

# Chapter 11 Inheritance and Polymorphism

Dr. Asem Kitana

Dr. Abdallah Karakra



# Inheritance

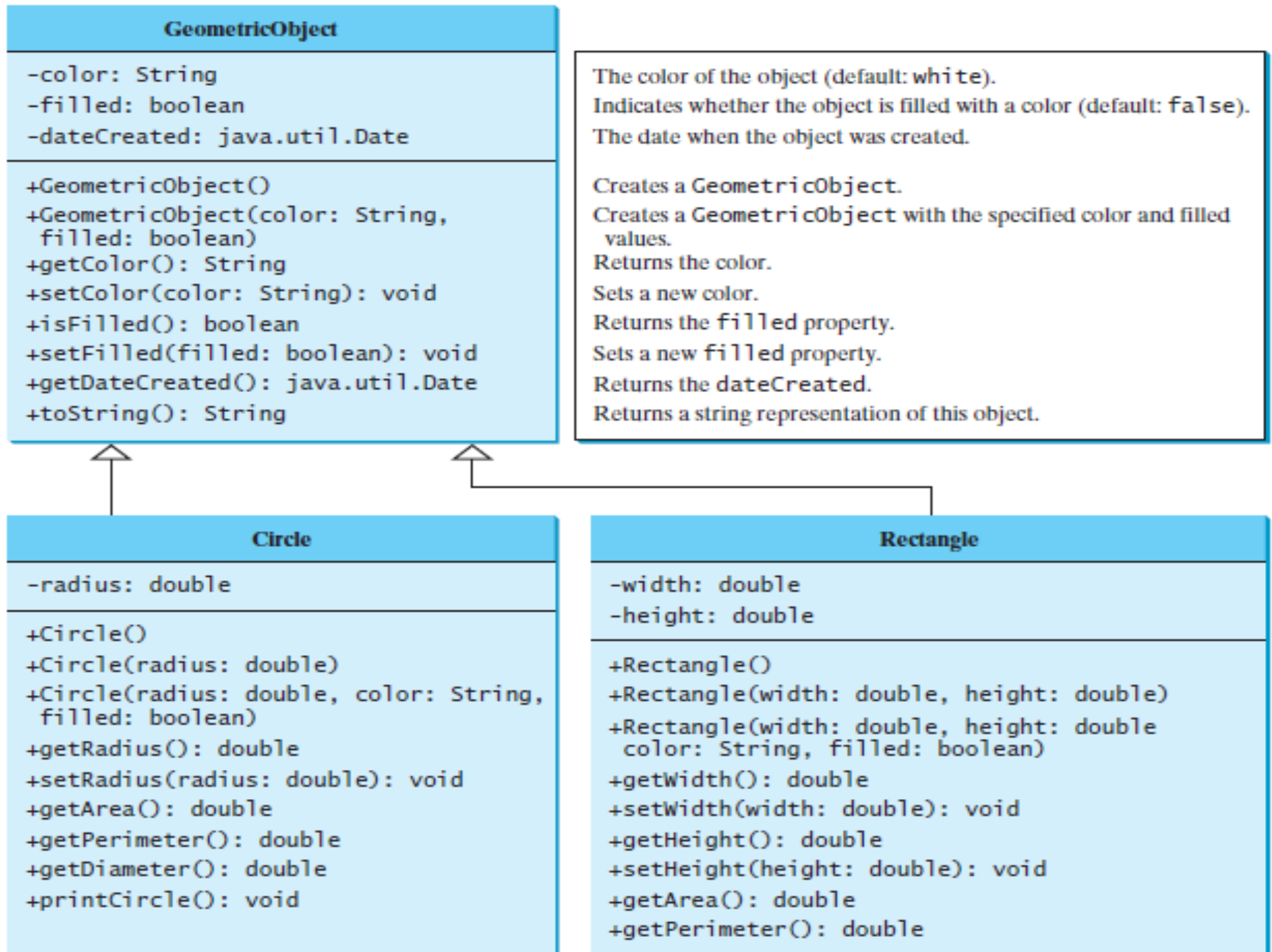
Suppose you will define classes to model circles, rectangles, and triangles. These classes have many common features. What is the best way to design these classes so to avoid redundancy? The answer is to use inheritance.



# Superclasses and Subclasses

- Inheritance enables you to define a general class (i.e., a superclass) and later extend it to more specialized classes (i.e., subclasses).
- A triangular arrow pointing to the superclass is used to denote the inheritance relationship between the classes involved.





```
public class Person {  
    private String name;  
    private int age;  
    // constructor
```

```
public String getName() {return name;}  
public void setName(String name) {this.name=name;}
```

```
public int getAge() {return age;}  
public void setAge(int age) {this.age=age;}
```

```
...  
}
```



General  
class

```
public class Student extends Person { private int  
studentNumber;  
//constructor  
public int getStudentNumber () {return studentNumber;}  
public void setStudentNumber(int studentNumber)  
{this.studentNumber = studentNumber;}
```

```
}
```



specific class

```
public class Person {  
    private String name;  
    private int age;  
    // constructor
```

```
public String getName() {return name;}  
public void setName(String name) {this.name=name;}
```

```
public int getAge() {return age;}  
public void setAge(int age) {this.age=age;}
```

```
...
```

```
}
```

General  
class

```
public class Employee extends Person {  
    private double salary;  
    private String departmentName;  
    public double getSalary() {return salary;}
```

```
public void setSalary(double salary) {this.salary =salary;}
```

```
public String getDepartmentName() {return departmentName;}
```

```
public void setDepartmentName(String departmentName)  
{this.departmentName = departmentName;}
```

specific  
class

# Are superclass's Constructor Inherited?

No. They are **not inherited**.

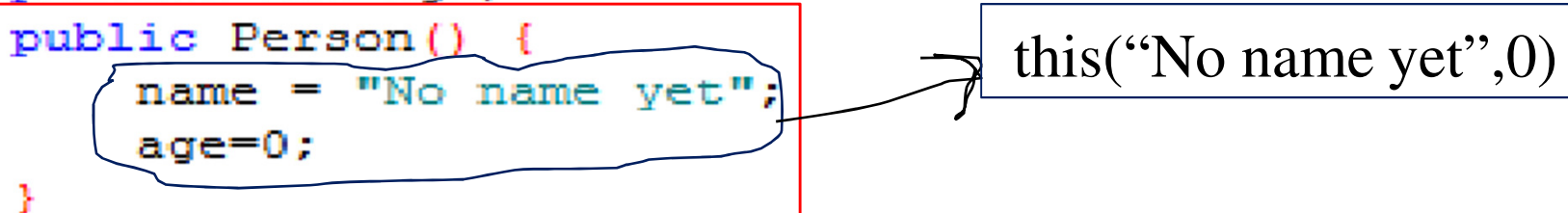
They are invoked **explicitly or implicitly**. **Explicitly** using the **super** keyword.

**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**.*



```
public class Person {
    private String name;
    private int age;
    public Person() {
        name = "No name yet";
        age=0;
    }
    public Person(String name, int age) {

        this.name = name;
        this.age = age;
    }
    public String getName() {return name;}
    public void setName(String name) {this.name = name;}
    public int getAge() {    return age;}
    public void setAge(int age) {    this.age = age;}
    public void writeOutput() {
        System.out.println("Name: " + name + "Age: "+age);
    }
}
```





```
public class Student extends Person
{
```

```
    private int studentNumber;
```

```
    public Student () {
        super ();
        studentNumber = 0; //Indicating
    }
```

```
    public Student (String name,int age ,int studentNumber){
        super (name,age);
        this.studentNumber = studentNumber;
    }
```

```
    public void reset (String name, int age ,int studentNumber){
        setName (name);
        setAge (age);
        this.studentNumber = studentNumber;
    }
```

```
    public int getStudentNumber () {return studentNumber;}
```

```
    public void setStudentNumber(int studentNumber) {this.studentNumber = studentNumber;}
```

```
    public void writeOutput ()
    {
        System.out.println ("Name: " + getName ());
        System.out.println ("Age: " + getAge ());
        System.out.println ("Student Number: " + studentNumber);
    }
```

```
}
```

```
        public String getName () {return name;}
        public void setName (String name) {this.name = name;}
        public int getAge () { return age;}
        public void setAge (int age) { this.age = age;}
        public void writeOutput () {
            System.out.println ("Name: " + name + "Age: "+age);
        }
```

# 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



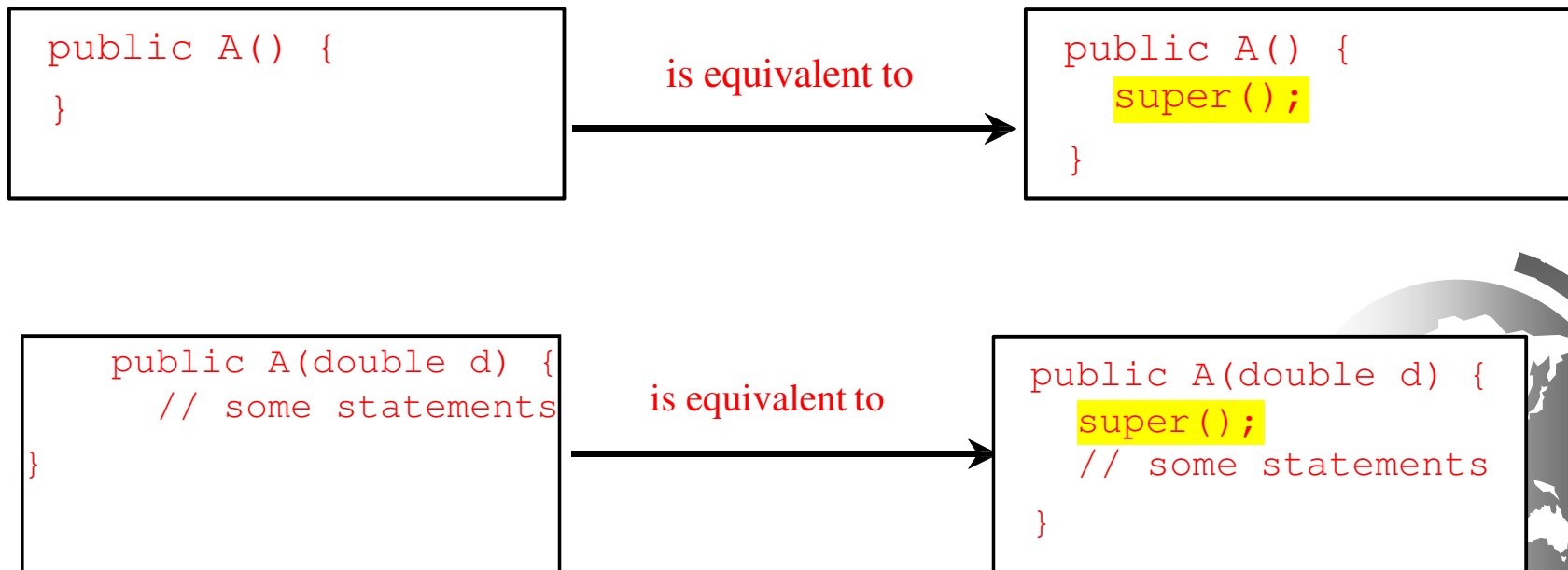
# calling a superclass constructor

- The syntax to call a superclass's constructor is:  
**super()**, or **super(parameters)**
- The statement **super()** invokes the no-arg constructor of its superclass, and the statement **super(arguments)** invokes the superclass constructor that matches the **arguments**.
- The statement **super()** or **super(arguments)** must be the first statement of the subclass's 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,



# 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 known as *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");
    }
}
```

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

**Design Guide** If possible, you should provide a no-arg constructor for every class to make the class easy to extend and to avoid errors

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");  
    }  
}
```

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.



# 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 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.



# 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);
    }
}
```

Output

10.0

10.0

```
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);
    }
}
```

Output

10

20.0

# Calling Superclass Methods

You could rewrite the printCircle() method in the Circle class as follows: (both are correct)

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

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



# NOTE

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.**



# 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**.

The hidden static methods can be invoked using the syntax **SuperClassName.staticMethodName**.

# Check Point

True or false? A subclass is a subset of a superclass.

False.

A subclass is an extension of a superclass and normally contains more details information than its superclass.

What keyword do you use to define a subclass?

The extends keyword is used to define a subclass that extends a superclass.

What is single inheritance? What is multiple inheritance? Does Java support multiple inheritance?

Single inheritance allows a subclass to extend only one superclass. Multiple inheritance allows a subclass to extend multiple classes. Java does not allow multiple inheritance.

# The Object Class and Its Methods

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.

```
public class Circle {  
    ...  
}
```

Equivalent

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



# The toString() method in Object

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

```
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.



# toString

```
public class Test {  
    public static void main (String [] args) {  
        Test testObject = new Test();  
        System.out.println(testObject.toString()); //Test@4617c264  
    }  
}
```

# toString

```
public class Test extends Test2 {  
    public static void main (String [] args) {  
        Test testObject = new Test();  
        System.out.println(testObject.toString()); //Hello from Test2  
    }  
}  
  
class Test2{  
    public String toString() {  
        return "Hello from Test2";  
    }  
}
```

# Polymorphism

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

A class defines a type. A type defined by a subclass is called a **subtype**, and a type defined by its superclass is called a **supertype**. Therefore, you can say that **Circle** is a subtype of **GeometricObject** and **GeometricObject** is a supertype for **Circle**.



# Polymorphism

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.



# Polymorphism

```
public class Test{  
  
    public static void main (String []args){  
        D d = new D();  
        C c= new C();  
        C a=new A();  
        C b =new B();  
        d.poly(c);  
        d.poly(a);  
        d.poly(b);  
    }  
}
```

```
class A extends C{  
  
    public void print(){  
        System.out.println("Hello, From A");  
    }  
}
```

```
class B extends C{  
  
    public void print(){  
        System.out.println("Hello, From B");  
    }  
}
```

```
class C {  
  
    public void print(){  
        System.out.println("Hello, From C");  
    }  
}
```

```
class D {  
  
    public void poly (C obj){  
        obj.print();  
    }  
}
```

Hello, From C

Hello, From A

Hello, From B



# Generic Programming

```
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";  
    }  
}
```

```
}
```

Polymorphism allows methods to be used generically for a wide range of object arguments. This is known as generic programming. If a method's parameter type is a superclass (e.g., `Object`), you may pass an object to this method of any of the parameter's subclasses (e.g., `Student` or `String`). When an object (e.g., a `Student` object or a `String` object) is used in the method, the particular implementation of the method of the object that is invoked (e.g., `toString`) is determined dynamically.

```
Student  
Student  
Person  
java.lang.Object@130c19b
```

# Casting Objects

You have already used the casting operator to convert 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. In the preceding section, the statement

```
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`.

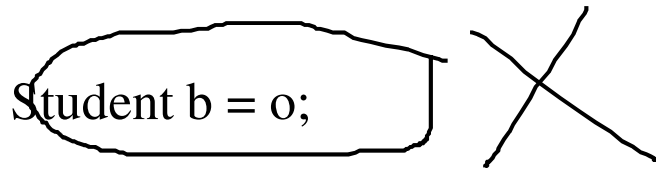




# Why Casting Is Necessary?

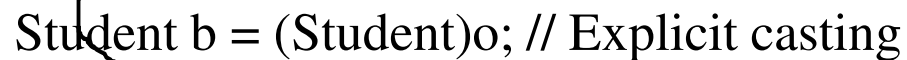
Suppose you want to assign the object reference `o` to a variable of the `Student` type using the following statement:

```
Student b = o;
```



**A compile error would occur.** Why does the statement **Object o = new Student()** work and the statement **Student b = o** doesn't? **This is because a Student object is always an instance of Object, but an Object is not necessarily an instance of Student.** Even though you can see that `o` is really a `Student` object, the compiler is not so clever to know it. To tell the compiler that `o` is a `Student` object, use an explicit casting. The syntax is similar to the one used for casting among primitive data types. Enclose the target object type in parentheses and place it before the object to be cast, as follows:

```
Student b = (Student)o; // Explicit casting
```



# 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());  
...  
}
```



```
public class Test{

    public static void main (String [] args){
        Object obj= new Circle();
        Circle c1 = new Circle();
        Rectangle rect = new Rectangle ();
        Shape sh =new Rectangle();
        boolean res=obj instanceof Circle;
        boolean res2=rect instanceof Rectangle;
        boolean res3=obj instanceof Rectangle;
        boolean res4=c1 instanceof Object;
        boolean res5=c1 instanceof Shape;
        System.out.println(res+ " "+ res2+ " "+res3+ " "+res4+ " "+res5);
    }

}

class Shape {

}

class Circle extends Shape {

}

class Rectangle extends Shape{

}
```

true true false true true

# TIP

To help understand casting, you may also consider the analogy of fruit, apple, and orange with the Fruit class as the superclass for Apple and Orange. An apple is a fruit, so you can always safely assign an instance of Apple to a variable for Fruit. However, a fruit is not necessarily an apple, so you have to use explicit casting to assign an instance of Fruit to a variable of Apple.



# The equals Method

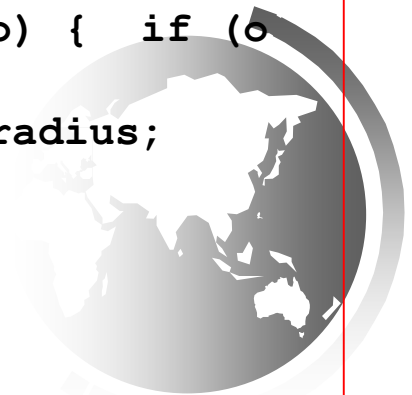
The `equals()` method compares the contents of two objects. The default implementation of the equals method in the Object class is as follows:

```
public boolean equals(Object obj) {  
    return this == obj;  
}
```

This implementation checks whether two reference variables point to the same object using the `==` operator

For example, the equals method is overridden in the Circle class.

```
public boolean equals(Object o) {  
    if (o instanceof Circle) {  
        return radius == ((Circle)o).radius;  
    }  
    else  
        return false;  
}
```



# NOTE

The `==` comparison operator is used for comparing two primitive data type values or for determining whether two objects have the same references. The `equals` method is intended to test whether two objects have the same contents, provided that the method is modified in the defining class of the objects. The `==` operator is stronger than the `equals` method, in that the `==` operator checks whether the two reference variables refer to the same object.

# Example: equals

```
public class TestEquals{  
  
    public static void main (String [] args){  
  
        Circle c1= new Circle (3);  
        Circle c2= new Circle (3);  
        Rectangle r1= new Rectangle (1,2);  
        Rectangle r2= new Rectangle (1,2);  
        System.out.println(c1.equals(c2));  
        System.out.println(r1.equals(r2));  
  
    }  
}
```

```
class Circle{  
  
    private double radius;  
  
    //Setter && getter Method  
  
    public Circle (double radius){  
  
        this.radius=radius;  
  
    }  
}
```

```
class Rectangle {  
  
    private double length,width;  
    public Rectangle (double length, double width){  
        this.length=length;  
        this.width=width;  
    }  
    public boolean equals (Object o){  
        if (o instanceof Rectangle){  
            if (this.length== ((Rectangle)o).length && this.width== ((Rectangle)o).width )  
                return true;  
        }  
  
        return false;  
    }  
}
```



false  
true

# The ArrayList Class

You can create an array to store objects. But the array's size is fixed once the array is created. Java provides the ArrayList class that can be used to store an unlimited number of objects.

*An  
ArrayList  
object can  
be used to  
store a list  
of objects.*

## java.util.ArrayList<E>

```
+ArrayList()  
+add(o: E) : void  
+add(index: int, o: E) : void  
+clear(): void  
+contains(o: Object): boolean  
+get(index: int) : E  
+indexOf(o: Object) : int  
+isEmpty(): boolean  
+lastIndexOf(o: Object) : int  
+remove(o: Object): boolean  
+size(): int  
+remove(index: int) : boolean  
+set(index: int, o: E) : E
```

Creates an empty list.

Appends a new element *o* at the end of this list.

Adds a new element *o* at the specified index in this list.

Removes all the elements from this list. Returns true if this list contains the element *o*.

Returns the element from this list at the specified index.

Returns the index of the first matching element in this list.

Returns true if this list contains no elements.

Returns the index of the last matching element in this list.

Removes the element *o* from this list.

Returns the number of elements in this list. Removes the element at the specified index. Sets the element at the specified index.

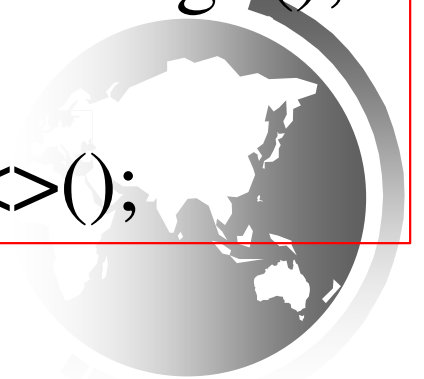


# Generic Type

`ArrayList` is known as a **generic class with a generic type E**. You can specify a concrete type to replace E when creating an `ArrayList`. For example, the following statement creates an `ArrayList` and assigns its reference to variable `cities`. This `ArrayList` object can be used to store strings.

```
ArrayList<String> cities = new ArrayList<String>();
```

```
ArrayList<String> cities = new ArrayList<>();
```



# Differences and Similarities between Arrays and ArrayList

<i>Operation</i>	<i>Array</i>	<i>ArrayList</i>
Creating an array/ArrayList	<code>String[] a = new String[10]</code>	<code>ArrayList&lt;String&gt; list = new ArrayList&lt;&gt;();</code>
Accessing an element	<code>a[index]</code>	<code>list.get(index);</code>
Updating an element	<code>a[index] = "London";</code>	<code>list.set(index, "London");</code>
Returning size	<code>a.length</code>	<code>list.size();</code>
Adding a new element		<code>list.add("London");</code>
Inserting a new element		<code>list.add(index, "London");</code>
Removing an element		<code>list.remove(index);</code>
Removing an element		<code>list.remove(Object);</code>
Removing all elements		<code>list.clear();</code>



# Example

```
import java.util.ArrayList;
public class TestArrayList{

    public static void main(String []args){
        ArrayList<String> list = new ArrayList<String>(); // [ ]
        System.out.println(list.size()); //0
        list.add("Hello"); // [Hello]
        list.add("Hi"); // [Hello,Hi]
        System.out.println("Size is "+list.size() +list); //Size is 2[Hello, Hi]
        list.add(1,"welcome"); // [Hello,welcome,Hi]
        System.out.println("Size is "+list.size() +list); //Size is 3[Hello, welcome, Hi]
        list.set(1,"Salam"); // {Hello, Salam, Hi}
        System.out.println("Size is "+list.size() +list); //Size is 3[Hello, Salam, Hi]
        String s = list.get(0); //Hello
        System.out.println(s); //Hello
        list.remove("Hi"); // [Hello, Salam]
        System.out.println("Size is "+list.size() +list);
        list.clear(); // []
        System.out.println("Size is "+list.size() +list); //Size is 0[]
    }
}
```

# Check Point

Suppose you want to create an **ArrayList** for storing integers. Can you use the following code to create a list?

```
ArrayList<int> list = new ArrayList<int>();
```

No. This will not work because the elements stored in an **ArrayList** must be of an object type. You cannot use a primitive data type such as **int** to replace a generic type. However, you can create an **ArrayList** for storing **Integer** objects as follows:

```
ArrayList<Integer> list = new ArrayList<Integer>();
```



## LISTING 11.9 DistinctNumbers.java

```
1 import java.util.ArrayList;
2 import java.util.Scanner;
3
4 public class DistinctNumbers {
5     public static void main(String[] args) {
6         ArrayList<Integer> list = new ArrayList<Integer>();
7
8         Scanner input = new Scanner(System.in);
9         System.out.print("Enter integers (input ends with 0): ");
10        int value;
11
12        do {
13            value = input.nextInt(); // Read a value from the input
14
15            if (!list.contains(value) && value != 0)
16                list.add(value); // Add the value if it is not in the list
17        } while (value != 0);
18
19        // Display the distinct numbers
20        for (int i = 0; i < list.size(); i++)
21            System.out.print(list.get(i) + " ");
22    }
23 }
```

```
Enter numbers (input ends with 0): 1 2 3 2 1 6 3 4 5 4 5 1 2 3 0 ↵Enter
The distinct numbers are: 1 2 3 6 4 5
```



# Array Lists from/to Arrays

Creating an **ArrayList** from an array of objects:

```
String[] array = {"red", "green", "blue"};  
ArrayList<String> list = new ArrayList<>(Arrays.asList(array));
```

Creating an **array of objects** from an **ArrayList**:

```
String[] array1 = new String[list.size()]; list.toArray(array1);
```

```
import java.util.ArrayList;  
import java.util.Arrays;  
public class HelloWorld{  
  
    public static void main(String []args){  
        String [] obj = {"Hello", "Hi", "Welcome"};  
        ArrayList <String> list = new ArrayList <>(Arrays.asList(obj));  
        System.out.println(list); //[Hello, Hi, Welcome]  
    }  
}
```

# max and min in an Array List

```
String[] array = {"red", "green", "blue"};  
System.out.println(java.util.Collections.max( new  
ArrayList<String>(Arrays.asList(array))));
```

```
String[] array = {"red", "green", "blue"};  
System.out.println(java.util.Collections.min(  
new ArrayList<String>(Arrays.asList(array))));
```

```
import java.util.ArrayList;  
import java.util.Arrays;  
public class HelloWorld{  
  
    public static void main(String []args){  
        String [] obj = {"Hello", "Hi", "Welcome"};  
        ArrayList <String> list = new ArrayList <>(Arrays.asList(obj));  
        System.out.println(list); //[Hello, Hi, Welcome]  
        System.out.println(java.util.Collections.max(list));//Welcome  
        System.out.println(java.util.Collections.min(list));//Hello  
    }  
}
```

# Shuffling an Array List

```
Integer[] array = {3, 5, 95, 4, 15, 34, 3, 6, 5};  
ArrayList<Integer> list = new  
    ArrayList<>(Arrays.asList(array));  
java.util.Collections.shuffle(list);  
System.out.println(list);
```

[95, 4, 5, 6, 34, 3, 5, 15, 3]

[6, 5, 5, 4, 3, 15, 3, 34, 95]

[6, 3, 4, 95, 15, 3, 5, 34, 5]





# The protected Modifier

The `protected` modifier can be applied on data and methods in a class. A protected data or a protected method in a `public class` can be accessed by any class in the same package or its subclasses, even if the subclasses are in a different package.

private, default, protected, public

Visibility increases  
→  
private, none (if no modifier is used), protected, public

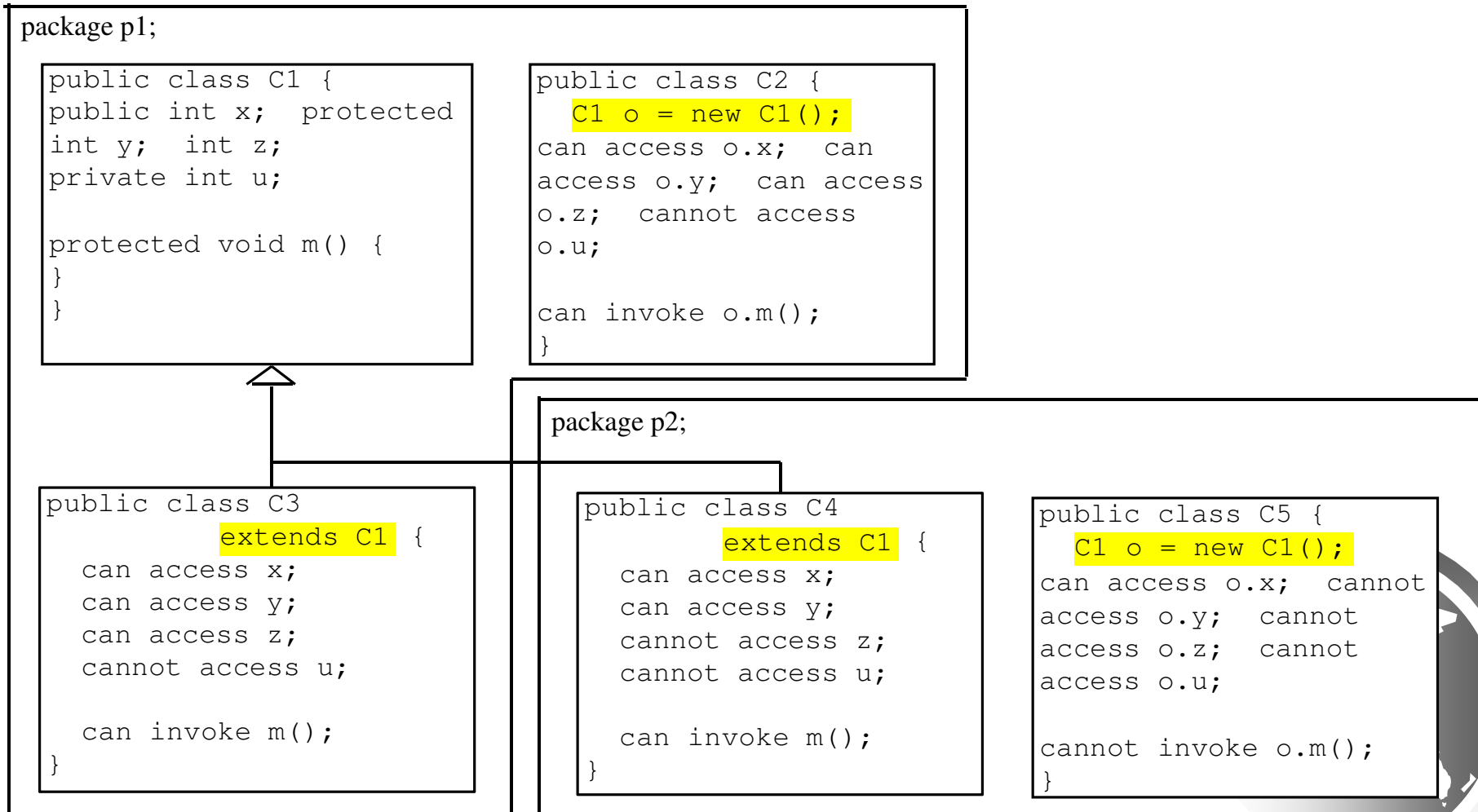


# Accessibility Summary

Modifier on members in a class	Accessed from the same class	Accessed from the same package	Accessed from a subclass	Accessed from a different package
public	✓	✓	✓	✓
protected	✓	✓	✓	–
default	✓	✓	–	–
private	✓	–	–	–



# Visibility Modifiers



## A Subclass **Cannot Weaken the Accessibility**

A subclass may override a **protected method in its superclass and change its visibility to public**. However, a subclass cannot weaken the accessibility of a method defined in the superclass. For example, if a method is defined **as public in the superclass, it must be defined as public in the subclass**.



# NOTE

The modifiers are used on classes and class members (data and methods), except that the final modifier can also be used on local variables in a method. A final local variable is a constant inside a method.



# 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.



# Check Point

How do you prevent a class from being extended? How do you prevent a method from being overridden?

Use the final keyword.



# Check Point

Indicate true or false for the following statements:

- a. A protected data field or method can be accessed by any class in the same package.
- b. A protected data field or method can be accessed by any class in different packages.
- c. A protected data field or method can be accessed by its subclasses in any package.
- d. A final class can have instances.
- e. A final class can be extended.
- f. A final method can be overridden.

g. True.

h. False. (But yes in a subclass that extends the class where the protected data field is defined.)

i. True.

j. Answer: True

k. Answer: False

l. Answer: False



# Review of concepts



# Inheritance

Allow us to specify *relationships between types*

- Abstraction, generalization, specification
- The “is-a” relationship
- Examples?

Why is this useful in programming?

- Next slide



# Why useful: Code Reuse

General functionality can be written once and applied to *\*any\** subclass

Subclasses can specialize by **adding members** and **methods**, or **overriding functions**

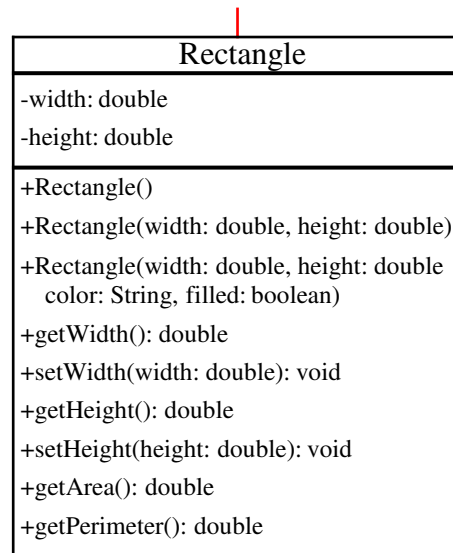
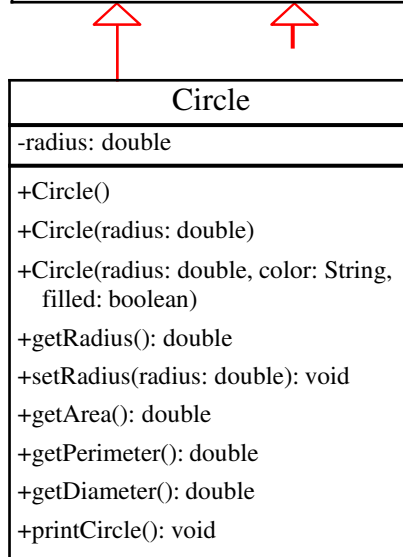
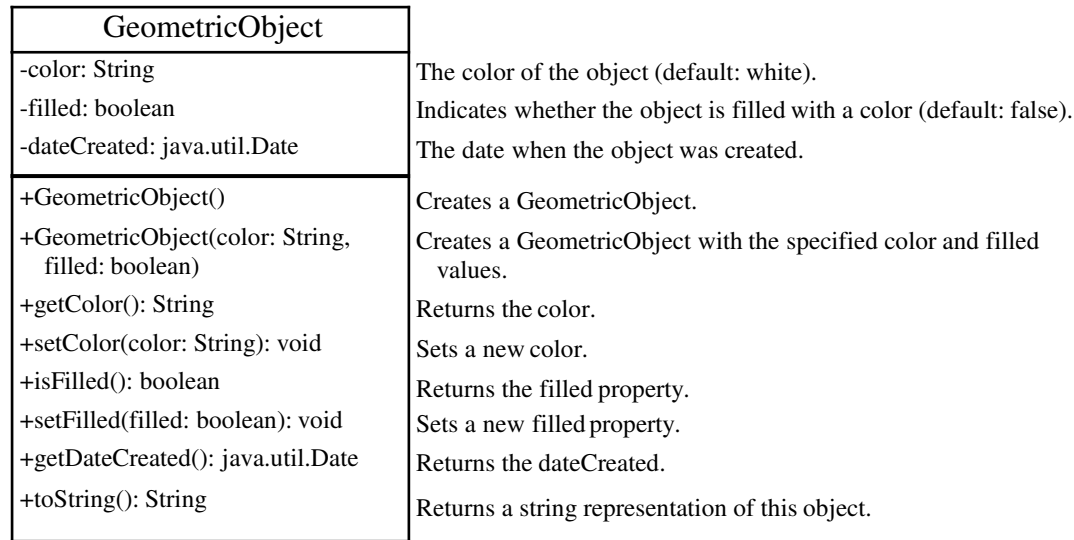


# Inheritance Basics

- ❑ **Inheritance** allows programmer to define a **general class (superclass)**
- ❑ Later you define a more **specific class (subclass)**
  - Adds new details to general definition
- ❑ **New class inherits all properties of initial, general class**



# Superclasses and Subclasses



The **setColor** and **setFilled** methods to set the **color** and **filled** properties. These two public methods are defined in the base class **GeometricObject** and are inherited in **Circle** and **Rectangle**, so they can be used in the derived class.