

# 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.
- Object-oriented programming allows you to define new classes from existing classes. This is called *inheritance*.



# Objectives

- ◆ To define a subclass from a superclass through inheritance.
- ◆ To invoke the superclass's constructors and methods using the **super** keyword.
- ◆ To override instance methods in the subclass.
- ◆ To distinguish differences between overriding and overloading.
- ◆ To explore the **toString()** method in the **Object** class.
- ◆ To discover polymorphism and dynamic binding .
- ◆ To describe casting and explain why explicit casting is necessary.
- ◆ To explore the **equals** method in the **Object** class.
- ◆ To store, retrieve, and manipulate objects in an **ArrayList**.
- ◆ To enable data and methods in a superclass accessible from subclasses using the **protected** visibility modifier.
- ◆ To prevent class extending and method overriding using the **final** modifier.



# Superclasses and Subclasses

Inheritance enables you to define a general class (i.e., a superclass) and later extend it to more specialized classes (i.e., subclasses).



# Superclasses and Subclasses

- Different classes may have some common properties and behaviors, which can be generalized in a class that can be shared by other classes.
- You can define a specialized class that extends the generalized class.
- The specialized classes inherit the properties and methods from the general class.



# Superclasses and Subclasses

GeometricObject	
-color: String	The color of the object (default: white).
-filled: boolean	Indicates whether the object is filled with a color (default: false).
-dateCreated: java.util.Date	The date when the object was created.
+GeometricObject()	Creates a GeometricObject.
+GeometricObject(color: String, filled: boolean)	Creates a GeometricObject with the specified color and filled values.
+getColor(): String	Returns the color.
+setColor(color: String): void	Sets a new color.
+isFilled(): boolean	Returns the filled property.
+setFilled(filled: boolean): void	Sets a new filled property.
+getDateCreated(): java.util.Date	Returns the dateCreated.
+toString(): String	Returns a string representation of this object.

A triangular arrow pointing to the superclass is used to denote the inheritance relationship between the two classes involved

Circle	
-radius: double	
+Circle()	
+Circle(radius: double)	
+Circle(radius: double, color: String, filled: boolean)	
+getRadius(): double	
+setRadius(radius: double): void	
+getArea(): double	
+getPerimeter(): double	
+getDiameter(): double	
+printCircle(): void	

Rectangle	
-width: double	
-height: double	
+Rectangle()	
+Rectangle(width: double, height: double)	
+Rectangle(width: double, height: double, color: String, filled: boolean)	
+getWidth(): double	
+setWidth(width: double): void	
+getHeight(): double	
+setHeight(height: double): void	
+getArea(): double	
+getPerimeter(): double	

A subclass inherits accessible data fields and methods from its superclass and may also add new data fields and methods.

# Superclasses and Subclasses

Syntax:

Subclass                      Superclass  
    ↓                              ↓  
`public class Circle extends GeometricObject`

Subclass                      Superclass  
    ↓                              ↓  
`public class Rectangle extends GeometricObject`

GeometricObject

CircleFromSimpleGeometricObject

RectangleFromSimpleGeometricObject

TestCircleRectangle

Run

# Using the Keyword `super`

The keyword `super` refers to the superclass of the class in which `super` appears. This keyword can be used in two ways:

- To call a superclass constructor
- To call a superclass method





# Calling Superclass Constructors

- 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.



# Calling Superclass Constructors

The syntax to call a superclass's constructor is:

**super()**, or **super(parameters)**;

The statement **super()** invokes the no-arg constructor of its superclass, and the statement **super(arguments)** invokes the superclass constructor that matches the **arguments**.



# Calling Superclass Constructors

The statement **super()** or **super(arguments)** must be the first statement of the subclass's constructor; this is the only way to explicitly invoke a superclass constructor.

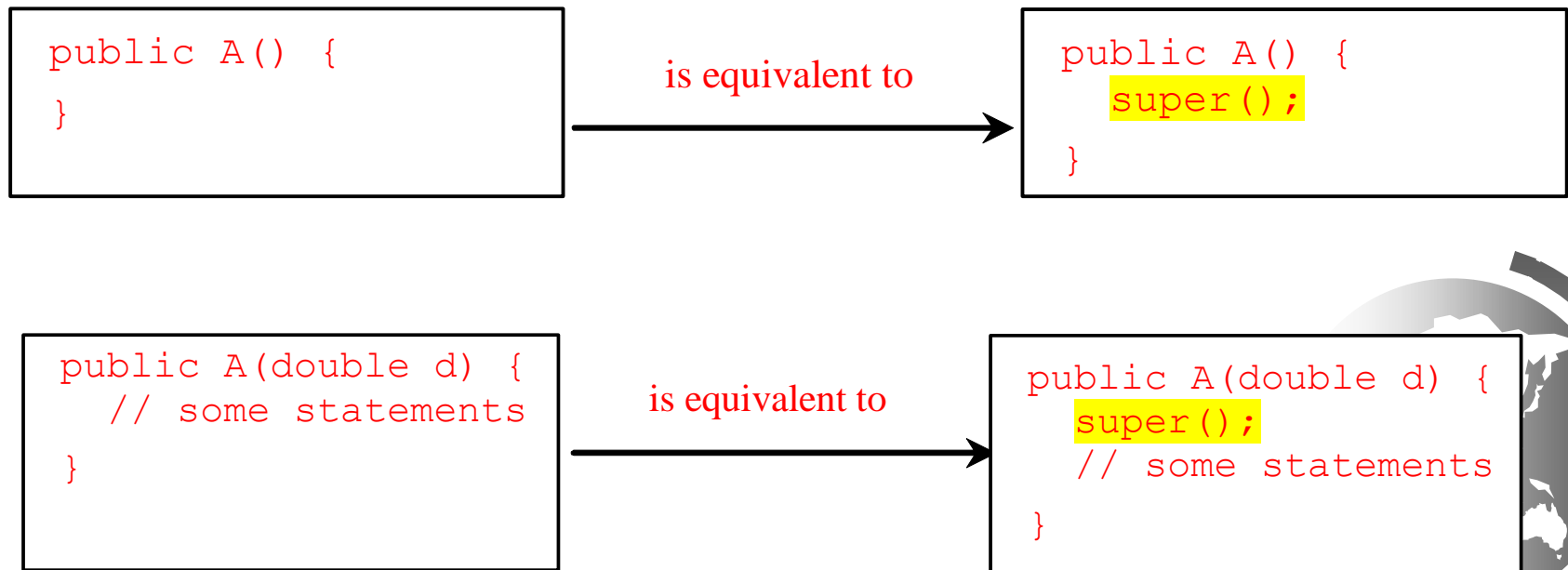
Example:

```
public CircleFromSimpleGeometricObject(  
    double radius, String color, boolean filled) {  
    super(color, filled);  
    this.radius = radius;  
}
```



# Superclass's Constructor Is Always Invoked

A constructor may invoke an overloaded constructor or its superclass's constructor. If none of them is invoked explicitly, the compiler puts super() as the first statement in the constructor. For example,



# CAUTION

- You must use the keyword super to call the superclass constructor.
- Invoking a superclass constructor's name in a subclass causes a syntax error.
- Java requires that the statement that uses the keyword super appear first in the constructor.



# Constructor Chaining

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



# Constructor Chaining

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

```
public class Faculty extends Employee {
    public static void main(String[] args) {
        new Faculty();
    }

    public Faculty() {
        System.out.println("(4) Faculty's no-arg constructor is invoked");
    }
}

class Employee extends Person {
    public Employee() {
        this("(2) Invoke Employee's overloaded constructor");
        System.out.println("(3) Employee's no-arg constructor is invoked");
    }

    public Employee(String s) {
        System.out.println(s);
    }
}

class Person {
    public Person() {
        System.out.println("(1) Person's no-arg constructor is invoked");
    }
}
```



# Trace Execution

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

1. Start from the  
main method





# Trace Execution

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

2. Invoke Faculty  
constructor



# Trace Execution

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

3. Invoke Employee's no-arg constructor



# Trace Execution

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

4. Invoke Employee(String)  
constructor



# Trace Execution

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

5. Invoke Person() constructor

# Trace Execution

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

6. Execute println

# Trace Execution

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

7. Execute println

# Trace Execution

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



8. Execute println

# Trace Execution

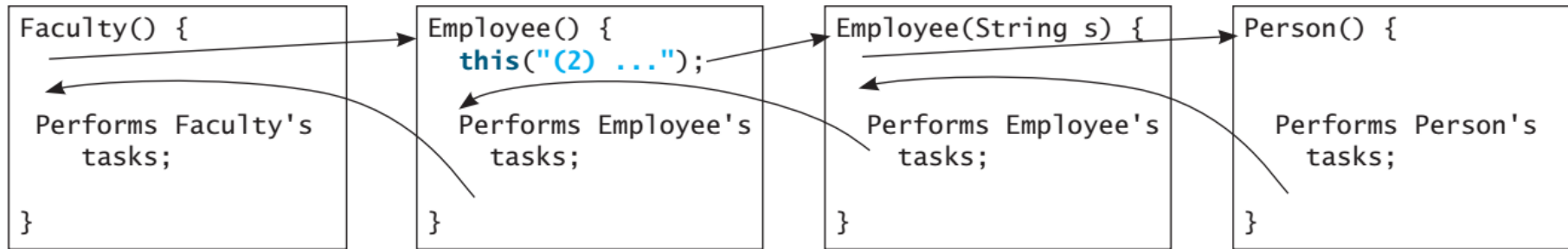
```
public class Faculty extends Employee {  
    public static void main(String[] args) {  
        new Faculty();  
    }  
  
    public Faculty() {  
        System.out.println("(4) Faculty's no-arg constructor is invoked");  
    }  
}  
  
class Employee extends Person {  
    public Employee()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

9. Execute println





# Constructor Chaining (cont.)



## Output:

- (1) Performs Person's tasks
- (2) Invoke Employee's overloaded constructor
- (3) Performs Employee's tasks
- (4) Performs Faculty's tasks

# 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");  
    }  
}
```

Since no constructor is explicitly defined in **Apple**, **Apple**'s default no-arg constructor is defined implicitly. Since **Apple** is a subclass of **Fruit**, **Apple**'s default constructor automatically invokes **Fruit**'s no-arg constructor. However, **Fruit** does not have a no-arg constructor, because **Fruit** has an explicit constructor defined. Therefore, the program cannot be compiled.

# Defining a Subclass

A subclass inherits from a superclass. You can also:

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



# Calling Superclass Methods

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

**super.method(parameters);**

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*.
- *To override a method, the method must be defined in the subclass using the same signature and the same return type as in its superclass.*



# Overriding Methods in the Superclass

```
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;  
    }  
}
```



# Overriding vs. Overloading

- Overloading means to define multiple methods with the same name but different signatures.
- Overriding means to provide a new implementation for a method in the subclass.



# Overriding vs. Overloading

```
public class Test {  
    public static void main(String[] args) {  
        A a = new A();  
        a.p(10);  
        a.p(10.0);  
    }  
}  
  
class B {  
    public void p(double i) {  
        System.out.println(i * 2);  
    }  
}  
  
class A extends B {  
    // This method overrides the method in B  
    public void p(double i) {  
        System.out.println(i);  
    }  
}
```

```
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);  
    }  
}
```



# Overriding

- To avoid mistakes, you can use a special Java syntax, called *override annotation*, to place **@Override** before the method in the subclass. For example:

```
1 public class CircleFromSimpleGeometricObject
2     extends SimpleGeometricObject {
3     // Other methods are omitted
4
5     @Override
6     public String toString() {
7         return super.toString() + "\nradius is " + radius;
8     }
9 }
```

- If a method with this annotation does not override its superclass's method, the compiler will report an error.



# NOTE

- An instance method can be overridden only if it is accessible.
- Thus a *private* method cannot be overridden, because it is not accessible outside its own class.
- If a method defined in a subclass is private in its superclass, the two methods are completely unrelated.



# NOTE

- Like an instance method, a static method can be inherited.
- However, a static method cannot be overridden.
- If a static method defined in the superclass is redefined in a subclass, the method defined in the superclass is hidden.



# The 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 (@), the object's memory address in hexadecimal.

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

The code displays something like **Loan@15037e5**. This message is not very helpful or informative. Usually you should override the toString method so that it returns a digestible string representation of the object.



# Exercise #1

```
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();  
    }  
}
```

Output:

A's no-arg constructor is invoked



## Exercise #2: What problem arises in compiling the following program?

```
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();  
    }  
}
```

**Compile error:** The default constructor of B attempts to invoke the default of constructor of A, but class A's default constructor is not defined.



# Exercise #3

```
public class Test {  
    public static void main(String[] args) {  
        A a = new A(3);  
    }  
}  
  
class A extends B {  
    public A(int t) {  
        System.out.println("A's constructor is invoked");  
    }  
}  
  
class B {  
    public B() {  
        System.out.println("B's constructor is invoked");  
    }  
}
```

Output: B's constructor is invoked  
A's constructor is invoked





# Polymorphism

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



# Polymorphism

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



# Polymorphism

- Every instance of a subclass is also an instance of its superclass, but not vice versa.
- For example, every circle is a geometric object, but not every geometric object is a circle.
- Therefore, you can always pass an instance of a subclass to a parameter of its superclass type



PolymorphismDemo

Run

# Dynamic Binding

- Same method can be implemented in several classes in the class relationship hierarchy. JVM decides which method to invoke at runtime.

```
Object o = new GeometricObject();  
System.out.println(o.toString());
```

- **Declared Type:** Type that declares a variable
- **Actual Type:** Type that is assigned based on the Class whose constructor is used
- JVM searches for the implementation of the method starting from lowest child method in the hierarchy till it finds an implementation.



# Dynamic Binding

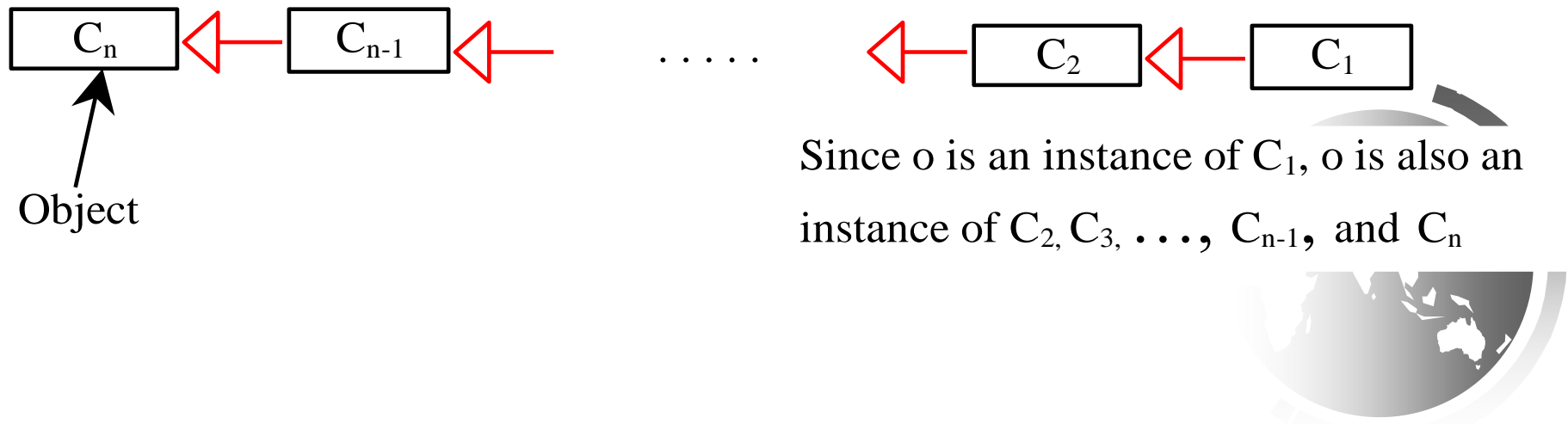
Dynamic binding works as follows:

- Suppose an object **o** is an instance of classes **C<sub>1</sub>**, **C<sub>2</sub>**, ..., **C<sub>n-1</sub>**, and **C<sub>n</sub>**, where **C<sub>1</sub>** is a subclass of **C<sub>2</sub>**, **C<sub>2</sub>** is a subclass of **C<sub>3</sub>**, ..., and **C<sub>n-1</sub>** is a subclass of **C<sub>n</sub>**.
- That is, **C<sub>n</sub>** is the most general class, and **C<sub>1</sub>** is the most specific class. In Java, **C<sub>n</sub>** is the **Object** class.



# Dynamic Binding

- If **o** invokes a method **p**, the JVM searches the implementation for the method **p** in **C<sub>1</sub>**, **C<sub>2</sub>**, ..., **C<sub>n-1</sub>** and **C<sub>n</sub>**, in this order, until it is found.
- Once an implementation is found, the search stops and the first-found implementation is invoked.



# Polymorphism and Dynamic Binding

```
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.

- 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 (JVM) at runtime. This capability is known as *dynamic binding*.



DynamicBindingDemo

Run

# 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";
    }
}
```

- **Generic programming:** Polymorphism allows methods to be used generically for a wide range of object arguments.
- 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.

# Exercise #4: Show 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)

**Person**  
**Student**

```
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)

**Person**  
**Person**



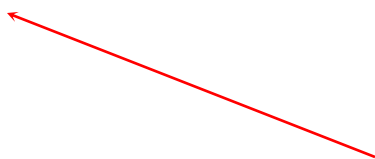
# Casting Objects

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;
```

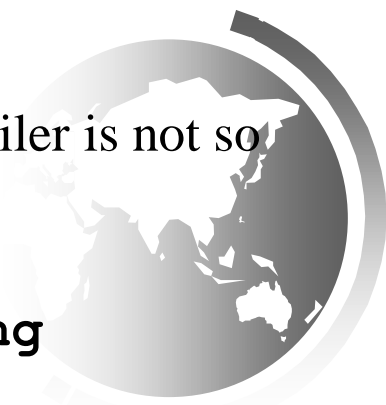
A compile error would occur.

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.

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



# Why Casting Is Necessary?

- 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
```



# Upcasting

- It is always possible to cast an instance of a subclass to a variable of a superclass (known as *upcasting*), because an instance of a subclass is *always* an instance of its superclass.



# Downcasting

- When casting an instance of a superclass to a variable of its subclass (known as *downcasting*), explicit casting must be used to confirm your intention to the compiler with the **(SubclassName)** cast notation.



# Casting

- For the casting to be successful, you must make sure that the object to be cast is an instance of the subclass.
- If the superclass object is not an instance of the subclass, a runtime ***ClassCastException*** occurs.
- For example, if an object is not an instance of **Student**, it cannot be cast into a variable of **Student**.





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



# Caution!

- ➡ The object member access operator (.) precedes the casting operator. Hence parentheses is used to ensure casting before compiler uses the . Operator

```
( (Circle) object ) . 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**.



# Exercise #5: What is wrong in the following code?

```
public class Test {  
    public static void main(String[] args) {  
        Object fruit = new Fruit();  
        Object apple = (Apple)fruit;  
    }  
}
```

```
class Apple extends Fruit {  
}
```

```
class Fruit {  
}
```

Causes a runtime  
ClassCastException



## 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



# Exercise #6:

**11.25** For the **GeometricObject** and **Circle** classes in Listings 11.1 and 11.2, answer the following questions:

a. Assume **circle** and **object** created as follows:

```
Circle circle = new Circle(1);  
GeometricObject object = new GeometricObject();
```

Are the following Boolean expressions true or false?

```
(circle instanceof GeometricObject)  
(object instanceof GeometricObject)  
(circle instanceof Circle)  
(object instanceof Circle)
```

True  
True  
True  
False

b. Can the following statements be compiled?

```
Circle circle = new Circle(5);  
GeometricObject object = circle;
```

Yes, because you can always cast from subclass to superclass.

c. Can the following statements be compiled?

```
GeometricObject object = new GeometricObject();  
Circle circle = (Circle)object;
```

Runtime exception (ClassCastException)

# 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.





# == for String

```
String obj1 = new String("xyz");  
String obj2 = new String("xyz");  
if(obj1 == obj2) → false
```

```
String obj1 = new String("xyz");  
String obj2 = obj1;  
if(obj1 == obj2) → true
```



# *equals* for String

String class overrides the method so that it checks only the values of the strings, not their locations in memory.

```
String obj1 = new String("xyz");  
String obj2 = new String("xyz");  
if(obj1.equals(obj2)) → true
```



## Exercise #7:

Show the output of running class **Test** with the **Circle** class in (a) and in (b), respectively.

```
public class Test {  
    public static void main(String[] args) {  
        Object circle1 = new Circle();  
        Object circle2 = new Circle();  
        System.out.println(circle1.equals(circle2));  
    }  
}
```

```
class Circle {  
    double radius;  
  
    public boolean equals(Circle circle) {  
        return this.radius == circle.radius;  
    }  
}
```

(a)

**False**

```
class Circle {  
    double radius;  
  
    public boolean equals(Object circle) {  
        return this.radius ==  
            ((Circle)circle).radius;  
    }  
}
```

(b)

**True**



# The ArrayList Class

- *An **ArrayList** object can be used to store a list of objects.*
- You can create an array to store objects. But, once the array is created, its size is fixed.
- Java provides the **ArrayList** class, which can be used to store an unlimited number of objects.



# The ArrayList Class

## **java.util.ArrayList<E>**

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

Creates an empty list.

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

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

Removes all the elements from this list.

Returns true if this list contains the element *o*.

Returns the element from this list at the specified index.

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

Returns true if this list contains no elements.

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

Removes the element *o* from this list.

Returns the number of elements in this list.

Removes the element at the specified index.

Sets the element at the specified index.



# Generic Type

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

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

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



TestArrayList

Run

# Differences and Similarities between Arrays and ArrayList

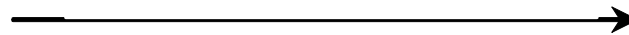
<i>Operation</i>	<i>Array</i>	<i>ArrayList</i>
Creating an array/ArrayList	<code>String[] a = new String[10]</code>	<code>ArrayList&lt;String&gt; list = new ArrayList&lt;&gt;();</code>
Accessing an element	<code>a[index]</code>	<code>list.get(index);</code>
Updating an element	<code>a[index] = "London";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



# 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

package p1;

```
public class C1 {  
    public int x;  
    protected int y;  
    int z;  
    private int u;  
  
    protected void m() {  
    }  
}
```

```
public class C2 {  
    C1 o = new C1();  
    can access o.x;  
    can access o.y;  
    can access o.z;  
    cannot access o.u;  
  
    can invoke o.m();  
}
```

public class C3  
 extends C1 {  
 can access x;  
 can access y;  
 can access z;  
 cannot access u;  
  
 can invoke m();  
}

package p2;

```
public class C4  
    extends C1 {  
    can access x;  
    can access y;  
    cannot access z;  
    cannot access u;  
  
    can invoke m();  
}
```

```
public class C5 {  
    C1 o = new C1();  
    can access o.x;  
    cannot access o.y;  
    cannot access o.z;  
    cannot access o.u;  
  
    cannot invoke o.m();  
}
```

# 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.



# 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.



# 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.



# Exercise #8:

In the following code, the classes **A** and **B** are in the same package. If the question marks in (a) are replaced by blanks, can class **B** be compiled? If the question marks are replaced by **private**, can class **B** be compiled? If the question marks are replaced by **protected**, can class **B** be compiled?

```
package p1;

public class A {
    ? int i;

    ? void m() {
        ...
    }
}
```

(a)

```
package p1;

public class B extends A {
    public void m1(String[] args) {
        System.out.println(i);
        m();
    }
}
```

(b)

Blank → Yes  
Private → No  
Protected → Yes



# Exercise #9:

In the following code, the classes **A** and **B** are in different packages. If the question marks in (a) are replaced by blanks, can class **B** be compiled? If the question marks are replaced by **private**, can class **B** be compiled? If the question marks are replaced by **protected**, can class **B** be compiled?

```
package p1;  
  
public class A {  
    ? int i;  
  
    ? void m() {  
        ...  
    }  
}
```

(a)

```
package p2;  
  
public class B extends A {  
    public void m1(String[] args) {  
        System.out.println(i);  
        m();  
    }  
}
```

(b)

Blank → No

Private → No

Protected → Yes

