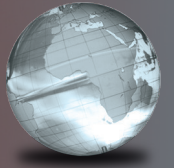


GLOBAL
EDITION



Introduction to Java™ Programming and Data Structures

Comprehensive Version

Y. Daniel Liang

TWELFTH EDITION



INTRODUCTION TO
JAVATM
PROGRAMMING AND
DATA STRUCTURES
COMPREHENSIVE VERSION

Twelfth Edition
Global Edition

Y. Daniel Liang

Georgia Southern University



LISTING 9.10 TestPassObject.java

pass object

object parameter

```
1 public class TestPassObject {
2     /** Main method */
3     public static void main(String[] args) {
4         // Create a Circle object with radius 1
5         Circle myCircle =
6             new Circle(1); // Use the Circle class in Listing 9.8
7
8         // Print areas for radius 1, 2, 3, 4, and 5.
9         int n = 5;
10        printAreas(myCircle, n);
11
12        // See myCircle.radius and times
13        System.out.println("\n" + "Radius is " + myCircle.getRadius());
14        System.out.println("n is " + n);
15    }
16
17    /** Print a table of areas for radius */
18    public static void printAreas(Circle c, int times) {
19        System.out.println("Radius \t\tArea");
20        while (times >= 1) {
21            System.out.println(c.getRadius() + "\t\t" + c.getArea());
22            c.setRadius(c.getRadius() + 1);
23            times--;
24        }
25    }
26 }
```



Radius	Area
1.0	3.141592653589793
2.0	12.566370614359172
3.0	28.274333882308138
4.0	50.26548245743669
5.0	78.53981633974483
Radius is 6.0	
n is 5	

The **Circle** class is defined in Listing 9.8. The program passes a **Circle** object **myCircle** and an integer value from **n** to invoke **printAreas(myCircle, n)** (line 10), which prints a table of areas for radii **1, 2, 3, 4, and 5**, as presented in the sample output.

Figure 9.18 shows the call stack for executing the methods in the program. Note the objects are stored in a heap (see Section 7.6).

When passing an argument of a primitive data type, the value of the argument is passed. In this case, the value of **n (5)** is passed to **times**. Inside the **printAreas** method, the content of **times** is changed; this does not affect the content of **n**.

When passing an argument of a reference type, the reference of the object is passed. In this case, **c** contains a reference for the object that is also referenced via **myCircle**. Therefore, changing the properties of the object through **c** inside the **printAreas** method has the same effect as doing so outside the method through the variable **myCircle**. Pass-by-value on references can be best described semantically as *pass-by-sharing*; that is, the object referenced in the method is the same as the object being passed.

pass-by-sharing

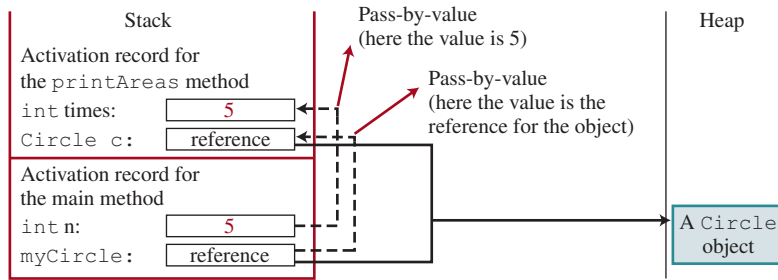


FIGURE 9.18 The value of `n` is passed to `times`, and the reference to `myCircle` is passed to `c` in the `printAreas` method.

9.10.1 Describe the difference between passing a parameter of a primitive type and passing a parameter of a reference type. Show the output of the following programs:



```
public class Test {
    public static void main(String[] args) {
        Count myCount = new Count();
        int times = 0;

        for (int i = 0; i < 100; i++)
            increment(myCount, times);

        System.out.println("count is " + myCount.count);
        System.out.println("times is " + times);
    }

    public static void increment(Count c, int times) {
        c.count++;
        times++;
    }
}
```

```
public class Count {
    public int count;

    public Count (int c) {
        count = c;
    }

    public Count () {
        count = 1;
    }
}
```

9.10.2 Show the output of the following program:

```
public class Test {
    public static void main(String[] args) {
        Circle circle1 = new Circle(1);
        Circle circle2 = new Circle(2);

        swap1(circle1, circle2);
        System.out.println("After swap1: circle1 = " +
            circle1.radius + " circle2 = " + circle2.radius);

        swap2(circle1, circle2);
        System.out.println("After swap2: circle1 = " +
            circle1.radius + " circle2 = " + circle2.radius);
    }

    public static void swap1(Circle x, Circle y) {
        Circle temp = x;
        x = y;
        y = temp;
    }
}
```

```

        public static void swap2(Circle x, Circle y) {
            double temp = x.radius;
            x.radius = y.radius;
            y.radius = temp;
        }
    }

    class Circle {
        double radius;

        Circle(double newRadius) {
            radius = newRadius;
        }
    }

```

9.10.3 Show the output of the following code:

```

public class Test {
    public static void main(String[] args) {
        int[] a = {1, 2};
        swap(a[0], a[1]);
        System.out.println("a[0] = " + a[0]
            + " a[1] = " + a[1]);
    }

    public static void swap(int n1, int n2) {
        int temp = n1;
        n1 = n2;
        n2 = temp;
    }
}

```

(a)

```

public class Test {
    public static void main(String[] args) {
        int[] a = {1, 2};
        swap(a);
        System.out.println("a[0] = " + a[0]
            + " a[1] = " + a[1]);
    }

    public static void swap(int[] a) {
        int temp = a[0];
        a[0] = a[1];
        a[1] = temp;
    }
}

```

(b)

```

public class Test {
    public static void main(String[] args) {
        T t = new T();
        swap(t);
        System.out.println("e1 = " + t.e1
            + " e2 = " + t.e2);
    }

    public static void swap(T t) {
        int temp = t.e1;
        t.e1 = t.e2;
        t.e2 = temp;
    }
}

class T {
    int e1 = 1;
    int e2 = 2;
}

```

(c)

```

public class Test {
    public static void main(String[] args) {
        T t1 = new T();
        T t2 = new T();
        System.out.println("t1's i = " +
            t1.i + " and j = " + t1.j);
        System.out.println("t2's i = " +
            t2.i + " and j = " + t2.j);
    }
}

class T {
    static int i = 0;
    int j = 0;

    T() {
        i++;
        j = 1;
    }
}

```

(d)

9.10.4 What is the output of the following programs?

```
import java.util.Date;

public class Test {
    public static void main(String[] args) {
        Date date = null;
        m1(date);
        System.out.println(date);
    }

    public static void m1(Date date) {
        date = new Date();
    }
}
```

(a)

```
import java.util.Date;

public class Test {
    public static void main(String[] args) {
        Date date = new Date(1234567);
        m1(date);
        System.out.println(date.getTime());
    }

    public static void m1(Date date) {
        date = new Date(7654321);
    }
}
```

(b)

```
import java.util.Date;

public class Test {
    public static void main(String[] args) {
        Date date = new Date(1234567);
        m1(date);
        System.out.println(date.getTime());
    }

    public static void m1(Date date) {
        date.setTime(7654321);
    }
}
```

(c)

```
import java.util.Date;

public class Test {
    public static void main(String[] args) {
        Date date = new Date(1234567);
        m1(date);
        System.out.println(date.getTime());
    }

    public static void m1(Date date) {
        date = null;
    }
}
```

(d)

9.11 Array of Objects

An array can hold objects as well as primitive-type values.

Chapter 7, Single-Dimensional Arrays, described how to create arrays of primitive-type elements. You can also create arrays of objects. For example, the following statement declares and creates an array of 10 **Circle** objects:

```
Circle[] circleArray = new Circle[10];
```

To initialize **circleArray**, you can use a **for** loop as follows:

```
for (int i = 0; i < circleArray.length; i++) {
    circleArray[i] = new Circle();
}
```

An array of objects is actually an *array of reference variables*. Thus, invoking **circleArray[1].getArea()** involves two levels of referencing, as shown in Figure 9.19. **circleArray** references the entire array, and **circleArray[1]** references a **Circle** object.



Note

When an array of objects is created using the **new** operator, each element in the array is a reference variable with a default value of **null**.




```

41  /** Add circle areas */
42  public static double sum(Circle[] circleArray) {
43      // Initialize sum
44      double sum = 0;
45
46      // Add areas to sum
47      for (int i = 0; i < circleArray.length; i++)
48          sum += circleArray[i].getArea();
49
50      return sum;
51  }
52  }

```

pass array of objects

Radius	Area
70.577708	15649.941866
44.152266	6124.291736
24.867853	1942.792644
5.680718	101.380949
36.734246	4239.280350

The total area of circles is 28056.687544



The program invokes `createCircleArray()` (line 8) to create an array of five circle objects. Several circle classes were introduced in this chapter. This example uses the `Circle` class introduced in Section 9.9, Data Field Encapsulation.

The circle radii are randomly generated using the `Math.random()` method (line 19). The `createCircleArray` method returns an array of `Circle` objects (line 23). The array is passed to the `printCircleArray` method, which displays the radius and area of each circle and the total area of the circles.

The sum of the circle areas is computed by invoking the `sum` method (line 38), which takes the array of `Circle` objects as the argument and returns a `double` value for the total area.

9.11.1 What is wrong in the following code?

```

1  public class Test {
2      public static void main(String[] args) {
3          java.util.Date[] dates = new java.util.Date[10];
4          System.out.println(dates[0]);
5          System.out.println(dates[0].toString());
6      }
7  }

```

9.12 Immutable Objects and Classes

You can define immutable classes to create immutable objects. The contents of immutable objects cannot be changed.

Normally, you create an object and allow its contents to be changed later. However, occasionally it is desirable to create an object whose contents cannot be changed once the object has been created. We call such an object an *immutable object* and its class an *immutable class*. The `String` class, for example, is immutable. If you deleted the setter method in the `Circle` class in Listing 9.8, the class would be immutable because radius is private and cannot be changed without a setter method.

If a class is immutable, then all its data fields must be private and it cannot contain public setter methods for any data fields. A class with all private data fields and no mutators is not



VideoNote

Immutable objects and this keyword

immutable object
immutable class