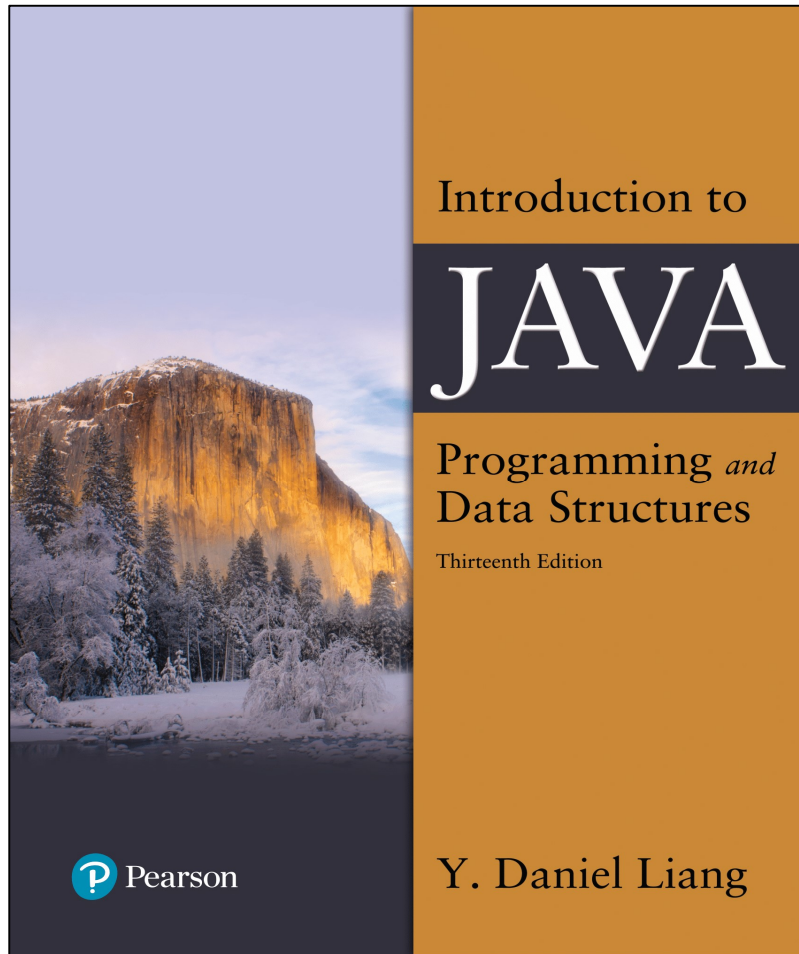


# Introduction to Java Programming and Data Structures

Thirteenth Edition



## Chapter 11

Inheritance and  
Polymorphism

# 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.

# Objectives (1 of 2)

**11.1** To define a subclass from a superclass through inheritance (§11.2).

**11.2** To invoke the superclass's constructors and methods using the **super** keyword (§11.3).

**11.3** To override instance methods in the subclass (§11.4).

**11.4** To distinguish differences between overriding and overloading (§11.5).

**11.5** To explore the `toString()` method in the `Object` class (§11.6).

**11.6** To discover polymorphism and dynamic binding (§§11.7–11.8).

**11.7** To describe casting and explain why explicit downcasting is necessary (§11.9).

# Objectives (2 of 2)

**11.8** To explore the `equals` method in the `Object` class (§11.10).

**11.9** To store, retrieve, and manipulate objects in an `ArrayList` (§11.11).

**11.10** To construct an array list from an array, to sort and shuffle a list, and to obtain max and min element from a list (§11.12).

**11.11** To implement a `Stack` class using `ArrayList` (§11.13).

**11.12** To enable data and methods in a superclass accessible from subclasses using the `protected` visibility modifier (§11.14).

**11.13** To prevent class extending and method overriding using the `final` modifier (§11.14).

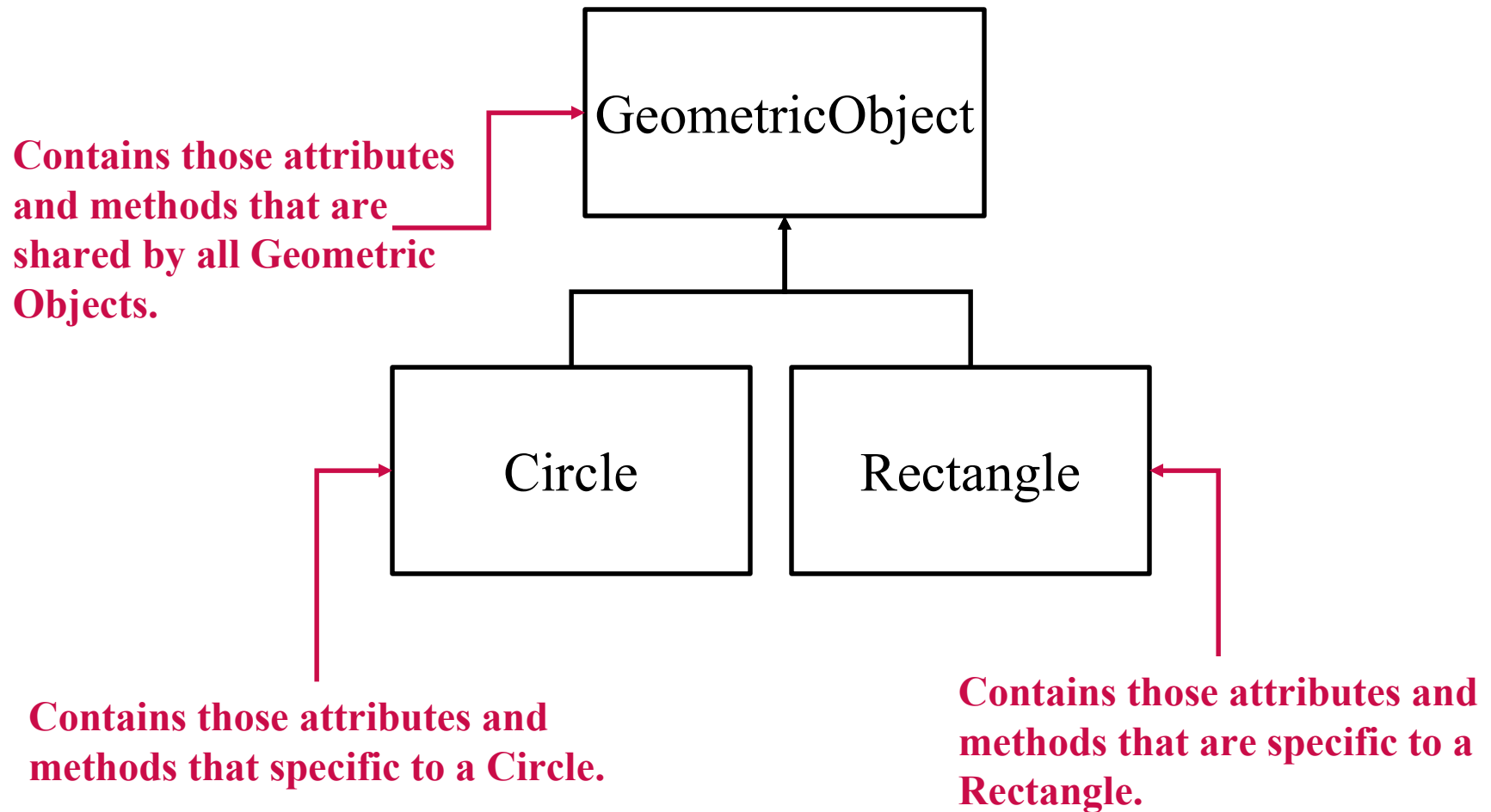
**11.14** To automatically generate boilerplate code using Lombok (§11.16).

# What is Inheritance?

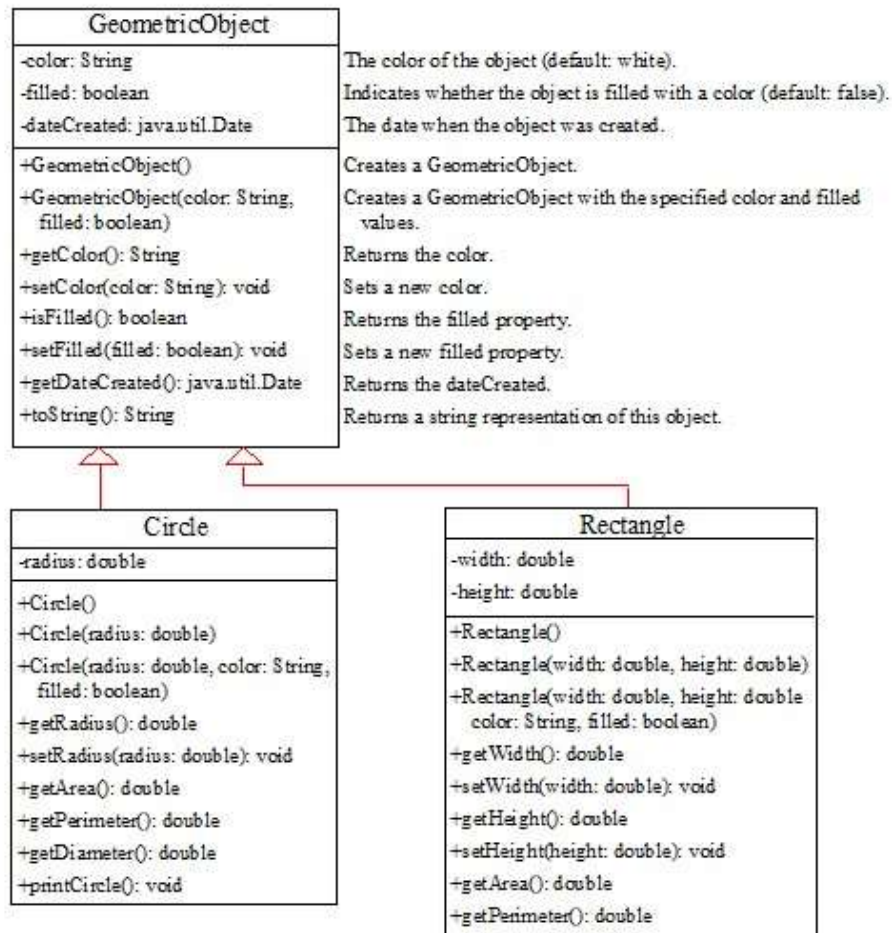
## Generalization vs. Specialization

- Real-life objects are typically specialized versions of other more general objects.
- The term “Geometric Objects” describes a very general type of Geometric Objects with numerous characteristics.
- Circles and Rectangles are Geometric Objects
  - They share the general characteristics of an Geometric Objects (color, filled, date created) .
  - However, they have special characteristics of their own.
    - Circles have a radius
    - Rectangles have height and width.
- Circles and Rectangles are specialized versions of an Geometric Objects .

# Inheritance



# Superclasses and Subclasses



[GeometricObject](#)

[Circle](#)

[Rectangle](#)

[TestCircleRectangle](#)

# Are Superclass's Constructor Inherited?

No. They are not inherited.

They are invoked explicitly or implicitly.

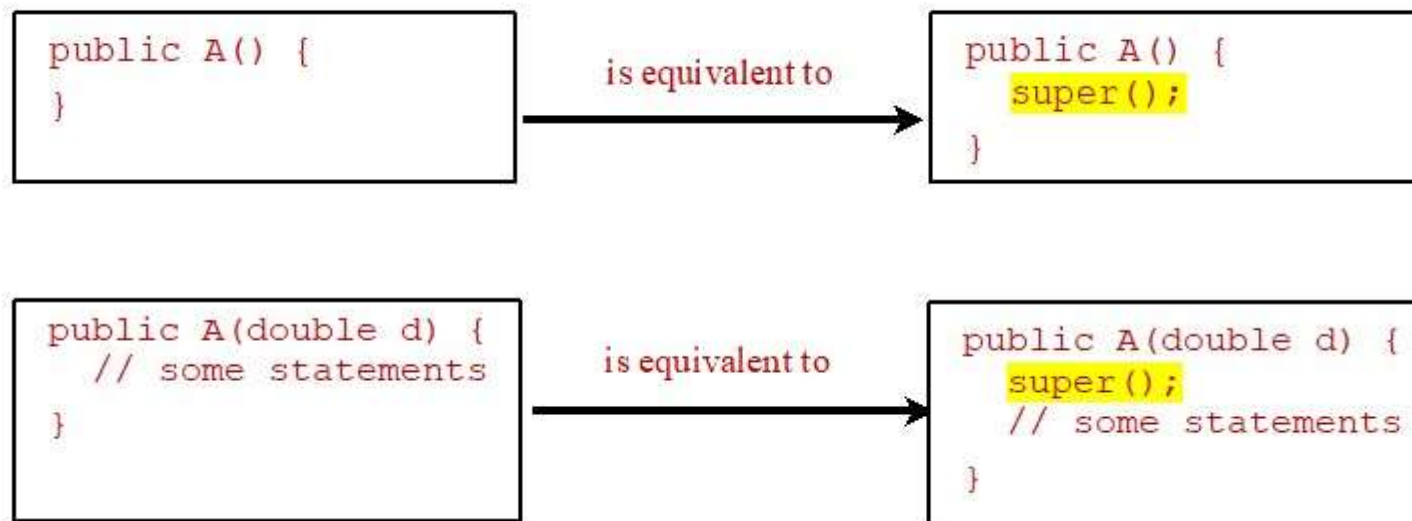
Explicitly using the super keyword.

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.

# Constructor Chaining (1 of 2)

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() {
```

# Constructor Chaining (2 of 2)

```
        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 (1 of 9)

```
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 (2 of 9)

```
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 (3 of 9)

```
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 (4 of 9)

```
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 (5 of 9)

```
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 (6 of 9)

```
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 (7 of 9)

```
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 (8 of 9)

```
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 (9 of 9)

```
public class Faculty extends Employee {  
    public static void main(String[] args) {  
        new Faculty();  
    }  
  
    public Faculty() {  
        System.out.println("(4) Faculty's no-arg constructor is invoked");  
    }  
}  
  
class Employee extends Person {  
    public Employee() {  
        this("(2) Invoke Employee's overloaded constructor");  
        System.out.println("(3) Employee's no-arg constructor is invoked");  
    }  
  
    public Employee(String s) {  
        System.out.println(s);  
    }  
}  
  
class Person {  
    public Person() {  
        System.out.println("(1) Person's no-arg constructor is invoked");  
    }  
}
```

9. Execute println

# Example on the Impact of a Superclass Without no-arg Constructor

Find out the errors in the program:

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

# 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);  
}
```

# Check Point

What is the output of running the class C in (a)?

```
class A {  
    public A() {  
        System.out.println(  
            "A's no-arg constructor is invoked");  
    }  
}  
  
class B extends A {  
}  
  
public class C {  
    public static void main(String[] args) {  
        B b = new B();  
    }  
}
```

a

What problem arises in compiling the program in (b)?

```
class A {  
    public A(int x) {  
    }  
}  
class B extends A {  
    public B() {  
    }  
}  
public class C {  
    public static void main(String[] args) {  
        B b = new B();  
    }  
}
```

b

# 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 (1 of 4)

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 (2 of 4)

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);  
    }  
}
```

```
public class Circle {  
    private double radius;  
  
    public Circle(double radius) {  
        radius = radius;  
    }  
  
    public double getRadius() {  
        return radius;  
    }  
  
    public double getArea() {  
        return radius * radius * Math.PI;  
    }  
}
```

```
class B extends Circle {  
    private double length;  
  
    B(double radius, double length)  
    {  
        Circle(radius);  
        length = length;  
    }  
  
    @Override  
    public double getArea() {  
        return getArea() * length;  
    }  
}
```

# Check Point

- If a method in a subclass has the same signature as a method in its superclass with the same return type, is the method overridden or overloaded?
- If a method in a subclass has the same signature as a method in its superclass with a different return type, will this be a problem?
- If a method in a subclass has the same name as a method in its superclass with different parameter types, is the method overridden or overloaded?



# 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](#)

# Polymorphism, Dynamic Binding and Generic Programming (1 of 2)

```
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";  
    }  
}
```

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

## DynamicBindingDemo

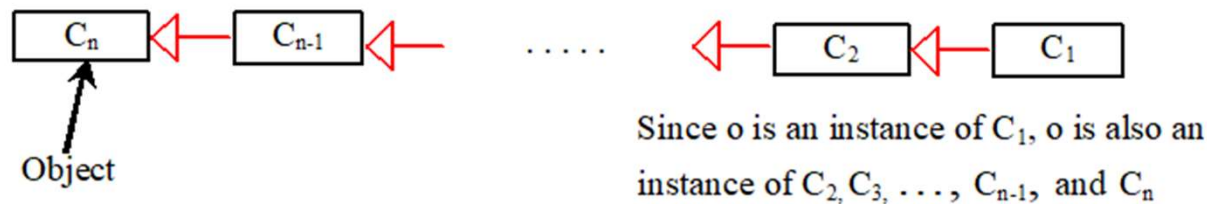
# Polymorphism, Dynamic Binding and Generic Programming (2 of 2)

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**.

# 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

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.

# Check Point

➤ What is wrong in the following code

```
public class Test {  
    public static void main(String[] args) {  
        Integer[] list1 = {12, 24, 55, 1};  
        Double[] list2 = {12.4, 24.0, 55.2, 1.0};  
        int[] list3 = {1, 2, 3};  
        printArray(list1);  
        printArray(list2);  
        printArray(list3);  
    }  
  
    public static void printArray(Object[] list) {  
        for (Object o: list)  
            System.out.print(o + " ");  
        System.out.println();  
    }  
}
```



### What is the output of the following code:

```
public class Test {  
    public static void main(String[] args) {  
        new Person().printPerson();  
        new Student().printPerson();  
    }  
}  
class Student extends Person {  
    @Override  
    public String getInfo() {  
        return "Student";  
    }  
}  
class Person {  
    public String getInfo() {  
        return "Person";  
    }  
    public void printPerson() {  
        System.out.println(getInfo());  
    }  
}
```

a

```
public class Test {  
    public static void main(String[] args) {  
        new Person().printPerson();  
        new Student().printPerson();  
    }  
}  
class Student extends Person {  
    private String getInfo() {  
        return "Student";  
    }  
}  
class Person {  
    private String getInfo() {  
        return "Person";  
    }  
    public void printPerson() {  
        System.out.println(getInfo());  
    }  
}
```

b

# Generic Programming (1 of 2)

```
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.toSt
ring());
    }
}
```

```
class GraduateStudent extends
Student {
}
class Student extends Person {
    public String toString() {
        return "Student";
    }
}
class Person extends Object {
    public String toString() {
        return "Person";
    }
}
```

# Generic Programming (2 of 2)

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());  
    ...  
}
```

# Java 16 instanceof Pattern Matching

When using the `if` statement to test if an object is an instance of a class, you can specify a **binding variable**. If the result of the `instanceof` operator is true, then the object being tested is assigned to the binding variable. This new syntax, known as **`instanceof` pattern matching**, became a standard feature since Java 16. You can simplify the code in lines 15-24 using this new feature as follows:



# Java 16 New Features on instanceof

```
if (myObject instanceof Circle circle) {  
    System.out.println("The circle area is " +  
        circle.getArea());  
    System.out.println("The circle diameter is " +  
        circle.getDiameter());  
}  
else if (myObject instanceof Rectangle rectangle) {  
    System.out.println("The rectangle area is " +  
        rectangle.getArea());  
}
```

## 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](#)

# 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 (3 of 4)

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

# Differences and Similarities Between Arrays and ArrayList

Operation	Array	ArrayList
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>

## DistinctNumbers



# 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.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<>(Arrays.asList(array));  
java.util.Collections.shuffle(list);  
System.out.println(list);
```

# Stack Animation

<https://liveexample.pearsoncmg.com/dsanimation/StackBook.html>

Stack Animation by Y. Daniel Liang

Enter a value and click the Push button to push the value into the stack. Click the Pop button to remove the top element from the stack.

Top →

5
3
3
4

Enter a value:

# The MyStack Classes

A stack to hold objects.

## MyStack

MyStack	
-list: ArrayList	A list to store elements.
+isEmpty(): boolean	Returns true if this stack is empty.
+getSize(): int	Returns the number of elements in this stack.
+peek(): Object	Returns the top element in this stack.
+pop(): Object	Returns and removes the top element in this stack.
+push(o: Object): void	Adds a new element to the top of this stack.
+search(o: Object): int	Returns the position of the first element in the stack from the top that matches the specified element.

# The protected Modifier

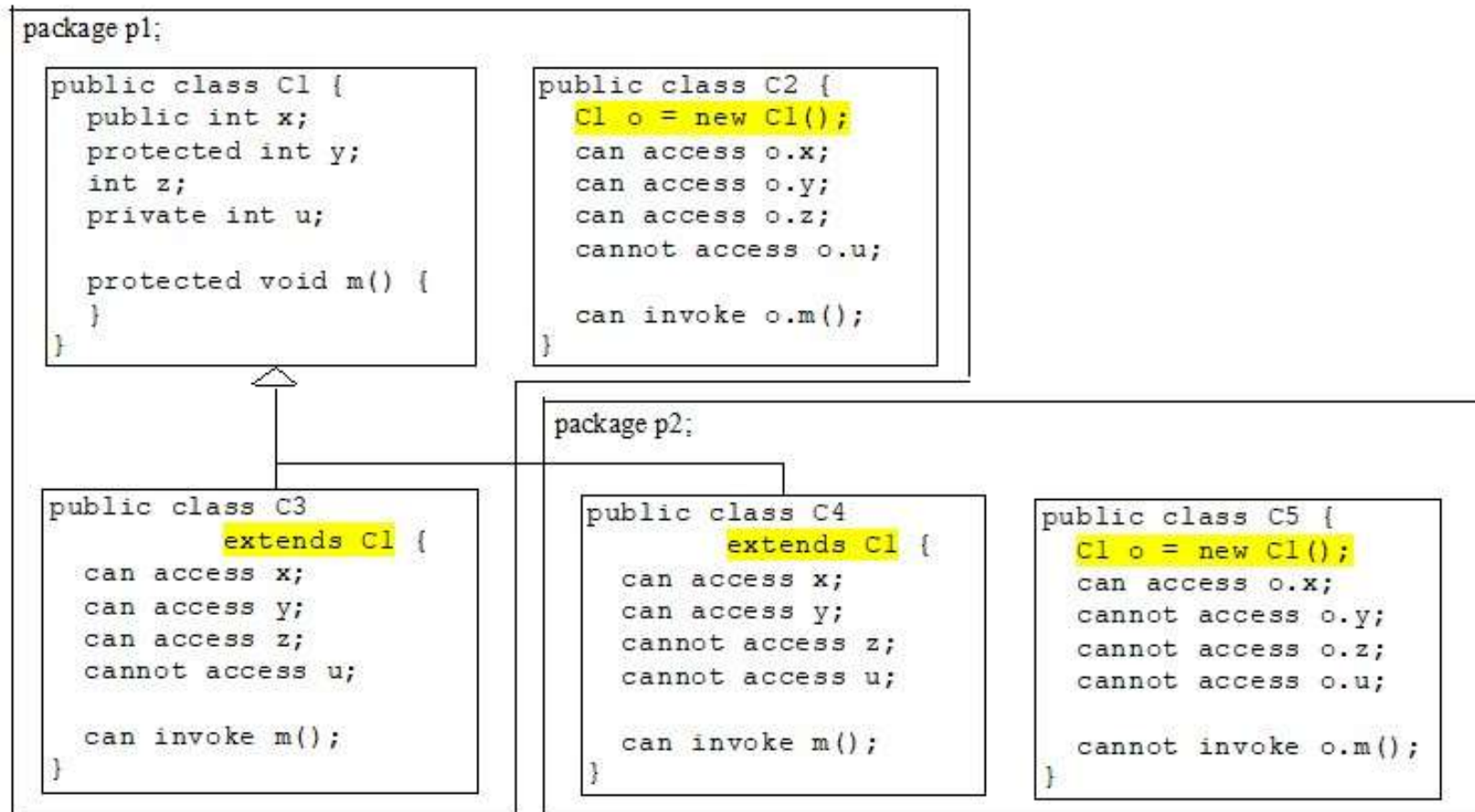
- The `protected` modifier can be applied on data and methods in a class. A protected data or a protected method in a public class can be accessed by any class in the same package or its subclasses, even if the subclasses are in a different package.
- `private`, `default`, `protected`, `public`

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

# Accessibility Summary

<b>Modifier on members in a class</b>	<b>Accessed from the same class</b>	<b>Accessed from the same package</b>	<b>Accessed from a subclass</b>	<b>Accessed from a different package</b>
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 (4 of 4)

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.