CANADIAN
UNIVERSITY
OF BANGLADESH
Inspiring Applied Knowledge

School of Science & Engineering
Department of CSE
Canadian University of Bangladesh

Lecture-10: **Inheritance & Polymorphism**

Instructor: Prof. Miftahur Rahman, Ph.D.
Semester: Summer 2024

Object-Oriented Problem Solving

Inheritance & Polymorphism

Based on Chapter 11 of "Introduction to Java Programming" by Y. Daniel Liang.

Outline

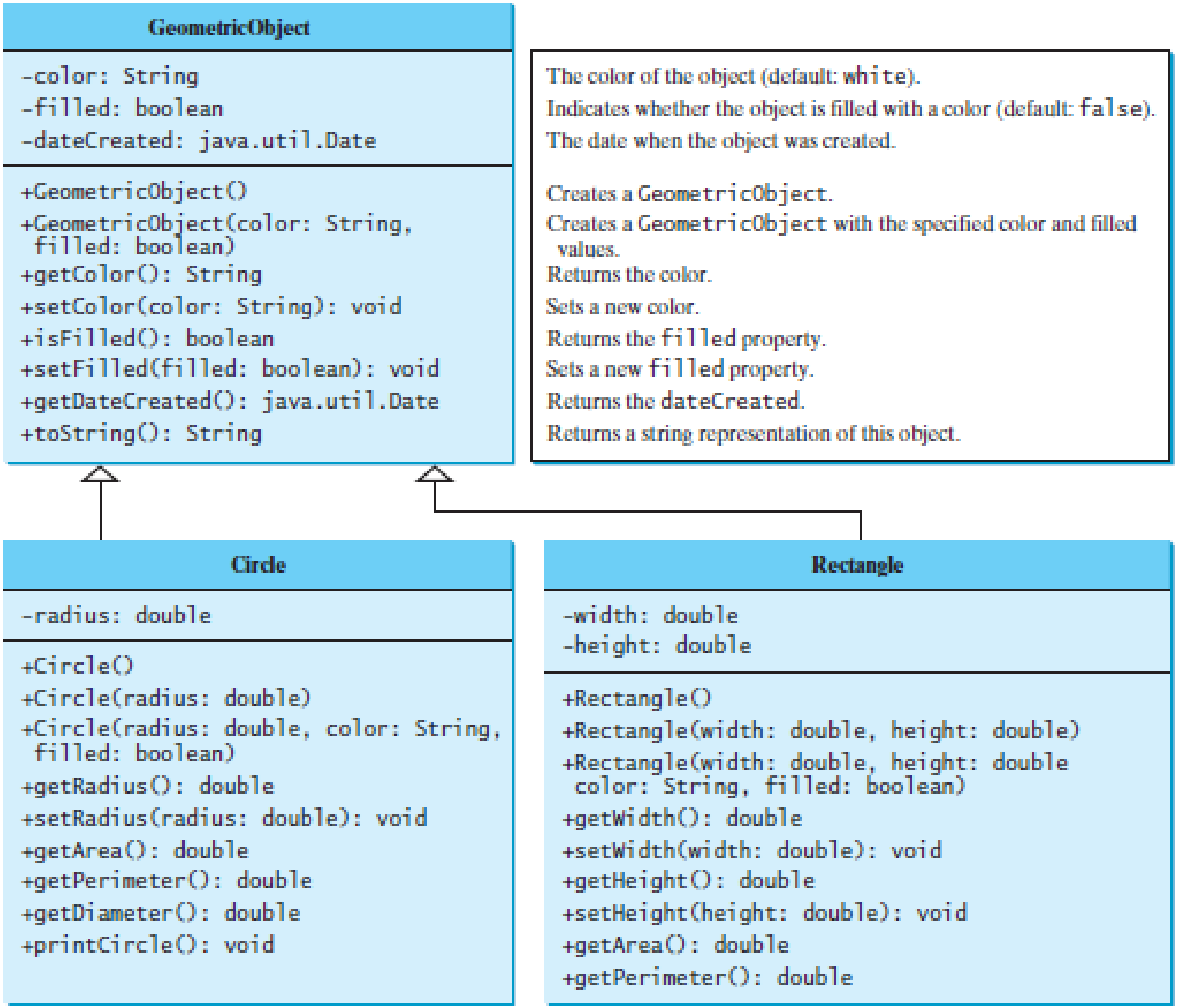
- Superclasses and Subclasses (11.2)
- Using the Super keyword (11.3)
- Overriding Methods (11.4)
- Overriding vs. Overloading (11.5)
- The Object Class and its toString() (11.6)
- Polymorphism (11.7)
- Dynamic Binding (11.8)
- Casting Objects (11.9)
- The Object’s equals Method (11.10)
- The Protected Data and Methods (11.14)
- Preventing Extending and Overriding (11.15)

Superclasses and Subclasses

- Classes are used to model objects of the same type.
- Different classes may have some common properties and behaviors.
- *Inheritance* allows you to:
  - Define a generalized class that includes the common properties and behavior.
  - Define specialized classes that extend the generalized class.
    - *Inherit* the properties and methods from the general class.
    - Add new properties and methods.

Superclasses and Subclasses (Cont.)

- In Java terminology, a class *C1* extended from another class *C2* is called a *subclass*, and *C2* is called a *superclass*.
  - A superclass is also referred to as a *parent class* or a *base class*, and a subclass as a *child class*, an *extended class*, or a *derived class*.
- A subclass:
  - inherits accessible data fields and methods from its superclass and,
  - may also add new data fields and methods.


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

GeometricObject Class

```
// LISTING 11.1 SimpleGeometricObject.java
public class SimpleGeometricObject {
private String color = "white";
private boolean filled;
private java.util.Date dateCreated;

 /** Construct a default geometric object */
public SimpleGeometricObject() {
dateCreated = new java.util.Date();
}
/** Construct a geometric object with the specified color
 * and filled value */
public SimpleGeometricObject(String color, boolean filled) {
dateCreated = new java.util.Date();
this.color = color;
this.filled = filled;
}
```

GeometricObject Class (continued)

```java
/** Return color */
public String getColor() {
return color;
}
/** Set a new color */
public void setColor(String color) {
this.color = color;

}
/** Return filled. Since filled is boolean,

 its getter method is named isFilled */
 public boolean isFilled() {
 return filled;
 }
 /** Set a new filled */
 public void setFilled(boolean filled) {
 this.filled = filled;
 }
```

GeometricObject Class (continued)

```
/** Get dateCreated */
 public java.util.Date getDateCreated() {
 return dateCreated;
 }

 /** Return a string representation of this object */
 public String toString() {
 return "created on " + dateCreated + "\ncolor: " + color + " and
filled: " + filled;
 }
}
```

Circle Class:

```
// LISTING 11.2 CircleFromSimpleGeometricObject.java
public class CircleFromSimpleGeometricObject extends SimpleGeometricObject {
    private double radius;
    // Default constructor
    public CircleFromSimpleGeometricObject() {
    }
    // Constructor with radius parameter
    public CircleFromSimpleGeometricObject(double radius) {
        this.radius = radius;
    }
    // Constructor with radius, color, and filled parameters
    public CircleFromSimpleGeometricObject(double radius, String color, boolean filled) {
        this.radius = radius;
        setColor(color);
        setFilled(filled);
    }
```

Circle Class (continued):

```
 /** Return radius */
public double getRadius() {
return radius;
 }
/** Set a new radius */
public void setRadius(double radius) {
this.radius = radius;
}
/** Return area */
public double getArea() {
return radius * radius * Math.PI;
}
/** Return diameter */
public double getDiameter() {
return 2 * radius;
}
```

Circle Class (continued):

```
/** Return perimeter */
public double getPerimeter() {
return 2 * radius * Math.PI;
}
/** Print the circle info */
public void printCircle() {
System.out.println("The circle is created " + getDateCreated() +
" and the radius is " + radius);
}
}
```

Rectangle Class

```
// LISTING 11.3 RectangleFromSimpleGeometricObject.java
public class RectangleFromSimpleGeometricObject
extends SimpleGeometricObject {
private double width;
private double height;
public RectangleFromSimpleGeometricObject() {
}
public RectangleFromSimpleGeometricObject(
double width, double height) {
this.width = width;
this.height = height;
}
public RectangleFromSimpleGeometricObject(
double width, double height, String color, boolean filled) {
this.width = width;
this.height = height;
setColor(color);
setFilled(filled);
}
```

Rectangle Class (continued)

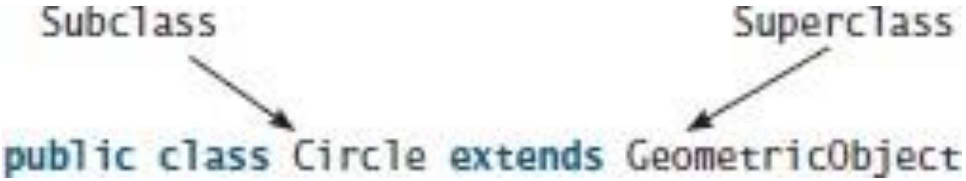
```
/** Return width */
public double getWidth() {
return width;
 }
/** Set a new width */
public void setWidth(double width) {
this.width = width;
 }
/** Return height */
public double getHeight() {
return height;
}
/** Set a new height */
public void setHeight(double height) {
this.height = height;
 }
```

Rectangle Class (continued)

```
/** Return area */
public double getArea() {
return width * height;
 }
/** Return perimeter */
public double getPerimeter() {
return 2 * (width + height);
 }
 }
```

Comments

- The *Circle* class extends the *GeometricObject* using the following syntax:

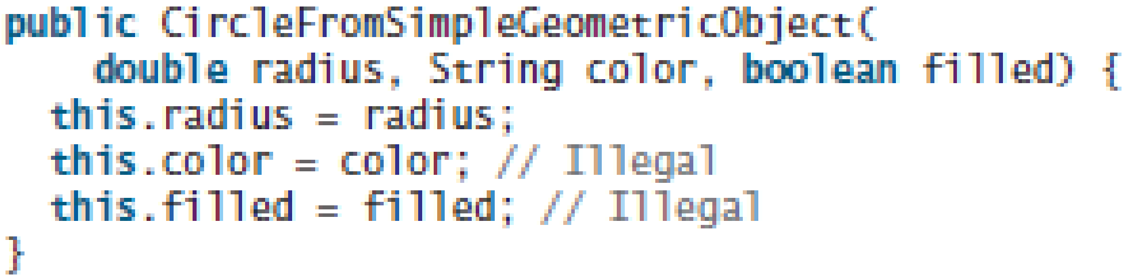


```
public class Circle extends GeometricObject
```

- The keyword *extends* tells the compiler that the *Circle* class extends the *GeometricObject* class, thus inheriting the methods *getColor*, *setColor*, *isFilled*, *setFilled*, and *toString*.
- The overloaded constructor *Circle(double radius, String color, boolean filled)* is implemented by invoking the *setColor* and *setFilled* methods to set the *color* and *filled* properties.
  - These two public methods are defined in the superclass *GeometricObject* and are inherited in *Circle*, so they can be used in the *Circle* class.

Comments (Cont.)

- You might attempt to use the data fields *color* and *filled* directly in the constructor as follows:



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

- This is wrong, because the private data fields *color* and *filled* in the *GeometricObject* class cannot be accessed in any class other than in the *GeometricObject* class itself.
  - The only way to read and modify *color* and *filled* is through their getter and setter methods.

// LISTING 11.4 TestCircleRectangle.java

```java
public class TestCircleRectangle {
public static void main(String[] args) {
CircleFromSimpleGeometricObject circle = new CircleFromSimpleGeometricObject(1);
System.out.println("A circle " + circle.toString());
System.out.println("The color is " + circle.getColor());
System.out.println("The radius is " + circle.getRadius());
System.out.println("The area is " + circle.getArea());
System.out.println("The diameter is " + circle.getDiameter());
RectangleFromSimpleGeometricObject rectangle = new
RectangleFromSimpleGeometricObject(2, 4);
System.out.println("\nA rectangle " + rectangle.toString());
System.out.println("The area is " + rectangle.getArea());
System.out.println("The perimeter is " +
rectangle.getPerimeter());

 }
}
```

Important Notes Regarding Inheritance (1)

- Contrary to conventional interpretation, a subclass is not a subset of its superclass.
  - In fact, a subclass usually contains more information and methods than its superclass.
- Private data fields in a superclass are not accessible outside the class.
  - They cannot be used directly in a subclass.
  - They can only be accessed/mutated through public accessors/mutators if defined in the superclass.

Important Notes Regarding Inheritance (2)

- Inheritance is used to model the *is-a* relationship.
  - Do not blindly extend a class just for the sake of reusing methods.
  - For example, it makes no sense for a *Tree* class to extend a *Person* class, even though they share common properties such as height and weight.
- Some programming languages allow you to derive a subclass from several classes.
  - This capability is called *multiple inheritance*.
  - Java does not allow multiple inheritance.
    - A Java class may inherit directly from only one class.
  - Multiple inheritance can be achieved through interfaces in Java.

The *Super* Keyword

- The keyword *super* refers to the superclass and can be used to:
  - Call a superclass constructor.
  - Call a superclass method.

Using the *Super* Keyword to Call a Superclass Constructor

- Remember that a constructor is used to construct an instance of a class.
- Unlike properties and methods, the constructors of a superclass <u>are not inherited</u> by a subclass.
  - They can only be invoked from the constructors of the subclasses using the keyword *super*.
- The syntax to call a superclass's constructor is:
  - *super()*, or *super(arguments)*;
  - The statement *super()* invokes the no-arg constructor of its superclass.
  - The statement *super(arguments)* invokes the superclass constructor that matches the *arguments*.

Using the *Super* Keyword to Call a Superclass Constructor: Example

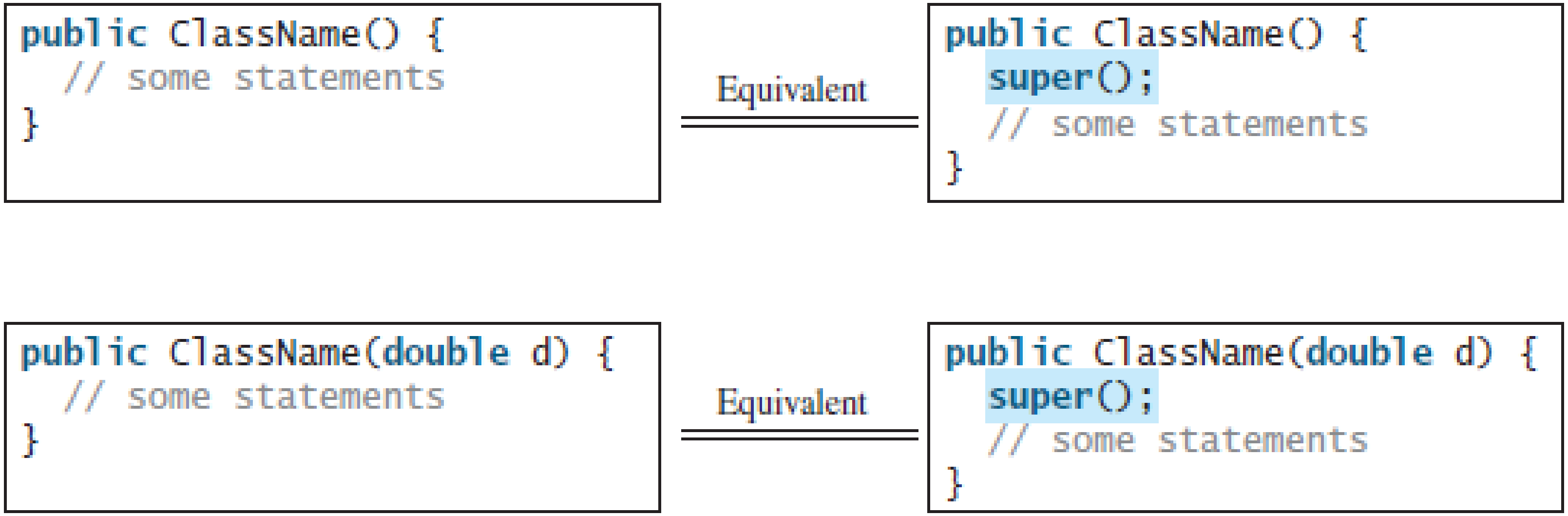
- The statement *super()* or *super(parameters)* must appear in the first line of the subclass's constructor.
- The following constructor can be added to the *Circle* class of the previous example:

```
public Circle (double radius){
      super();
      this.radius = radius;
}
```

Invokes the no-arg constructor, which is the default constructor of the GeometricObject class.

Constructor Chaining

- A constructor may invoke an overloaded constructor (using *this*) or its superclass constructor (using *super*).
- If neither is invoked explicitly, the compiler automatically puts *super()* as the first statement in the constructor.



```
public ClassName() {
  // some statements
}
```

Equivalent

```
public ClassName() {
  super();
  // some statements
}
```

```
public ClassName(double d) {
  // some statements
}
```

Equivalent

```
public ClassName(double d) {
  super();
  // some statements
}
```

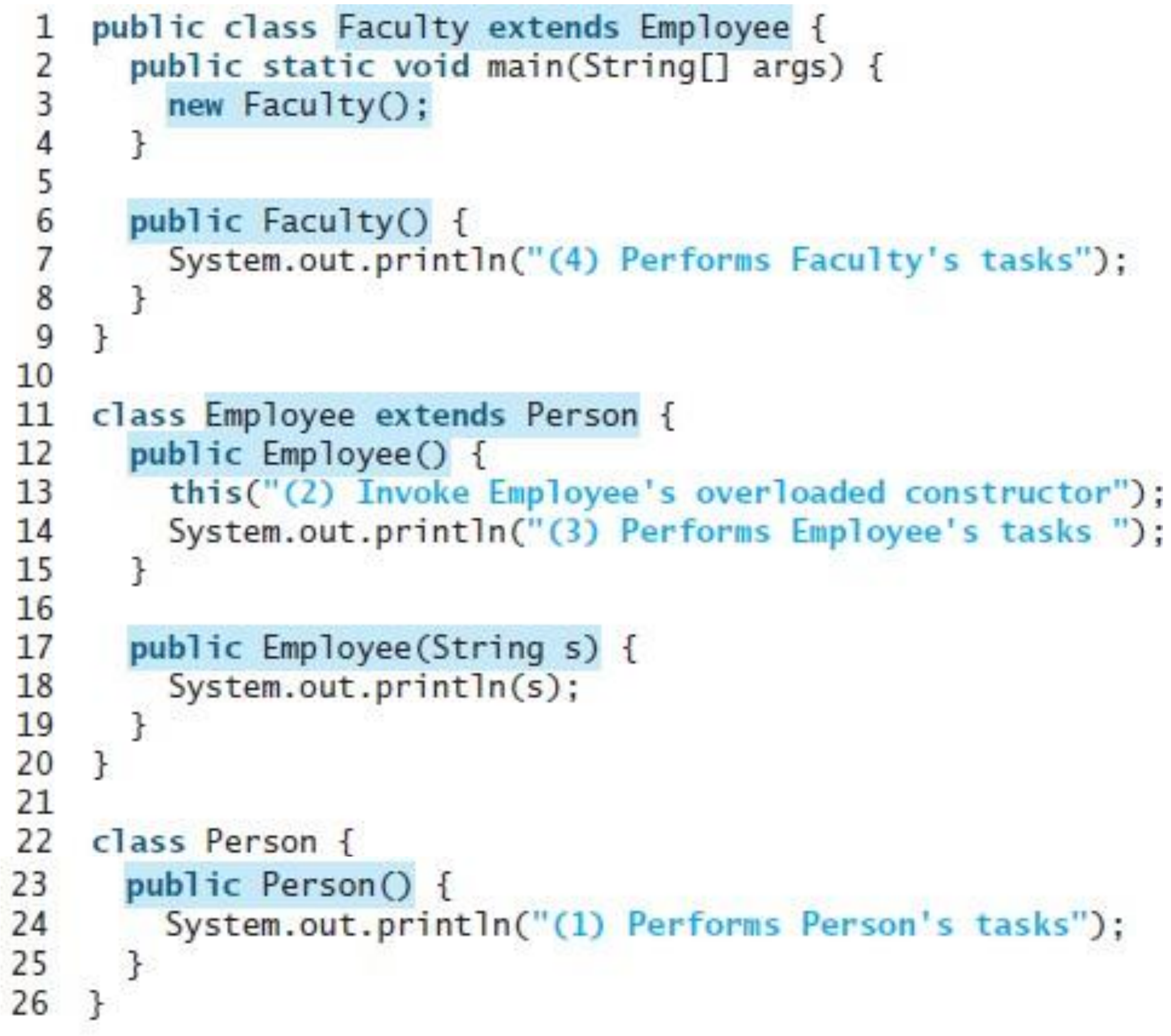
Constructor Chaining (Cont.)

- In any case, constructing an instance of a class invokes the constructors of all the superclasses along the inheritance hierarchy.

Constructor Chaining

  - When constructing an object of a subclass, the subclass constructor first invokes its superclass constructor before performing its own tasks.
  - If the superclass is derived from another class, the superclass constructor invokes its parent-class constructor before performing its tasks.
  - This process continues until the last constructor along the inheritance hierarchy is called.

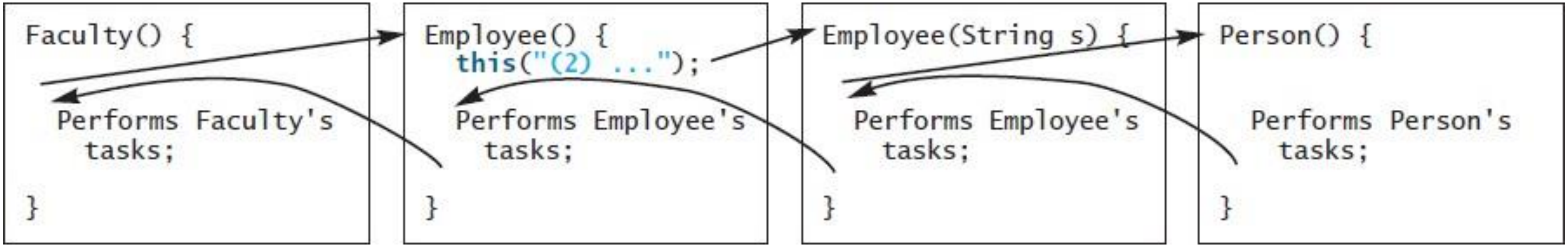
Constructor Chaining: Example



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

  public Faculty() {
    System.out.println("(4) Performs Faculty's tasks");
  }
}

class Employee extends Person {
  public Employee() {
    this("(2) Invoke Employee's overloaded constructor");
    System.out.println("(3) Performs Employee's tasks ");
  }

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

class Person {
  public Person() {
    System.out.println("(1) Performs Person's tasks");
  }
}
```

Constructor Chaining: Example (Cont.)

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



```
Faculty() {

  Performs Faculty's
    tasks;

}
```

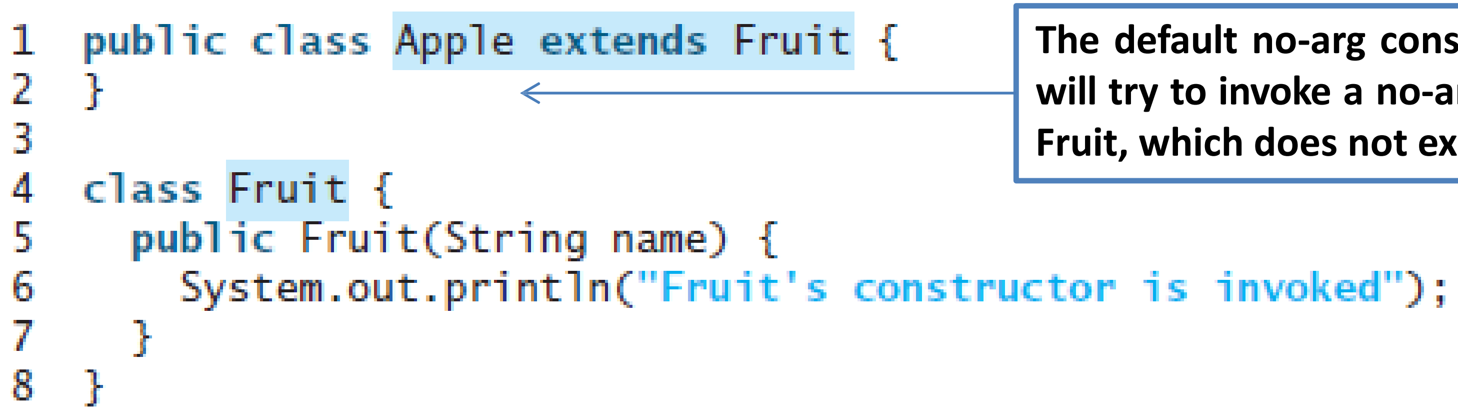
```
Employee() {
  this("(2) ...");

  Performs Employee's
    tasks;

}
```

```
Employee(String s) {

  Performs Employee's
    tasks;

}
```

```
Person() {

  Performs Person's
    tasks;

}
```

Caution!!

- If a class is designed to be extended, it is better to provide a no-arg constructor to avoid programming errors.
- Example: this code cannot be compiled:

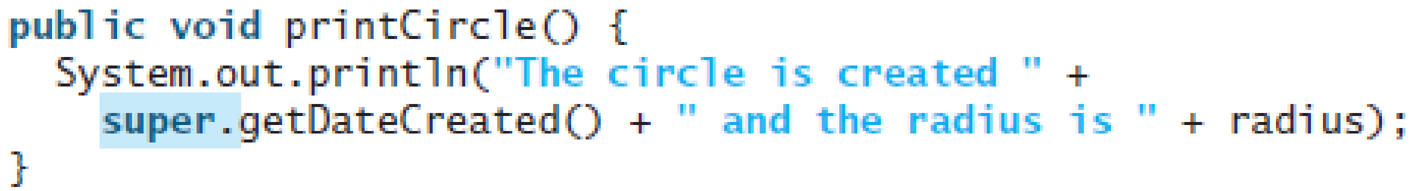


```java
public class Apple extends Fruit {
}

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

The default no-arg constructor of Apple will try to invoke a no-arg constructor of Fruit, which does not exist!

Using the *Super* Keyword to Call a Superclass Method

- The keyword *super* can 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:

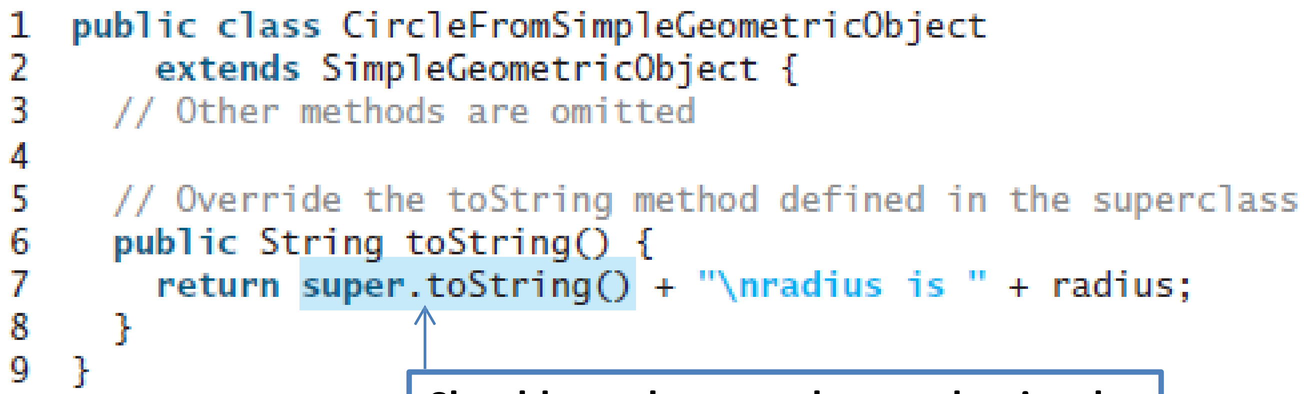


```
public void printCircle() {
  System.out.println("The circle is created " +
    super.getDateCreated() + " and the radius is " + radius);
}
```

- It is not necessary to put super before *getDateCreated()* in this case, however, because *getDateCreated* is a method in the *GeometricObject* class and is inherited by the Circle class.
  - Cases were the *super* keyword is needed to invoke the superclass methods will be showed when methods overriding is introduced.

Overriding Methods

- 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*.
- The *toString* method in the *GeometricObject* class returns the string representation of a geometric object.
- This method can be overridden to return the string representation of a circle:



```
public class CircleFromSimpleGeometricObject
    extends SimpleGeometricObject {
  // Other methods are omitted

  // Override the toString method defined in the superclass
  public String toString() {
    return super.toString() + "\nradius is " + radius;
  }
}
```

Should use the super keyword to invoke the *toSrting* method of the superclass *GeometricObject*.

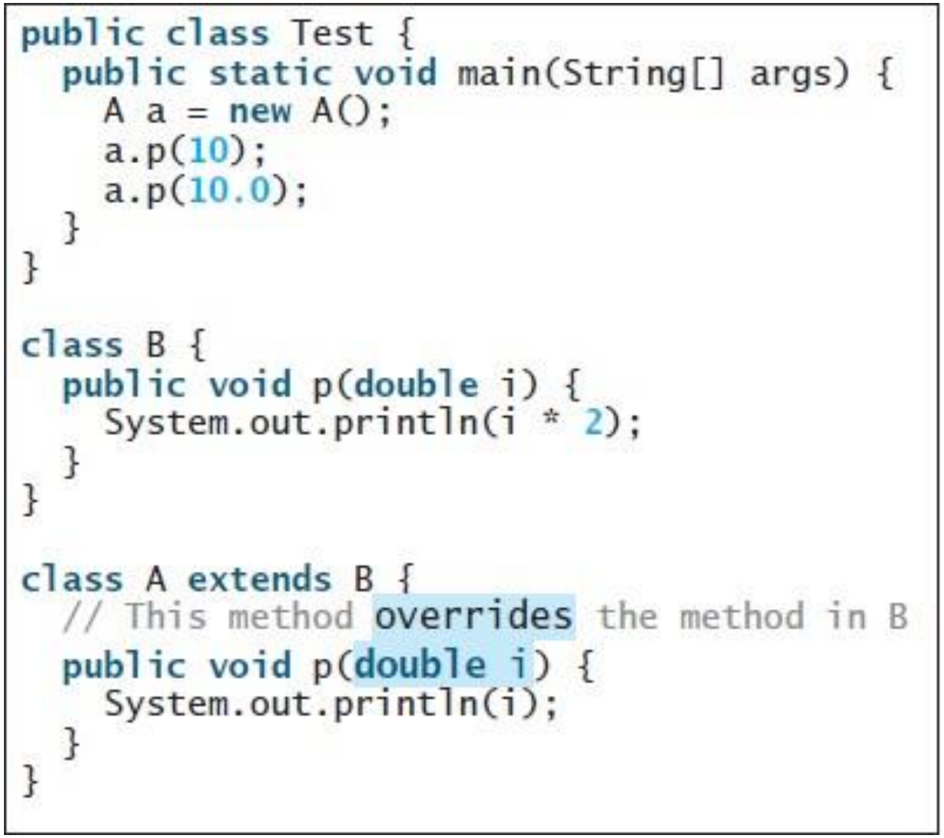
Overriding Methods (Cont.)

- 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.
- Like an instance method, a static method can be inherited. However a static method cannot be overridden.
  - If a static method defined in the superclass is redefined in a subclass, the method defined in the superclass is hidden.
  - The hidden static methods can be invoked using the syntax *SuperClassName.staticMethodName*.

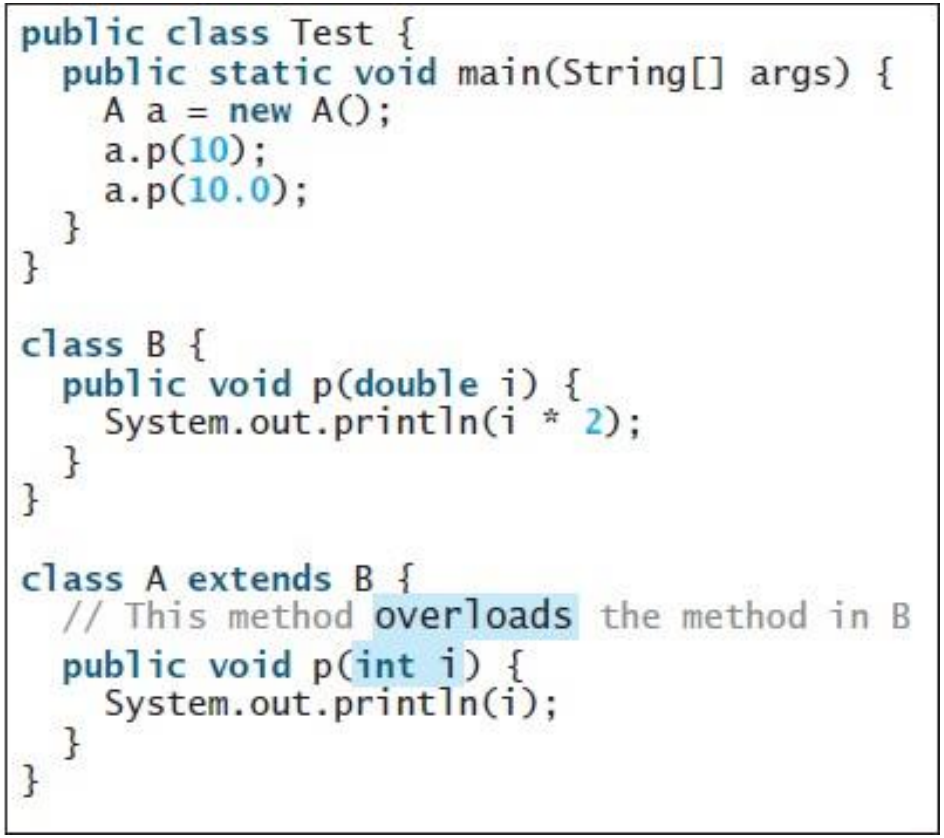
Overriding vs. Overloading

- 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.
  - The method should be defined in the subclass using the same signature and the same return type.

Overriding vs. Overloading: Example



```
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);
  }
}
```



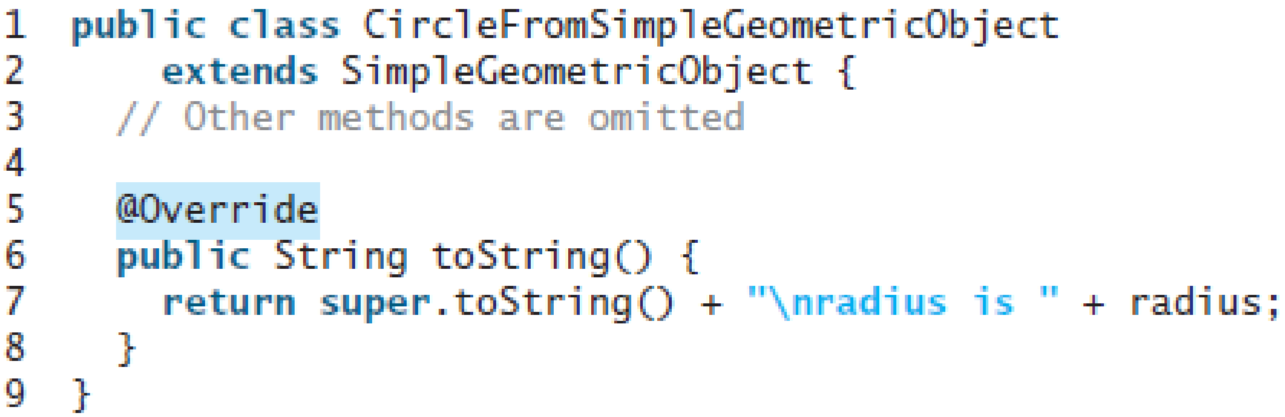
```
public class Test {
  public static void main(String[] args) {
    A a = new A();
    a.p(10);
    a.p(10.0);
  }
}

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 vs. Overloading: Notes

- Overridden methods are in different classes related by inheritance; overloaded methods can be either in the same class or different classes related by inheritance.
- Overridden methods have the same signature and return type; overloaded methods have the same name but a different parameter list.

Override Annotation

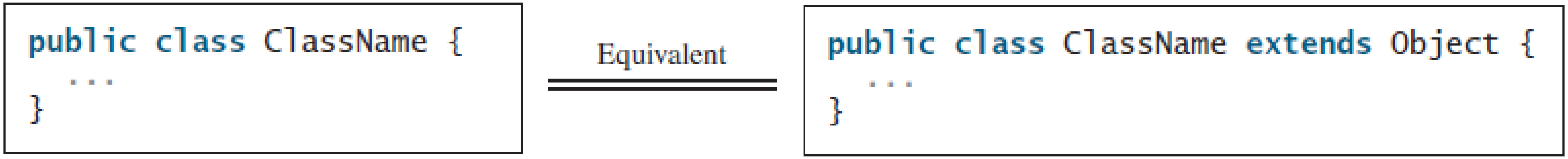
- To avoid mistakes, you can use a special Java syntax, called *override annotation*:
  - Place @Override before the method in the subclass.
- This annotation denotes that the annotated method is required to override a method in the superclass.
  - If a method with this annotation does not override its superclass's method, the compiler will report an error.
- For example, if *toString* is mistyped as *tostring*, a compile error is reported. If the override annotation isn't used, the compile won't report an error. Using annotation avoids mistakes.:



```
public class CircleFromSimpleGeometricObject
    extends SimpleGeometricObject {
  // Other methods are omitted

  @Override
  public String toString() {
    return super.toString() + "\nradius is " + radius;
  }
}
```

The Object Class and Its toString() Method

- *Every class in Java is descended from the java.lang.Object class.*
- If no inheritance is defined when a class is defined, the superclass of the class is *Object* by default.
- For example the following two class definitions are the same:



```
public class ClassName {
  ...
}
```

Equivalent

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

The Object Class and Its toString() Method (Cont.)

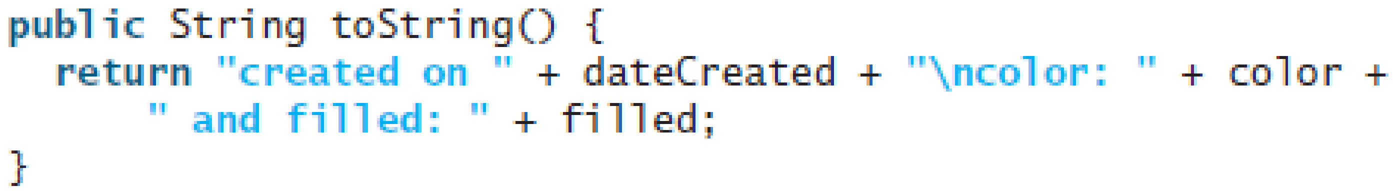
- One of the most important methods provided by the *Object* class is the method *toString*.
- The signature of the *toString* method is:
  - ***public String toString()***
- Invoking *toString()* on an object returns a string that describes the object.
  - By default, it returns a string consisting of a class name of which the object is an instance, an at sign (@), and the object's memory address in hexadecimal.

    Circle c = new Circle();

    System.out.println(c.toString());
  - For example, the output of the following code is something like: Circle@780324ff
  - This message is not very helpful or informative.
  - Usually you should override the *toString* method so that it returns a descriptive string representation of the object.

The Object Class and Its toString() Method (Cont.)

- Usually, we override the *toString* method so that it returns a descriptive string representation of the object.
- For example, the *toString* method in the *Object* class was overridden in the *GeometricObject* class as follows:



```
public String toString() {
  return "created on " + dateCreated + "\ncolor: " + color +
      " and filled: " + filled;
}
```

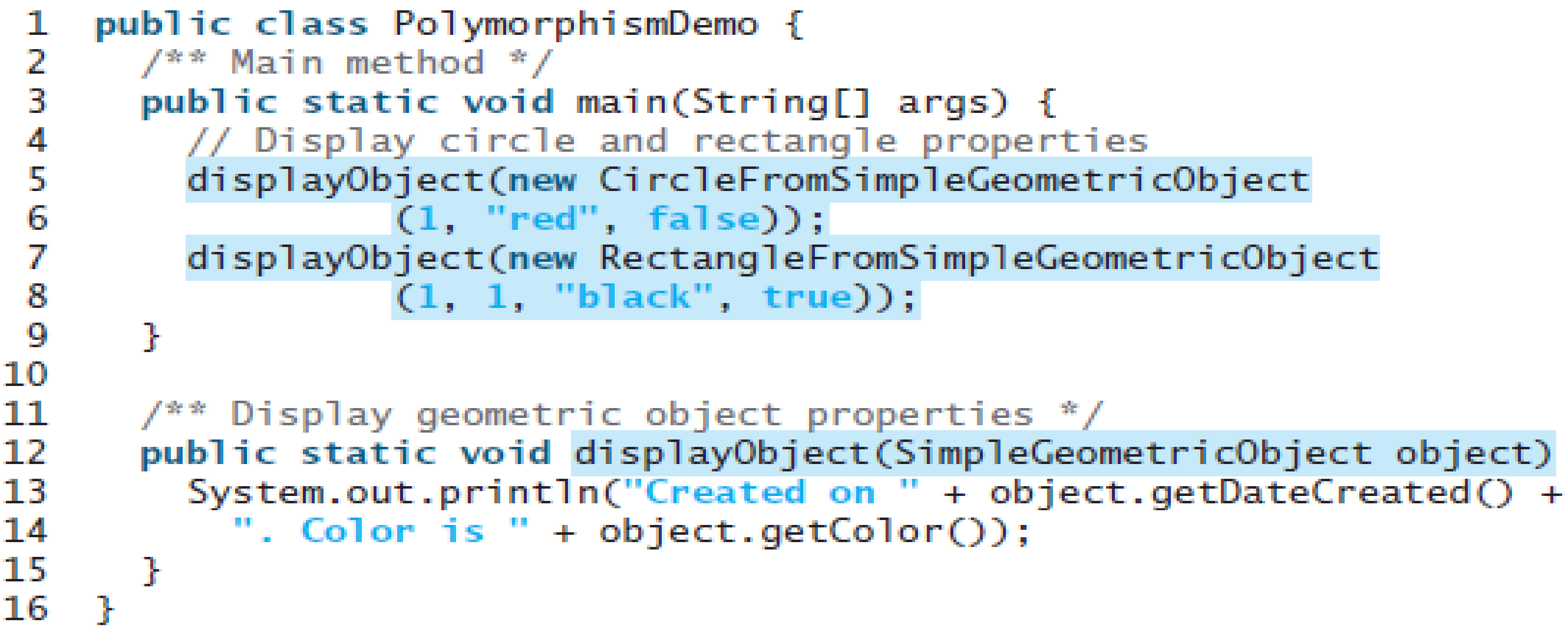
- You can also pass an object to invoke *System.out.println(object)* and *System.out.print(object)*.
  - This is equivalent to invoking *System.out.println(object.toString())* and *System.out.print(object.toString())*.

Polymorphism

- The three pillars of object-oriented programming are:
  - Encapsulation
  - Inheritance, and
  - Polymorphism.
- The inheritance relationship enables a subclass to inherit features from its superclass with additional new features.
- 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*.
- A subclass is a specialization of its superclass; 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.

Polymorphism (Cont.)

- *Polymorphism* means that a variable of a supertype can refer to a subtype object.
  - You can always pass an instance of a subclass to a parameter of its superclass type.
  - An object of a subclass can be used wherever its superclass object is used.



```java
public class PolymorphismDemo {
  /** Main method */
  public static void main(String[] args) {
    // Display circle and rectangle properties
    displayObject(new CircleFromSimpleGeometricObject
             (1, "red", false));
    displayObject(new RectangleFromSimpleGeometricObject
             (1, 1, "black", true));
  }

  /** Display geometric object properties */
  public static void displayObject(SimpleGeometricObject object) {
    System.out.println("Created on " + object.getDateCreated() +
      ". Color is " + object.getColor());
  }
}
```

Dynamic Binding

- A method can be implemented in several classes along the inheritance chain.
  - The JVM decides which method is invoked at runtime.
- A method can be defined in a superclass and overridden in its subclass.
- For example, the *toString()* method is defined in the *Object* class and overridden in *GeometricObject*.

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

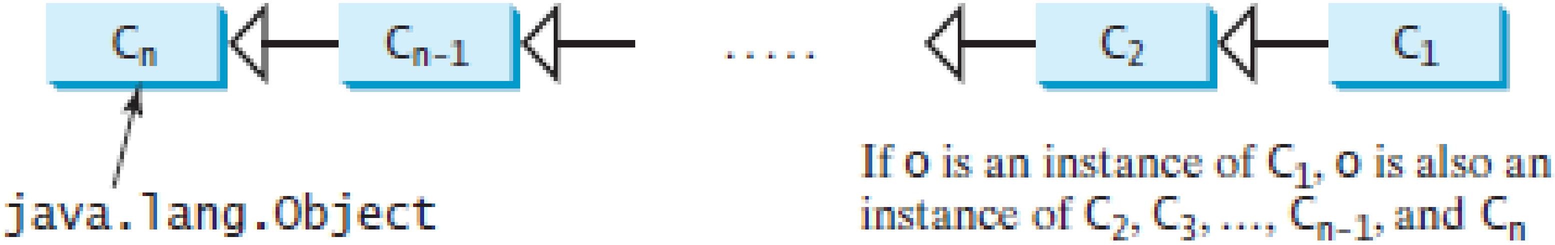
- Which *toString()* method is invoked by *o*?

Dynamic Binding (Cont.)

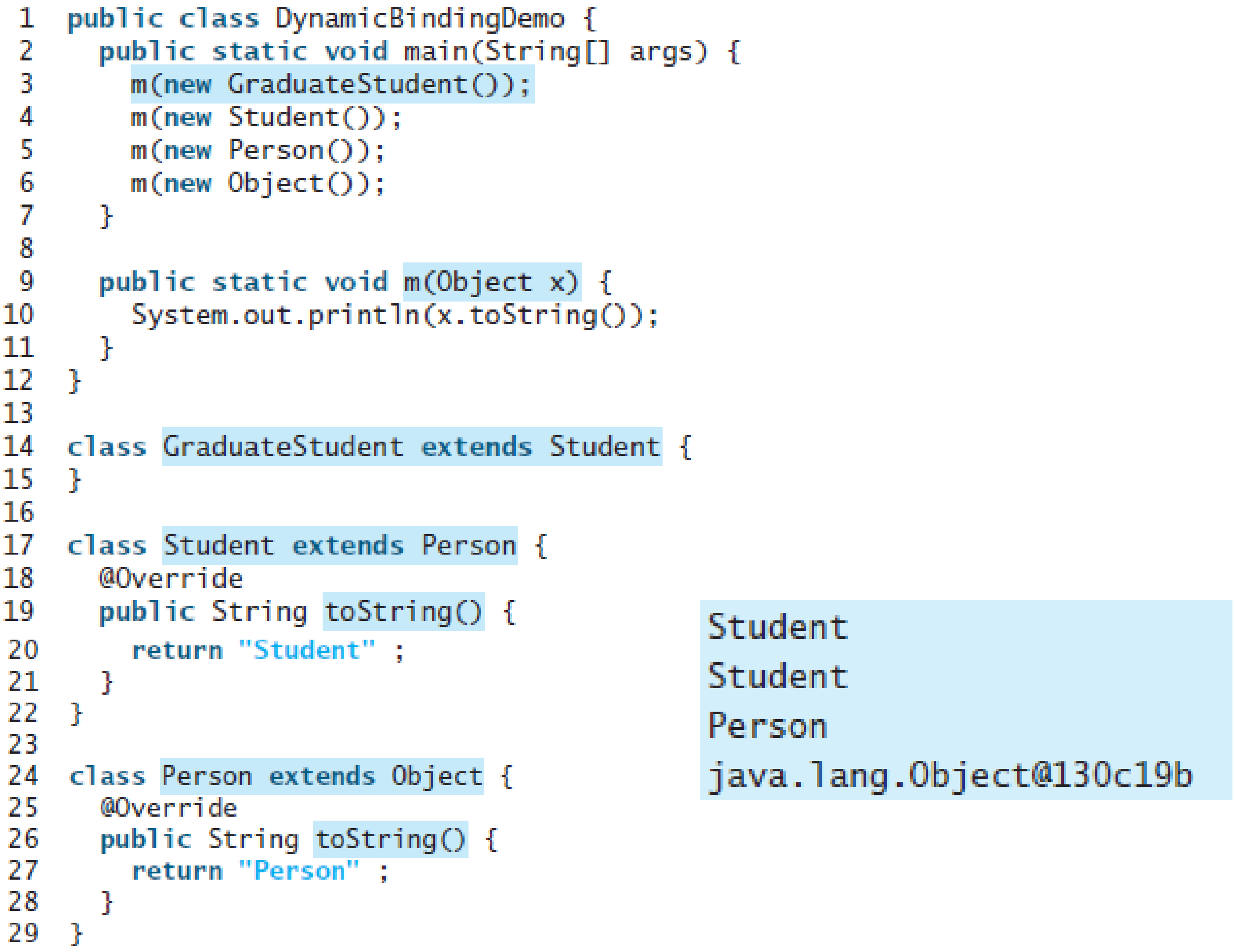
- The type that declares a variable is called the variable's ***declared type***.
  - In the previous example, *o*'s declared type is *Object*.
  - A variable of a reference type can hold a *null* value or a reference to an instance of the declared type.
  - The instance may be created using the constructor of the declared type or its subtype.
- The ***actual type*** of the variable is the actual class for the object referenced by the variable.
  - Here *o*'s actual type is *GeometricObject*, because *o* references an object created using *new GeometricObject()*.
- Which *toString()* method is invoked by *o* is determined by *o*'s **actual type**. This is known as *dynamic binding*.

Dynamic Binding (Cont.)

- Suppose an object *o* is an instance of classes *C1*, *C2*, . . . , *Cn-1*, and *Cn*, where *C1* is a subclass of *C2*, *C2* is a subclass of *C3*, . . . , and *Cn-1* is a subclass of *Cn*, as shown in the figure.
- That is, *Cn* is the most general class, and *C1* is the most specific class.
- In Java, *Cn* is the *Object* class.
- If *o* invokes a method *p*, the JVM searches for the implementation of the method *p* in *C1*, *C2*, . . . , *Cn-1*, and ***Cn***, in this order, until it is found.
- Once an implementation is found, the search stops and the first-found implementation is invoked.



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



```
public class DynamicBindingDemo {
  public static void main(String[] args) {
    m(new GraduateStudent());
    m(new Student());
    m(new Person());
    m(new Object());
  }

  public static void m(Object x) {
    System.out.println(x.toString());
  }
}

class GraduateStudent extends Student {
}

class Student extends Person {
  @Override
  public String toString() {
    return "Student" ;
  }
}

class Person extends Object {
  @Override
  public String toString() {
    return "Person" ;
  }
}
```

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

Dynamic Binding (Cont.)

- Matching a method signature and binding a method implementation are two separate issues.
- The **declared type** of the reference variable decides which method to match at compile time.
- The compiler finds a matching method according to the parameter type, number of parameters, and order of the parameters at compile time.
- A method may be implemented in several classes along the inheritance chain. The JVM dynamically binds the implementation of the method at runtime, decided by the **actual type** of the variable.

Casting Objects

- One object reference can be typecast into another object reference.
  - This is called casting object.
- 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 an instance of *Object*.
- Suppose you want to assign the object reference *o* to a variable of the *Student* type using the following statement:

  Student b = o;

  In this case a compile error would occur.

Casting Objects (Cont.)

- The reason is that 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 clever enough to know it.
- To tell the compiler that *o* is a *Student* object, use ***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:

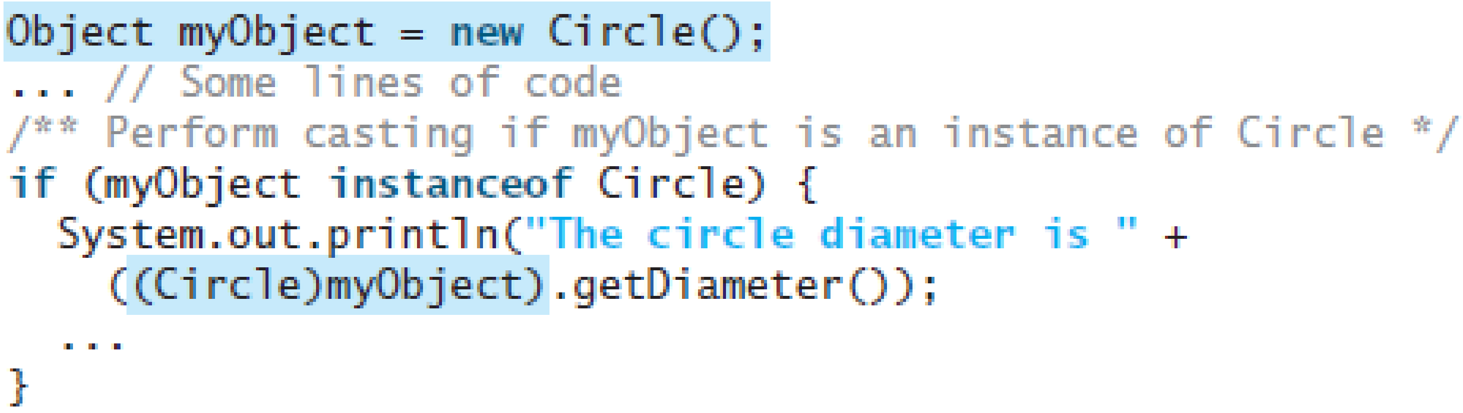
  Student b = (Student)o; // Explicit casting

Casting Objects (Cont.)

- 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.
- 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 Objects (Cont.)

- 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*.
- It is a good practice, therefore, to ensure that the object is an instance of another object before attempting a casting.
  - This can be accomplished by using the *instanceof* operator.

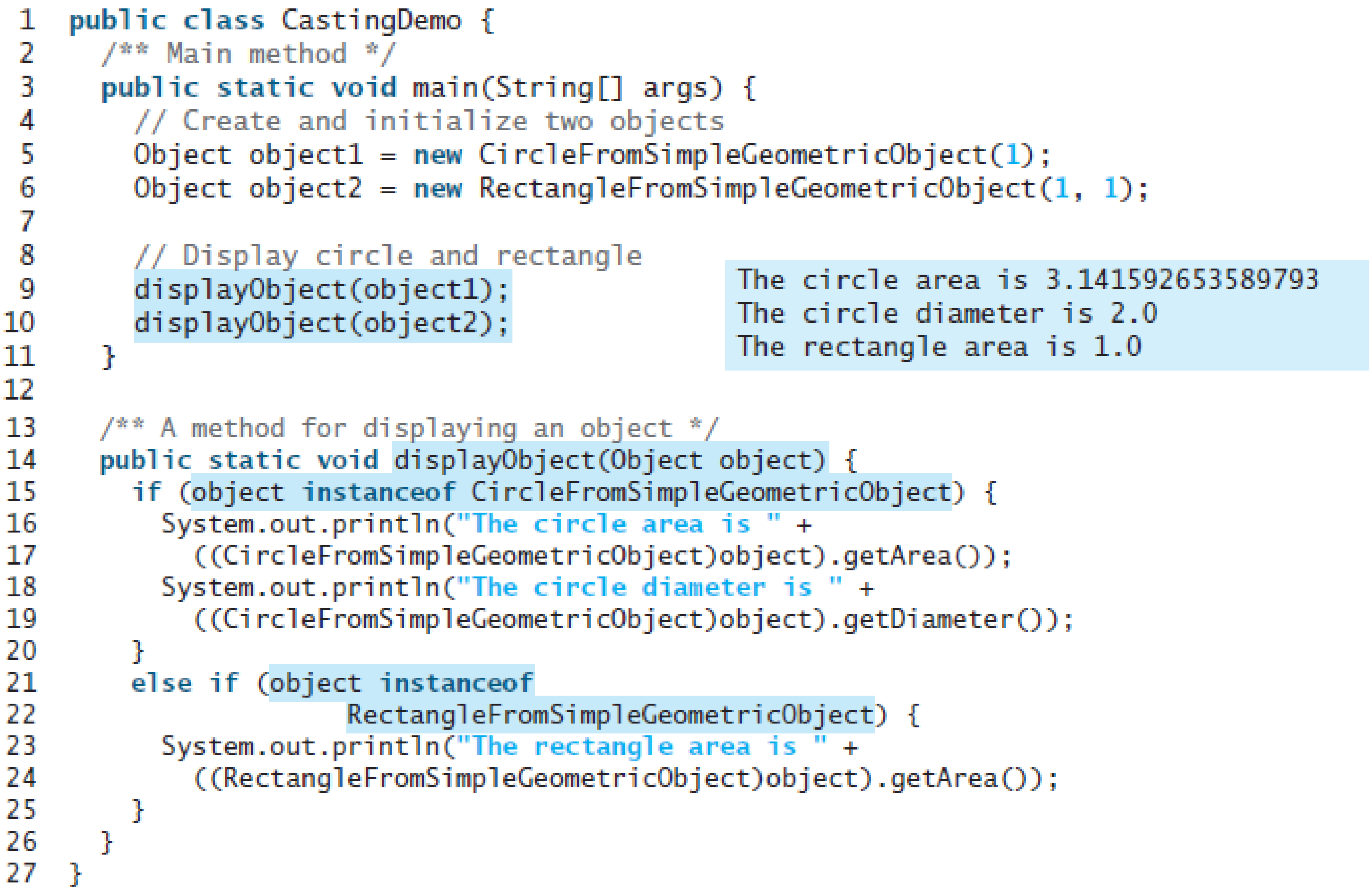


```
Object myObject = new Circle();
... // Some lines of code
/** Perform casting if myObject is an instance of Circle */
if (myObject instanceof Circle) {
  System.out.println("The circle diameter is " +
    ((Circle)myObject).getDiameter());
  ...
}
```

Why Casting is Necessary?

- You may be wondering why casting is necessary. The variable *myObject* is declared *Object*.
- The ***declared type*** decides which method to match at compile time.
  - Using *myObject.getDiameter()* would cause a compile error, because the *Object* class does not have the *getDiameter* method.
  - The compiler cannot find a match for *myObject.getDiameter()*.
- Therefore, it is necessary to cast *myObject* into the *Circle* type to tell the compiler that *myObject* is also an instance of *Circle*.
- Why not define *myObject* as a *Circle* type in the first place?
  - To enable **generic programming**, it is a good practice to define a variable with a supertype, which can accept an object of any subtype.

Casting and Polymorphism



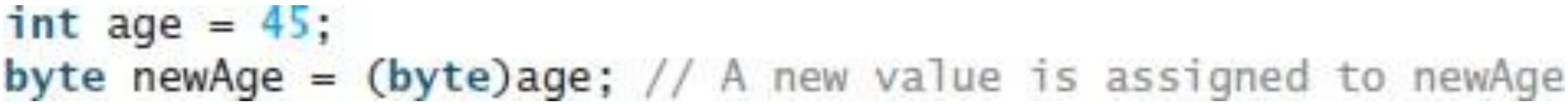
```java
public class CastingDemo {
  /** Main method */
  public static void main(String[] args) {
    // Create and initialize two objects
    Object object1 = new CircleFromSimpleGeometricObject(1);
    Object object2 = new RectangleFromSimpleGeometricObject(1, 1);

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

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

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

Comments

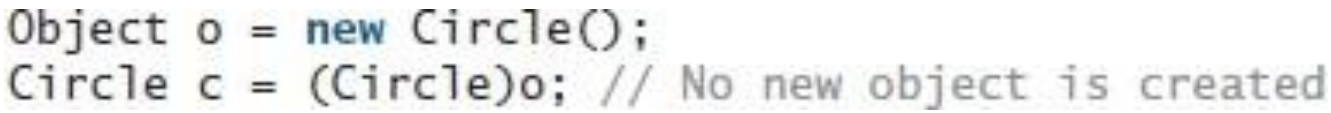
- The object member access operator (.) precedes the casting operator.
  - Use parentheses to ensure that casting is done before the . operator, as in

  ((Circle)object).getArea();
- Casting a primitive type value is different from casting an object reference.
  - Casting a primitive type value returns a new value. For example:



```
int age = 45;
byte newAge = (byte)age; // A new value is assigned to newAge
```

  - However, casting an object reference does not create a new object. For example:



```
Object o = new Circle();
Circle c = (Circle)o; // No new object is created
```

The Object’s *equals* Method

- Another method defined in the *Object* class that is often used is the *equals* method. Its signature is

```
public boolean equals(Object o)
```

- This method tests whether two objects are equal. The syntax for invoking it is:

```
object1.equals(object2);
```

- The default implementation of the *equals* method in the *Object* class is:

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

- This implementation checks whether two reference variables point to the same object using the == operator.
- You should override this method in your custom class to test whether two distinct objects have the same content.

The Object’s *equals* Method (Cont.)

- The *equals* method is overridden in many classes in the Java API, such as *java.lang.String* and *java.util.Date*, to compare whether the contents of two objects are equal.
- You can override the *equals* method in the *Circle* class to compare whether two circles are equal based on their radius as follows:

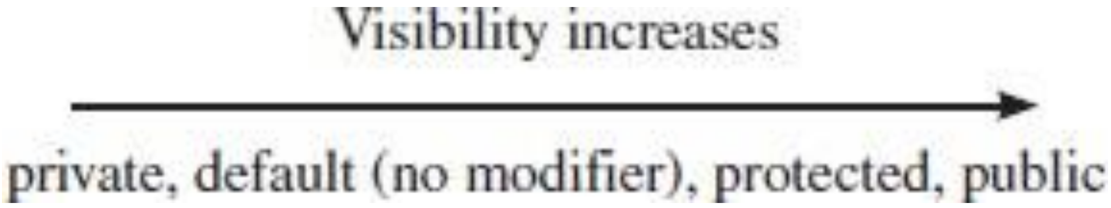
```
public boolean equals(Object o) {
      if (o instanceof Circle)
            return radius == ((Circle)o).radius;
      else
            return this == o;
}
```

- Using the signature *equals(SomeClassName obj)* (e.g., *equals(Circle c))* to override the *equals* method in a subclass is a common mistake. You should use *equals(Object ob*j).

The Protected Data and Methods

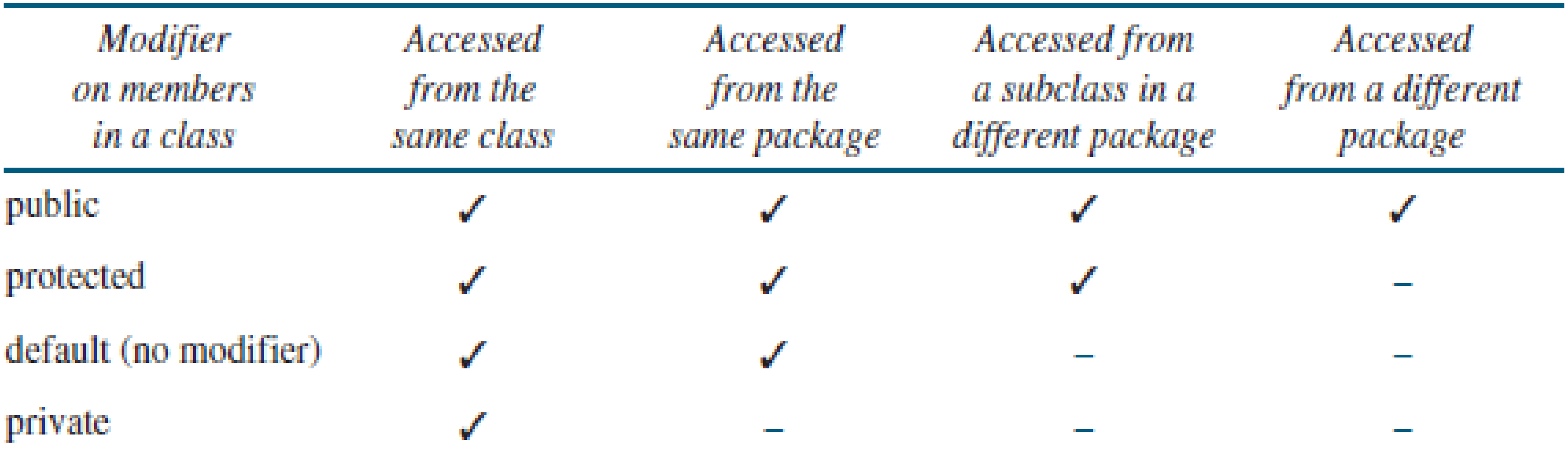
- So far you have used the *private* and *public* keywords to specify whether data fields and methods can be accessed from outside of the class.
- *Private* members can be accessed only from inside of the class, and *public* members can be accessed from any other classes.
- Often it is desirable to allow subclasses to access data fields or methods defined in the superclass, but not to allow non-subclasses to access these data fields and methods.
- To accomplish this, you can use the *protected* keyword.
  - This way you can access protected data fields or methods in a superclass from its subclasses.

The Protected Data and Methods (Cont.)



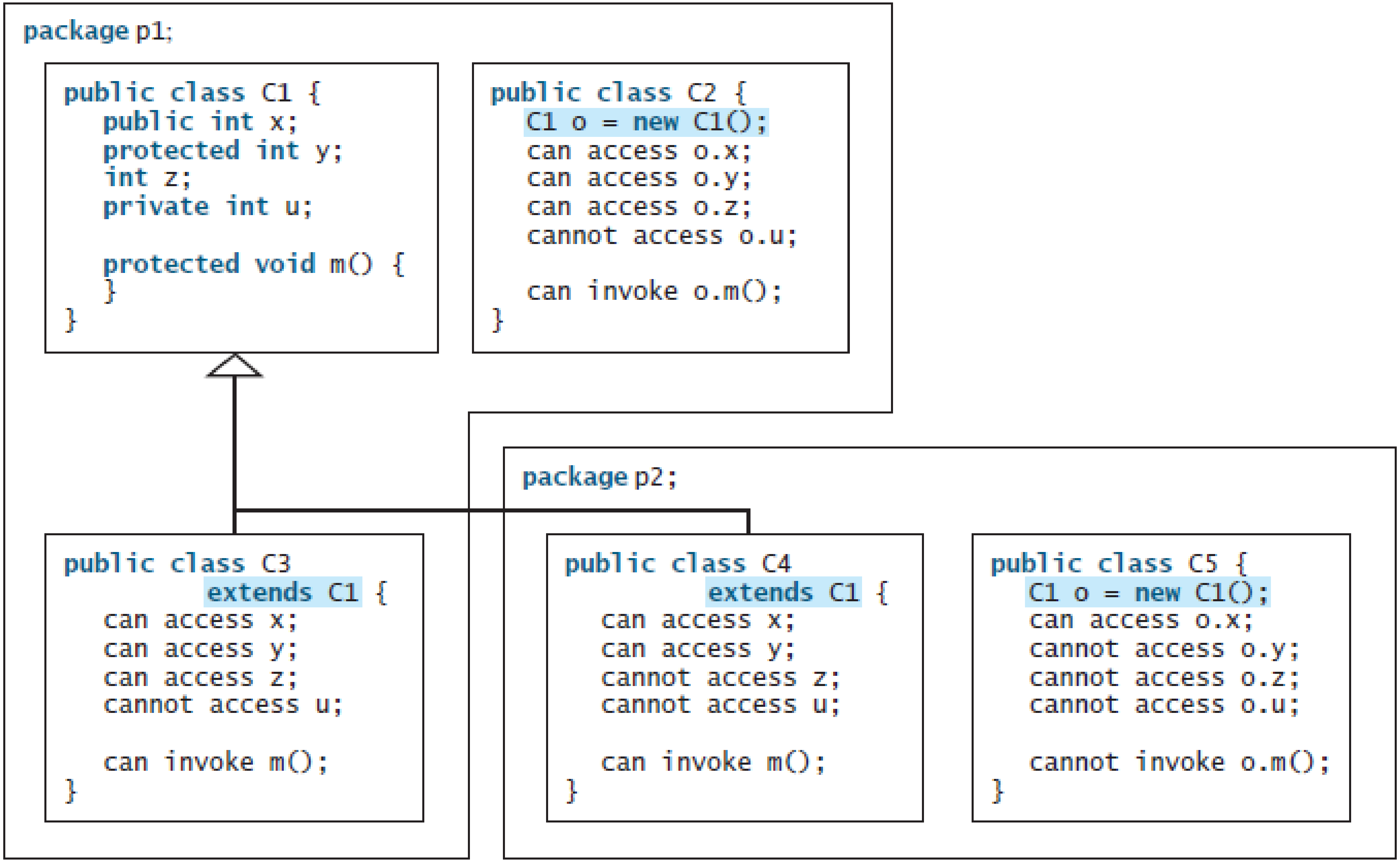
Visibility increases

private, default (no modifier), protected, public



Modifier on members in a class	*Accessed from the same class*	*Accessed from the same package*	*Accessed from a subclass in a different package*	*Accessed from a different package*
public	✓	✓	✓	✓
protected	✓	✓	✓	–
default (no modifier)	✓	✓	–	–
private	✓	–	–	–

The Protected Data and Methods (Cont.)



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();
}
```

Visibility Modifiers (Comments)

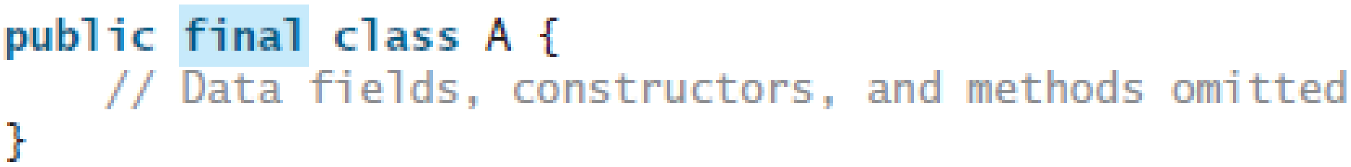
- Your class can be used in two ways:

  (1) for creating instances of the class and

  (2) for defining subclasses by extending the class.
- Make the members *private* if they are not intended for use from outside the class.
- Make the members *public* if they are intended for the users of the class.
- Make the fields or methods *protected* if they are intended for the extenders of the class but not for the users of the class.
- A subclass cannot weaken the accessibility of a method defined in the superclass when overriding it.
  - For example, if a method is defined as public in the superclass, it must be defined as public in the subclass.

Preventing Extending and Overriding

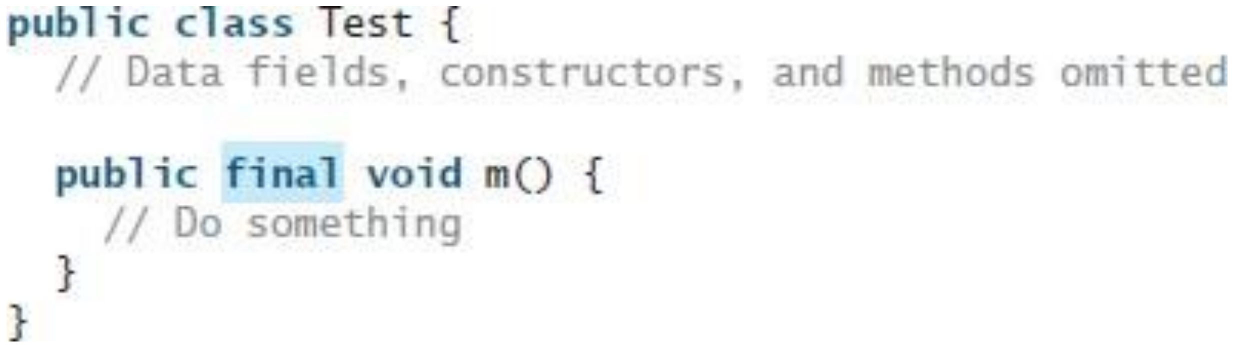
- You may occasionally want to prevent classes from being extended.
- In such cases, use the *final* modifier to indicate that a class is final and cannot be a parent class.
- The *Math* class is a final class. The *String*, StringBuilder, and *StringBuffer* classes are also final classes.



```
public final class A {
    // Data fields, constructors, and methods omitted
}
```

Preventing Extending and Overriding (Cont.)

- You also can define a method to be *final*; a final method cannot be overridden by its subclasses.
- For example, the following method *m* is final and cannot be overridden:



```
public class Test {
  // Data fields, constructors, and methods omitted

  public final void m() {
    // Do something
  }
}
```