

Inheritance and Polymorphism

Overriding Methods

Overloading Methods

Overriding vs. Overloading

The Object Class and its toString() Method

Overriding Methods

A subclass inherits methods from a superclass. Sometimes it is necessary for the subclass to modify the implementation of a method defined in the superclass. This is referred to as method overriding.

```
public class Circle extends GeometricObject {  
    // Other methods are omitted  
  
    // This method Overrides the toString method defined in GeometricObject  
    // This means that this method modifies the code of the toString in GeometricObject  
    public String toString() {  
        return super.toString() + "\nradius is " + radius;  
    }  
}  
  
public class GeometricObject {  
    // Other methods are omitted  
    public String toString() {  
        String s = "The color is " + color;  
        if (filled)  
            s += ".\nThe geometric object is filled";  
        else  
            s += ".\nThe geometric object is not filled";  
        return s;  
    }  
}
```

Overriding Methods

To avoid mistakes, you can use a special Java syntax, called *override annotation*, to place **@Override** before the method in the subclass as in the below example.

```
public class Circle extends GeometricObject {  
    // Other methods are omitted  
  
    @Override  
    public String toString() { // overrides the toString method in GeometricObject  
        return super.toString() + "\nradius is " + radius;  
    }  
}
```

This annotation denotes that the annotated method is required to override a method in the superclass. If a method with this annotation does not override its superclass's method, the compiler will report an error. For example, if **toString** is mistyped as **tostring**, a compile error is reported. If the override annotation isn't used, the compile won't report an error. Using annotation avoids mistakes.

Overriding Methods

- An instance (non static) method can be overridden only if it is accessible. Thus a private method cannot be overridden, because it is not accessible outside its own class.
- Like an instance method, a static method can be inherited. However, a static method cannot be overridden.

Overloading Methods

Overloading methods enables you to define the methods with the same name as long as their signatures (parameter lists) are different.

The **max** method below works only with the **double** data type.

```
public static double max(double num1, double num2) {  
    if (num1 > num2)  
        return num1;  
    else  
        return num2;  
}
```

But what if you need to determine which of two integer numbers has the maximum value? The solution is to create another method with the same name but different parameters, as shown in the next slide.

Overloading Methods

```
public class TestMethodOverloading {  
    public static void main(String[] args) {  
        // Invoke the max method with the double parameters  
        System.out.println("The maximum of 3 and 4 is " + max(3, 4));  
        // Invoke the max method with the double parameters  
        System.out.println("The maximum of 3.0 and 5.4 is " + max(3.0, 5.4));  
    }  
    // Find the max of two int values  
    public static int max(int num1, int num2) {  
        if (num1 > num2)  
            return num1;  
        return num2;  
    }  
    // Find the max of two double values  
    public static double max(double num1, double num2) {  
        if (num1 > num2)  
            return num1;  
        return num2;  
    }  
}
```

Output

The maximum of 3 and 4 is 4
The maximum of 3.0 and 5.4 is 5.4

Overloading Methods

If you call **max** with **int** parameters, the **max** method that expects **int** parameters will be invoked; if you call **max** with **double** parameters, the **max** method that expects **double** parameters will be invoked.

When calling `max(3, 4)`, the `max` method for finding the maximum of two integers is invoked. When calling `max(3.0, 5.4)`, the `max` method for finding the maximum of two doubles is invoked.

This is referred to as *method overloading*; that is, two methods have the same name but different signatures (parameter lists) within one class. The Java compiler determines which method to use based on the method signature.

Overloading Methods

Overloading is allowed if the methods signature is different meaning if one of the following three cases is true:

1. If the type of the parameters is different:

```
public static int max(int num1, int num2) {  
    // body  
}  
  
public static double max(double num1, double num2) {  
    // body  
}
```

2. If the number of the parameters is different:

```
public static int sum(int num1, int num2) {  
    // body  
}  
  
public static int sum(int num1, int num2, int num3) {  
    // body  
}
```

3. If the order of the parameters is different:

```
public static void method(int num, char ch) {  
    // body  
}  
  
public static void method(char ch, int num) {  
    // body  
}
```

Overloading Methods

Overloading methods can make programs clearer and more readable. Methods that perform the same function with different types of parameters should be given the same name.

Overloaded methods must have different parameter lists. You cannot overload methods based on different modifiers or return types. For example, the below is not overloading and will cause a syntax error since although the return types are the different, the methods signatures (parameter lists) are the same.

```
public static int max(int num1, int num2) {  
    // body  
}  
  
public static double max(int num1, int num2) {  
    // body  
}
```

Overriding vs. Overloading

Let us use an example to show the differences between overriding and overloading. In (a) below, the method **p(double i)** in class **A** overrides the same method defined in class **B**. In (b), however, the class **A** has two overloaded methods: **p(double i)** and **p(int i)**. The method **p(double i)** is inherited from **B**.

```
public class Test {  
    public static void main(String[] args) {  
        A a = new A();  
        a.p(10);  
        a.p(10.0);  
    }  
}  
  
class B {  
    public void p(double i) {  
        System.out.println(i * 2);  
    }  
}  
  
class A extends B {  
    // This method overrides the method in B  
    public void p(double i) {  
        System.out.println(i);  
    }  
}  
// (a)
```

```
public class Test {  
    public static void main(String[] args) {  
        A a = new A();  
        a.p(10);  
        a.p(10.0);  
    }  
}  
  
class B {  
    public void p(double i) {  
        System.out.println(i * 2);  
    }  
}  
  
class A extends B {  
    // This method overloads the method in B  
    public void p(int i) {  
        System.out.println(i);  
    }  
}  
// (b)
```

Overriding vs. Overloading

In the previous example, when you run the **Test** class in (a), both **a.p(10)** and **a.p(10.0)** invoke the **p(double i)** method defined in class **A** to display **10.0**. When you run the **Test** class in (b), **a.p(10)** invokes the **p(int i)** method defined in class **A** to display **10**, and **a.p(10.0)** invokes the **p(double i)** method defined in class **B** to display **20.0**.

Overloading means to define multiple methods with the same name but different signatures. Overriding means to provide a new implementation for a method in the subclass.

Note the following:

- Overridden methods are in different classes related by inheritance; overloaded methods can be either in the same class or different classes related by inheritance.
- Overridden methods have the same name, signature and return type; overloaded methods have the same name but a different signature.

The Object Class and its toString() Method

Every class in Java is a subclass of the class **Object** found in the java library in the lang package.

If no inheritance is specified when a class is defined, the superclass of the class is **Object** by default. For example, the following two class definitions are the same:

```
public class ClassName {  
    ...  
}
```

Equivalent

```
public class ClassName extends Object {  
    ...  
}
```

The Object Class and its toString() Method

It is important to be familiar with the methods provided by the **Object** class so that you can use them in your classes.

This section introduces the **toString** method in the **Object** class. The signature of the **toString()** method is:

```
public String toString()
```

The Object Class and its toString() Method

As you know from chapter 8, the `toString()` method is used to return a string representation of the object. If no `toString()` is defined in a class then a call to `toString()` will actually call the `toString()` of its superclass `Object`. The `toString()` of the class `Object` returns a string consisting of a class name of which the object is an instance, the at sign (`@`), and a number representing this object. Assume that the class `Loan` does not contain a `toString()` method, and assume we wrote the following two statements:

```
Loan loan = new Loan();  
System.out.println(loan.toString());
```

Then the output will look something like `Loan@15037e5`. This message is not very helpful or informative. Usually you should override the `toString` method so that it returns a useful string representation of the object.

Class Circle without a toString() method:

```
public class Circle { // This class inherits the toString of Object
    private double radius;
    public Circle(double r){
        radius = r;
    }
    // other methods
}
```

```
public class Application {
    public static void main(String [] args) {
        Circle c1 = new Circle(1.0);
        System.out.println(c1.toString());
    }
}
```

Output
Circle@13237a3

Class Circle with a toString() method:

```
public class Circle {  
    private double radius;  
  
    public Circle(double r) {  
        radius = r;  
    }  
  
    @Override  
    public String toString() { // Overrides the toString of Object  
        return "The radius is " + radius;  
    }  
}  
  
public class Application {  
    public static void main(String [] args) {  
        Circle c1 = new Circle(1.0);  
        System.out.println(c1.toString());  
    }  
}
```

Output
The radius is 1.0