Inheritance

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

- 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 enable data and methods in a superclass accessible from subclasses using the **protected** visibility modifier.

Introduction

- Inheritance
  - A new class is created by acquiring an existing class’s members and possibly embellishing them with new or modified capabilities.
  - Can save time during program development by basing new classes on existing proven and debugged high-quality software.
  - Increases the likelihood that a system will be implemented and maintained effectively.

Introduction (Cont.)

- When creating a class, rather than declaring completely new members, you can designate that the new class should *inherit* the members of an existing class.
  - Existing class is the superclass
  - New class is the subclass
- A subclass can be a superclass of future subclasses.
- A subclass can add its own fields and methods.
- A subclass is more specific than its superclass and represents a more specialized group of objects.
- The subclass exhibits the behaviors of its superclass and can add behaviors that are specific to the subclass.
  - This is why inheritance is sometimes referred to as specialization.

Introduction (Cont.)

- The direct superclass is the superclass from which the subclass explicitly inherits.
- An indirect superclass is any class above the direct superclass in the class hierarchy.
- The Java class hierarchy begins with class `Object` (in package `java.lang`)
  - *Every* class in Java directly or indirectly extends (or “inherits from”) `Object`.
- Java supports only single inheritance, in which each class is derived from exactly one direct superclass.

Introduction (Cont.)

- We distinguish between the *is-a* relationship and the *has-a* relationship
- *Is-a* represents inheritance
  - In an *is-a* relationship, an object of a subclass can also be treated as an object of its superclass
- *Has-a* represents composition
  - In a *has-a* relationship, an object contains as members references to other objects

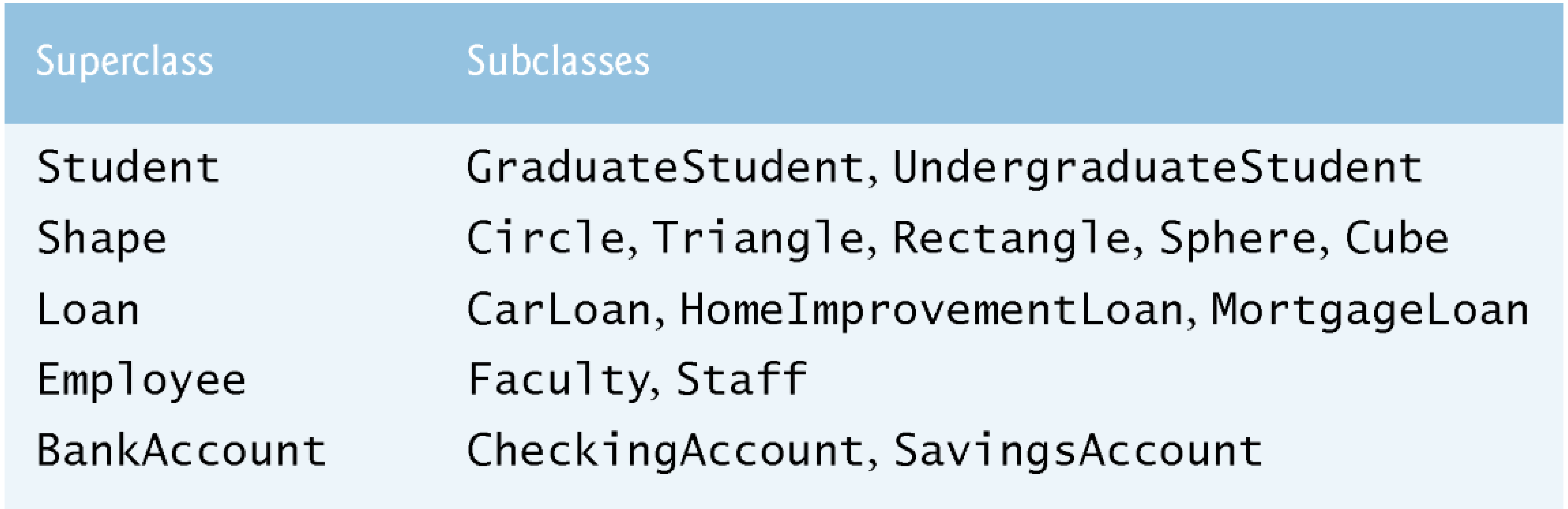
Superclasses and Subclasses

- Superclasses tend to be “more general” and subclasses “more specific.”
- Because every subclass object *is an* object/instance of its superclass, and one superclass can have many subclasses, the set of objects represented by a superclass is typically larger than the set of objects represented by any of its subclasses.

Superclasses and Subclasses (Cont.)

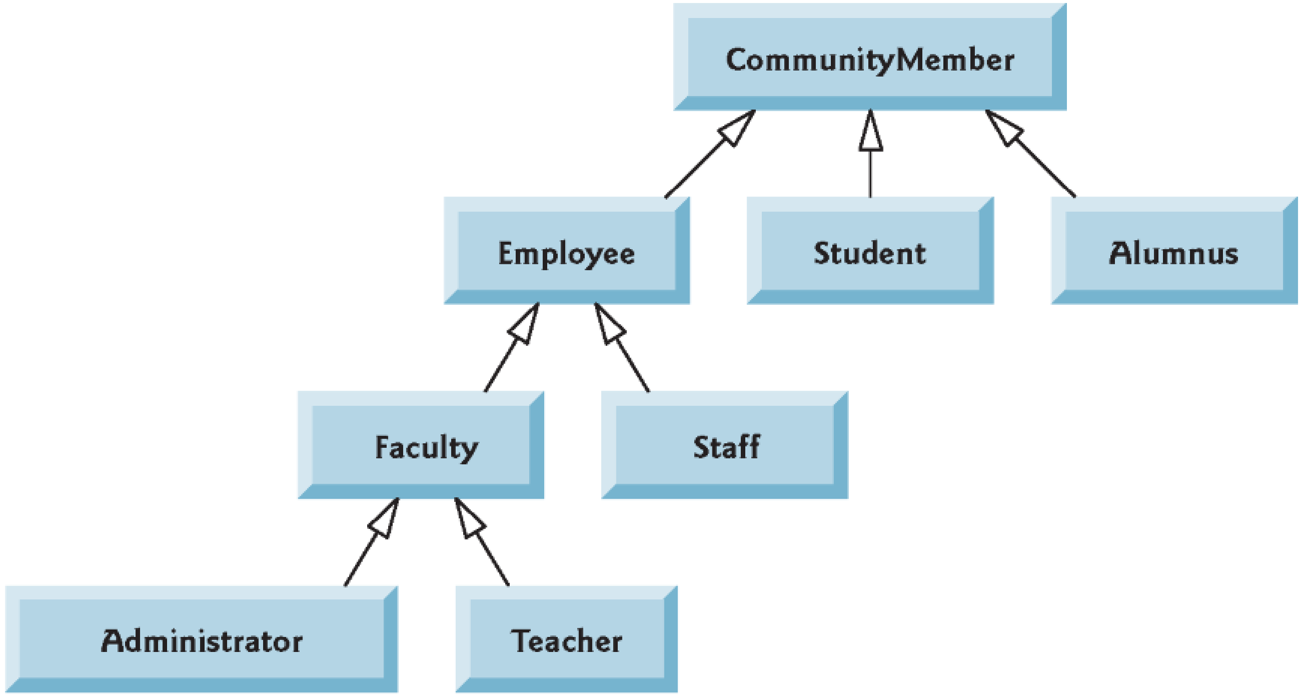
- Not every class relationship is an inheritance relationship.
- *Has-a* relationship
  - Create classes by composition of existing classes.
  - Example: Given the classes `Employee`, `BirthDate` and `TelephoneNumber`, it’s improper to say that an `Employee` *is a* `BirthDate` or that an `Employee` *is a* `TelephoneNumber`.
  - However, an `Employee` *has a* `BirthDate`, and an `Employee` *has a* `TelephoneNumber`.

Inheritance Examples:

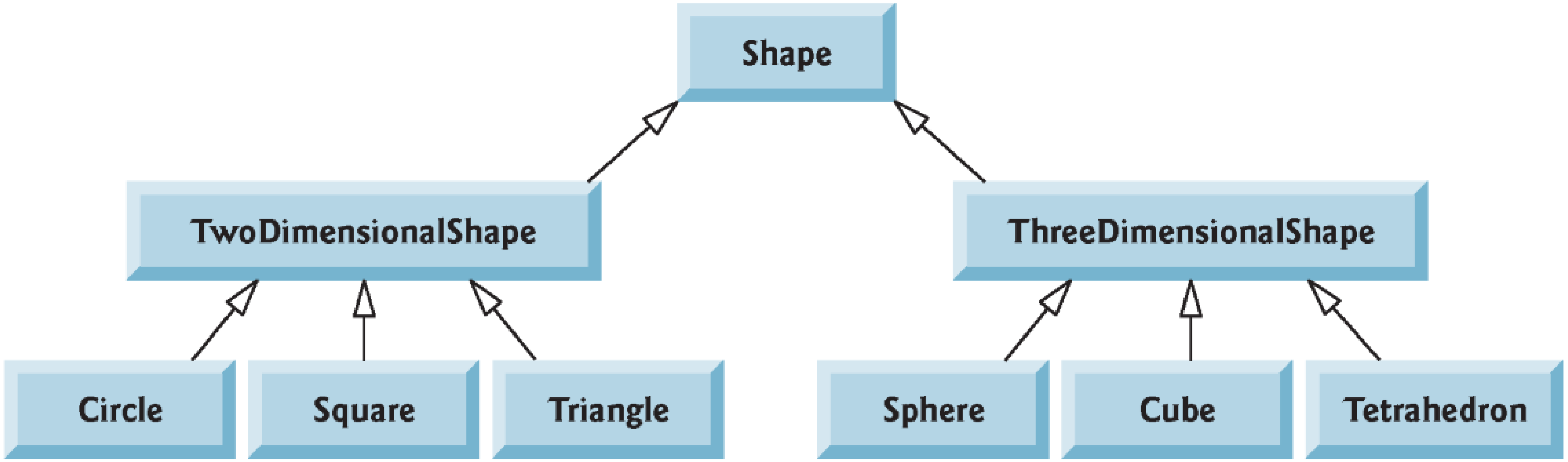


Superclass	Subclasses
Student	GraduateStudent, UndergraduateStudent
Shape	Circle, Triangle, Rectangle, Sphere, Cube
Loan	CarLoan, HomeImprovementLoan, MortgageLoan
Employee	Faculty, Staff
BankAccount	CheckingAccount, SavingsAccount

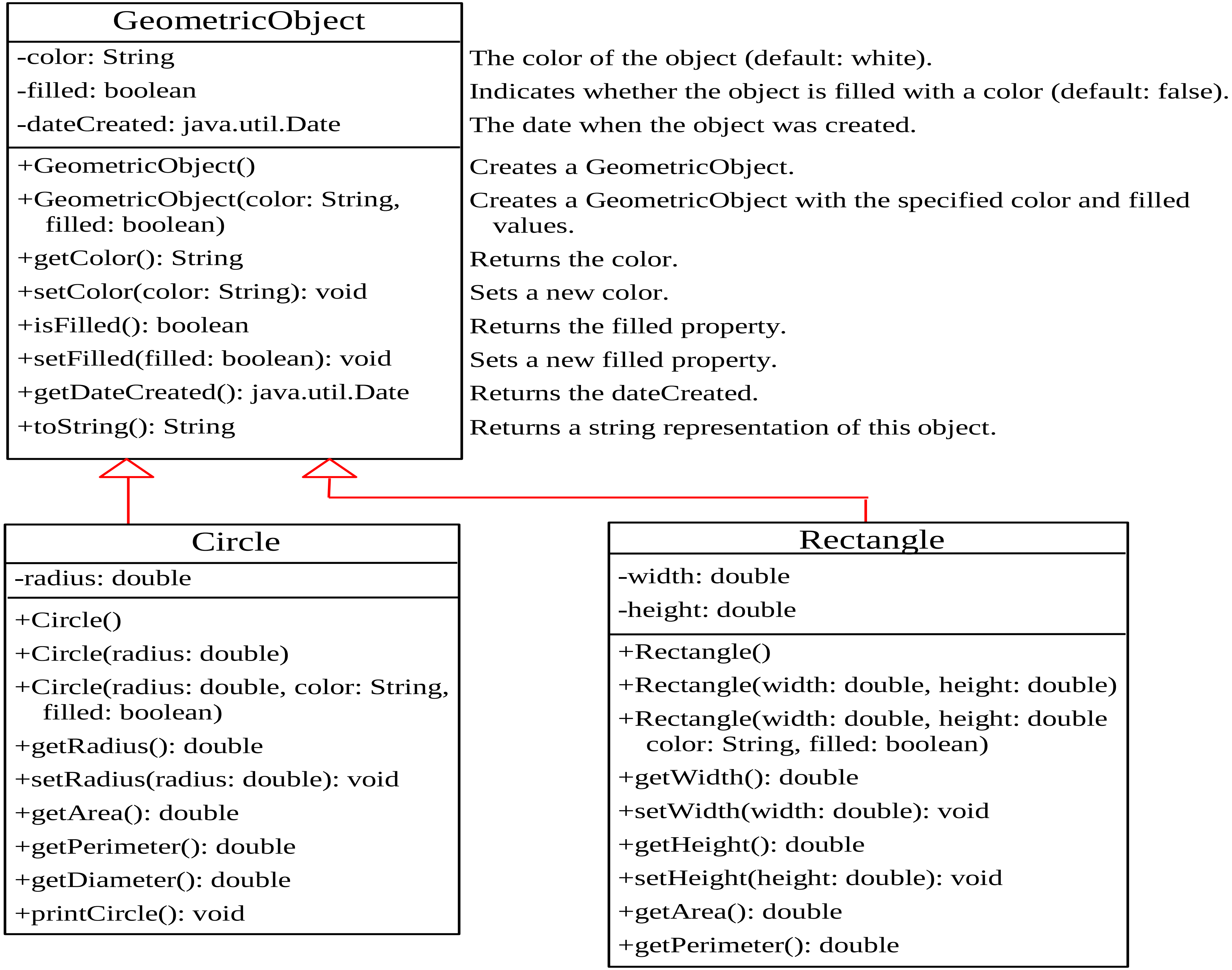
Inheritance Example



Inheritance Example



Superclass and Subclasses



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

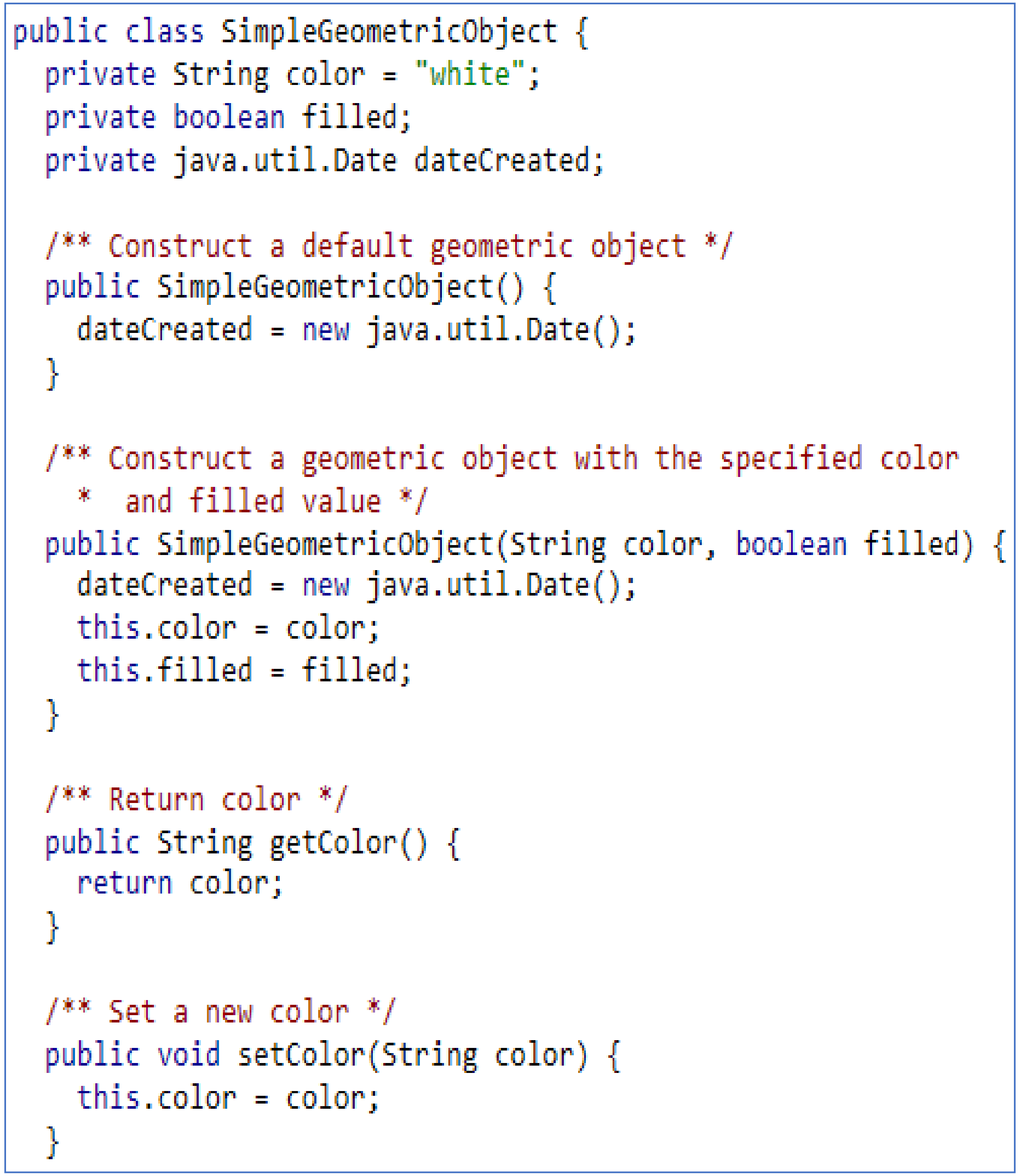
-radius: double

+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

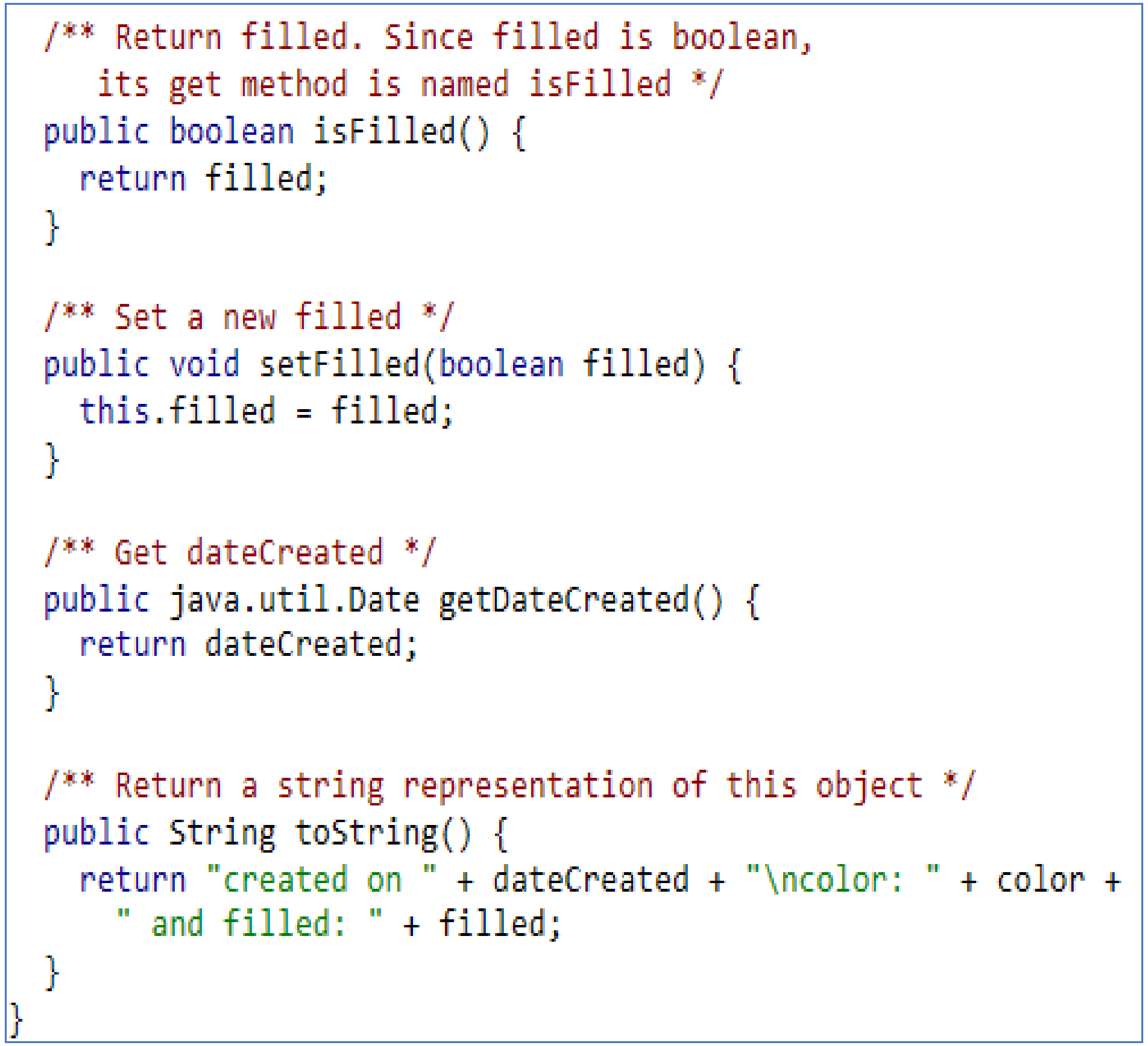
Rectangle

-width: double
-height: double

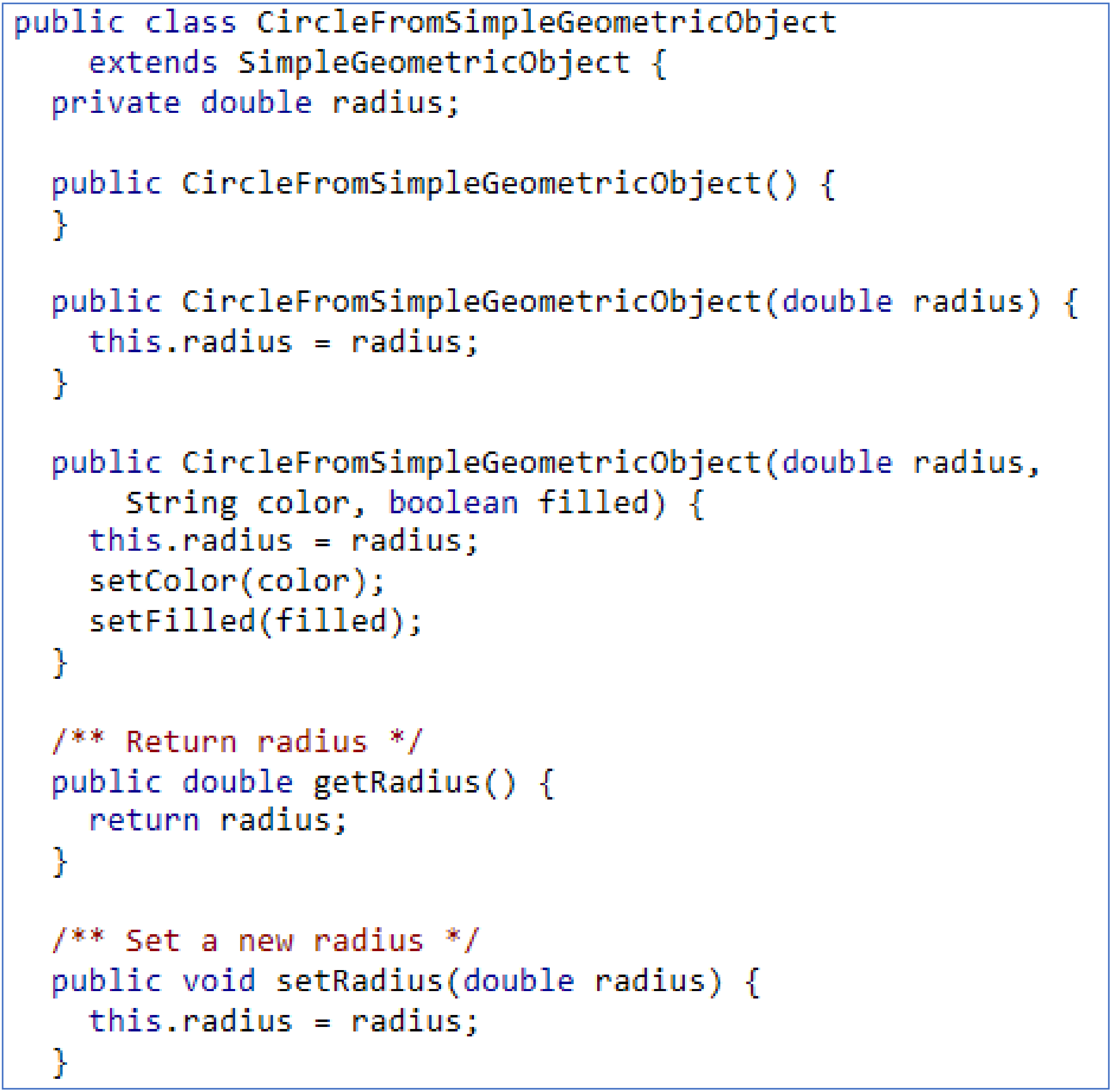
+Rectangle()
+Rectangle(width: double, height: double)
+Rectangle(width: double, height: double color: String, filled: boolean)
+getWidth(): double
+setWidth(width: double): void
+getHeight(): double
+setHeight(height: double): void
+getArea(): double
+getPerimeter(): double



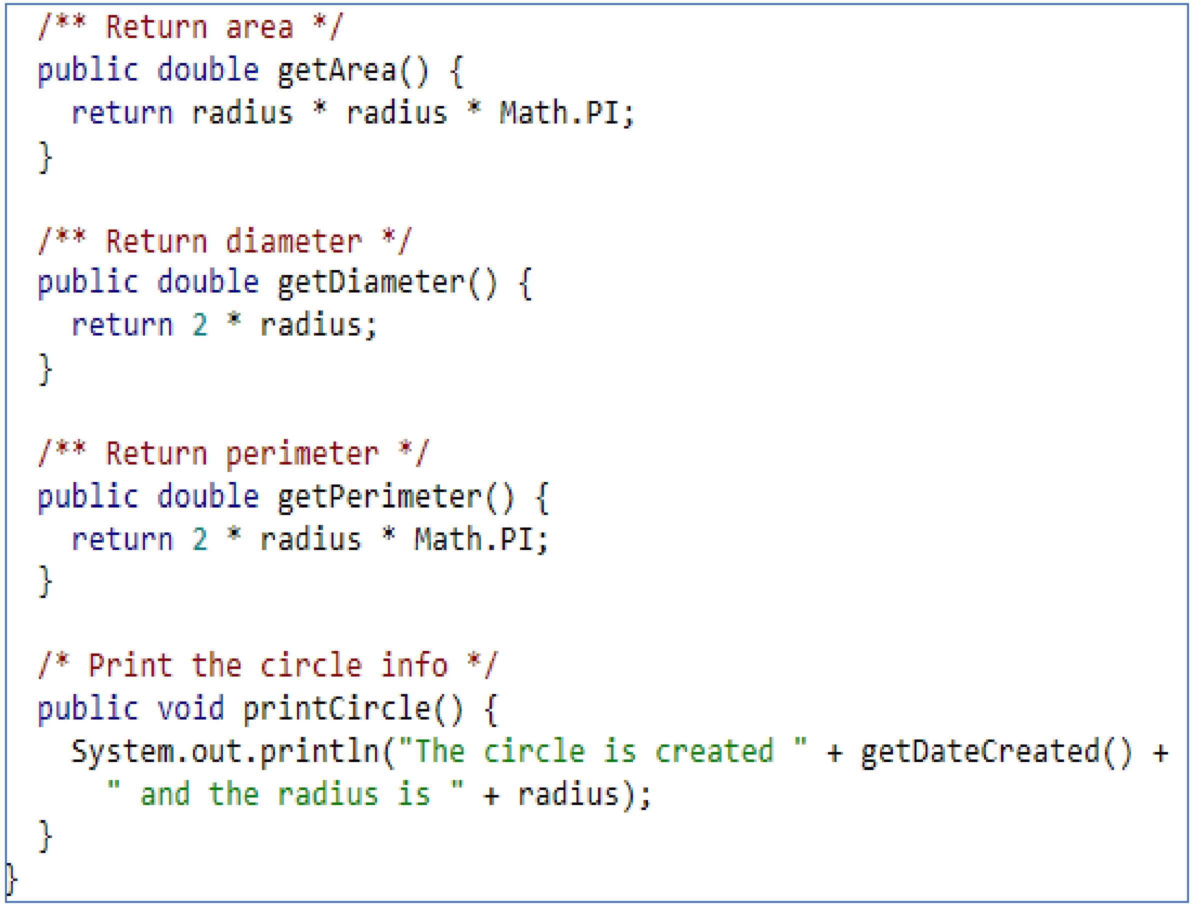
```java
public class SimpleGeometricObject {
  private String color = "white";
  private boolean filled;
  private java.util.Date dateCreated;

  /** 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;
  }

  /** Return color */
  public String getColor() {
    return color;
  }

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



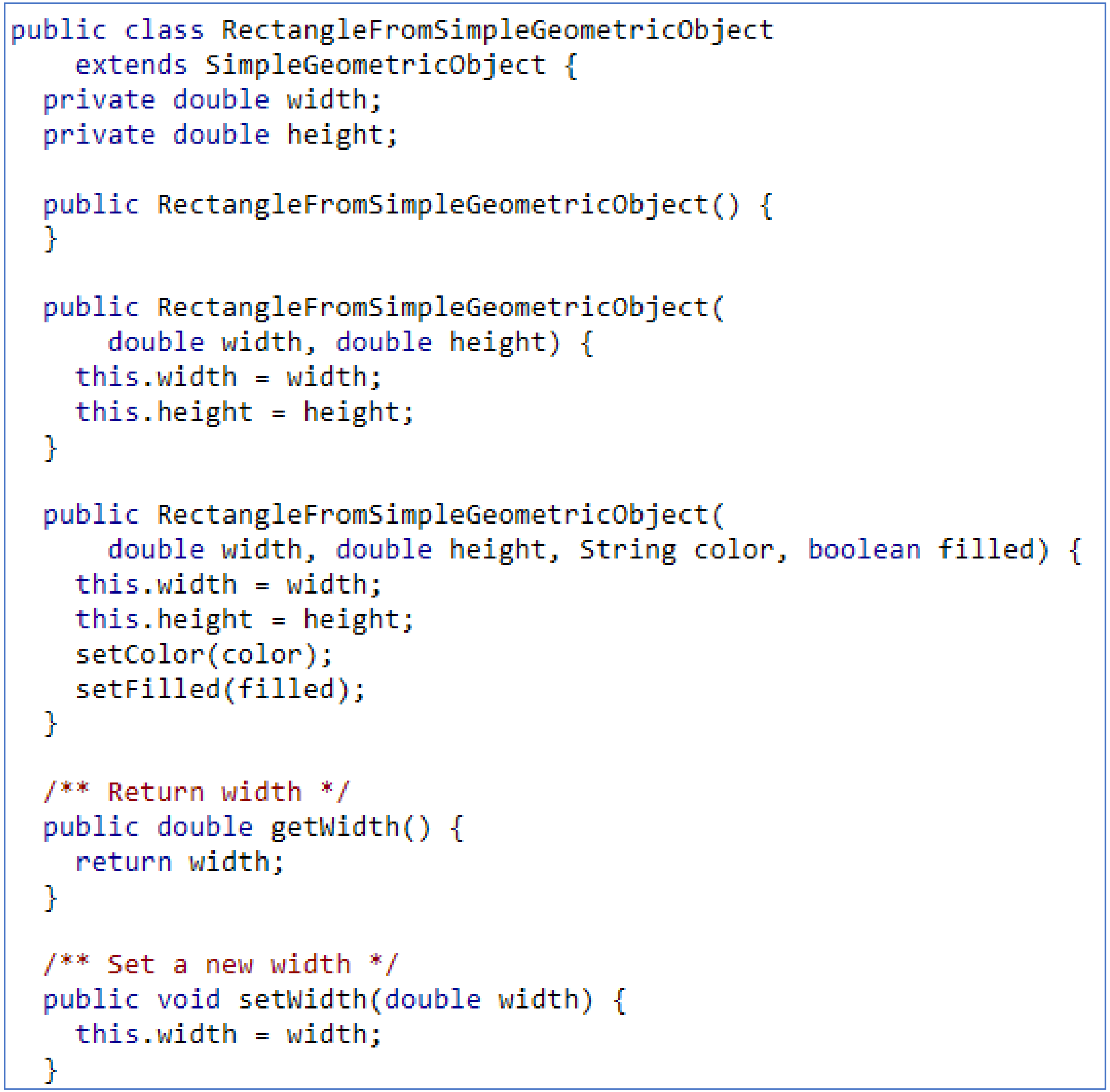
```java
  /** Return filled. Since filled is boolean,
     its get method is named isFilled */
  public boolean isFilled() {
    return filled;
  }

  /** Set a new filled */
  public void setFilled(boolean filled) {
    this.filled = filled;
  }

  /** 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;
  }
}
```



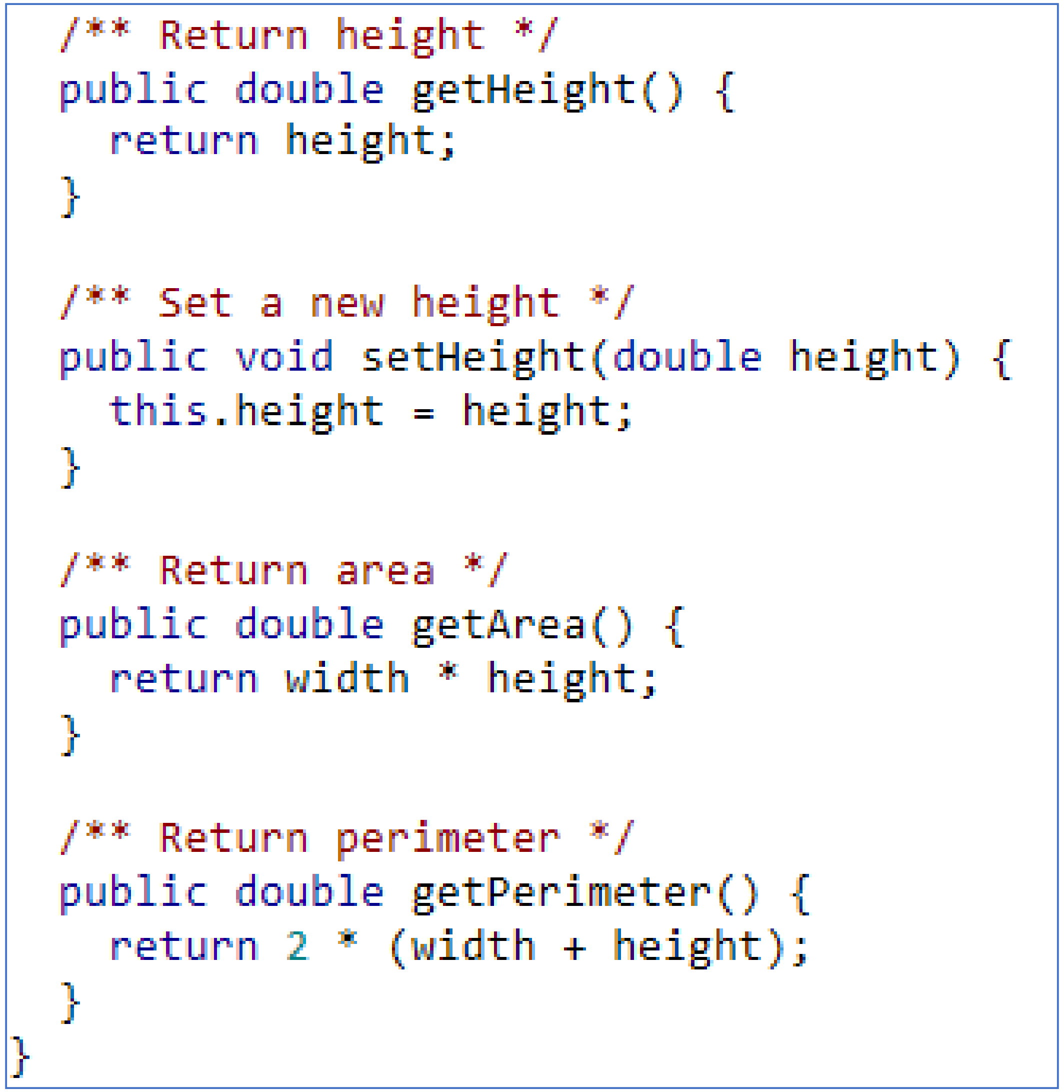
```java
public class CircleFromSimpleGeometricObject
    extends SimpleGeometricObject {
  private double radius;

  public CircleFromSimpleGeometricObject() {
  }

  public CircleFromSimpleGeometricObject(double radius) {
    this.radius = radius;
  }

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

  /** Return radius */
  public double getRadius() {
    return radius;
  }

  /** Set a new radius */
  public void setRadius(double radius) {
    this.radius = radius;
  }
```



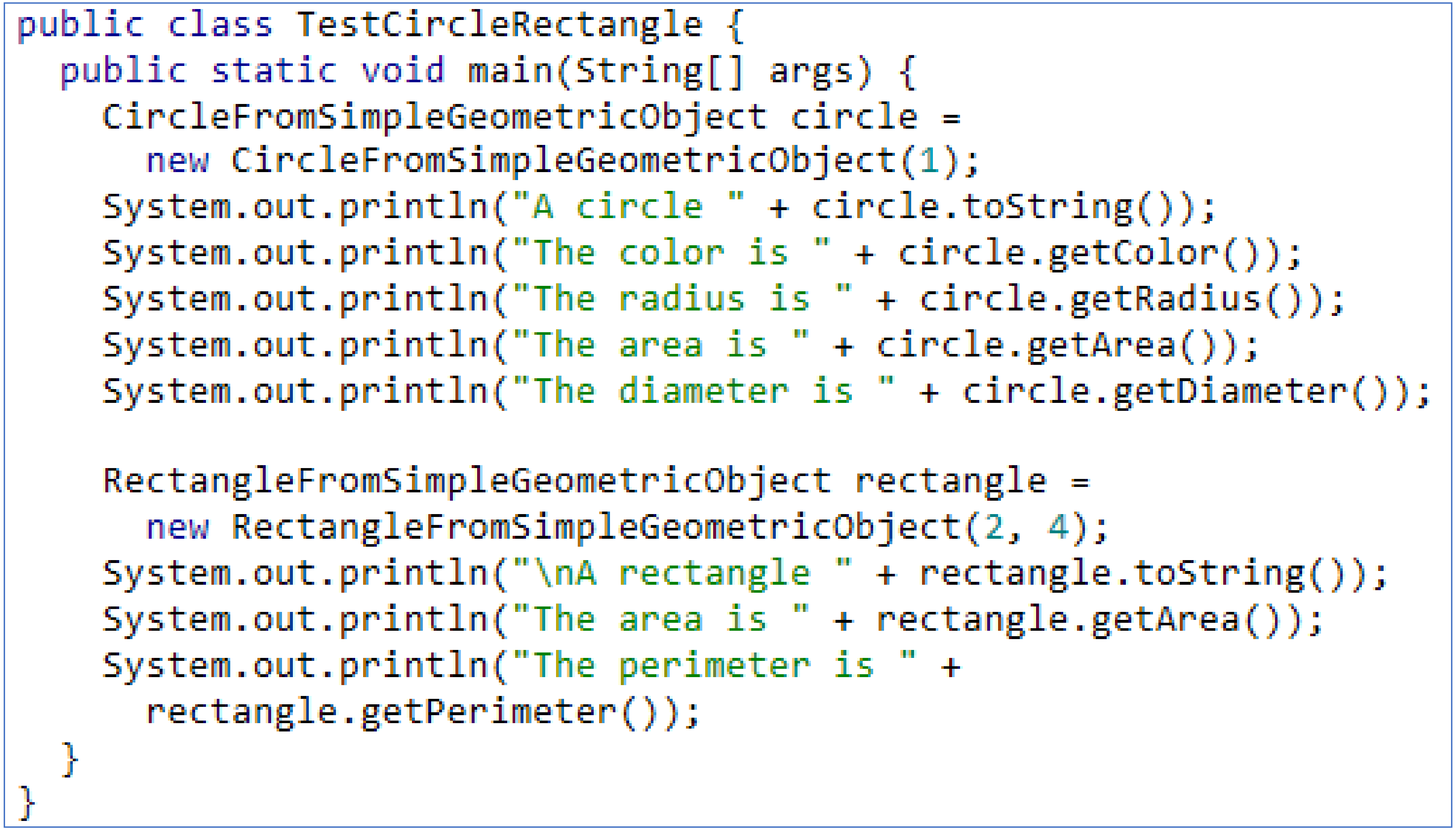
```java
  /** Return area */
  public double getArea() {
    return radius * radius * Math.PI;
  }

  /** Return diameter */
  public double getDiameter() {
    return 2 * radius;
  }

  /** 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);
  }
}
```



```
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);
  }

  /** 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;
  }

  /** Return area */
  public double getArea() {
    return width * height;
  }

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



```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());
  }
}
```

Are superclass's Constructor Inherited?

No. They are not inherited.

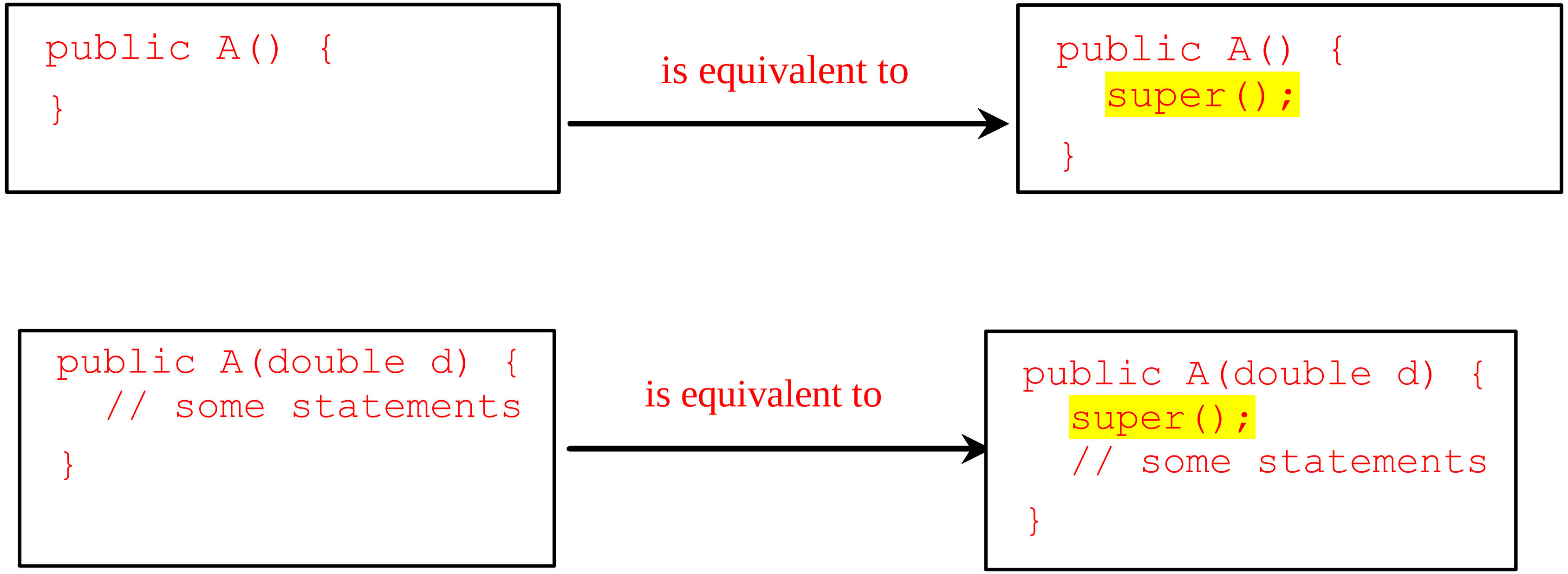
They are invoked explicitly or implicitly.

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,



```
public A() {
}
```

is equivalent to

```
public A() {
  super();
}
```

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

is equivalent to

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

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

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

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

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

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

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

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

animation

Trace Execution

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

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

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

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

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

1. Start from the main method

animation

Trace Execution

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

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

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

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

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

2. Invoke Faculty constructor

animation

Trace Execution

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

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

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

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

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

3. Invoke Employee’s no-arg constructor

animation

Trace Execution

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

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

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

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

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

4. Invoke Employee(String) constructor

animation

Trace Execution

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

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

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

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

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

5. Invoke Person() constructor

animation

Trace Execution

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

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

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

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

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

6. Execute println

animation

Trace Execution

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

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

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

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

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

7. Execute println

animation

Trace Execution

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

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

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

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

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

8. Execute println

animation

Trace Execution

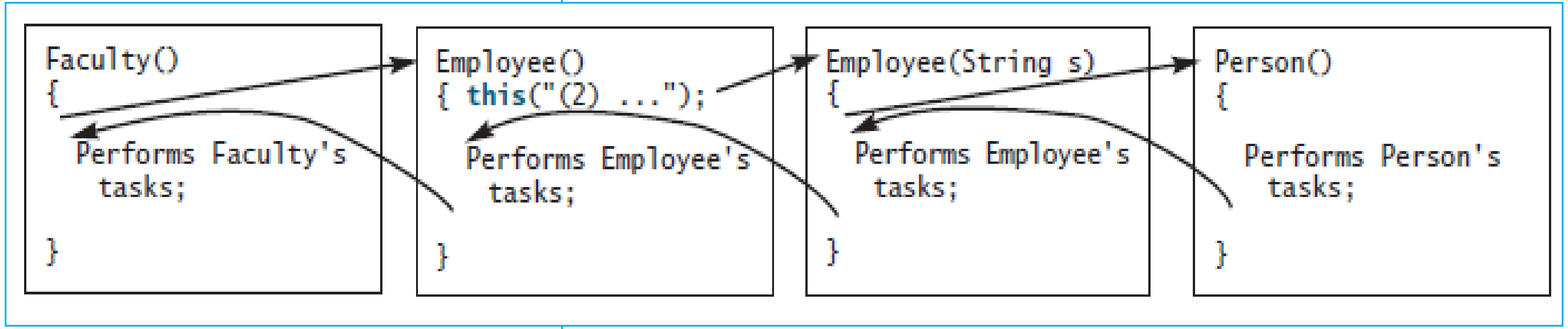
```
public class Faculty extends Employee {
  public static void main(String[] args) {
    new Faculty();
  }

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

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

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

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

9. Execute println

```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");
  }
}
```

Output

```
(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;

}
```

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");
  }
}
```

Superclasses and Subclasses (Cont.)

- Objects of all classes that extend a common superclass can be treated as objects of that superclass.
  - Commonality expressed in the members of the superclass.
- Inheritance issue
  - A subclass can inherit methods that it does not need or should not have.
  - Even when a superclass method is appropriate for a subclass, that subclass often needs a customized version of the method.
  - The subclass can override (redefine) the superclass method with an appropriate implementation.

Defining a Subclass

A subclass inherits from a superclass. You can also:

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

Calling Superclass Methods

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

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

Overriding Methods in the Superclass

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

```
public class Circle extends GeometricObject {
  // Other methods are omitted


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

NOTE

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

NOTE

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

Overriding vs. Overloading

```java
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);
  }
}
```

Test class in (a):
Both a.p(10) and a.p(10.0) display 10.0

```java
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);
  }
}
```

Test class in (b):
a.p(10) display 10 and a.p(10.0) display 20.0

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.

Creating and Using a CommissionEmployee Class

- Class CommissionEmployee (Fig. 9.4) extends class Object (from package java.lang).
  - CommissionEmployee inherits Object's methods.
  - If you don't explicitly specify which class a new class extends, the class extends Object implicitly.

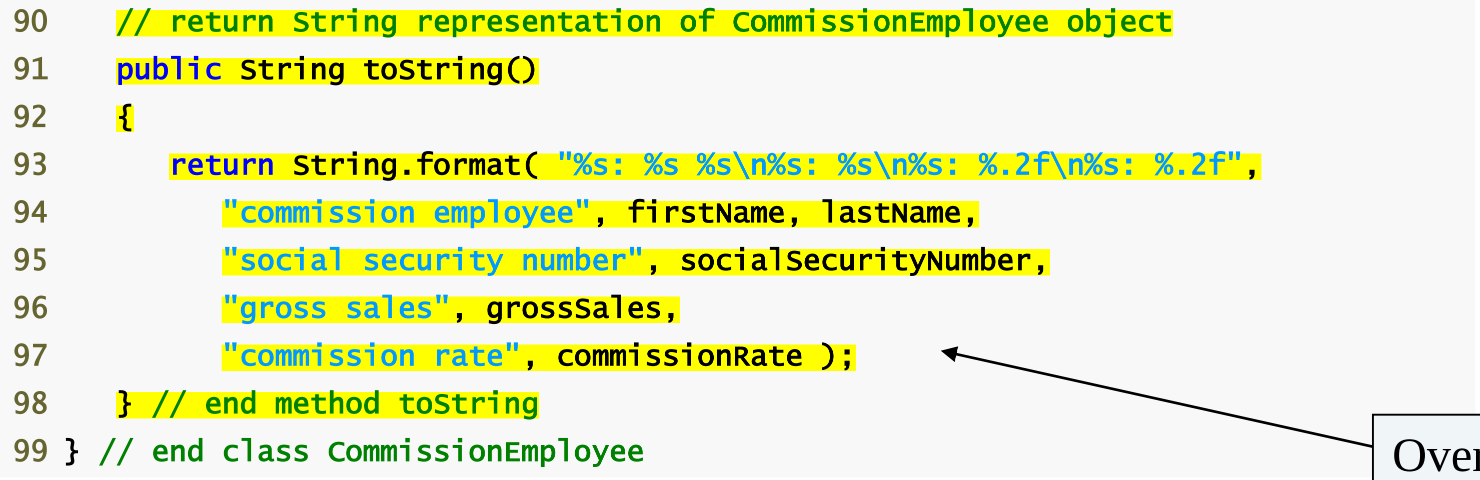
```java
// Fig. 9.4: CommissionEmployee.java
// CommissionEmployee class represents a commission employee.

public class CommissionEmployee extends Object
{
   private String firstName;
   private String lastName;
   private String socialSecurityNumber;
   private double grossSales; // gross weekly sales
   private double commissionRate; // commission percentage

   // five-argument constructor
   public CommissionEmployee( String first, String last, String ssn,
      double sales, double rate )
   {
      // implicit call to Object constructor occurs here
      firstName = first;
      lastName = last;
      socialSecurityNumber = ssn;
      setGrossSales( sales ); // validate and store gross sales
      setCommissionRate( rate ); // validate and store commission rate
   } // end five-argument CommissionEmployee constructor

   // set first name
   public void setFirstName( String first )
   {
      firstName = first;
   } // end method setFirstName

```

Class `CommissionEmployee` `extends` class `Object`

Invoke methods `setGrossSales` and `setCommissionRate` to validate data

```java
   // return first name
   public String getFirstName()
   {
      return firstName;
   } // end method getFirstName

   // set last name
   public void setLastName( String last )
   {
      lastName = last;
   } // end method setLastName

   // return last name
   public String getLastName()
   {
      return lastName;
   } // end method getLastName

   // set social security number
   public void setSocialSecurityNumber( String ssn )
   {
      socialSecurityNumber = ssn; // should validate
   } // end method setSocialSecurityNumber

   // return social security number
   public String getSocialSecurityNumber()
   {
      return socialSecurityNumber;
   } // end method getSocialSecurityNumber

   // set gross sales amount
   public void setGrossSales( double sales )
   {
      grossSales = ( sales < 0.0 ) ? 0.0 : sales;
   } // end method setGrossSales

   // return gross sales amount
   public double getGrossSales()
   {
      return grossSales;
   } // end method getGrossSales

   // set commission rate
   public void setCommissionRate( double rate )
   {
      commissionRate = ( rate > 0.0 && rate < 1.0 ) ? rate : 0.0;
   } // end method setCommissionRate

   // return commission rate
   public double getCommissionRate()
   {
      return commissionRate;
   } // end method getCommissionRate

   // calculate earnings
   public double earnings()
   {
      return commissionRate * grossSales;
   } // end method earnings

```

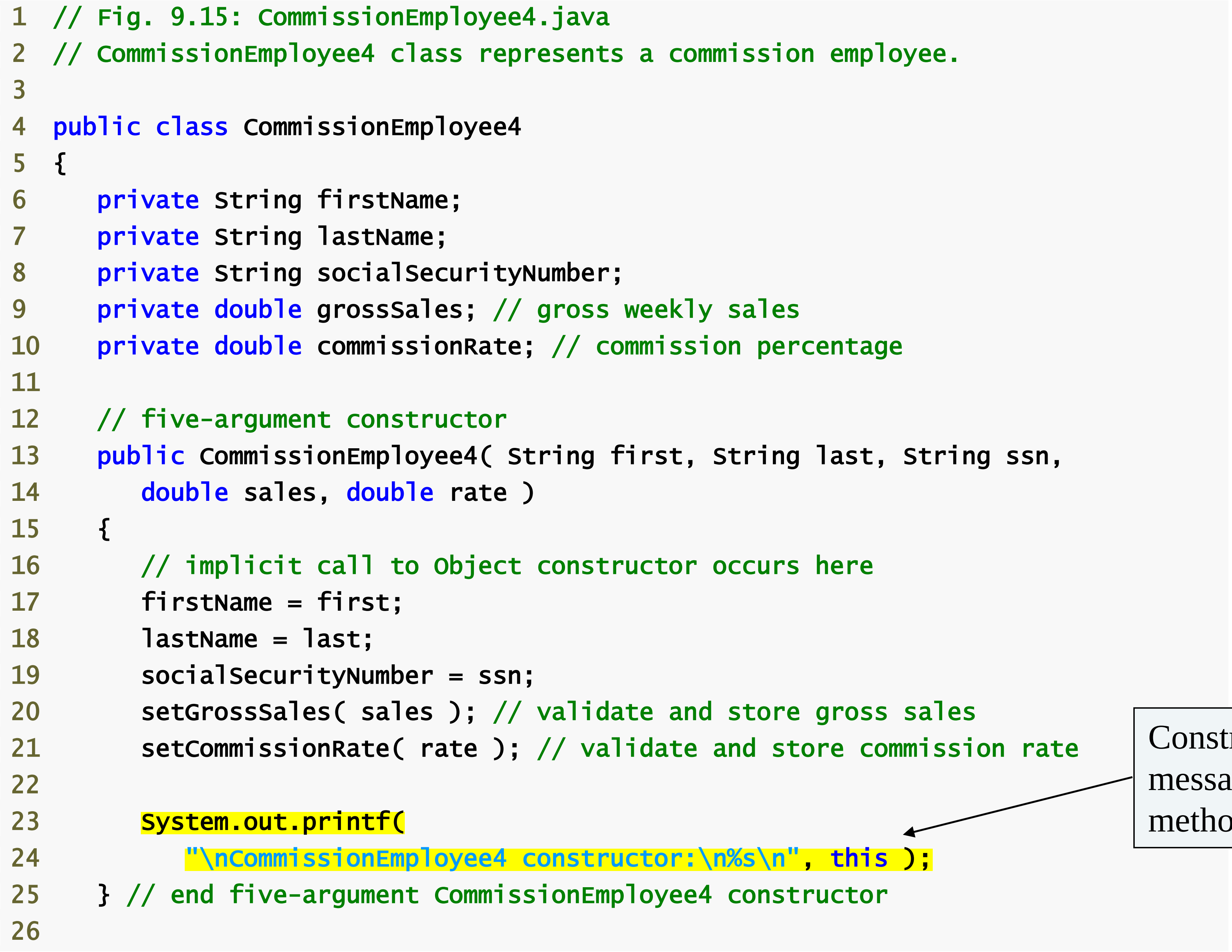
Calculate earnings



```java
   // return String representation of CommissionEmployee object
   public String toString()
   {
      return String.format( "%s: %s %s\n%s: %s\n%s: %.2f\n%s: %.2f",
         "commission employee", firstName, lastName,
         "social security number", socialSecurityNumber,
         "gross sales", grossSales,
         "commission rate", commissionRate );
   } // end method toString
} // end class CommissionEmployee
```

Override method `toString` of class `Object`

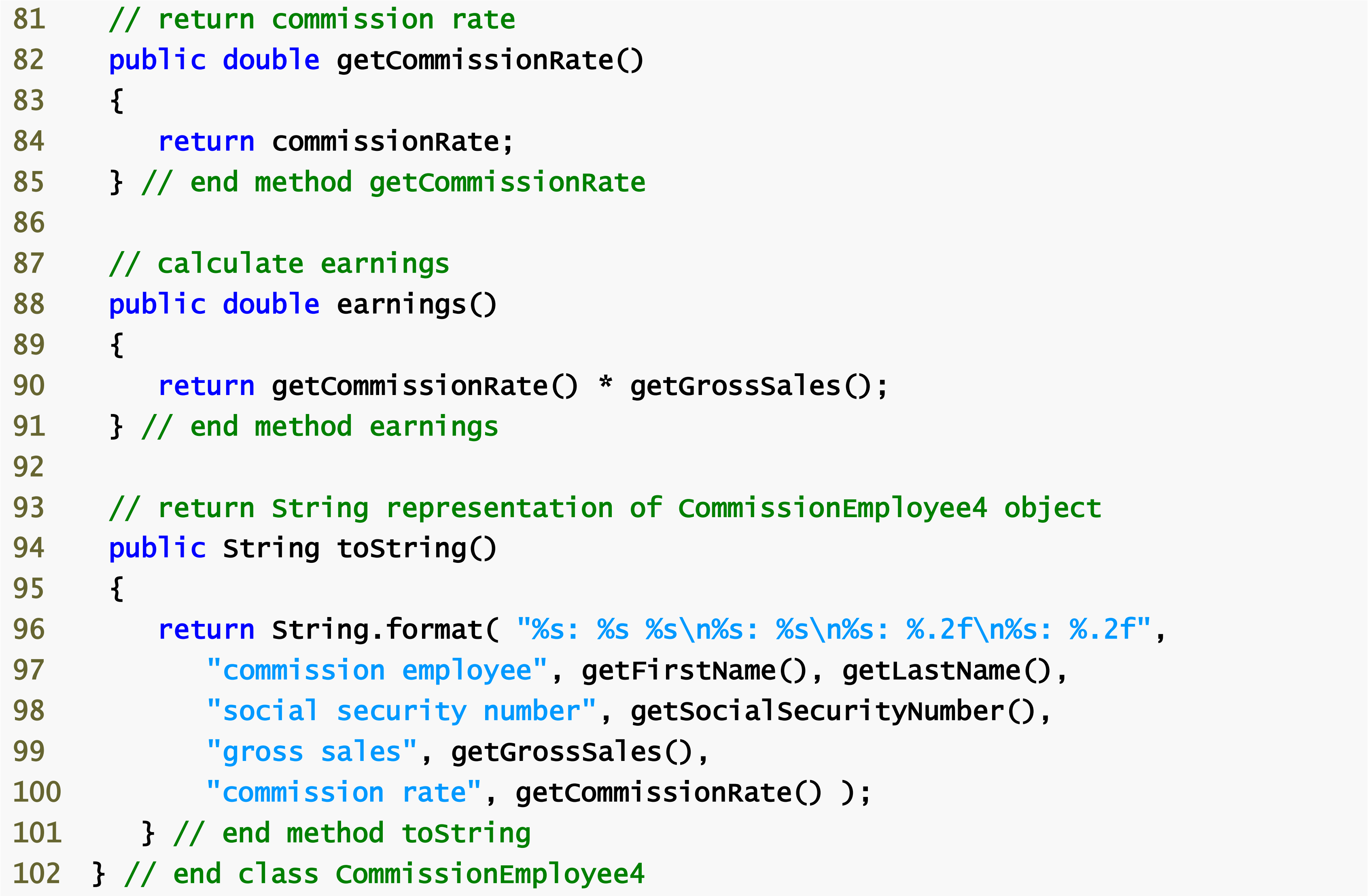
Example of inheritance from super class by invoking super constructor with arguments.



```java
// Fig. 9.15: CommissionEmployee4.java
// CommissionEmployee4 class represents a commission employee.

public class CommissionEmployee4
{
   private String firstName;
   private String lastName;
   private String socialSecurityNumber;
   private double grossSales; // gross weekly sales
   private double commissionRate; // commission percentage

   // five-argument constructor
   public CommissionEmployee4( String first, String last, String ssn,
      double sales, double rate )
   {
      // implicit call to Object constructor occurs here
      firstName = first;
      lastName = last;
      socialSecurityNumber = ssn;
      setGrossSales( sales ); // validate and store gross sales
      setCommissionRate( rate ); // validate and store commission rate

      System.out.printf(
         "\nCommissionEmployee4 constructor:\n%s\n", this );
   } // end five-argument CommissionEmployee4 constructor

```

Constructor outputs message to demonstrate method call order.

```java
   // set first name
   public void setFirstName( String first )
   {
      firstName = first;
   } // end method setFirstName

   // return first name
   public String getFirstName()
   {
      return firstName;
   } // end method getFirstName

   // set last name
   public void setLastName( String last )
   {
      lastName = last;
   } // end method setLastName

   // return last name
   public String getLastName()
   {
      return lastName;
   } // end method getLastName

   // set social security number
   public void setSocialSecurityNumber( String ssn )
   {
      socialSecurityNumber = ssn; // should validate
   } // end method setSocialSecurityNumber

   // return social security number
   public String getSocialSecurityNumber()
   {
      return socialSecurityNumber;
   } // end method getSocialSecurityNumber

   // set gross sales amount
   public void setGrossSales( double sales )
   {
      grossSales = ( sales < 0.0 ) ? 0.0 : sales;
   } // end method setGrossSales

   // return gross sales amount
   public double getGrossSales()
   {
      return grossSales;
   } // end method getGrossSales

   // set commission rate
   public void setCommissionRate( double rate )
   {
      commissionRate = ( rate > 0.0 && rate < 1.0 ) ? rate : 0.0;
   } // end method setCommissionRate

```



```
 // return commission rate
 public double getCommissionRate()
 {
    return commissionRate;
 } // end method getCommissionRate

 // calculate earnings
 public double earnings()
 {
    return getCommissionRate() * getGrossSales();
 } // end method earnings

 // return String representation of CommissionEmployee4 object
 public String toString()
 {
    return String.format( "%s: %s %s\n%s: %s\n%s: %.2f\n%s: %.2f",
       "commission employee", getFirstName(), getLastName(),
       "social security number", getSocialSecurityNumber(),
       "gross sales", getGrossSales(),
       "commission rate", getCommissionRate() );
   } // end method toString
} // end class CommissionEmployee4
```

```java
// Fig. 9.16: BasePlusCommissionEmployee5.java
// BasePlusCommissionEmployee5 class declaration.

public class BasePlusCommissionEmployee5 extends CommissionEmployee4
{
   private double baseSalary; // base salary per week

   // six-argument constructor
   public BasePlusCommissionEmployee5( String first, String last,
      String ssn, double sales, double rate, double salary )
   {
      super( first, last, ssn, sales, rate );
      setBaseSalary( salary ); // validate and store base salary

      System.out.printf(
         "\nBasePlusCommissionEmployee5 constructor:\n%s\n", this );
   } // end six-argument BasePlusCommissionEmployee5 constructor

   // set base salary
   public void setBaseSalary( double salary )
   {
      baseSalary = ( salary < 0.0 ) ? 0.0 : salary;
   } // end method setBaseSalary

```

Constructor outputs message to demonstrate method call order.

```java
   // return base salary
   public double getBaseSalary()
   {
      return baseSalary;
   } // end method getBaseSalary

   // calculate earnings
   public double earnings()
   {
      return getBaseSalary() + super.earnings();
   } // end method earnings

   // return String representation of BasePlusCommissionEmployee5
   public String toString()
   {
      return String.format( "%s %s\n%s: %.2f", "base-salaried",
         super.toString(), "base salary", getBaseSalary() );
   } // end method toString
} // end class BasePlusCommissionEmployee5
```

```java
// Fig. 9.17: ConstructorTest.java
// Display order in which superclass and subclass constructors are called.

public class ConstructorTest
{
   public static void main( String args[] )
   {
      CommissionEmployee4 employee1 = new CommissionEmployee4(
         "Bob", "Lewis", "333-33-3333", 5000, .04 );

      System.out.println();
      BasePlusCommissionEmployee5 employee2 =
         new BasePlusCommissionEmployee5(
         "Lisa", "Jones", "555-55-5555", 2000, .06, 800 );

      System.out.println();
      BasePlusCommissionEmployee5 employee3 =
         new BasePlusCommissionEmployee5(
         "Mark", "Sands", "888-88-8888", 8000, .15, 2000 );
   } // end main
} // end class ConstructorTest
```

Instantiate `CommissionEmployee4` object

Instantiate two `BasePlusCommissionEmployee5` objects to demonstrate order of subclass and superclass constructor method calls.

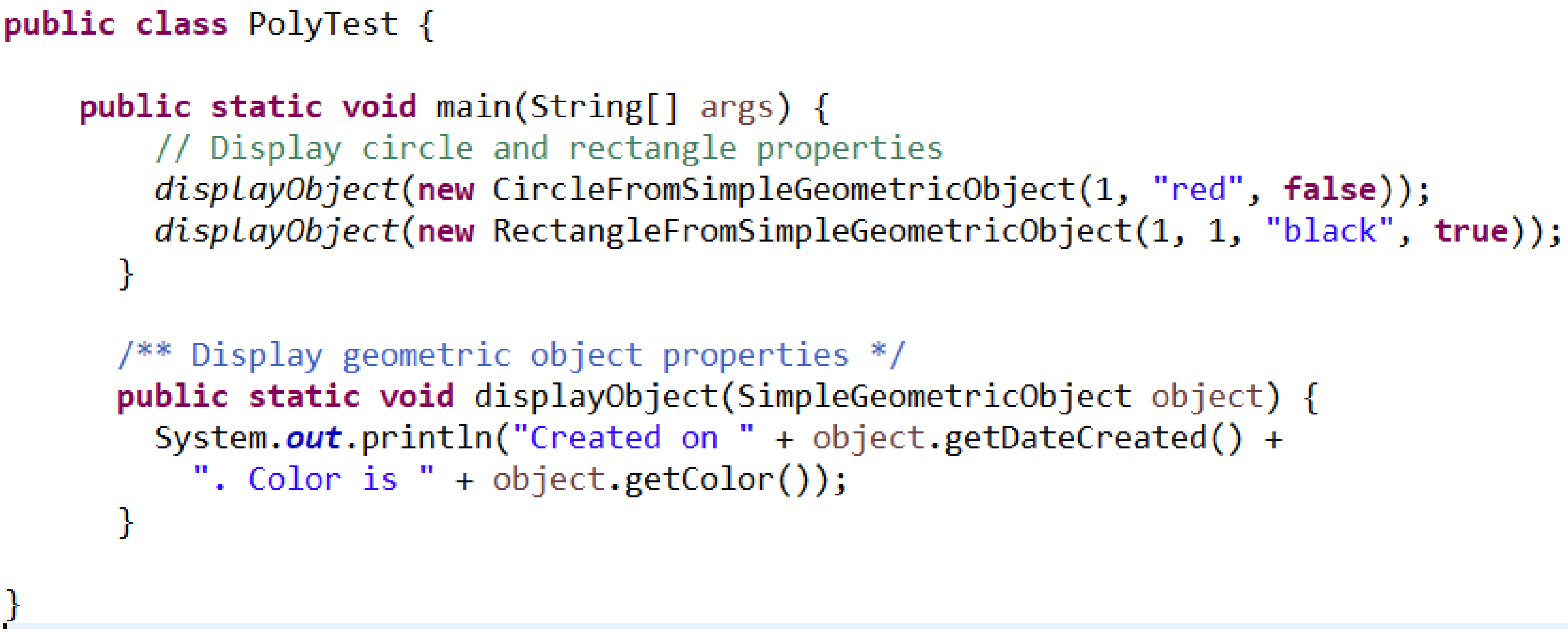
```
CommissionEmployee4 constructor:
commission employee: Bob Lewis
social security number: 333-33-3333
gross sales: 5000.00
commission rate: 0.04

CommissionEmployee4 constructor:
base-salaried commission employee: Lisa Jones
social security number: 555-55-5555
gross sales: 2000.00
commission rate: 0.06
base salary: 0.00

BasePlusCommissionEmployee5 constructor:
base-salaried commission employee: Lisa Jones
social security number: 555-55-5555
gross sales: 2000.00
commission rate: 0.06
base salary: 800.00


CommissionEmployee4 constructor:
base-salaried commission employee: Mark Sands
social security number: 888-88-8888
gross sales: 8000.00
commission rate: 0.15
base salary: 0.00

BasePlusCommissionEmployee5 constructor:
base-salaried commission employee: Mark Sands
social security number: 888-88-8888
gross sales: 8000.00
commission rate: 0.15
base salary: 2000.00
```

Subclass `BasePlusCommissionEmployee5` constructor body executes after superclass `CommissionEmployee4`'s constructor finishes execution.

Polymorphism

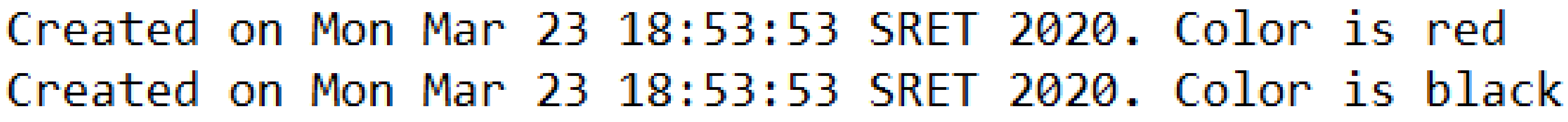
Polymorphism means that a variable of a supertype can refer to a subtype object.

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**.



```java
public class PolyTest {

    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());
    }

}
```

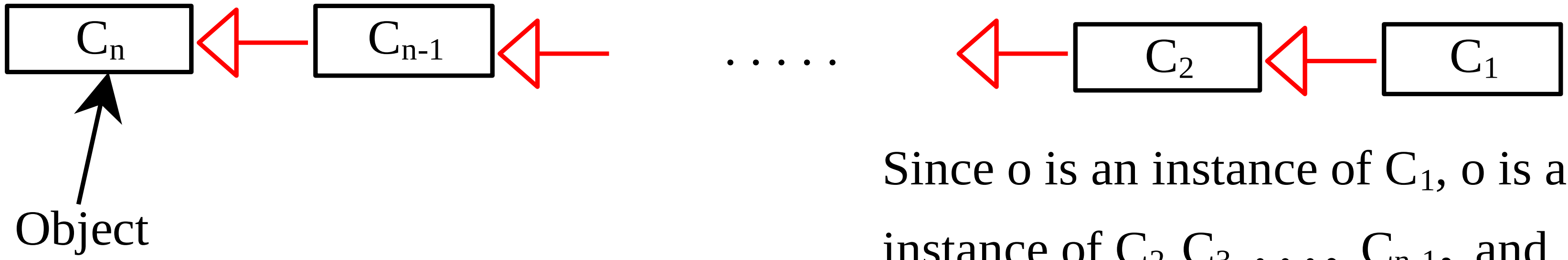
Output:



```
Created on Mon Mar 23 18:53:53 SRET 2020. Color is red
Created on Mon Mar 23 18:53:53 SRET 2020. Color is black
```

Dynamic Binding

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



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

Polymorphism, Dynamic Binding and Generic Programming

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

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

class GraduateStudent extends Student
{
}

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

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

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

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

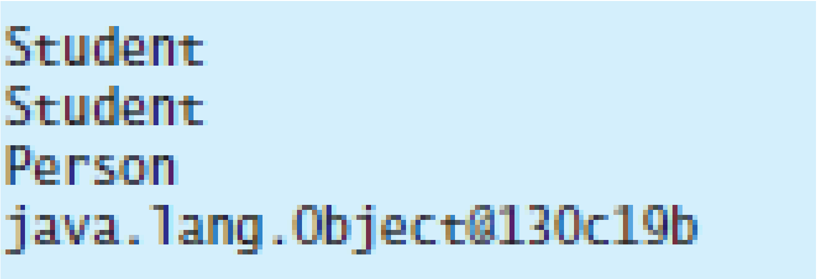
When the method m(Object x) is executed, the argument x’s toString method is invoked. x may be an instance of GraduateStudent, Student, Person, or Object. Classes GraduateStudent, Student, Person, and Object have their own implementation of the toString method. Which implementation is used will be determined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

Generic Programming

```
public class PolymorphismDemo {
  public static void main(String[]
args) {
    m(new GraduateStudent());
    m(new Student());
    m(new Person());
    m(new Object());
  }
   public static void m(Object x) {
    System.out.println(x.toString());
  }
}

class GraduateStudent extends Student {
}

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

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

Polymorphism allows methods to be used generically for a wide range of object arguments. This is known as generic programming. If a method's parameter type is a superclass (e.g., Object), you may pass an object to this method of any of the parameter's subclasses (e.g., Student or String). When an object (e.g., a Student object or a String object) is used in the method, the particular implementation of the method of the object that is invoked (e.g., toString) is determined dynamically.

Output



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

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.

Summary of the Allowed Assignments Between Superclass and Subclass Variables

- Superclass and subclass assignment rules
  - Assigning a superclass reference to a superclass variable is straightforward
  - Assigning a subclass reference to a subclass variable is straightforward
  - Assigning a subclass reference to a superclass variable is safe because of the *is-a* relationship
    - Referring to subclass-only members through superclass variables is a compilation error
  - Assigning a superclass reference to a subclass variable is a compilation error
    - Downcasting can be used to avoid this error

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.

Common Programming Error

- Assigning a superclass variable to a subclass variable (without an explicit cast) is a compilation error.

- When downcasting an object, a `ClassCastException` occurs, if at execution time the object does not have an *is-a* relationship with the type specified in the cast operator. An object can be cast only to its own type or to the type of one of its superclasses.

Why Casting Is Necessary?

Suppose you want to assign the object reference o to a variable of the Student type using the following statement:

```
Student b = o;
```

A compile error would occur. Why does the statement **Object o = new Student()** work and the statement **Student b = o** doesn’t? This is because a Student object is always an instance of Object, but an Object is not necessarily an instance of Student. Even though you can see that o is really a Student object, the compiler is not so clever to know it. To tell the compiler that o is a Student object, use an explicit casting. The syntax is similar to the one used for casting among primitive data types. Enclose the target object type in parentheses and place it before the object to be cast, as follows:

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

Casting from Superclass to Subclass

Explicit casting must be used when casting an object from a superclass to a subclass. This type of casting may not always succeed.

```
Apple x = (Apple)fruit;

Orange x = (Orange)fruit;
```

The instanceof Operator

Use the instanceof operator to test whether an object is an instance of a class:

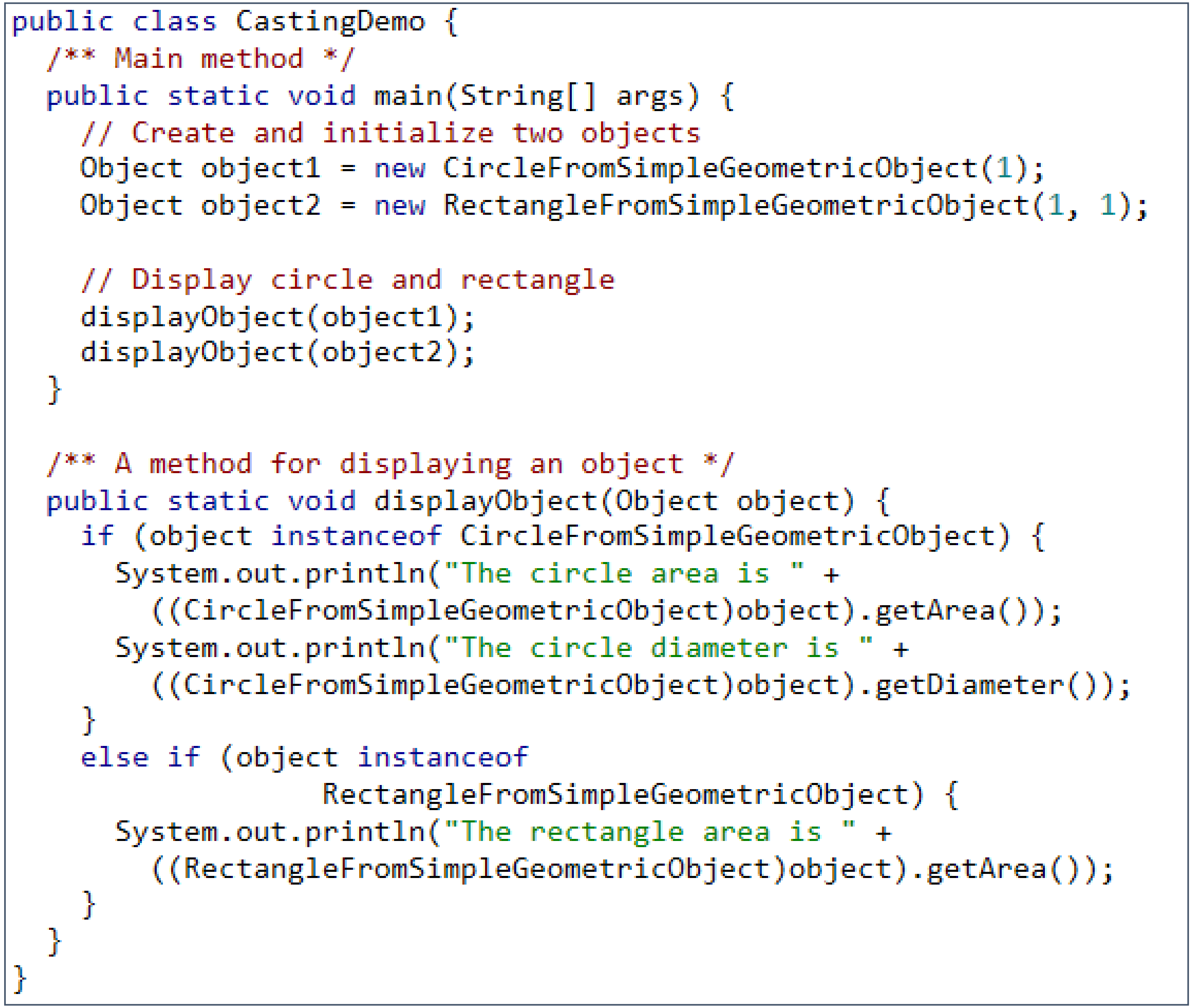
```
Object myObject = new Circle();
... // Some lines of code
/** Perform casting if myObject is an instance
  of Circle */
if (myObject instanceof Circle) {
   System.out.println("The circle diameter is "
  +
     ((Circle)myObject).getDiameter());
   ...
}
```

TIP

To help understand casting, you may also consider the analogy of fruit, apple, and orange with the Fruit class as the superclass for Apple and Orange. An apple is a fruit, so you can always safely assign an instance of Apple to a variable for Fruit. However, a fruit is not necessarily an apple, so you have to use explicit casting to assign an instance of Fruit to a variable of Apple.

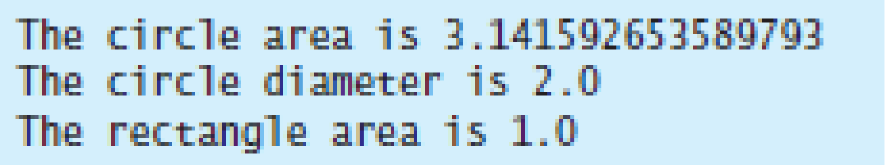
Example: Demonstrating Polymorphism and Casting

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



```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());
    }
  }
}
```

Output

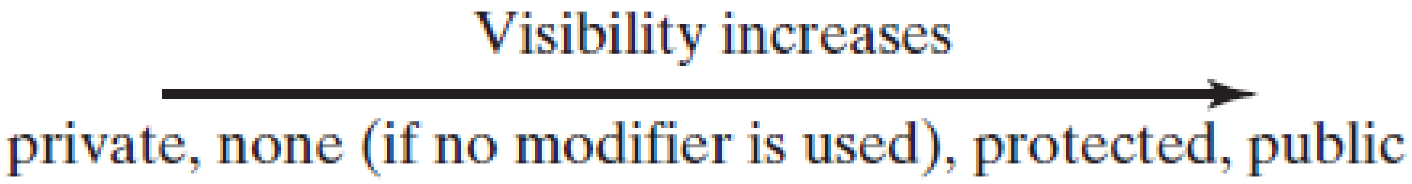


```
The circle area is 3.141592653589793
The circle diameter is 2.0
The rectangle area is 1.0
```

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



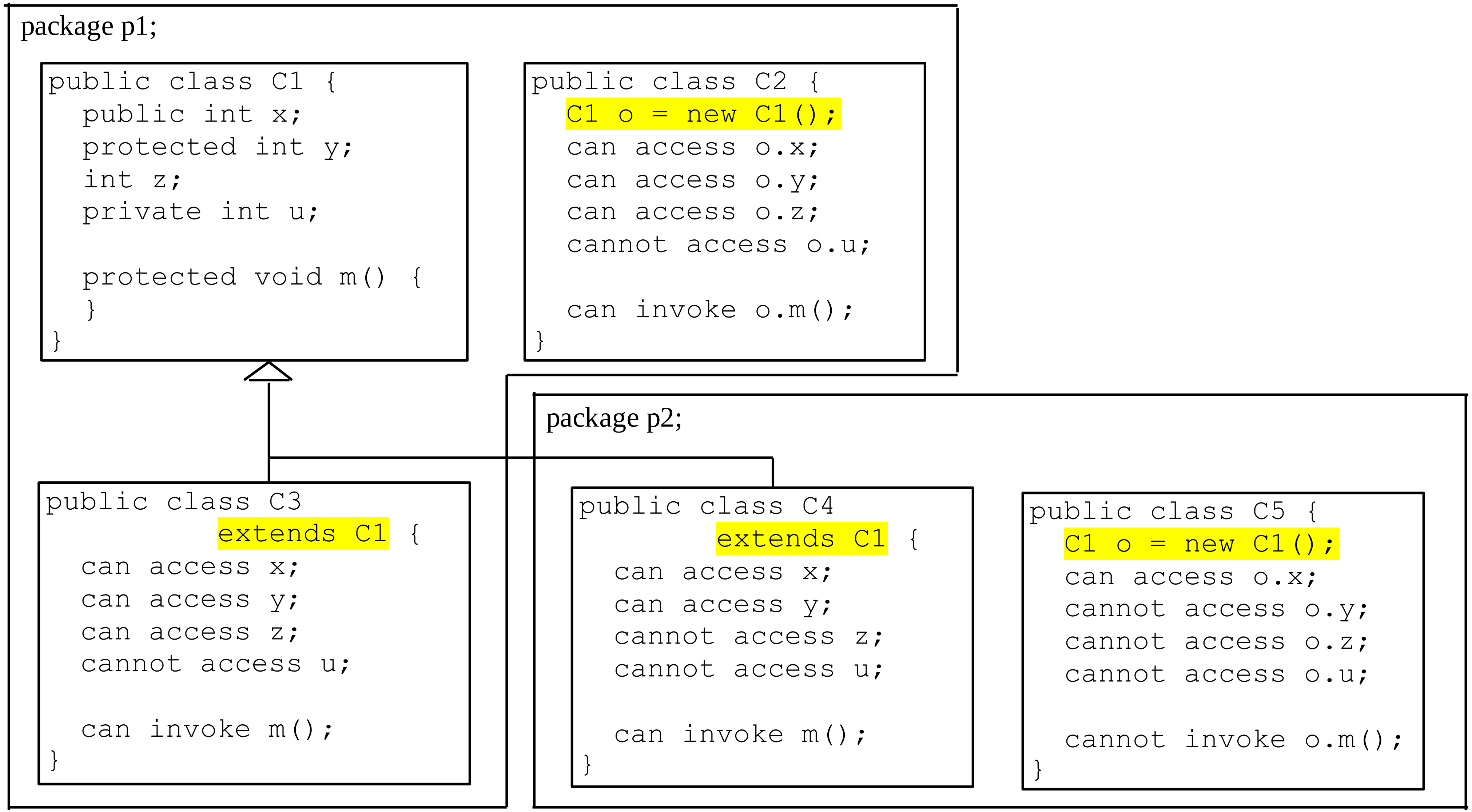
Visibility increases

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.

NOTE

The modifiers are used on classes and class members (data and methods), except that the final modifier can also be used on local variables in a method. A final local variable is a constant inside a method.

The `final` Modifier

- ❑The `final` class cannot be extended:

```
    final class Math {
      ...
    }
```

- ❑The `final` variable is a constant:

```
    final static double PI = 3.14159;
```

- ❑The `final` method cannot be overridden by its subclasses.