Objects and Classes

OO Programming Concepts

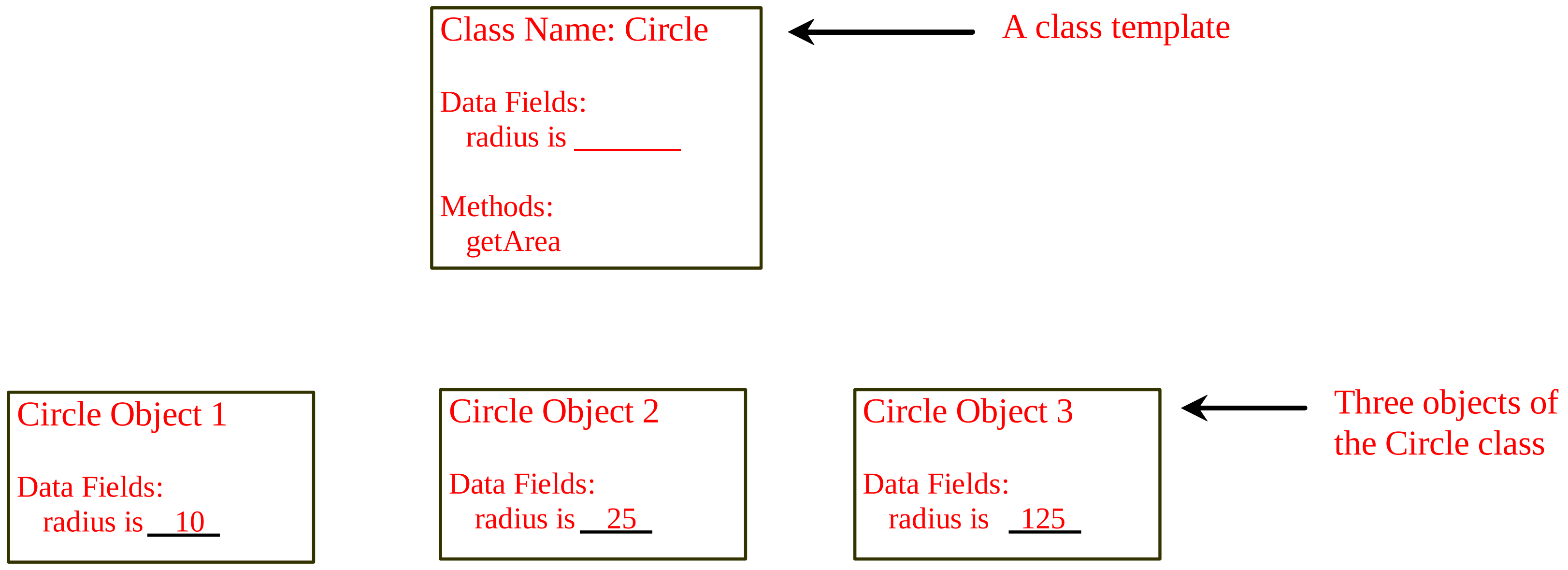
- Object-oriented programming (OOP) involves programming using objects. An *object* represents an entity in the real world that can be distinctly identified. For example, a student, a desk, a circle, a button, and even a loan can all be viewed as objects.

OO Programming Concepts

- 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. 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.

Objects

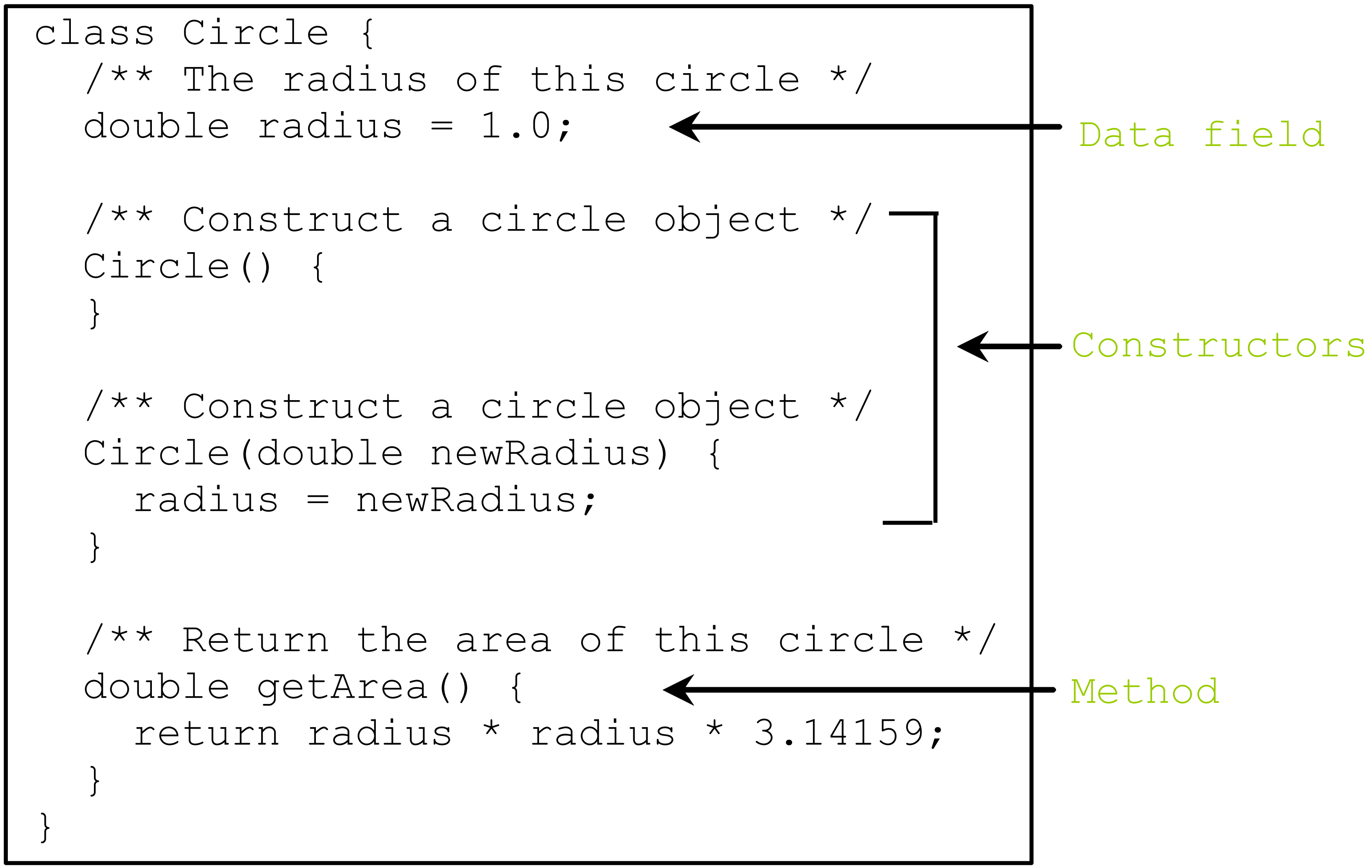


An object has both a state and behavior. The state defines the object, and the behavior defines what the object does.

Classes

Classes are constructs that define objects of the same type. A Java class uses variables to define data fields and methods to define behaviors. Additionally, a class provides a special type of methods, known as constructors, which are invoked to construct objects from the class.

Classes



```
class Circle {
  /** The radius of this circle */
  double radius = 1.0;

  /** 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 * 3.14159;
  }
}
```

Unified Modeling Language (UML) Class Diagram

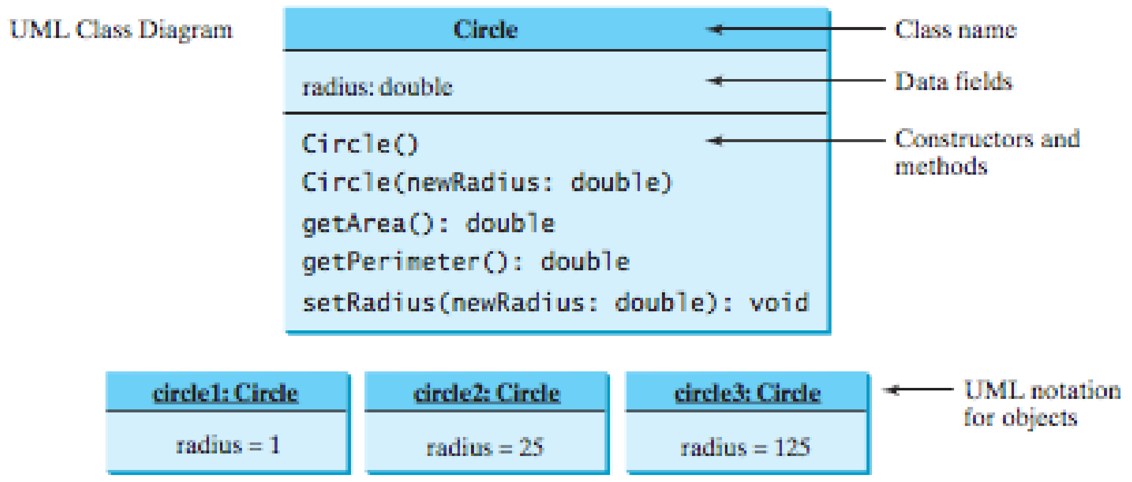


FIGURE 9.4 Classes and objects can be represented using UML notation.

In the class diagram, the data field is denoted as
dataFieldName: dataFieldType

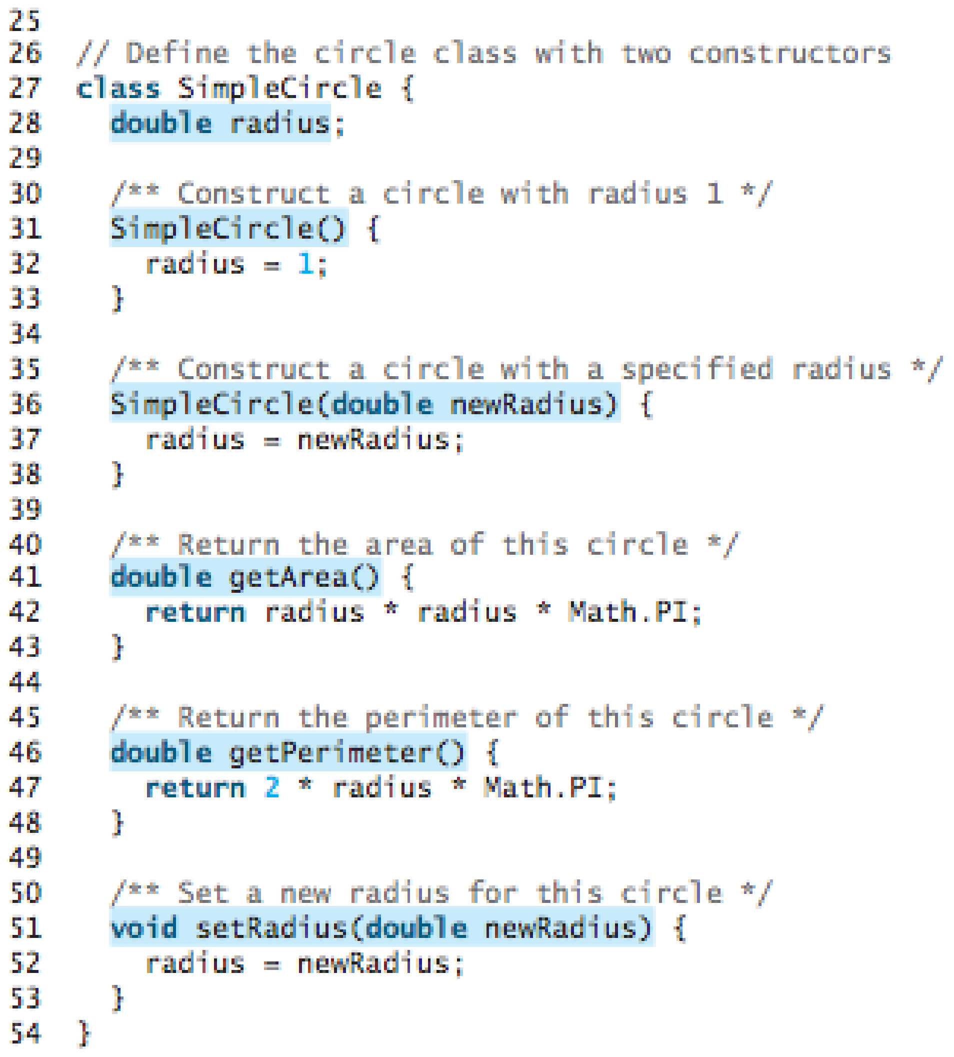
The constructor is denoted as:
ClassName(parameterName: parameterType)

The method is denoted as
methodName(parameterName: parameterType): returnType

Example: *SimpleCircle* class

```java
public class TestSimpleCircle {
  /** Main method */
  public static void main(String[] args) {
    // Create a circle with radius 1
    SimpleCircle circle1 = new SimpleCircle();
    System.out.println("The area of the circle of radius "
      + circle1.radius + " is " + circle1.getArea());

    // Create a circle with radius 25
    SimpleCircle circle2 = new SimpleCircle(25);
    System.out.println("The area of the circle of radius "
      + circle2.radius + " is " + circle2.getArea());

    // Create a circle with radius 125
    SimpleCircle circle3 = new SimpleCircle(125);
    System.out.println("The area of the circle of radius "
      + circle3.radius + " is " + circle3.getArea());

    // Modify circle radius
    circle2.radius = 100; // or circle2.setRadius(100)
    System.out.println("The area of the circle of radius "
      + circle2.radius + " is " + circle2.getArea());

  }
}
```

Example: *SimpleCircle* class



```java

// Define the circle class with two constructors
class SimpleCircle {
  double radius;

  /** Construct a circle with radius 1 */
  SimpleCircle() {
    radius = 1;
  }

  /** Construct a circle with a specified radius */
  SimpleCircle(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 a new radius for this circle */
  void setRadius(double newRadius) {
    radius = newRadius;
  }
}
```

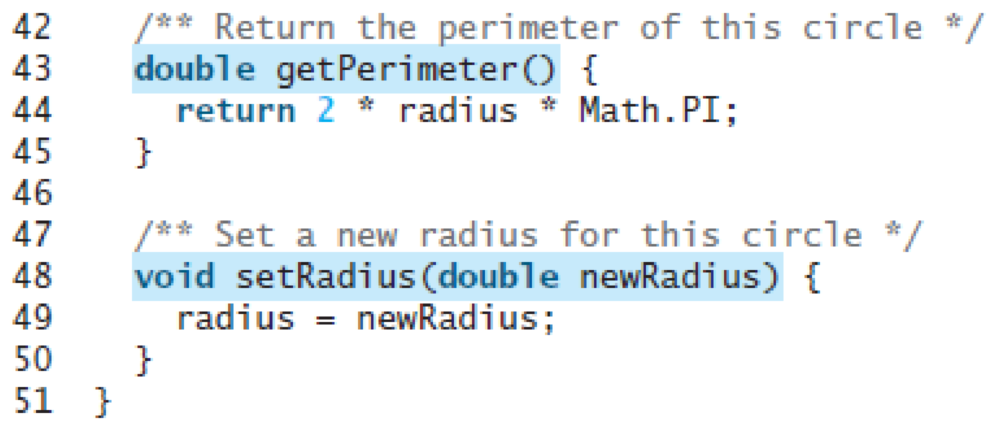
Example: *SimpleCircle* class

- The program contains two classes. The first of these, TestSimpleCircle, is the main class. Its sole purpose is to test the second class, SimpleCircle. Such a program that uses the class is often referred to as a *client* of the class. When you run the program, the Java runtime system 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. Therefore, the file name is **TestSimpleCircle.java**, since TestSimpleCircle is public.
- Each class in the source code is compiled into a **.class** file. When you compile **TestSimpleCircle.java**, two class files **TestSimpleCircle.class** and **SimpleCircle.class** are generated,

Combine Two Classes into One

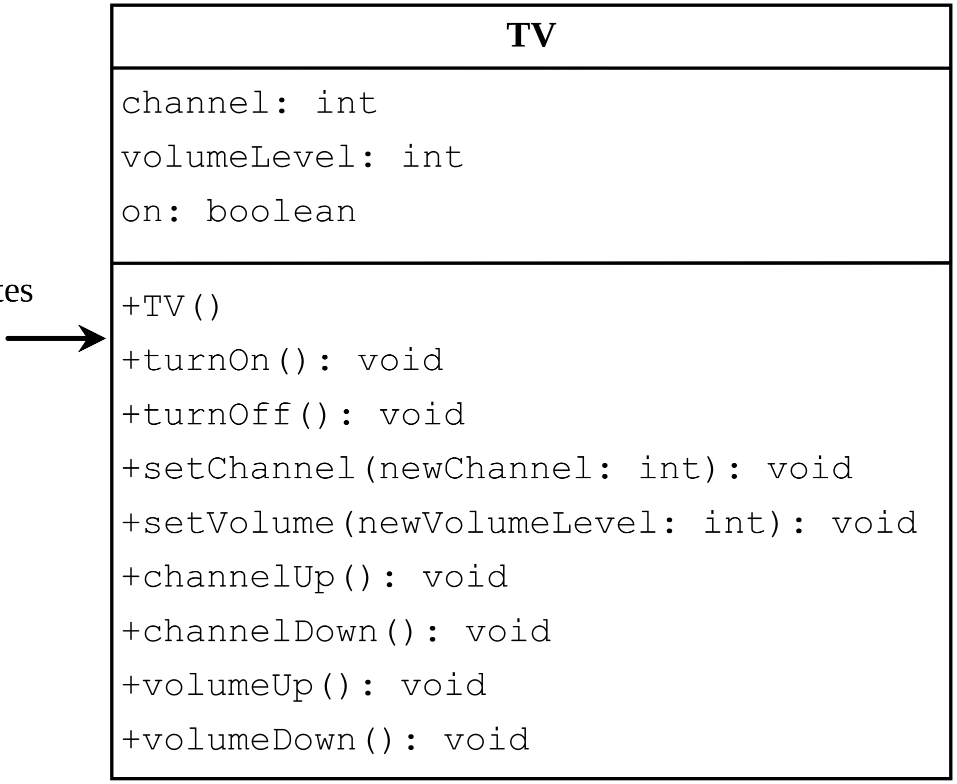
```java
public class SimpleCircle {
  /** Main method */
  public static void main(String[] args) {
    // Create a circle with radius 1
    SimpleCircle circle1 = new SimpleCircle();
    System.out.println("The area of the circle of radius "
      + circle1.radius + " is " + circle1.getArea());

    // Create a circle with radius 25
    SimpleCircle circle2 = new SimpleCircle(25);
    System.out.println("The area of the circle of radius "
      + circle2.radius + " is " + circle2.getArea());

    // Create a circle with radius 125
    SimpleCircle circle3 = new SimpleCircle(125);
    System.out.println("The area of the circle of radius "
      + circle3.radius + " is " + circle3.getArea());

    // Modify circle radius
    circle2.radius = 100;
    System.out.println("The area of the circle of radius "
      + circle2.radius + " is " + circle2.getArea());
  }

  double radius;

  /** Construct a circle with radius 1 */
  SimpleCircle() {
    radius = 1;
  }

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

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

```

Combine Two Classes into One



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

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

Example: Defining Classes and Creating Objects



The + sign indicates a public modifier.

TV	
`channel: int`	The current channel (1 to 120) of this TV.
`volumeLevel: int`	The current volume level (1 to 7) of this TV.
`on: boolean`	Indicates whether this TV is on/off.
`+TV()`	Constructs a default TV object.
`+turnOn(): void`	Turns on this TV.
`+turnOff(): void`	Turns off this TV.
`+setChannel(newChannel: int): void`	Sets a new channel for this TV.
`+setVolume(newVolumeLevel: int): void`	Sets a new volume level for this TV.
`+channelUp(): void`	Increases the channel number by 1.
`+channelDown(): void`	Decreases the channel number by 1.
`+volumeUp(): void`	Increases the volume level by 1.
`+volumeDown(): void`	Decreases the volume level by 1.

The constructor and methods in the **TV** class are defined public so they can be accessed from other classes.

Example: Defining Classes and Creating Objects

```java
public class TV {
  int channel = 1; // Default cl
  int volumeLevel = 1; // Defau
  boolean on = false; // TV is

  public TV() {
  }

  public void turnOn() {
    on = true;
  }

  public void turnOff() {
```

```java
      on = false;
    }

    public void setChannel(int newChannel) {
      if (on && newChannel >= 1 && newChannel <= 120)
        channel = newChannel;
    }

    public void setVolume(int newVolumeLevel) {
      if (on && newVolumeLevel >= 1 && newVolumeLevel <= 7)
        volumeLevel = newVolumeLevel;
    }

    public void channelUp() {
      if (on && channel < 120)
        channel++;
    }

    public void channelDown() {
      if (on && channel > 1)
        channel--;
    }

    public void volumeUp() {
      if (on && volumeLevel < 7)
        volumeLevel++;
    }

    public void volumeDown() {
      if (on && volumeLevel > 1)
        volumeLevel--;
    }
  }
```

Example: Defining Classes and Creating Objects

```java
public class TestTV {
  public static void main(String[] args) {
    TV tv1 = new TV();
    tv1.turnOn();
    tv1.setChannel(30);
    tv1.setVolume(3);

    TV tv2 = new TV();
    tv2.turnOn();
    tv2.channelUp();
    tv2.channelUp();
    tv2.volumeUp();

    System.out.println("tv1's channel is " + tv1.channel
      + " and volume level is " + tv1.volumeLevel);
    System.out.println("tv2's channel is " + tv2.channel
      + " and volume level is " + tv2.volumeLevel);
  }
}
```

Constructors

```
Circle() {
}

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

Constructors are a special kind of methods that are invoked to construct objects.

Constructors, cont.

A constructor with no parameters is referred to as a *no-arg constructor*.

- 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 Using Constructors

```
new ClassName();
```

Example:

```
new Circle();
```

```
new Circle(5.0);
```

Default Constructor

A class may be defined without constructors. In this case, a no-arg constructor with an empty body is implicitly defined in the class. This constructor, called *a default constructor*, is provided automatically *only if no constructors are explicitly defined in the class*.

Declaring Object Reference Variables

- Objects are accessed via the object’s *reference variables*, which contain references to the objects.
- To reference an object, assign the object to a reference variable.
- A class is a *reference type*, which means that a variable of the class type can reference an instance of the class.
- To declare a reference variable, use the syntax:

```
    ClassName objectRefVar;
```

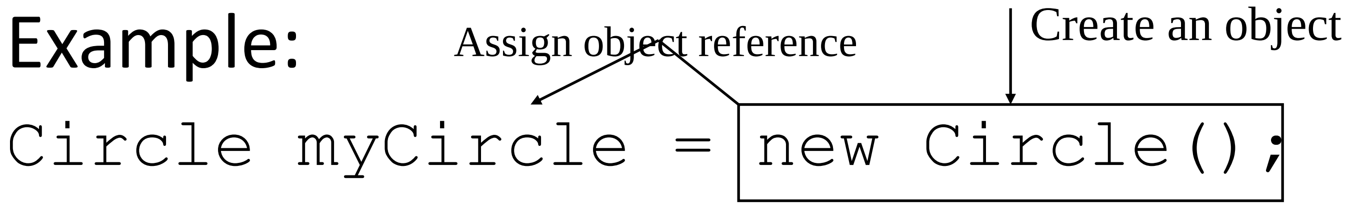
Example:

```
    Circle myCircle;
```

Declaring/Creating Objects in a Single Step

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

Example:



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

Accessing Object’s Members

- Referencing the object’s data is done using the *dot operator* or *object member access operator*:

```
objectRefVar.data
```

e.g., `myCircle.radius`

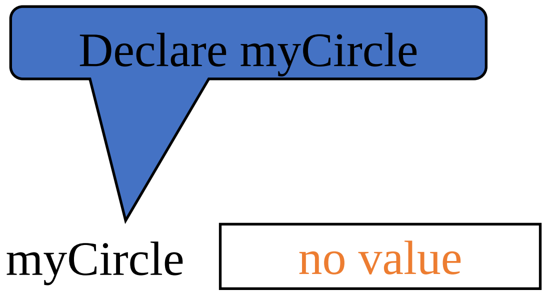
- Invoking the object’s method:

```
objectRefVar.methodName(arguments)
```

e.g., `myCircle.getArea()`

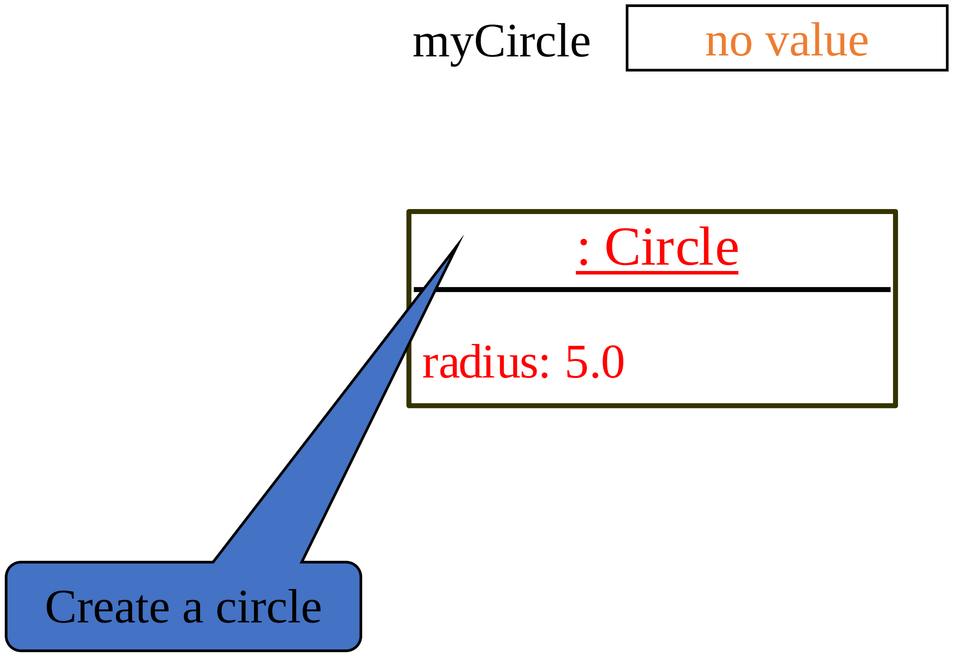
Trace Code

```
Circle myCircle = new Circle(5.0);

Circle yourCircle = new Circle();

yourCircle.radius = 100;
```



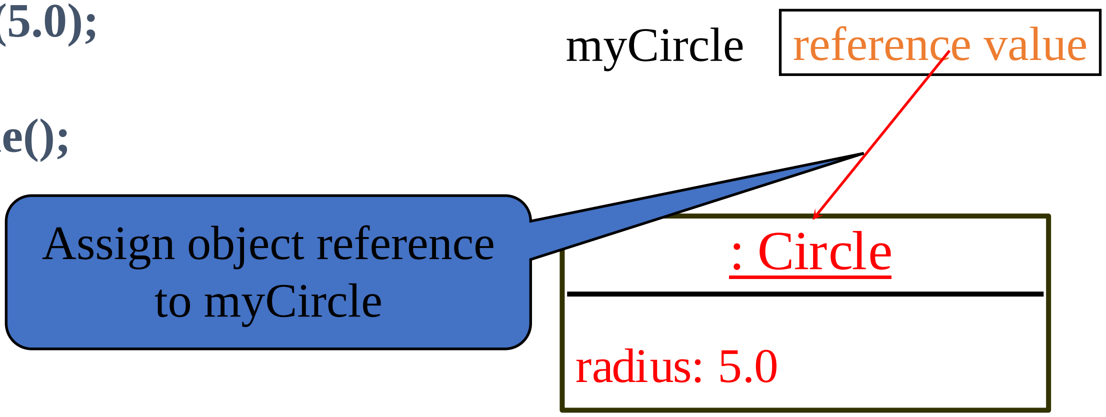
Trace Code, cont.

```
Circle myCircle = new Circle(5.0);

Circle yourCircle = new Circle();

yourCircle.radius = 100;
```



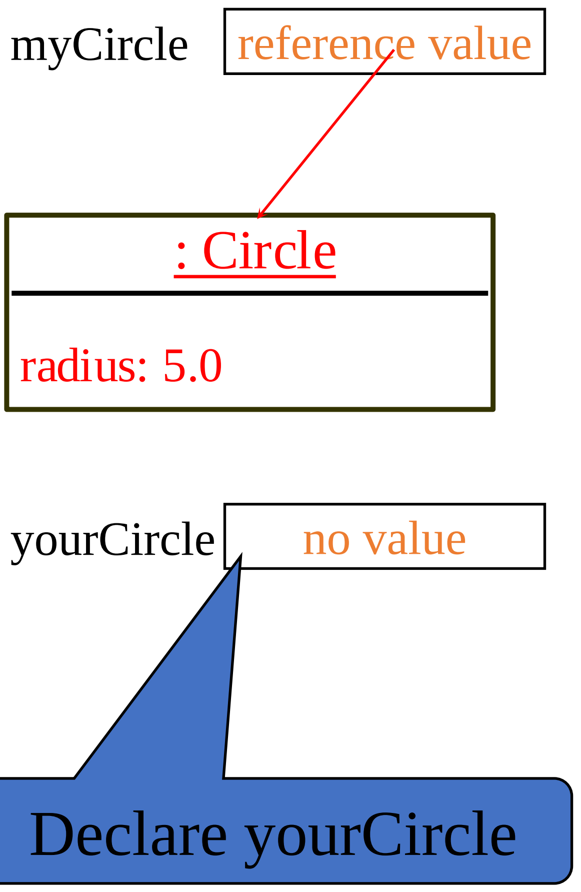
Trace Code, cont.

```
Circle myCircle = new Circle(5.0);

Circle yourCircle = new Circle();

yourCircle.radius = 100;
```



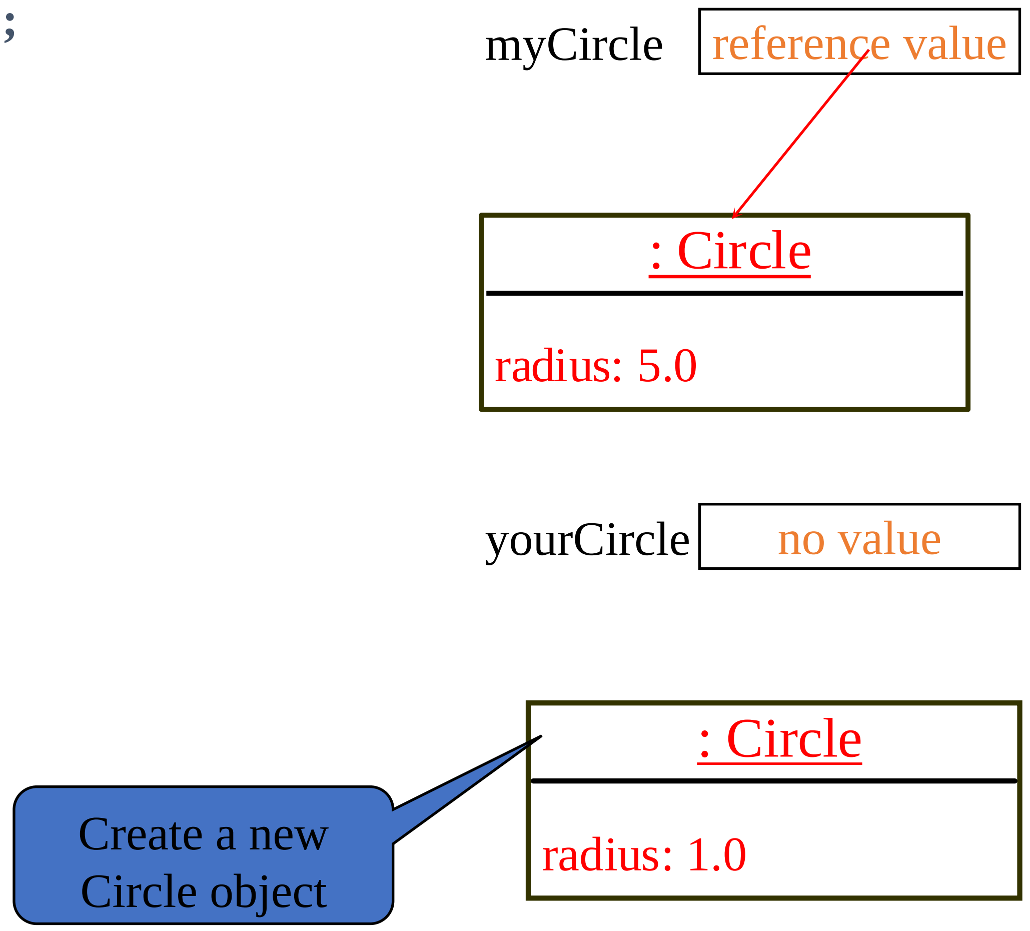
Trace Code, cont.

```
Circle myCircle = new Circle(5.0);

Circle yourCircle = new Circle();

yourCircle.radius = 100;
```



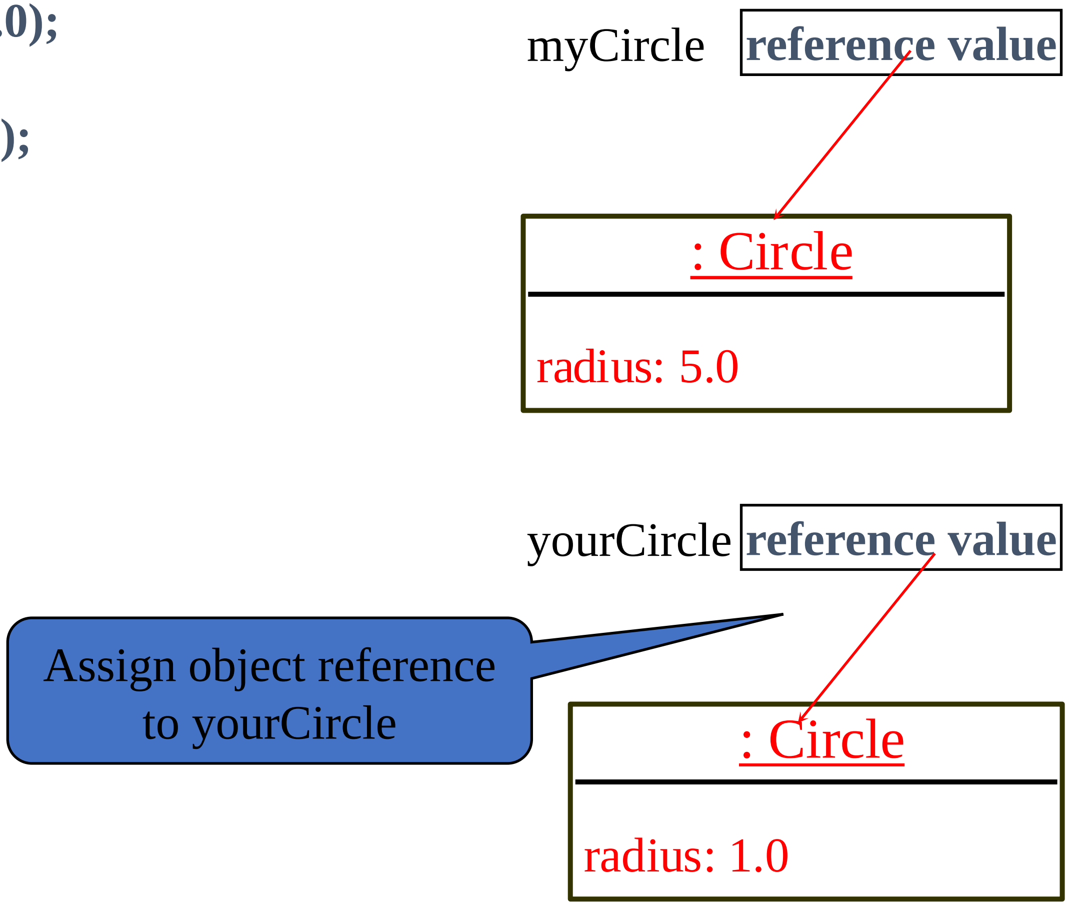
Trace Code, cont.

```
Circle myCircle = new Circle(5.0);

Circle yourCircle = new Circle();

yourCircle.radius = 100;
```



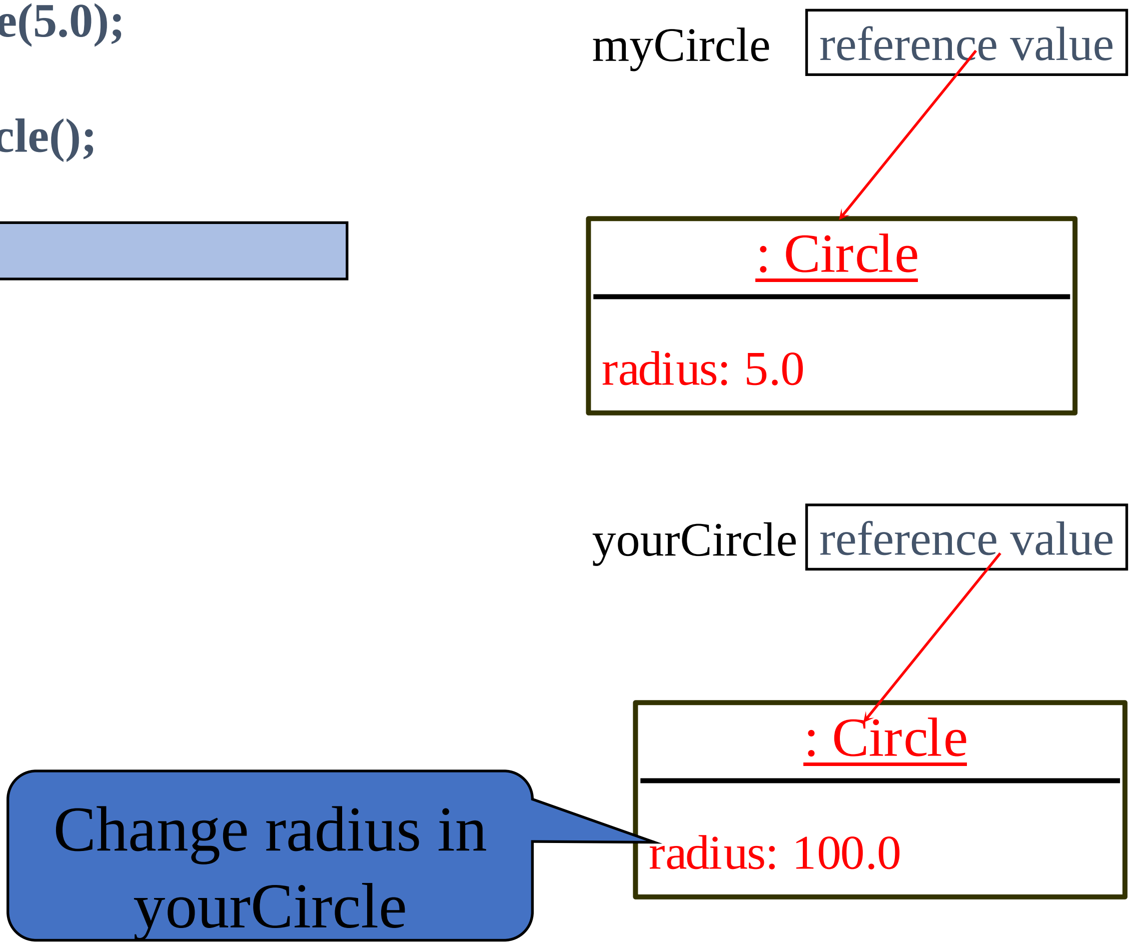
Trace Code, cont.

```
Circle myCircle = new Circle(5.0);

Circle yourCircle = new Circle();

yourCircle.radius = 100;
```



Trace Code, cont.

```
Circle myCircle = new Circle(5.0);

Circle yourCircle = new Circle();

yourCircle.radius = 100;
```



Caution

- The data field **radius** is referred to as an *instance variable*, because it is dependent on a specific instance. For the same reason, the method **getArea** is referred to as an *instance method*, because you can invoke it only on a specific instance.
- The object on which an instance method is invoked is called a *calling object*.

The null Value

If a data field of a reference type does not reference any object, the data field holds a special literal value, null.

Default Value for a Data Field

The default value of a data field is null for a reference type, 0 for a numeric type, false for a boolean type, and '\u0000' for a char type. However, Java assigns no default value to a local variable inside a method.

```
public class Test {
  public static void main(String[] args) {
    Student student = new Student();
    System.out.println("name? " + student.name);
    System.out.println("age? " + student.age);
    System.out.println("isScienceMajor? " + student.isScienceMajor);
    System.out.println("gender? " + student.gender);
  }
}
```

Example

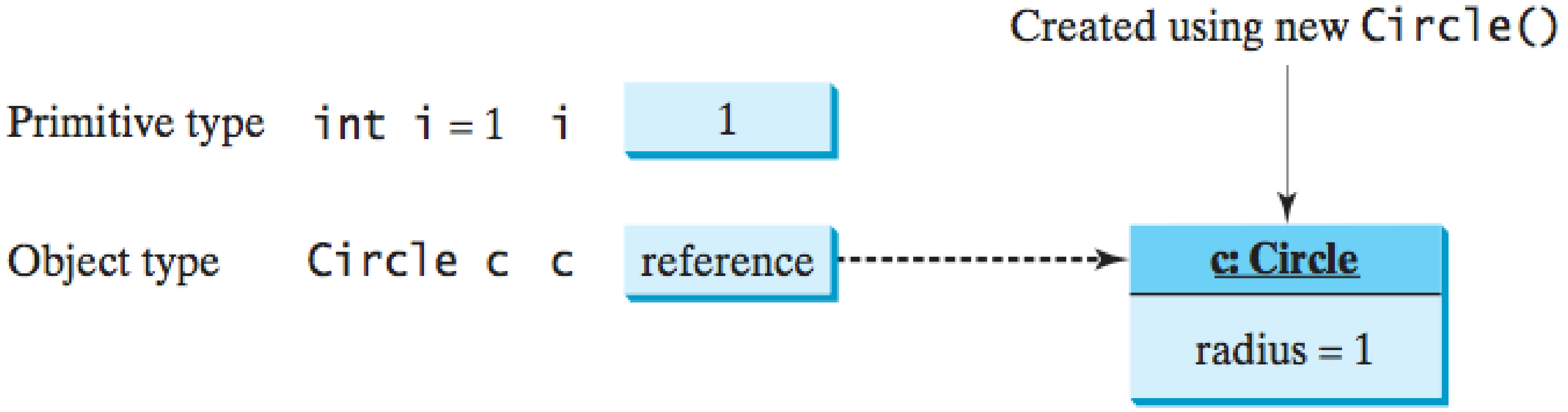
Java assigns no default value to a local variable inside a method.

```
public class Test {
  public static void main(String[] args) {
    int x; // x has no default value
    String y; // y has no default value
    System.out.println("x is " + x);
    System.out.println("y is " + y);
  }
}
```

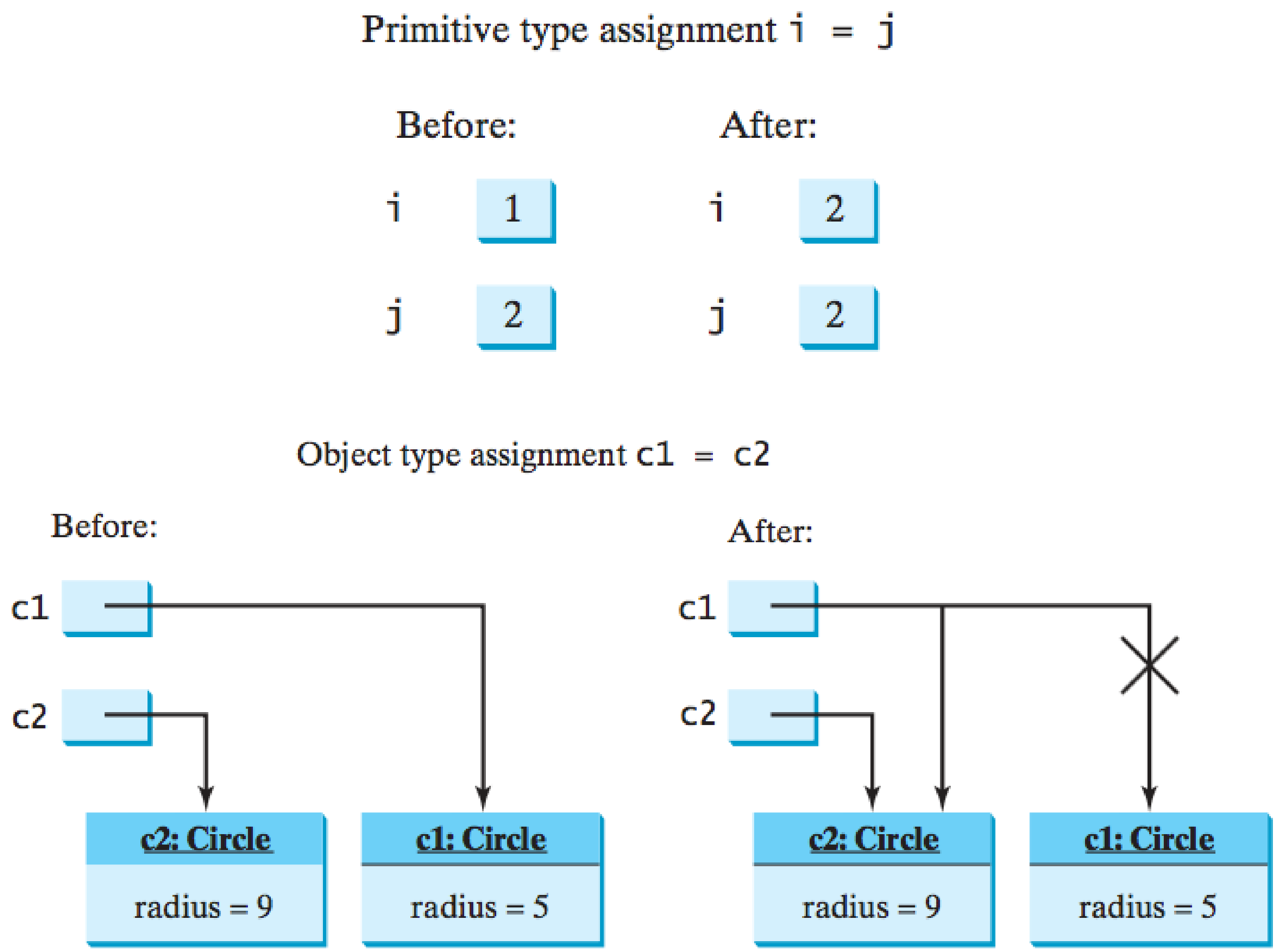
Compile error: variable not initialized

Differences between Variables of Primitive Data Types and Object 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.



Copying Variables of Primitive Data Types and Object Types



Garbage Collection

As shown in the previous figure, after the assignment statement c1 = c2, c1 points to the same object referenced by c2. The object previously referenced by c1 is no longer referenced. This object is known as garbage. Garbage is automatically collected by JVM. The JVM will automatically collect the space if the object is not referenced by any variable .

Instance Variables, and Methods

- Instance variables belong to a specific instance.
- Instance methods are invoked by any instance of the class.

Static Variables, Constants, and Methods

- Static variables are shared by all the instances of the class.
- Static methods are not tied to a specific object. Because of this, a static method cannot access instance members of the class
- Static constants are final variables shared by all the instances of the class.
- A non-static (or instance) variable is tied to a specific instance

Static Variables, Constants, and Methods

- Static variables store values for the variables in a common memory location. Because of this common location, if one object changes the value of a static variable, all objects of the same class are affected.
- Java supports static methods as well as static variables. *Static methods* can be called without creating an instance of the class.

Static Variables, Constants, and Methods, cont.

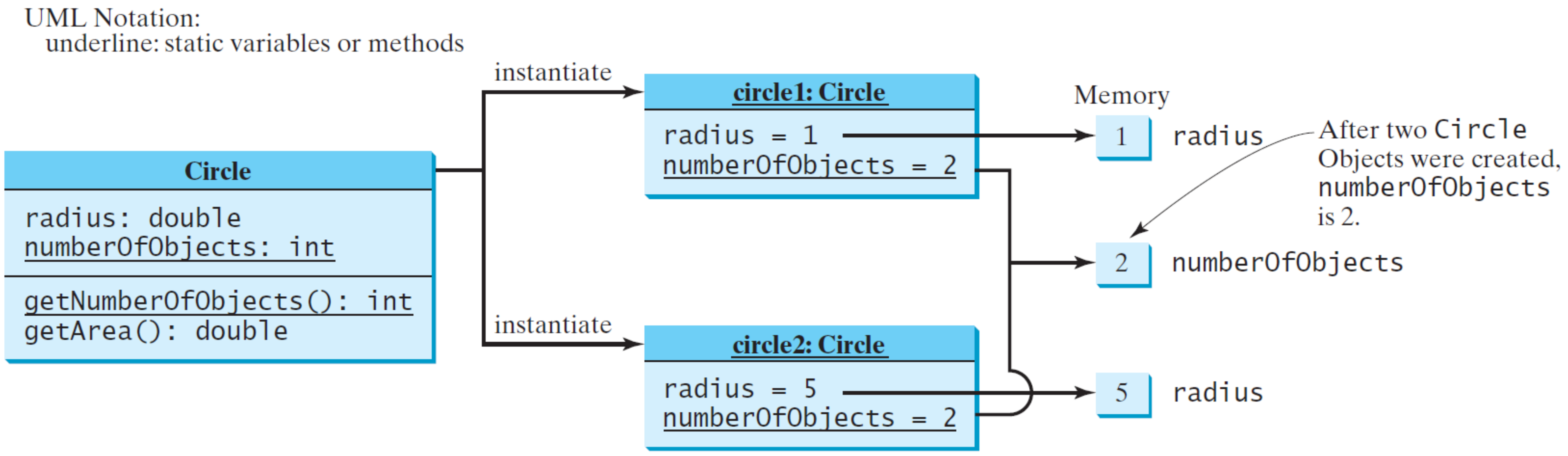
- To declare static variables, constants, and methods, use the static modifier. For example, the constant **PI** in the **Math** class is defined as

  final static double PI=**3.14159265358979323846**

- Let’s modify the **Circle** class by adding a static variable **numberOfObjects** to count the number of circle objects created. When the first object of this class is created, **numberOfObjects** is **1**. When the second object is created, **numberOfObjects** becomes **2**. The UML of the new circle class is shown below

Static Variables, Constants, and Methods, cont.

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



CircleWithStaticMembers Class

```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;
  }
}
```

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 + ")");
  }
}
```

Static variables and methods can be accessed without creating objects. Line 6 displays the number of objects, which is **0**, since no objects have been created.

Static and Instance Methods

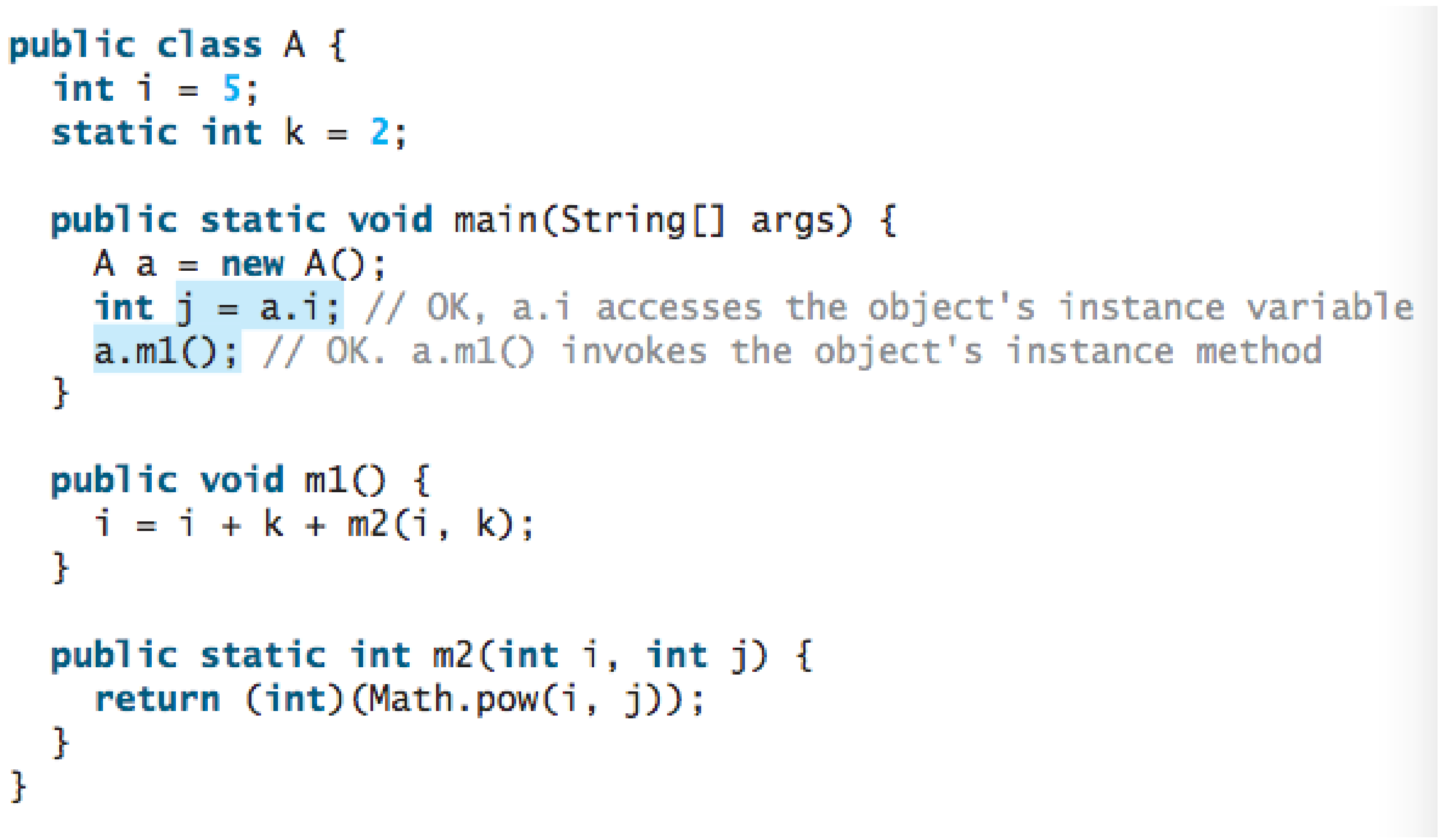
Instance method adalah metode yang dihubungkan dengan sebuah objek dan dapat mengakses baik instance data field (data field yang unik untuk setiap objek) maupun static data field (data field yang terkait dengan kelas itu sendiri). Selain itu, instance method juga dapat memanggil baik instance method maupun static method.

static method adalah metode yang terkait dengan kelas itu sendiri dan tidak terkait dengan objek spesifik apa pun. Static method hanya dapat mengakses static data field dan memanggil static method lainnya. Namun, static method tidak dapat mengakses instance data field atau memanggil instance method, karena instance data field dan instance method terkait dengan objek yang spesifik dan tidak dapat diakses tanpa objek yang sesuai.

Examples

```
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));
  }
}
```

Examples



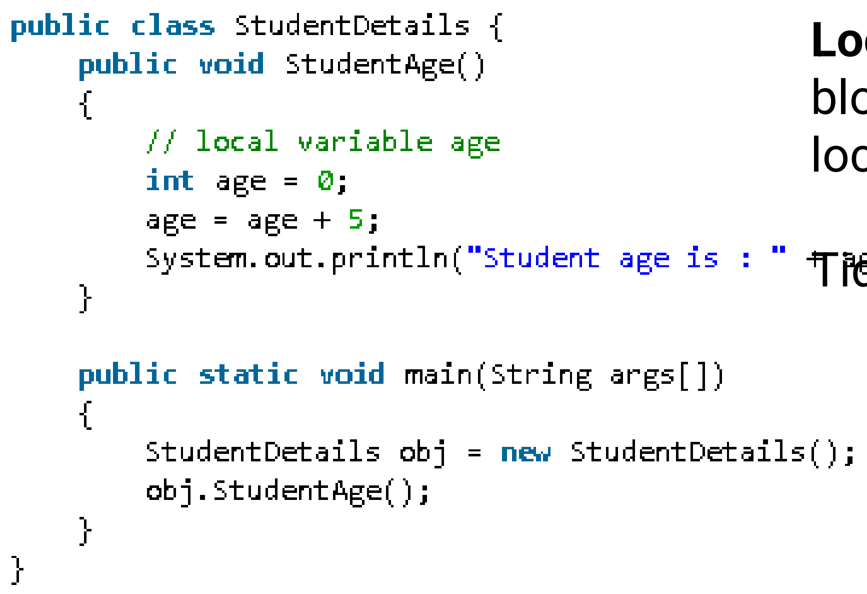
```java
public class A {
  int i = 5;
  static int k = 2;

  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));
  }
}
```

Review Scope Variable

Variable Scope

- Variable access
  - Lokal : in method, hanya bisa diakses di method yang mendeklarasikan variable tersebut
  - Instance: in class, access with object in main
  - Static: in class, access anywhere

Local



```java
public class StudentDetails {
    public void StudentAge()
    {
        // local variable age
        int age = 0;
        age = age + 5;
        System.out.println("Student age is : " + age);
    }

    public static void main(String args[])
    {
        StudentDetails obj = new StudentDetails();
        obj.StudentAge();
    }
}
```

Local Variables: A variable defined within a block or method or constructor is called local variable

Tidak bisa diakses dari main

Output:

Student age is : 5

Local

```
public class StudentDetails {
    public void StudentAge()
    { // local variable age
        int age = 0;
        age = age + 5;
    }

    public static void main(String args[])
    {
        // using local variable age outside it's scope
        System.out.println("Student age is : " + age);
    }
}
```

```
Compilation Error in java code :-
prog.java:12: error: cannot find symbol
        System.out.println("Student age is : " + age);
                                                 ^
  symbol:   variable age
  location: class StudentDetails
1 error
```

Instance Variables

- **Instance Variables**: Instance variables are non-static variables and are declared in a class outside any method, constructor or block.
- Bisa diakses dari main melalui objek

```java
class Marks {
    // These variables are instance variables.
    // These variables are in a class
    // and are not inside any function
    int engMarks;
    int mathsMarks;
    int phyMarks;
}

class MarksDemo {
    public static void main(String args[])
    {
        // first object
        Marks obj1 = new Marks();
        obj1.engMarks = 50;
        obj1.mathsMarks = 80;
        obj1.phyMarks = 90;

        // second object
        Marks obj2 = new Marks();
        obj2.engMarks = 80;
        obj2.mathsMarks = 60;
        obj2.phyMarks = 85;

        // displaying marks for first object
        System.out.println("Marks for first object:");
        System.out.println(obj1.engMarks);
        System.out.println(obj1.mathsMarks);
        System.out.println(obj1.phyMarks);

        // displaying marks for second object
        System.out.println("Marks for second object:");
        System.out.println(obj2.engMarks);
        System.out.println(obj2.mathsMarks);
        System.out.println(obj2.phyMarks);
    }
}
```

static variables

```
class Emp {

    // static variable salary
    public static double salary;
    public static String name = "Harsh";
}

public class EmpDemo {
    public static void main(String args[])
    {

        // accessing static variable without object
        Emp.salary = 1000;
        System.out.println(Emp.name + "'s average salary:"
                                + Emp.salary);
    }
}
```

To access static variables in main,

No need to create an object of that class,

can simply access the variable as:

class_name.variable_name;

```
Harsh's average salary:1000.0
```

`myMethod()` prints a text (the action), when it is **called**. To call a method, write the method's name followed by two parentheses **()** and a semicolon**;**

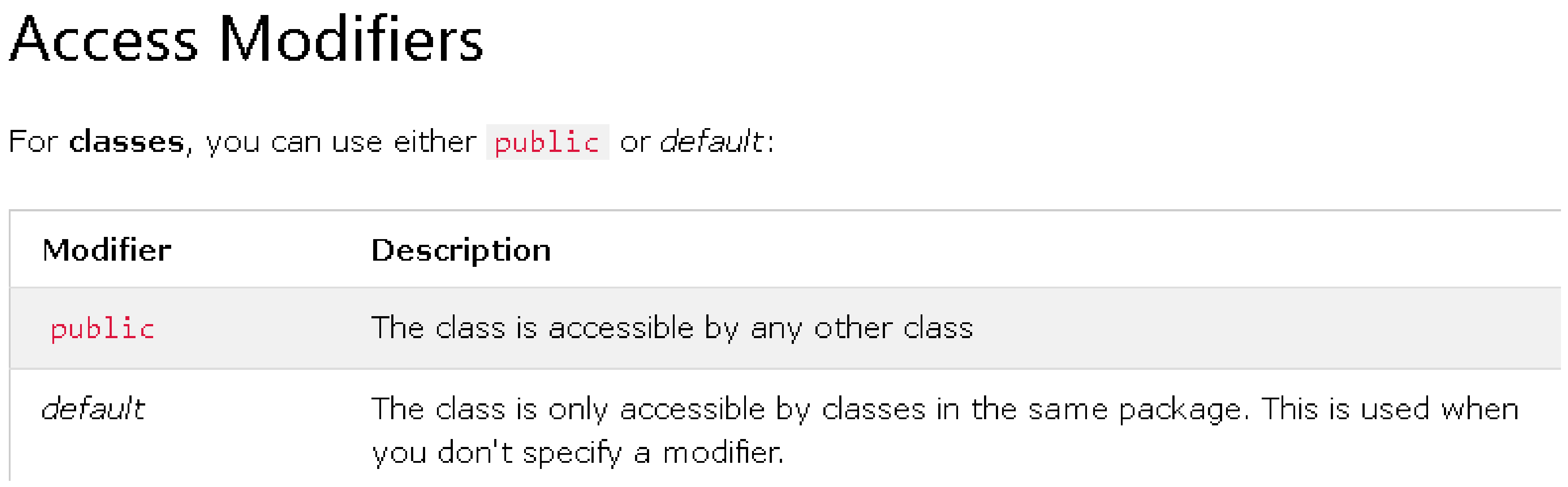
Example

Inside `main`, call `myMethod()`:

```
public class MyClass {
  static void myMethod() {
    System.out.println("Hello World!");
  }

  public static void main(String[] args) {
    myMethod();
  }
}

// Outputs "Hello World!"
```

ACCESS MODIFIER

Access Modifier For Class

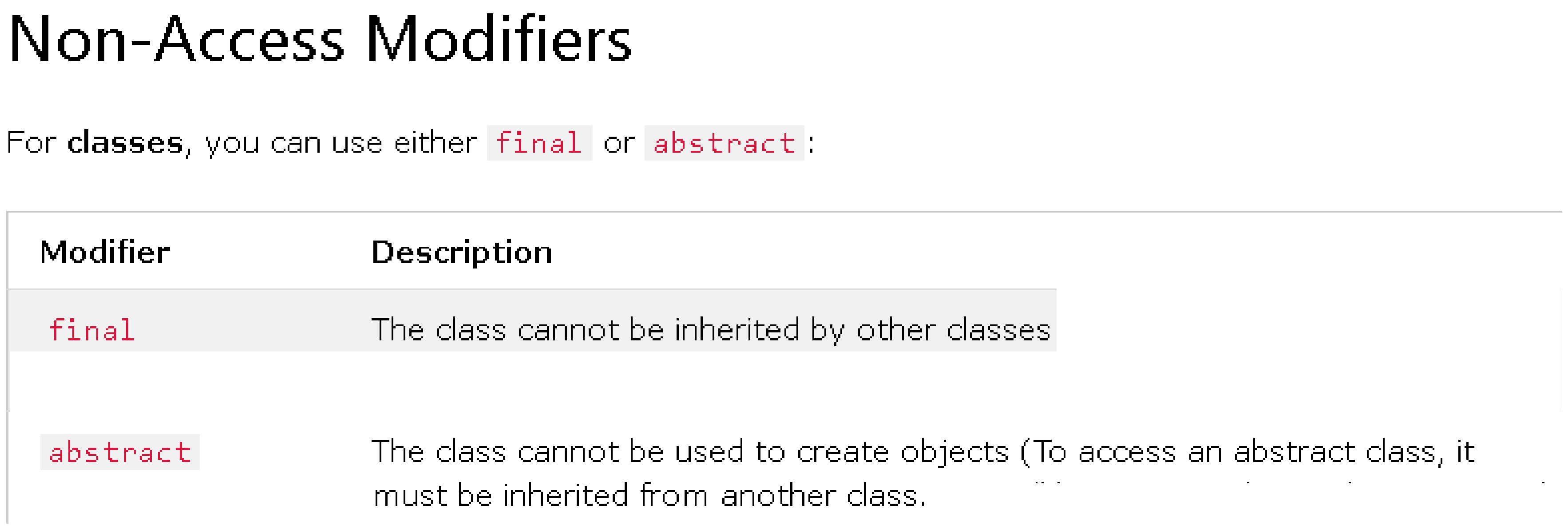


Access Modifiers

For **classes**, you can use either `public` or *default*:

Modifier	Description
`public`	The class is accessible by any other class
default	The class is only accessible by classes in the same package. This is used when you don't specify a modifier.

Non- Access Modifier For Class

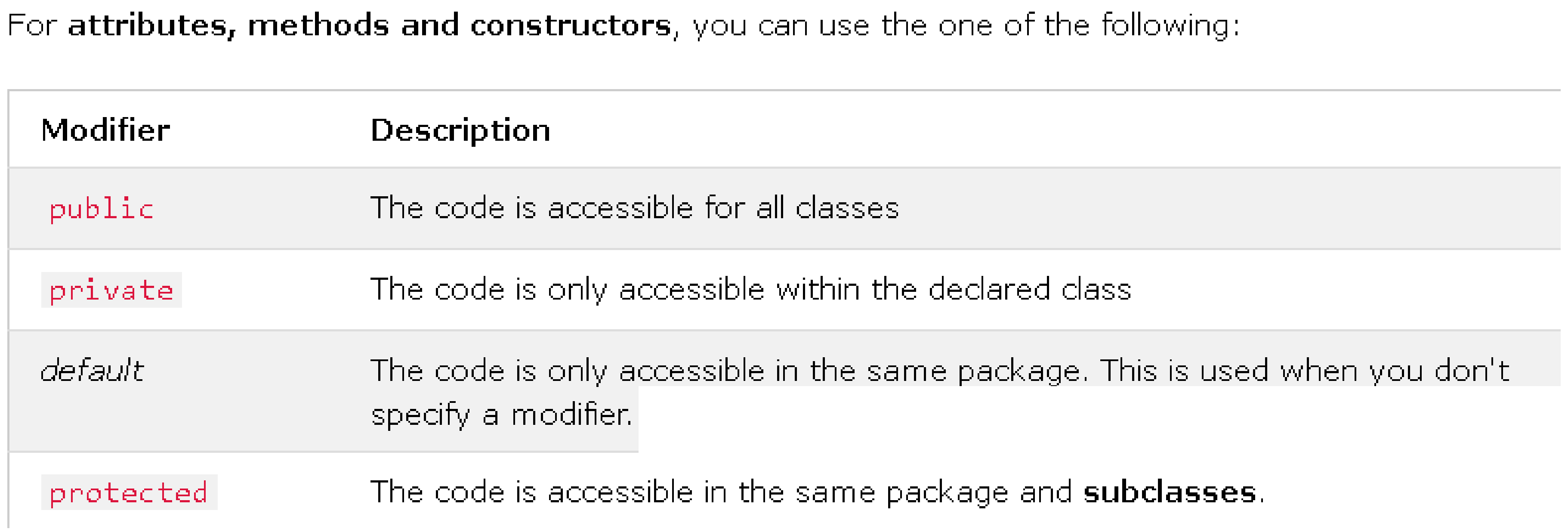


Non-Access Modifiers

For **classes**, you can use either `final` or `abstract`:

Modifier	Description
`final`	The class cannot be inherited by other classes
`abstract`	The class cannot be used to create objects (To access an abstract class, it must be inherited from another class.

Access Modifier Attr, Method and Constructor



For **attributes, methods and constructors**, you can use the one of the following:

Modifier	Description
`public`	The code is accessible for all classes
`private`	The code is only accessible within the declared class
default	The code is only accessible in the same package. This is used when you don't specify a modifier.
`protected`	The code is accessible in the same package and **subclasses**.

Package

- A package in Java is used to group related classes. Think of it as **a folder in a file directory**.
- We use packages to avoid name conflicts, and to write a better maintainable code. Packages are divided into two categories:
  - Built-in Packages (packages from the Java API)
  - User-defined Packages (create your own packages)

Package

The library is divided into **packages** and **classes**. Meaning you can either import a single class (along with its methods and attributes), or a whole package that contain all the classes that belong to the specified package.

Syntax

```
import packagename.Class;   // Import a single class
import packagename.*;   // Import the whole package
import java.util.Scanner;
import java.util.*;
```

Here:
→ **java** is a top level package
→ **util** is a sub package
→ and **Scanner** is a class which is present in the sub package **util**.

To use the `Scanner` class, create an object of the class and use any of the available methods found in the `Scanner` class documentation. In our example, we will use the `nextLine()` method, which is used to read Strings:

Example

```java
import java.util.Scanner;  // Import the Scanner class

class MyClass {
  public static void main(String[] args) {
    Scanner myObj = new Scanner(System.in);  // Create a Scanner object
    System.out.println("Enter username");

    String userName = myObj.nextLine();  // Read user input
    System.out.println("Username is: " + userName);  // Output user input
  }
}
```

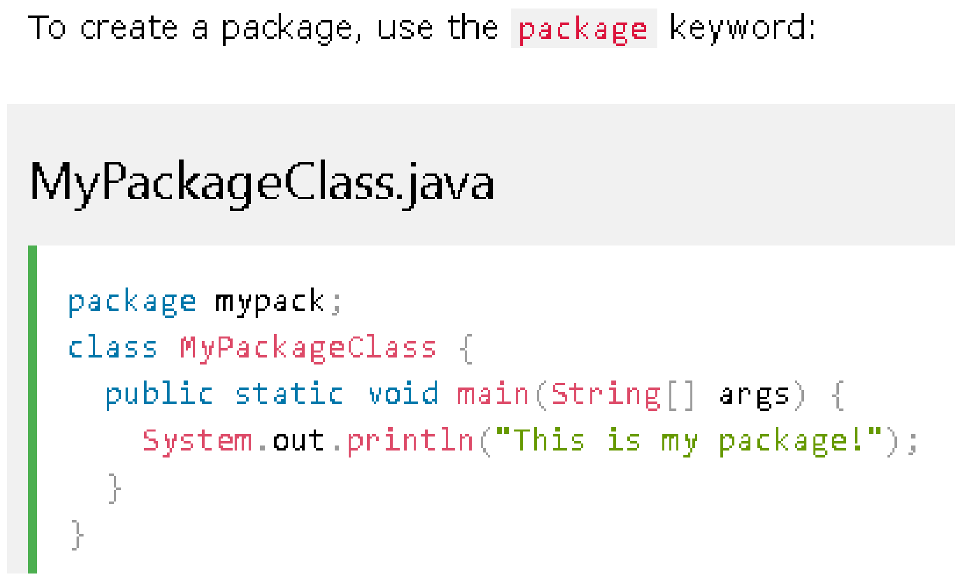
Input Types

In the example above, we used the `nextLine()` method, which is used to read Strings. To read other types, look at the table below:

Method	Description
`nextBoolean()`	Reads a `boolean` value from the user
`nextByte()`	Reads a `byte` value from the user
`nextDouble()`	Reads a `double` value from the user
`nextFloat()`	Reads a `float` value from the user
`nextInt()`	Reads a `int` value from the user
`nextLine()`	Reads a `String` value from the user
`nextLong()`	Reads a `long` value from the user
`nextShort()`	Reads a `short` value from the user

User-defined Packages

- To create your own package, you need to understand that Java uses a file system directory to store them. Just like folders on your computer
  - └── root
  - └── mypack
  - └── MyPackageClass.java



To create a package, use the `package` keyword:

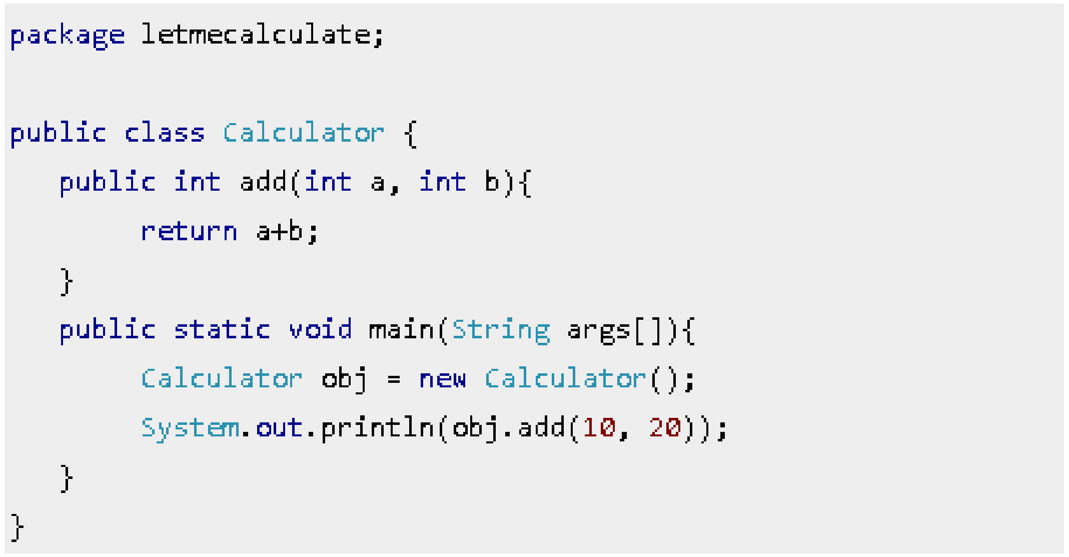
MyPackageClass.java

```java
package mypack;
class MyPackageClass {
  public static void main(String[] args) {
    System.out.println("This is my package!");
  }
}
```

Contoh

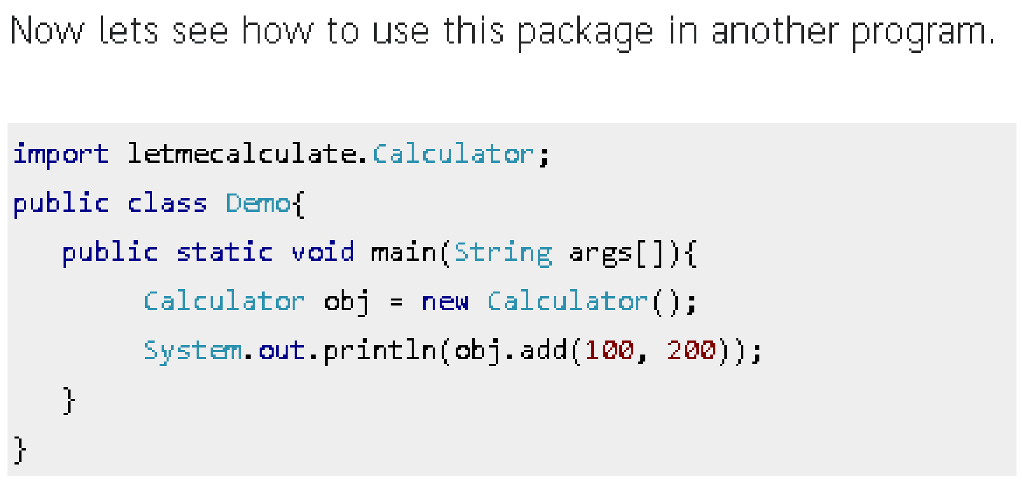
- created a class Calculator inside a package name letmecalculate.
- To create a class inside a package, declare the package name in the first statement in your program.
- A class can have only one package declaration.

Calculator.java file created inside a package letmecalculate



```
package letmecalculate;

public class Calculator {
   public int add(int a, int b){
        return a+b;
   }
   public static void main(String args[]){
        Calculator obj = new Calculator();
        System.out.println(obj.add(10, 20));
   }
}
```

Contoh



Now lets see how to use this package in another program.

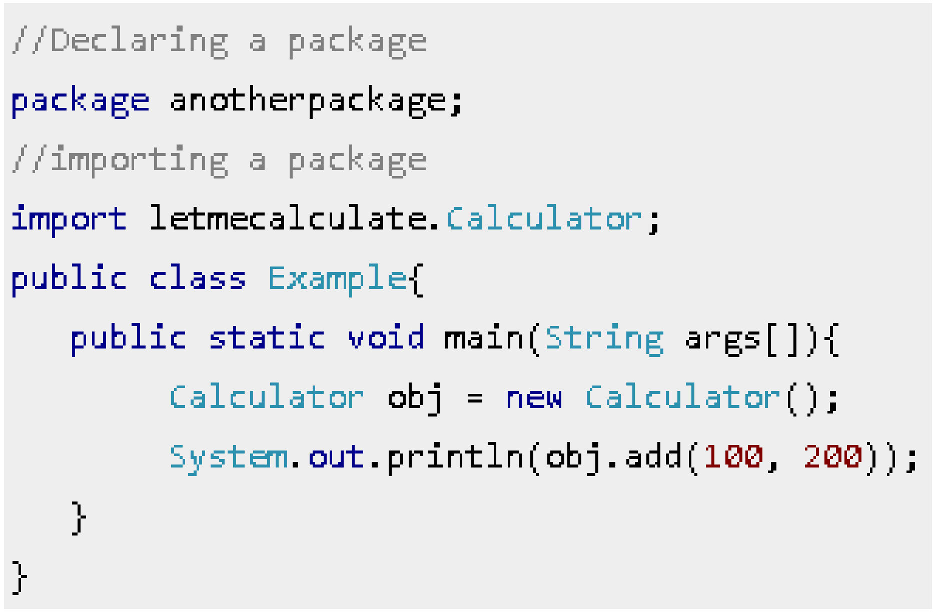
```
import letmecalculate.Calculator;
public class Demo{
   public static void main(String args[]){
        Calculator obj = new Calculator();
        System.out.println(obj.add(100, 200));
   }
}
```

To use the class Calculator, I have imported the package letmecalculate. In the above program I have imported the package as letmecalculate.Calculator, this only imports the Calculator class. However if you have several classes inside package letmecalculate then you can import the package like this, to use all the classes of this package.

import letmecalculate.*;

Creating a class inside package while importing another package

As we have seen that both package declaration and package import should be the first statement in your java program.



```
//Declaring a package
package anotherpackage;
//importing a package
import letmecalculate.Calculator;
public class Example{
   public static void main(String args[]){
      Calculator obj = new Calculator();
      System.out.println(obj.add(100, 200));
   }
}
```

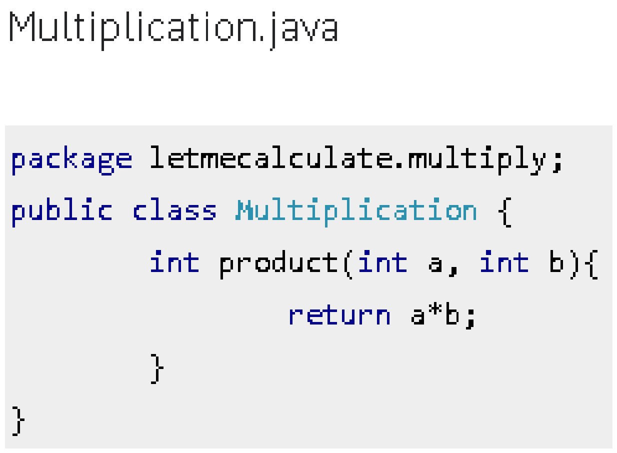
So the order in this case should be:

→ package declaration

→ package import

Sub packages in Java

- A package inside another package is known as sub package. For example If we create a package inside letmecalculate package then that will be called sub package. Lets say we have created another package inside letmecalculate and the sub package name is multiply



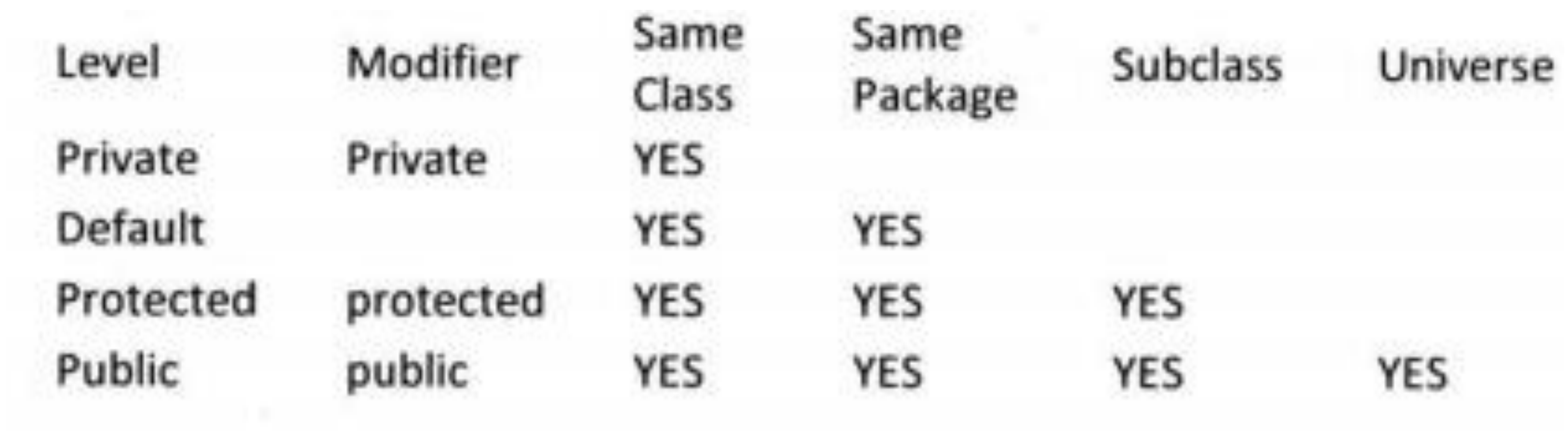
Multiplication.java

```
package letmecalculate.multiply;
public class Multiplication {
        int product(int a, int b){
                return a*b;
        }
}
```

if We need to use this Multiplication class we need to import the package like this:

import letmecalculate.multiply;

Perbandingan access modifier



Level	Modifier	Same Class	Same Package	Subclass	Universe
Private	Private	YES			
Default		YES	YES		
Protected	protected	YES	YES	YES	
Public	public	YES	YES	YES	YES

Public access modifier

- The members, methods and classes that are declared public can be accessed from anywhere. This modifier doesn’t put any restriction on the access.

public access modifier example in java

Lets take the same example that we have seen above but this time the method addTwoNumbers() has public modifier and class Test is able to access this method without even extending the Addition class. This is because public modifier has visibility everywhere.

Addition.java

```
package abcpackage;

public class Addition {

   public int addTwoNumbers(int a, int b){
        return a+b;
   }
}
```

Test.java

```
package xyzpackage;
import abcpackage.*;
class Test{
   public static void main(String args[]){
      Addition obj = new Addition();
      System.out.println(obj.addTwoNumbers(100, 1));
   }
}
```

Output:

```
101
```

Private access modifier

- The scope of private modifier is limited to the class only.
- Private Data members and methods are only accessible within the class
- Class and **Interface** cannot be declared as private
- If a class has **private constructor** then you cannot create the object of that class from outside of the class.

Private access modifier example in java

This example throws compilation error because we are trying to access the private data member and method of class ABC in the class Example. The private data member and method are only accessible within the class.

```
class ABC{
   private double num = 100;
   private int square(int a){
        return a*a;
   }
}
public class Example{
   public static void main(String args[]){
        ABC obj = new ABC();
        System.out.println(obj.num);
        System.out.println(obj.square(10));
   }
}
```

Output:

```
Compile - time error
```

Visibility Modifiers and Accessor/Mutator Methods

- You can use the **public** visibility modifier for classes, methods, and data fields to denote that they can be accessed from any other classes.
- 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*.
- Packages can be used to organize classes. To do so, you need to add the following line 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 and Accessor/Mutator Methods

By default, the class, variable, or method can be accessed by any class in the same package.

- `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 declaring class.

Public getter (Accessor) and setter (Mutator) methods are used to read and modify private properties.

Examples

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

Visibility Modifier of Classes

```
package p1;

class C1 {
  ...
}
```

```
package p1;

public class C2 {
  can access C1
}
```

```
package p2;

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

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

NOTE

- 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).

```java
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.

```java
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.

NOTE

- 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.

- In most cases, the constructor should be public. However, if you want to prohibit the user from creating an instance of a class, use a *private constructor*. For example, there is no reason to create an instance from the **Math** class, because all of its data fields and methods are static. To prevent the user from creating objects from the **Math** class, the constructor in **java.lang.Math** is defined as private.

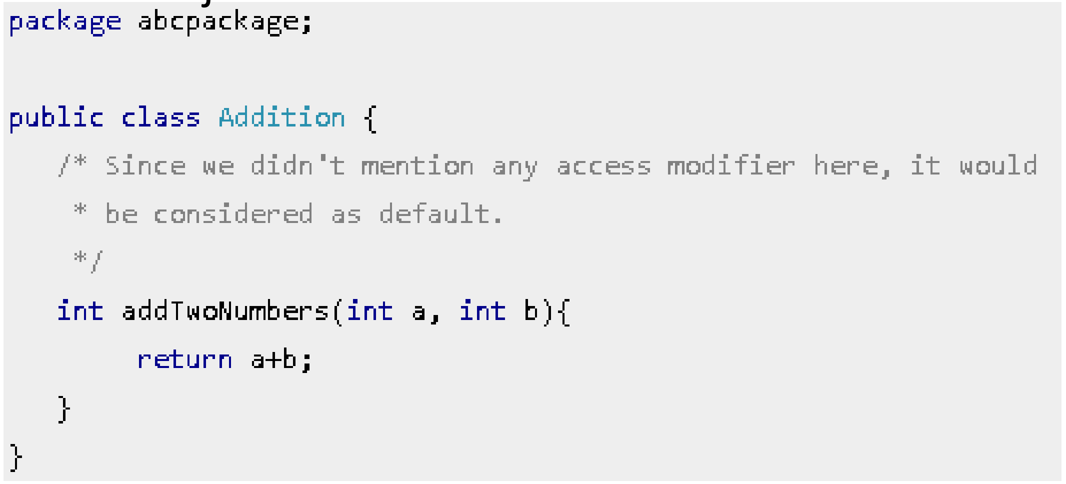
Default

- When we do not mention any access modifier, it is called default access modifier.
- The scope of this modifier is limited to the package only. This means that if we have a class with the default access modifier in a package, only those classes that are in this package can access this class. No other class outside this package can access this class.
- Similarly, if we have a default method or data member in a class, it would not be visible in the class of another package.

Default

In this example we have two classes, Test class is trying to access the default method of Addition class, since class Test belongs to a different package, this program would throw compilation error, because the scope of default modifier is limited to the same package in which it is declared.

Addition.java



```java
package abcpackage;

public class Addition {
   /* Since we didn't mention any access modifier here, it would
    * be considered as default.
    */
   int addTwoNumbers(int a, int b){
        return a+b;
   }
}
```

Default Cont.

Test.java

```java
package xyzpackage;

/* We are importing the abcpackage
 * but still we will get error because the
 * class we are trying to use has default access
 * modifier.
 */
import abcpackage.*;
public class Test {
   public static void main(String args[]){
        Addition obj = new Addition();
        /* It will throw error because we are trying to access
         * the default method in another package
         */
        obj.addTwoNumbers(10, 21);
   }
}
```

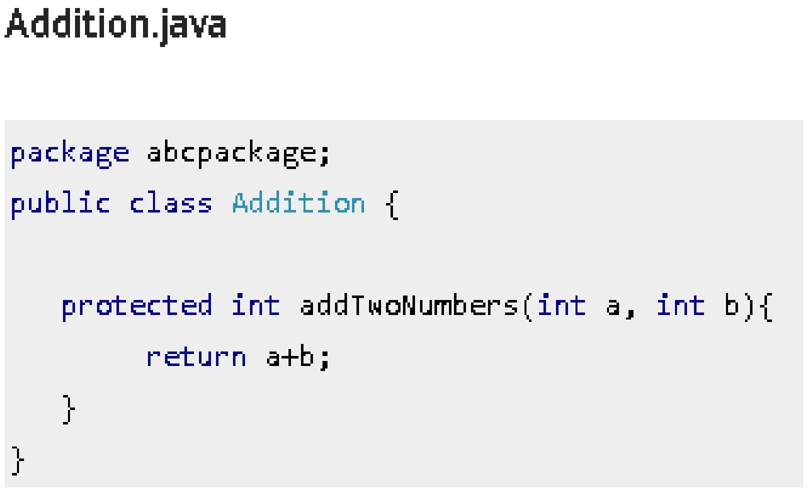
```
Exception in thread "main" java.lang.Error: Unresolved compilation problem:
The method addTwoNumbers(int, int) from the type Addition is not visible
at xyzpackage.Test.main(Test.java:12)
```

Protected Access Modifier

Protected data member and method are only accessible by the classes of the same package and the subclasses present in any package. You can also say that the protected access modifier is similar to default access modifier with one exception that it has visibility in sub classes. Classes cannot be declared protected. This access modifier is generally used in a parent child relationship.

Protected access modifier example in Java

- In this example the class Test which is present in another package is able to call the addTwoNumbers() method, which is declared protected. This is because the Test class extends class Addition and the protected modifier allows the access of protected members in subclasses (in any packages).



Addition.java

```java
package abcpackage;
public class Addition {

   protected int addTwoNumbers(int a, int b){
        return a+b;
   }
}
```

Protected access modifier example in Java

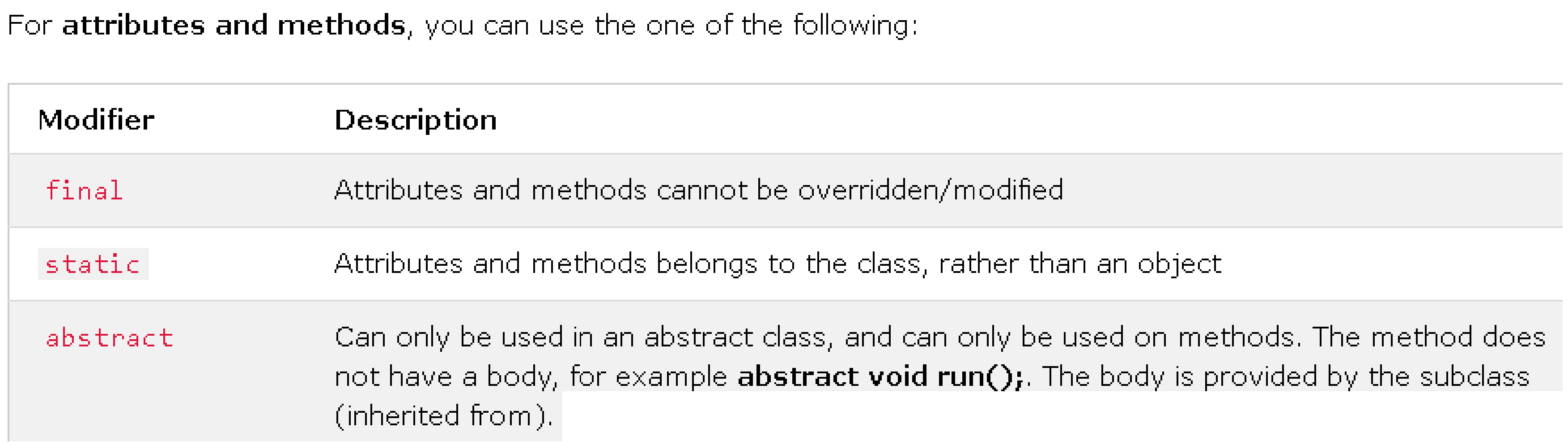
Test.java

```
package xyzpackage;
import abcpackage.*;
class Test extends Addition{
   public static void main(String args[]){
        Test obj = new Test();
        System.out.println(obj.addTwoNumbers(11, 22));
   }
}
```

Output:

```
33
```

Non-Access Modifier For Method



For **attributes and methods**, you can use the one of the following:

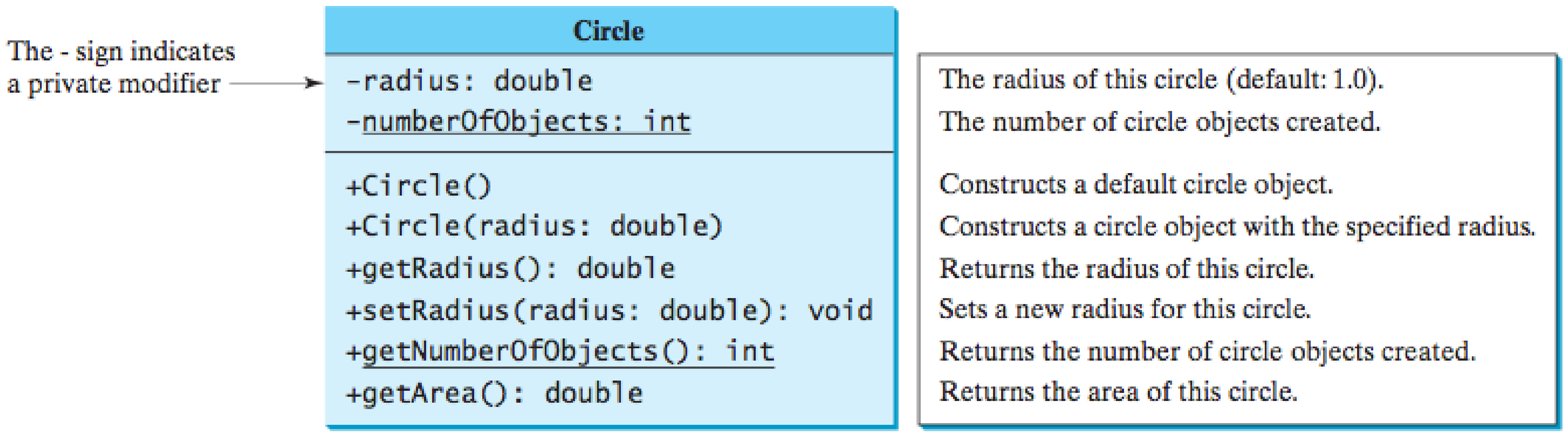
Modifier	Description
`final`	Attributes and methods cannot be overridden/modified
`static`	Attributes and methods belongs to the class, rather than an object
`abstract`	Can only be used in an abstract class, and can only be used on methods. The method does not have a body, for example **abstract void run();**. The body is provided by the subclass (inherited from).

Data Field Encapsulation

- *Making data fields private* helps to make code easy to maintain since the client programs cannot modify them.
- To prevent direct modifications of data fields, declaring the data fields private is known as *data field encapsulation*.
- To make a private data field accessible, provide a *getter* method to return its value.
- To enable a private data field to be updated, provide a *setter* method to set a new value.
- A getter method is also referred to as an *accessor* and a setter to a *mutator*.

Encapsulation

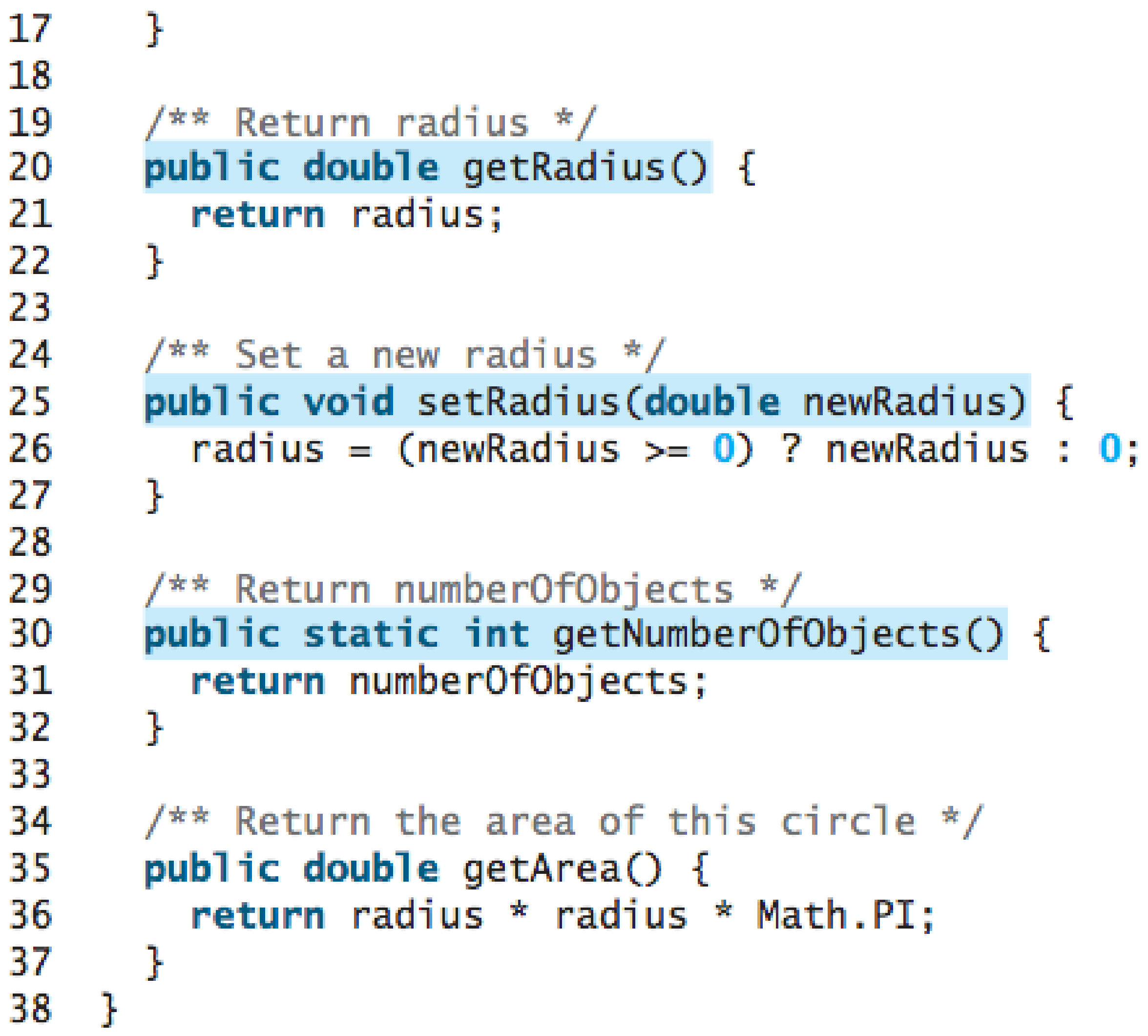
Example of Data Field Encapsulation


The - sign indicates a private modifier
Circle
-radius: double
-numberOfObjects: int
+Circle()
+Circle(radius: double)
+getRadius(): double
+setRadius(radius: double): void
+getNumberOfObjects(): int
+getArea(): double
The radius of this circle (default: 1.0).
The number of circle objects created.
Constructs a default circle object.
Constructs a circle object with the specified radius.
Returns the radius of this circle.
Sets a new radius for this circle.
Returns the number of circle objects created.
Returns the area of this circle.

Example of Data Field Encapsulation

```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++;
```

Example of Data Field Encapsulation



```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;
  }
}
```

Example of Data Field Encapsulation

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

Passing Primitive dan Reference

Passing Objects to Methods

- ❑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)

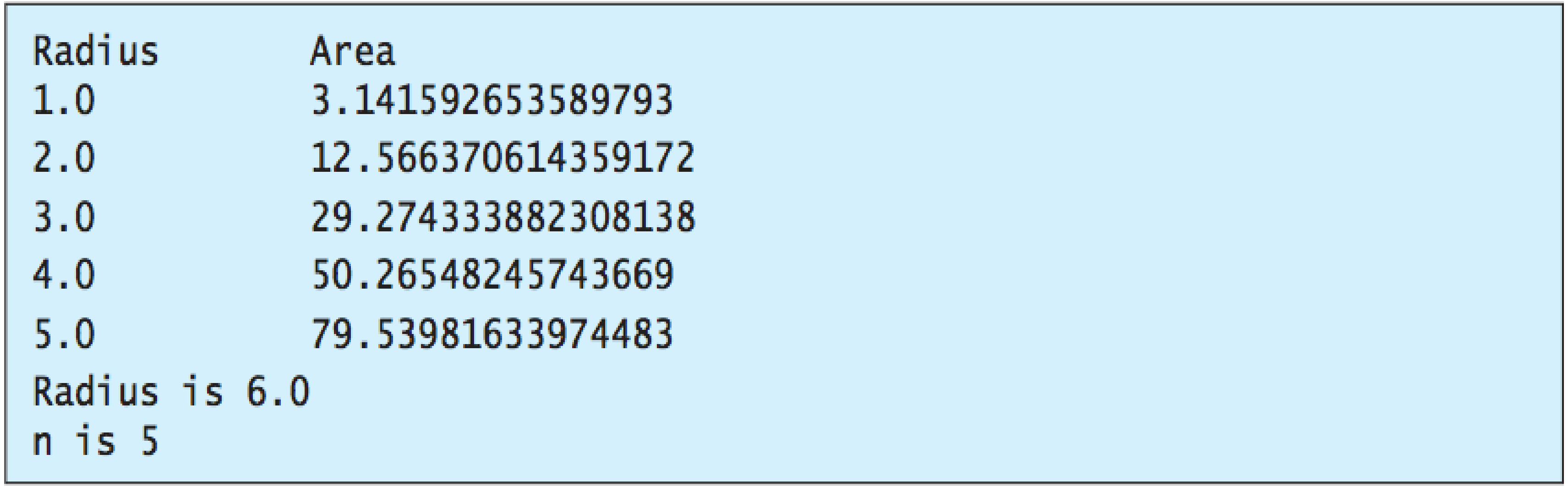
Passing Objects to Methods

```
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 a primitive type value and a reference value

```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--;
    }
  }
}
```

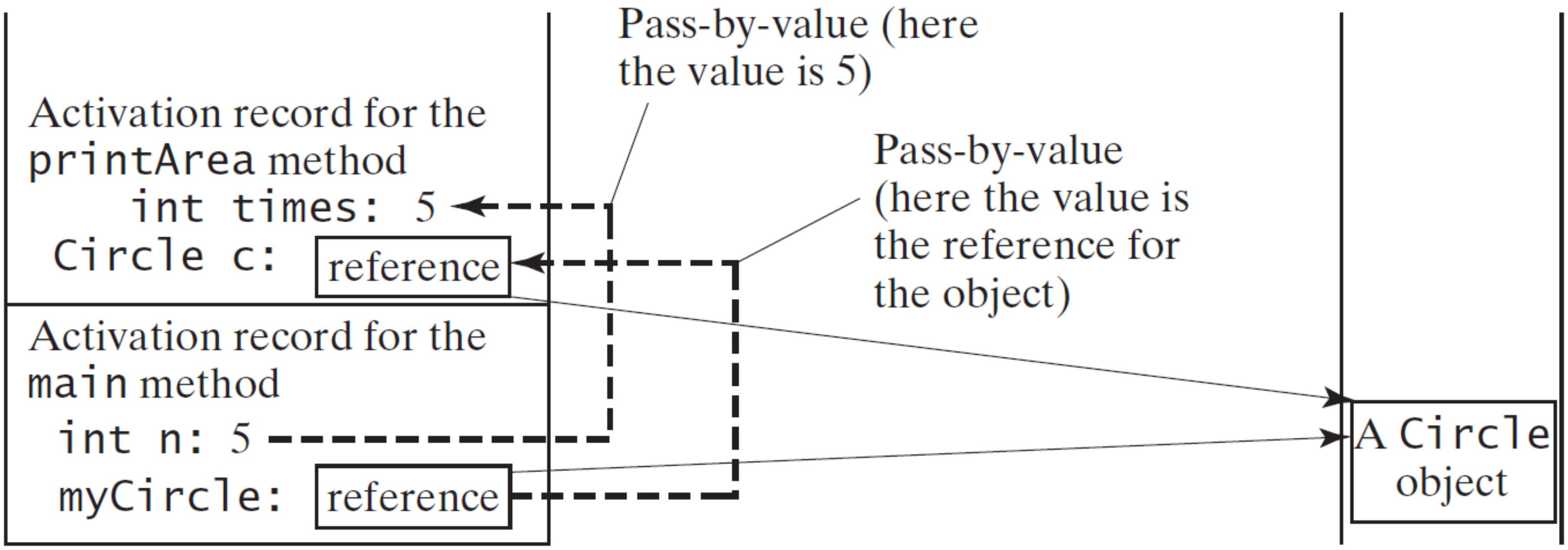
Passing a primitive type value and a reference value



```
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
```

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.

Passing Objects in Methods


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

Array Object

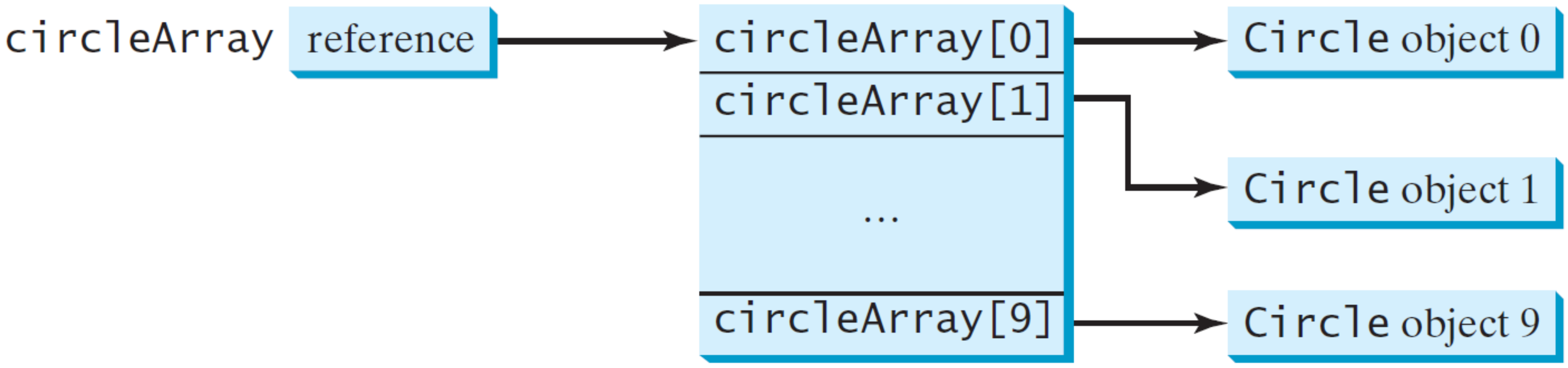
Array of Objects

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

An array of objects is actually an *array of reference variables*. So invoking circleArray[1].getArea() involves two levels of referencing as shown in the next figure. circleArray references to the entire array. circleArray[1] references to a Circle object.

Array of Objects

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



Array of Objects

- To initialize **circleArray**, you can use a **for** loop

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

- An array of objects is actually an *array of reference variables*.
- 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**.

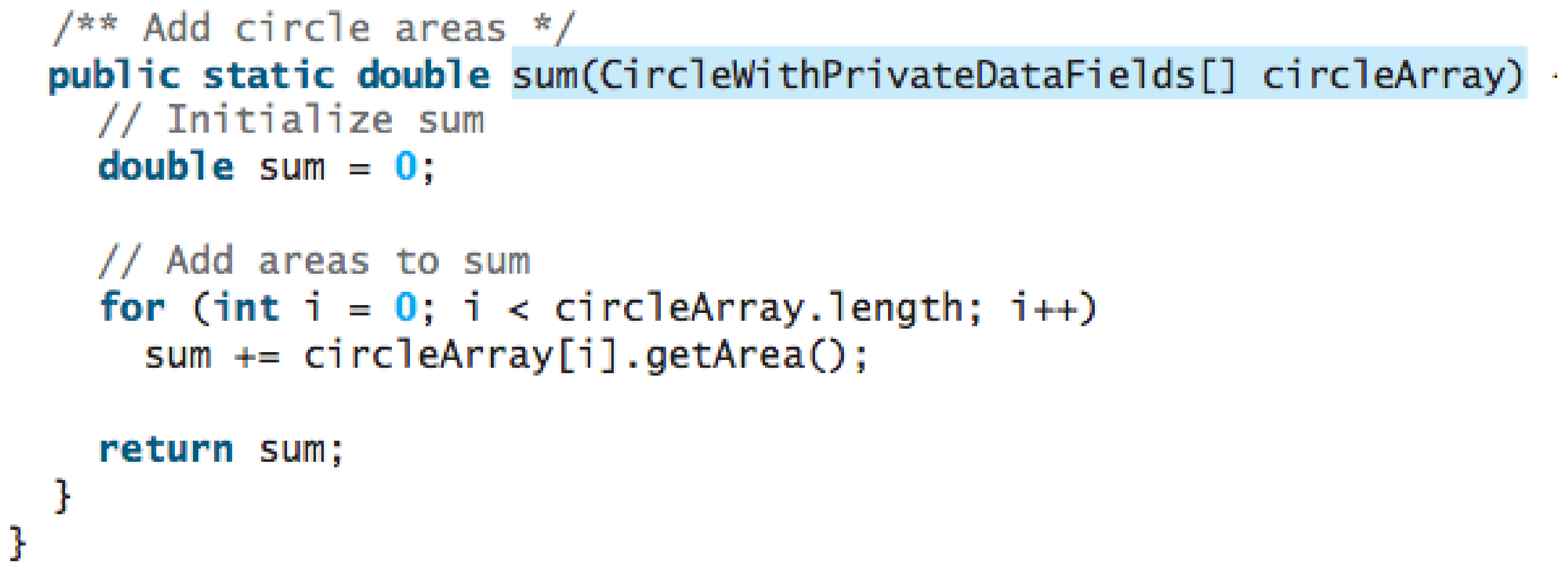
Summarizing the areas of the circles

```
public class TotalArea {
  /** Main method */
  public static void main(String[] args) {
    // Declare circleArray
    CircleWithPrivateDataFields[] circleArray;

    // Create circleArray
    circleArray = createCircleArray();

    // Print circleArray and total areas of the circles
    printCircleArray(circleArray);
  }

  /** Create an array of Circle objects */
  public static CircleWithPrivateDataFields[] createCircleArray() {
    CircleWithPrivateDataFields[] circleArray =
      new CircleWithPrivateDataFields[5];

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

    // Return Circle array
    return circleArray;
  }
```

Summarizing the areas of the circles

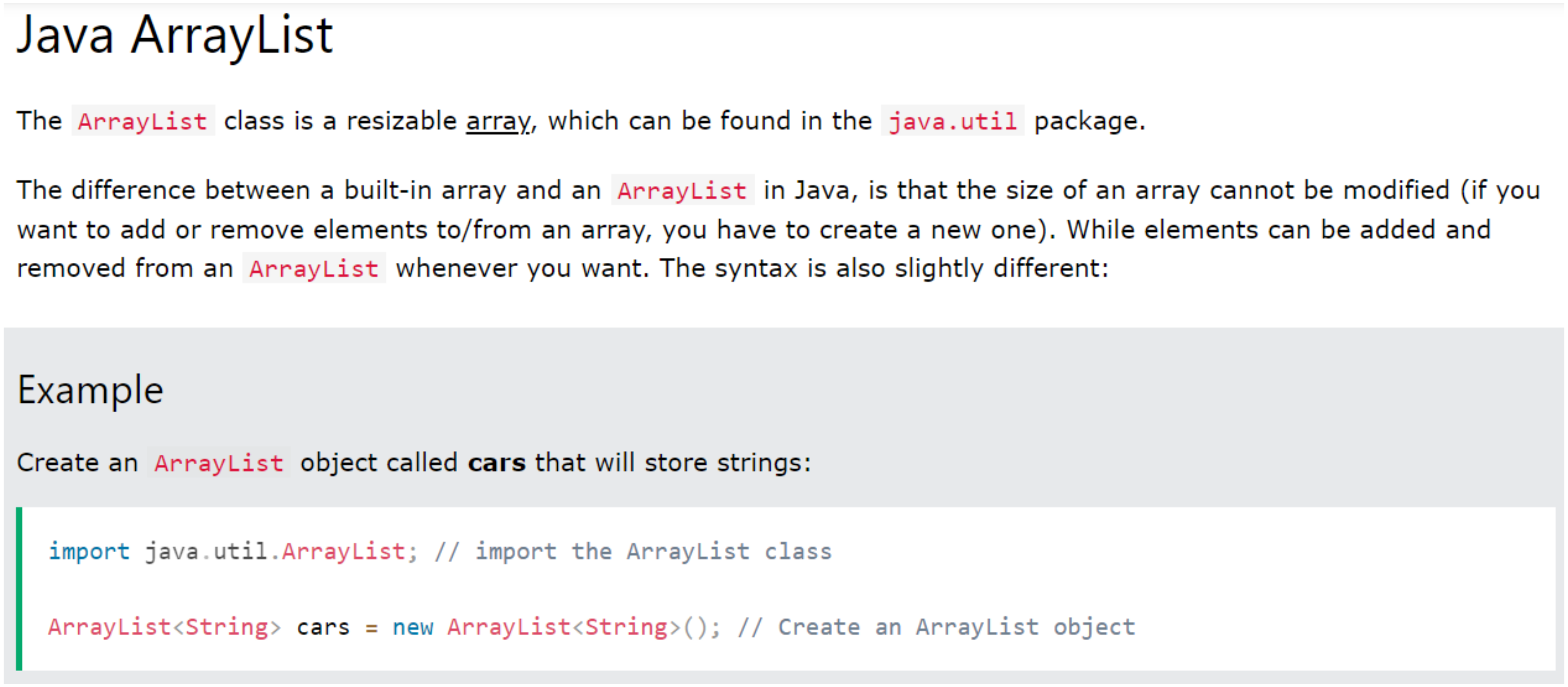
```java
/** Print an array of circles and their total area */
public static void printCircleArray(
    CircleWithPrivateDataFields[] circleArray) {
  System.out.printf("%-30s%-15s\n", "Radius", "Area");
  for (int i = 0; i < circleArray.length; i++) {
    System.out.printf("%-30f%-15f\n", circleArray[i].getRadius(),
      circleArray[i].getArea());
  }

  System.out.println("------------------------------------------");

  // Compute and display the result
  System.out.printf("%-30s%-15f\n", "The total area of circles is",
   sum(circleArray) );
}
```

Summarizing the areas of the circles



```java
  /** Add circle areas */
  public static double sum(CircleWithPrivateDataFields[] circleArray)
    // Initialize sum
    double sum = 0;

    // Add areas to sum
    for (int i = 0; i < circleArray.length; i++)
      sum += circleArray[i].getArea();

    return sum;
  }
}
```

Array List



Java ArrayList

The `ArrayList` class is a resizable array, which can be found in the `java.util` package.

The difference between a built-in array and an `ArrayList` in Java, is that the size of an array cannot be modified (if you want to add or remove elements to/from an array, you have to create a new one). While elements can be added and removed from an `ArrayList` whenever you want. The syntax is also slightly different:

Example

Create an `ArrayList` object called **cars** that will store strings:

```
import java.util.ArrayList; // import the ArrayList class

ArrayList<String> cars = new ArrayList<String>(); // Create an ArrayList object
```

Add Items

The `ArrayList` class has many useful methods. For example, to add elements to the `ArrayList`, use the `add()` method:

Example

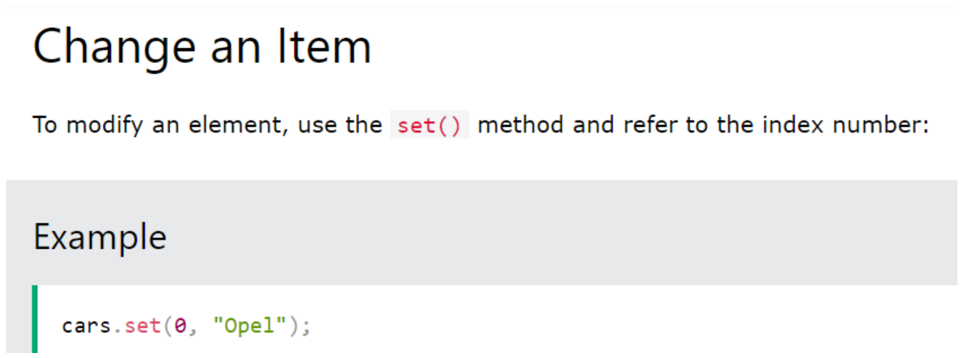
```
import java.util.ArrayList;

public class Main {
  public static void main(String[] args) {
    ArrayList<String> cars = new ArrayList<String>();
    cars.add("Volvo");
    cars.add("BMW");
    cars.add("Ford");
    cars.add("Mazda");
    System.out.println(cars);
  }
}
```

Access an Item

To access an element in the `ArrayList`, use the `get()` method and refer to the index number:

Example

```
cars.get(0);
```

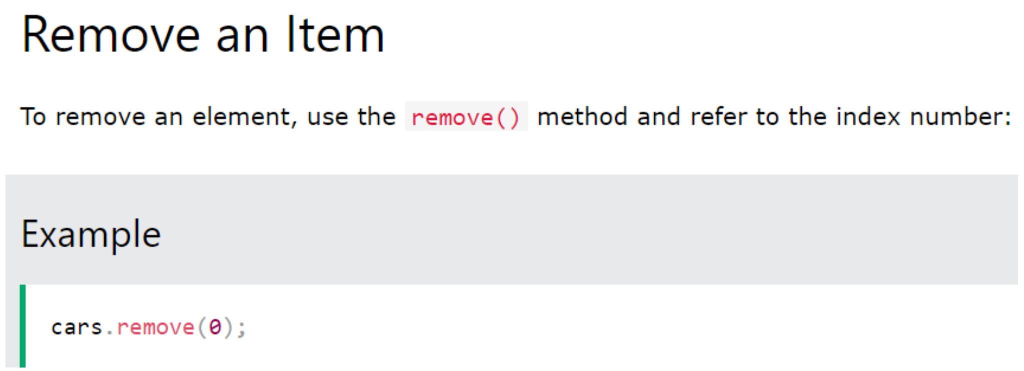


Change an Item

To modify an element, use the `set()` method and refer to the index number:

Example

```
cars.set(0, "Opel");
```

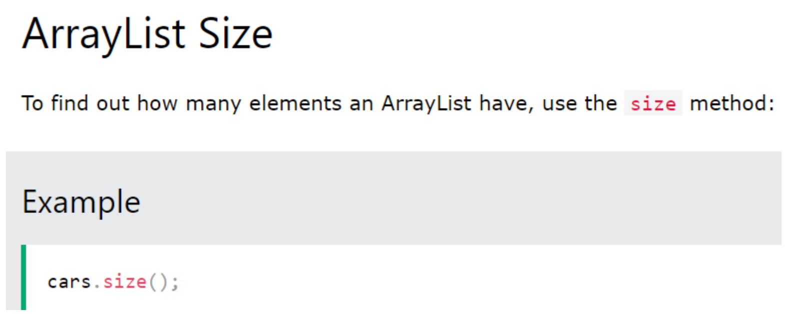


Remove an Item

To remove an element, use the `remove()` method and refer to the index number:

Example

```
cars.remove(0);
```



ArrayList Size

To find out how many elements an ArrayList have, use the `size` method:

Example

```
cars.size();
```

Loop Through an ArrayList

Loop through the elements of an `ArrayList` with a `for` loop, and use the `size()` method to specify how many times the loop should run:

Example

```java
public class Main {
  public static void main(String[] args) {
    ArrayList<String> cars = new ArrayList<String>();
    cars.add("Volvo");
    cars.add("BMW");
    cars.add("Ford");
    cars.add("Mazda");
    for (int i = 0; i < cars.size(); i++) {
      System.out.println(cars.get(i));
    }
  }
}
```

Other Types

Elements in an ArrayList are actually objects. In the examples above, we created elements (objects) of type "String". Remember that a String in Java is an object (not a primitive type). To use other types, such as int, you must specify an equivalent wrapper class: `Integer`. For other primitive types, use: `Boolean` for boolean, `Character` for char, `Double` for double, etc:

Example

Create an `ArrayList` to store numbers (add elements of type `Integer`):

```java
import java.util.ArrayList;

public class Main {
  public static void main(String[] args) {
    ArrayList<Integer> myNumbers = new ArrayList<Integer>();
    myNumbers.add(10);
    myNumbers.add(15);
    myNumbers.add(20);
    myNumbers.add(25);
    for (int i : myNumbers) {
      System.out.println(i);
    }
  }
```

Return and Passing Arraylist

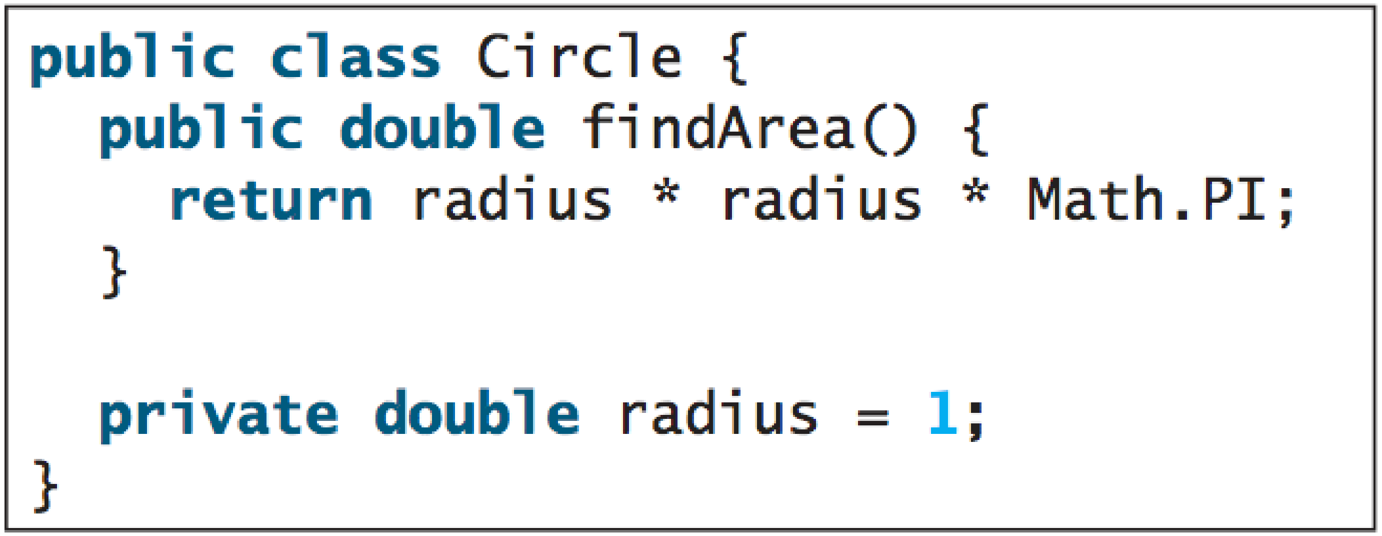
```
ArrayList<Integer> collect(ArrayList<Integer> data){
      return data;
   }
```

“This” Keyword according to scope variable

Scope of Variables

- ❑The scope of instance and static variables is the entire class. They can be declared anywhere inside a class.
- ❑The scope of a local variable (i.e. a variable defined in a method) 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.

Scope of Variables

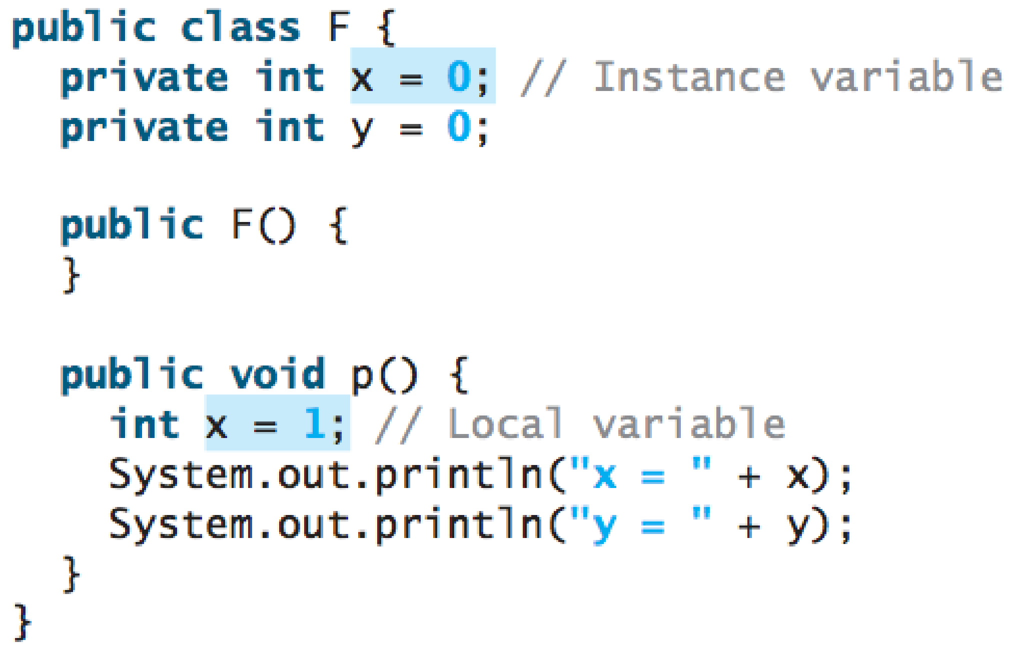


```
public class Circle {
  public double findArea() {
    return radius * radius * Math.PI;
  }

  private double radius = 1;
}
```

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

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*.

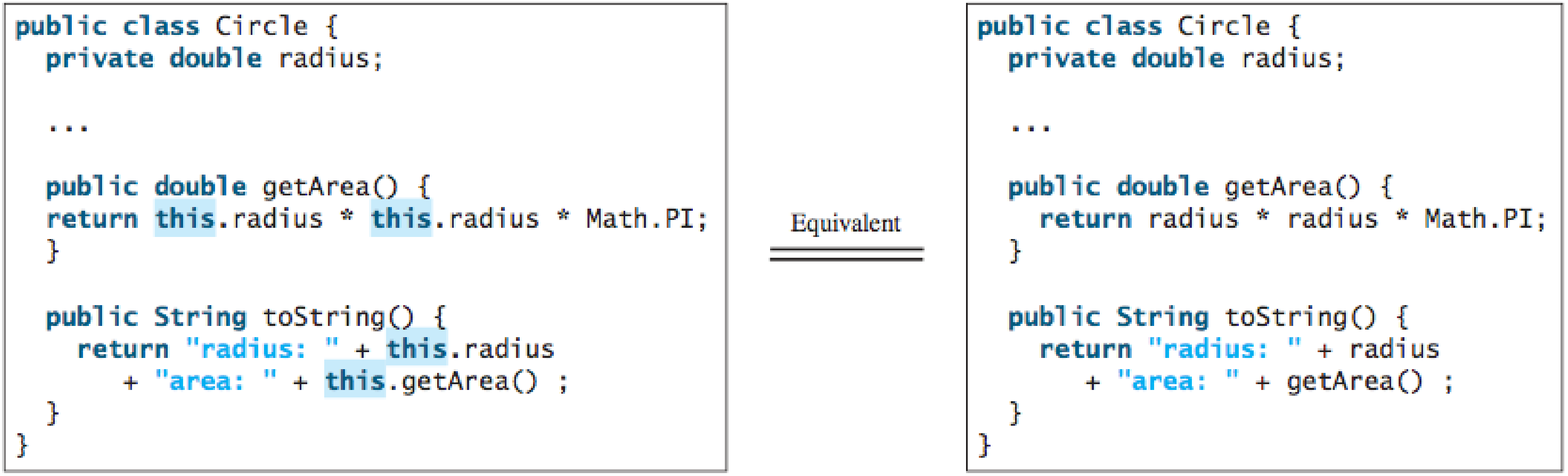


```
public class F {
  private int x = 0; // Instance variable
  private int y = 0;

  public F() {
  }

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

The this Keyword

- The <u>this</u> keyword is the name of a reference that refers to an object itself. One common use of the <u>this</u> keyword is reference a class’s *hidden data fields*.
- Another common use of the <u>this</u> keyword to enable a constructor to invoke another constructor of the same class.

The this Keyword

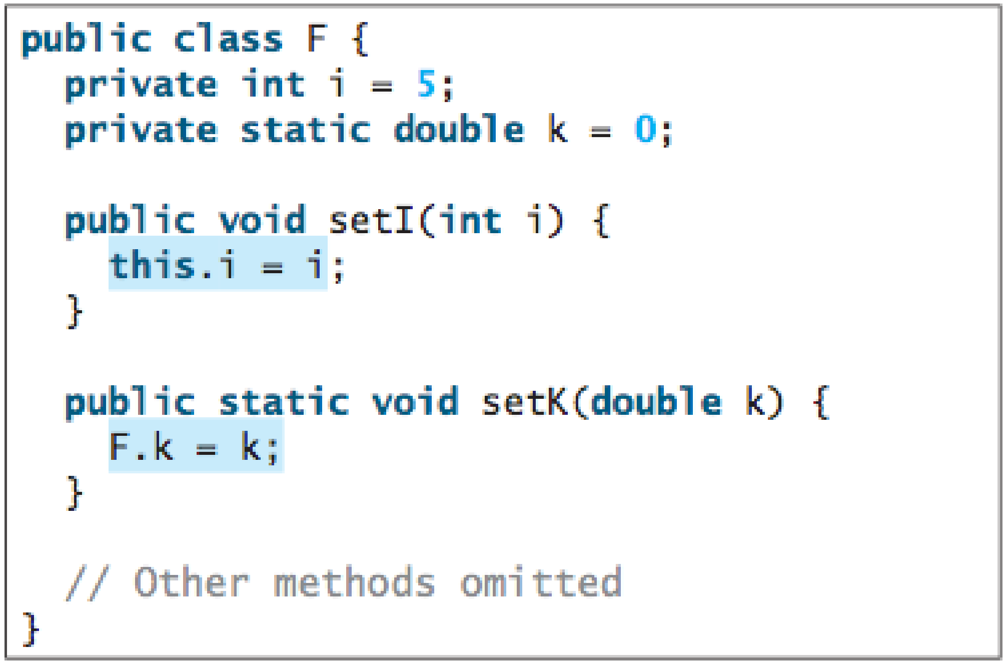


```java
public class Circle {
  private double radius;

  ...

  public double getArea() {
  return this.radius * this.radius * Math.PI;
  }

  public String toString() {
    return "radius: " + this.radius
       + "area: " + this.getArea() ;
  }
}
```

```java
public class Circle {
  private double radius;

  ...

  public double getArea() {
    return radius * radius * Math.PI;
  }

  public String toString() {
    return "radius: " + radius
       + "area: " + getArea() ;
  }
}
```

Reference the Hidden Data Fields

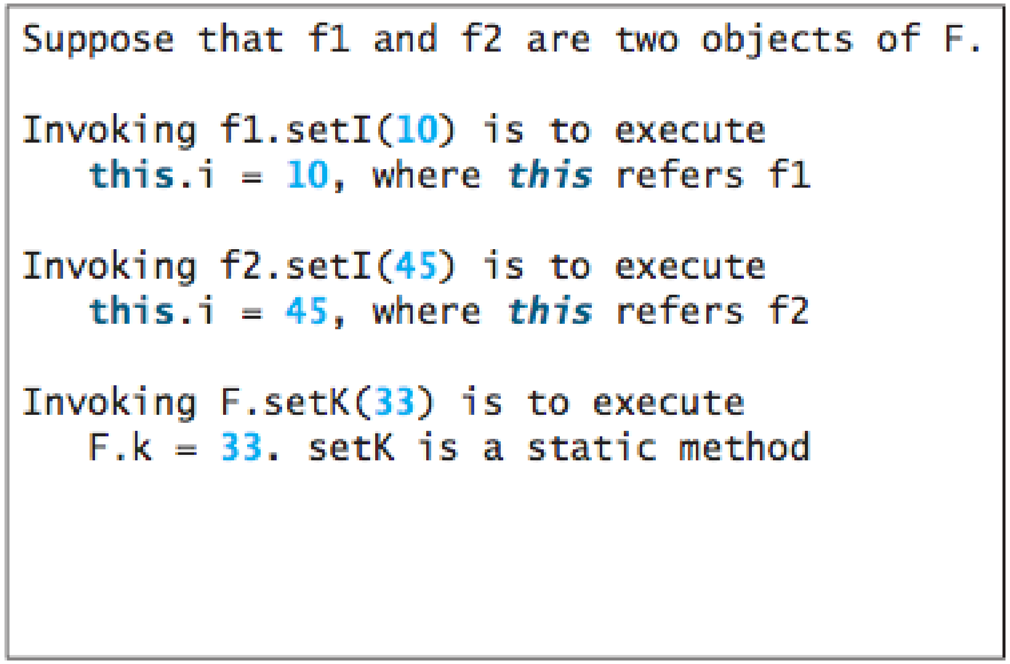
The **this** keyword can be used to reference a class's *hidden data fields*. For example, a data-field name is often used as the parameter name in a setter method for the data field. In this case, the data field is hidden in the setter method. You need to reference the hidden data-field name in the method in order to set a new value to it. A hidden static variable can be accessed using the keyword **this.**



```
public class F {
  private int i = 5;
  private static double k = 0;

  public void setI(int i) {
    this.i = i;
  }

  public static void setK(double k) {
    F.k = k;
  }

  // Other methods omitted
}
```

Reference the Hidden Data Fields

The **this** keyword gives us a way to reference the object that invokes an instance method. To invoke **f1.setI(10)**, **this.i = i** is executed, which assigns the value of parameter **i** to the data field **i** of this calling object **f1**. The keyword **this** refers to the object that invokes the instance method **setI**, as shown below:



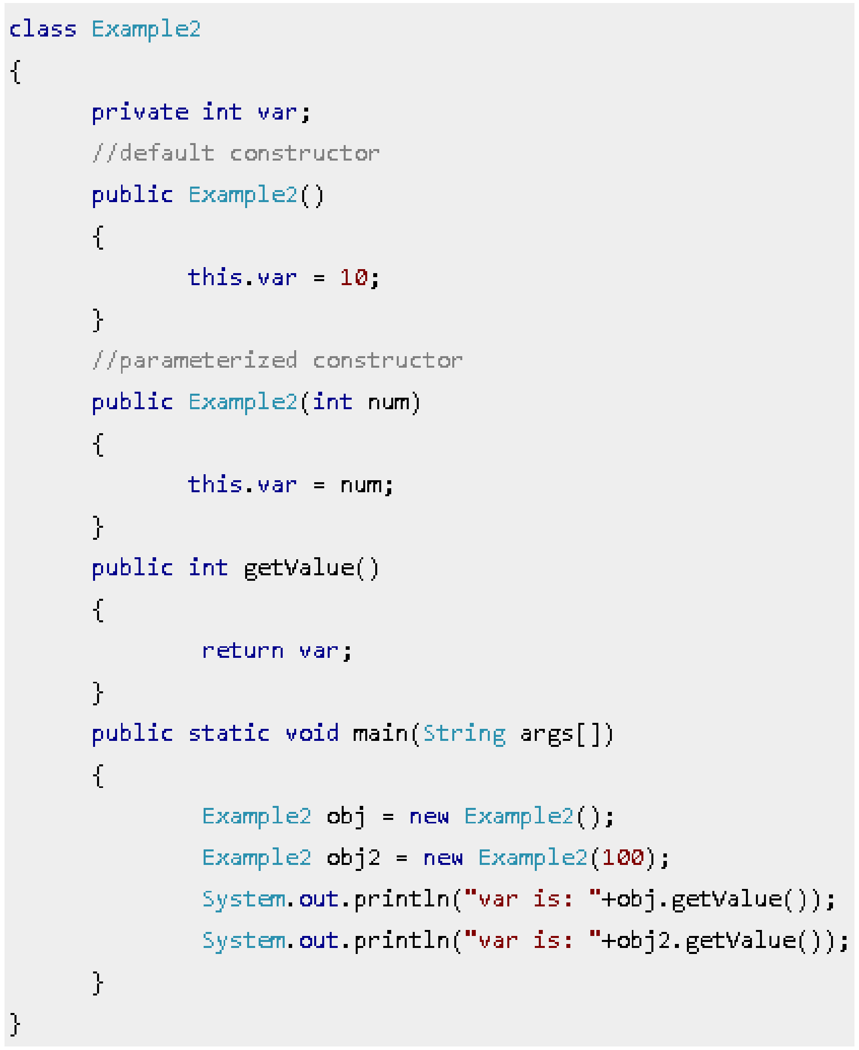
```
Suppose that f1 and f2 are two objects of F.

Invoking f1.setI(10) is to execute
   this.i = 10, where this refers f1

Invoking f2.setI(45) is to execute
   this.i = 45, where this refers f2

Invoking F.setK(33) is to execute
   F.k = 33. setK is a static method
```

Constructor overload and chaining

```
public class Hello {
    String name;
    //Constructor
    Hello(){
        this.name = "Beginners “;
    }
    public static void main(String[] args) {
        Hello obj = new Hello();
        System.out.println(obj.name);
    }
}
```

Parameterized :Overload

- Output
- var is: 10
- var is: 100



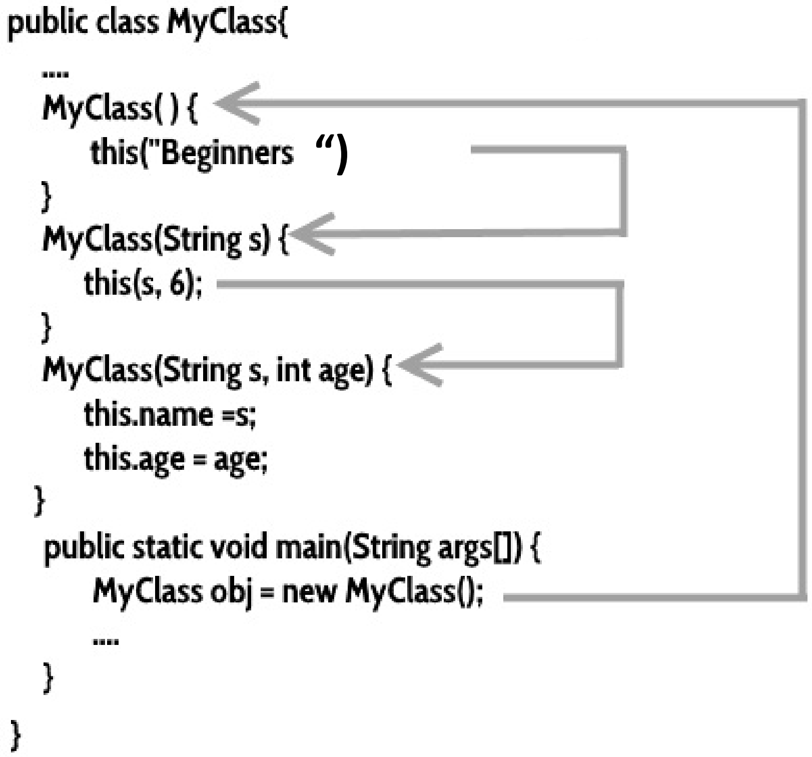
```
class Example2
{
        private int var;
        //default constructor
        public Example2()
        {
               this.var = 10;
        }
        //parameterized constructor
        public Example2(int num)
        {
               this.var = num;
        }
        public int getValue()
        {
                return var;
        }
        public static void main(String args[])
        {
                Example2 obj = new Example2();
                Example2 obj2 = new Example2(100);
                System.out.println("var is: "+obj.getValue());
                System.out.println("var is: "+obj2.getValue());
        }
}
```

Parameterized: Chaining

- Calling a constructor from another constructor of same class is known as Constructor chaining.
- Constructor Chaining in Java is used to
  - pass parameters through multiple different constructors using a single object.
  - perform multiple tasks through a single constructor instead of writing each task in a single constructor.
  - create a separate constructor for each task and make links or chains among them, so as to increase the readability of the code.
- Suppose, If we do not perform chaining among constructors and they require a specific parameter, then we will need to initialize that parameter twice in each constructor.
- Note: In Java, it is invalid and illegal to call the constructor directly by name. We have to use **this** to call a constructor.

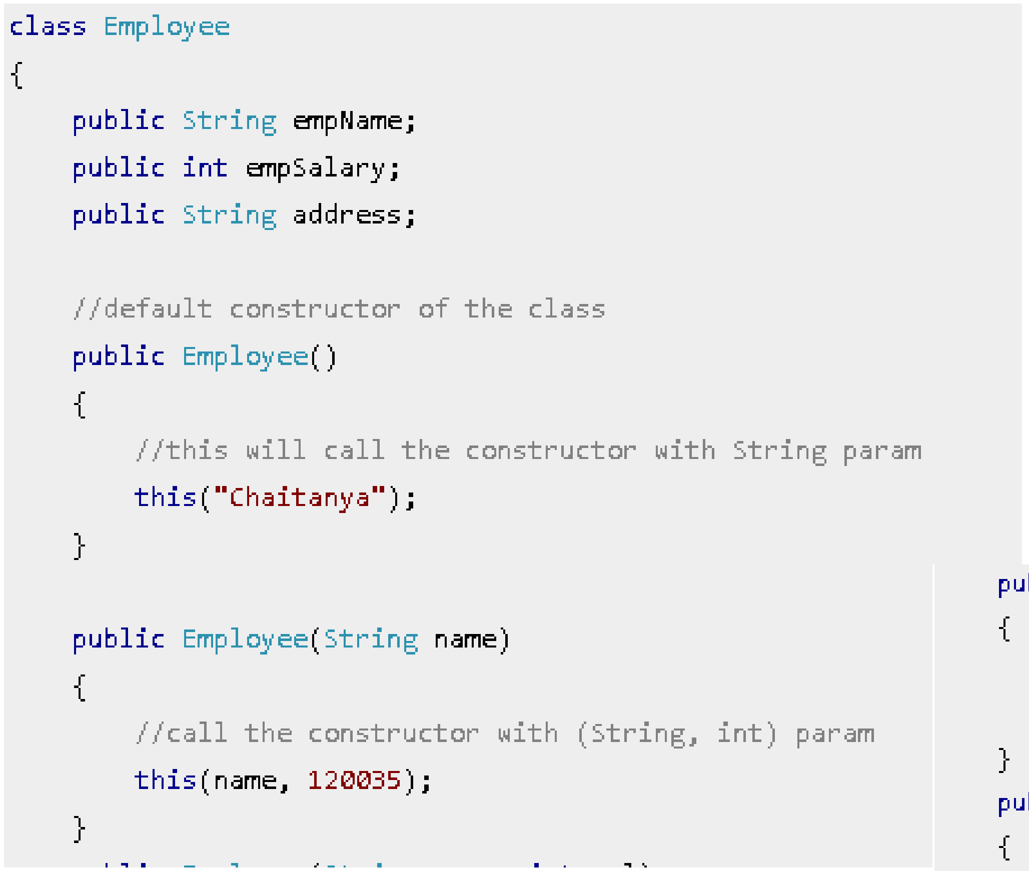
Parameterized: Chaining

- constructors with fewer arguments should call those with more

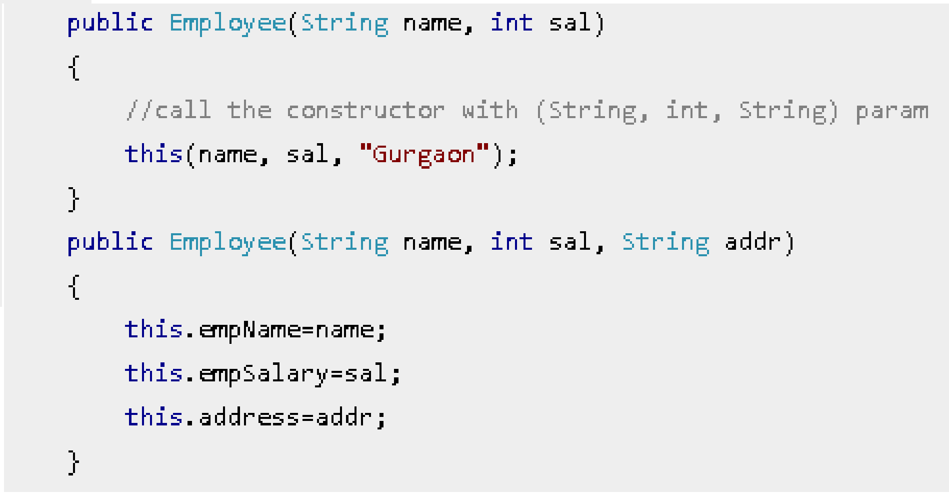


```
public class MyClass{
    ....
    MyClass( ) {
        this("Beginners  ")
    }
    MyClass(String s) {
        this(s, 6);
    }
    MyClass(String s, int age) {
        this.name =s;
        this.age = age;
    }
    public static void main(String args[]) {
        MyClass obj = new MyClass();
        ....
    }
}
```

Example

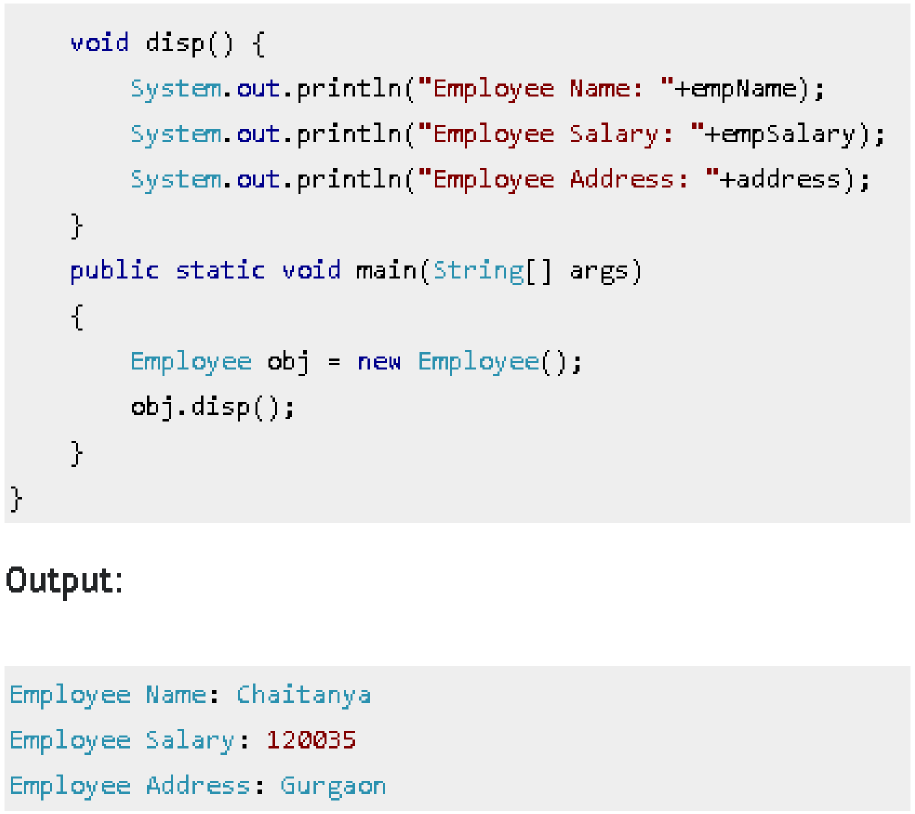


```
class Employee
{
    public String empName;
    public int empSalary;
    public String address;

    //default constructor of the class
    public Employee()
    {
        //this will call the constructor with String param
        this("Chaitanya");
    }

    public Employee(String name)
    {
        //call the constructor with (String, int) param
        this(name, 120035);
    }
```



```
public Employee(String name, int sal)
{
    //call the constructor with (String, int, String) param
    this(name, sal, "Gurgaon");
}
public Employee(String name, int sal, String addr)
{
    this.empName=name;
    this.empSalary=sal;
    this.address=addr;
}
```

Main



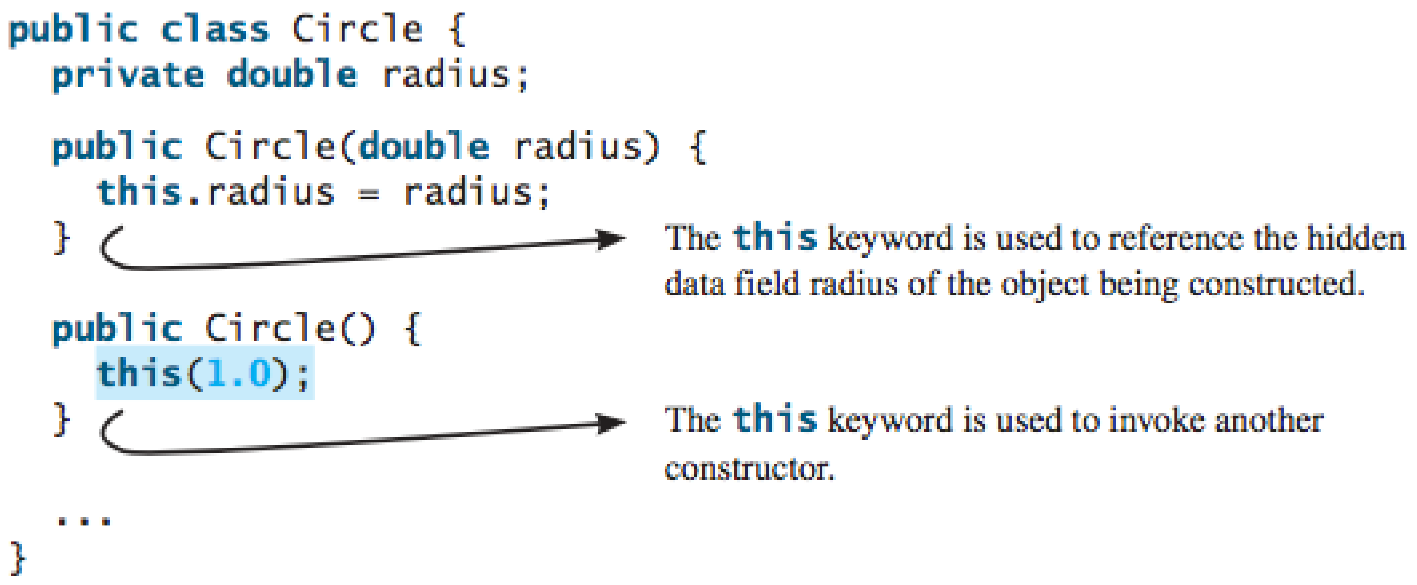
```java
    void disp() {
        System.out.println("Employee Name: "+empName);
        System.out.println("Employee Salary: "+empSalary);
        System.out.println("Employee Address: "+address);
    }
    public static void main(String[] args)
    {
        Employee obj = new Employee();
        obj.disp();
    }
}
```

Output:

```
Employee Name: Chaitanya
Employee Salary: 120035
Employee Address: Gurgaon
```

Calling Overloaded Constructor

The **this** keyword can be used to invoke another constructor of the same class. For example, you can rewrite the **Circle** class as follows:



```
public class Circle {
  private double radius;

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

  public Circle() {
    this(1.0);
  }

  ...
}
```

The **this** keyword is used to reference the hidden data field radius of the object being constructed.

The **this** keyword is used to invoke another constructor.

Contoh Soal

Buatlah sebuah program sederhana untuk mengelola daftar mata kuliah yang diambil oleh beberapa mahasiswa.

Gunakan konsep OOP dan memanfaatkan ArrayList untuk menyimpan daftar mata kuliah tiap mahasiswa.

Tiap mahasiswa dapat menambahkan dan menghapus mata kuliah yang diambil

dan juga dapat menampilkan daftar mata kuliah yang sedang diambilnya