CEN 419

Introduction to Java Programming

Dr. H. Esin ÜNAL

FALL 2021

Slides are modified from original slides of Y. Daniel Liang

Motivations

Suppose you will **define classes to model:**

- ***circles,***
- ***rectangles***
- ***triangles***

These classes have many common features.

What is the best way to design these classes so to avoid redundancy?

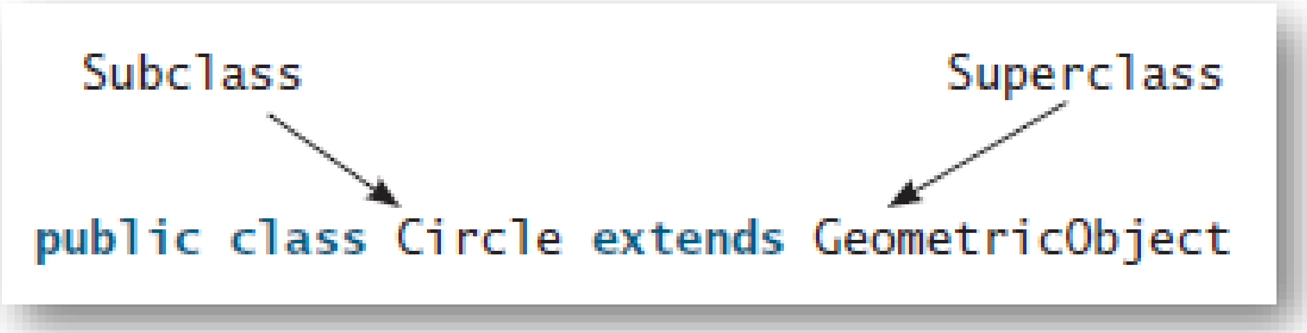
The answer is to use inheritance.

Inheritance

- Object-oriented programming allows you to define new classes from existing classes. This is called **inheritance**.
- You can define a specialized class that extends the generalized class. The specialized classes inherit the properties and methods from the general class.
- Such an *inherited class* is called a **subclass** of its *parent class* or **superclass**.
- It is a mechanism for code reuse.

Superclasses and Subclasses

- The keyword **extends** tells the compiler that the **Circle** class extends the **GeometricObject** class, thus inheriting the methods it has.

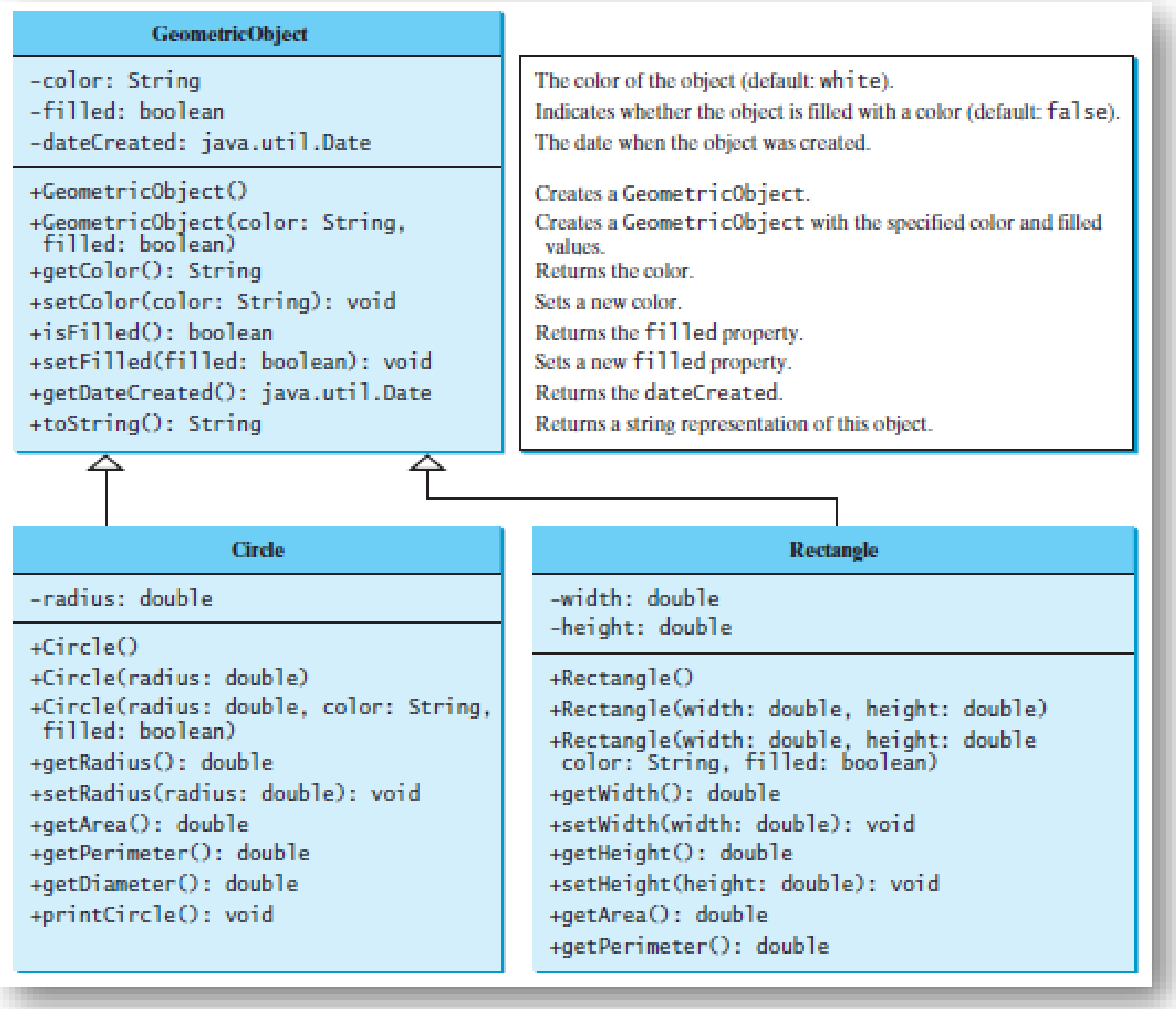


```
public class Circle extends GeometricObject
```

NOTE:

Even if you don’t inherit a class from another class, the compiler automatically inherit the class from **Object** class. Every class you declare is inherited directly or indirectly from the **Object** class.

Superclasses and Subclasses



GeometricObject

```
-color: String
-filled: boolean
-dateCreated: java.util.Date

+GeometricObject()
+GeometricObject(color: String,
 filled: boolean)
+getColor(): String
+setColor(color: String): void
+isFilled(): boolean
+setFilled(filled: boolean): void
+getDateCreated(): java.util.Date
+toString(): String
```

The color of the object (default: white).
Indicates whether the object is filled with a color (default: false).
The date when the object was created.

Creates a GeometricObject.
Creates a GeometricObject with the specified color and filled values.
Returns the color.
Sets a new color.
Returns the filled property.
Sets a new filled property.
Returns the dateCreated.
Returns a string representation of this object.

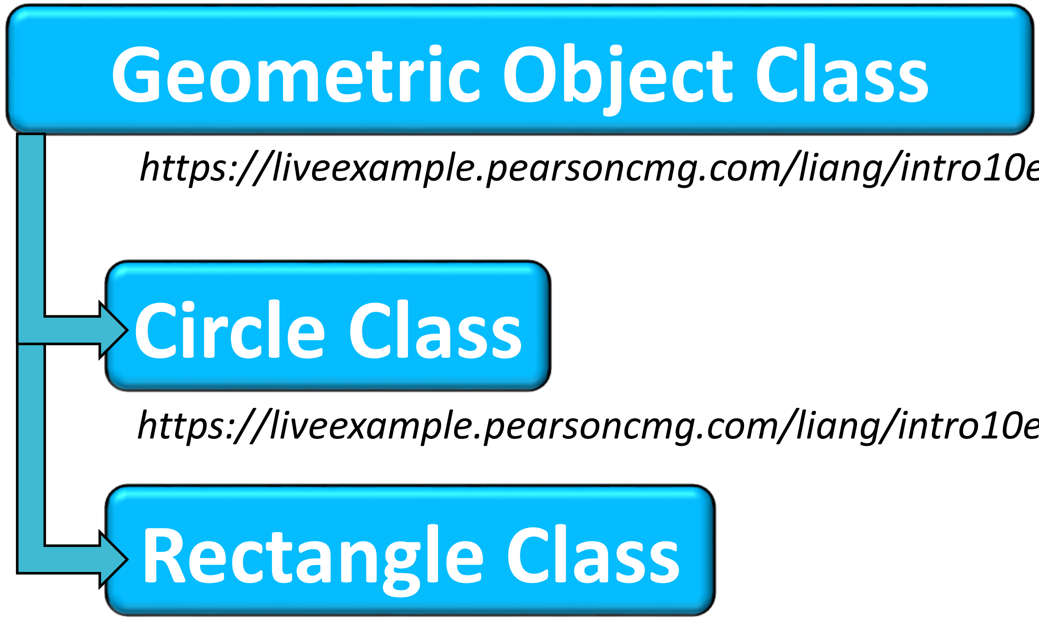
Circle

```
-radius: double

+Circle()
+Circle(radius: double)
+Circle(radius: double, color: String,
 filled: boolean)
+getRadius(): double
+setRadius(radius: double): void
+getArea(): double
+getPerimeter(): double
+getDiameter(): double
+printCircle(): void
```

Rectangle

```
-width: double
-height: double

+Rectangle()
+Rectangle(width: double, height: double)
+Rectangle(width: double, height: double
 color: String, filled: boolean)
+getWidth(): double
+setWidth(width: double): void
+getHeight(): double
+setHeight(height: double): void
+getArea(): double
+getPerimeter(): double
```

Superclasses and Subclasses



Geometric Object Class

https://liveexample.pearsoncmg.com/liang/intro10e/html/SimpleGeometricObject.html

Circle Class

https://liveexample.pearsoncmg.com/liang/intro10e/html/CircleFromSimpleGeometricObject.html

Rectangle Class

https://liveexample.pearsoncmg.com/liang/intro10e/html/RectangleFromSimpleGeometricObject.html

Test Class

https://liveexample.pearsoncmg.com/liang/intro10e/html/TestCircleRectangle.html

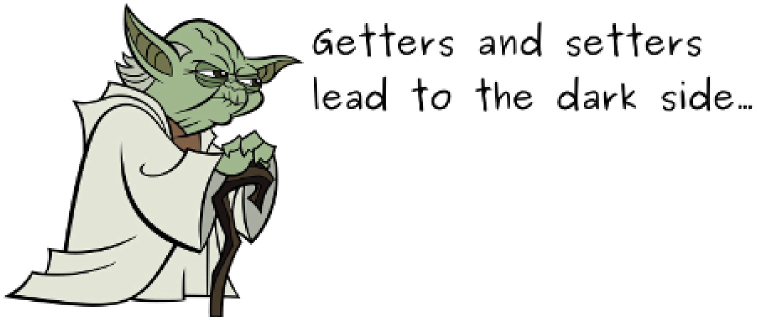
A Simpler Example

```
class Shape{
  int positionX;
  int positionY;
  void move(int newX, int newY){
    positionX = newX;
    positionY = newY;
  }
}
class Circle extends Shape{
  int radius;
  void scale(int scaleFactor){
    radius *= scaleFactor;
  }
}
class Rectangle extends Shape{
  int width;
  int height;
  void scale(int scaleFactor){
    width *= scaleFactor;
    height *= scaleFactor;
  }
}
```

```
Circle c = new Circle();
c.positionX = 10;
c.positionY = 20;
c.radius = 3;
c.move(11,11);
c.scale(5);
```

Important Points of Inheritance

1. Contrary to the conventional interpretation, ***a subclass is not a subset of its superclass***. In fact, a subclass usually contains more information and methods than its superclass.
2. Private data fields in a superclass are not accessible outside the class. Therefore, they cannot be used directly in a subclass. They can, however, be accessed/mutated through public ***getter*** and ***setter*** methods if defined in the superclass.



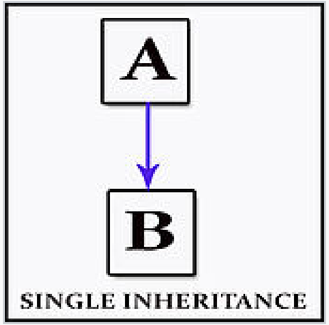
Important Points of Inheritance

3. Inheritance is used to model the is-a relationship.

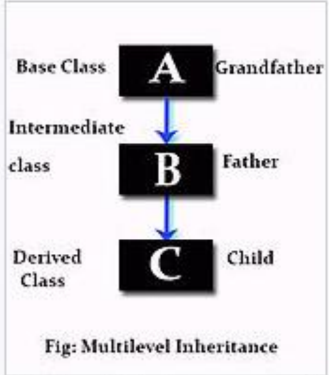
- **Do not blindly extend a class just for the sake of reusing methods.** For example, it makes no sense for a Tree class to extend a Person class, even though they share common properties such as height and weight. A subclass and its superclass must have the is-a relationship.
- **Not all is-a relationships should be modeled using inheritance.** For example, a square is a rectangle, but you should not extend a Square class from a Rectangle class, because the width and height properties are not appropriate for a square. Instead, you should define a Square class to extend the GeometricObject class and define the side property for the side of a square.

Important Points of Inheritance

4. Java does ***not allow multiple inheritance***. A Java class may inherit directly from only one superclass. This restriction is known as ***single inheritance***.
5. However, ***Multilevel inheritance*** is allowed, where a subclass is inherited from another subclass.



Single inheritance



Multilevel inheritance

A derived class with multilevel inheritance is declared as follows:

```
Class A(...); //Base class
Class B : public A(...); //B derived from A
Class C : public B(...); //C derived from B
```

Using the **super** Keyword

- A subclass inherits accessible data fields and methods from its superclass. Does it inherit constructors?
- **No. They are not inherited. They are invoked explicitly or implicitly.**
- The keyword **super** refers to the superclass and can be used:

1. To call a superclass constructor
2. To call a superclass method

Calling Superclass Constructors

- **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 are invoked explicitly or implicitly.
- In order to invoke explicitly use the **super** keyword.

Calling Superclass Constructors

- They **can only be called 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***.

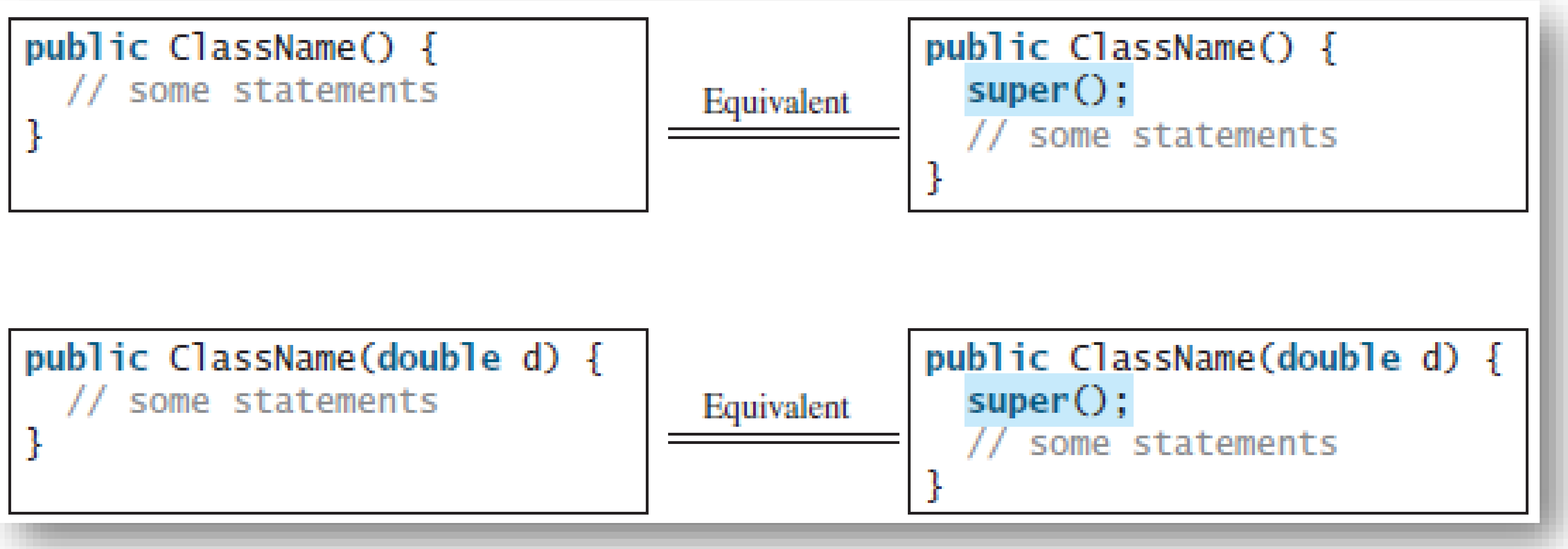
- The syntax to call a superclass's constructor is:

  super(), or **super**(parameters);

- The statement **super()** or **super(arguments)** must be the **first** statement of the subclass's constructor; this is the only way to explicitly invoke a superclass constructor.

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 ClassName() {
  // some statements
}
```

Equivalent

```
public ClassName() {
  super();
  // some statements
}
```

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

Equivalent

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

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.

Constructor Chaining

- Constructing an instance of a class invokes all the **superclasses' constructors** along the inheritance chain.
- The subclass constructor first invokes its superclass constructor before performing its own tasks.
- This is known as **constructor chaining**.

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");
  }
}
```

6. Execute println

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");
  }
}
```

7. Execute println

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");
  }
}
```

8. Execute println

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

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");
  }
}
```

So, the output is:

```
(1) Person's no-arg constructor is invoked
(2) Invoke Employee’s overloaded constructor
(3) Employee's no-arg constructor is invoked
(4) Faculty's no-arg constructor is invoked
```

CAUTION

Consider the following code:

```
public class Apple extends Fruit {
}

class Fruit {
  public Fruit(String name) {
    System.out.println("Fruit's constructor is invoked");
  }
}
```

Since no constructor is explicitly defined in Apple, Apple's default no-arg constructor is defined implicitly. Since **Apple** is a subclass of **Fruit**, **Apple**'s default constructor automatically invokes **Fruit**'s no-arg constructor. However, **Fruit** does not have a no-arg constructor, because **Fruit** has an explicit constructor defined. Therefore, the program cannot be compiled.

Calling Superclass Methods

- The keyword **super** can also be used to reference a method other than the constructor in the superclass.
- The syntax is:

super.method(parameters);

Defining a Subclass

A **subclass inherits from a superclass**.

You can also:

- Add new properties
- Add new methods
- Override the methods of the superclass

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***.

- 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.

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

NOTE

- An instance method can be overridden only if it is accessible. Thus a private method cannot be overridden. *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**.*

Overriding vs. Overloading

```java
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);
  }
}
```

```java
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);
  }
}
```

The example above show the differences between overriding and overloading.
In (a), the method p(double i) in class A overrides the same method in class B.
In (b), the class A has two overloaded methods: p(double i) and p(int i).
The method p(double i) is inherited from B.

Overriding vs. Overloading

- **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 signature and return type; overloaded methods have the same name but a different parameter list.

NOTE

- To avoid mistakes, you **can (not must)** 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;
   }
}
```