



COMPUTER SCIENCE DEPARTMENT FACULTY OF ENGINEERING AND TECHNOLOGY

ADVANCED PROGRAMMING COMP231

Lecturer :Farid Mohammad

Inheritance 2



Overriding Methods

To *override* a method,

the method must be *defined in the subclass*

using the *same signature and the same return type* as in its *superclass*.

```
class Cat{  
    public void Sound(){  
        System.out.println("meow");  
    }  
}  
class Lion extends Cat{  
    public void sniff(){  
        System.out.println("sniff");  
    }  
    public void Sound(){  
        System.out.println("roar");  
    }  
}
```

Overriding

Same method name and same parameters

// Method Overriding Example

//Base class

```
class Parent {  
    public void display() {  
        System.out.println("parent method is executed");  
    }  
}
```

//Derived or Inherited class

```
class Child extends Parent {  
  
    //Below method overrides the Parent display() method  
    @Override  
    public void display() {  
        System.out.println("child method is executed");  
    }  
}
```

//Driver class

```
public class Main {  
    public static void main(String args[])  
    {  
        // If a Parent type reference refers  
        // to a Parent object, then Parent's  
        // display() is called  
        Parent parentObject = new Parent();  
        parentObject.display();  
  
        // If a Parent type reference refers  
        // to a Child object Child's display()  
        // is called. This is called RUN TIME  
        // POLYMORPHISM.  
        Parent childObject = new Child();  
        childObject.display();  
    }  
}
```

■ An **instance 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.

If a method defined in a subclass is private in its superclass, the two methods are completely unrelated.

■ Like an instance method, a **static method can be inherited**. However, a static method **cannot be overridden**. If a static method defined in the superclass is redefined in a subclass, the method defined in the superclass is hidden. The hidden static methods can be invoked using the syntax **SuperClassName.staticMethodName**.

11.7 True or false? You can override a private method defined in a superclass.

11.8 True or false? You can override a static method defined in a superclass.

11.9 How do you explicitly invoke a superclass's constructor from a subclass?

11.10 How do you invoke an overridden superclass method from a subclass?

11.5 Overriding vs. Overloading

*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.*

```
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)

To avoid mistakes, you can use a special Java syntax, called *override annotation*, to place

@Override before the method in the subclass. For example:
`public class CircleFromSimpleGeometricObject extends SimpleGeometricObject {`
`// Other methods are omitted`

```
@Override
public String toString() {
    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.

11.11 Identify the problems in the following code:

```
public class Circle {
    private double radius;

    public Circle(double radius) {
        radius = radius;
    }

    public double getRadius() {
        return radius;
    }

    public double getArea() {
        return radius * radius * Math.PI;
    }
}

class B extends Circle {
    private double length;

    B(double radius, double length) {
        Circle(radius);
        length = length;
    }

    @Override
    public double getArea() {
        return getArea() * length;
    }
}
```

11.12 Explain the difference between method overloading and method overriding.

11.13 If a method in a subclass has the **same signature** as a method in its superclass with the same **return type**, is the method **overridden** or **overloaded**?

11.14 If a method in a subclass has the **same signature** as a method in its superclass with a **different return type**, will this be a problem?

11.15 If a method in a subclass has the **same name** as a method in its superclass with **different parameter types**, is the method overridden or overloaded?

11.16 What is the benefit of using the **@Override** annotation?

The Object Class and Its toString() Method

Every class in Java is descended from the `java.lang.Object` class.

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:

<pre>public class ClassName { ... }</pre>	Equivalent	<pre>public class ClassName extends Object { ... }</pre>
---	------------	--

There are some methods provided by the `Object`

This section introduces the `toString` method in the `Object` class.

The signature of the `toString()` method is:
public String toString()

Invoking `toString()` on an object **returns a string that describes the object.**

By default, it returns a string consisting of a **class name** of which the object is an instance, an at sign (**@**), and the **object's memory address in hexadecimal.**

For example, consider the following code

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

The output for this code displays something like **Loan@15037e5.**

This message is not very helpful or informative. **Usually you should override** the `toString` method so that it returns a descriptive string representation of the object.

For example, the `toString` method in the `Object` class was overridden in the `GeometricObject` class in lines 46–49 in Listing 11.1

as follows:

```
public String toString() {  
return "created on " + dateCreated + "\ncolor: "  
+ color +  
" and filled: " + filled;  
}
```

Polymorphism

Polymorphism means that a variable of a supertype can refer to a subtype object.

The three pillars of object-oriented programming are **encapsulation**, **inheritance**, and **polymorphism**.

You have already learned the first two. This section introduces polymorphism.

You can say that **Circle** is a subtype of **GeometricObject** and **GeometricObject** is a supertype for **Circle**.

The inheritance relationship **enables** a subclass to inherit features from its superclass with additional new features.

A subclass is a specialization of its superclass; every instance of a subclass is also an instance of its superclass, but not vice versa.

For example, every circle is a geometric object, but not every geometric object is a circle.

Therefore, you can always **pass** an instance of a subclass to a parameter of its **superclass type**. Consider the code in

LISTING 11.5 PolymorphismDemo.java

```
1 public class PolymorphismDemo {
2     /** Main method */
3     public static void main(String[] args) {
4         // Display circle and rectangle properties
5         displayObject(new CircleFromSimpleGeometricObject
6             (1, "red", false));           polymorphic call
7         displayObject(new RectangleFromSimpleGeometricObject
8             (1, 1, "black", true));     polymorphic call
9     }
10
11     /** Display geometric object properties */
12     public static void displayObject(SimpleGeometricObject object) {
13         System.out.println("Created on " + object.getDateCreated() +
14             ". Color is " + object.getColor());
15     }
16 }
```

The definition of "Polymorphism" is hidden in its name itself. "**Poly**" means "**many**" and "**morphs**" means "**forms**".

Thus "Polymorphism" may be considered as a concept in Java in which an entity(object), variable or function may present in one or many forms.

The method `displayObject` (line 12) takes a parameter of the `GeometricObject` type.

You can invoke `displayObject` by passing any instance of `GeometricObject`

(e.g., `new CircleFromSimpleGeometricObject(1, "red", false)`

and `new RectangleFromSimpleGeometricObject(1, 1, "black", false)` in lines 5–8).

An object of a **subclass** can be used **wherever** its **superclass** object is used. This is commonly known as *polymorphism*

11.8 Dynamic Binding

A method can be implemented in several classes along the inheritance chain.

The JVM decides which method is invoked at runtime.

A method can be defined in a superclass and **overridden** in its subclass.

For example, the `toString()` method is defined in the `Object` class and overridden in `GeometricObject`.

Consider the following code:
`Object o = new GeometricObject();`
`System.out.println(o.toString());`

To answer this question, we first introduce two terms: **declared type** and **actual type**.

A variable must be declared a type.

Here `o`'s **declared type** is `Object`.

Here `o`'s **actual type** is `GeometricObject`,

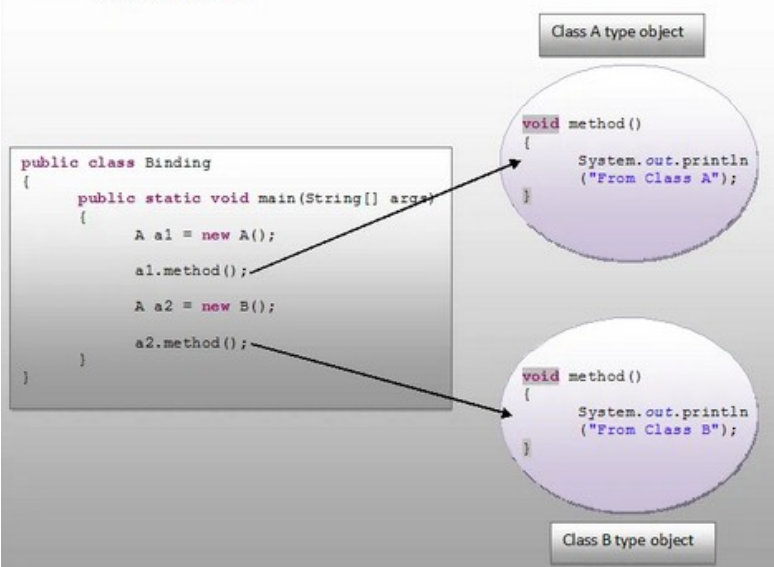
because `o` references an object created using `new GeometricObject()`.

Which `toString()` method is invoked by `o` is determined by `o`'s actual type.

This is known as dynamic binding.

Which `toString()` method is invoked by `o`?

Dynamic Binding



11.19 Can you assign `new int[50]`, `new Integer[50]`, `new String[50]`, or `new Object[50]`, into a variable of `Object[]` type?

11.20 What is wrong in the following code?

```
public class Test {
    public static void main(String[] args) {
        Integer[] list1 = {12, 24, 55, 1};
        Double[] list2 = {12.4, 24.0, 55.2, 1.0};
        int[] list3 = {1, 2, 3};
        printArray(list1);
        printArray(list2);
        printArray(list3);
    }

    public static void printArray(Object[] list) {
        for (Object o: list)
            System.out.print(o + " ");
        System.out.println();
    }
}
```

11.21 Show the output of the following code:

```
public class Test {
    public static void main(String[] args) {
        new Person().printPerson();
        new Student().printPerson();
    }
}

class Student extends Person {
    @Override
    public String getInfo() {
        return "Student";
    }
}

class Person {
    public String getInfo() {
        return "Person";
    }

    public void printPerson() {
        System.out.println(getInfo());
    }
}
```

(a)

```
public class Test {
    public static void main(String[] args) {
        new Person().printPerson();
        new Student().printPerson();
    }
}

class Student extends Person {
    private String getInfo() {
        return "Student";
    }
}

class Person {
    private String getInfo() {
        return "Person";
    }

    public void printPerson() {
        System.out.println(getInfo());
    }
}
```

(b)