INHERITANCE AND POLYMORPHISM

Why Inheritance?

- Suppose you will define classes to model circles, rectangles, and triangles. These classes have many common features. What is the best way to design these classes so to avoid redundancy?

- The answer is to use inheritance.

Inheritance

- In the Java language, classes can be *derived* from other classes, thereby *inheriting* fields and methods from those classes.

- A class that is derived from another class is called a *subclass* (also a *derived class*, *extended class*, or *child class*).

- The class from which the subclass is derived is called a *superclass* (also a *base class* or a *parent class*).

Inheritance

- Excepting *Object*, which has no superclass, every class has one and only one direct superclass (single inheritance).
- In the absence of any other explicit superclass, every class is implicitly a subclass of *Object*.
- A subclass inherits all the *members* (fields, methods, and nested classes) from its superclass.
- Constructors are not members, so they are not inherited by subclasses, but the constructor of the superclass can be invoked from the subclass.

GeometricObject Class

```
public class GeometricObject {
  private String color;
  private boolean filled;

  protected GeometricObject() {
    this.color = "white";
  }
  protected GeometricObject(String color, boolean filled) {
    this.color = color;
    this.filled = filled;
  }

  public String getColor() {
    return color;
  }
  public void setColor(String color) {
    this.color = color;
  }
  public boolean isFilled() {
    return filled;
  }
  public void setFilled(boolean filled) {
    this.filled = filled;
  }
  public void printInfo(){
    System.out.println("The Geometric is "+color);
  }
}
```

Circle Class

```
public class Circle extends GeometricObject {
  private double radius;

  public Circle() {
    this(1.0);
  }
  public Circle(double radius) {
    this(radius, "white", false);
  }
  public Circle(double radius, String color, boolean filled) {
    super(color, filled);
    this.radius = radius;
  }

  public double getRadius() {
    return radius;
  }
  public void setRadius(double radius) {
    this.radius = radius;
  }
  public double getArea() {
    return radius*radius*Math.PI;
  }
  public double getPerimeter() {
    return 2*radius*Math.PI;
  }
}
```

TestInheritence Class

```
public class TestInheritance {
  public static void main(String[] args){
   System.out.println("---Create a Circle c1---");
   Circle c1 = new Circle();
   if(c1 instanceof GeometricObject){
     System.out.println("c1 is a GeometricObject");}
   if(c1 instanceof Circle){
     System.out.println("c1 is a Circle");}
   if(c1 instanceof Object){
     System.out.println("c1 is an Object");}

   System.out.println("---Create a GeometricObject g1---");
   if(g1 instanceof Circle){
     System.out.println("g1 is a Circle");}
   if(g1 instanceof GeometricObject){
     System.out.println("g2 is a GeometricObject");}

   System.out.println("---Create a GeometricObject g2---");
   GeometricObject g2 = (GeometricObject) new Circle();
   if(g2 instanceof Circle){
     System.out.println("g2 is a Circle");}
   if(g2 instanceof GeometricObject){
     System.out.println("g2 is a GeometricObject");}
  }
}
```

Are superclass's Constructor Inherited?

- No. They are not inherited.
- They are invoked explicitly or implicitly.
- Explicitly using the super keyword.
- A constructor is used to construct an instance of a class. Unlike properties and methods, a superclass's constructors are not inherited in the subclass. They can only be invoked from the subclasses' constructors, using the keyword super. *If the keyword super is not explicitly used, the superclass's no-arg constructor is automatically invoked.*

Superclass's Constructor Is Always Invoked

- A constructor may invoke an overloaded constructor or its superclass's constructor. If none of them is invoked explicitly, the compiler puts super() as the first statement in the constructor. For example,

```
public A() {
}
```

is equivalent to

```
public A() {
  super();
}
```

```
public A(double d) {
  // some statements
}
```

is equivalent to

```
public A(double d) {
  super();
  // some statements
}
```

Using the Keyword `super`

- The keyword `super` refers to the superclass of the class in which `super` appears. This keyword can be used in two ways:
  - To call a superclass constructor
  - To call a superclass method
- CAUTION ! You must use the keyword super to call the superclass constructor. Invoking a superclass constructor's name in a subclass causes a syntax error. Java requires that the statement that uses the keyword super appear first in the constructor.

Constructor Chaining

Constructing an instance of a class invokes all the superclasses' constructors along the inheritance chain. This is called *constructor chaining*.

```
public class Faculty extends Employee {
  public static void main(String[] args) {
    new Faculty();
  }

  public Faculty() {
    System.out.println("(4) Faculty's no-arg constructor is invoked");
  }
}

class Employee extends Person {
  public Employee() {
    this("(2) Invoke Employee's overloaded constructor");
    System.out.println("(3) Employee's no-arg constructor is invoked");
  }

  public Employee(String s) {
    System.out.println(s);
  }
}

class Person {
  public Person() {
    System.out.println("(1) Person's no-arg constructor is invoked");
  }
}
```

Trace Execution

```
public class Faculty extends Employee {
  public static void main(String[] args) {
    new Faculty();
  }

  public Faculty() {
    System.out.println("(4) Faculty's no-arg constructor is invoked");
  }
}

class Employee extends Person {
  public Employee() {
    this("(2) Invoke Employee's overloaded constructor");
    System.out.println("(3) Employee's no-arg constructor is invoked");
  }

  public Employee(String s) {
    System.out.println(s);
  }
}

class Person {
  public Person() {
    System.out.println("(1) Person's no-arg constructor is invoked");
  }
}
```

1. Start from the main method

Trace Execution

```
public class Faculty extends Employee {
  public static void main(String[] args) {
    new Faculty();
  }

  public Faculty() {
    System.out.println("(4) Faculty's no-arg constructor is invoked");
  }
}

class Employee extends Person {
  public Employee() {
    this("(2) Invoke Employee's overloaded constructor");
    System.out.println("(3) Employee's no-arg constructor is invoked");
  }

  public Employee(String s) {
    System.out.println(s);
  }
}

class Person {
  public Person() {
    System.out.println("(1) Person's no-arg constructor is invoked");
  }
}
```

2. Invoke Faculty constructor

Trace Execution

```
public class Faculty extends Employee {
  public static void main(String[] args) {
    new Faculty();
  }

  public Faculty() {
    System.out.println("(4) Faculty's no-arg constructor is invoked");
  }
}

class Employee extends Person {
  public Employee() {
    this("(2) Invoke Employee's overloaded constructor");
    System.out.println("(3) Employee's no-arg constructor is invoked");
  }

  public Employee(String s) {
    System.out.println(s);
  }
}

class Person {
  public Person() {
    System.out.println("(1) Person's no-arg constructor is invoked");
  }
}
```

3. Invoke Employee's no-arg constructor

Trace Execution

```
public class Faculty extends Employee {
  public static void main(String[] args) {
    new Faculty();
  }

  public Faculty() {
    System.out.println("(4) Faculty's no-arg constructor is invoked");
  }
}

class Employee extends Person {
  public Employee() {
    this("(2) Invoke Employee's overloaded constructor");
    System.out.println("(3) Employee's no-arg constructor is invoked");
  }

  public Employee(String s) {
    System.out.println(s);
  }
}

class Person {
  public Person() {
    System.out.println("(1) Person's no-arg constructor is invoked");
  }
}
```

4. Invoke Employee(String) constructor

Trace Execution

```
public class Faculty extends Employee {
  public static void main(String[] args) {
    new Faculty();
  }

  public Faculty() {
    System.out.println("(4) Faculty's no-arg constructor is invoked");
  }
}

class Employee extends Person {
  public Employee() {
    this("(2) Invoke Employee's overloaded constructor");
    System.out.println("(3) Employee's no-arg constructor is invoked");
  }

  public Employee(String s) {
    System.out.println(s);
  }
}

class Person {
  public Person() {
    System.out.println("(1) Person's no-arg constructor is invoked");
  }
}
```

5. Invoke Person() constructor

Trace Execution

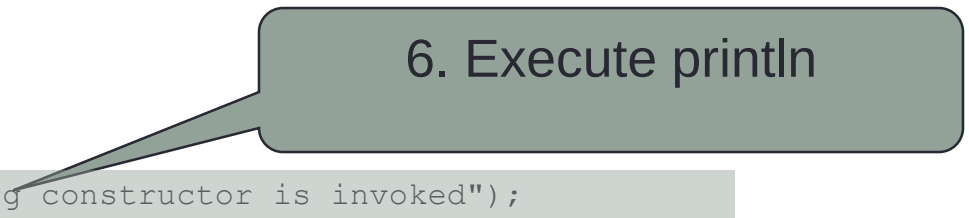
```
public class Faculty extends Employee {
  public static void main(String[] args) {
    new Faculty();
  }

  public Faculty() {
    System.out.println("(4) Faculty's no-arg constructor is invoked");
  }
}

class Employee extends Person {
  public Employee() {
    this("(2) Invoke Employee's overloaded constructor");
    System.out.println("(3) Employee's no-arg constructor is invoked");
  }

  public Employee(String s) {
    System.out.println(s);
  }
}

class Person {
  public Person() {
    System.out.println("(1) Person's no-arg constructor is invoked");
  }
}
```



Trace Execution

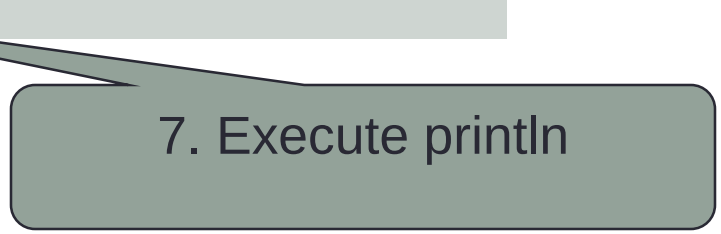
```
public class Faculty extends Employee {
  public static void main(String[] args) {
    new Faculty();
  }

  public Faculty() {
    System.out.println("(4) Faculty's no-arg constructor is invoked");
  }
}

class Employee extends Person {
  public Employee() {
    this("(2) Invoke Employee's overloaded constructor");
    System.out.println("(3) Employee's no-arg constructor is invoked");
  }

  public Employee(String s) {
    System.out.println(s);
  }
}

class Person {
  public Person() {
    System.out.println("(1) Person's no-arg constructor is invoked");
  }
}
```



Trace Execution

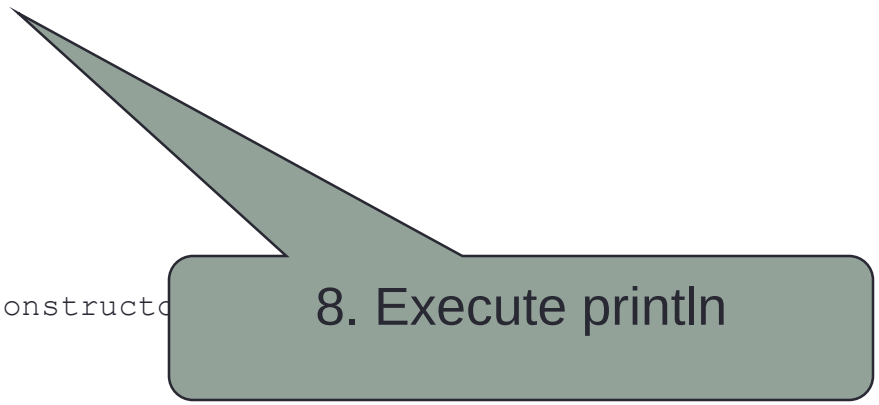
```
public class Faculty extends Employee {
  public static void main(String[] args) {
    new Faculty();
  }

  public Faculty() {
    System.out.println("(4) Faculty's no-arg constructor is invoked");
  }
}

class Employee extends Person {
  public Employee() {
    this("(2) Invoke Employee's overloaded constructor");
    System.out.println("(3) Employee's no-arg constructor is invoked");
  }

  public Employee(String s) {
    System.out.println(s);
  }
}

class Person {
  public Person() {
    System.out.println("(1) Person's no-arg constructo
  }
}
```



Trace Execution

```
public class Faculty extends Employee {
  public static void main(String[] args) {
    new Faculty();
  }

  public Faculty() {
    System.out.println("(4) Faculty's no-arg constructor is invoked");
  }
}

class Employee extends Person {
  public Employee() {
    this("(2) Invoke Employee's overloaded constructor"
    System.out.println("(3) Employee's no-arg constructor is invoked");
  }

  public Employee(String s) {
    System.out.println(s);
  }
}

class Person {
  public Person() {
    System.out.println("(1) Person's no-arg constructor is invoked");
  }
}
```

9. Execute println

Declaring a Subclass and Call Superclass Methods

- A subclass extends properties and methods from the superclass. You can also:
  - Add new properties
  - Add new methods
  - Override the methods of the superclass

- You can call superclass' methods by using a `super` keyword

Overriding Methods in the Superclass

- 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


  /** Override the printInfo method defined in GeometricObject */
  public void printInfo() {
    super.printInfo();
    System.out.println( "It is a circle with radius of " + radius);
  }
}
```

Overriding Methods in the Superclass

- 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.

Overriding vs. Overloading

```
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);
  }
}
```

```
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);
  }
}
```

Polymorphism

```
public class PolymorphismDemo {
  public static void main(String[] args) {
    m(new GraduateStudent());
    m(new Student());
    m(new Person());
    m(new Object());
  }

  public static void m(Object x) {
    System.out.println(x.toString());
  }
}

class GraduateStudent extends Student {
}

class Student extends Person {
  public String toString() {
    return "Student";
  }
}

class Person extends Object {
  public String toString() {
    return "Person";
  }
}
```

- An object of a subtype can be used wherever its supertype value is required. This feature is known as *polymorphism*.

Casting Objects

- You have already used the casting operator to convert. In the preceding section, the statement variables of one primitive type to another. *Casting* can also be used to convert an object of one class type to another within an inheritance hierarchy `m(new Student());` assigns the object new Student() to a parameter of the Object type.
- This statement is equivalent to:

```
Object o = new Student(); // Implicit casting
m(o);
```

The statement Object o = new Student(), known as implicit casting, is legal because an instance of Student is automatically an instance of Object.

Casting from Superclass to Subclass

- Explicit casting must be used when casting an object from a superclass to a subclass. This type of casting may not always succeed.

```
Apple x = (Apple)fruit;

Orange x = (Orange)fruit;
```

The `instanceof` Operator

- Use the `instanceof` operator to test whether an object is an instance of a class:

```
Object myObject = new Circle();
... // Some lines of code
/** Perform casting if myObject is an instance of
  Circle */
if (myObject instanceof Circle) {
   System.out.println("The circle diameter is " +
     ((Circle)myObject).getDiameter());
   ...
}
```

The `final` Modifier

- The `final` class cannot be extended:

```
final class Math {
  ...
 }
```

- The `final` variable is a constant:

```
final static double PI = 3.14159;
```

- The `final` method cannot be overridden by its subclasses.