What are Objects?

- Object-oriented programming (OOP) involves programming using objects.
- An **object** represents an entity in the real world that can be distinctly identified
- Objects are reusable software components that model real world items.
- Humans think in terms of objects, for instance an animal, a planet, a car, a student, a desk, a circle, a button, and even a loan can all be viewed as objects.
- An object has a unique identity, state, and behavior.

Object State

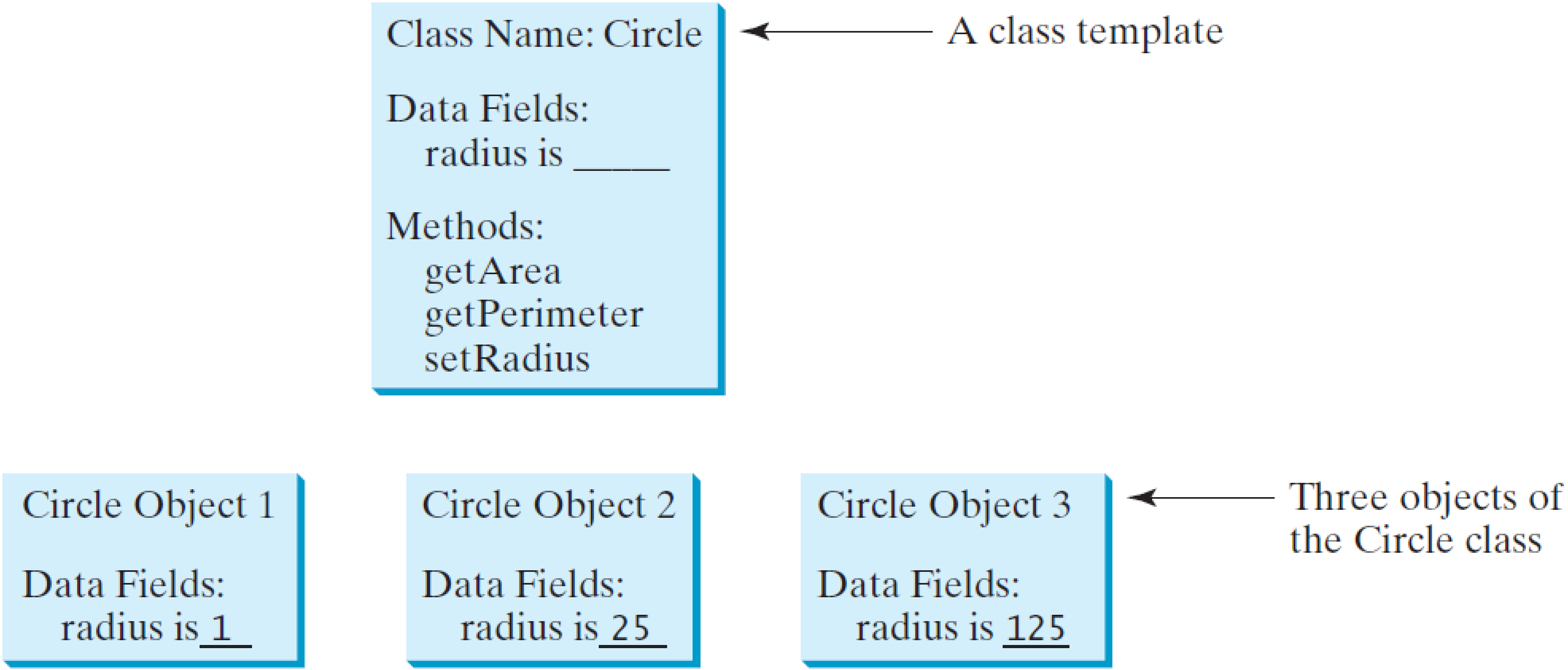
- The **state** of an object (also known as its properties or attributes) is represented by data fields with their current values. (e.g., size, shape, color and weight)
- A circle object, for example, has a data field **radius**, which is the property that characterizes a circle.
- A rectangle object has the data fields **width** and **height**, which are the properties that characterize a rectangle.

Object Behavior

- The **behavior** of an object (also known as its **actions**) is defined by **methods**.
- To invoke a method on an object is to ask the object to perform an action.
- For example, you may define methods named **getArea()** and **getPerimeter()** for circle objects.
- A circle object may invoke **getArea()** to return its area and **getPerimeter()** to return its perimeter.
- You may also define the **setRadius(radius)** method. A circle object can invoke this method to change its radius.

Defining Classes for Objects

- Objects of the same type are defined using a common class.
- A **class** is a template, blueprint, or **contract** that defines what an object's data fields and methods will be.
- Additionally, a class provides methods of a special type, known as **constructors**, which are invoked to create a **new object**.
- Constructor is used to initialize the state of an object.
- 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.
- The relationship between classes and objects is analogous to that between an apple-pie recipe and apple pies:
  - You can make as many apple pies as you want from a single recipe.



Class Name: Circle

Data Fields:
radius is _____

Methods:
getArea
getPerimeter
setRadius

A class template

Circle Object 1

Data Fields:
radius is 1

Circle Object 2

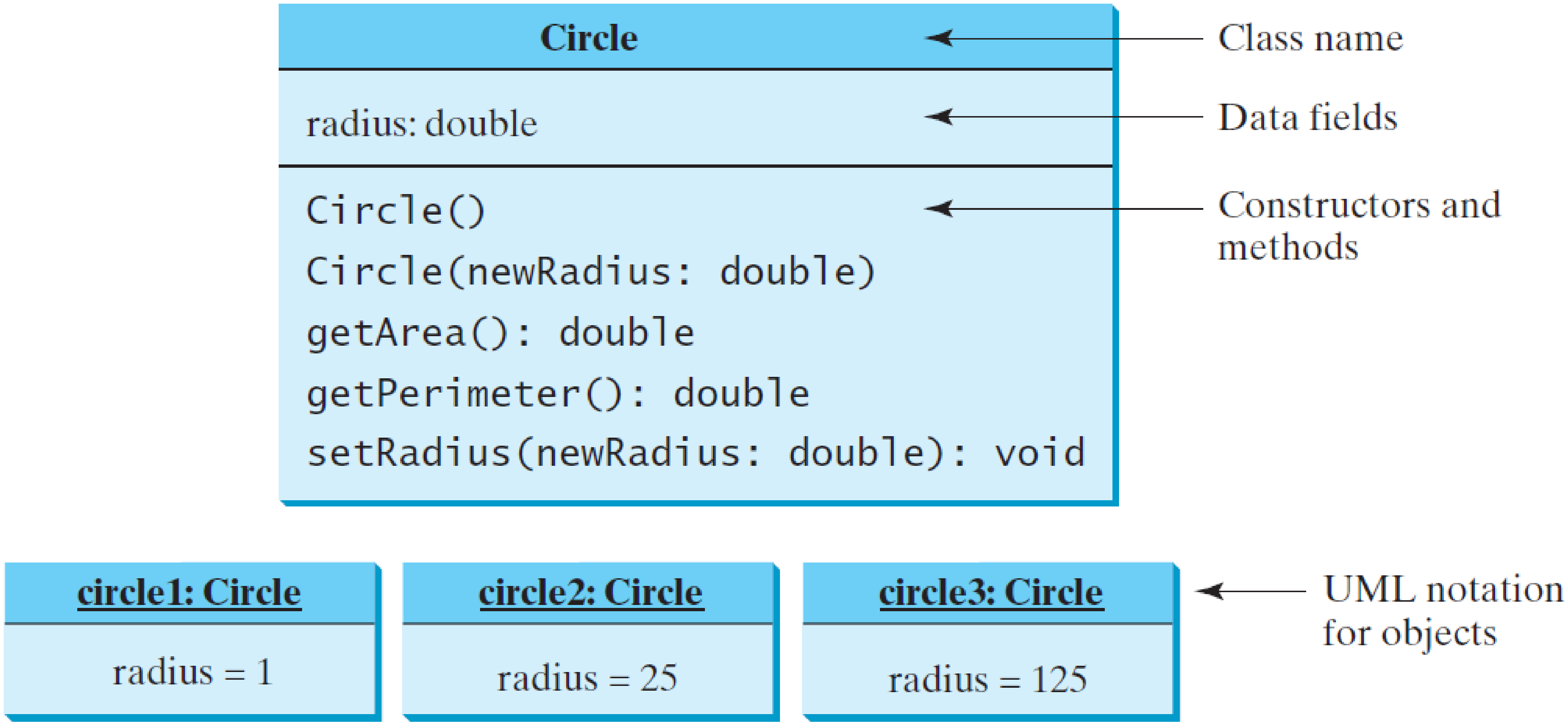
Data Fields:
radius is 25

Circle Object 3

Data Fields:
radius is 125

Three objects of the Circle class

UML Class Diagram



Circle

radius: double

Circle()

Circle(newRadius: double)

getArea(): double

getPerimeter(): double

setRadius(newRadius: double): void

Class name

Data fields

Constructors and methods

circle1: Circle

radius = 1

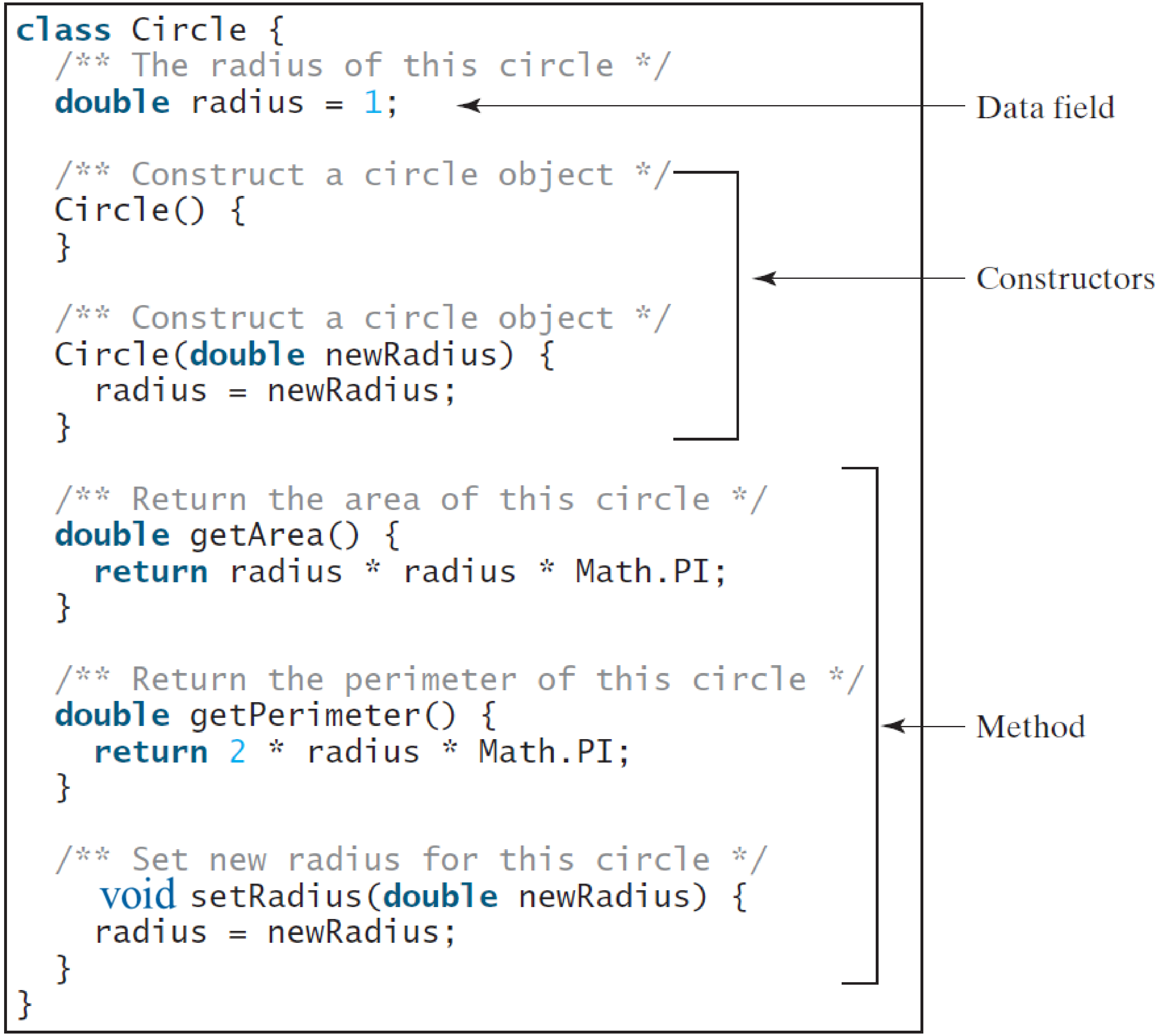
circle2: Circle

radius = 25

circle3: Circle

radius = 125

UML notation for objects



```java
class Circle {
  /** The radius of this circle */
  double radius = 1;

  /** Construct a circle object */
  Circle() {
  }

  /** Construct a circle object */
  Circle(double newRadius) {
    radius = newRadius;
  }

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

  /** Return the perimeter of this circle */
  double getPerimeter() {
    return 2 * radius * Math.PI;
  }

  /** Set new radius for this circle */
    void setRadius(double newRadius) {
    radius = newRadius;
  }
}
```

Constructing Objects Using Constructors

- A constructor is invoked to create an object using the new operator.
  - A constructor 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.

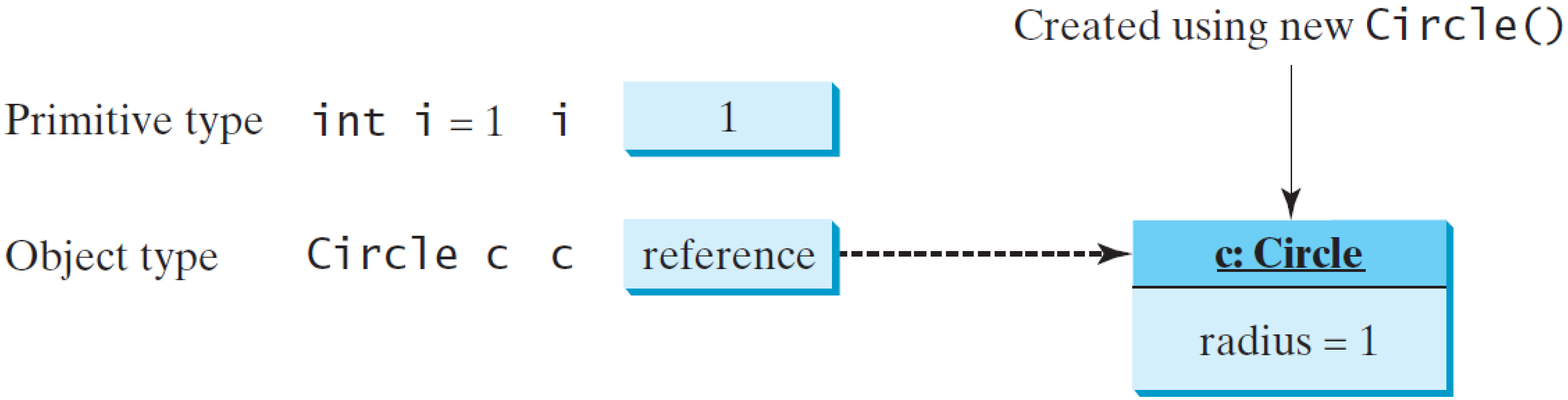
```
ClassName objectRefVar;
objectRefVar = new ClassName();

ClassName objectRefVar = new ClassName();
```

Import Declaration

- Helps the compiler locate a class that is used in this program.
- Rich set of predefined classes that you can reuse rather than “reinventing the wheel.”
- Classes are grouped into packages—named groups of related classes—and are collectively referred to as the Java class library, or the Java Application Programming Interface (Java API).
- You use keyword import to identify the predefined classes used in a Java program.

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.
- For example 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 memory.



Created using new `Circle()`

Primitive type `int i` = 1 `i` 1

Object type `Circle c` `c` reference

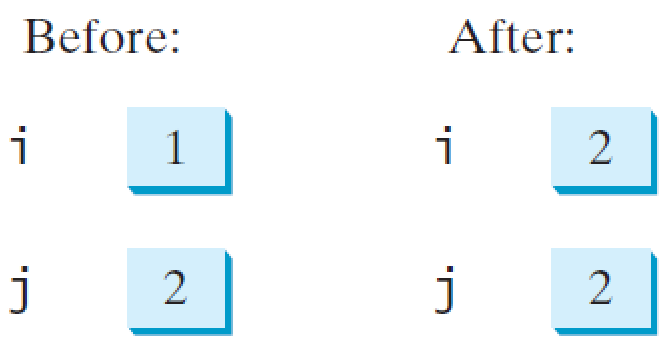
c: Circle

radius = 1

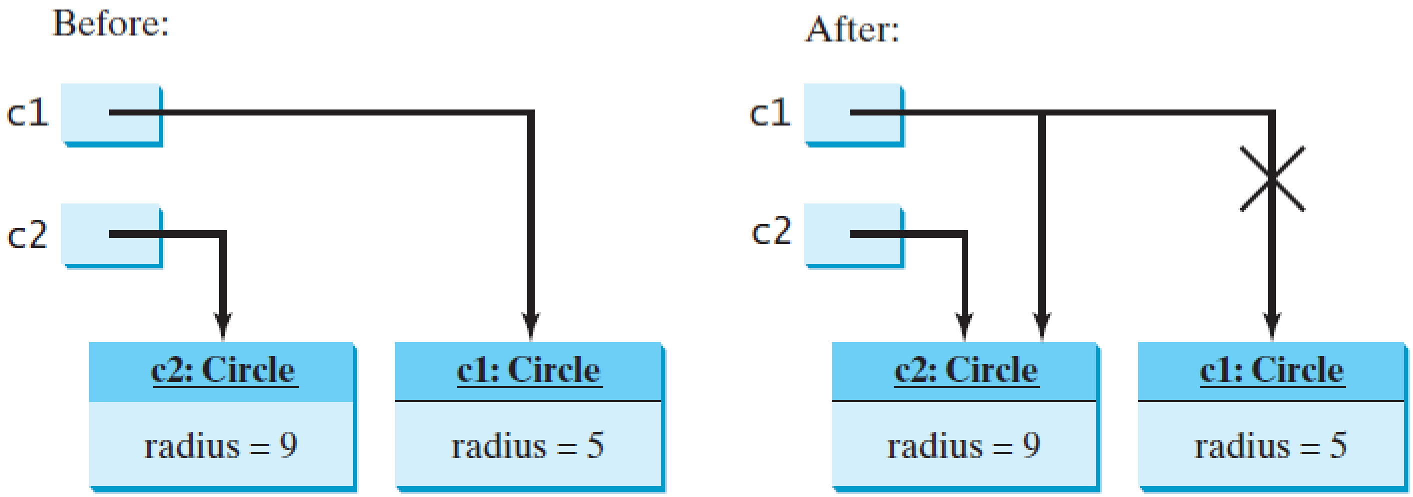
Differences between Variables of Primitive Types and Reference Types (Cont.)

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

Primitive type assignment `i = j`



Object type assignment `c1 = c2`



Accessing Objects via Reference Variables

- An object’s data and methods can be accessed through the dot (.) operator via the object’s reference variable.
  - **`objectRefVar.dataField`** references a data field in the object.
  - **`objectRefVar.method(arguments)`** invokes a method on the object.

```
Scanner input = new Scanner(System.in);
int i = input.nextInt();
```

Static Variables and Methods

- A static variable is shared by **all** objects of the class. A static method **cannot** access instance members of the class.
- If you want all the instances of a class to share data, use static variables, also known as class variables.
  - **`ClassName.dataField`** references a static data field in the objects.
- Static methods can be called **without** creating an instance of the class.
  - **`ClassName.method(arguments)`** invokes a static method in the class.
- To declare a static variable or define a static method, put the modifier **static** in the variable or method declaration.

Visibility Modifiers

- Visibility modifiers can be used to specify the visibility of a class and its members.
- You can use the **public** visibility modifier 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.

- If no visibility modifier is used, then by default the classes, methods, and data fields are accessible by any class in the same package. This is known as **package-private** or **package-access**.

- To **prevent** direct modifications of data fields, you should declare the data fields private, using the **private** modifier. This is known as **data field encapsulation**.

Analyzing Our First Java Program

- What is System?
    - **class**
- What is System.out?
    - **PrintStream object**
    - Standard output object.
    - Allows Java applications to display strings in the command window from which the Java application executes.
- What about System.out.println()?!
    - **A method within PrintStream class**

Problem

- Find the sum of integers from 1 to 10, from 20 to 30, and from 35 to 45, respectively.

Problem

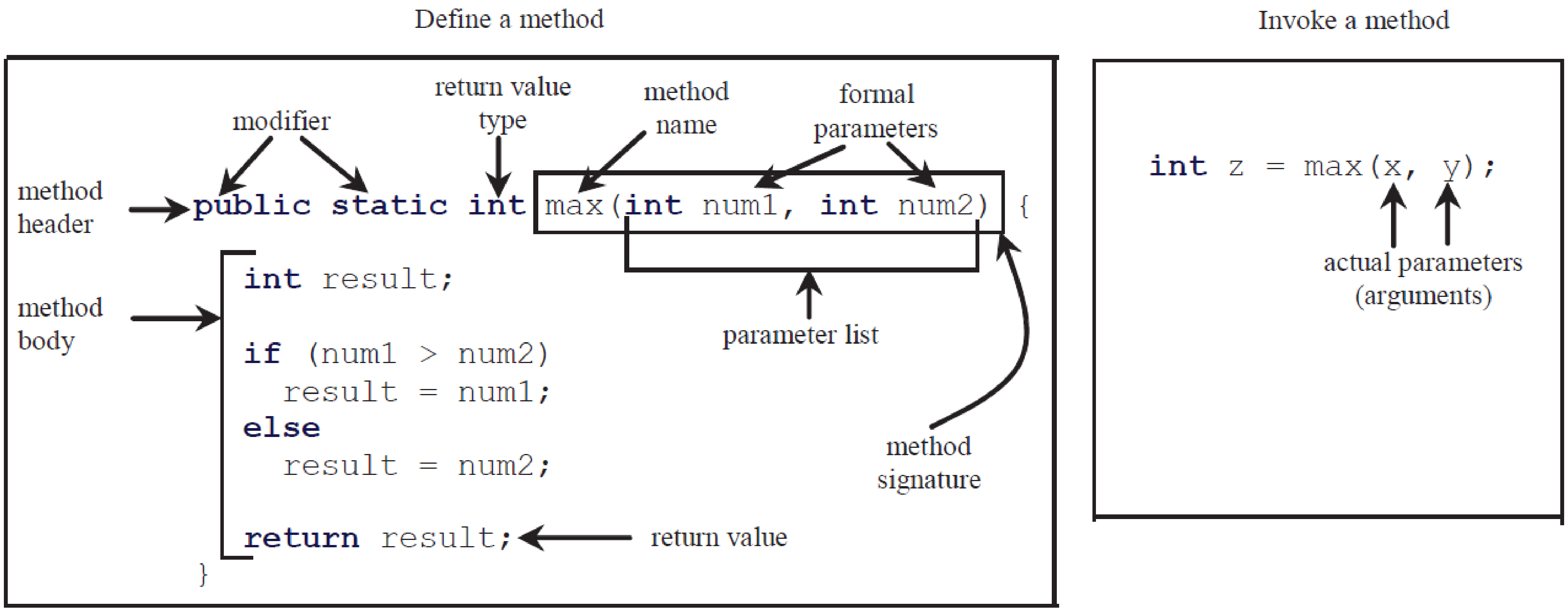
```
int sum = 0;
for (int i = 1; i <= 10; i++)
      sum += i;
System.out.println("Sum from 1 to 10 is " + sum);

sum = 0;
for (int i = 20; i <= 30; i++)
      sum += i;
System.out.println("Sum from 20 to 30 is " + sum);

sum = 0;
for (int i = 35; i <= 45; i++)
      sum += i;
System.out.println("Sum from 35 to 45 is " + sum);
```

Solution

```
public static int sum(int i1, int i2) {
   int sum = 0;
   for (int i = i1; i <= i2; i++)
      sum += i;
   return sum;
}

public static void main(String[] args) {
System.out.println("Sum from 1 to 10 is " + sum(1, 10));
System.out.println("Sum from 20 to 30 is " + sum(20, 30));
System.out.println("Sum from 35 to 45 is " sum(35, 45));
}
```

Defining Methods

- A method is a collection of statements that are grouped together to perform an operation.

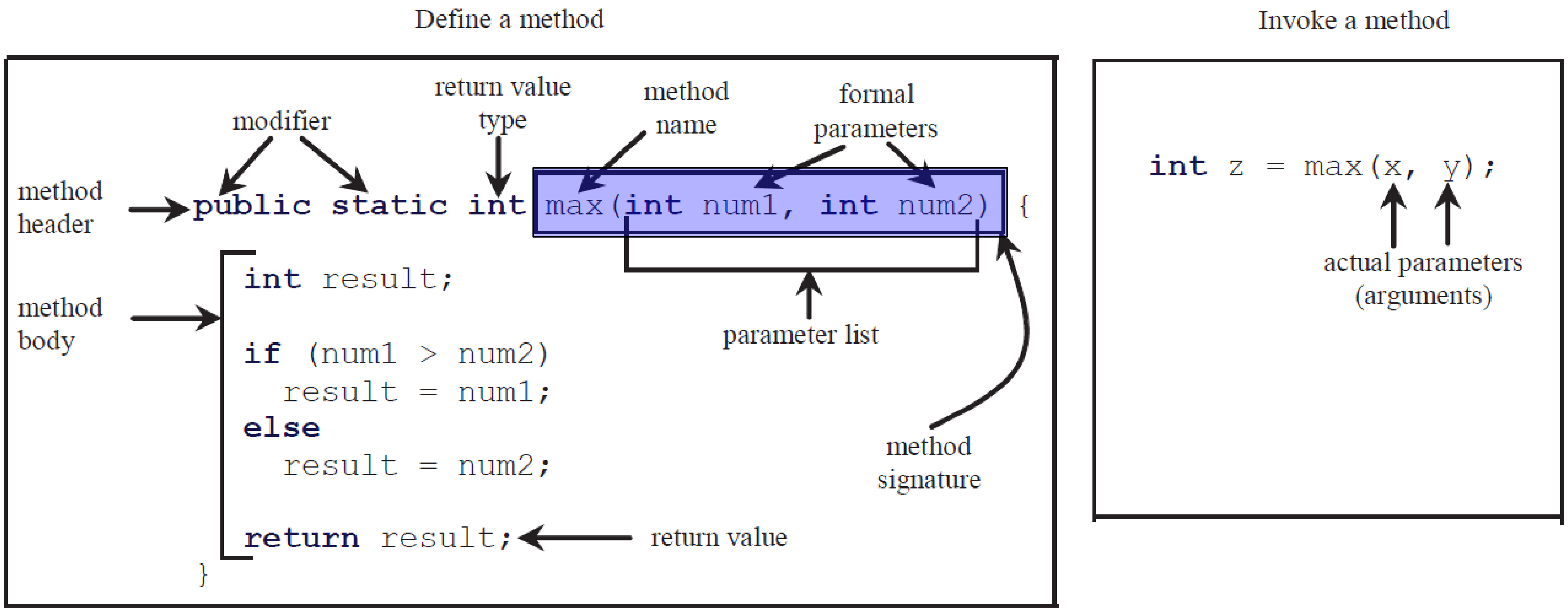


```
public static int max(int num1, int num2) {
   int result;

   if (num1 > num2)
     result = num1;
   else
     result = num2;

   return result;
}
```

```
int z = max(x, y);
```

Method Signature

- *Method signature* is the combination of the method name and the parameter list.



Define a method

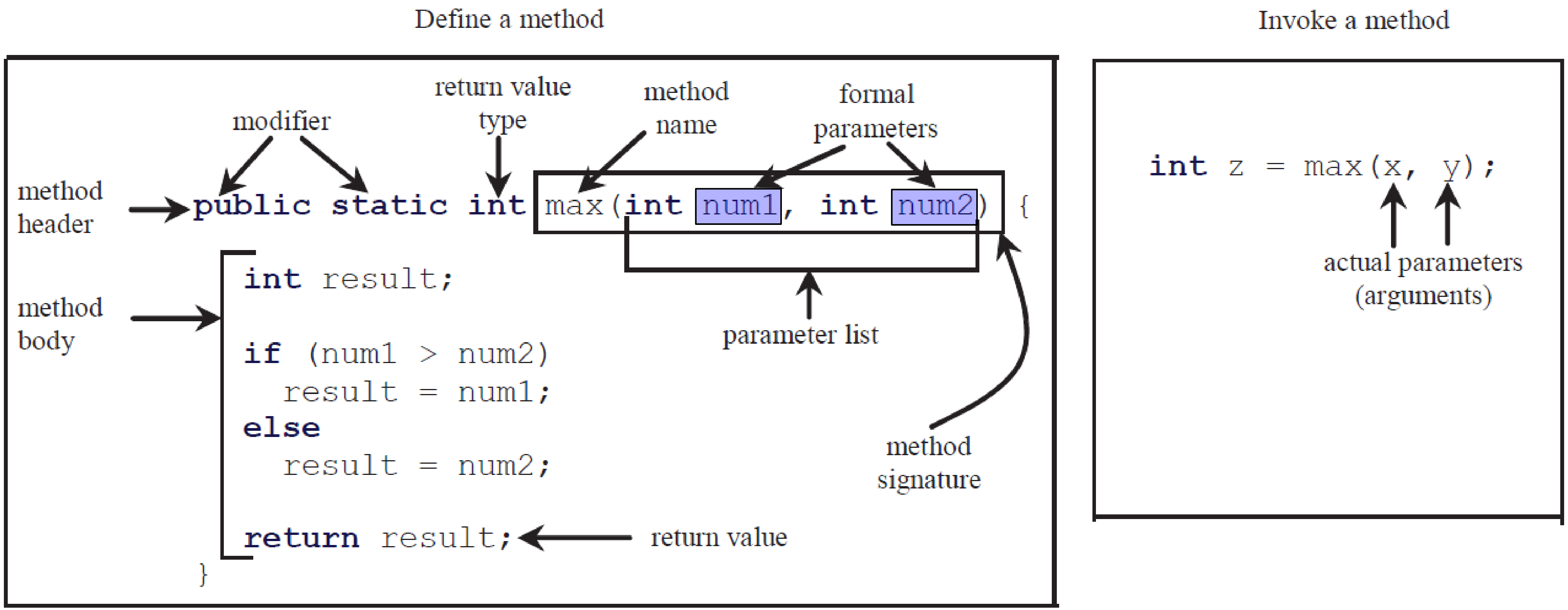
```
public static int max(int num1, int num2) {
   int result;

   if (num1 > num2)
     result = num1;
   else
     result = num2;

   return result;
}
```

Invoke a method

```
int z = max(x, y);
```

Formal Parameters

- The variables defined in the method header are known as *formal parameters*.



Define a method

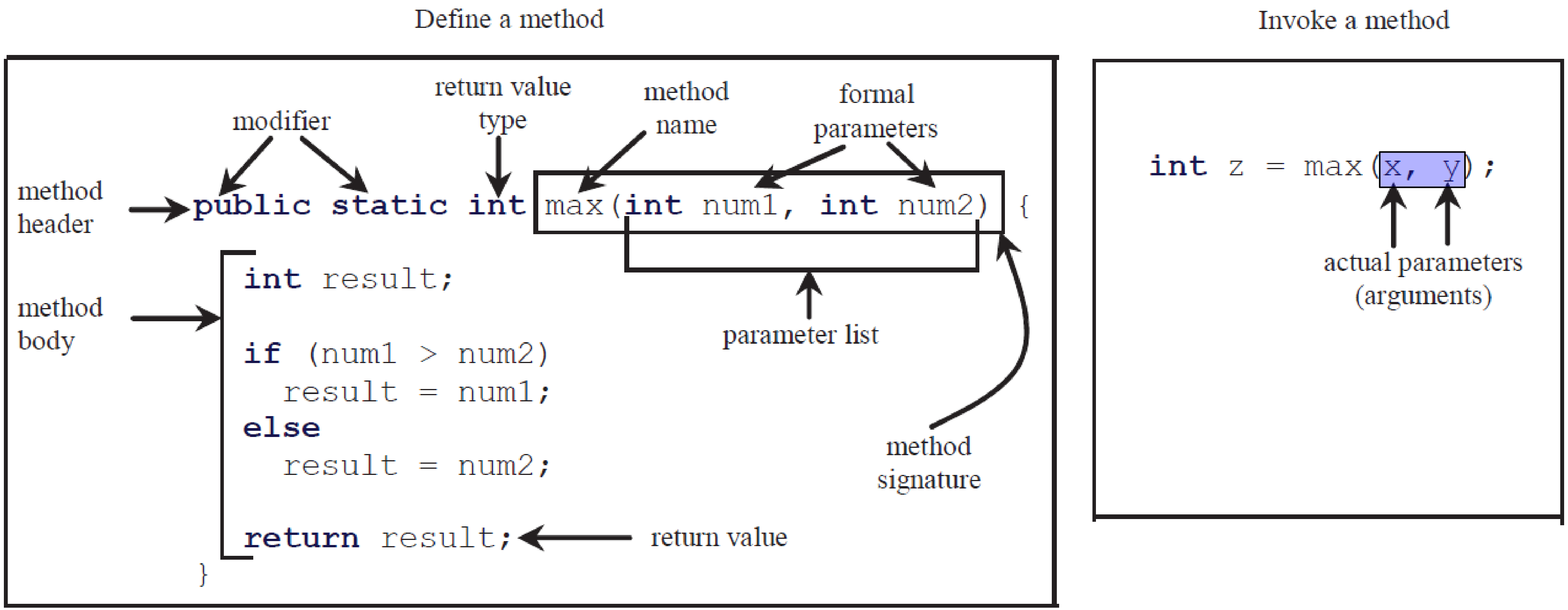
```
public static int max(int num1, int num2) {
  int result;

  if (num1 > num2)
    result = num1;
  else
    result = num2;

  return result;
}
```

Invoke a method

```
int z = max(x, y);
```

Actual Parameters

- When a method is invoked, you pass a value to the parameter. This value is referred to as *actual parameter or argument*.



Define a method

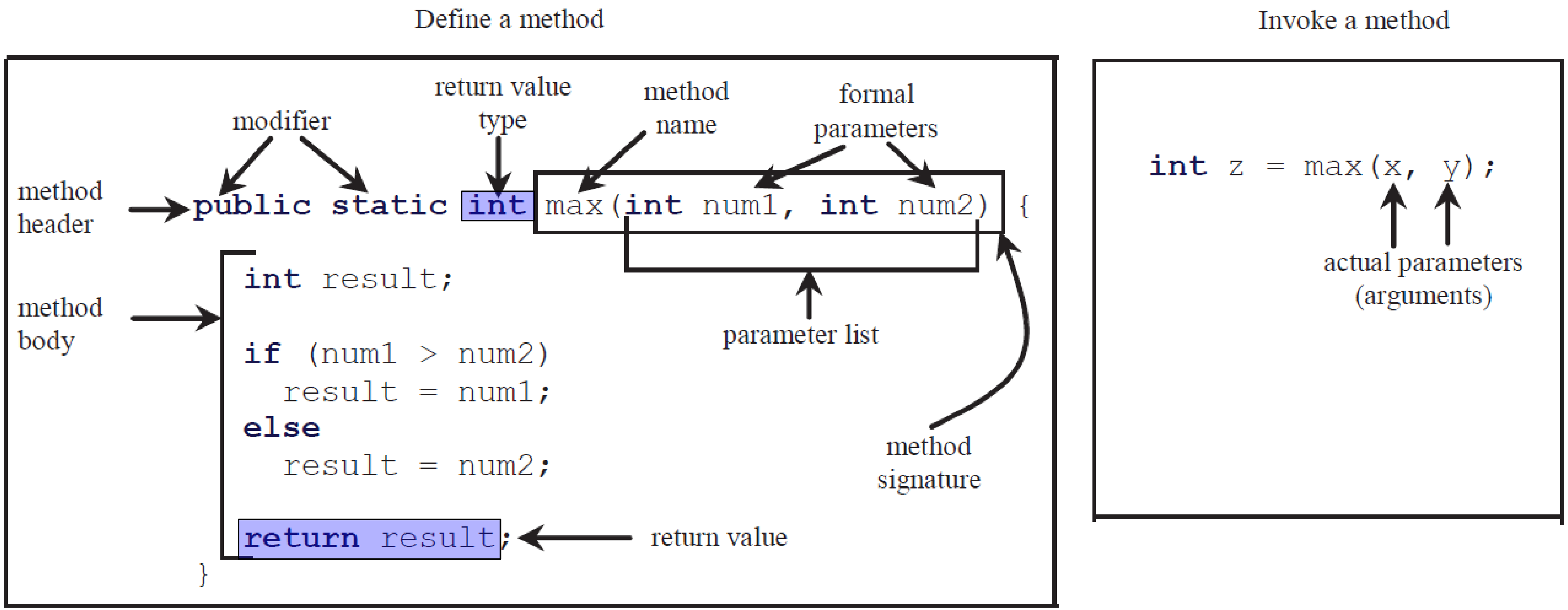
```
public static int max(int num1, int num2) {
  int result;

  if (num1 > num2)
    result = num1;
  else
    result = num2;

  return result;
}
```

Invoke a method

```
int z = max(x, y);
```

Return Value Type

- A method may return a value. The <u>returnValueType</u> is the data type of the value the method returns. If the method does not return a value, the <u>returnValueType</u> is the keyword <u>void</u>. For example, the <u>returnValueType</u> in the <u>main</u> method is <u>void</u>.



Define a method

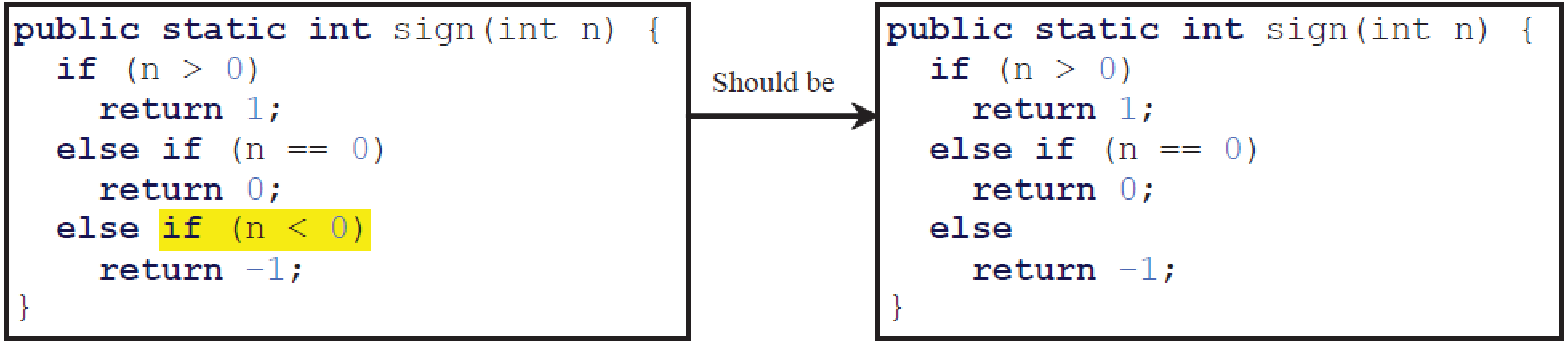
```
public static int max(int num1, int num2) {
  int result;

  if (num1 > num2)
    result = num1;
  else
    result = num2;

  return result;
}
```

Invoke a method

```
int z = max(x, y);
```

CAUTION

- A return statement is required for a value-returning method. The method shown below in (a) is logically correct, but it has a compilation error because the Java compiler thinks it’s possible that this method does not return any value.



```
public static int sign(int n) {
  if (n > 0)
    return 1;
  else if (n == 0)
    return 0;
  else if (n < 0)
    return -1;
}
```

Should be

```
public static int sign(int n) {
  if (n > 0)
    return 1;
  else if (n == 0)
    return 0;
  else
    return -1;
}
```

(a) (b)

- To fix this problem, delete *if (n < 0)* in (a), so that the compiler will see a return statement to be reached regardless of how the if statement is evaluated.

void Method

- This type of method does not return a value. The method performs some actions.

Passing Parameters

```
public static void nPrintln(String message, int n) {
   for (int i = 0; i < n; i++)
      System.out.println(message);
}
```

- Suppose you invoke the method using

```
nPrintln(“Welcome to Java”, 5);
```

- What is the output?

- Suppose you invoke the method using

```
nPrintln(“Computer Science”, 15);
```

- What is the output?

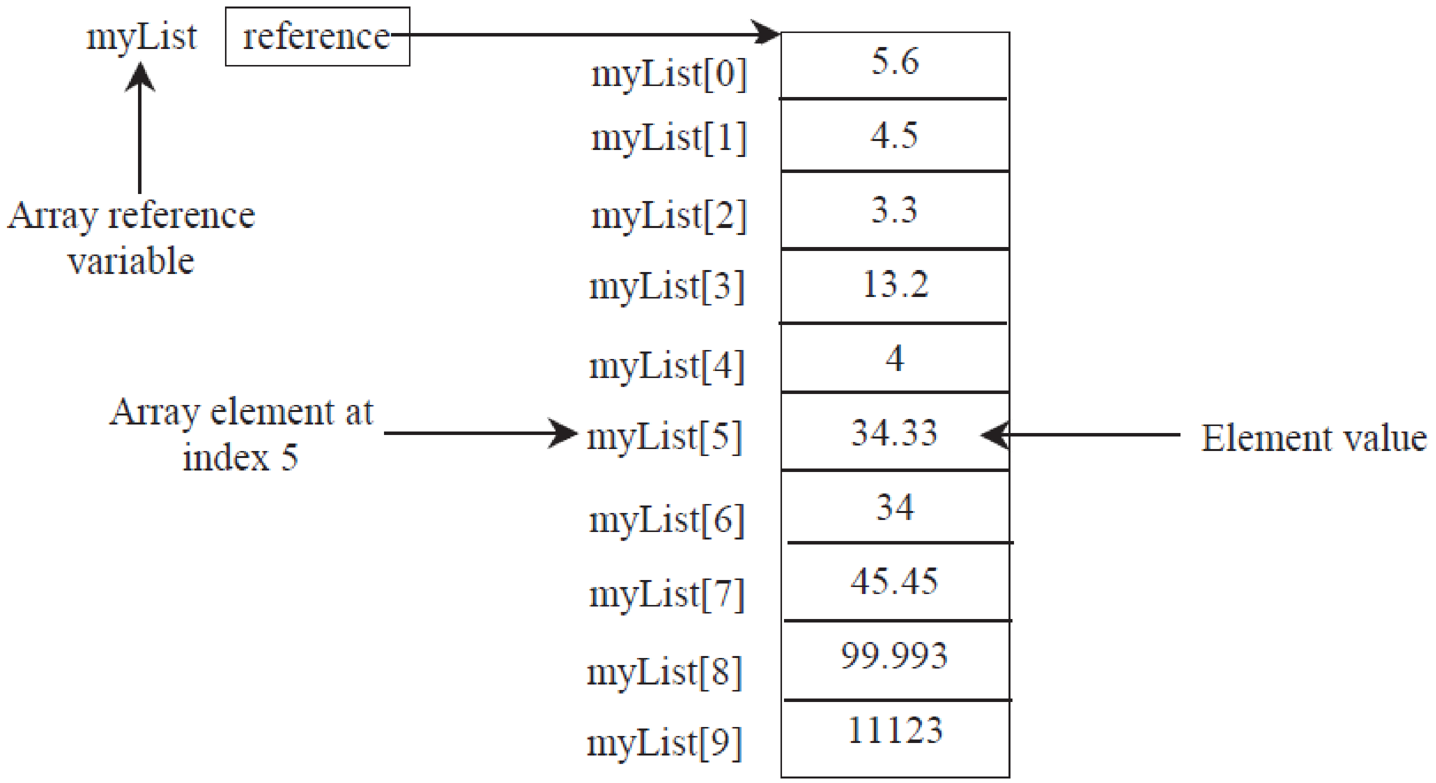
Example

- Create a circle class according to the UML that you have seen early

Introducing Arrays

- Array is a data structure that represents a collection of the same types of data.

```
double[] myList = new double[10];
```



myList reference

Array reference variable

myList[0] 5.6
myList[1] 4.5
myList[2] 3.3
myList[3] 13.2
myList[4] 4
myList[5] 34.33
myList[6] 34
myList[7] 45.45
myList[8] 99.993
myList[9] 11123

Array element at index 5

Element value

Declaring Array Variables

- `datatype[] arrayRefVar;`

  Example:

  ```
  double[] myList;
  ```

- `datatype arrayRefVar[];` // This style is allowed, but not preferred

  Example:

  ```
  double myList[];
  ```

Creating Arrays

```
arrayRefVar = new datatype[arraySize];
```

Example:

```
myList = new double[10];
```

`myList[0]` references the first element in the array.

`myList[9]` references the last element in the array.

Declaring and Creating in One Step

➢ `datatype[] arrayRefVar = new    datatype[arraySize];`

Example :

```
double[] myList = new double[10];
```

➢ `datatype arrayRefVar[] = new    datatype[arraySize];`

Example :

```
double myList[] = new double[10];
```

The Length of an Array

Once an array is created, its size is fixed. It cannot be changed. You can find its size using

```
    arrayRefVar.length
```

For example,

```
    myList.length returns 10
```

Indexed Variables

- The array elements are accessed through the index. The array indices are *0-based*, i.e., it starts from 0 to arrayRefVar.length-1.

- Example :

  myList holds ten double values and the indices are from 0 to 9.

- Each element in the array is represented by using the following syntax, known as an *indexed variable*:

```
arrayRefVar[index];
```

Using Indexed Variables

- After an array is created, an indexed variable can be used in the same way as a regular variable.

- For example, the following code adds the value in myList[0] and myList[1] to myList[2].

```
myList[2] = myList[0] + myList[1];
```

Array Initializers

```
double[] myList = new double[4];
myList[0] = 1.9;
myList[1] = 2.9;
myList[2] = 3.4;
myList[3] = 3.5;
```

Declaring, creating, initializing Using the Shorthand Notation

- Declaring, creating, initializing in one statement:

```
double[] myList = {1.9, 2.9, 3.4, 3.5};
```

- This shorthand notation is equivalent to the statements in the previous slide

- This shorthand syntax must be in one statement.

CAUTION

- Using the shorthand notation, you have to declare, create, and initialize the array all in one statement. Splitting it would cause a syntax error. For example, the following is wrong:

```
double[] myList;

myList = {1.9, 2.9, 3.4, 3.5};
```

Initializing arrays with input values

```
java.util.Scanner input = new
    java.util.Scanner(System.in);
System.out.print("Enter " + myList.length +
    " values: ");
for (int i = 0; i < myList.length; i++)
    myList[i] = input.nextDouble();
```

Initializing arrays with random values

```
for (int i = 0; i < myList.length; i++) {
    myList[i] = Math.random() * 100;
}
```

Printing arrays

```
for (int i = 0; i < myList.length; i++) {
    System.out.print(myList[i] + " ");
}
```

Summing all elements

```
double total = 0;
for (int i = 0; i < myList.length; i++) {
    total += myList[i];
}
```

Finding the largest element

```
double max = myList[0];
for (int i = 1; i < myList.length; i++) {
    if (myList[i] > max) max = myList[i];
}
```

Example

- Write a program that read an integer from user, create a new double array with a size of that integer, then assign a double random numbers from 0 to 100 then print all the array values, sum of the value and the max value. For printing use prnitf to print a value with two floating point

Inheritance

- *Inheritance* — Object-oriented programming allows you to define new classes from existing classes.
- Suppose you need to define classes to model circles, rectangles, and triangles.
  - These classes have many common features.
- Avoid redundancy;
- Easy to comprehend system;
- Easy to maintain system;

Inheritance (Cont.)

- A form of software reuse in which a new class is created by absorbing an existing class’s members and 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.

Inheritance (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
- Each 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.

Inheritance (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.

Superclasses and Subclasses

- Superclasses tend to be “more general” and subclasses “more specific.”
- Because every subclass object *is an* object 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.)

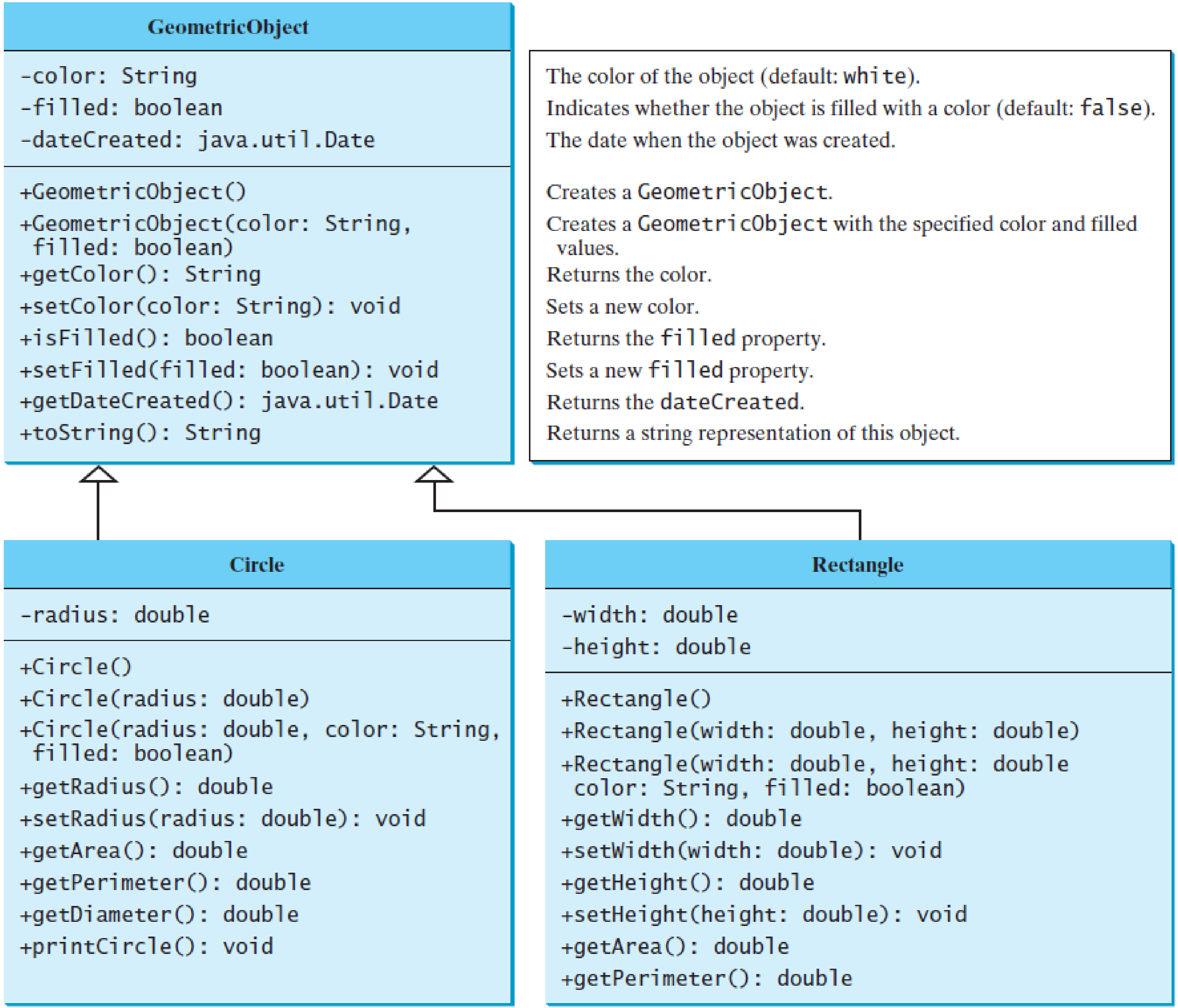
- Inheritance enables you to define a general class (i.e., a superclass) and later extend it to more specialized classes (i.e., subclasses).
- Different classes may have some common properties and behaviors, which can be generalized in a class that can be shared by other classes.
- You can define a specialized class that extends the generalized class.
  - The specialized classes inherit the properties and methods from the general class.

Superclasses and Subclasses (Cont.)

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

Superclasses and Subclasses - Example

- Consider geometric objects.
    - Model geometric objects such as circles and rectangles.
    - Geometric objects have many common properties and behaviors.
        - They can be drawn in a certain color and be filled or unfilled.
- A general class `GeometricObject` can be used to model all geometric objects.
    - Define the `Circle` class that extends the `GeometricObject` class.
    - `Rectangle` can also be defined as a subclass of `GeometricObject`.


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
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

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.

Constructors in Subclasses

- Instantiating a subclass object begins a chain of constructor calls
  - The subclass constructor, before performing its own tasks, invokes its direct superclass’s constructor
- If the superclassis derived from another class, the superclassconstructor invokes the constructor of the next class up the hierarchy, and so on.
- The last constructor called in the chain is always class `Object`’s constructor.
- Original subclass constructor’s body finishes executing last.
- Each superclass’s constructor manipulates the superclass instance variables that the subclass object inherits.

Object Class

- All classes in Java inherit directly or indirectly from `Object`, so its 11 methods are inherited by all other classes.
- Can learn more about Object's methods in the online API documentation and in *The Java Tutorial at :*

    java.sun.com/javase-/6/docs/api/java/lang/Object.html

or

    java.sun.com/docs/books/tutorial/java/IandI/objectclass.html

- Every array has an overridden clonemethod that copies the array.
  - If the array stores references to objects, the objects are not copied—a shallow copy is performed.