CANADIAN UNIVERSITY OF BANGLADESH

Inspiring Applied Knowledge

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

Lecture-8: Objects & Classes (Part II)

Prerequisite: CSE 1101
Semester: Summer 2024

Object-Oriented Problem Solving

Objects & Classes (Part II)

Based on Chapter 9 of “Introduction to Java Programming” by Y. Daniel Liang.

Outline

- Static Variables, Constants, and Methods (9.7)
- Visibility Modifiers (9.8)
- Data Field Encapsulation (9.9)
- Passing Objects to Methods (9.10)
- The Scope of Variables (9.13)

Static Variables, Constants, and Methods

- All variables declared in the data fields of the previous examples are called *instance variables*.
- An *instance variable* is tied to a specific instance of the class.
  - It is not shared among objects of the same class.
  - It has independent memory storage for each instance.
- In the following example, the *radius* of the first object "*circle1*" is independent of the *radius* of the second object "*circle2*":

```
Circle circle1 = new Circle();
Circle circle2 = new Circle(5);
```

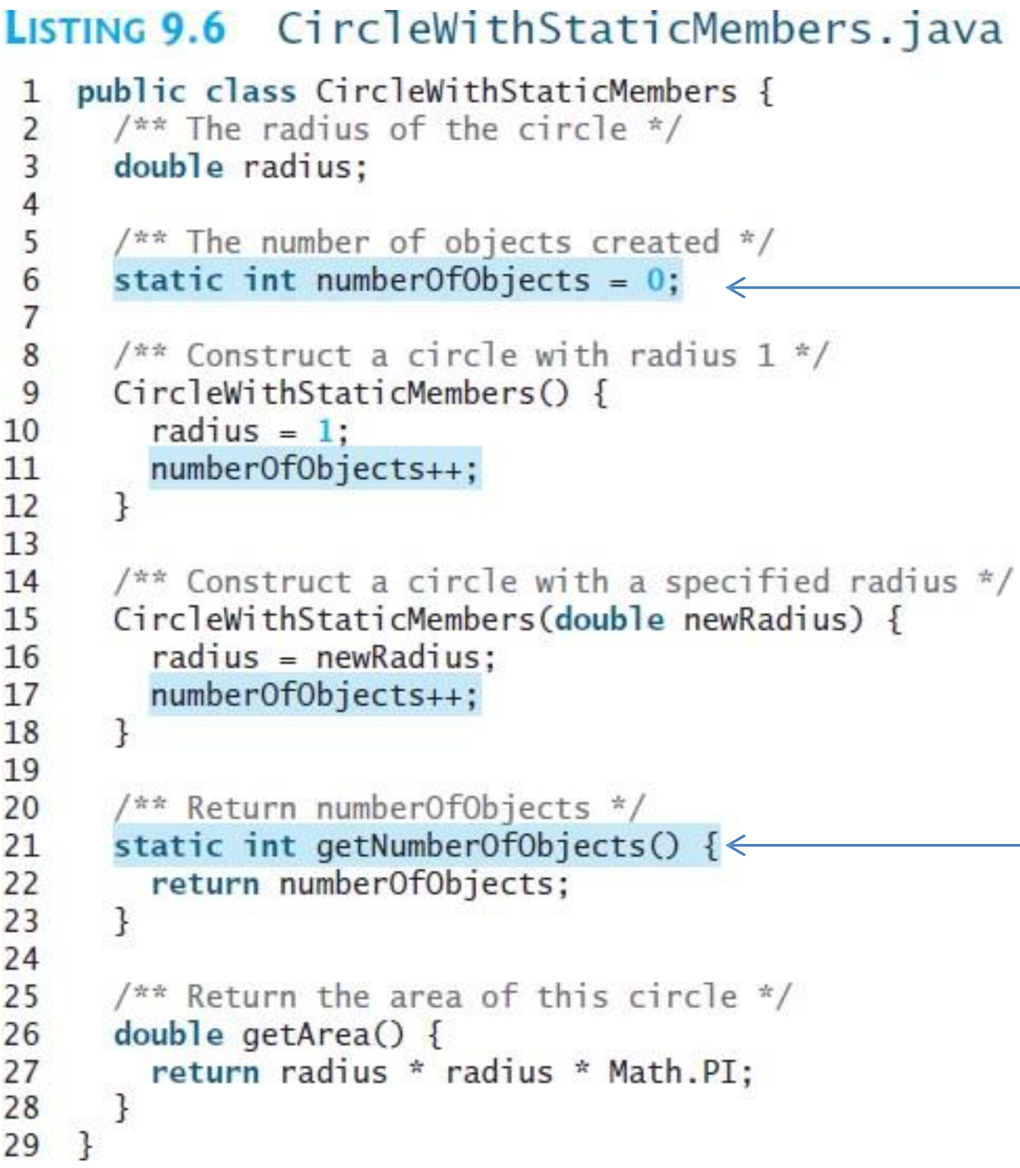
Static Variables, Constants, and Methods (Cont.)

- *Static variables*, also known as *class variables*, store values for the variables in a common memory location.
  - A *static variable* is used when it is wanted that all instances of the class to share data.
  - If one instance of the class changes the value of a static variables, all instances of the same class are affected.
- Static methods can be called without creating an instance of the class.

Static Variables, Constants, and Methods (Cont.)

- To declare a static variable or define a static method, put the modifier *static* in the variable or method declaration.
- Since constants in a class are shared by all objects of the class, they should be declared static.
  - *final static double PI = 3.14159265358979323846;*
- Static variables and methods can be accessed from a reference variable or from their class name.

Example

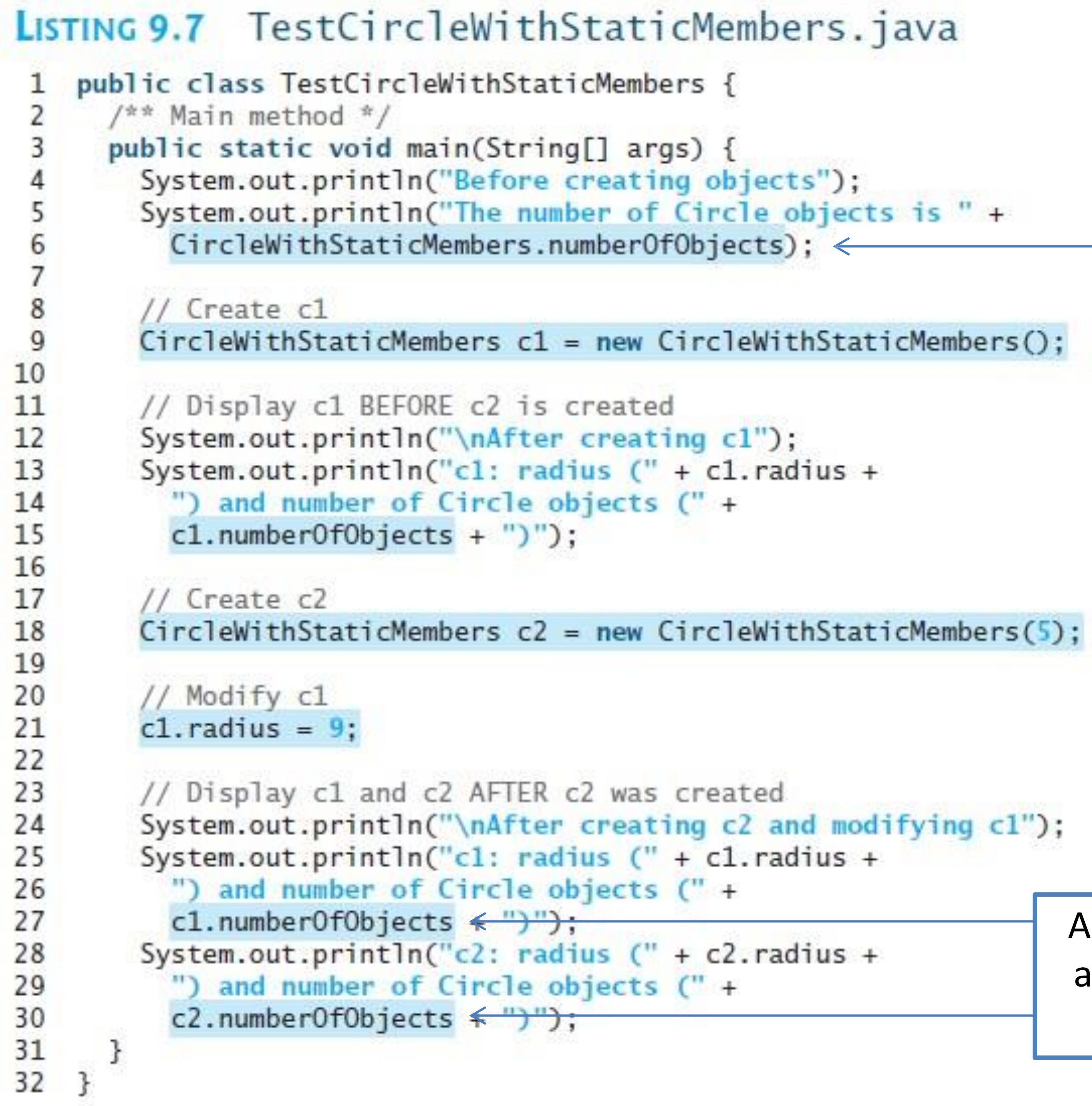


LISTING 9.6 CircleWithStaticMembers.java

```java
public class CircleWithStaticMembers {
  /** The radius of the circle */
  double radius;

  /** The number of objects created */
  static int numberOfObjects = 0;

  /** Construct a circle with radius 1 */
  CircleWithStaticMembers() {
    radius = 1;
    numberOfObjects++;
  }

  /** Construct a circle with a specified radius */
  CircleWithStaticMembers(double newRadius) {
    radius = newRadius;
    numberOfObjects++;
  }

  /** Return numberOfObjects */
  static int getNumberOfObjects() {
    return numberOfObjects;
  }

  /** Return the area of this circle */
  double getArea() {
    return radius * radius * Math.PI;
  }
}
```

static variable

static method



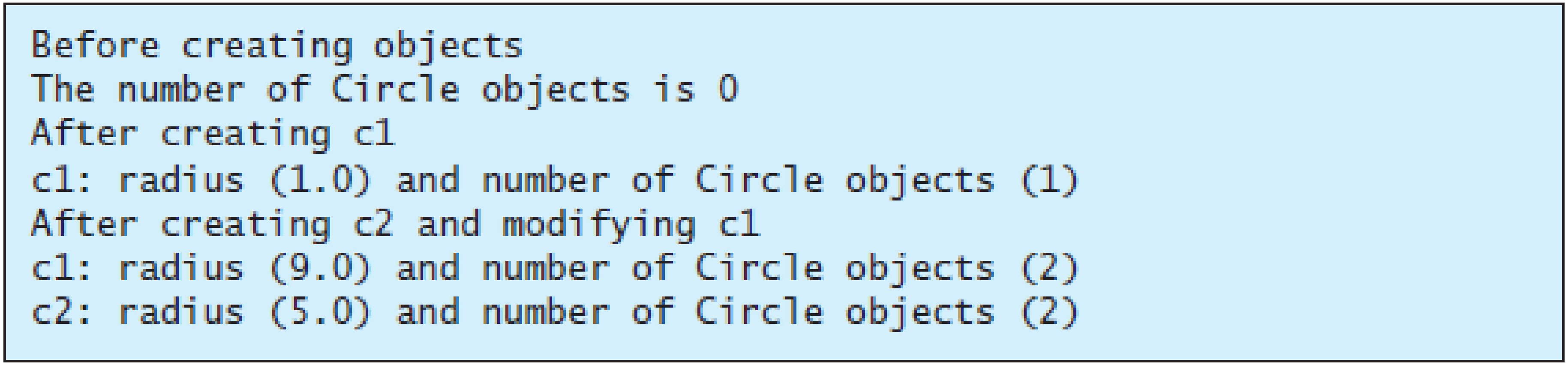
LISTING 9.7 TestCircleWithStaticMembers.java

```java
public class TestCircleWithStaticMembers {
  /** Main method */
  public static void main(String[] args) {
    System.out.println("Before creating objects");
    System.out.println("The number of Circle objects is " +
      CircleWithStaticMembers.numberOfObjects);

    // Create c1
    CircleWithStaticMembers c1 = new CircleWithStaticMembers();

    // Display c1 BEFORE c2 is created
    System.out.println("\nAfter creating c1");
    System.out.println("c1: radius (" + c1.radius +
      ") and number of Circle objects (" +
      c1.numberOfObjects + ")");

    // Create c2
    CircleWithStaticMembers c2 = new CircleWithStaticMembers(5);

    // Modify c1
    c1.radius = 9;

    // Display c1 and c2 AFTER c2 was created
    System.out.println("\nAfter creating c2 and modifying c1");
    System.out.println("c1: radius (" + c1.radius +
      ") and number of Circle objects (" +
      c1.numberOfObjects + ")");
    System.out.println("c2: radius (" + c2.radius +
      ") and number of Circle objects (" +
      c2.numberOfObjects + ")");
  }
}
```

A static variable can be accessed via its class name.

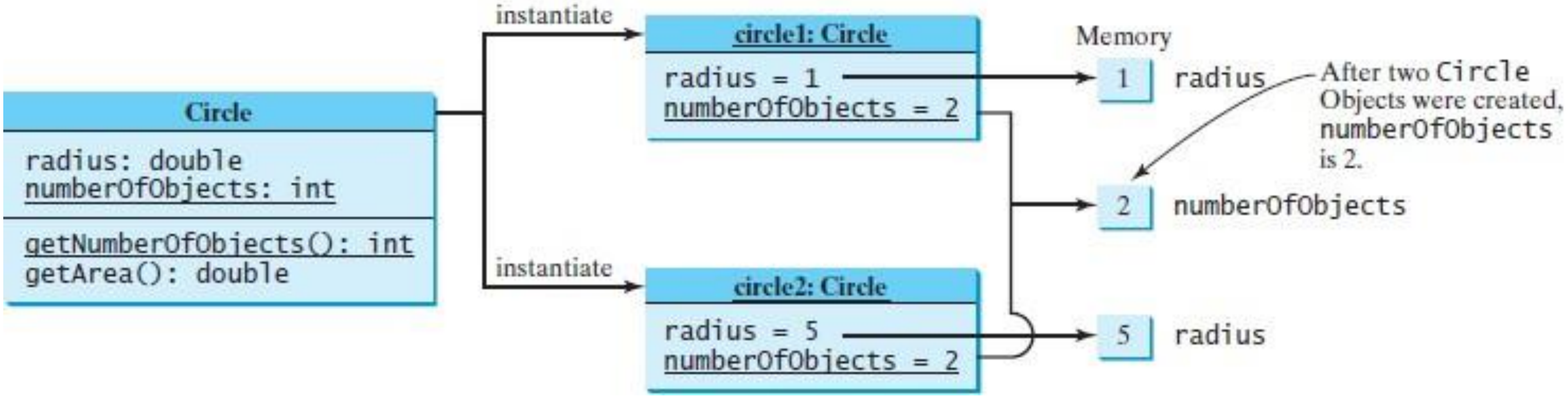
A static variable can also be accessed via objects of the class.

Example (Output)



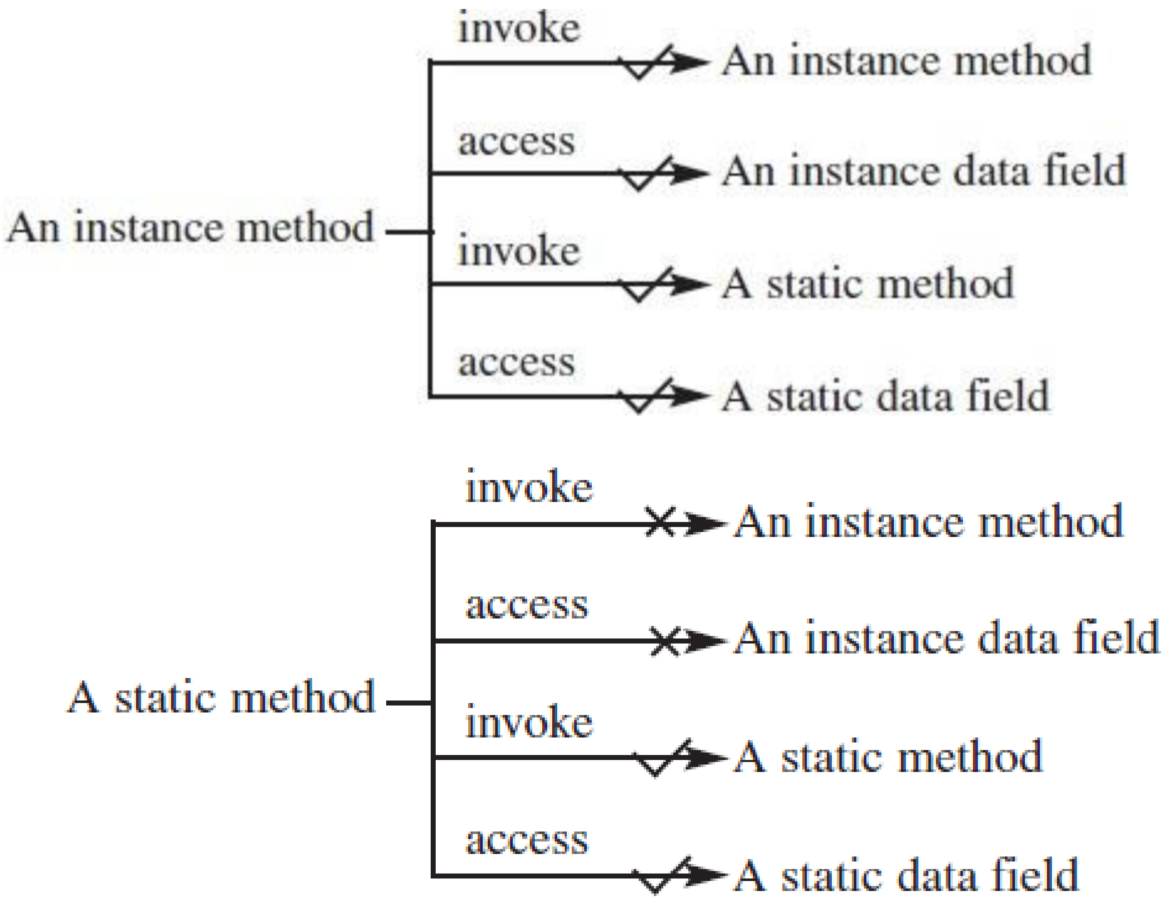
```
Before creating objects
The number of Circle objects is 0
After creating c1
c1: radius (1.0) and number of Circle objects (1)
After creating c2 and modifying c1
c1: radius (9.0) and number of Circle objects (2)
c2: radius (5.0) and number of Circle objects (2)
```

UML Class Diagram: Circle with Static Members



Static members are underlined in UML class diagrams.

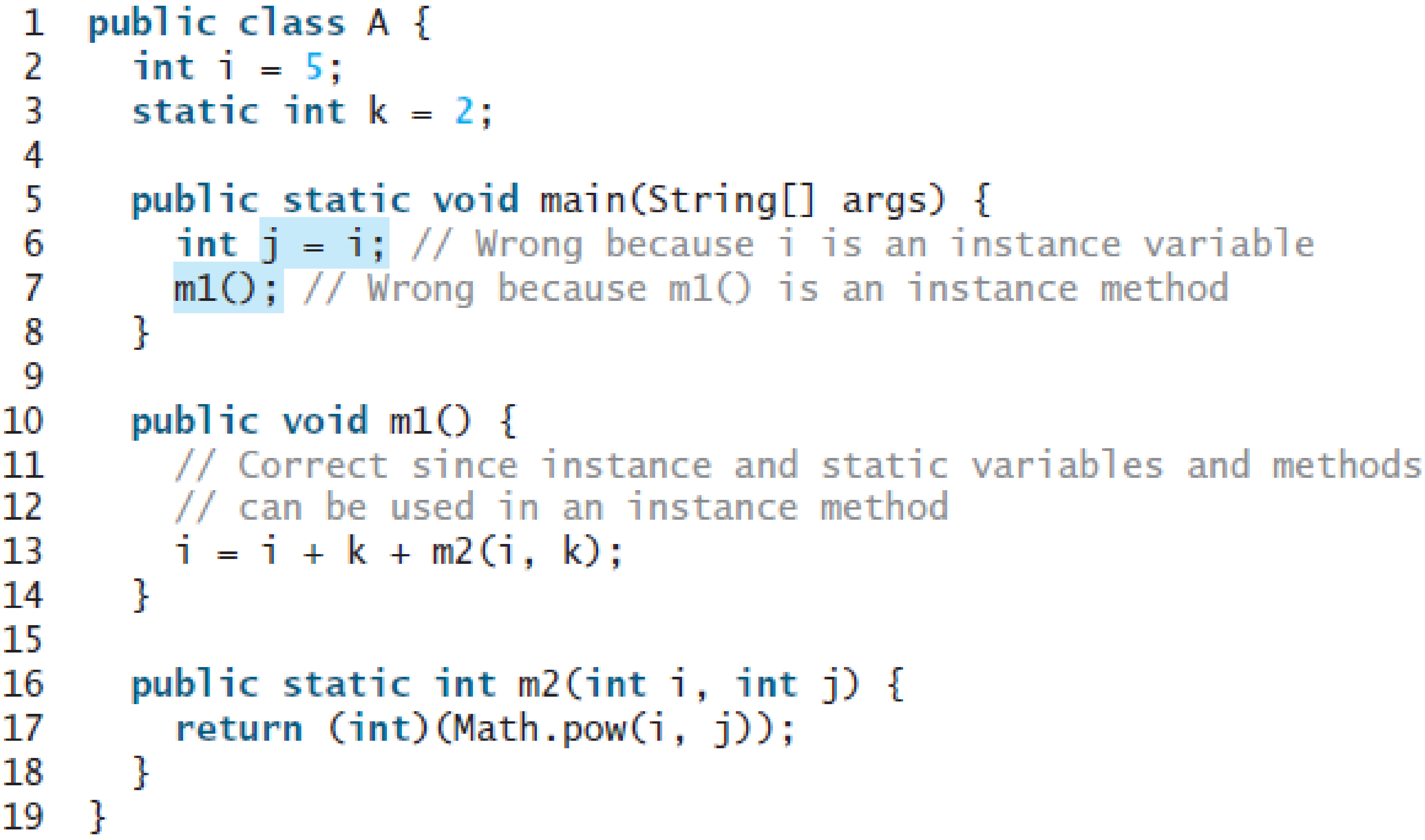
Relationship between Static and Instance Members



An instance method
- invoke ✓ An instance method
- access ✓ An instance data field
- invoke ✓ A static method
- access ✓ A static data field

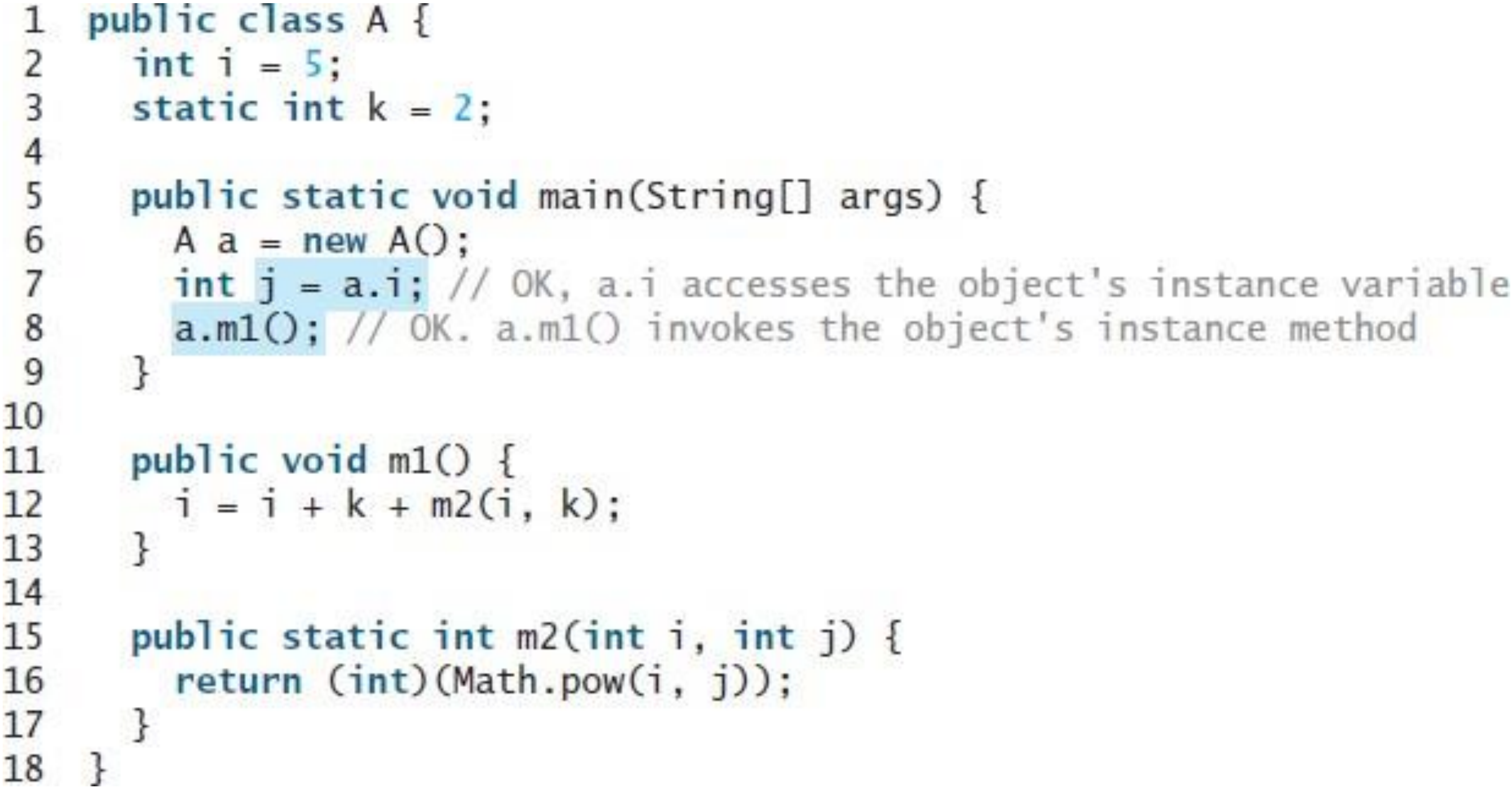
A static method
- invoke ✗ An instance method
- access ✗ An instance data field
- invoke ✓ A static method
- access ✓ A static data field

Relationship between Static and Instance Members (Example 1)



```java
public class A {
  int i = 5;
  static int k = 2;

  public static void main(String[] args) {
    int j = i; // Wrong because i is an instance variable
    m1(); // Wrong because m1() is an instance method
  }

  public void m1() {
    // Correct since instance and static variables and methods
    // can be used in an instance method
    i = i + k + m2(i, k);
  }

  public static int m2(int i, int j) {
    return (int)(Math.pow(i, j));
  }
}
```

Relationship between Static and Instance Members (Example 2)



```java
public class A {
  int i = 5;
  static int k = 2;

  public static void main(String[] args) {
    A a = new A();
    int j = a.i; // OK, a.i accesses the object's instance variable
    a.m1(); // OK. a.m1() invokes the object's instance method
  }

  public void m1() {
    i = i + k + m2(i, k);
  }

  public static int m2(int i, int j) {
    return (int)(Math.pow(i, j));
  }
}
```

Instance or Static?

- How to decide whether a variable or method should be an instance one or static one?
  - A variable or method that is *dependent* on a specific instance of the class should be an *instance* variable or method.
    - Example: *radius* and *getArea* of the *Circle* class; each circle has its own radius and area.
  - A variable or method that is *not dependent* on a specific instance of the class should be a *static* variable or method.
    - Example: *numberOfObjects* of the Circle class; all circles should share this value.

Visibility Modifiers

- *Visibility modifiers* can be used to specify the *visibility* of a class and its members.
- A *visibility modifier* specifies how data fields and methods in a class can be accessed from <u>outside the class</u>.
  - There is no restriction on accessing data fields and methods from inside the class.

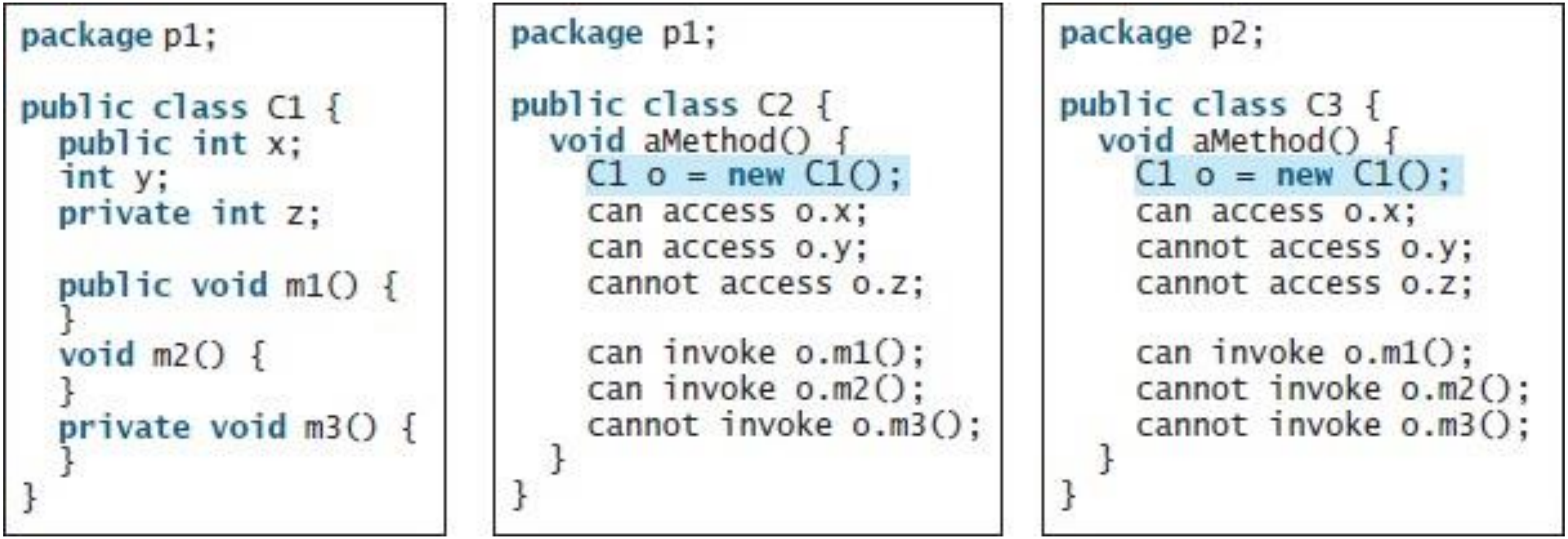
Visibility Modifiers: The Default

- If no visibility modifier is used, then by *default* the classes, methods, and data fields are <u>accessible by any class in the same package</u>.
  - This is known as *package-private* or *package-access*.
- *Packages* are used to organize classes. To do so, you need to add the following statement as the first statement in the program.
  - *package packageName;*
- If a class is defined without the package statement, it is said to be placed in the default package.

Visibility Modifiers: Public and Private

- The *public* modifier can be used for classes, methods and data fields to denote that they can be accessed from any other classes.
- The *private* modifier makes methods and data fields accessible only from within its own class.

Visibility Modifiers: Methods and Data Fields Example



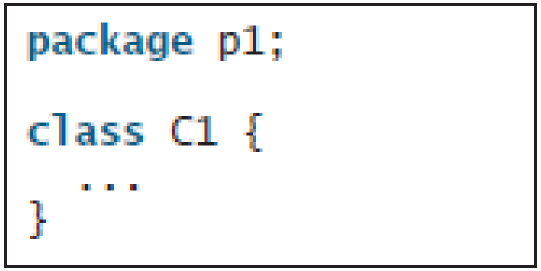
```
package p1;

public class C1 {
  public int x;
  int y;
  private int z;

  public void m1() {
  }
  void m2() {
  }
  private void m3() {
  }
}
```

```
package p1;

public class C2 {
  void aMethod() {
    C1 o = new C1();
    can access o.x;
    can access o.y;
    cannot access o.z;

    can invoke o.m1();
    can invoke o.m2();
    cannot invoke o.m3();
  }
}
```

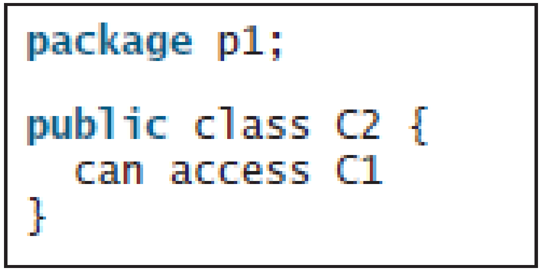
```
package p2;

public class C3 {
  void aMethod() {
    C1 o = new C1();
    can access o.x;
    cannot access o.y;
    cannot access o.z;

    can invoke o.m1();
    cannot invoke o.m2();
    cannot invoke o.m3();
  }
}
```

- The *private* modifier restricts access to its defining class.
- The *default* modifier restricts access to a package.
- The *public* modifier enables unrestricted access.

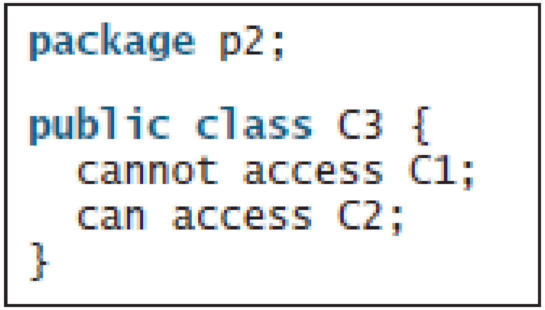
Visibility Modifiers: Classes Example



```
package p1;

class C1 {
  ...
}
```

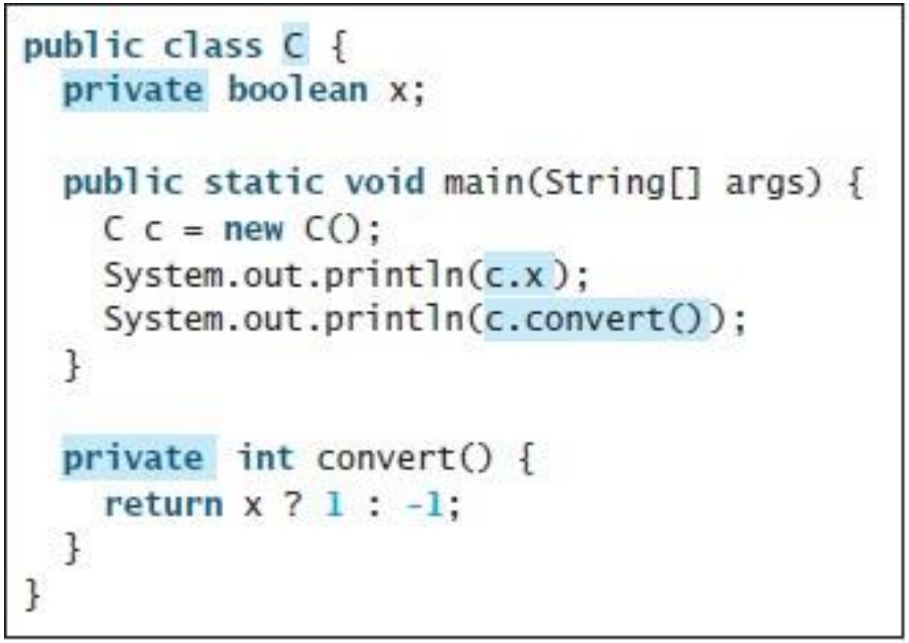


```
package p1;

public class C2 {
  can access C1
}
```



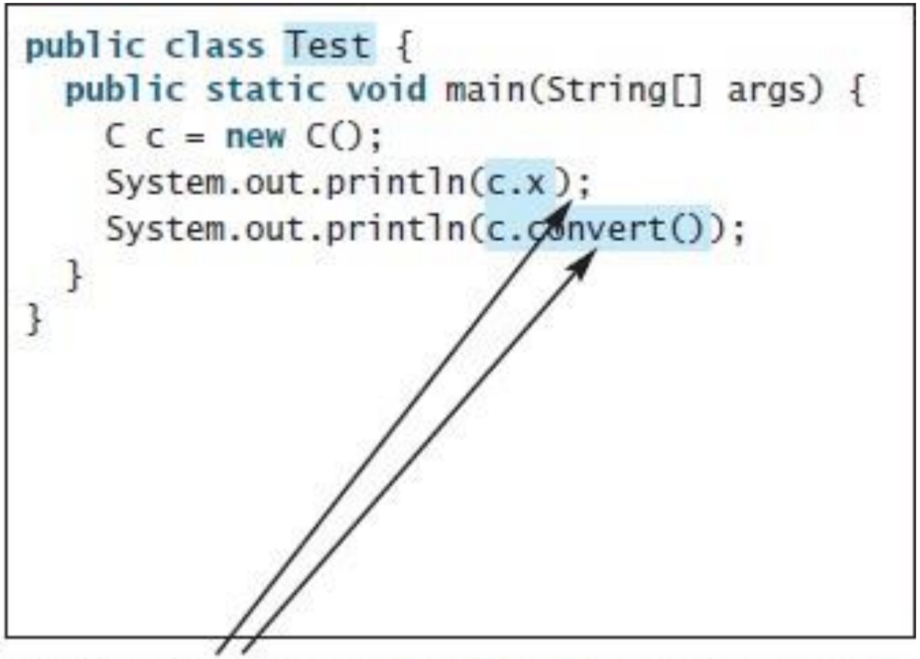
```
package p2;

public class C3 {
  cannot access C1;
  can access C2;
}
```

Visibility Modifiers: Another Example



```
public class C {
  private boolean x;

  public static void main(String[] args) {
    C c = new C();
    System.out.println(c.x);
    System.out.println(c.convert());
  }

  private int convert() {
    return x ? 1 : -1;
  }
}
```

(a) This is okay because object c is used inside the class C.



```
public class Test {
  public static void main(String[] args) {
    C c = new C();
    System.out.println(c.x);
    System.out.println(c.convert());
  }
}
```

(b) This is wrong because x and convert are private in class C.

Visibility Modifiers: Comments

- The *private* modifier applies only to the members of a class.
- The *public* modifier can apply to a class or members of a class.
- Using the modifiers *public* and *private* on local variables would cause a compile error.

Data Field Encapsulation

- It is not a good practice to allow data fields to be directly modified.
  - Data may be tampered with.
  - The class becomes difficult to maintain and vulnerable to bugs.
- To prevent direct modifications of data fields, you should declare the data fields *private*.
  - This is known as *data field encapsulation*.

Data Field Encapsulation (Cont.)

- A private data field cannot be accessed by an object from outside the class that defines the private field.
- However, a client often needs to retrieve and modify a data field.
- To make a private data field accessible:
  - Provide a *getter (accessor)* method to return its value.
  - Provide a *setter (mutator)* method set a new value to it.

Data Field Encapsulation (Cont.)

- A *getter* method has the following signature:

  public returnType getPropertyName()

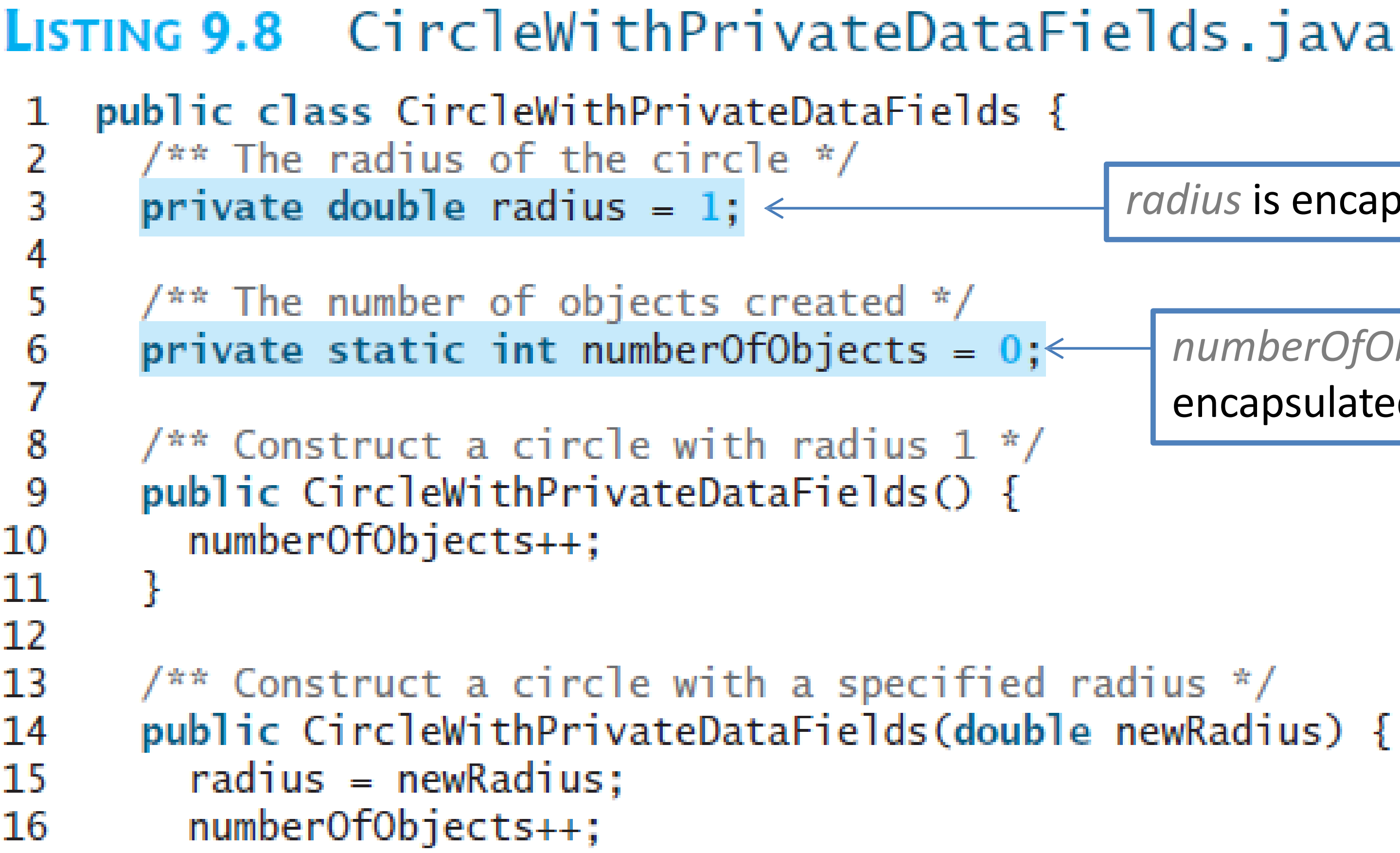
  - If the *returnType* is *boolean*, the *get* method is defined as follows by convention:

  public boolean isProperyName()

- A set method has the following signature:

  public void setPropertyName(dataType propertyValue)

Example



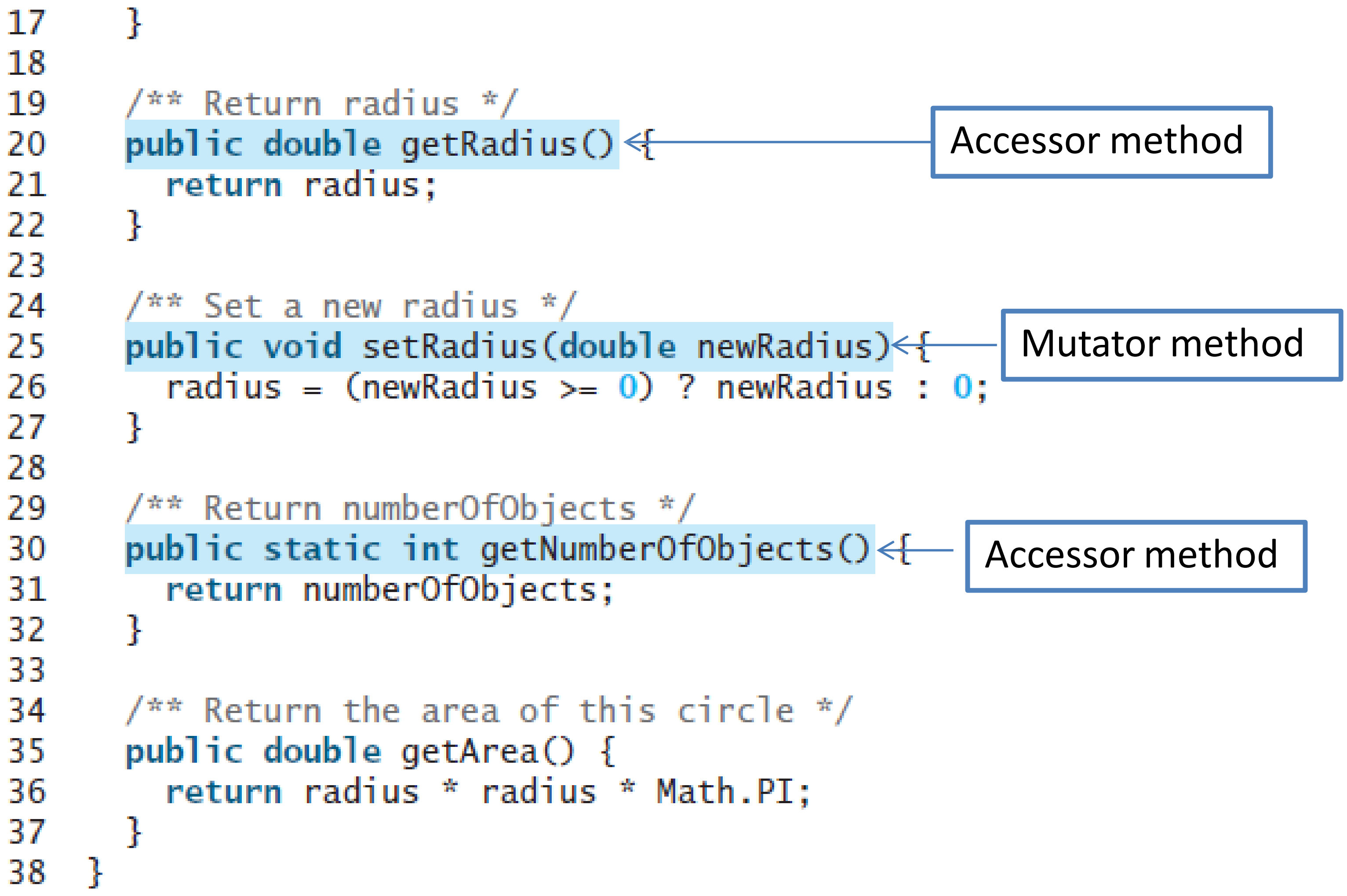
LISTING 9.8 CircleWithPrivateDataFields.java

```
public class CircleWithPrivateDataFields {
  /** The radius of the circle */
  private double radius = 1;

  /** The number of objects created */
  private static int numberOfObjects = 0;

  /** Construct a circle with radius 1 */
  public CircleWithPrivateDataFields() {
    numberOfObjects++;
  }

  /** Construct a circle with a specified radius */
  public CircleWithPrivateDataFields(double newRadius) {
    radius = newRadius;
    numberOfObjects++;
```

radius is encapsulated

numberOfObjects is encapsulated

Example (Cont.)



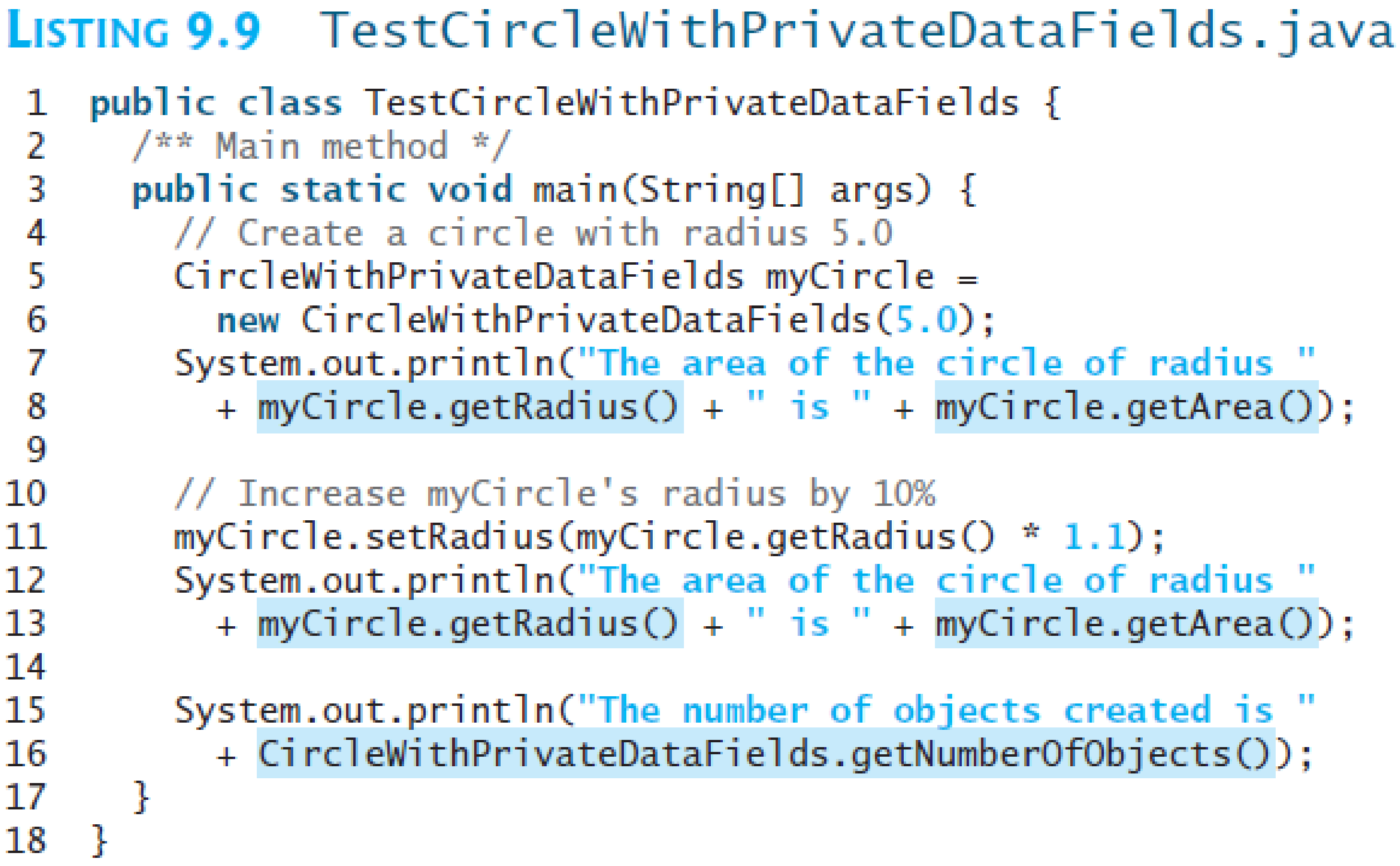
```java
  }

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

  /** Set a new radius */
  public void setRadius(double newRadius) {
    radius = (newRadius >= 0) ? newRadius : 0;
  }

  /** Return numberOfObjects */
  public static int getNumberOfObjects() {
    return numberOfObjects;
  }

  /** Return the area of this circle */
  public double getArea() {
    return radius * radius * Math.PI;
  }
}
```

Accessor method

Mutator method

Accessor method

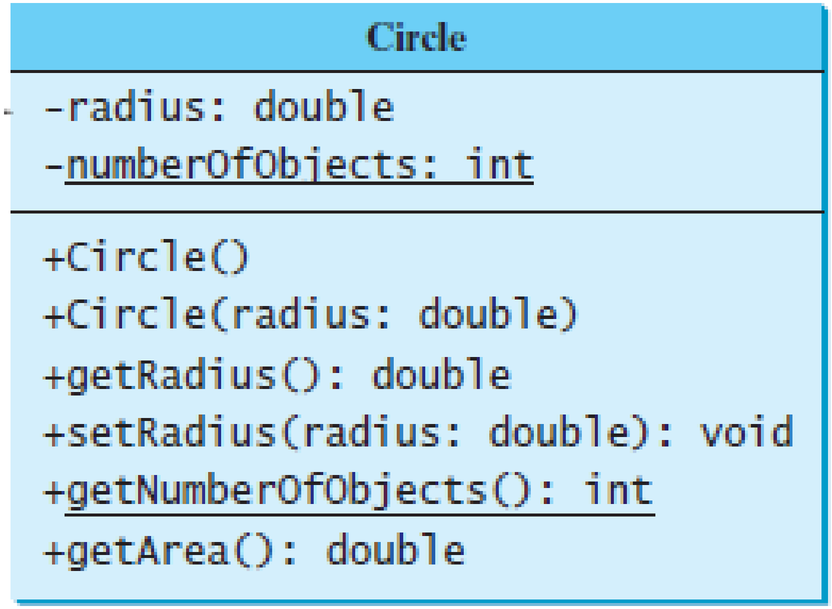
Example (Cont.)



LISTING 9.9 TestCircleWithPrivateDataFields.java

```java
public class TestCircleWithPrivateDataFields {
  /** Main method */
  public static void main(String[] args) {
    // Create a circle with radius 5.0
    CircleWithPrivateDataFields myCircle =
      new CircleWithPrivateDataFields(5.0);
    System.out.println("The area of the circle of radius "
      + myCircle.getRadius() + " is " + myCircle.getArea());

    // Increase myCircle's radius by 10%
    myCircle.setRadius(myCircle.getRadius() * 1.1);
    System.out.println("The area of the circle of radius "
      + myCircle.getRadius() + " is " + myCircle.getArea());

    System.out.println("The number of objects created is "
      + CircleWithPrivateDataFields.getNumberOfObjects());
  }
}
```

UML Class Diagram: Circle with Private Data Fields



Circle

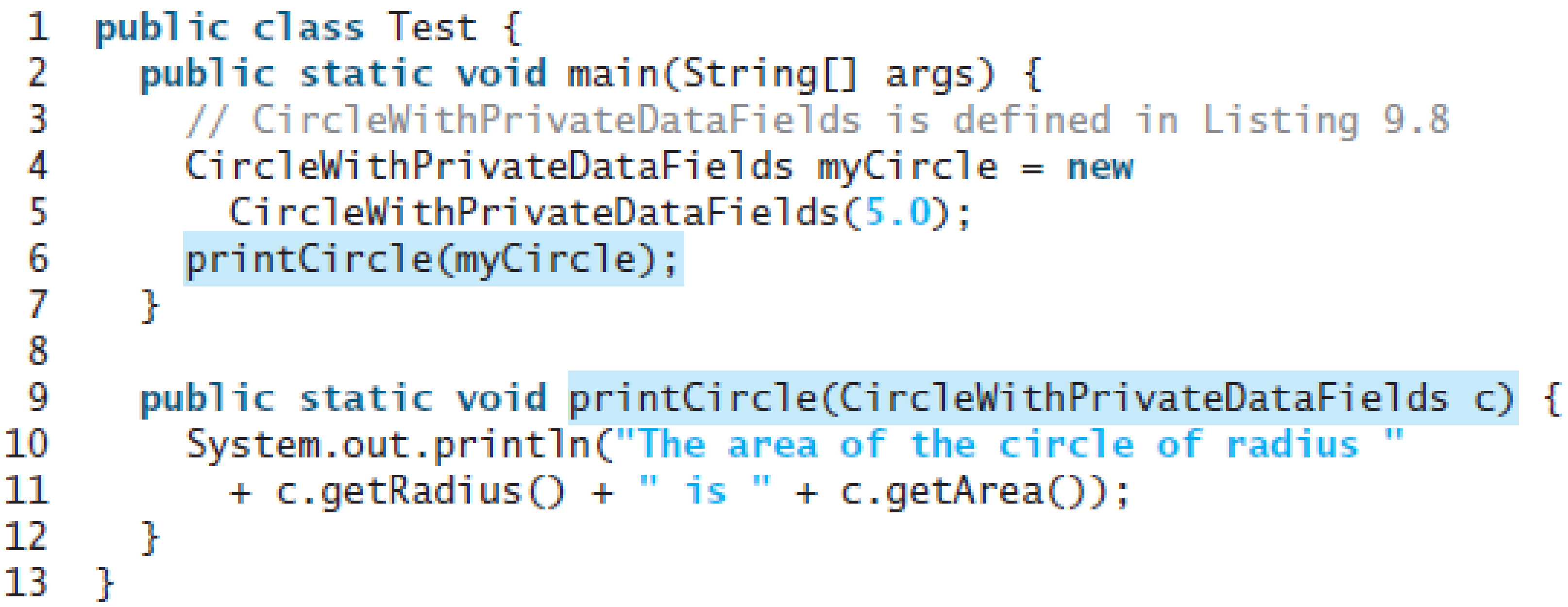
-radius: double
-numberOfObjects: int

+Circle()
+Circle(radius: double)
+getRadius(): double
+setRadius(radius: double): void
+getNumberOfObjects(): int
+getArea(): double

- The (-) sign indicates a private modifier.
- The (+) sign indicates a public modifier.

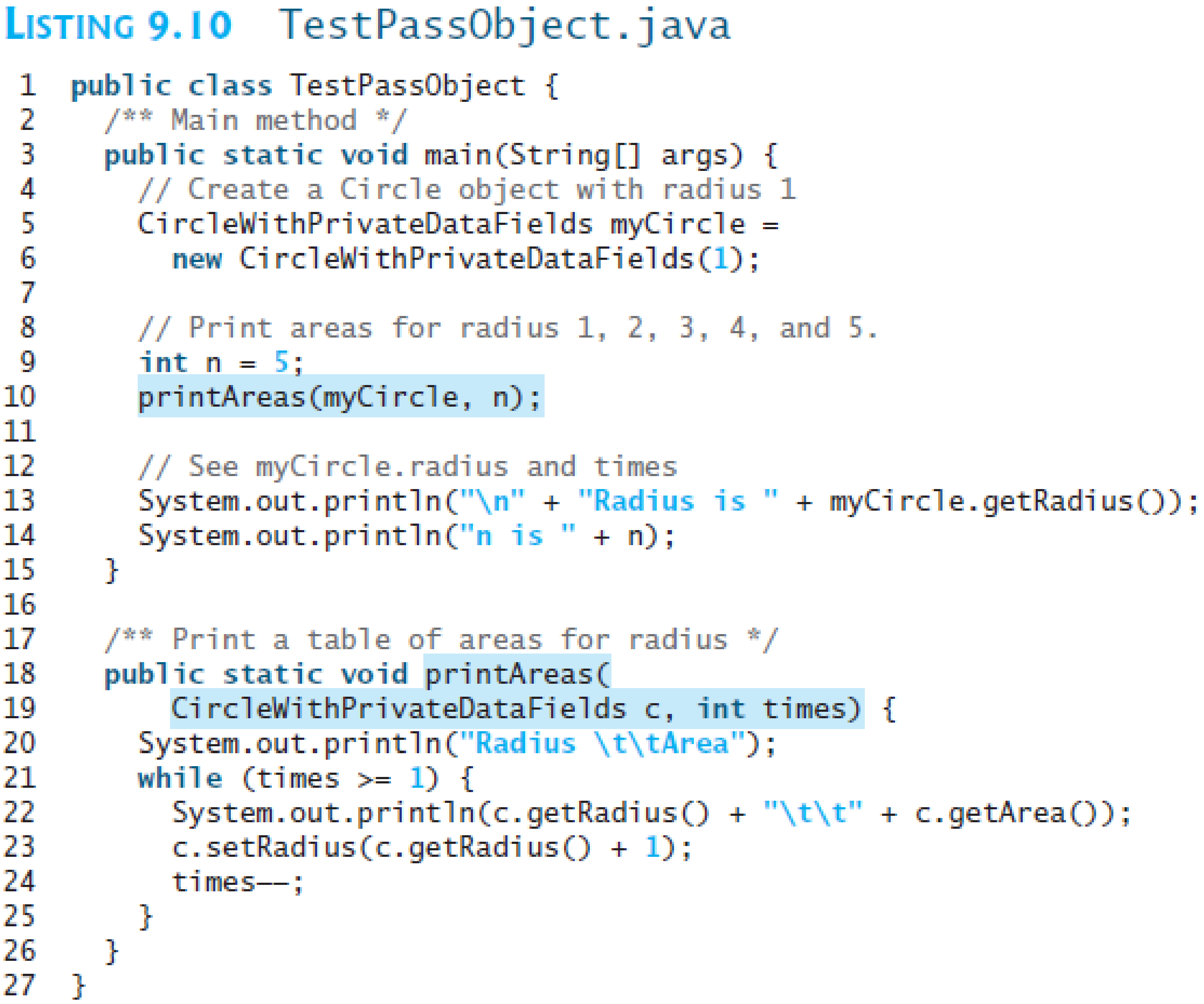
Passing Objects to Methods

- Passing an object to a method is to pass the reference of the object.
- The following code passes the *myCircle* object as an argument to the *printCircle* method:



```
public class Test {
  public static void main(String[] args) {
    // CircleWithPrivateDataFields is defined in Listing 9.8
    CircleWithPrivateDataFields myCircle = new
      CircleWithPrivateDataFields(5.0);
    printCircle(myCircle);
  }

  public static void printCircle(CircleWithPrivateDataFields c) {
    System.out.println("The area of the circle of radius "
      + c.getRadius() + " is " + c.getArea());
  }
}
```

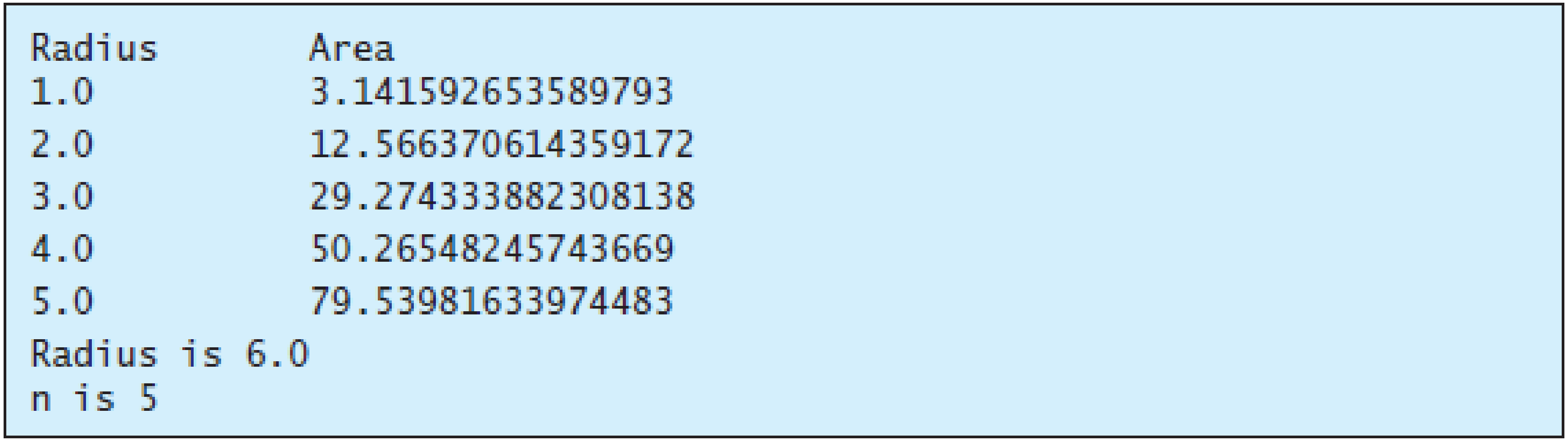
Passing Objects to Methods (Example)



LISTING 9.10 TestPassObject.java

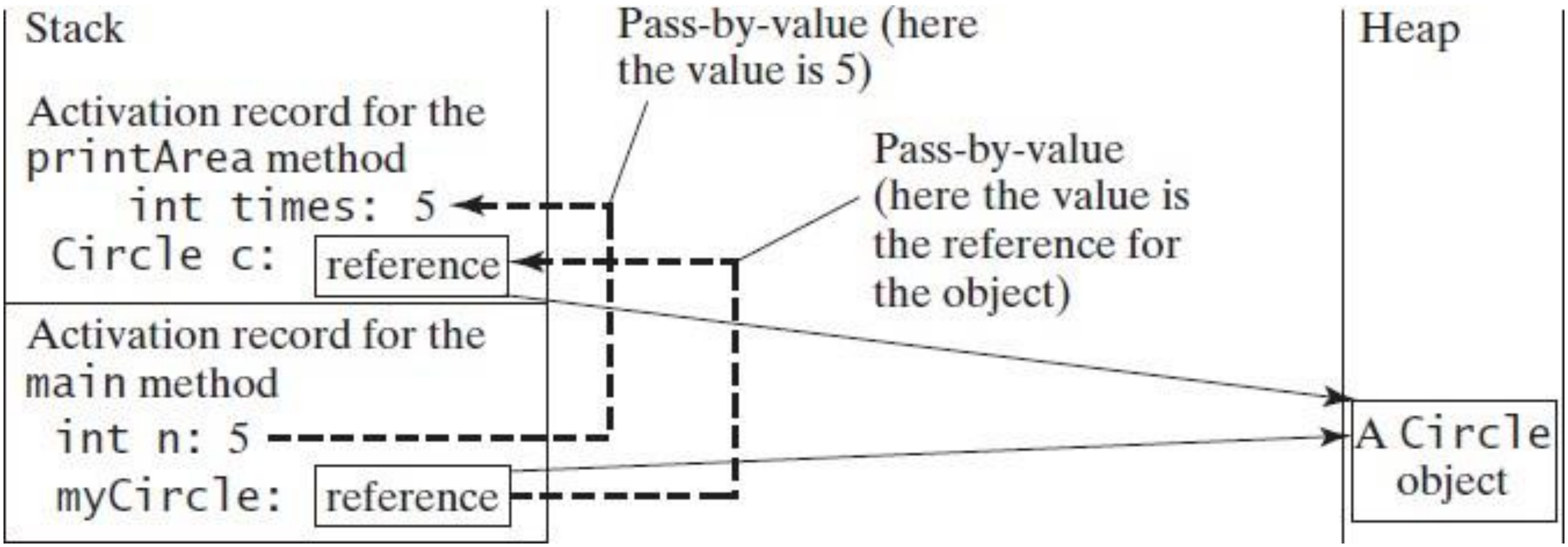
```java
public class TestPassObject {
  /** Main method */
  public static void main(String[] args) {
    // Create a Circle object with radius 1
    CircleWithPrivateDataFields myCircle =
      new CircleWithPrivateDataFields(1);

    // Print areas for radius 1, 2, 3, 4, and 5.
    int n = 5;
    printAreas(myCircle, n);

    // See myCircle.radius and times
    System.out.println("\n" + "Radius is " + myCircle.getRadius());
    System.out.println("n is " + n);
  }

  /** Print a table of areas for radius */
  public static void printAreas(
      CircleWithPrivateDataFields c, int times) {
    System.out.println("Radius \t\tArea");
    while (times >= 1) {
      System.out.println(c.getRadius() + "\t\t" + c.getArea());
      c.setRadius(c.getRadius() + 1);
      times--;
    }
  }
}
```

Passing Objects to Methods Example Output



```
Radius       Area
1.0          3.141592653589793
2.0          12.566370614359172
3.0          29.274333882308138
4.0          50.26548245743669
5.0          79.53981633974483
Radius is 6.0
n is 5
```

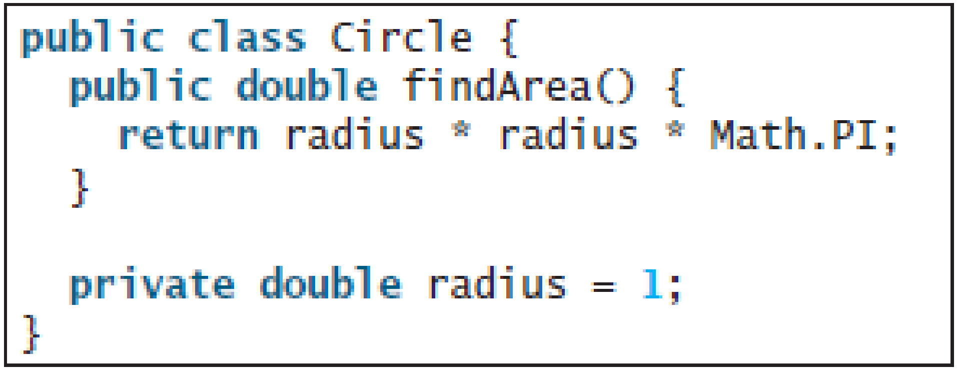
Passing Objects to Methods Example Explanation


Stack
Activation record for the printArea method
int times: 5
Circle c:
reference
Activation record for the main method
int n: 5
myCircle:
reference
Pass-by-value (here the value is 5)
Pass-by-value (here the value is the reference for the object)
Heap
A Circle object

The Scope of Variables

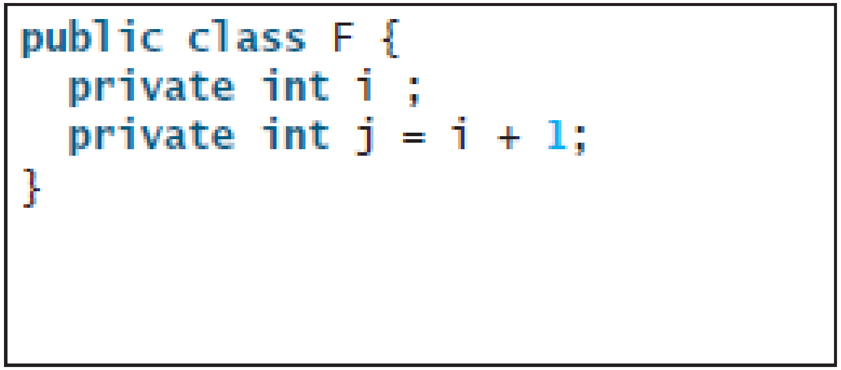
- The scope of a *class's variables* or *data fields* is the *entire class*, regardless of where the variables are declared.
- A class's variables and methods can appear in any order in the class.
  - The exception is when a data field is initialized based on a reference to another data field.

The Scope of Variables (Cont.)



```
public class Circle {
  public double findArea() {
    return radius * radius * Math.PI;
  }

  private double radius = 1;
}
```

(a) The variable radius and method findArea() can be declared in any order.



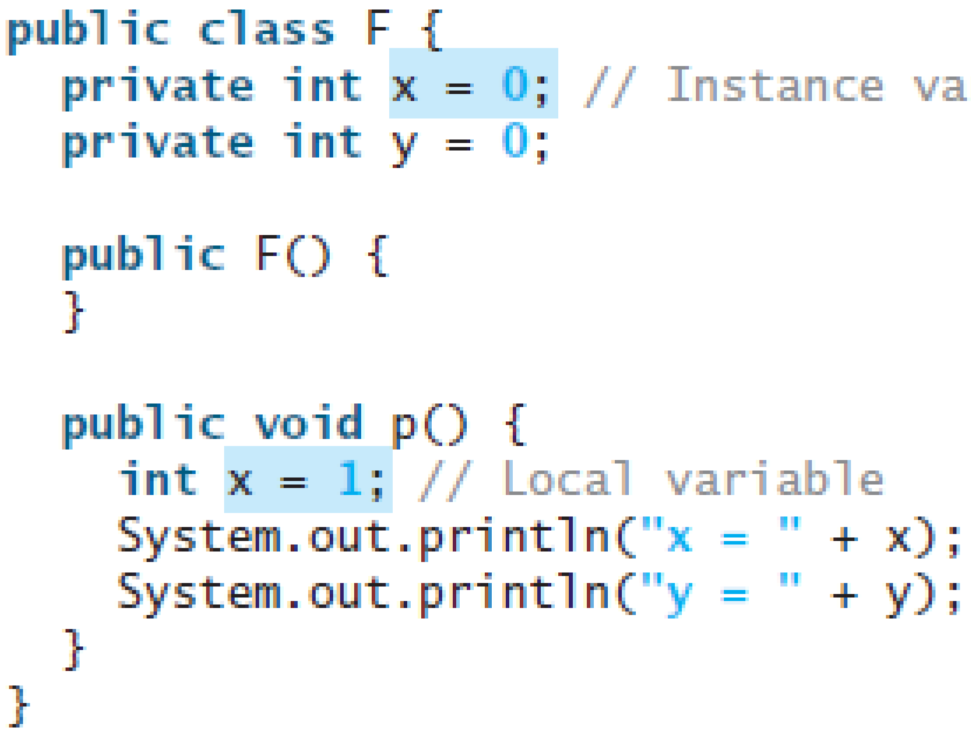
```
public class F {
  private int i ;
  private int j = i + 1;
}
```

(b) i has to be declared before j because j's initial value is dependent on i.

The Scope of Variables (Cont.)

- You can declare a class’s variable only once.
  - But you can declare the same variable name in a method many times in different nonnesting blocks.
- If a local variable has the same name as a class’s variable, the local variable takes precedence and the class’s variable with the same name is *hidden*.

The Scope of Variables (Cont.)



```
public class F {
  private int x = 0; // Instance variable
  private int y = 0;

  public F() {
  }

  public void p() {
    int x = 1; // Local variable
    System.out.println("x = " + x);
    System.out.println("y = " + y);
  }
}
```

If the following statements are created in the *main* method, what is the output?
F fObject = new F();
fObject.print();