



COMPUTER SCIENCE DEPARTMENT FACULTY OF  
ENGINEERING AND TECHNOLOGY

## ADVANCED PROGRAMMING COMP231

Instructor :Murad Njoun  
Office : Masri322

### Chapter 11 Inheritance and Polymorphism



liang introduction to java programming 11th edition ,2019 Edit By : Mr.Murad Njoun

1

## Motivations

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?

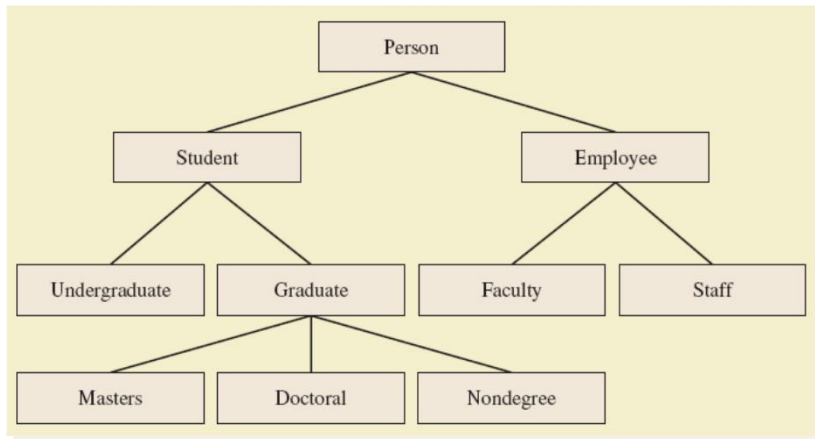
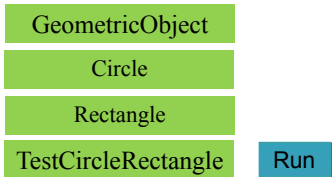
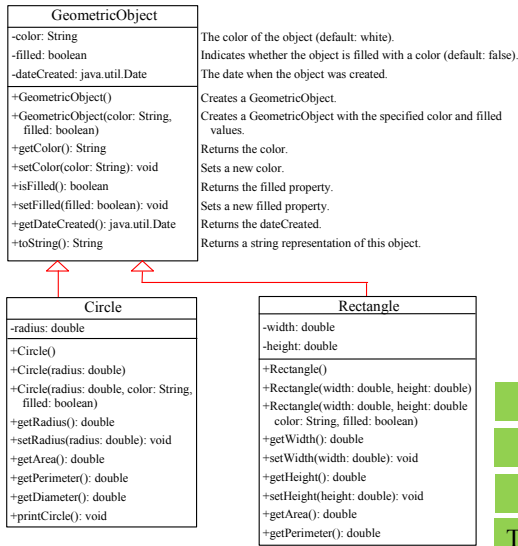
The answer is to use **inheritance**.



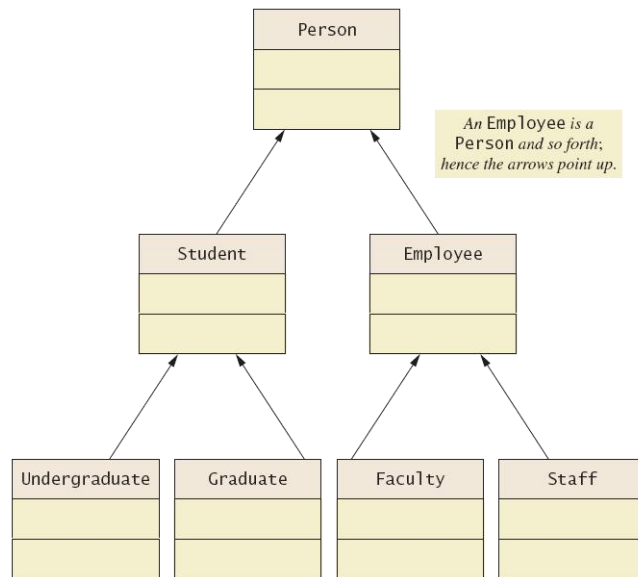
liang introduction to java programming 11th edition ,2019 Edit By : Mr.Murad Njoun

2

# Superclasses and Subclasses

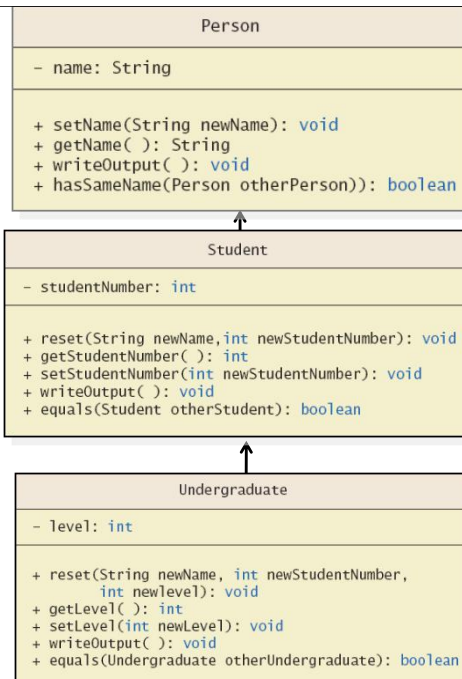


## superclass and subclass



liang introduction to java programming 11th edition ,2019 Edit By : Mr.Murad Njoum

5



## Are superclass's Constructor Inherited?

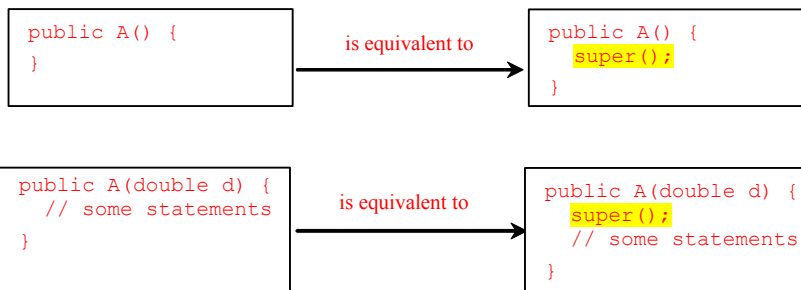
- ❖ No. They are not inherited.
- ❖ They are invoked explicitly or implicitly.
- ❖ Explicitly using the **super keyword**.
- ❖ A constructor is used to construct **an instance of a class**. Unlike properties and methods, **a superclass's constructors** are not inherited in the subclass.
- ❖ They can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



## 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 (using super with constructor, methods, variables)

```
class A{
int a,b;
A(int x, int y)
    {a=x; b=y;}

int multi(){
    return a*b;
}
}
```

```
class B extends A{
int c;
B(int x, int y, int z)
    {super(x,y); //first
    c=z;}
int multi(){
    return a*b*c;
}
}
```



Cont..

```
class A{
void hello(){
    System.out.println("Hello 1");
}
}
class B extend A{
void hello(){
    System.out.println("Hello 2");}
void display()
    {hello(); super.hello();
}
}
```



Cont..

```
class A{
    int a=8;

}
class B extend A{
    int a =7;
    void display()
    { System.out.println(a);
      System.out.println(super.a);
    }
}
```



```
class A {
    int a, b;
    A(int x, int y) {
        a = x;
        b = y;
    }
    int multi() {
        return a * b;
    }
}
class B extends A {
    int c;

    B(int x, int y, int z) {
        super(x, y); // first
        c = z;
    }

    public int methodX() {
        int d = super.multi();
        return d;
    }

    @Override
    int multi() {
        return a * b * c;
    }
}
```

```
public class testmethodoverload {
    public static void main(String[] args) {
        A a = new A(5, 5);
        System.out.println(a.multi());

        B b = new B(5, 10, 15);
        System.out.println(b.multi());
        System.out.println(b.methodX());
    }
}
```

25  
750  
50



## 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");
    }
}
```



animation

## 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





animation

## 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



animation

## 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



animation

## 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



animation

## 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



animation

## 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



animation

## 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



animation

## 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



animation

## 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



```

class A {
    A(){
        System.out.println("Print A constructor");
    }
}

class B extends A {
    B(){
        System.out.println("Print B constructor");
    }
}

class C extends B {
    C(){
        System.out.println("Print C constructor");
    }
}

public class superClass {

    public static void main(String[] args) {
        new C();
    }
}

```

```

class A {
    A(){
        System.out.println("Print A constructor");
    }
}

class B extends A {
    B(){
        super();
        System.out.println("Print B constructor");
    }
}

class C extends B {
    C(){
        super();
        System.out.println("Print C constructor");
    }
}

public class superClass {
    public static void main(String[] args) {
        new C();
    }
}

```

java

liang introduction to java programming 11th edition ,2019 Edit By : Mr.Murad Njoun

25

❖ Example on the Impact of a Superclass without no-arg Constructor They can only be invoked from the subclasses' constructors, using the keyword super.

### Find out the errors in the program:

```

public class Apple extends Fruit {
    Apple(){
        super("Hi:");
    }
}

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

```

Fruit class doesn't contains default constructor  
i.e , Fruit(){} doesn't exists

Hi:Fruit's constructor is invoked



liang introduction to java programming 11th edition ,2019 Edit By : Mr.Murad Njoun

26

❖ If the keyword super is not explicitly used, the superclass's no-arg constructor is automatically invoked.

Find out the errors in the program:

```
public class Apple extends Fruit {
    Apple(){}
}

class Fruit {
    public Fruit() {
        System.out.println("C1: Fruit's constructor is invoked");
    }
    public Fruit(String name) {
        System.out.println("C2 :Fruit's constructor is invoked");
    }
}
```

Fruit class is defined , default constructor  
Super is implicitly created

C1 :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

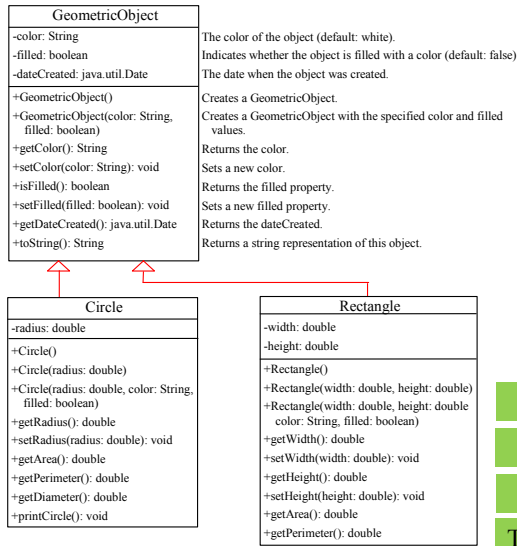
A subclass inherits methods from a superclass. Sometimes it is necessary for the subclass to modify the implementation of a method defined in the superclass. This is referred to as **method overriding**.

```
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;
    }
}
```



## Superclasses and Subclasses



liang introduction to java programming 11th edition ,2019 Edit By : Mr.Murad Njoum

31

## 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**.

**Both methods are private, not related to each other.**

**(hidden from out side of superclass and subclass)**



liang introduction to java programming 11th edition ,2019 Edit By : Mr.Murad Njoum

32



## Example

```
public class superClass {  
  
    public static void main(String[] args) {  
        C c=new C();  
        c.printMesg("Hi");  
    }  
}
```

Syntax error ? Why ?

Because private method print Message in super class is only for Class A

```
package project;  
  
class A {  
    A(){  
        System.out.println("Print A constructor");  
    }  
    private void printMesg(String str) {  
        System.out.print(str);  
    }  
}  
  
class B extends A {  
    B(){  
        super();  
        System.out.println("Print B constructor");  
    }  
}  
  
class C extends B {  
    C(){  
        super();  
        System.out.println("Print C constructor");  
    }  
    void printMesg(String str){  
        super.printMesg(str);  
    }  
}
```



## 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**.

→ Both methods in superclass and subclass defined as static, then you cannot view in related object.



# Example

```
public class staticExample {  
  
    public static void main(String[] args)  
    {  
        Three three = new Three();  
        three.printMesg("Hi");  
    }  
}
```

## Syntax error ? Why ?

Because static method print Message in super class cannot override and `super.printMesg(str)`, `super` can't use with static

liang introduction to java programming 11th edition ,2019 Edit By : Mr.Murad Njoum

```
package chapter11;  
  
class One {  
    One() {  
        System.out.println("Print A constructor");  
    }  
  
    public static void printMesg(String str) {  
        System.out.println(str + " from A");  
    }  
}  
  
class Two extends One {  
    Two() {  
        super();  
        System.out.println("Print B constructor");  
    }  
}  
  
class Three extends Two {  
    Three() {  
        super();  
        System.out.println("Print C constructor");  
    }  
  
    public void printMesg(String str) {  
        super.printMesg(str);  
        System.out.println(str + " from B");  
    }  
}
```



# Overriding تجاوز vs. Overloading الزائد التحميل

```
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);  
    }  
}
```

**Note:** methods are Overriding for superclass in subclass (in inheritance ) While method Overloading at same class for different passed parameters to methods in class.

liang introduction to java programming 11th edition ,2019 Edit By : Mr.Murad Njoum



## 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 (many behavior )

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**.

**Static polymorphism** in Java is achieved by method **overloading**

**Dynamic polymorphism** in Java is achieved by method **overriding**

PolymorphismDemo

Run



## Polymorphism, Dynamic Binding and 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";
    }
}
```

Student  
Student  
Person  
java.lang.Object@15db9742

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**.

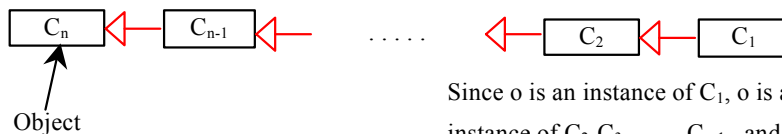
Run

DynamicBindingDemo



## Dynamic Binding

**Dynamic binding works as follows:** Suppose an object  $o$  is an instance of **classes**  $C_1, C_2, \dots, C_{n-1}$ , 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  $C_1, C_2, \dots, C_{n-1}$  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.**



## Method Matching vs. Binding (link, connect)

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;      Fruit f = new Apple();
```

```
Orange x = (Orange)fruit;   Fruit o = new Orange();
```



# 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



liang introduction to java programming 11th edition ,2019 Edit By : Mr.Murad Njoum

49

```
public class CastingDemo {
    /** Main method */
    public static void main(String[] args) {
        // Create and initialize two objects
        Object object1 = new Circle(1);
        Object object2 = new Rectangle(1, 1);

        // Display circle and rectangle
        displayObject(object1);
        displayObject(object2);
    }

    /** A method for displaying an object */
    public static void displayObject(Object myobject) {
        if (myobject instanceof Circle) {
            System.out.println("The circle area is " + ((Circle)myobject).getArea());
            System.out.println("The circle diameter is " + ((Circle)myobject).getDiameter());
        }
        else if (myobject instanceof Rectangle) {
            System.out.println("The rectangle area is " + ((Rectangle)myobject).getArea());
        }
    }
}
```



liang introduction to java programming 11th edition ,2019 Edit By : Mr.Murad Njoum

50

## 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.

<b>java.util.ArrayList&lt;E&gt;</b>	
+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



```

import java.util.ArrayList;

public class TestArrayList {
    public static void main(String[] args) {
        // Create a list to store cities
        ArrayList<String> cityList = new ArrayList<>();

        // Add some cities in the list
        cityList.add("Ramallah");
        // cityList now contains [Ramallah]
        cityList.add("Jericho");
        // cityList now contains [Ramallah, Jericho]
        cityList.add("Jerusalem");
        // cityList now contains [Ramallah, Jericho, Jerusalem]
        cityList.add("Nablus");
        // contains [Ramallah, Jericho, Jerusalem, Nablus]
        cityList.add("Jinen");
        // contains [Ramallah, Jericho, Jerusalem, Nablus, Jinen]
        cityList.add("Hebron");
        //contains[Ramallah, Jericho, Jerusalem, Nablus, Jinen,
        Hebron]

        System.out.println("List size? " + cityList.size());
        System.out.println("Is Ramallah in the list? " +
            cityList.contains("Ramallah"));
        System.out.println("The location of Hebron in the list? "
            + cityList.indexOf("Hebron"));
        System.out.println("Is the list empty? " +
            cityList.isEmpty()); // Print false
        // Insert a new city at index 2
    }
}

```

```

cityList.add(1, "Salfet");
// contains [Ramallah, Salfet, Jerusalem, Nablus, Jinen, Hebron]

// Remove a city from the list
cityList.remove(" Nablus ");
// contains[Ramallah, Salfet, Jerusalem, Jinen, Hebron]

// Remove a city at index 1
cityList.remove(4);
// contains [Ramallah, Salfet, Jerusalem, Jinen]

// Display the contents in the list
System.out.println(cityList.toString());

// Display the contents in the list in reverse order
for (int i = cityList.size() - 1; i >= 0; i--)
    System.out.print(cityList.get(i) + " ");
System.out.println();

// Create a list to store two circles
ArrayList<Circle> list = new ArrayList<>();

// Add two circles
list.add(new Circle(2));
list.add(new Circle(3));

// Display the area of the first circle in the list
System.out.println("The area of the circle? " +
    list.get(0).getArea());
}
}

```

## Differences and Similarities between Arrays and ArrayList

Operation	Array	ArrayList
Creating an array/ArrayList	String[] a = <b>new</b> String [10]	ArrayList<String> list = <b>new</b> 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



## Array Lists **from/to** Arrays

Creating an ArrayList from an array of objects:

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



## Shuffling an Array List

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



## 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

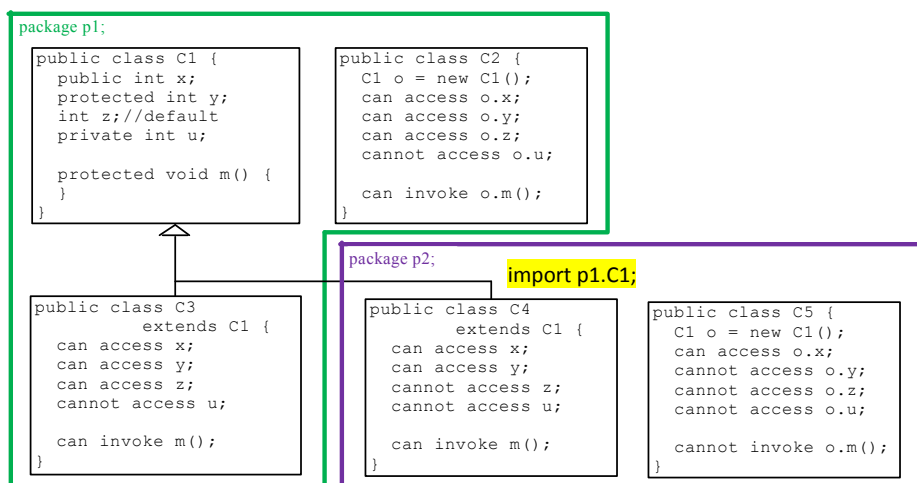


# Accessibility Summary

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** (رؤيتها وليس الوصول لها) **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 (default,public, abstarct,final) are used on class**

**The modifiers (default,public,protected,private, abstarct,final) 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.

