing the `BASE_SALARY`, which, in turn, calls the first constructor passing `null` for the date of birth.

Code 6-4 Example Overloading Constructors

```
1   public class Employee {
2     private static final double BASE_SALARY = 15000.00;
3     private String name;
4     private double salary;
5     private Date   birthDate;
6
7     public Employee(String name, double salary, Date DoB) {
8       this.name = name;
9       this.salary = salary;
10      this.birthDate = DoB;
11    }
12    public Employee(String name, double salary) {
13      this(name, salary, null);
14    }
15    public Employee(String name, Date DoB) {
16      this(name, BASE_SALARY, DoB);
17    }
18    public Employee(String name) {
19      this(name, BASE_SALARY);
20    }
21    // more Employee code...
22  }
```


The `this` keyword in a constructor must be the first line of code in the constructor. There can be more initialization code after the `this` call, but not before.

Constructors Are Not Inherited

Although a subclass inherits all of the methods and variables from a parent class, it does not inherit constructors.

There are only two ways in which a class can gain a constructor; either you write the constructor, or, because you have not written any constructors, the class has a single default constructor. A parent constructor is always called in addition to a child constructor. This is described in detail later in this module.

Invoking Parent Class Constructors

Like methods, constructors can call the non-private constructors of its immediate superclass.

Often you define a constructor that takes arguments and you want to use those arguments to control the construction of the parent part of an object. You can invoke a particular parent class constructor as part of a child class initialization by using the keyword `super` from the child constructor's *first* line. To control the invocation of the specific constructor, you must provide the appropriate arguments to `super()`. When there is no call to `super` with arguments, the parent constructor with zero arguments is called implicitly. In this case, if there is no parent constructor with zero arguments, a compiler error results.

The call to `super()` can take any number of arguments appropriate to the various constructors available in the parent class, but it must be the first statement in the constructor.

Assuming that the `Employee` class has the set of constructors that were defined in the Code 6-4 on page 6-17, then the following constructors in `Manager` might be defined. The constructor on Line 12 is illegal because the compiler inserts an implicit call to `super()`, and the `Employee` class has not provided a constructor without arguments.

```
1   public class Manager extends Employee {
2     private String department;
3
4     public Manager(String name, double salary, String dept) {
5       super(name, salary);
6       department = dept;
7     }
8     public Manager(String name, String dept) {
9       super(name);
10      department = dept;
11    }
12    public Manager(String dept) { // This code fails: no super()
13      department = dept;
14    }
15  }
```

When used, you must place `super` or `this` in the first line of the constructor. If you write a constructor that has neither a call to `super(...)` nor `this(...)`, the compiler inserts a call to the parent class constructor automatically, with no arguments. Other constructors can also call `super(...)` or `this(...)`, invoking a chain of constructors. What happens ultimately is that the parent class constructor (or possibly several) executes before any child class constructor in the chain.

Constructing and Initializing Objects: A Slight Reprise

Object initialization is a rather complex process. In the section "Constructing and Initializing Objects" in Module 3, "Identifiers, Keywords, and Types," you were exposed to a rudimentary explanation. In this section, you see the whole process.

First, the memory for the complete object is allocated and the default values for the instance variables are assigned. Second, the top-level constructor is called and follows these steps recursively down the inheritance tree:

1. Bind constructor parameters.
2. If explicit `this()`, call recursively, and then skip to Step 5.
3. Call recursively the implicit or explicit `super(...)`, except for `Object` because `Object` has no parent class.
4. Execute the explicit instance variable initializers.
5. Execute the body of the current constructor.

Constructor and Initialization Examples

For an example, use the following code for the `Manager` and `Employee` classes:

```
1   public class Object {
2     public Object() {}
3   }

1   public class Employee extends Object {
2     private String name;
3     private double salary = 15000.00;
4     private Date   birthDate;
5
6     public Employee(String n, Date DoB) {
7       // implicit super();
8       name = n;
9       birthDate = DoB;
10    }
11    public Employee(String n) {
12      this(n, null);
13    }
14  }
```

```
1   public class Manager extends Employee {
2     private String department;
3
4     public Manager(String n, String d) {
5       super(n);
6       department = d;
7     }
8   }
```

The following are the steps to construct `new Manager("Joe Smith", "Sales")`:

```
0 Basic initialization
  0.1 Allocate memory for the complete Manager object
  0.2 Initialize all instance variables to their default values
1 Call constructor: Manager("Joe Smith", "Sales")
  1.1 Bind constructor parameters: n="Joe Smith", d="Sales"
  1.2 No explicit this() call
  1.3 Call super(n) for Employee(String)
    1.3.1 Bind constructor parameters: n="Joe Smith"
    1.3.2 Call this(n, null) for Employee(String, Date)
      1.3.2.1 Bind constructor parameters: n="Joe Smith", DoB=null
      1.3.2.2 No explicit this() call
      1.3.2.3 Call super() for Object()
        1.3.2.3.1 No binding necessary
        1.3.2.3.2 No this() call
        1.3.2.3.3 No super() call (Object is the root)
        1.3.2.3.4 No explicit variable initialization for Object
        1.3.2.3.5 No method body to call
      1.3.2.4 Initialize explicit Employee variables: salary=15000.00;
      1.3.2.5 Execute body: name="Joe Smith"; date=null;
    1.3.3 - 1.3.4 Steps skipped
    1.3.5 Execute body: No body in Employee(String)
  1.4 No explicit initializers for Manager
  1.5 Execute body: department="Sales"
```

The Object Class

The Object class is the root of all classes in the Java technology programming language. If a class is declared with no extends clause, then the compiler adds implicitly the code extends Object to the declaration; for example:

```
public class Employee {
  // more code here
}
```

is equivalent to:

```
public class Employee extends Object {
  // more code here
}
```

This enables you to override several methods inherited from the Object class. The following sections describe two important Object methods.

The equals Method

The == operator performs an equivalent comparison. That is, for any reference values x and y, x==y returns true if and only if x and y refer to the same object.

The Object class in the java.lang package has the method public boolean equals(Object obj), which compares two objects for equality. When not overridden, an object's equals() method returns true only if the two references being compared refer to the same object. However, the intention of the equals() method is to compare the contents of two objects whenever possible. This is why the method is frequently overridden. For example, the equals() method in String class returns true if and only if the argument is not null and is a String object that represents the same sequence of characters as the String object with which the method is invoked.

Note – You should override the hashCode method whenever you override the equals method. A simple implementation could use a bitwise XOR on the hash codes of the elements tested for equality.

An equals Example

In this example, the MyDate class has been modified to include an equals method that tests against the year, month, and day attributes.

```
1   public class MyDate {
2     private int day;
3     private int month;
4     private int year;
5
6     public MyDate(int day, int month, int year) {
7       this.day   = day;
8       this.month = month;
9       this.year  = year;
10    }
11
12    public boolean equals(Object o) {
13      boolean result = false;
14      if ( (o != null) && (o instanceof MyDate) ) {
15        MyDate d = (MyDate) o;
16        if ( (day == d.day) && (month == d.month)
17             && (year == d.year) ) {
18          result = true;
19        }
20      }
21      return result;
22    }
23
24    public int hashCode() {
25      return (day ^ month ^ year);
26    }
27  }
```

The hashCode method implements a bitwise XOR of the date attributes. This guarantees that hash code for equal MyDate objects have the same value while making it likely that different dates will return different values.

The following program tests two MyDate objects that are not identical, but are equal relative to the year-month-day test.

```
1   class TestEquals {
2     public static void main(String[] args) {
3       MyDate  date1 = new MyDate(14, 3, 1976);
4       MyDate  date2 = new MyDate(14, 3, 1976);
5
```

```
6       if ( date1 == date2 ) {
7         System.out.println("date1 is identical to date2");
8       } else {
9         System.out.println("date1 is not identical to date2");
10      }
11
12      if ( date1.equals(date2) ) {
13        System.out.println("date1 is equal to date2");
14      } else {
15        System.out.println("date1 is not equal to date2");
16      }
17
18      System.out.println("set date2 = date1;");
19      date2 = date1;
20
21      if ( date1 == date2 ) {
22        System.out.println("date1 is identical to date2");
23      } else {
24        System.out.println("date1 is not identical to date2");
25      }
26    }
27  }
```

The execution of this test program generates the following output:

```
date1 is not identical to date2
date1 is equal to date2
set date2 = date1;
date1 is identical to date2
```

The toString Method

The toString method converts an object to a String representation. It is referenced by the compiler when automatic string conversion takes place. For example, the System.out.println() call:

```
Date now = new Date();
System.out.println(now);
```

is equivalent to:

```
System.out.println(now.toString());
```

The Object class defines a default toString method that returns the class name and its reference address (not normally useful). Many classes override toString to provide more useful information. For example, all wrapper classes (introduced later in this module) override toString to provide a string form of the value they represent. Even classes representing items without a string form often implement toString to return object state information for debugging purposes.

Wrapper Classes

The Java programming language does not look at primitive data types as objects. For example, numerical, Boolean, and character data are treated in the primitive form for the sake of efficiency. The Java programming language provides *wrapper* classes to manipulate primitive data elements as objects. Such data elements are *wrapped* in an object created around them. Each Java primitive data type has a corresponding wrapper class in the `java.lang` package. Each wrapper class object encapsulates a single primitive value. (See Table 6-2.)

Note – These wrapper classes implement *immutable* objects. That means that after the primitive value is initialized in the wrapper object, then there is no way to change that value.

Table 6-2 Wrapper Classes

Primitive Data Type	Wrapper Class
`boolean`	`Boolean`
`byte`	`Byte`
`char`	`Character`
`short`	`Short`
`int`	`Integer`
`long`	`Long`
`float`	`Float`
`double`	`Double`

You can construct a wrapper class object by passing the value to be wrapped into the appropriate constructor, as shown in Code 6-5.

Code 6-5 Examples of Primitive Boxing Using Wrapper Classes

```
int pInt = 420;
Integer wInt = new Integer(pInt); // this is called boxing
int p2 = wInt.intValue(); // this is called unboxing
```

Wrapper classes are useful when converting primitive data types because of the many wrapper class methods available; for example:

```
int x = Integer.valueOf(str).intValue();
```

or:

```
int x = Integer.parseInt(str);
```

Autoboxing of Primitive Types

If you have to change the primitive data types to their object equivalents (called *boxing*), then you need to use the wrapper classes. Also to get the primitive data type from the object reference (called *unboxing*), you need to use the wrapper class methods. All of this boxing and unboxing can clutter up your code and thus make your code difficult to understand. In J2SE version 5.0, the autoboxing feature enables you to assign and retrieve primitive types without the need of the wrapper classes.

Code 6-6 shows two simple cases of autoboxing and autounboxing. Compare this with Code 6-5 on page 6-26.

Code 6-6 Examples of Primitive Autoboxing

```
int pInt = 420;
Integer wInt = pInt; // this is called autoboxing
int p2 = wInt; // this is called autounboxing
```

The J2SE version 5.0 compiler will now create the wrapper object automatically when assigning a primitive to a variable of the wrapper class type. The compiler will also extract the primitive value when assigning from a wrapper object to a primitive variable.

This can be done when passing parameters to methods or even within arithmetic expressions.

Caution – Do not overuse the autoboxing feature. There is a hidden performance impact when a value is autoboxed or autounboxed. Mixing primitives and wrapper objects in arithmetic expressions within a tight loop might have a negative impact on the performance and throughput of your applications.