18. Objects and Classes

Java

Summer 2008

Instructor: Dr. Masoud Yaghini

Outline (1)

- Introduction
- Defining Classes for Objects
- Constructors
- Creating Objects
- Accessing an Object's Data and Methods
- An Example: CreatObjectDemo.java
- An Example: TestCircle1.java
- Reference Data Fields and the null Value
- Differences Between Variables of Primitive Types and Reference Types

Outline (2)

- Using Classes from the Java Library
- Static Variables, Constants, and Methods
- Visibility Modifiers
- Data Field Encapsulation
- Immutable Objects and Classes
- Passing Objects to Methods
- The Scope of Variables
- Array of Objects
- References

Introduction

Procedural Programming Languages

- Programming in procedural languages like C, Pascal, BASIC, and COBOL involves:
  - choosing data structures,
  - designing algorithms, and
  - translating algorithms into code.
- In procedural programming, data and operations on the data are separate, and this methodology requires sending data to methods.

OO Programming Concepts

- ***Object-oriented programming*** (OOP) involves programming using objects.
- An ***object*** represents an entity in the real world that can be distinctly identified. For example:
  - a student
  - a desk
  - a circle
  - a button
  - a loan

OO Programming Concepts

- An object has a unique identity, state, and behaviors.
- ***State*** **:**
  - The state of an object consists of a set of *data fields* (also known as *properties*) with their current values.
  - The state defines the object.
- ***Behavior*** **:**
  - The behavior of an object is defined by a set of methods.
  - Invoking a method on an object means that you ask the object to perform a task.
  - The behavior defines what the object does.

Defining Classes for Objects

Defining Classes for Objects

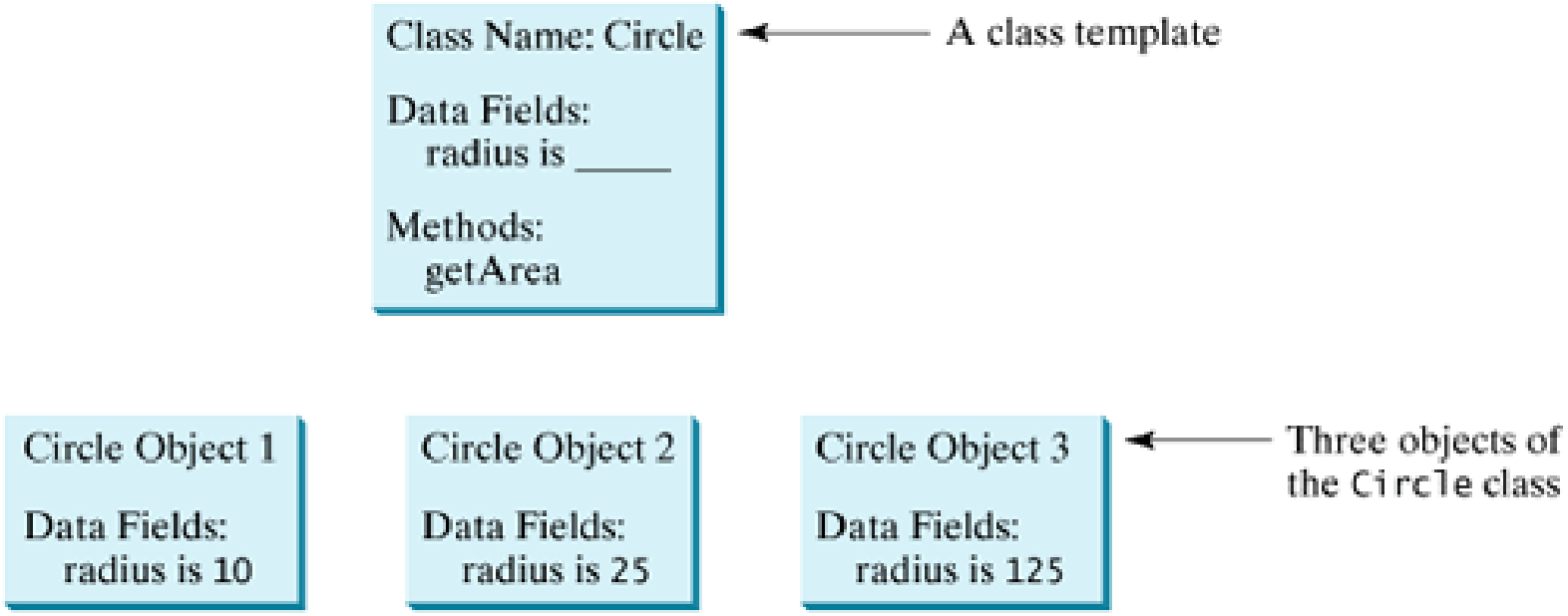
- A circle object, for example, has a data field, radius, which is the property that characterizes a circle.
- One behavior of a circle is that its area can be computed using the method getArea().

Defining Classes for Objects

- ***Classes*** are templates or blueprints that define objects of the same type
- A class defines what an object's data and methods will be.
- An object is an instance of a class.
- You can create many instances of a class.
- Creating an instance is referred to as ***instantiation***.
- The terms object and ***instance*** are often interchangeable.

Defining Classes for Objects

- This Figure shows a class named Circle and its three objects.



Defining Classes for Objects

- A Java ***class*** uses variables to define ***data fields*** and ***methods*** to define behaviors.
- A class provides methods of a special type, known as ***constructors***, which are invoked when a new object is created.
- A constructor is a special kind of method.
- A constructor can perform any action, but constructors are designed to perform initializing actions, such as initializing the data fields of objects.

Defining Classes for Objects

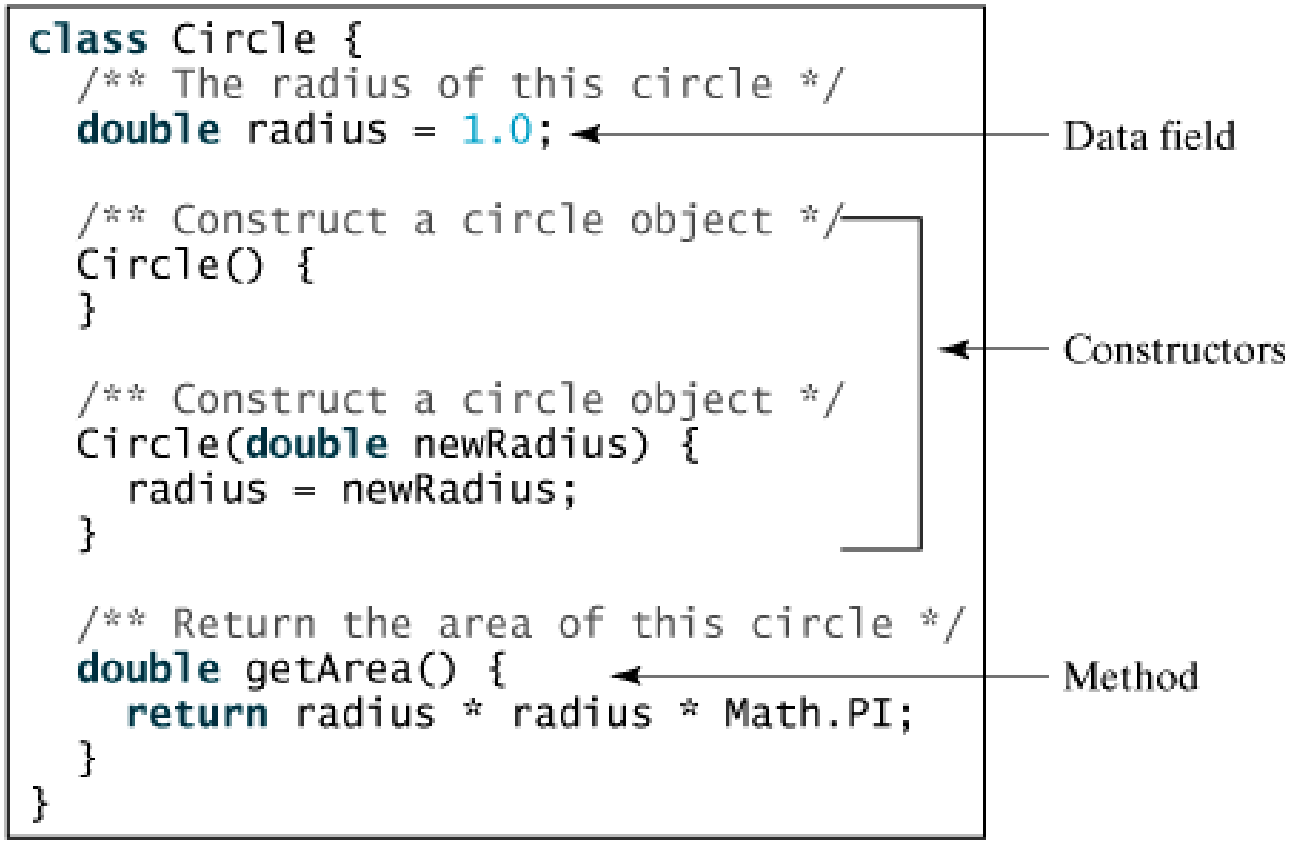
- General form of ***class declaration***:

```
class MyClass {
        // class body: field, constructor, and method declarations
}
```

- The class body (the area between the braces) contains:
  - **declarations for the fields** that provide the state of the class and its objects
  - **constructors** for initializing new objects
  - **methods** to implement the behavior of the class and its objects

Defining Classes for Objects

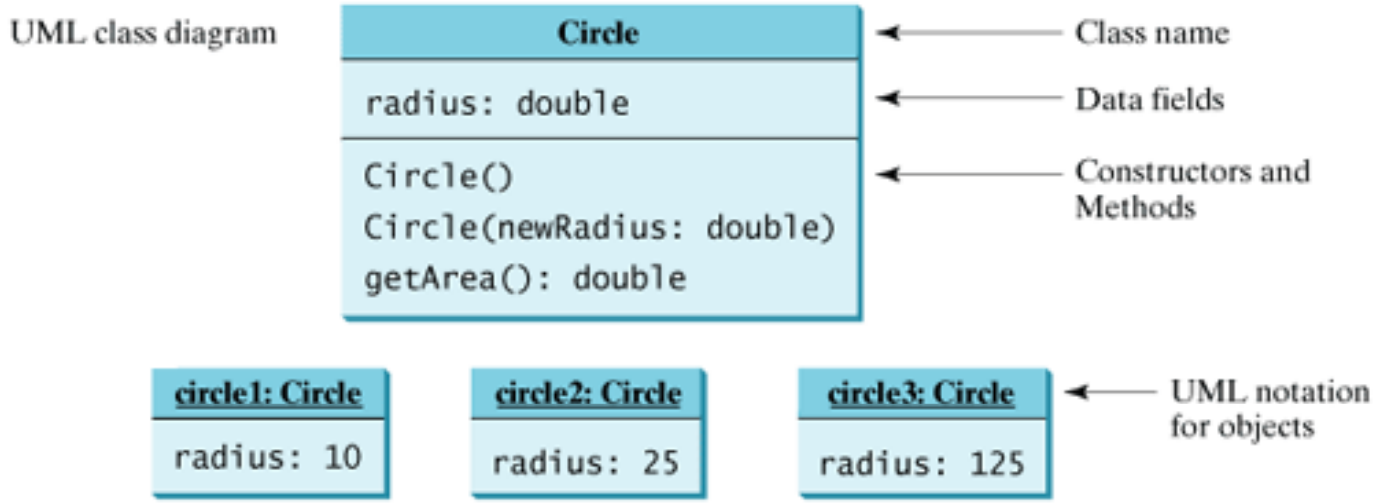
- An example of the class for Circle objects



Defining Classes for Objects

- The Circle class does not have a main method and therefore cannot be run.
- It is merely a definition used to declare and create Circle objects.
- The illustration of class templates and objects in can be standardized using UML (Unified Modeling Language) notations.

UML Class Diagram



- The data field is denoted as:

dataFieldName: dataFieldType

- The constructor is denoted as

ClassName(parameterName: parameterType)

- The method is denoted as:

methodName(parameterName: parameterType): returnType

Constructors

Constructors

- Constructors are a special kind of methods that are invoked to construct objects.
- The constructor has exactly the same name as the defining class.
- Like regular methods, constructors can be overloaded, making it easy to construct objects with different initial data values.

```
Circle() {
}

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

Constructors

- To construct an object from a class, invoke a constructor of the class using the new operator, as follows:

new ClassName(arguments);

- For example:
  - new Circle() creates an object of the Circle class using the first constructor defined in the Circle class
  - new Circle(5) creates an object using the second constructor defined in the Circle class.

Default Constructor

- A constructor with no parameters is referred to as a ***no-arg constructor*** (e.g., Circle()).
- A class may be declared without constructors.
- In this case, a no-arg constructor with an empty body is implicitly declared in the class.
- This constructor, called *a* ***default constructor***, is provided automatically only if no constructors are explicitly declared in the class.

Constructors

- Constructors are a special kind of method, with three differences:
  - Constructors must have the same name as the class itself.
  - Constructors do not have a return type—not even void.
  - Constructors are invoked using the new operator when an object is created. Constructors play the role of initializing objects.

Creating Objects

Creating Objects

- To create an object you should:
  - the declare of an object reference variable
  - the create of an object
  - the assign of the object reference to the variable

Creating Objects

- To reference an object, assign the object to a reference variable.
- Any variable of the class type can reference to an instance of the class.

- To declare an ***object reference variable***, use the syntax:

```
ClassName objectRefVar;
```

- Example:

```
Circle myCircle;
```

Creating Objects

- The variable myCircle can reference a Circle object.
- This statement creates an object and assigns its reference to myCircle.

```
myCircle = new Circle();
```

Creating Objects

- You can write one statement that combines
  - the declaration of an object reference variable,
  - the creation of an object, and
  - the assigning of the object reference to the variable.

```
ClassName objectRefVar = new ClassName();
```

- An example:

```
Circle myCircle = new Circle();
```

Creating Objects

- myCircle is not an object but it a variable that contains a reference to a Circle object.
- For simplicity, we say that myCircle is a Circle object

Accessing an Object's Data and Methods

Accessing an Object's Data and Methods

- After an object is created, its data can be accessed and its methods invoked using the ***dot operator*** **(.)**, also known as the object member access operator:
- To access a data field in the object:
  - objectRefVar.dataField
  - *e.g.*, `myCircle.radius`
- To invoke a method on the object:
  - objectRefVar.method(arguments)
  - *e.g.*, `myCircle.getArea()`

Accessing an Object's Data and Methods

- ***Instance variable*** **:**
  - The data field radius is referred to as an ***instance variable*** because it is dependent on a specific instance.
- ***Instance method*** **:**
  - The method getArea is referred to as an ***instance method***, because you can only invoke it on a specific instance.

- The object on which an instance method is invoked is referred to as a ***calling object***.

Anonymous Object

- You can create an object without explicitly assigning it to a variable, as shown below:

```
System.out.println("Area is " + new Circle(5).getArea());
```

- This statement creates a Circle object and invokes its getArea method to return its area.
- An object created in this way is known as an ***anonymous object***.

An Example: CreatObjectDemo.java

An example

```
1  public class CreateObjectDemo {
2
3      public static void main(String[] args) {
4
5          // Declare and create a point object
6          // and two rectangle objects.
7          Point originOne = new Point(23, 94);
8          Rectangle rectOne = new Rectangle(originOne, 100, 200);
9          Rectangle rectTwo = new Rectangle(50, 100);
10
11         // display rectOne's width, height, and area
12         System.out.println("Width of rectOne: " + rectOne.width);
13         System.out.println("Height of rectOne: " + rectOne.height);
14         System.out.println("Area of rectOne: " +
15                 rectOne.getArea());
16
17         // set rectTwo's position
18         rectTwo.origin = originOne;
19
```

An example

```
20          // display rectTwo's position
21          System.out.println("X Position of rectTwo: " +
22                  rectTwo.origin.x);
23          System.out.println("Y Position of rectTwo: " +
24                  rectTwo.origin.y);
25
26          // move rectTwo and display its new position
27          rectTwo.move(40, 72);
28          System.out.println("X Position of rectTwo: " +
29                  rectTwo.origin.x);
30          System.out.println("Y Position of rectTwo: " +
31                  rectTwo.origin.y);
32      }
33  }
```

An example

```
1  public class Point {
2      public int x = 0;
3      public int y = 0;
4
5      // constructor
6      public Point(int a, int b) {
7          x = a;
8          y = b;
9      }
10 }
```

An example

```
1  public class Rectangle {
2      public int width = 0;
3      public int height = 0;
4      public Point origin;
5
6      // four constructors
7      public Rectangle() {
8          origin = new Point(0, 0);
9      }
10
11     public Rectangle(Point p) {
12         origin = p;
13     }
14
15     public Rectangle(int w, int h) {
16         origin = new Point(0, 0);
17         width = w;
18         height = h;
19     }
20
```

An example

```
21      public Rectangle(Point p, int w, int h) {
22          origin = p;
23          width = w;
24          height = h;
25      }
26
27      // a method for moving the rectangle
28      public void move(int x, int y) {
29          origin.x = x;
30          origin.y = y;
31      }
32
33      // a method for computing the area of the rectangle
34      public int getArea() {
35          return width * height;
36      }
37  }
```

An example

- Here's the output:

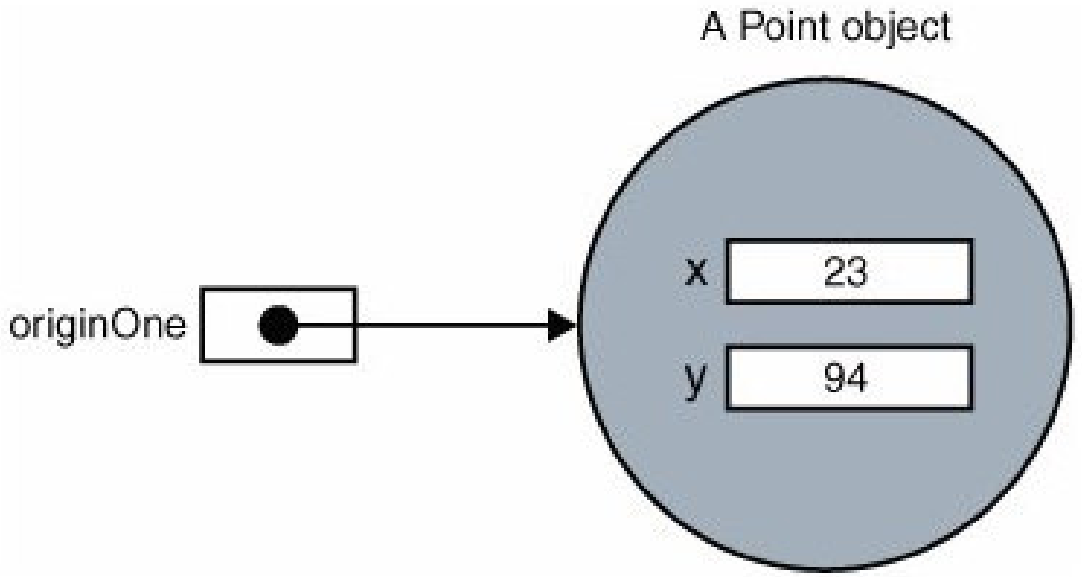
```
Width of rectOne: 100
Height of rectOne: 200
Area of rectOne: 20000
X Position of rectTwo: 23
Y Position of rectTwo: 94
X Position of rectTwo: 40
Y Position of rectTwo: 72
```

An example

- The following statement provides 23 and 94 as values for Point class arguments:

```
Point originOne = new Point(23, 94);
```

- originOne now points to a Point object.



An example

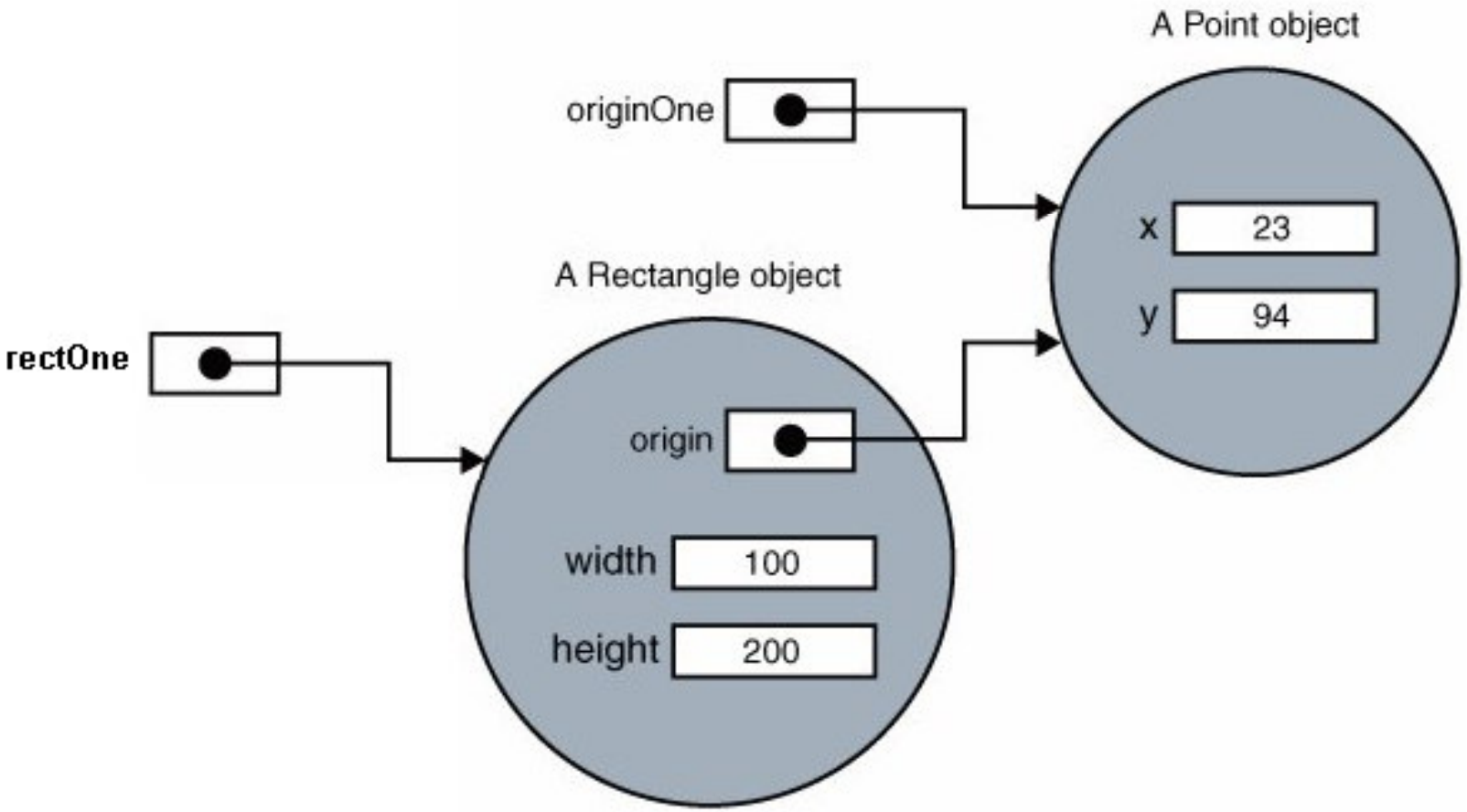
- Rectangle class has different constructors
- but when the Java compiler encounters the following code:

```
Rectangle rectOne = new Rectangle(originOne, 100, 200);
```

- It knows to invoke the constructor in the Rectangle class that requires a Point argument followed by two integer arguments.
- Now there are two references to the same Point object
- An object can have multiple references to it

An example

- rectOne now points to a Rectangle object there are two references to the same Point object:



An example

- The following line of code invokes the Rectangle constructor that requires two integer arguments, which provide the initial values for width and height. And it creates a new Point object whose x and y values are initialized to 0:

```
Rectangle rectTwo = new Rectangle(50, 100);
```

- The Rectangle constructor used in the following statement doesn't take any arguments, so it's called a no-argument constructor:

```
Rectangle rect = new Rectangle();
```

An Example: TestCircle1.java

An example

```
1  package chapter07;
2
3  public class TestCircle1 {
4    /** Main method */
5    public static void main(String[] args) {
6      // Create a circle with radius 5.0
7      Circle1 myCircle = new Circle1(5.0);
8      System.out.println("The area of the circle of radius "
9          + myCircle.radius + " is " + myCircle.getArea());
10
11     // Create a circle with radius 1
12     Circle1 yourCircle = new Circle1();
13     System.out.println("The area of the circle of radius "
14         + yourCircle.radius + " is " + yourCircle.getArea());
15
16     // Modify circle radius
17     yourCircle.radius = 100;
18     System.out.println("The area of the circle of radius "
19         + yourCircle.radius + " is " + yourCircle.getArea());
20   }
21 }
```

An example

```
22
23  // Define the circle class with two constructors
24  class Circle1 {
25    double radius;
26
27    /** Construct a circle with radius 1 */
28    Circle1() {
29      radius = 1.0;
30    }
31
32    /** Construct a circle with a specified radius */
33    Circle1(double newRadius) {
34      radius = newRadius;
35    }
36
37    /** Return the area of this circle */
38    double getArea() {
39      return radius * radius * Math.PI;
40    }
41  }
```

An example

- The program constructs a circle object with radius 5 and an object with radius 1 and displays the radius and area of each of the two circles.
- Change the radius of the second object to 100 and display its new radius and area

An example

- The program contains two classes.
- The first class, TestCircle1, is the main class. Its purpose is to test the second class, Circle1.
- Every time you run the program, the JVM invokes the main method in the main class.
- You can put the two classes into one file, but only one class in the file can be a public class.
- Furthermore, the public class must have the same name as the file name and the main method must be in a public class.

An example

- To write the getArea method in a procedural programming language like Pascal, you would pass radius as an argument to the method.
- But in object-oriented programming, radius and getArea are defined in the object.
- The radius is a data member in the object, which is accessible by the getArea method.
- In procedural programming languages, data and methods are separated, but in an object-oriented programming language, data and methods are grouped together.

Other way to write the program

- There are many ways to write Java programs.
- For instance, you can combine the two classes in the example into one, as shown in next slide.
- This demonstrates that you can test a class by simply adding a main method in the same class.

Other way to write the program

```
1  package chapter07;
2
3  public class Circle1 {
4    /** Main method */
5    public static void main(String[] args) {
6      // Create a circle with radius 5.0
7      Circle1 myCircle = new Circle1(5.0);
8      System.out.println("The area of the circle of radius "
9          + myCircle.radius + " is " + myCircle.getArea());
10
11     // Create a circle with radius 1
12     Circle1 yourCircle = new Circle1();
13     System.out.println("The area of the circle of radius "
14         + yourCircle.radius + " is " + yourCircle.getArea());
15
16     // Modify circle radius
17     yourCircle.radius = 100;
18     System.out.println("The area of the circle of radius "
19         + yourCircle.radius + " is " + yourCircle.getArea());
20   }
```

Other way to write the program

```
21
22      double radius;
23
24      /** Construct a circle with radius 1 */
25      Circle1() {
26          radius = 1.0;
27      }
28
29      /** Construct a circle with a specified radius */
30      Circle1(double newRadius) {
31          radius = newRadius;
32      }
33
34      /** Return the area of this circle */
35      double getArea() {
36          return radius * radius * Math.PI;
37      }
38  }
```

An example

- Recall that you use Math.methodName(arguments) (e.g., Math.pow(3, 2.5)) to invoke a method in the Math class.
- Can you invoke getArea() using Circle1.getArea()?
- The answer is no. All the methods in the Math class are static methods, which are defined using the static keyword.
- However, getArea() is an instance method, and thus non-static.
- It must be invoked from an object using objectRefVar.methodName(arguments) (e.g., myCircle.getArea()). "

Reference Data Fields and the null Value

Reference Data Fields and the null Value

- The data fields can be of reference types.
- For example, the following Student class contains a data field name of the String type.

```
class Student {
  String name; // name has default value null
  int age; // age has default value 0
  boolean isScienceMajor; // isScienceMajor has default value false
  char gender; // gender has default value '\u0000'
}
```

- name is a reference variable.
- String is a predefined Java class.

Reference Data Fields and the null Value

```
1  package chapter07;
2
3  class Test {
4    public static void main(String[] args) {
5      Student student = new Student();
6      System.out.println("name? " + student.name);
7      System.out.println("age? " + student.age);
8      System.out.println("isScienceMajor? " + student.isScienceMajor);
9      System.out.println("gender? " + student.gender);
10   }
11 }
12
13 class Student {
14   String name; // name has default value null
15   int age; // age has default value 0
16   boolean isScienceMajor; // isScienceMajor has default value false
17   char gender; // gender has default value '\u0000'
18 }
```

Reference Data Fields and the null Value

- If a data field of a reference type does not reference any object, the data field holds a special Java value, **null**.
- The default value of a data field is:
  - null for a reference type
  - 0 for a numeric type
  - false for a boolean type
  - '\u0000' for a char type

Reference Data Fields and the null Value

- Java assigns no default value to a local variable inside a method.
- The following code has a compilation error because local variables x and y are not initialized:

```
1  package chapter07;
2
3  class Test2 {
4    public static void main(String[] args) {
5      int x; // x has no default value
6      String y; // y has no default value
7      System.out.println("x is " + x);
8      System.out.println("y is " + y);
9    }
10 }
```

Differences Between Variables of Primitive Types and Reference Types

Objects and Classes

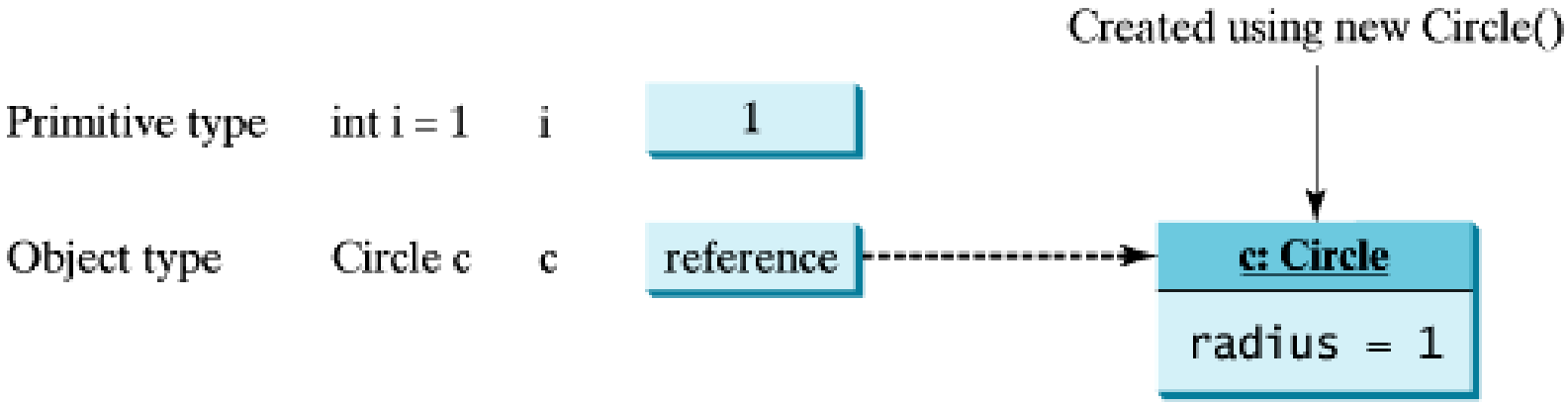
Differences Between Variables of Primitive Types and Reference Types

- Every variable represents a memory location that holds a value.
- When you declare a variable, you are telling the compiler what type of value the variable can hold.
- For a variable of a primitive type, the value is of the primitive type.
- For a variable of a reference type, the value is a reference to where an object is located.

Objects and Classes

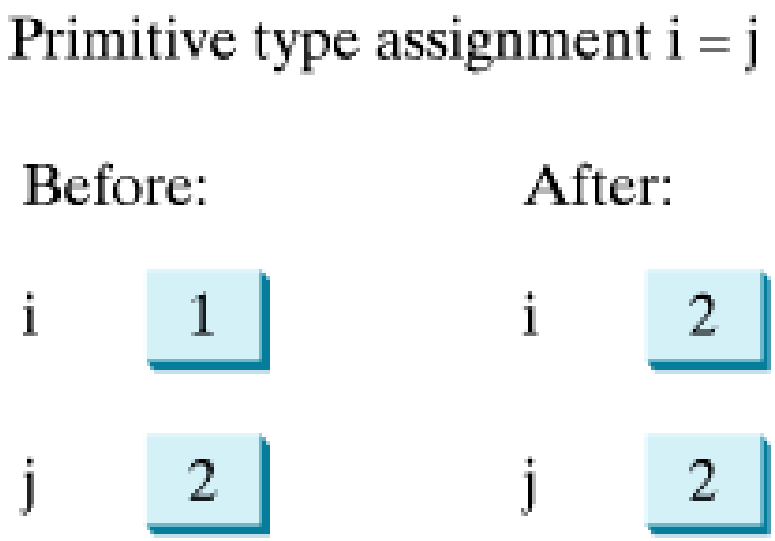
Differences Between Variables of Primitive Types and Reference Types

- The value of int variable i is int value 1, and the value of Circle object c holds a reference to where the contents of the Circle object are stored in the memory.



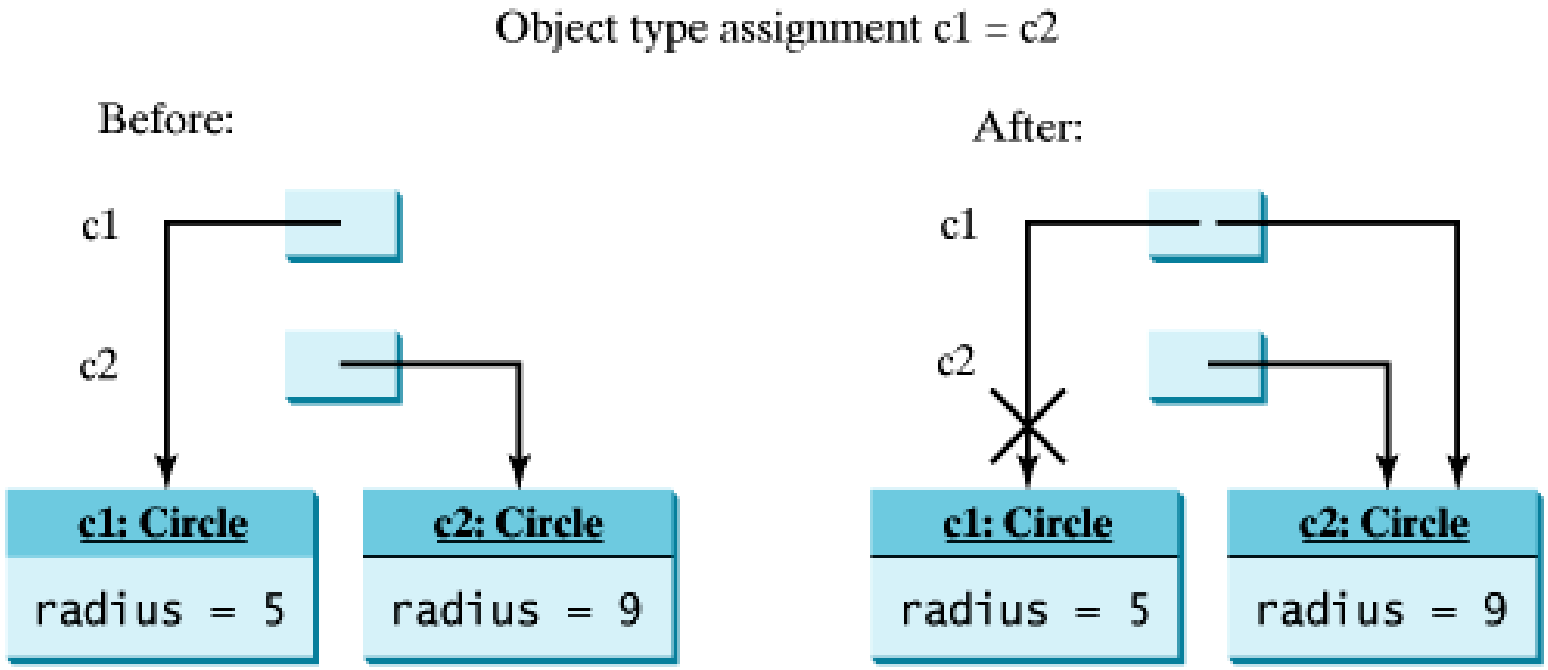
Differences Between Variables of Primitive Types and Reference Types

- When you assign one variable to another, the other variable is set to the same value.
- For a variable of a primitive type, the real value of one variable is assigned to the other variable.



Differences Between Variables of Primitive Types and Reference Types

- For a variable of a reference type, the reference of one variable is assigned to the other variable.



Differences Between Variables of Primitive Types and Reference Types

- After the assignment statement c1 = c2, c1 points to the same object referenced by c2.
- The object previously referenced by c1 is no longer useful and therefore is now known as garbage.
- Garbage occupies memory space.
- The JVM detects garbage and automatically reclaims the space it occupies.
- This process is called ***garbage collection***.

Objects and Classes

Differences Between Variables of Primitive Types and Reference Types

- If you know that an object is no longer needed, you can explicitly assign null to a reference variable for the object.
- The JVM will automatically collect the space if the object is not referenced by any variable.

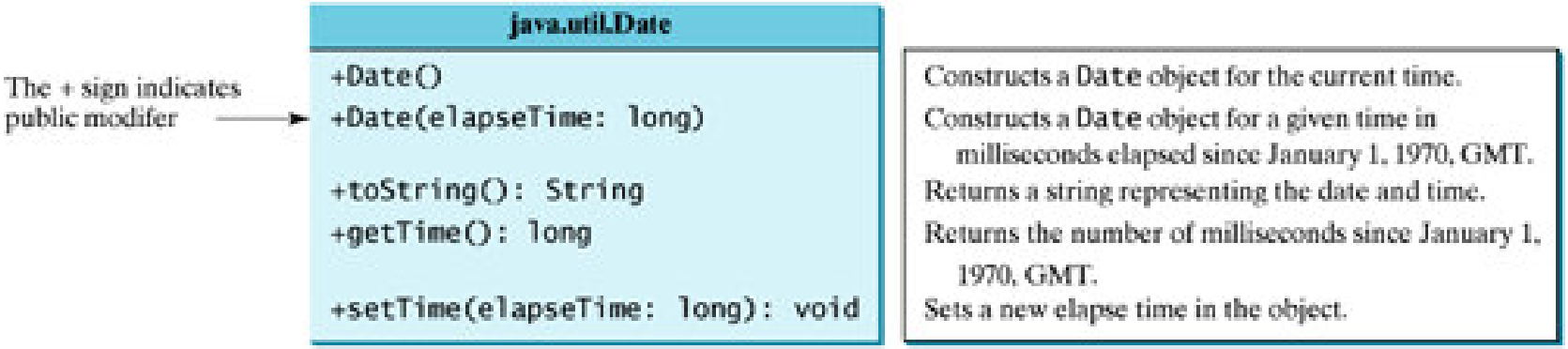
Using Classes from the Java Library

Using Classes from the Java Library

- You will frequently use the classes in the Java library to develop programs.
- This section gives some examples of the classes in the Java library.

The Date Class

- Java provides a system-independent encapsulation of date and time in the java.util.Date class.
- You can use the Date class to create an instance for the current date and time and use its toString method to return the date and time as a string.



The Date Class

- For example, the following code

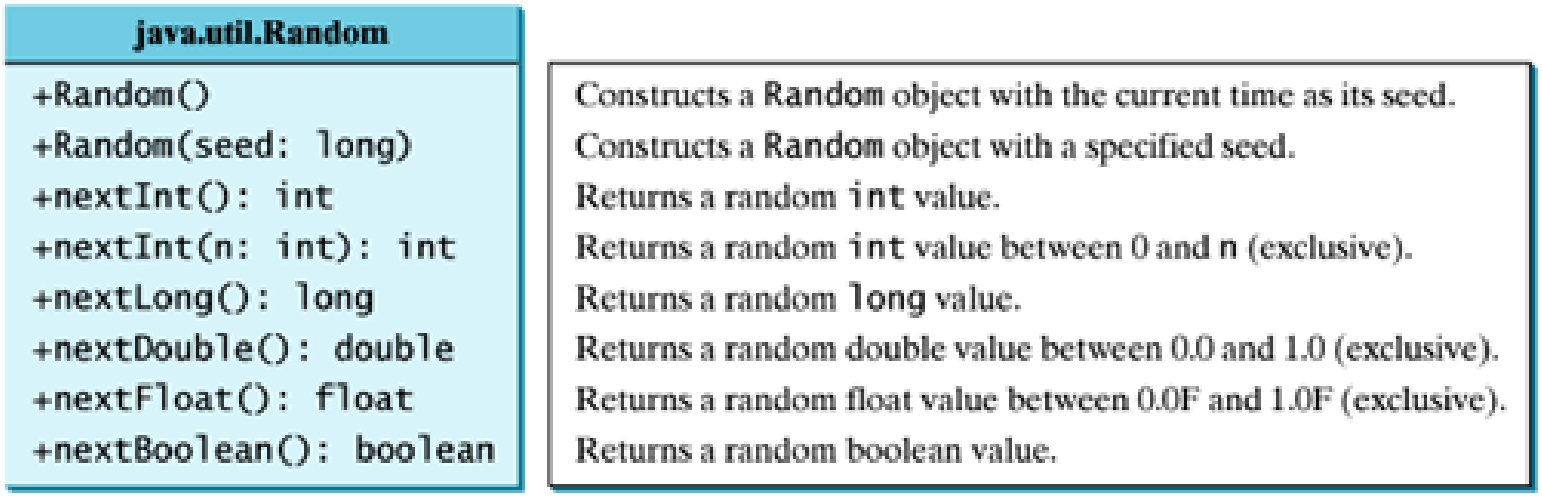
```
java.util.Date date = new java.util.Date();
System.out.println("The elapsed time since Jan 1, 1970 is " +
      date.getTime() + " milliseconds");
System.out.println(date.toString());
```

- displays the output like this:

```
The elapse time since Jan 1, 1970 is
1100547210284 milliseconds
Mon Nov 15 14:33:30 EST 2004
```

The Random Class

- You have used Math.random() to obtain a random double value between 0.0 and 1.0 (excluding 1.0).
- A more useful random number generator is provided in the java.util.Random class, as shown below:



The Random Class

- If two Random objects have the same seed, they will generate identical sequences of numbers. For example, the following code creates two Random objects with the same seed 3.

```
java.util.Random random1 = new java.util.Random(3);
System.out.print("From random1: ");
for (int i = 0; i < 10; i++)
   System.out.print(random1.nextInt(1000) + " ");
java.util.Random random2 = new java.util.Random(3);
System.out.print("\nFrom random2: ");
for (int i = 0; i < 10; i++)
   System.out.print(random2.nextInt(1000) + " ");
```

- The code generates the same sequence of random int values:

  From random1: 734 660 210 581 128 202 549 564 459 961

  From random2: 734 660 210 581 128 202 549 564 459 961

Static Variables, Constants, and Methods

Instance Variables, and Methods

- ***Instance variables*** belong to a specific instance.

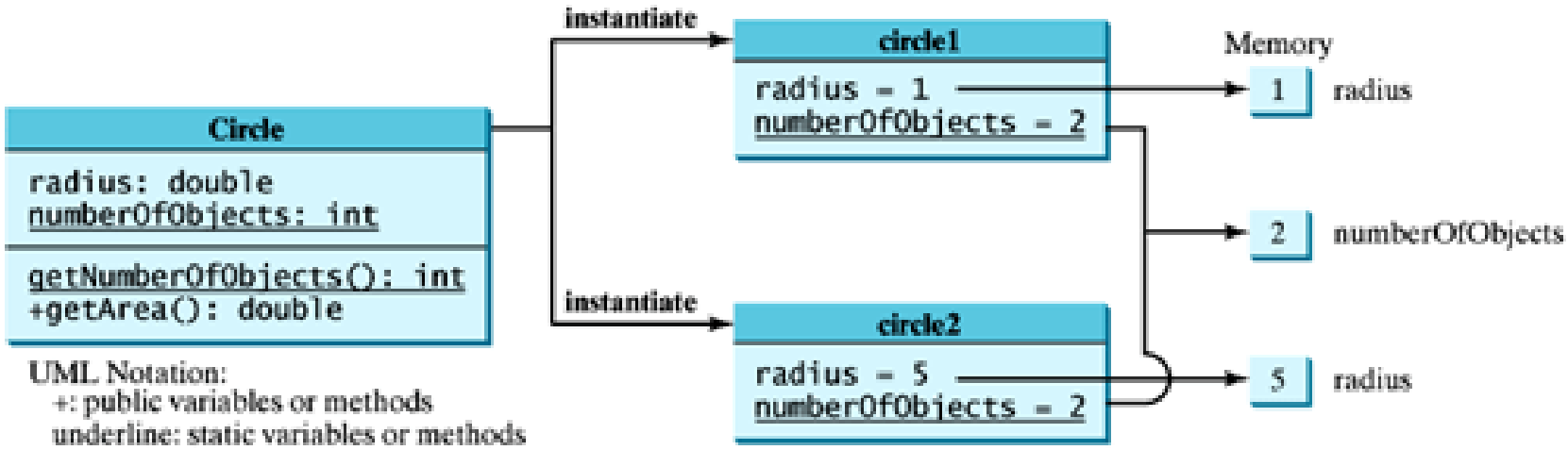
- ***Instance methods*** are invoked by an instance of the class.

Static Variables, Constants, and Methods

- ***Static variables*** (***Class variable***) are shared by all the instances of the class.
- ***Static methods*** are not tied to a specific object. Static methods can be called without creating an instance of the class.
- ***Static constants*** are final variables shared by all the instances of the class.
- To declare static variables, constants, and methods, use the static modifier.

Static Variables, Constants, and Methods

- Let us modify the Circle class by adding a static variable numberOfObjects to count the number of circle objects created and the static method getNumberOfObjects



Static Variables, Constants, and Methods

- Constants in a class are shared by all objects of the class.
- Thus, constants should be declared final static.
- For example, the constant PI in the Math class is defined as:

```
final static double PI = 3.14159265358979323846;
```

Circle2.java

```
1  package chapter07;
2
3  public class Circle2 {
4    /** The radius of the circle */
5    double radius;
6
7    /** The number of the objects created */
8    static int numberOfObjects = 0;
9
10   /** Construct a circle with radius 1 */
11   Circle2() {
12     radius = 1.0;
13     numberOfObjects++;
14   }
15
16   /** Construct a circle with a specified radius */
17   Circle2(double newRadius) {
18     radius = newRadius;
19     numberOfObjects++;
20   }
```

Circle2.java

```
21
22      /** Return numberOfObjects */
23      static int getNumberOfObjects() {
24        return numberOfObjects;
25      }
26
27      /** Return the area of this circle */
28      double getArea() {
29        return radius * radius * Math.PI;
30      }
31  }
```

TestCircle2.java

```
1  package chapter07;
2
3  public class TestCircle2 {
4    /** Main method */
5    public static void main(String[] args) {
6      // Create c1
7      Circle2 c1 = new Circle2();
8
9      // Display c1 BEFORE c2 is created
10     System.out.println("Before creating c2");
11     System.out.println("c1 is : radius (" + c1.radius +
12         ") and number of Circle objects (" +
13         Circle2.numberOfObjects + ")");
14
15     // Create c2
16     Circle2 c2 = new Circle2(5);
17
18     // Change the radius in c1
19     c1.radius = 9;
20
```

TestCircle2.java

```
21        // Display c1 and c2 AFTER c2 was created
22        System.out.println("\nAfter creating c2 and modifying " +
23            "c1's radius to 9");
24        System.out.println("c1 is : radius (" + c1.radius +
25            ") and number of Circle objects (" +
26            Circle2.numberOfObjects + ")");
27        System.out.println("c2 is : radius (" + c2.radius +
28            ") and number of Circle objects (" +
29            Circle2.numberOfObjects + ")");
30    }
31 }
```

TestCircle2.java

- The output of the program:

```
Before creating c2
c1 is : radius (1.0) and number of Circle objects (1)

After creating c2 and modifying c1's radius to 9
c1 is : radius (9.0) and number of Circle objects (2)
c2 is : radius (5.0) and number of Circle objects (2)
```

TestCircle2.java

- You can replace Circle2.numberOfObjects by c1.numberOfObjects and c2.numberOfObjects.
- You can also replace Circle2.numberOfObjects by Circle2.getNumberOfObjects().

Static Variables, Constants, and Methods

- To improve readability use ClassName.methodName(arguments) to invoke a static method and ClassName.staticVariable
- because the user can easily recognize the static method and data in the class.

Import static variables and methods

- You can import static variables and methods from a class.
- The imported data and methods can be referenced or called without specifying a class.
- For example, you can use PI (instead of Math.PI), and random() (instead of Math.random()),
- if you have the following import statement in the class:

```
import static java.lang.Math.*;
```

Static Variables, Constants, and Methods

- Instance methods can use both:
  - Static variables and methods, and
  - Instance variables and methods
- Static methods can use only:
  - Static variables and methods

- Because static variables and methods belong to the class as a whole and not to particular objects.

Static Variables, Constants, and Methods

- What is wrong?

```
 1  package chapter07;
 2
 3  public class Foo {
 4    int i = 5;
 5    static int k = 2;
 6
 7    public static void main(String[] args) {
 8      int j = i;
 9      m1();
10    }
11
12    public void m1() {
13      i = i + k + m2(i, k);
14    }
15
16    public static int m2(int i, int j) {
17      return (int)(Math.pow(i, j));
18    }
19  }
```

Static Variables, Constants, and Methods

```
1  package chapter07;
2
3  public class Foo {
4    int i = 5;
5    static int k = 2;
6
7    public static void main(String[] args) {
8      int j = i; // Wrong because i is an instance variable
9      m1(); // Wrong because m1() is an instance method
10   }
11
12   public void m1() {
13     // Correct since instance and static variables and methods
14     // can be used in an instance method
15     i = i + k + m2(i, k);
16   }
17
18   public static int m2(int i, int j) {
19     // Correct since i , j are local variables
20     return (int)(Math.pow(i, j));
21   }
22 }
```

Static Variables, Constants, and Methods

- How do you decide whether a variable or method should be an instance one or a static one?
- A variable or method that is dependent on a specific instance of the class should be an instance variable or method, otherwise it should be a static variable or method.
- None of the methods in the Math class is dependent on a specific instance. Therefore, these methods are static methods.
- The main method is static, and can be invoked directly from a class.

Visibility Modifiers

Visibility Modifiers

- Java provides several modifiers that control access to data fields, methods, and classes.
- By default, the class, variable, or method can be accessed by any class in the same package. This is known as ***package-private*** *or* ***package-access***.
- **public**
  - The class, data, or method is visible to any class in any package.
- **private**
  - The data or methods can be accessed only by the own class.

Visibility Modifiers

```
package p1;

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

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

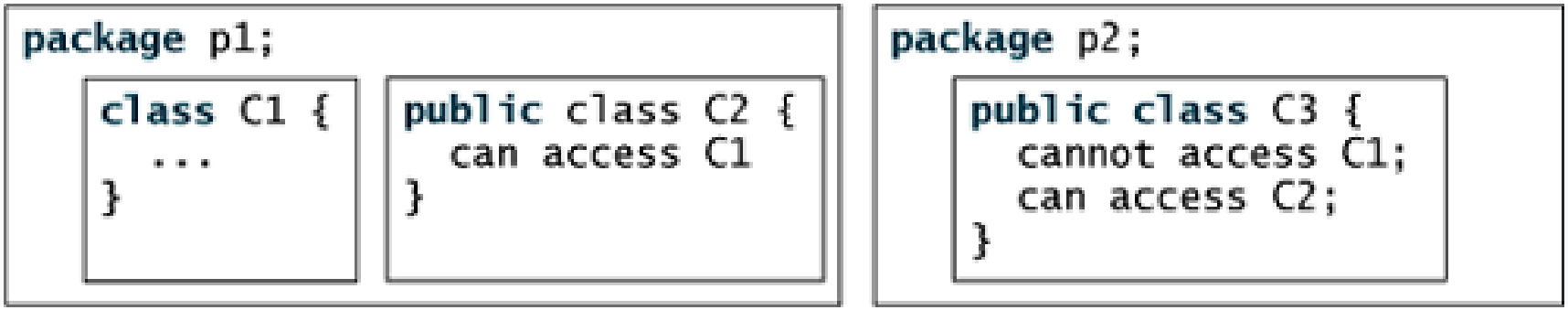
```
public class C2 {
  void aMethod() {
    C1 o = new C1();
    can access o.x;
    can access o.y;
    cannot access o.z;

    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 within a class
- The default modifier restricts access to within a package
- The public modifier enables unrestricted access

Visibility Modifiers

- If a class is not declared public, it can only be accessed within the same package



Visibility Modifiers

- An object cannot access its private members, as shown in (b). It is OK, however, if the object is declared in its own class, as shown in (a).

```
public class Foo {
  private boolean x;

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

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

(a) This is OK because object `foo` is used inside the `Foo` class

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

(b) This is wrong because `x` and `convert` are private in `Foo`.

Note

- Visibility modifiers are used for the members of the class, not local variables inside the methods.
- Using a visibility modifier on local variables would cause a compilation error.

Note

- In most cases, the constructor should be public.
- However, if you want to prohibit the user from creating an instance of a class, you can use a private constructor.
- For example, there is no reason to create an instance from the Math class because all of the data fields and methods are static.
- One solution is to define a dummy private constructor in the class.
- The Math class cannot be instantiated because it has a private constructor, as follows:

```
private Math() {
}
```

Data Field Encapsulation

Data Field Encapsulation

- Why Data Fields Should Be private?
- **To protect data.**
  - For example, numberOfObjects is to count the number of objects created, but it may be set to an arbitrary value (e.g., Circle2. numberOfObjects = 10).
- **To make class easy to maintain.**
  - Suppose you want to modify the Circle2 class to ensure that the radius is non-negative after other programs have already used the class.
  - You have to change not only the Circle2 class, but also the programs that use the Circle2 class.
  - Such programs are often referred to as ***clients***.

Data Field Encapsulation

- ***Data field encapsulation***
  - To prevent direct modifications of properties, you should declare the field private, using the private modifier.
  - This is known as *data field encapsulation*.
- To make a private data field accessible, provide a get method to return the value of the data field.
- To enable a private data field to be updated, provide a set method to set a new value.

Data Field Encapsulation

- A get method is referred to as a ***getter*** (or ***accessor***), and a set method is referred to as a ***setter*** (or ***mutator***).

- get method has the following signature:

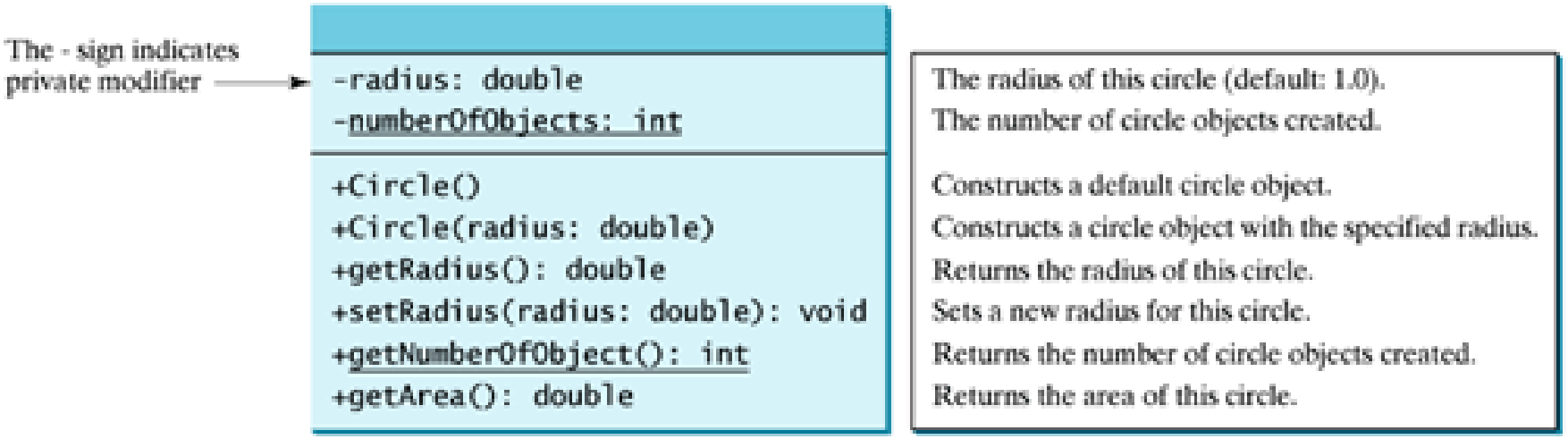
  public returnType getPropertyName()

- set method has the following signature:

  public void setPropertyName(dataType propertyValue)

Data Field Encapsulation

- The class diagram to create a new circle class with a private data field radius and its associated accessor and mutator methods.



Circle3.java

```
1  package chapter07;
2
3  public class Circle3 {
4    /** The radius of the circle */
5    private double radius = 1;
6
7    /** The number of the objects created */
8    private static int numberOfObjects = 0;
9
10   /** Construct a circle with radius 1 */
11   public Circle3() {
12     numberOfObjects++;
13   }
14
15   /** Construct a circle with a specified radius */
16   public Circle3(double newRadius) {
17     radius = newRadius;
18     numberOfObjects++;
19   }
20
```

Circle3.java

```
21    /** Return radius */
22    public double getRadius() {
23      return radius;
24    }
25
26    /** Set a new radius */
27    public void setRadius(double newRadius) {
28      radius = (newRadius >= 0) ? newRadius : 0;
29    }
30
31    /** Return numberOfObjects */
32    public static int getNumberOfObjects() {
33      return numberOfObjects;
34    }
35
36    /** Return the area of this circle */
37    public double getArea() {
38      return radius * radius * Math.PI;
39    }
40  }
```

TestCircle3.java: Demonstrate private modifier

```
1  package chapter07;
2
3  public class TestCircle3 {
4    /** Main method */
5    public static void main(String[] args) {
6      // Create a Circle with radius 5.0
7      Circle3 myCircle = new Circle3(5.0);
8      System.out.println("The area of the circle of radius "
9          + myCircle.getRadius() + " is " + myCircle.getArea());
10
11     // Increase myCircle's radius by 10%
12     myCircle.setRadius(myCircle.getRadius() * 1.1);
13     System.out.println("The area of the circle of radius "
14         + myCircle.getRadius() + " is " + myCircle.getArea());
15   }
16 }
```

TestCircle3.java: Demonstrate private modifier

- The output:

```
The area of the circle of radius 5.0 is 78.53981633974483
The area of the circle of radius 5.5 is 95.03317777109125
```

Note

- When you compile TestCircle3.java, the Java compiler automatically compiles Circle3.java if it has not been compiled since the last change.

Immutable Objects and Classes

Immutable Objects and Classes

- If the contents of an object cannot be changed once the object is created, the object is called an ***immutable object*** and its class is called an ***immutable class***.
- If you delete the set method in the Circle3 class in the preceding example, the class would be immutable because radius is private and cannot be changed without a set method.

Immutable Objects and Classes

- class with all private data fields and no mutators is not necessarily immutable.

Immutable Objects and Classes

```
1  package chapter07;
2
3  public class Student {
4    private int id;
5    private BirthDate birthDate;
6
7    public Student(int ssn, int year, int month, int day) {
8      id = ssn;
9      birthDate = new BirthDate(year, month, day);
10   }
11
12   public int getId() {
13     return id;
14   }
15
16   public BirthDate getBirthDate() {
17     return birthDate;
18   }
19 }
```

Immutable Objects and Classes

```
 1  package chapter07;
 2
 3  public class BirthDate {
 4     private int year;
 5     private int month;
 6     private int day;
 7
 8     public BirthDate(int newYear, int newMonth, int newDay) {
 9        year = newYear;
10        month = newMonth;
11        day = newDay;
12     }
13
14     public void setYear(int newYear) {
15        year = newYear;
16     }
17  }
```

Immutable Objects and Classes

```
1  package chapter07;
2
3  public class TestStudent {
4      public static void main(String[] args) {
5          Student student = new Student(111223333, 1970, 5, 3);
6          BirthDate date = student.getBirthDate();
7          date.setYear(2010); //Now the student birth year is changed!
8      }
9  }
10
```

What Class is Immutable?

- For a class to be immutable:
  - it must mark all data fields private and
  - provide no mutator methods and
  - no accessor methods that would return a reference to a mutable data field object.

Passing Objects to Methods

Passing Objects to Methods

- Like passing an array, passing an object is actually passing the reference of the object.
- Java uses exactly one mode of passing arguments: ***pass-by-value***.
  - Passing by value for primitive type value (the value is passed to the parameter)
  - Passing by value for reference type value (the value is the reference to the object

TestPassObject.java

```
1  package chapter07;
2
3  public class TestPassObject {
4    /** Main method */
5    public static void main(String[] args) {
6      // Create a Circle object with radius 1
7      Circle3 myCircle = new Circle3(1);
8
9      // Print areas for radius 1, 2, 3, 4, and 5.
10     int n = 5;
11     printAreas(myCircle, n);
12
13     // See myCircle.radius and times
14     System.out.println("\n" + "Radius is " + myCircle.getRadius());
15     System.out.println("n is " + n);
16   }
```

TestPassObject.java

```
17
18      /** Print a table of areas for radius */
19      public static void printAreas(Circle3 c, int times) {
20          System.out.println("Radius \t\tArea");
21          while (times >= 1) {
22              System.out.println(c.getRadius() + "\t\t" + c.getArea());
23              c.setRadius(c.getRadius() + 1);
24              times--;
25          }
26      }
27  }
```

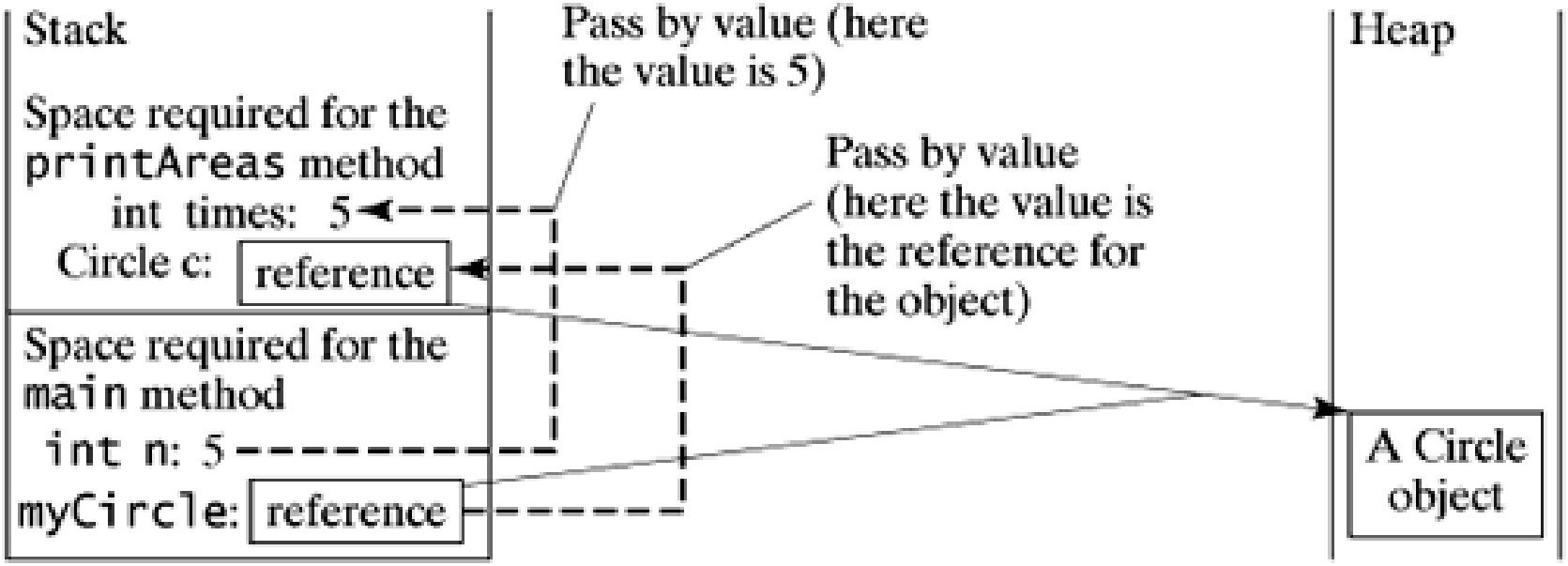
TestPassObject.java

- The output:

```
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
```

Passing Objects to Methods

- The figure shows the call stack for executing the methods in the program. Note that the objects are stored in a heap.



The Scope of Variables

The Scope of Variables

- In Methods chapter, discussed local variables and their scope rules.
- Local variables are declared and used inside a method locally.
- This section discusses the scope rules of all the variables in the context of a class.

The Scope of Variables

- **Local variables:**
  - A variable defined inside a method is referred to as a local variable.
  - The scope of a local variable starts from its declaration and continues to the end of the block that contains the variable.
  - A local variable must be initialized explicitly before it can be used.

The Scope of Variables

- **Instance and static variables:**
  - Instance and static variables in a class are referred to as the ***class's variables*** or ***data fields***.
  - The scope of a class's variables is the entire class, regardless of where the variables are declared.
  - A class's variables and methods can be declared in any order in the class

- You can declare a class's variable only once, but you can declare the same variable name in a method many times in different non-nesting blocks.

The Scope of Variables

- Example:

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

  private double radius = 1;
}
```

(a) variable radius and method getArea() can be declared in any order

```
public class Foo {
  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

- 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

```
1  package chapter07;
2
3  class Foo {
4    int x = 0; // instance variable
5    int y = 0;
6
7    public static void main(String[] args) {
8      Foo f = new Foo();
9      f.p();
10     System.out.println("After calling p() method");
11     System.out.println("f.x = " + f.x);
12     System.out.println("f.y = " + f.y);
13   }
14
15   void p() {
16     int x = 1; // local variable
17     System.out.println("Inside of p() method");
18     System.out.println("x = " + x);
19     System.out.println("y = " + y);
20   }
21 }
```

The Scope of Variables

- As demonstrated in the example, it is easy to make mistakes.
- To avoid confusion, do not declare the same variable name twice in a class, except for method parameters.

Array of Objects

Array of Objects

- Before arrays of primitive type elements were created. You can also create arrays of objects.
- The following statement declares and creates an array of ten Circle objects:

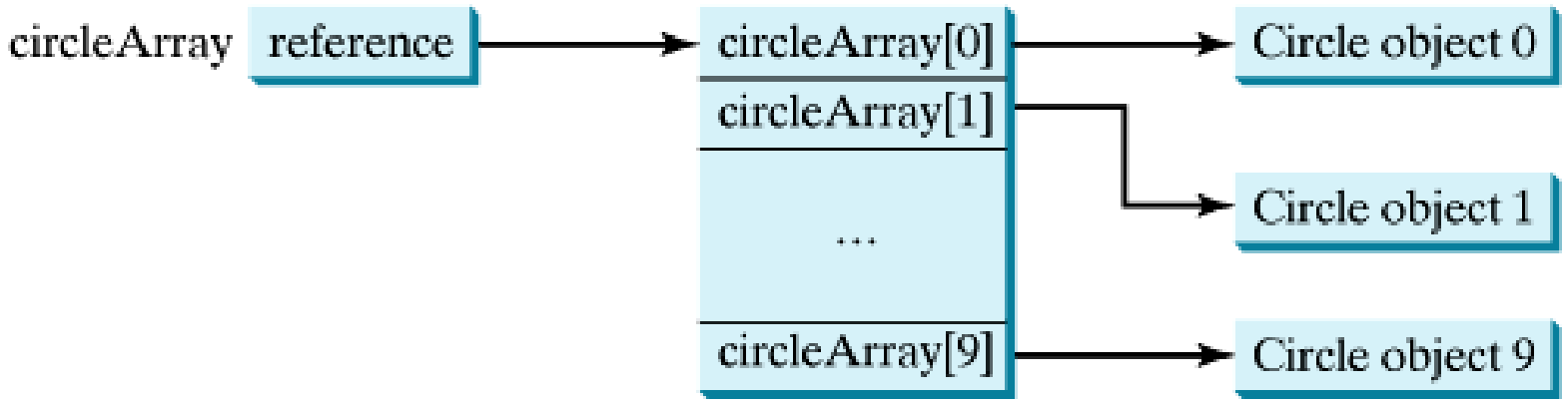
```
Circle[ ] circleArray = new Circle[10];
```

- To initialize the circleArray, you can use a for loop like this one:

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

Array of Objects

- An array of objects is actually an array of reference variables.
- So invoking circleArray[1].getArea() involves two levels of referencing



TotalArea.java

- TotalArea program summarizes the areas of an array of circles.
- The program creates circleArray, an array composed of ten Circle3 objects
- It then initializes circle radii with random values, and displays the total area of the circles in the array.

TotalArea.java

```
1  package chapter07;
2
3  public class TotalArea {
4    /** Main method */
5    public static void main(String[] args) {
6      // Declare circleArray
7      Circle3[] circleArray;
8
9      // Create circleArray
10     circleArray = createCircleArray();
11
12     // Print circleArray and total areas of the circles
13     printCircleArray(circleArray);
14   }
15
16   /** Create an array of Circle objects */
17   public static Circle3[] createCircleArray() {
18     Circle3[] circleArray = new Circle3[10];
19
20     for (int i = 0; i < circleArray.length; i++) {
21       circleArray[i] = new Circle3(Math.random() * 100);
22     }
```

TotalArea.java

```
23
24        // Return Circle array
25        return circleArray;
26    }
27
28    /** Print an array of circles and their total area */
29    public static void printCircleArray
30          (Circle3[] circleArray) {
31        System.out.println("Radius\t\t\t\t" + "Area");
32        for (int i = 0; i < circleArray.length; i++) {
33          System.out.print(circleArray[i].getRadius() + "\t\t" +
34                circleArray[i].getArea() + '\n');
35        }
36
37        System.out.println("----------------------------------------");
38
39        // Compute and display the result
40        System.out.println("The total areas of circles is \t" +
41              sum(circleArray));
42    }
```

TotalArea.java

```
43
44      /** Add circle areas */
45      public static double sum(Circle3[] circleArray) {
46        // Initialize sum
47        double sum = 0;
48
49        // Add areas to sum
50        for (int i = 0; i < circleArray.length; i++)
51          sum += circleArray[i].getArea();
52
53        return sum;
54      }
55    }
```

TotalArea.java

- The output:

```
Radius                                  Area
58.06880427956989 6                     10593.406541297387
36.33710413653297                       4148.112246400217
85.02001103760188                       22708.695490093254
99.67002343283416                       31208.938214899797
68.99814612628313                       14956.318906523336
66.51192311899847                       13897.890417494793
79.79530733791314                       20003.43485868224
11.2738794456952                        399.29755019510003
43.04292750675902                       5820.408629351761
43.85596734227498                       6042.369260300506
---------------------------------------
The total areas of circles is   129778.8721152384
```

References

References

- Y. Daniel Liang, **Introduction to Java Programming**, Sixth Edition, Pearson Education, 2007. (Chapter 6)

- S. Zakhour, S. Hommel, J. Royal, I. Rabinovitch, T. Risser, M. Hoeber, **The Java Tutorial: A Short Course on the Basics**, 4th Edition, Prentice Hall, 2006. (Chapter 4)

The End