CEN 419
Introduction to Java Programming

Dr. H. Esin ÜNAL

FALL 2021

Slides are modified from original slides of Y. Daniel Liang

OO Programming Concepts

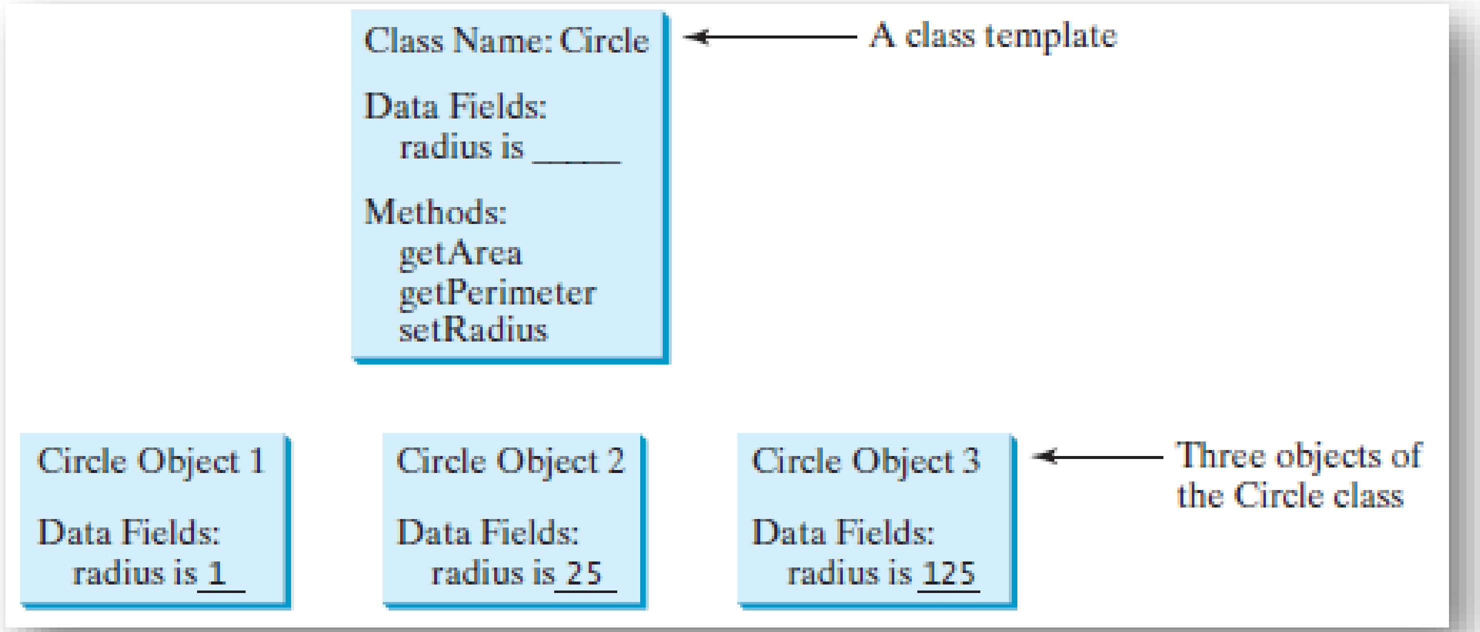
- An ***object*** represents an entity in the real world that can be distinctly identified. For example, a student, a desk, a circle, a button, and even a loan can all be viewed as objects.
- A ***class*** defines the ***properties*** and ***behaviors*** for objects.
- Object-oriented programming (OOP)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

Objects-Classes

- An object has a **unique identity**, **state**, and **behavior**.
- The ***state (properties or attributes)*** of an object consists of a set of *data fields* with their current values (e.g. A circle has a data field radius).
- The ***behavior (actions)*** of an object is defined by a set of methods (e.g. getArea(), getPerimeter()).
- *An object has both a state and behavior. The state defines the object, and the behavior defines what the object does.*

Objects-Classes

- 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***.



Objects-Classes

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

Constructors

Constructors are special kind of methods that are invoked to create objects using the **new** operator.

```
Circle() {
}


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

Constructors

- They have three peculiarities:
  - 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***.

Constructors

- Like regular methods, constructors can be overloaded.
- It is a common mistake to put the **void** keyword in front of a constructor.
  - For example,

```
    public void Circle() {
    }
```

  In this case, Circle() is a method, not a constructor.

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

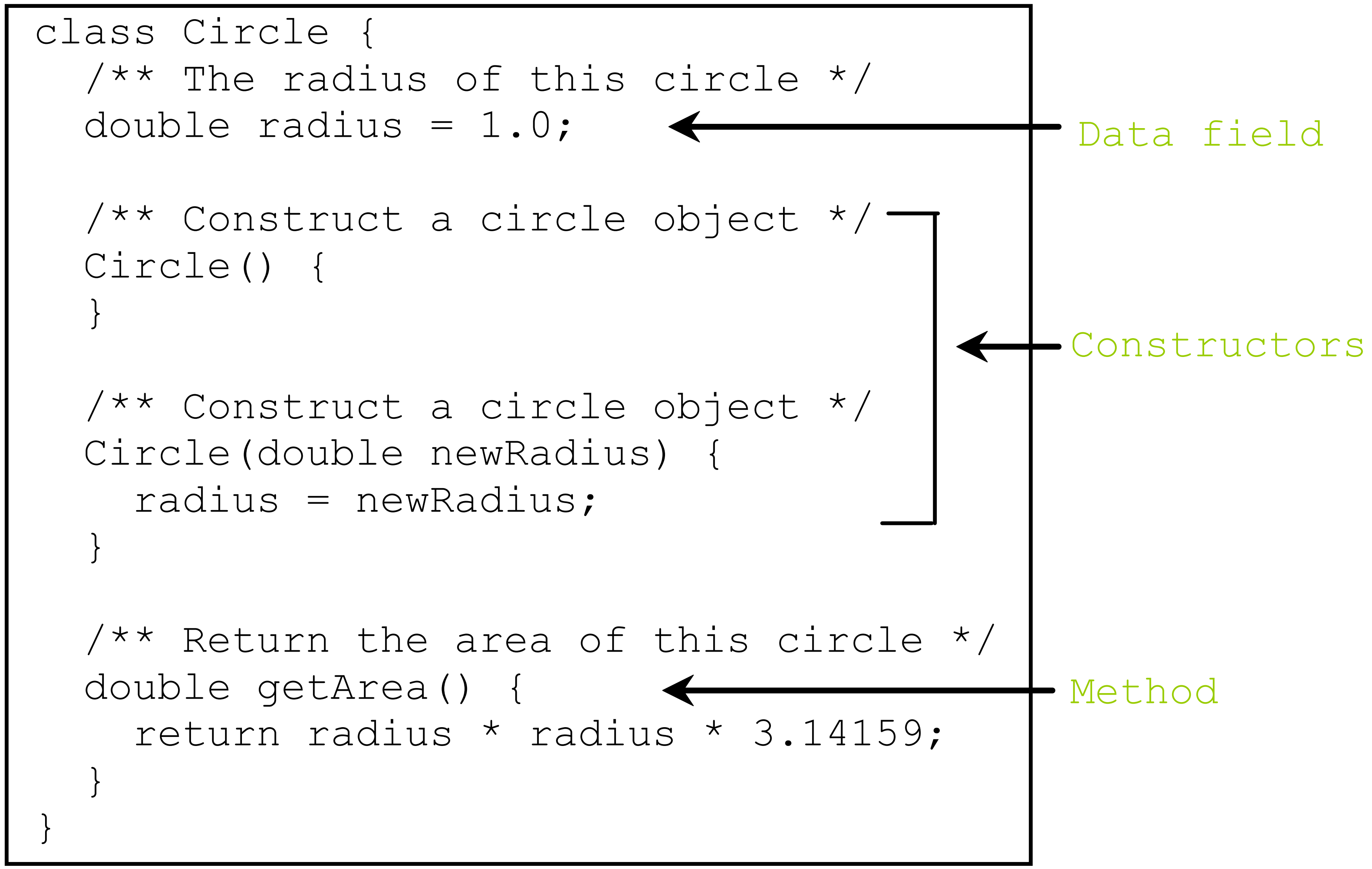
Default Constructor

- A class may be defined without constructors.

- In this case, a no-arg constructor with an empty body is implicitly defined in the class.

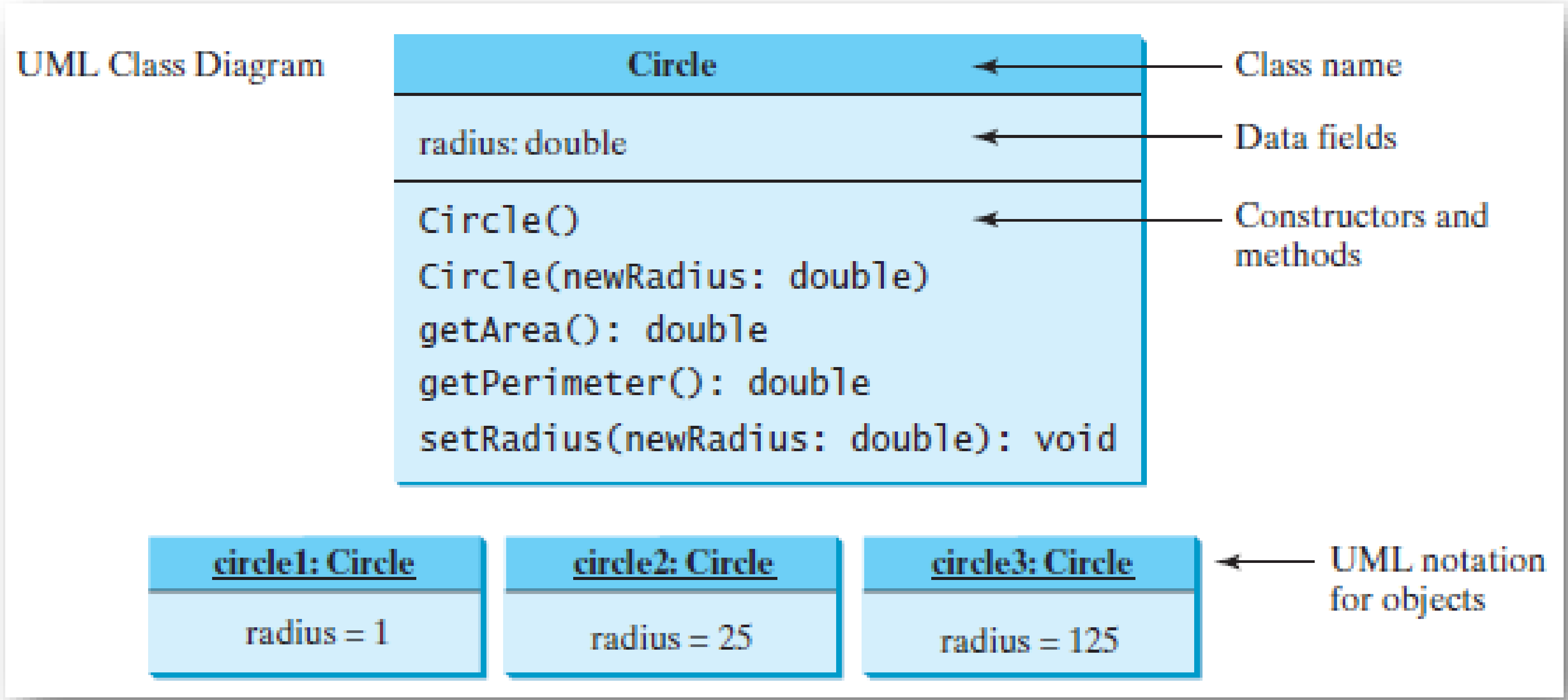
This constructor, called a ***default constructor***, is provided automatically ***only if no constructors are explicitly defined*** *in the class*.

Objects-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;
  }
}
```

UML Class Diagram



Example 1: Defining Classes and Creating Objects

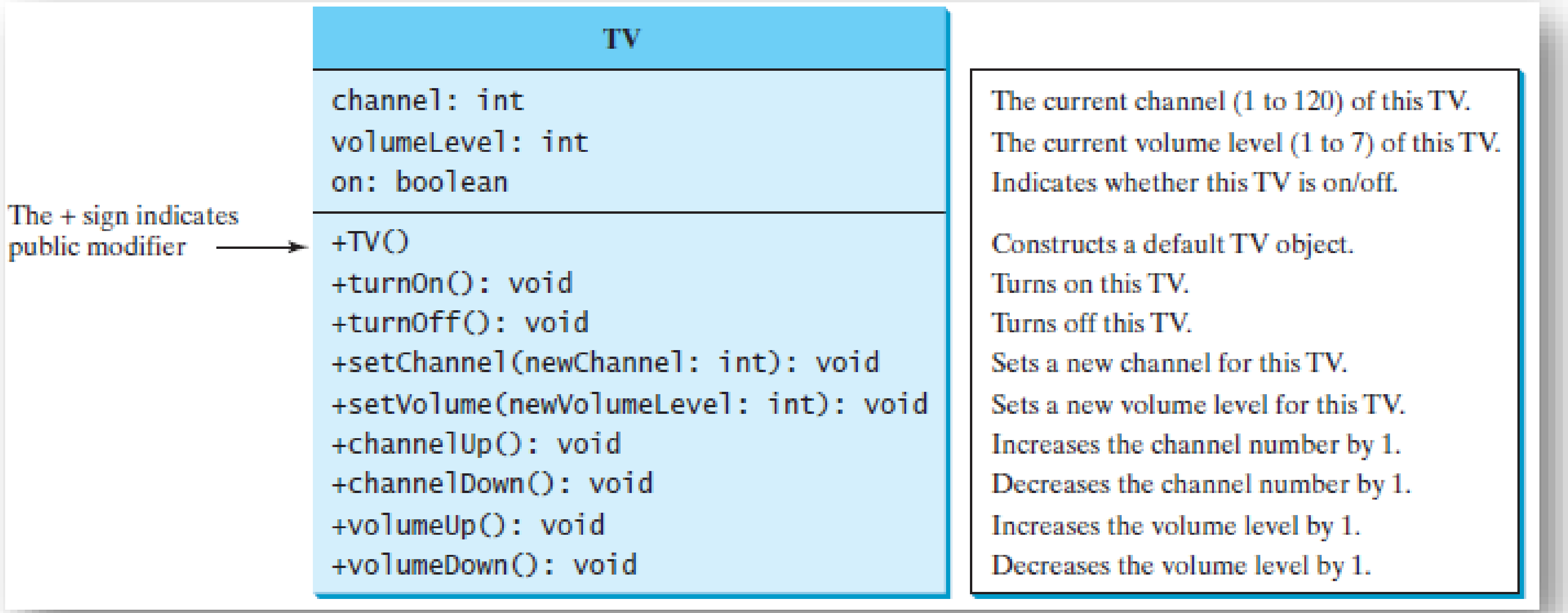
Objective: Demonstrate creating objects, accessing data, and using methods.

Test Simple Circle

Intro to Java Programming, Y. Daniel Liang - TestSimpleCircle.java (pearsoncmg.com)

- 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.
- Each class in the source code is compiled into a **.class** file.

Example 2: Defining Classes and Creating Objects



The + sign indicates 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.

Test TV

Intro to Java Programming, Y. Daniel Liang - TestTV.java (pearsoncmg.com)

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

Declaring Object Reference Variables

- To reference an object, assign the object to a reference variable.
- To declare a reference variable, use the syntax:

```
ClassName objectRefVar;
```

Reference type   Reference variable

<u>Example:</u>

```
Circle myCircle;
```

Creating Objects Using Constructors

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

```
new ClassName();
```

Example:

```
new Circle();
```

```
//creates an object of the Circle class using the first constructor on slide 11
```

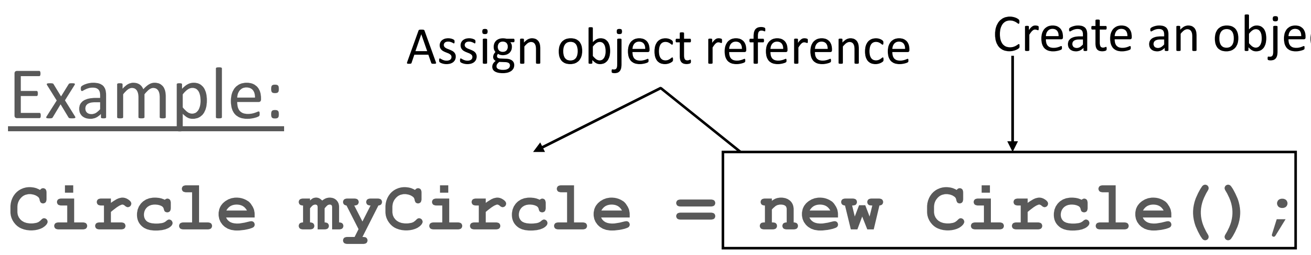
```
new Circle(5.0);
```

```
//creates an object of the Circle class using the second constructor on slide 11
```

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:

  `objectRefVar.data`

  e.g., `myCircle.radius`

- Invoking the object’s method:

  `objectRefVar.methodName(arguments)`

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

Caution

- 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 SimpleCircle.getArea()? (SimpleCircle is a class name like Math class)

The answer is NO. All the methods used before this chapter are static methods, which are defined using the static keyword. However, getArea() is non-static. It must be invoked from an object using:

```
    objectRefVar.methodName(arguments)
    (e.g. myCircle.getArea())
```

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. **String** is a predefined Java class.

```
public 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; // c has default value '\u0000'
}
```

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.

```
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);
  }
}
```

Caution

However, 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);
  }
}
```

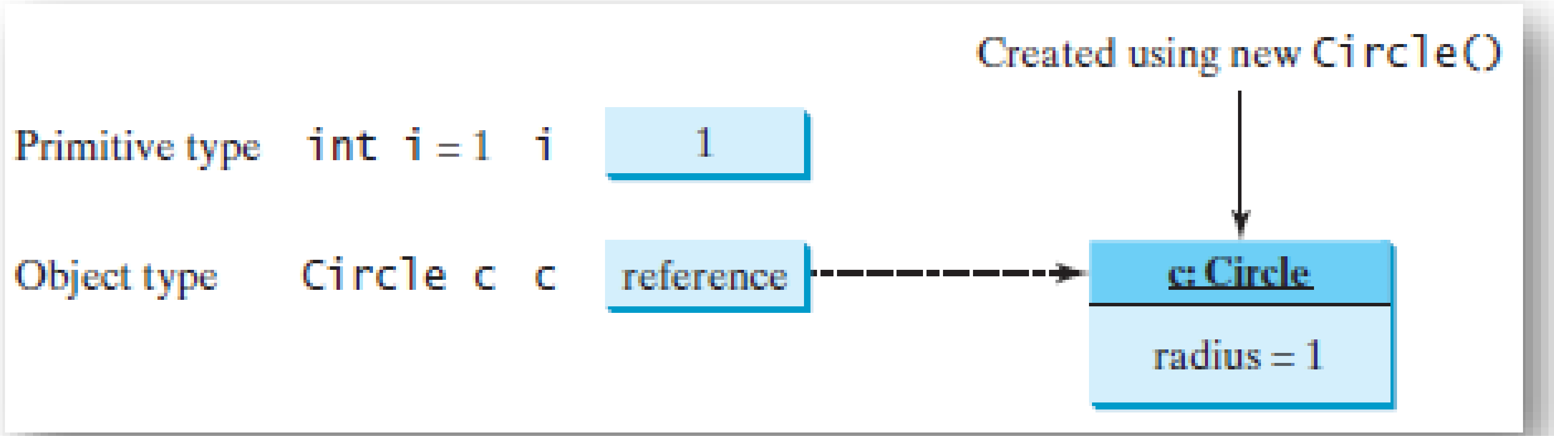
Compile error: variable not initialized

Caution

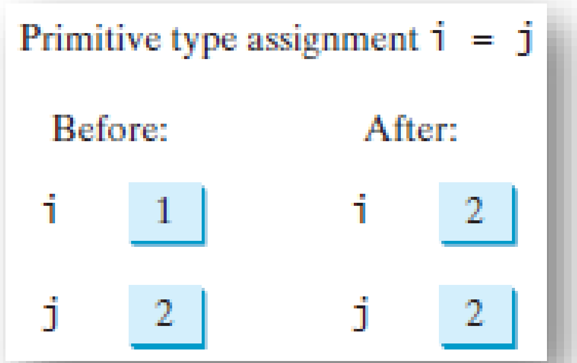
- **NullPointerException** is a common runtime error.
- It occurs when you invoke a method on a reference variable with a **null** value.
- Make sure you assign an object reference (with new operator) to the variable before invoking the method through the reference variable.

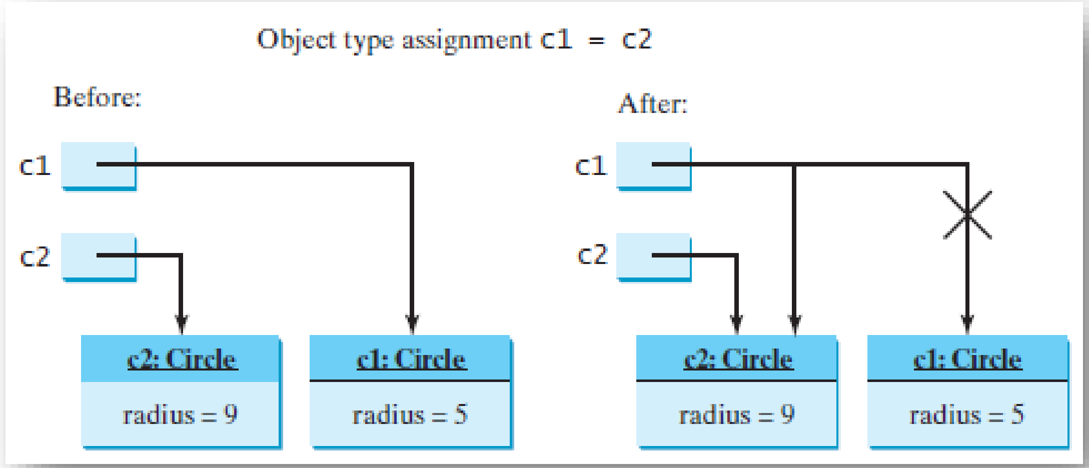
Differences between Variables of Primitive Data Types and Object Types

- ☞ 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





☞ 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

NOTE: 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 useful.
- This object is known as garbage. Garbage is automatically collected by JVM. This process is called garbage collection

TIP

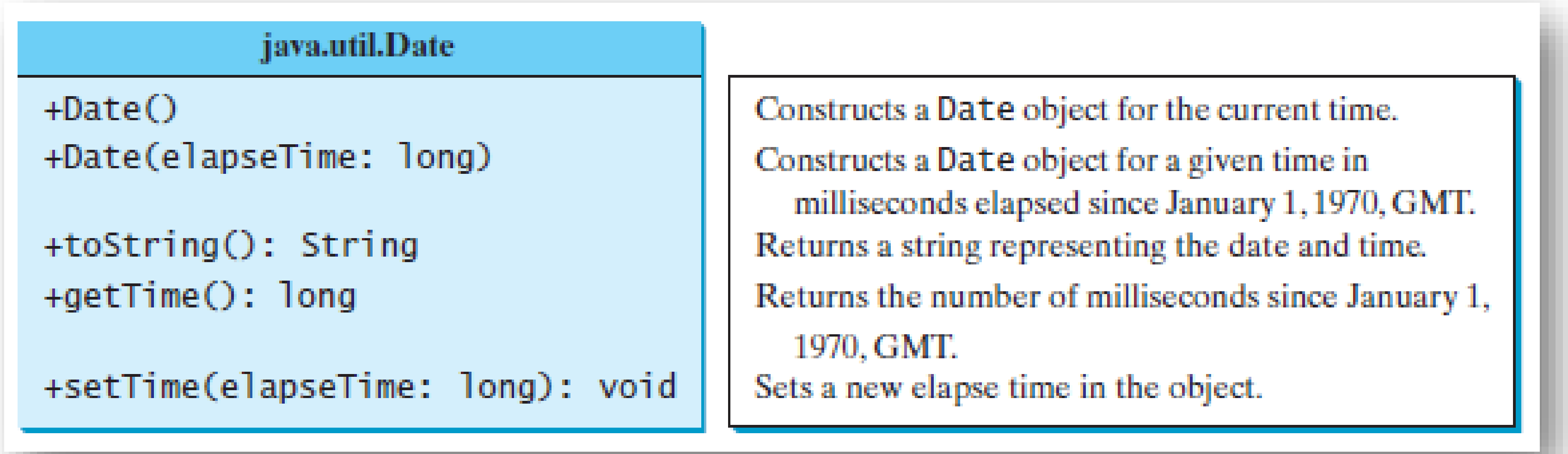
- If you know that an object is no longer needed, you can explicitly assign null to a reference variable for the object.
- The JVM will automatically collect the space if the object is not referenced by any variable .

Using Classes from the Java Library

- The Java API contains a rich set of classes for developing Java programs.
  - Date class
  - Random class
  - Point2D class …etc….

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.


java.util.Date	
`+Date()`	Constructs a Date object for the current time.
`+Date(elapseTime: long)`	Constructs a Date object for a given time in milliseconds elapsed since January 1, 1970, GMT.
`+toString(): String`	Returns a string representing the date and time.
`+getTime(): long`	Returns the number of milliseconds since January 1, 1970, GMT.
`+setTime(elapseTime: long): void`	Sets a new elapse time in the object.

The Date Class Example

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());
```

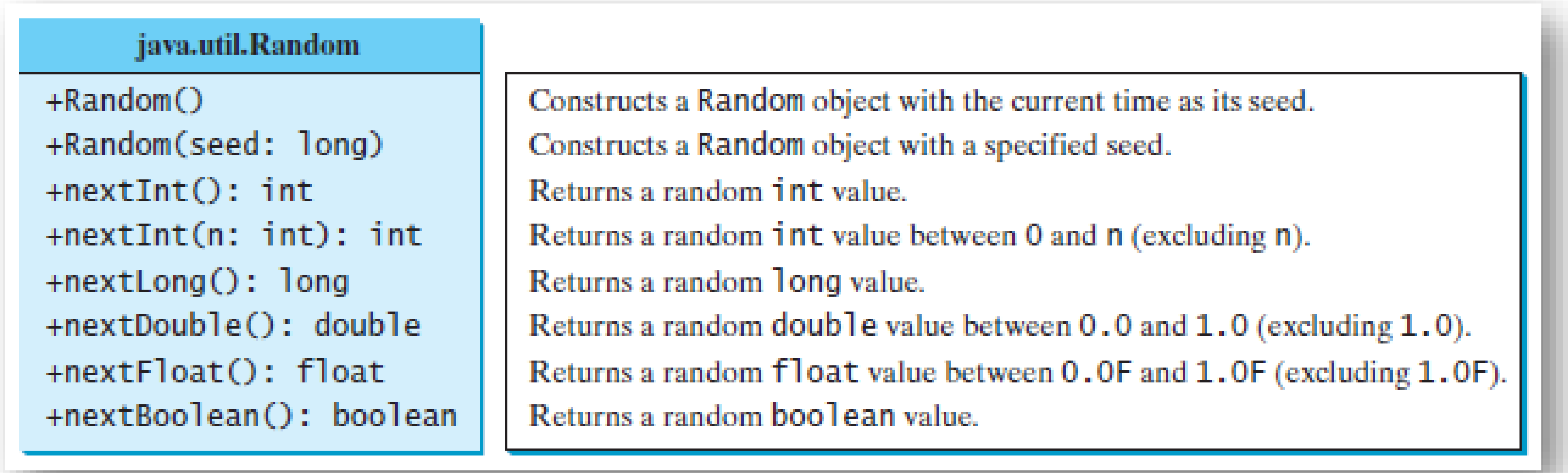
displays a string like:

```
The elapsed time since Jan 1, 1970 is 1493724045811
milliseconds
Tue May 02 14:20:45 EET 2017
```

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.



java.util.Random	
+Random()	Constructs a Random object with the current time as its seed.
+Random(seed: long)	Constructs a Random object with a specified seed.
+nextInt(): int	Returns a random int value.
+nextInt(n: int): int	Returns a random int value between 0 and n (excluding n).
+nextLong(): long	Returns a random long value.
+nextDouble(): double	Returns a random double value between 0.0 and 1.0 (excluding 1.0).
+nextFloat(): float	Returns a random float value between 0.0F and 1.0F (excluding 1.0F).
+nextBoolean(): boolean	Returns a random boolean value.

The Random Class Example

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 (*Default seed value is the current elapsed time*).

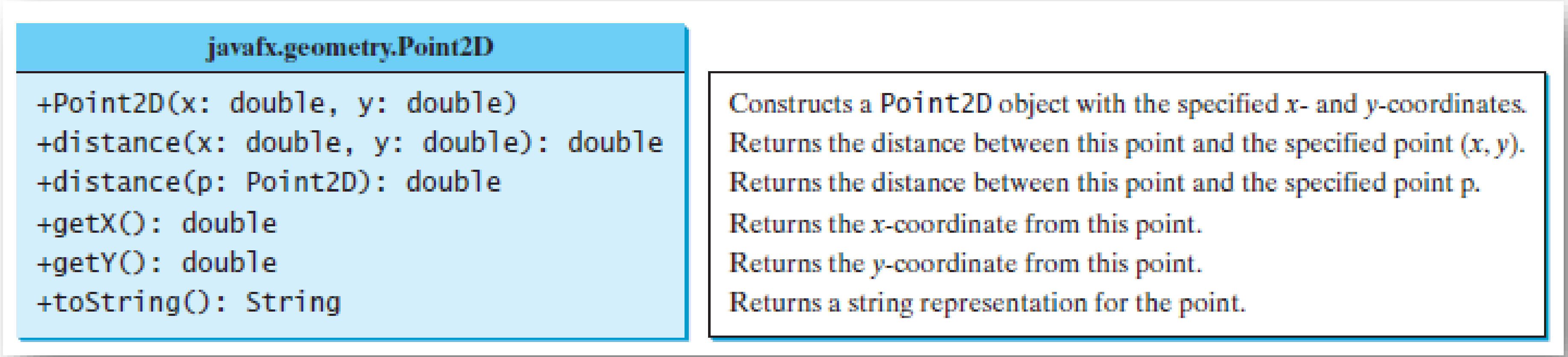
```
Random random1 = new Random(3);
System.out.print("From random1: ");
for (int i = 0; i < 10; i++)
  System.out.print(random1.nextInt(1000) + " ");
Random random2 = new Random(3);
System.out.print("\nFrom random2: ");
for (int i = 0; i < 10; i++)
  System.out.print(random2.nextInt(1000) + " ");
```

Output:

```
From random1: 734 660 210 581 128 202 549 564 459 961
From random2: 734 660 210 581 128 202 549 564 459 961
```

The **Point2D** Class

Java API has a convenient **Point2D** class in the **javafx.geometry** package for representing a point in a two-dimensional plane.



javafx.geometry.Point2D	
`+Point2D(x: double, y: double)`	Constructs a Point2D object with the specified *x*- and *y*-coordinates.
`+distance(x: double, y: double): double`	Returns the distance between this point and the specified point (*x*, *y*).
`+distance(p: Point2D): double`	Returns the distance between this point and the specified point p.
`+getX(): double`	Returns the *x*-coordinate from this point.
`+getY(): double`	Returns the *y*-coordinate from this point.
`+toString(): String`	Returns a string representation for the point.

Point2D Class

Intro to Java Programming, Y. Daniel Liang - TestPoint2D.java (pearsoncmg.com)

INSTANCE Variables, and Methods

- *Instance variables* belong to a specific instance. It is not shared among objects of the same class.
  - Circle circle1 = **new** Circle();

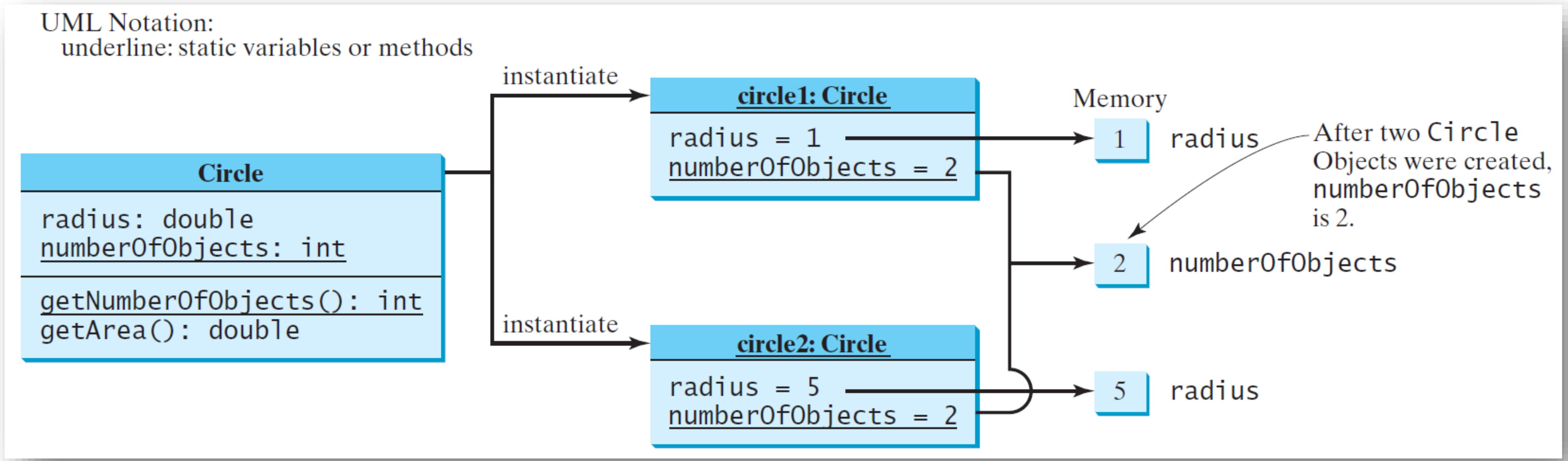
    Circle circle2 = **new** Circle(**5**);

    The **radius** in **circle1** is independent of the **radius** in **circle2** and is stored in a different memory location. Changes made to **circle1**'s **radius** do not affect **circle2**'s **radius**, and vice versa.
- *Instance methods* are invoked by an 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.
- *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



Instance variables belong to the instances and have memory storage independent of one another.

Static variables are shared by all the instances of the same class.

Example

- Instance methods (e.g., **getArea()**) and instance data (e.g., **radius**) belong to instances and ***can be used only after the instances are created***. They are accessed via a reference variable.
- Static methods (e.g., **getNumberOfObjects()**) and static data (e.g., **numberOfObjects**) ***can be accessed from a reference variable or from their class name***.

Circle with Static Members

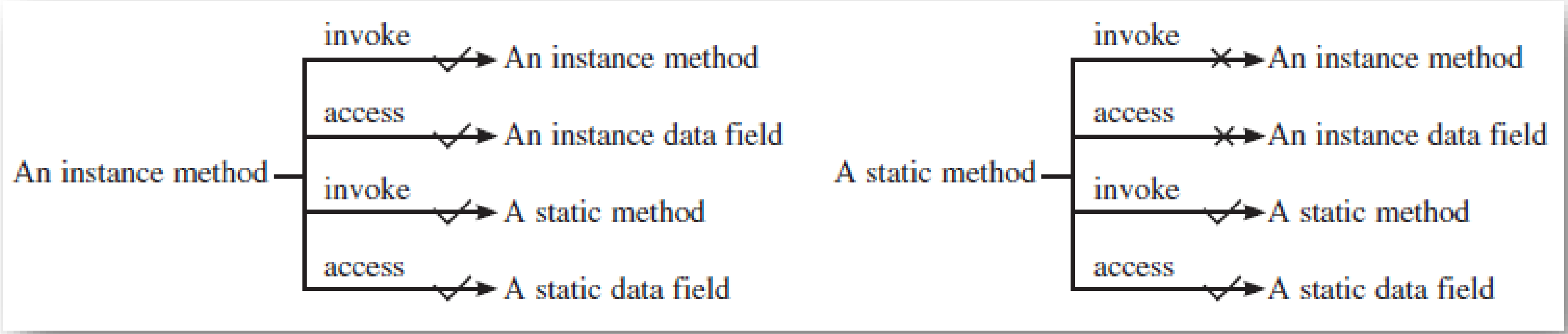
Intro to Java Programming, Y. Daniel Liang - CircleWithStaticMembers.java (pearsoncmg.com)

Test Circle with Static Members

Intro to Java Programming, 9E - TestCircleWithStaticDataFields.java (pearsoncmg.com)

STATIC vs INSTANCE Variables and Methods

- An instance method can invoke an instance or static method and access an instance or static data field.
- A static method can invoke a static method and access a static data field. However, a static method cannot invoke an instance method or access an instance data field, since static methods and static data fields don't belong to a particular object.


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

Example

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

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

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

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

Example

```
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));
    }
}
```

Visibility Modifiers

- *Visibility modifiers can be used to specify the visibility of a class and its members.*
- There are four different visibility modifiers:
  - default
  - public
  - private
  - protected (will be given later)

Visibility Modifiers

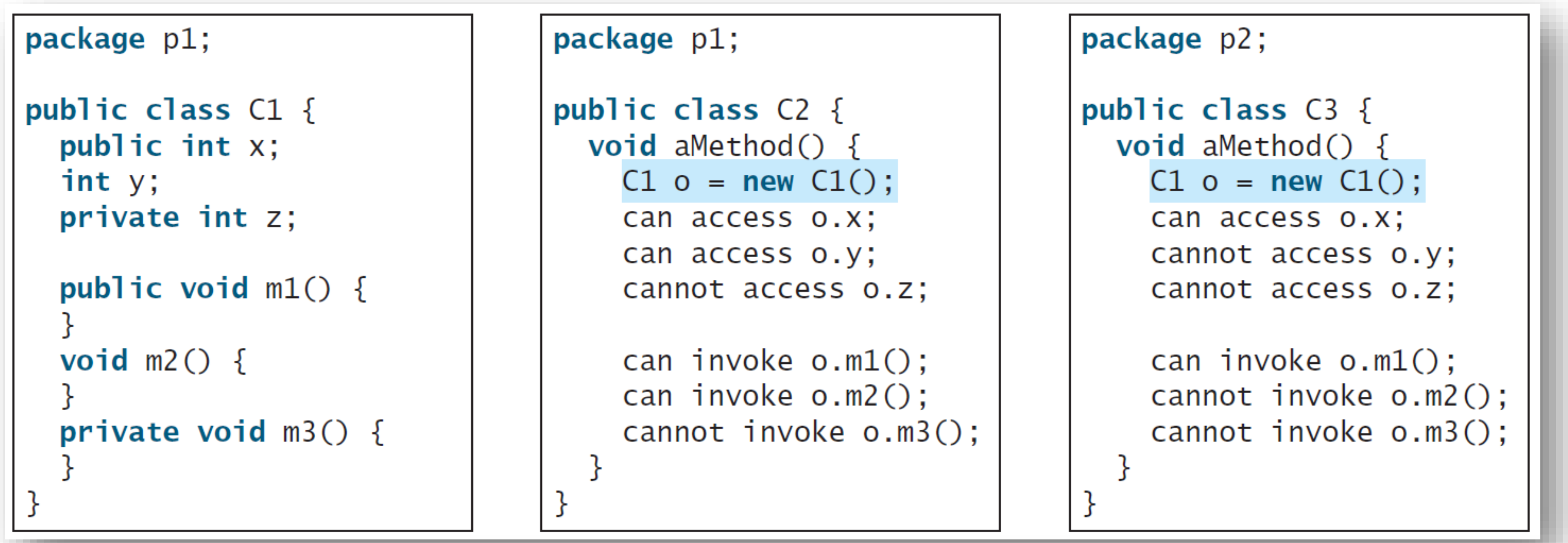
- By **`default`**, the class, variable, or method can be accessed by any class in the same package.
- **`public`**

  The class, data, or method is visible to any class in any package.
- **`private`**

  The data or methods can be accessed only from within its own class.

  The get and set methods are used to read and modify private properties.

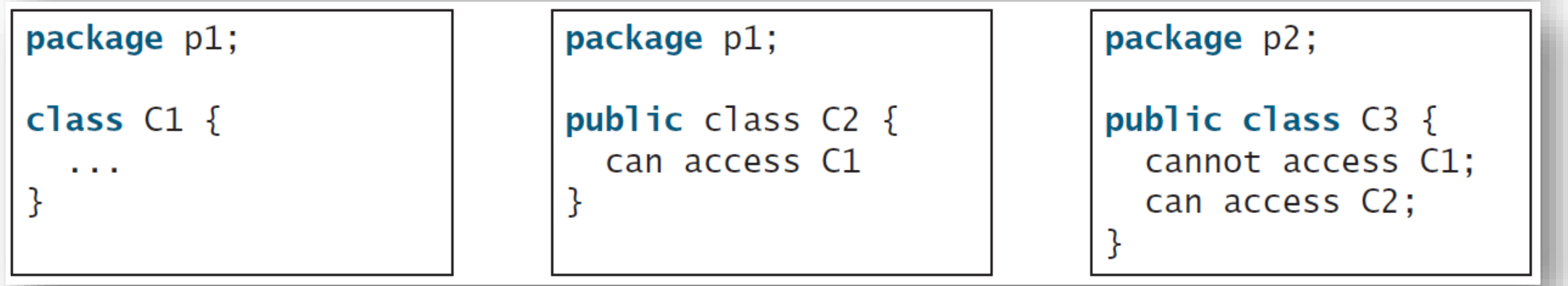
Visibility Modifiers



```
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();
  }
}
```



```
package p1;

class C1 {
  ...
}
```

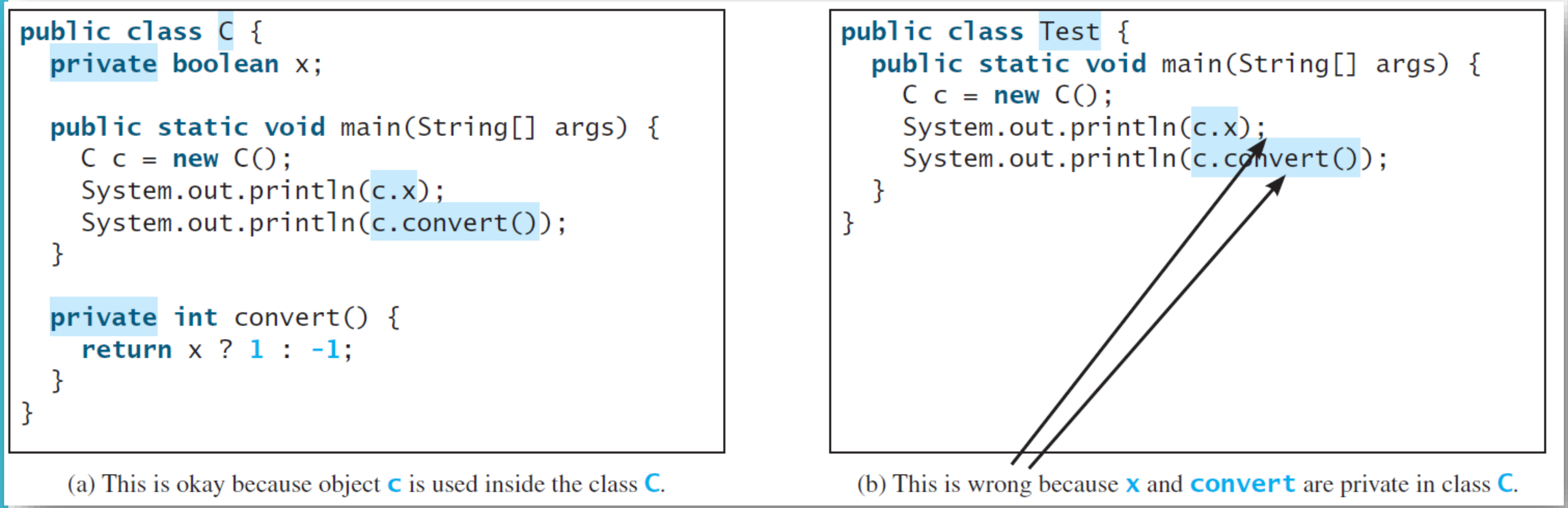
```
package p1;

public class C2 {
  can access C1
}
```

```
package p2;

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

The private modifier restricts access to within a class, the default modifier restricts access to within a package, and the public modifier enables unrestricted access.

Example

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 C {
  private boolean x;

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

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

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

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

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

Caution

- 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.

NOTE

- 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 follows:

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

Data Field Encapsulation

Why data fields should be private?

- To protect data.
- To make code easy to maintain.

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***.

Getter-Setter Methods

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

Getter-Setter Methods

- A **getter method** has the following signature:

```
public returnType getPropertyName()
```

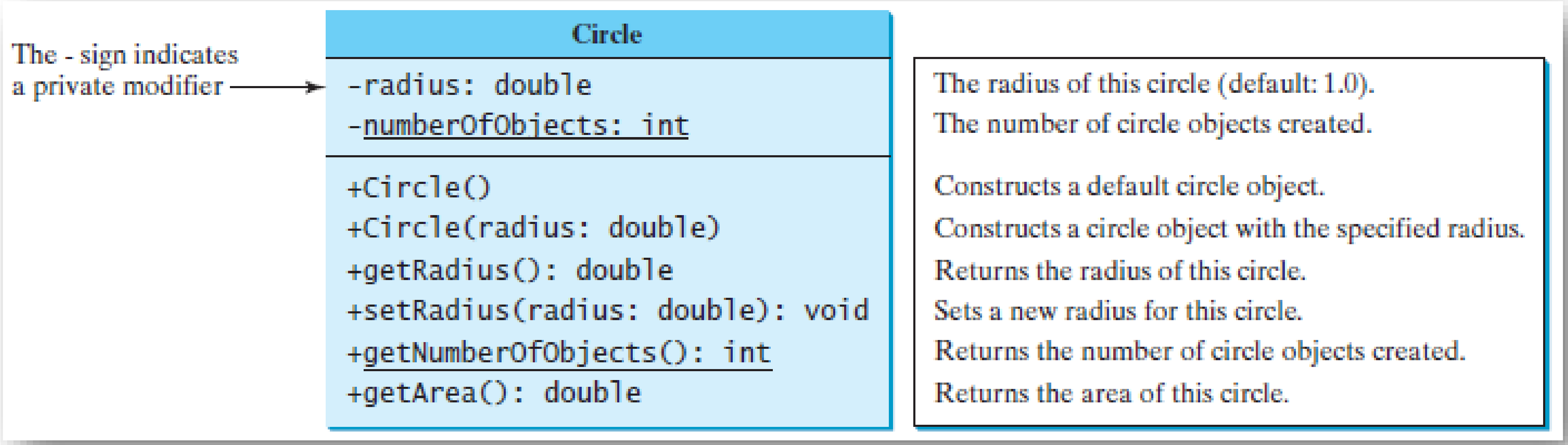
- If the **returnType** is **boolean**, the getter method should be defined as follows by convention:

```
public boolean isPropertyName()
```

- A **setter method** has the following signature:

```
public void setPropertyName(dataType propertyValue)
```

Example Data Field Encapsulation



The - sign indicates a private modifier

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

Circle with Private Data Field

Intro to Java Programming, Y. Daniel Liang - CircleWithPrivateDataFields.java (pearsoncmg.com)

The Client Program

Intro to Java Programming, Y. Daniel Liang - TestCircleWithPrivateDataFields.java (pearsoncmg.com)

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)

```
public class Test {
 public static void main(String[] args) {
  Circle myCircle = new Circle(5.0);
  printCircle(myCircle);
 }
 public static void printCircle(Circle c) {
  System.out.println("The area of the circle of radius " +
  c.getRadius() + " is " + c.getArea());
 }
}
```

Example Passing Objects to Methods

- Obective: Demonstrating the difference between passing a primitive type value and passing a reference value.

Test Passing Objects

Intro to Java Programming, Y. Daniel Liang - TestPassObject.java (pearsoncmg.com)

Array of Objects

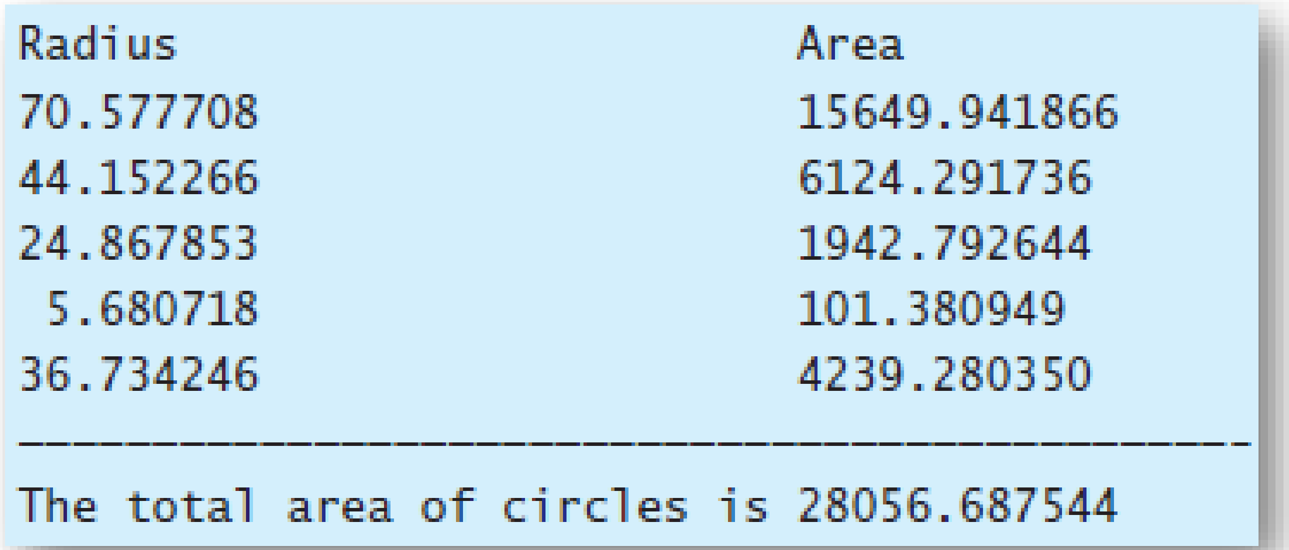
- An array can hold objects as well as primitive type values. For example, the following statement declares and creates an array of ten Circle objects:

```
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: `circleArray` references to the entire array, `circleArray[1]` references to a Circle object.

Example Array of Objects

- Objective: Summarizing the areas of the circles



```
Radius                                   Area
70.577708                                15649.941866
44.152266                                6124.291736
24.867853                                1942.792644
 5.680718                                101.380949
36.734246                                4239.280350
-----------------------------------------------
The total area of circles is 28056.687544
```

Test Area of Circles

Intro to Java Programming, Y. Daniel Liang - TotalArea.java (pearsoncmg.com)

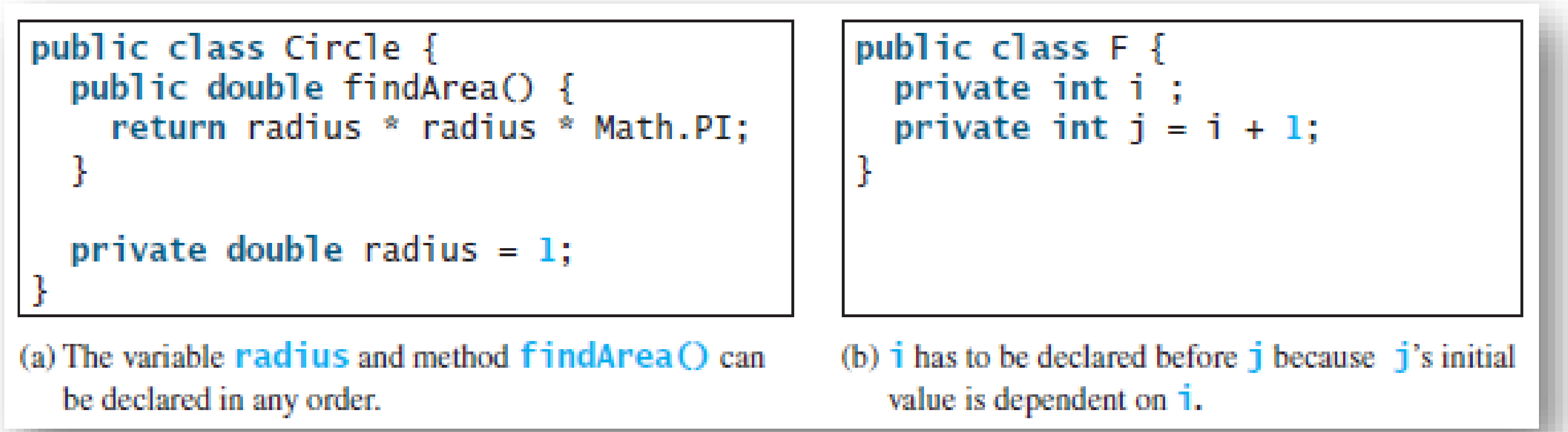
Scope of Variables

- ❑The scope of ***instance and static variables (class's variables or data fields)*** is the entire class. They can be declared anywhere inside a class.

- ❑The scope of a ***local variable*** (defined inside 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

- A class's variables and methods can appear in any order in the class (a).
- The exception is when a data field is initialized based on a reference to another data field. In such cases, the other data field must be declared first (b).



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

  private double radius = 1;
}
```

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

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

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

Caution

1. You can declare a class’s variable only once, but you can declare the same variable name in a method many times in different nonnesting blocks.
2. 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.

Example

```
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);
  }
}
```

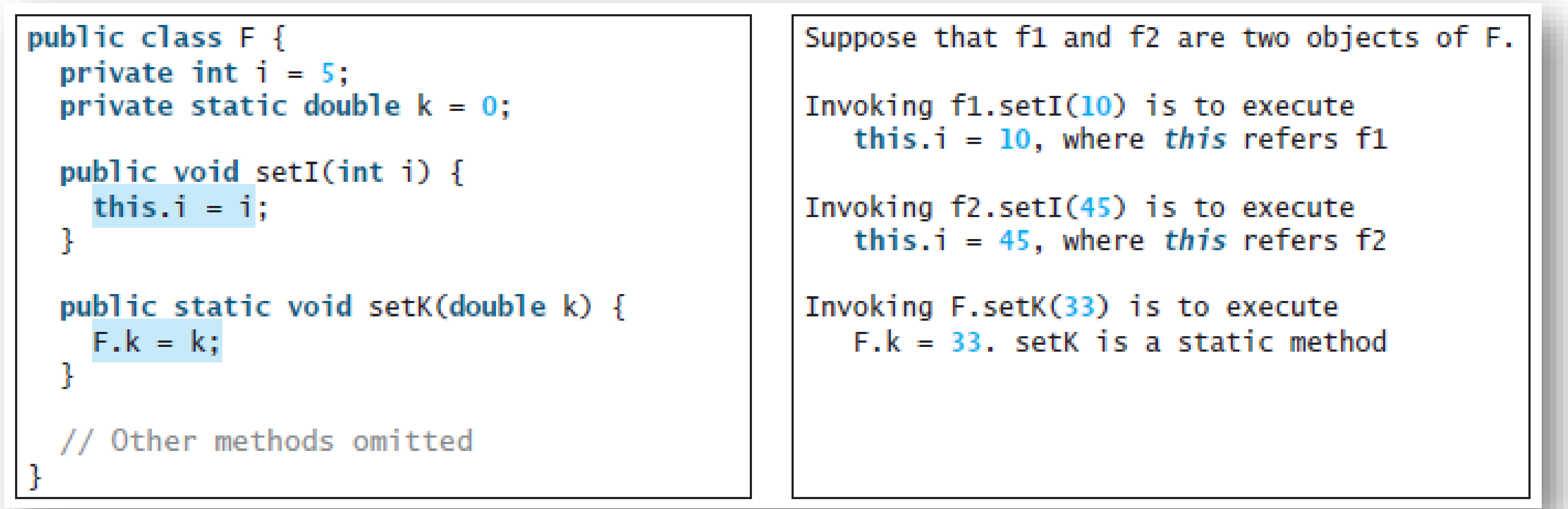
What is the output for **f.p()**, where **f** is an instance of **F**?

The output for **f.p()** is **1** for **x** and **0** for **y**.

The `this` Keyword

- ❑Within an instance method or a constructor, **this** is a reference to the *current object* — the object whose method or constructor is being called.
- ❑One common use of the this keyword is reference a class's *hidden data fields*.
- ❑Another common use of the this keyword is to enable a constructor to invoke another constructor of the same class.

Reference the Hidden Data Fields



```
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
}
```

```
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
```

- ✓ 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**.

- ✓ The line **F.k = k** means that the value in parameter **k** is assigned to the static data field **k** of the class, which is shared by all the objects of the class.

Reference the Hidden Data Fields

For example, the Point class was written like this:

```
public class Point {
    public int x = 0;
    public int y = 0;

    //constructor
    public Point(int a, int b) {
        x = a;
        y = b;
    }
}
```

But it could have been written like this:

```
public class Point {
    public int x = 0;
    public int y = 0;

    //constructor
    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }
}
```

Each argument to the constructor shadows one of the object's fields — inside the constructor **x** is a local copy of the constructor's first argument. To refer to the Point field **x**, the constructor must use **this.x**.

Calling Overloaded Constructor

```
public class Circle {
  private double radius;

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

  public Circle() {
    this(1.0);
  }

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

This must be explicitly used to reference the data field radius of the object being constructed

This is used to invoke another constructor
Must appear before any other statement

Every instance variable belongs to an instance represented by this, which is normally omitted

Calling Overloaded Constructor

```
public class Rectangle {
    private int x, y;
    private int width, height;

    public Rectangle() {
        this(0, 0, 1, 1);
    }
    public Rectangle(int width, int height) {
        this(0, 0, width, height);
    }
    public Rectangle(int x, int y, int width, int height) {
        this.x = x;
        this.y = y;
        this.width = width;
        this.height = height;
    }
    ...
}
```

This class contains a set of constructors. Each constructor initializes some or all of the rectangle's member variables.