Inheritance and Polymorphism

Inheritance

- **Inheritance in Java** is a mechanism in which one object acquires all the properties and behaviors of a parent object. It is an important part of OOPs (Object Oriented programming system).
- The idea behind inheritance in Java is that you can create new classes that are built upon existing classes. When you inherit from an existing class, you can reuse methods and fields of the parent class. Moreover, you can add new methods and fields in your current class also.
- Inheritance represents the **IS-A relationship** which is also known as a *parent-child* relationship.

Why use inheritance in java

- For Method Overriding (so runtime polymorphism can be achieved).
- For Code Reusability.

Terms used in Inheritance

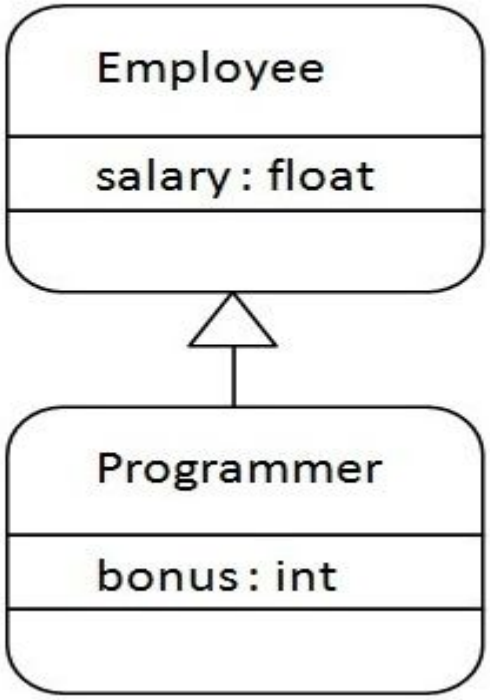
- **Class:** A class is a group of objects which have common properties. It is a template or blueprint from which objects are created.
- **Sub Class/Child Class:** Subclass is a class which inherits the other class. It is also called a derived class, extended class, or child class.
- **Super Class/Parent Class:** Superclass is the class from where a subclass inherits the features. It is also called a base class or a parent class.
- **Reusability:** As the name specifies, reusability is a mechanism which facilitates you to reuse the fields and methods of the existing class when you create a new class. You can use the same fields and methods already defined in the previous class.

The syntax of Java Inheritance

```
class Subclass-name extends Superclass name
{
  //methods and fields
}
```

- The **extends keyword** indicates that you are making a new class that derives from an existing class. The meaning of "extends" is to increase the functionality.
- In the terminology of Java, a class which is inherited is called a parent or superclass, and the new class is called child or subclass.

Java Inheritance Example



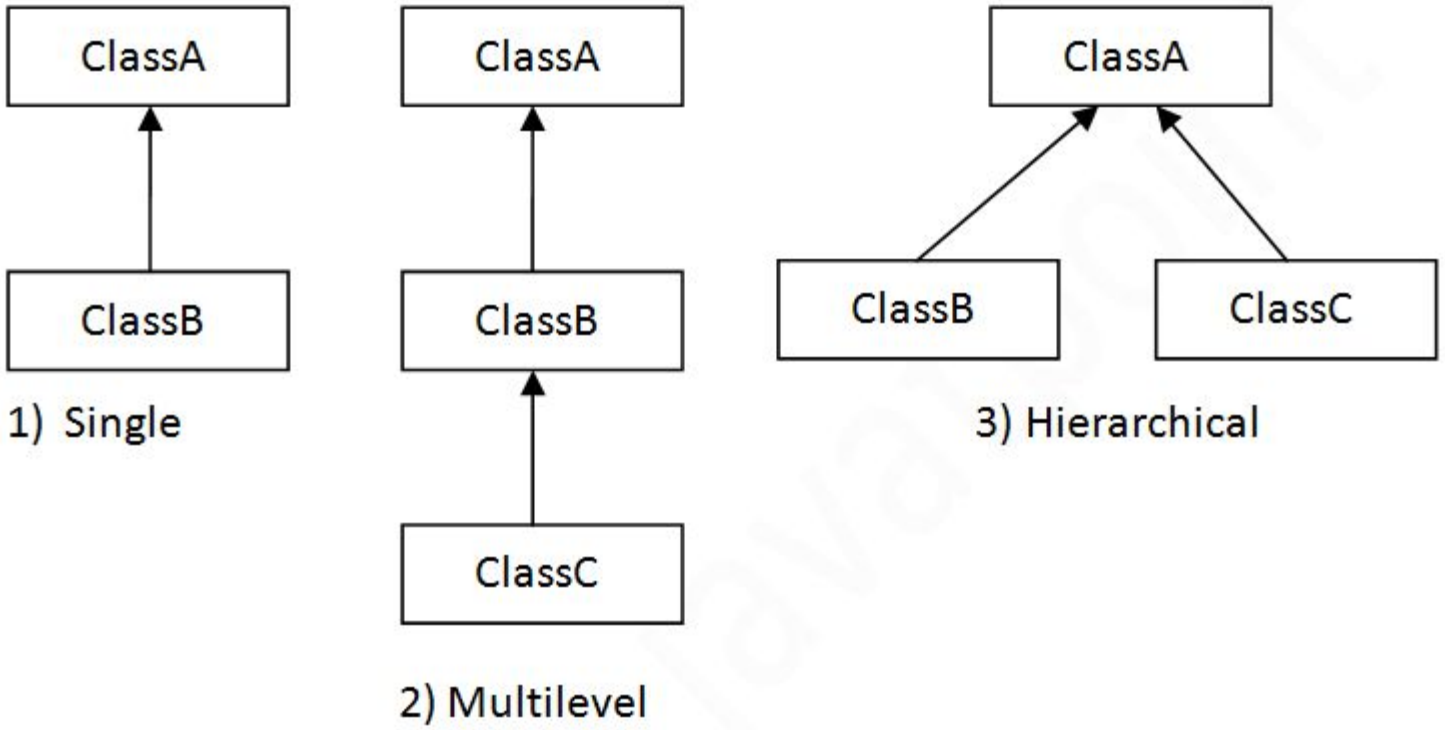
Example

```
class Employee{
 float salary=40000;
}
class Programmer extends Employee{
 int bonus=10000;
 public static void main(String args[]){
   Programmer p=new Programmer();
   System.out.println("Programmer salary is:"+p.salary);
   System.out.println("Bonus of Programmer is:"+p.bonus);
}
}
```

Output: Programmer salary is:40000.0

Bonus of programmer is:10000

Types of inheritance in java



Single Inheritance Example

```
class Animal{
void eat(){System.out.println("eating...");}
}
class Dog extends Animal{
void bark(){System.out.println("barking...");}
}
class TestInheritance{
public static void main(String args[]){
Dog d=new Dog();
d.bark();
d.eat();
}}
```

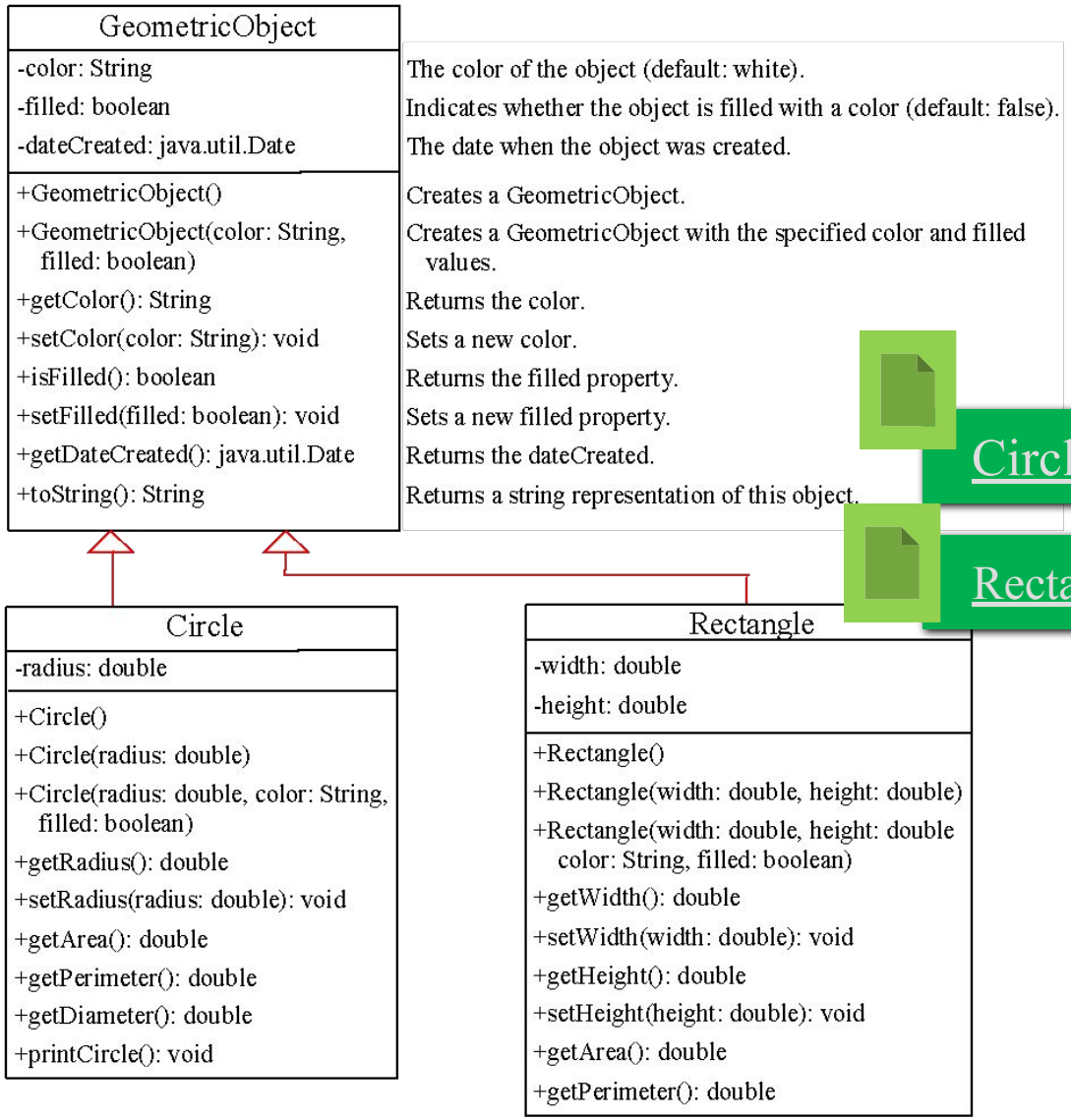
Multilevel Inheritance Example

```
class Animal{
void eat(){System.out.println("eating...");}
}
class Dog extends Animal{
void bark(){System.out.println("barking...");}
}
class BabyDog extends Dog{
void weep(){System.out.println("weeping...");}
}
class TestInheritance2{
public static void main(String args[]){
BabyDog d=new BabyDog();
d.weep();
d.bark();
d.eat();
}}
```

Hierarchical Inheritance Example

```
class Animal{
void eat(){System.out.println("eating...");}
}
class Dog extends Animal{
void bark(){System.out.println("barking...");}
}
class Cat extends Animal{
void meow(){System.out.println("meowing...");}
}
class TestInheritance3{
public static void main(String args[]){
Cat c=new Cat();
c.meow();
c.eat();
}}
```

Superclasses and Subclasses



GeometricObject

CircleFromSimpleGeometricObject

RectangleFromSimpleGeometricObject

TestCircleRectangle

Run

```
class Vehicle {
  protected String brand = "Ford";
  public void honk() {
    System.out.println("Tuut, tuut!");
  }
}

class Car extends Vehicle {
  private String modelName = "Mustang";
  public static void main(String[] args) {
    Car myFastCar = new Car();
    myFastCar.honk();
    System.out.println(myFastCar.brand + " " + myFastCar.modelName);
  }
}
```

Protected Modifier

- Did you notice the protected modifier in Vehicle?
- We set the **brand** attribute in **Vehicle** to a protected access modifier. If it was set to private, the Car class would not be able to access it.

LISTING 11.1 SimpleGeometricObject.java

```
1  public class SimpleGeometricObject {
2    private String color = "white";
3    private boolean filled;
4    private java.util.Date dateCreated;
5
6    /** Construct a default geometric object */
7    public SimpleGeometricObject() {
8      dateCreated = new java.util.Date();
9    }
10
11   /** Construct a geometric object with the specified color
12     *  and filled value */
13   public SimpleGeometricObject(String color, boolean filled) {
14     dateCreated = new java.util.Date();
15     this.color = color;
16     this.filled = filled;
17   }
18
19   /** Return color */
20   public String getColor() {
21     return color;
22   }
23
24   /** Set a new color */
25   public void setColor(String color) {
26     this.color = color;
27   }
28
```

```
29    /** Return filled. Since filled is boolean,
30       its getter method is named isFilled */
31    public boolean isFilled() {
32      return filled;
33    }
34
35    /** Set a new filled */
36    public void setFilled(boolean filled) {
37      this.filled = filled;
38    }
39
40    /** Get dateCreated */
41    public java.util.Date getDateCreated() {
42      return dateCreated;
43    }
44
45    /** Return a string representation of this object */
46    public String toString() {
47      return "created on " + dateCreated + "\ncolor: " + color +
48        " and filled: " + filled;
49    }
50  }
```

LISTING 11.1 SimpleGeometricObject.java

LISTING 11.2 CircleFromSimpleGeometricObject.java

```
1  public class CircleFromSimpleGeometricObject
2      extends SimpleGeometricObject
3    private double radius;
4
5    public CircleFromSimpleGeometricObject() {
6    }
7
8    public CircleFromSimpleGeometricObject(double radius) {
9      this.radius = radius;
10   }
11
12   public CircleFromSimpleGeometricObject(double radius,
13       String color, boolean filled) {
14     this.radius = radius;
15     setColor(color);
16     setFilled(filled);
17   }
18
```

LISTING 11.2 CircleFromSimpleGeometricObject.java

```
19    /** Return radius */
20    public double getRadius() {
21      return radius;
22    }
23
24    /** Set a new radius */
25    public void setRadius(double radius) {
26      this.radius = radius;
27    }
28
29    /** Return area */
30    public double getArea() {
31      return radius * radius * Math.PI;
32    }
33
34    /** Return diameter */
35    public double getDiameter() {
36      return 2 * radius;
37    }
38
39    /** Return perimeter */
40    public double getPerimeter() {
41      return 2 * radius * Math.PI;
42    }
43
44    /** Print the circle info */
45    public void printCircle() {
46      System.out.println("The circle is created " + getDateCreated() +
47        " and the radius is " + radius);
48    }
49  }
```

LISTING 11.3 RectangleFromSimpleGeometricObject.java

```
1  public class RectangleFromSimpleGeometricObject
2      extends SimpleGeometricObject {
3    private double width;
4    private double height;
5
6    public RectangleFromSimpleGeometricObject() {
7    }
8
9    public RectangleFromSimpleGeometricObject(
10       double width, double height) {
11     this.width = width;
12     this.height = height;
13   }
14
15   public RectangleFromSimpleGeometricObject(
16       double width, double height, String color, boolean filled) {
17     this.width = width;
18     this.height = height;
19     setColor(color);
20     setFilled(filled);
21   }
22
```

```
23    /** Return width */
24    public double getWidth() {
25      return width;
26    }
27
28    /** Set a new width */
29    public void setWidth(double width) {
30      this.width = width;
31    }
32
33    /** Return height */
34    public double getHeight() {
35      return height;
36    }
37
38    /** Set a new height */
39    public void setHeight(double height) {
40      this.height = height;
41    }
42
43    /** Return area */
44    public double getArea() {
45      return width * height;
46    }
47
48    /** Return perimeter */
49    public double getPerimeter() {
50      return 2 * (width + height);
51    }
52  }
```

LISTING 11.3 RectangleFromSimpleGeometricObject.java

LISTING 11.4 TestCircleRectangle.java

```
1  public class TestCircleRectangle {
2    public static void main(String[] args) {
3      CircleFromSimpleGeometricObject circle =
4        new CircleFromSimpleGeometricObject(1);
5      System.out.println("A circle " + circle.toString());
6      System.out.println("The color is " + circle.getColor());
7      System.out.println("The radius is " + circle.getRadius());
8      System.out.println("The area is " + circle.getArea());
9      System.out.println("The diameter is " + circle.getDiameter());
10
11     RectangleFromSimpleGeometricObject rectangle =
12       new RectangleFromSimpleGeometricObject(2, 4);
13     System.out.println("\nA rectangle " + rectangle.toString());
14     System.out.println("The area is " + rectangle.getArea());
15     System.out.println("The perimeter is " +
16       rectangle.getPerimeter());
17   }
18 }
```

```
A circle created on Thu Feb 10 19:54:25 EST 2011
color: white and filled: false
The color is white
The radius is 1.0
The area is 3.141592653589793
The diameter is 2.0
A rectangle created on Thu Feb 10 19:54:25 EST 2011
color: white and filled: false
The area is 8.0
The perimeter is 12.0
```

Are superclass's Constructor Inherited?

No. They are not inherited.

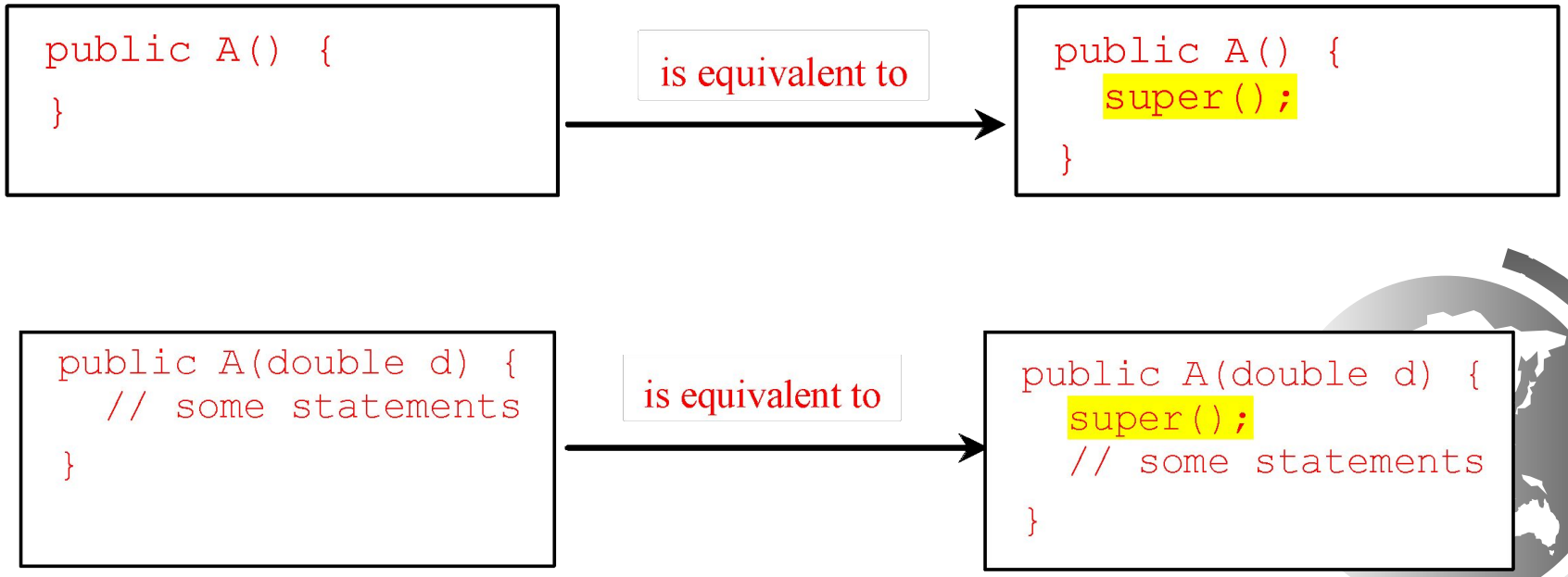
They are invoked explicitly or implicitly.

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,



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
- To call superclass variable

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.

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");
  }
}
```

Defining a Subclass

A subclass inherits from a superclass. You can also:

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

Calling Superclass Methods

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

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

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

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.

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);
  }
}
```

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 a number representing this 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.

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**.

PolymorphismDemo

Run

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));
7      displayObject(new RectangleFromSimpleGeometricObject
8                (1, 1, "black", true));
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 }
```

```
Created on Mon Mar 09 19:25:20 EDT 2011. Color is red
Created on Mon Mar 09 19:25:20 EDT 2011. Color is black
```

Polymorphism, Dynamic Binding and Generic Programming

```
Listing 11.6 DynamicBindingDemo.java
public class DynamicBindingDemo {
  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";
  }
}
```

Method m takes a parameter of the Object type. You can invoke it with any object.

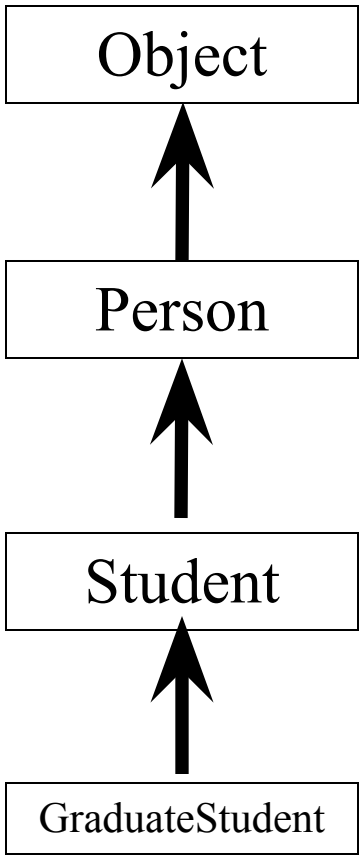
An object of a subtype can be used wherever its supertype value is required. This feature is known as *polymorphism*.

When the method m(Object x) is executed, the argument x's toString method is invoked. x may be an instance of GraduateStudent, Student, Person, or Object. Classes GraduateStudent, Student, Person, and Object have their own implementation of the toString method. Which implementation is used will be determined dynamically by the Java Virtual Machine at runtime. This capability is known as *dynamic binding*.

Animation

DynamicBindingDemo

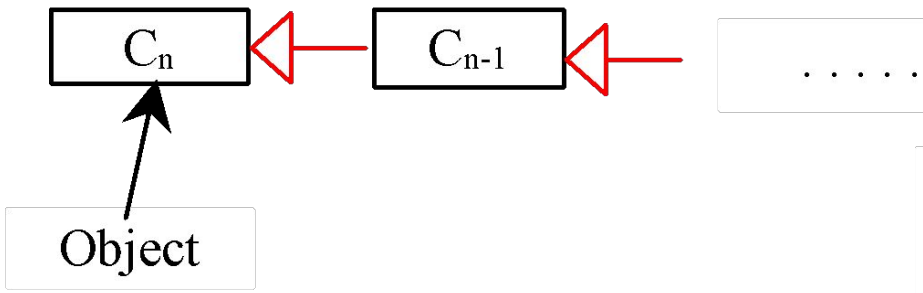
Run




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

Dynamic Binding

Dynamic binding works as follows: Suppose an object o is an instance of classes C_1, C_2, ..., C_{n-1}, and C_n, where C_1 is a subclass of C_2, C_2 is a subclass of C_3, ..., and C_{n-1} is a subclass of C_n. That is, C_n is the most general class, and C_1 is the most specific class. In Java, C_n is the Object class. If o invokes a method p, the JVM searches the implementation for the method p in C_1, C_2, ..., C_{n-1}, and C_n, in this order, until it is found. Once an implementation is found, the search stops and the first-found implementation is invoked.



Since o is an instance of C_1, o is also an instance of C_2, C_3, . . . , C_{n-1}, and C_n

Method Matching vs. Binding

Matching a method signature and binding a method implementation are two issues. The compiler finds a matching method according to parameter type, number of parameters, and order of the parameters at compilation time. A method may be implemented in several subclasses. The Java Virtual Machine dynamically binds the implementation of the method at runtime.

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.

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
```

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());
  ...
}
```

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.

Example: Demonstrating Polymorphism and Casting

This example creates two geometric objects: a circle, and a rectangle, invokes the displayGeometricObject method to display the objects. The displayGeometricObject displays the area and diameter if the object is a circle, and displays area if the object is a rectangle.

CastingDemo

Run

LISTING 11.7 CastingDemo.java

```
1  public class CastingDemo {
2    /** Main method */
3    public static void main(String[] args) {
4      // Create and initialize two objects
5      Object object1 = new CircleFromSimpleGeometricObject(1);
6      Object object2 = new RectangleFromSimpleGeometricObject(1, 1);
7
8      // Display circle and rectangle
9      displayObject(object1);
10     displayObject(object2);
11   }
12
13   /** A method for displaying an object */
14   public static void displayObject(Object object) {
15     if (object instanceof CircleFromSimpleGeometricObject) {
16       System.out.println("The circle area is " +
17         ((CircleFromSimpleGeometricObject)object).getArea());
18       System.out.println("The circle diameter is " +
19         ((CircleFromSimpleGeometricObject)object).getDiameter());
20     }
21     else if (object instanceof
22                   RectangleFromSimpleGeometricObject) {
23       System.out.println("The rectangle area is " +
24         ((RectangleFromSimpleGeometricObject)object).getArea());
25     }
26   }
27 }
```

```
The circle area is 3.141592653589793
The circle diameter is 2.0
The rectangle area is 1.0
```

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;
}
```

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

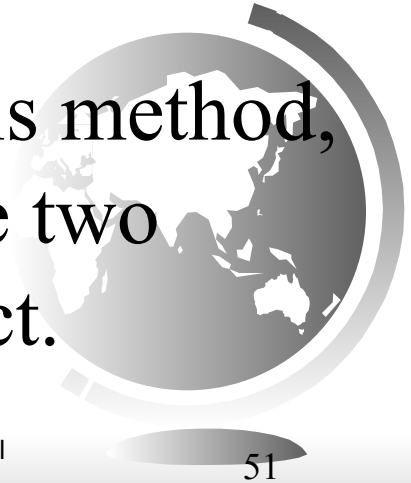
```
public boolean equals(Object o) {
  if (o instanceof Circle) {
    return radius == ((Circle)o).radius;
  }
  else
    return false;
}
```

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.



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.

java.util.ArrayList<E>	
+ArrayList()	Creates an empty list
+add(o: E) : void	Appends a new element o at the end of this list.
+add(index: int, o: E) : void	Adds a new element o at the specified index in this list.
+clear(): void	Removes all the elements from this list.
+contains(o: Object): boolean	Returns true if this list contains the element o.
+get(index: int) : E	Returns the element from this list at the specified index.
+indexOf(o: Object) : int	Returns the index of the first matching element in this list.
+isEmpty(): boolean	Returns true if this list contains no elements.
+lastIndexOf(o: Object) : int	Returns the index of the last matching element in this list.
+remove(o: Object): boolean	Removes the element o from this list.
+size(): int	Returns the number of elements in this list.
+remove(index: int) : boolean	Removes the element at the specified index.
+set(index: int, o: E) : E	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<>();
```

TestArrayList

Run

LISTING 11.8 TestArrayList.java

```
1  import java.util.ArrayList;
2
3  public class TestArrayList {
4    public static void main(String[] args) {
5      // Create a list to store cities
6      ArrayList<String> cityList = new ArrayList<>();
7
8      // Add some cities in the list
9      cityList.add("London");
10     // cityList now contains [London]
11     cityList.add("Denver");
12     // cityList now contains [London, Denver]
13     cityList.add("Paris");
14     // cityList now contains [London, Denver, Paris]
15     cityList.add("Miami");
16     // cityList now contains [London, Denver, Paris, Miami]
17     cityList.add("Seoul");
18     // Contains [London, Denver, Paris, Miami, Seoul]
19     cityList.add("Tokyo");
20     // Contains [London, Denver, Paris, Miami, Seoul, Tokyo]
21
22     System.out.println("List size? " + cityList.size());
23     System.out.println("Is Miami in the list? " +
24       cityList.contains("Miami"));
25     System.out.println("The location of Denver in the list? "
26       + cityList.indexOf("Denver"));
27     System.out.println("Is the list empty? " +
28       cityList.isEmpty()); // Print false
29
```

LISTING 11.8 TestArrayList.java

```
30      // Insert a new city at index 2
31      cityList.add(2, "Xian");
32      // Contains [London, Denver, Xian, Paris, Miami, Seoul, Tokyo]
33
34      // Remove a city from the list
35      cityList.remove("Miami");
36      // Contains [London, Denver, Xian, Paris, Seoul, Tokyo]
37
38      // Remove a city at index 1
39      cityList.remove(1);
40      // Contains [London, Xian, Paris, Seoul, Tokyo]
41
42      // Display the contents in the list
43      System.out.println(cityList.toString());
44
45      // Display the contents in the list in reverse order
46      for (int i = cityList.size() - 1; i >= 0; i--)
47        System.out.print(cityList.get(i) + " ");
48      System.out.println();
49
50      // Create a list to store two circles
51      ArrayList<CircleFromSimpleGeometricObject> list
52        = new ArrayList<>();
53
54      // Add two circles
55      list.add(new CircleFromSimpleGeometricObject(2));
56      list.add(new CircleFromSimpleGeometricObject(3));
```

LISTING 11.8 TestArrayList.java

```
57
58      // Display the area of the first circle in the list
59      System.out.println("The area of the circle? " +
60        list.get(0).getArea());
61    }
62  }
```

```
List size? 6
Is Miami in the list? True
The location of Denver in the list? 1
Is the list empty? false
[London, Xian, Paris, Seoul, Tokyo]
Tokyo Seoul Paris Xian London
The area of the circle? 12.566370614359172
```

Differences and Similarities between Arrays and ArrayList

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

DistinctNumbers

Run

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<>();
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 ↵
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);
```

max and min in an Array List

```
String[] array = {"red", "green", "blue"};

System.out.pritnln(java.util.Collections.max(
  new ArrayList<String>(Arrays.asList(array))));
```

```
String[] array = {"red", "green", "blue"};

System.out.pritnln(java.util.Collections.min(
  new ArrayList<String>(Arrays.asList(array))));
```

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);
```

The MyStack Classes

A stack to hold objects.

MyStack

MyStack
-list: ArrayList
+isEmpty(): boolean
+getSize(): int
+peek(): Object
+pop(): Object
+push(o: Object): void
+search(o: Object): int

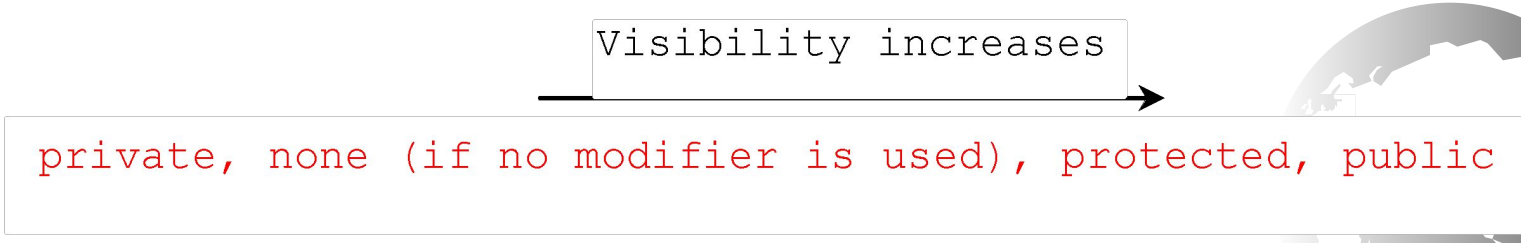
A list to store elements.
Returns true if this stack is empty.
Returns the number of elements in this stack.
Returns the top element in this stack.
Returns and removes the top element in this stack.
Adds a new element to the top of this stack.
Returns the position of the first element in the stack from the top that matches the specified element.

LISTING 11.10 MyStack.java

```
1  import java.util.ArrayList;
2
3  public class MyStack {
4    private ArrayList<Object> list = new ArrayList<>();
5
6    public boolean isEmpty() {
7      return list.isEmpty();
8    }
9
10   public int getSize() {
11     return list.size();
12   }
13
14   public Object peek() {
15     return list.get(getSize() - 1);
16   }
17
18   public Object pop() {
19     Object o = list.get(getSize() - 1);
20     list.remove(getSize() - 1);
21     return o;
22   }
23
24   public void push(Object o) {
25     list.add(o);
26   }
27
28   @Override
29   public String toString() {
30     return "stack: " + list.toString();
31   }
32 }
```

The `protected` Modifier

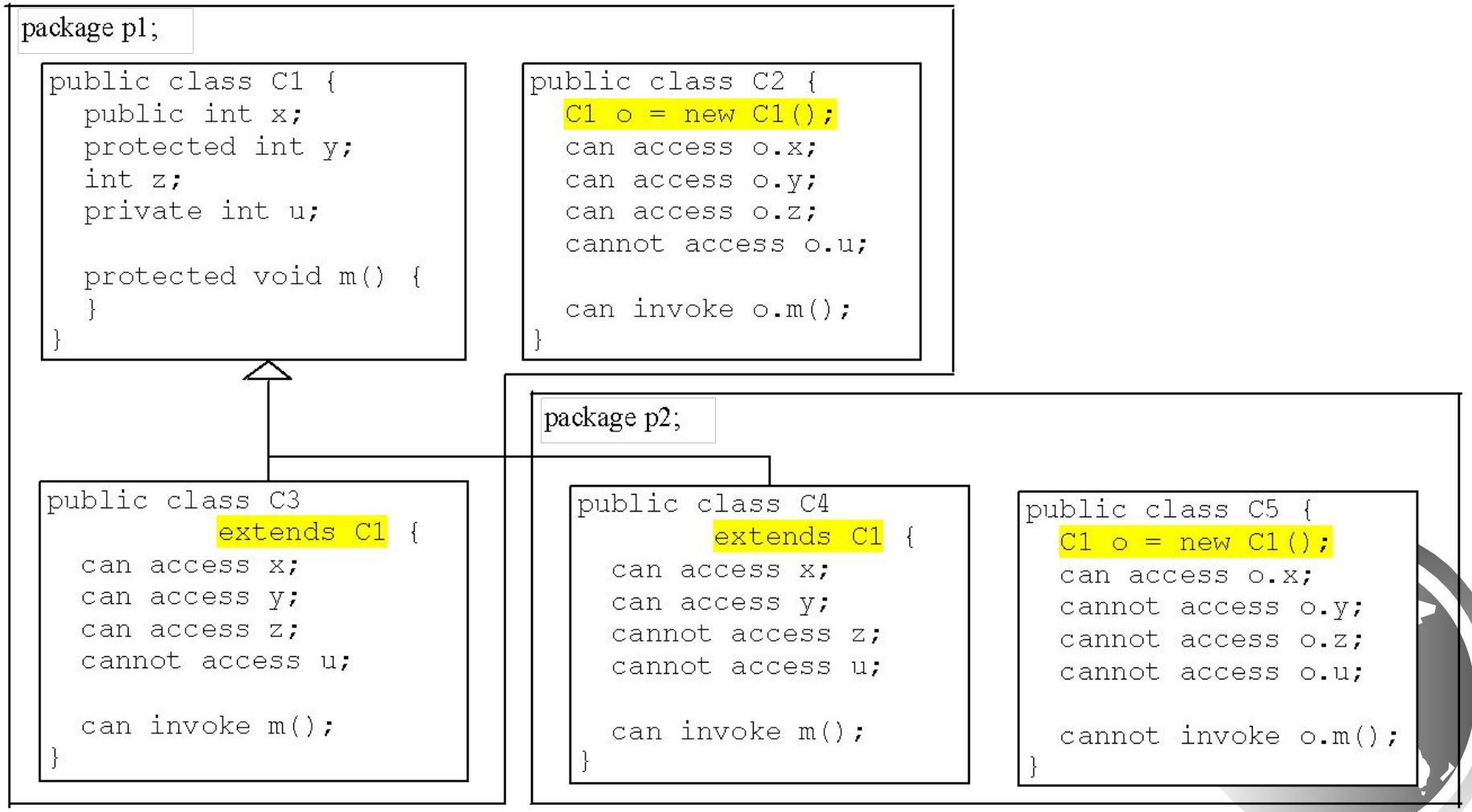
- The `protected` modifier can be applied on data and methods in a class. A protected data or a protected method in a public class can be accessed by any class in the same package or its subclasses, even if the subclasses are in a different package.
- private, default, protected, public



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.