

Inheritance

Chapter11

- 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.

Notes

- 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**.
- **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 **accessors/mutators** if defined in the superclass.
- 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.

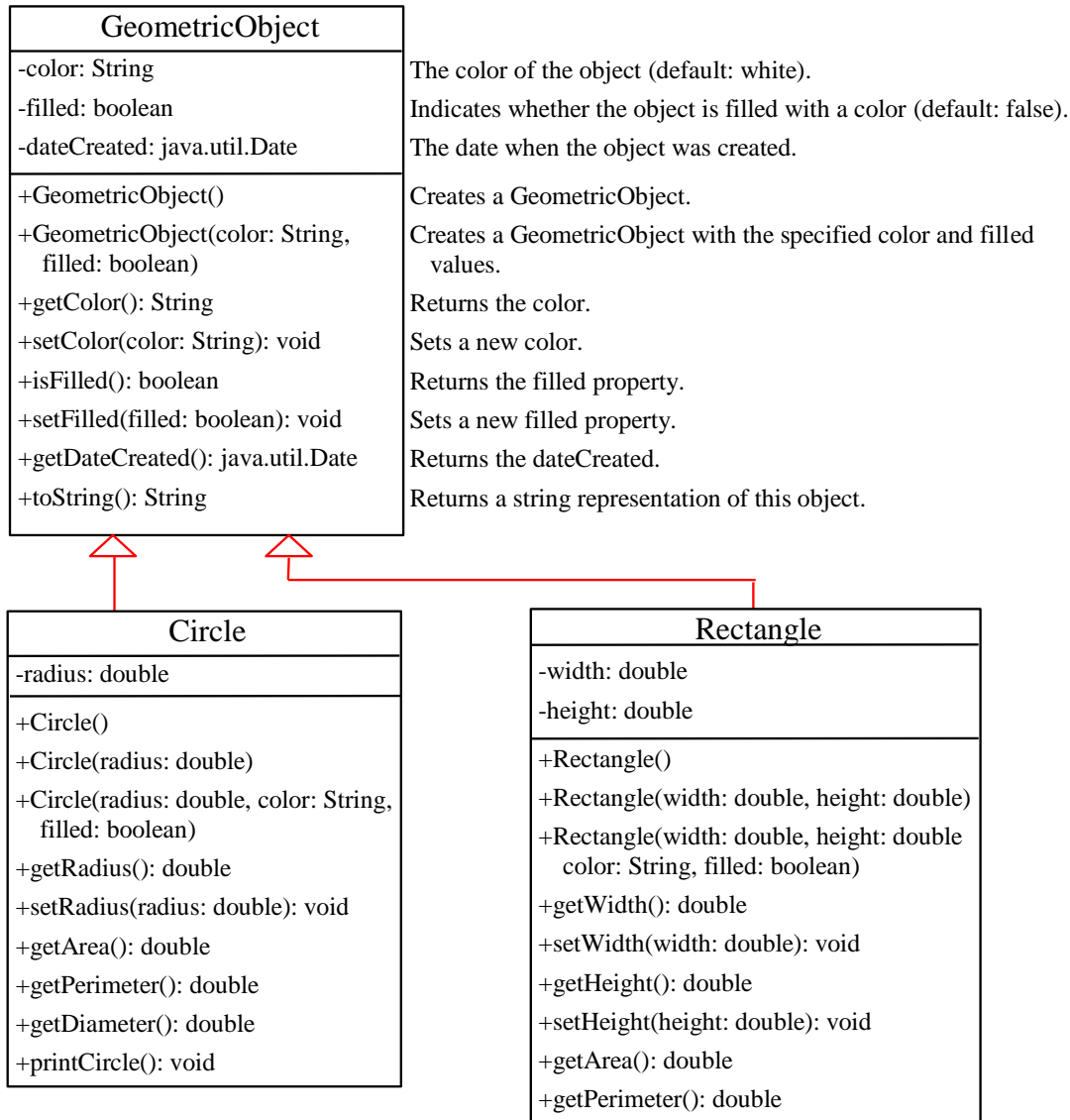
- Inheritance is used to model the is-a relationship. Do not blindly extend a class just for the sake of **reusing methods**.
- Imagine a tree and a person → is it consider as inheritance?
- even though they share common properties such as height and weight. A subclass and its superclass must have the *is-a* relationship
- Some programming languages allow you to derive a subclass from several classes. This capability is known **as multiple inheritance**
- Can you guess what if the java allowed multiple inheritance or not?

Java does not allow multiple inheritance

This restriction is known as single inheritance

If you use the **extends** keyword to define a subclass, it allows **only one** parent class.

Superclasses and Subclasses

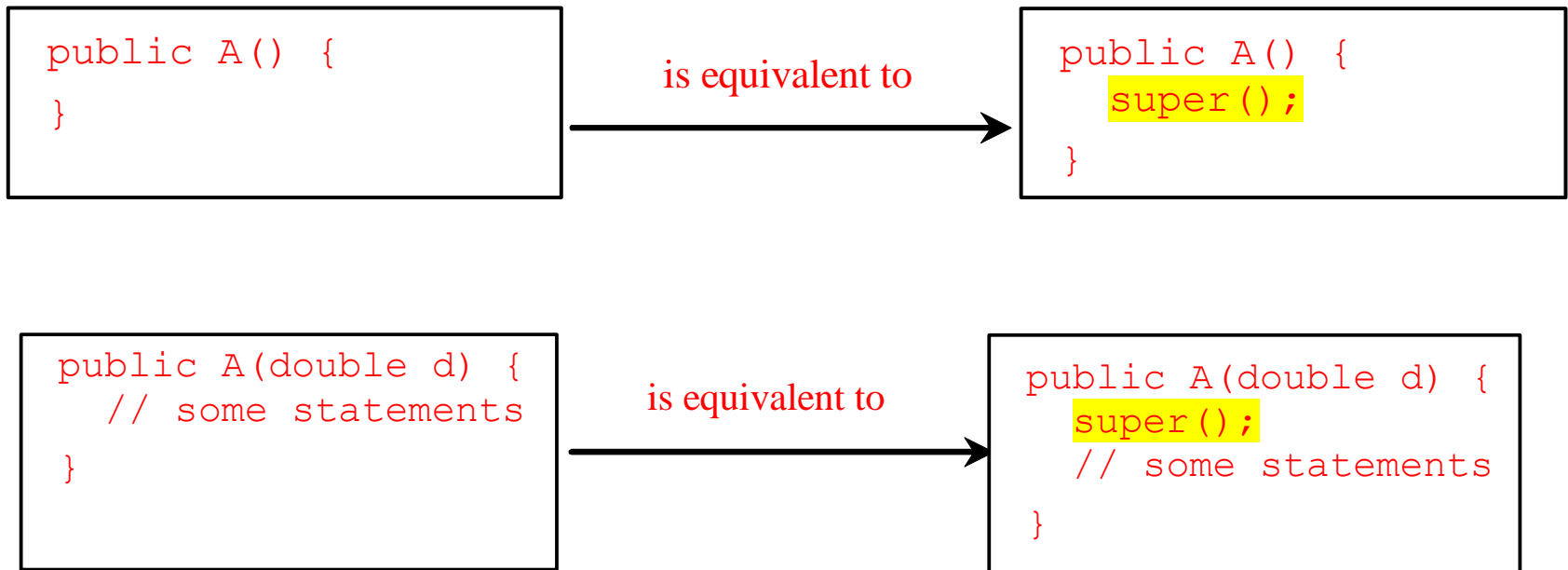


Are superclass's Constructor Inherited?

- No , they are not inherited.
- The superclass constructors will be implicitly or explicitly invoked
- Explicitly by using the keyword: **super**
- The main purpose of the constructor → Create object
- Unlike properties and methods, a superclass's constructors are not inherited in the subclass
- If the super is not explicitly used, the superclass's default constructor is automatically invoked

Tip

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,



Why use super keyword

- Super keyword refers to the superclass of the class in which super appears
- Two ways for using super:
 - To call a superclass constructor
 - To call a superclass method

Tip

- The keyword **super** should be used for calling a superclass constructor
- Invoking a superclass constructor's name in a subclass causes an error
- Can you guess what kind of error?

Syntax error

Java rule: the keyword **super** should appear first in the constructor

- A subclass inherits accessible data fields and methods from its superclass

Calling Superclass Constructors

- A constructor is used to construct an instance of a class(object)
- Unlike properties and methods, the constructors of a superclass are **not** inherited by a subclass.
- They can only be invoked from the constructors of the subclasses using the keyword **super**.
super(), or **super(parameters)**;
- The statement **super()** or **super(arguments)** must be the first statement of the subclass's constructor.

Calling Superclass Methods

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

super.method(parameters);

```
public void printCircle() {  
    System.out.println("The circle is created " +  
        super.getDateCreated() + " and the radius is " + radius);  
}
```

- It is not necessary to put **super** before **getDateCreated()** in this case, because **getDateCreated** is a method in the **GeometricObject** class and is inherited by the **Circle** class.

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.
- **method overriding**—Sometimes it is necessary for the subclass to **modify** the implementation of a method defined 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**.

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 toString method defined in GeometricObject */  
    public String toString() {  
        return super.toString() + "\nradius is " + radius;  
    }  
}
```

NOTE

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

- 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.
- 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**.


```

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);
    }
}

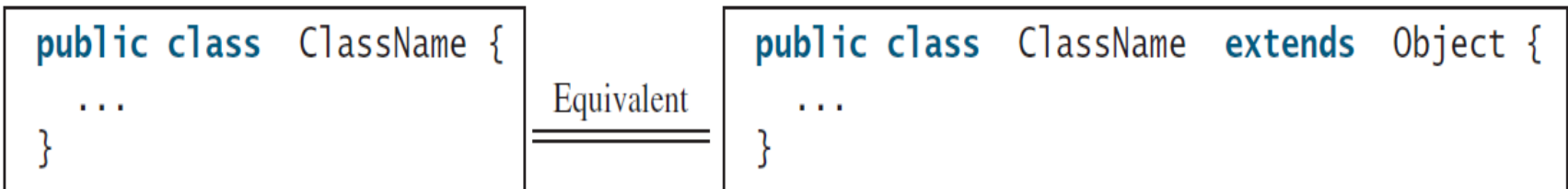
class A extends B {
    // This method overloads the method in B
    public void p(int i) {
        System.out.println(i);
    }
}

```

(b)

Object Class

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



toString() Method

- 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.

The toString() method in Object

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

The 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 digestible string representation of the object.

Constructor chaining

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

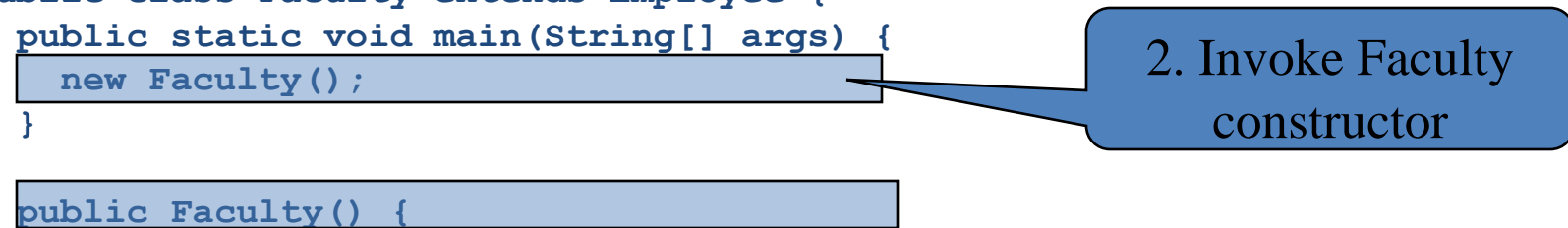
Trace Execution

```
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");  
    }  
}
```



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

Example on the Impact of a Superclass without no-arg Constructor

Find out the errors in the program:

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