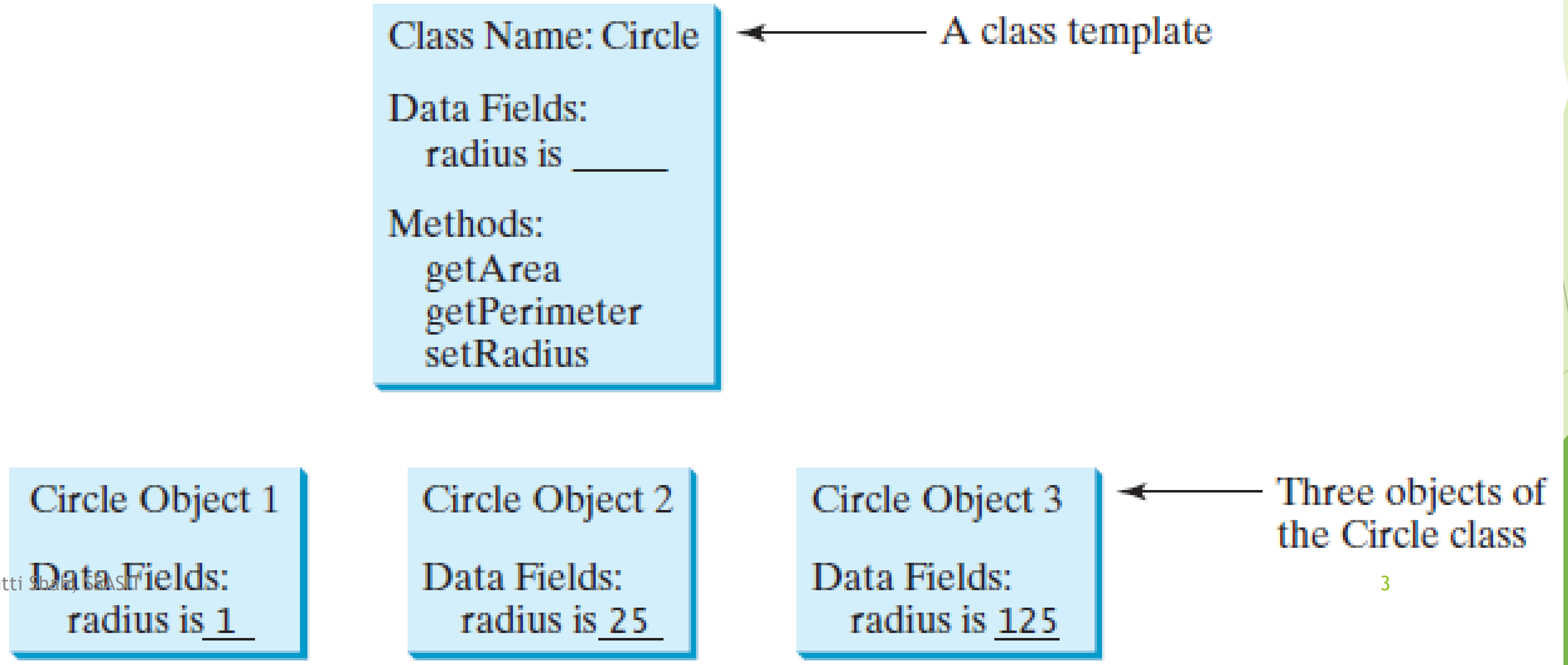
OBJECTS AND CLASSES

Defining Classes for Objects

- *A class defines the properties and behaviors for objects*
- An *object* represents an entity in the real world that can be distinctly identified
- An object has a unique identity, state, and behaviour
- *state* of an object (also known as its *properties* or *attributes*) is represented by *data fields* with their current values
- *behavior* of an object (also known as its *actions*) is defined by methods
- 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

Defining Classes for Objects (Contd…)

- A Java class uses variables to define data fields and methods to define actions
- a class provides methods of a special type, known as *constructors*
  - constructors are designed to initialize the data fields of objects



Class Name: Circle

Data Fields:
radius is _____

Methods:
getArea
getPerimeter
setRadius

A class template

Circle Object 1

Data Fields:
radius is 1

Circle Object 2

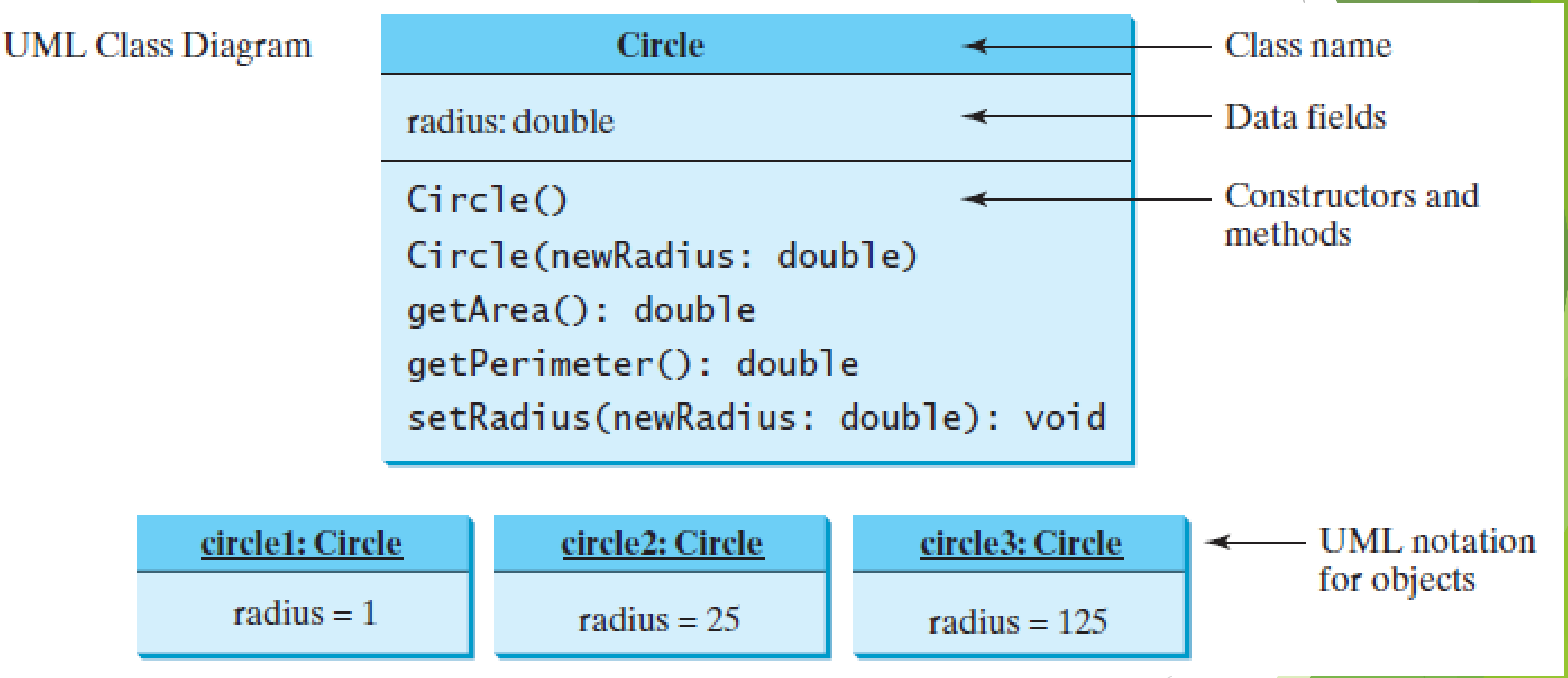
Data Fields:
radius is 25

Circle Object 3

Data Fields:
radius is 125

Three objects of the Circle class

UML Class Diagram



UML Class Diagram

Circle

radius: double

```
Circle()
Circle(newRadius: double)
getArea(): double
getPerimeter(): double
setRadius(newRadius: double): void
```

Constructing Objects Using Constructors

- *A constructor is invoked to create an object using the* **new** *operator*
- three characteristics:
  - A constructor must have the same name as the class itself
  - Constructors do not have a return type—not even **void**
  - Constructors are invoked using the **new** operator when an object is created
- constructors can be overloaded
- Constructors are used to construct objects
  - **new** ClassName(arguments);
- A class normally provides a constructor without arguments, Such a constructor is referred to as a *no-arg* or *no-argument constructor*
- A class may be defined without constructors
- *default constructor* is provided automatically *only if no constructors are explicitly defined in the class*

Accessing Objects via Reference Variables

- *An object's data and methods can be accessed through the dot (.) operator via the object's reference variable*
- Newly created objects are allocated in the memory.
- Object reference variables are declared using the following syntax:
  - ClassName objectRefVar;
  - Circle myCircle;
- The following statement creates an object and assigns its reference to **myCircle**
  - myCircle = **new** Circle();
- declaration of an object reference variable, creation of an object, and assigning of an object reference to the variable
  - ClassName objectRefVar = **new** ClassName();
  - Circle myCircle = **new** Circle();

Accessing an Object’s Data and Methods

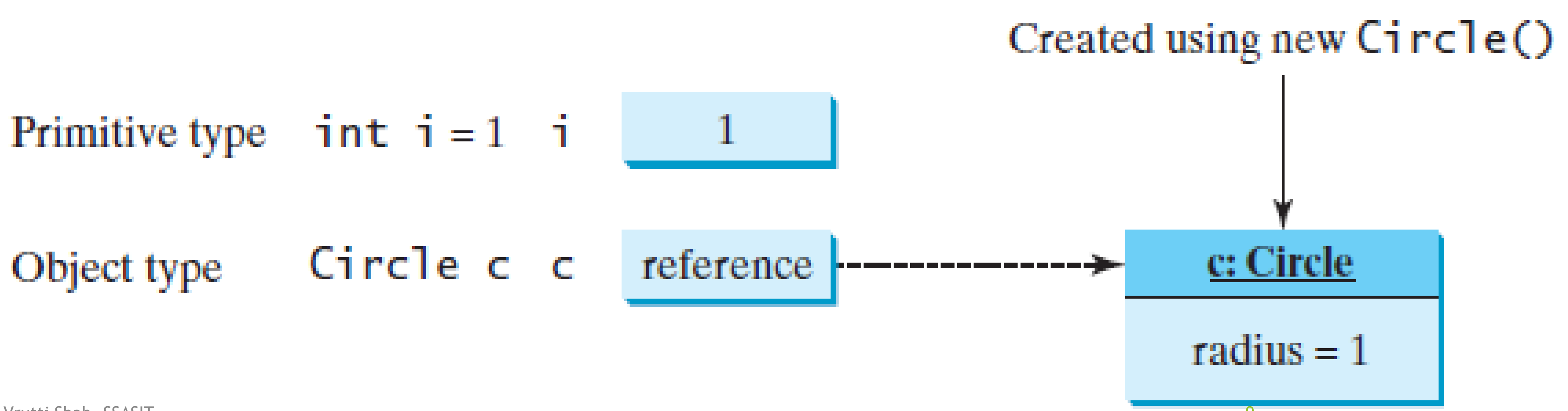
- After an object is created, its data can be accessed and its methods can be invoked using the *dot operator* (.)
  - known as the *object member access operator*
  - objectRefVar.dataField references a data field in the object, e.g. myCircle.radius
  - objectRefVar.method(arguments) invokes a method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

Reference Data Fields and the **null** Value

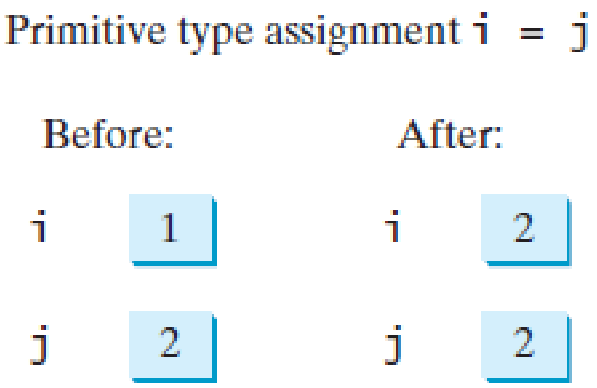
- If a data field of a reference type does not reference any object, the data field holds a special Java value, **null**
- default value of a data field is **null** for a reference type
- **0** for a numeric type, **false** for a **boolean** type, and **\u0000** for a **char** type

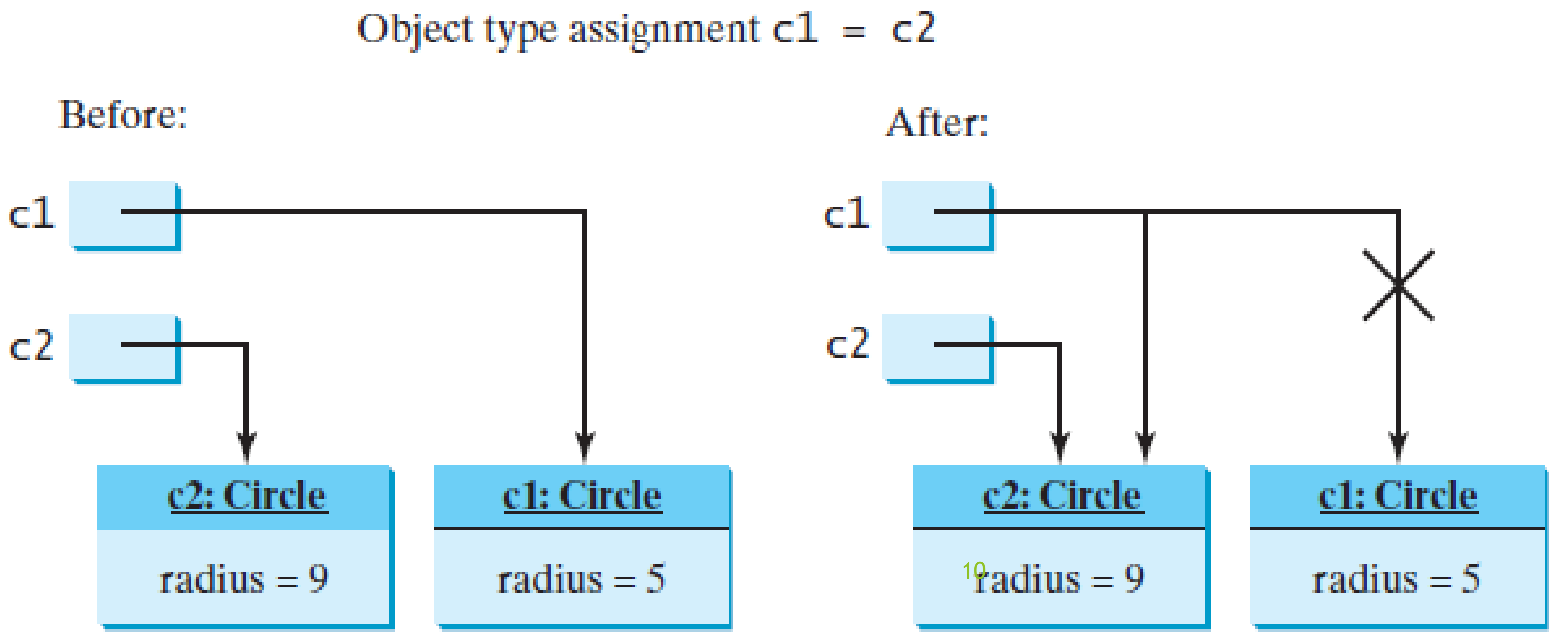
Differences between Variables of Primitive Types and Reference Types

- 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



- 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





Using Classes from the Java Library
The **Date** Class

Sr. No.	Method	Description
1	Date()	Constructs a Date object for the current time
2	Date(elapseTime: long)	Constructs a Date object for a given time in milliseconds elapsed since January 1, 1970,GMT
3	toString(): String	Returns a string representing the date and time
4	getTime(): long	Returns the number of milliseconds since January 1, 1970 GMT
5	setTime(elapseTime: long): void	Sets a new elapse time in the object

The Random Class

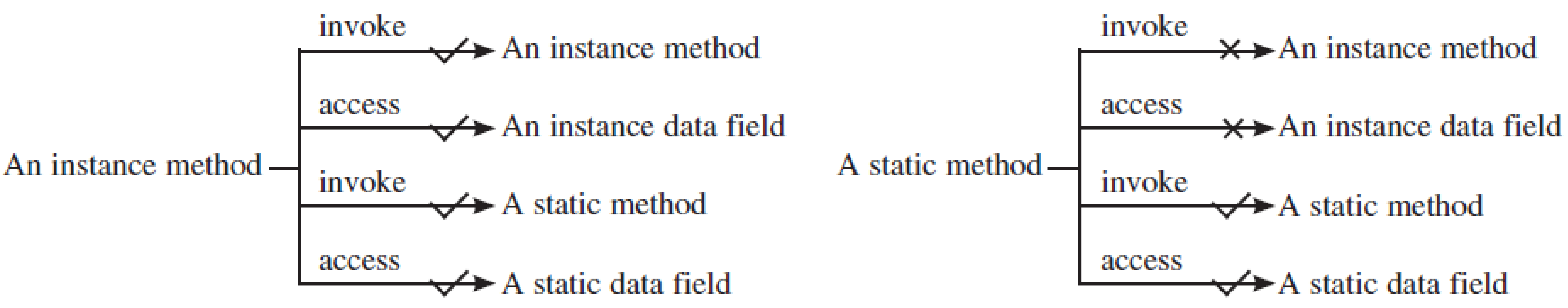
Sr. No.	Method	Description
1	Random()	Constructs a Random object with the current time as its seed
2	Random(seed: long)	Constructs a Random object with a specified seed
3	nextInt(): int	Returns a random int value
4	nextInt(n: int): int	Returns a random int value between 0 and n (excluding n)
5	nextLong(): long	Returns a random long value
6	nextDouble(): double	Returns a random double value between 0.0 and 1.0 (excluding 1.0)
7	nextFloat(): float	Returns a random float value between 0.0F and 1.0F (excluding 1.0F)
8	nextBoolean(): boolean	Returns a random boolean value

The Point2D Class

Sr. No.	Method	Description
1	Point2D(x: double, y: double)	Constructs a Point2D object with the specified *x*- and *y*-coordinates
2	distance(x: double, y: double): double	Returns the distance between this point and the specified point (*x*, *y*)
3	distance(p: Point2D): double	Returns the distance between this point and the specified point p
4	getX(): double	Returns the *x*-coordinate from this point
5	getY(): double	Returns the *y*-coordinate from this point
6	toString(): String	Returns a string representation for the point

Static Variables

- *A static variable is shared by all objects of the class*
- *A static method cannot access instance members of the class*
- data field in the class is known as an *instance variable*
- instance variable is tied to a specific instance of the class, it is not shared among objects of the same class
- If you want all the instances of a class to share data, use *static variables (class variables)*
- Static variables store values for the variables in a common memory location
  - if one object changes the value of a static variable, all objects of the same class are affected
- *Static methods* can be called without creating an instance of the class



An instance method

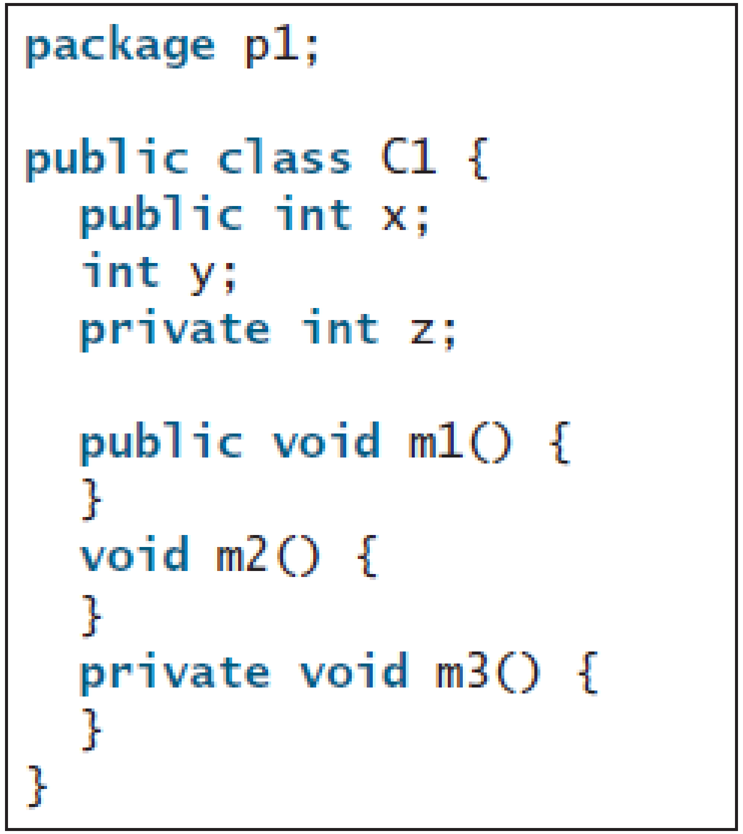
- invoke ✓ An instance method
- access ✓ An instance data field
- invoke ✓ A static method
- access ✓ A static data field

A static method

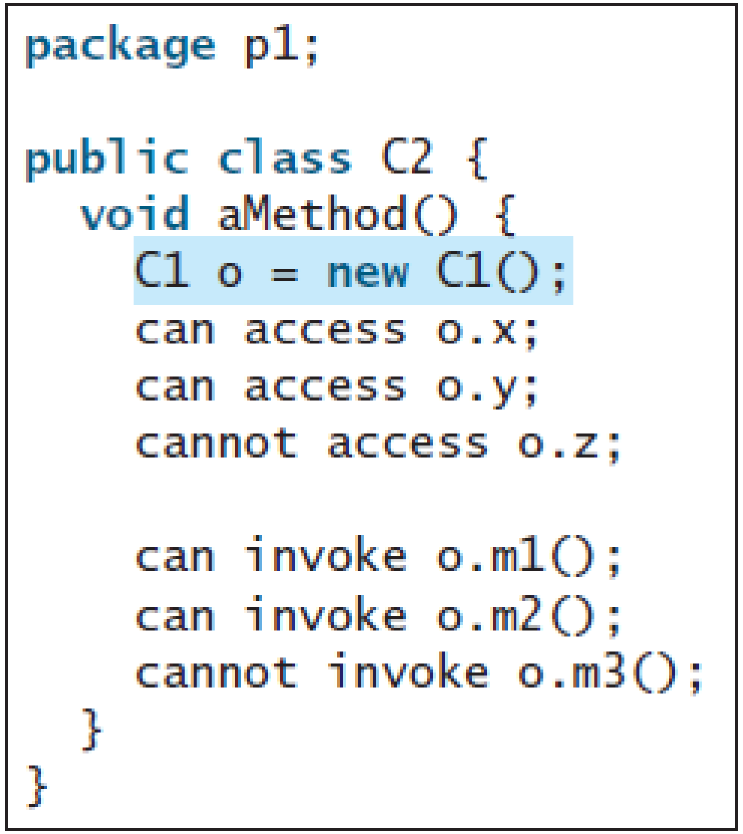
- invoke ✗ An instance method
- access ✗ An instance data field
- invoke ✓ A static method
- access ✓ A static data field

Visibility Modifiers

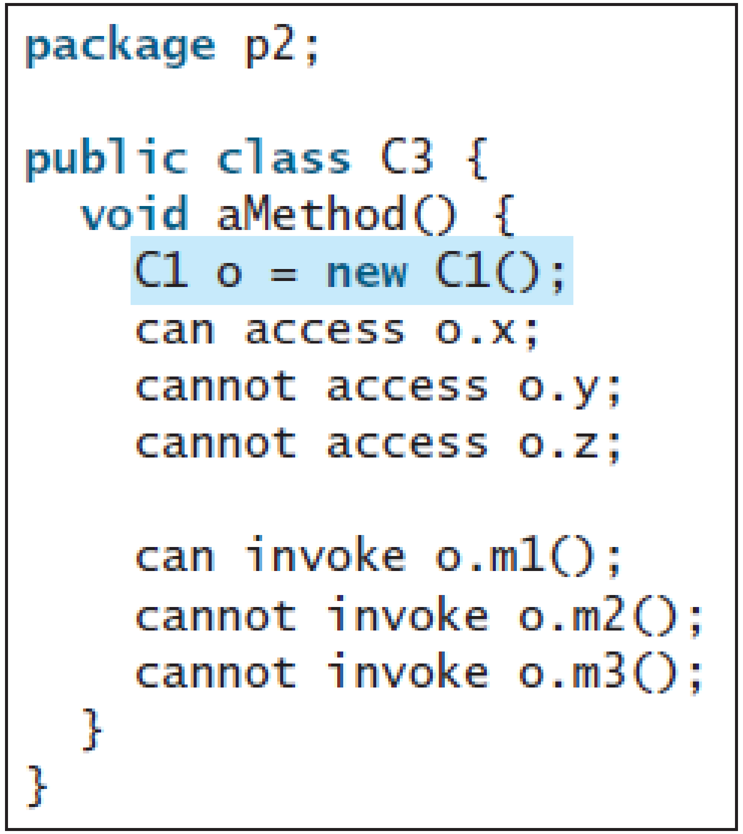
- *Visibility modifiers can be used to specify the visibility of a class and its members*
- **public** visibility modifier for classes, methods, and data fields to denote that they can be accessed from any other classes
- **private** modifier makes methods and data fields accessible only from within its own class
- There is no restriction on accessing data fields and methods from inside the class



```
package p1;

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

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

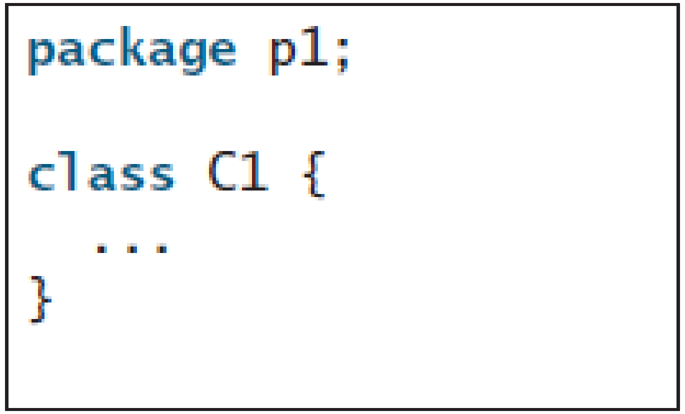


```
package p1;

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();
  }
}
```

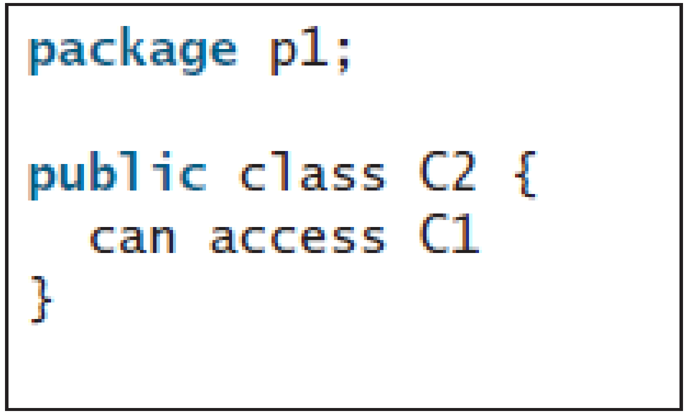


```
package p2;

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();
  }
}
```

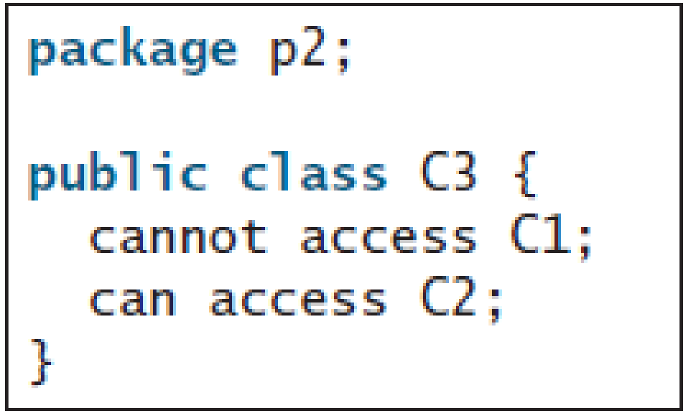
- If a class is not defined as public, it can be accessed only within the same package.



```
package p1;

class C1 {
  ...
}
```

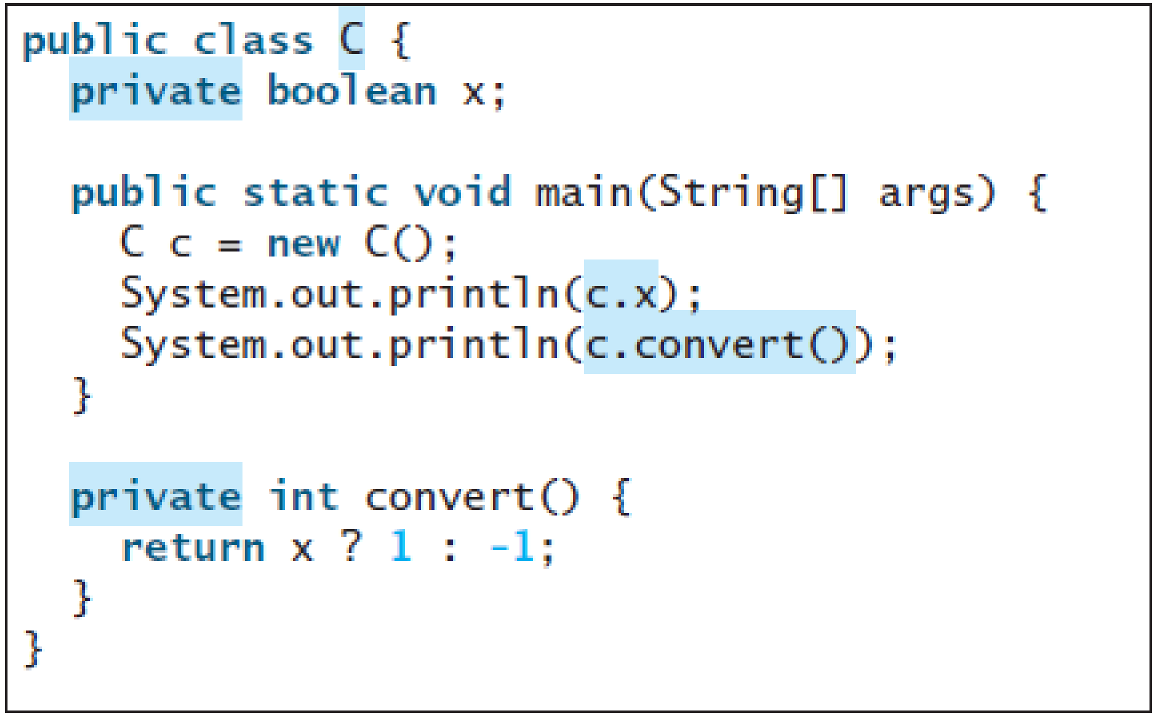


```
package p1;

public class C2 {
  can access C1
}
```

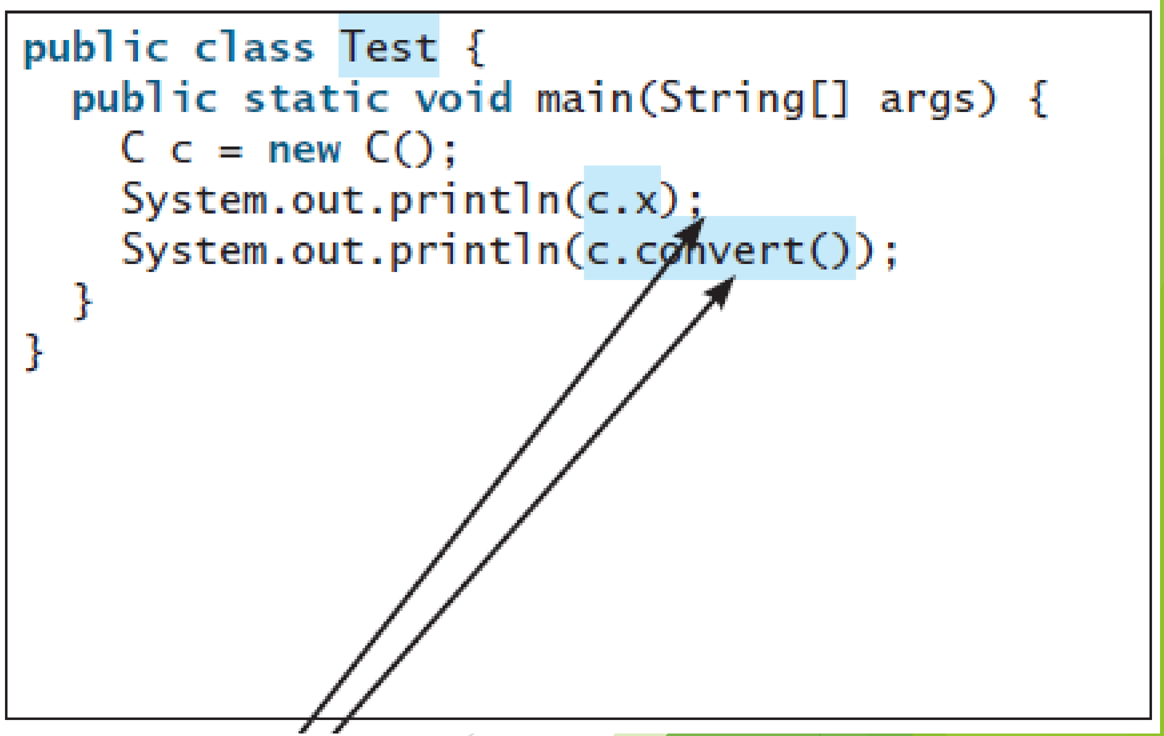


```
package p2;

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

- There is no restriction on accessing data fields and methods from inside the class



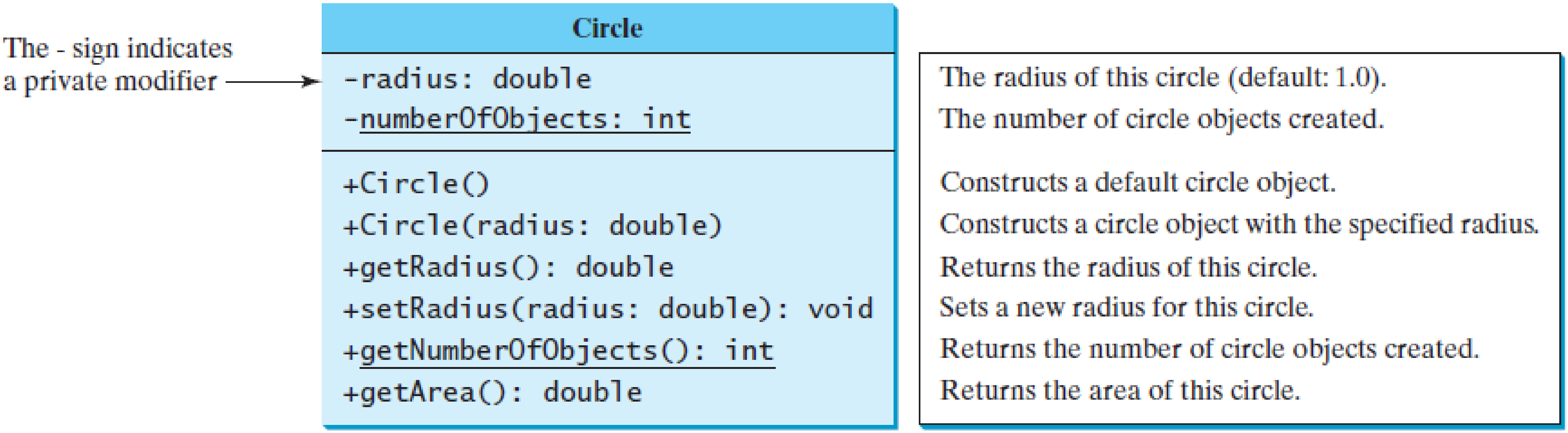
```
public class C {
  private boolean x;

  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;
  }
}
```



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

Data Field Encapsulation

- *Making data fields private protects data and makes the class easy to maintain.*
- If data members are public:
  - First, data may be tampered with
  - class becomes difficult to maintain and vulnerable to bugs
- 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*
- 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



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.

Passing Objects to Methods

- *Passing an object to a method is to pass the reference of the object*
- Like passing an array, passing an object is actually passing the reference of the object

Array of Objects

- *An array can hold objects as well as primitive type values*
- Circle[] circleArray = **new** Circle[**10**];
- To initialize circleArray
  - **for** (**int** i = **0**; i < circleArray.length; i++) {

```
circleArray[i] = new Circle();
}
```

Immutable Objects and Classes

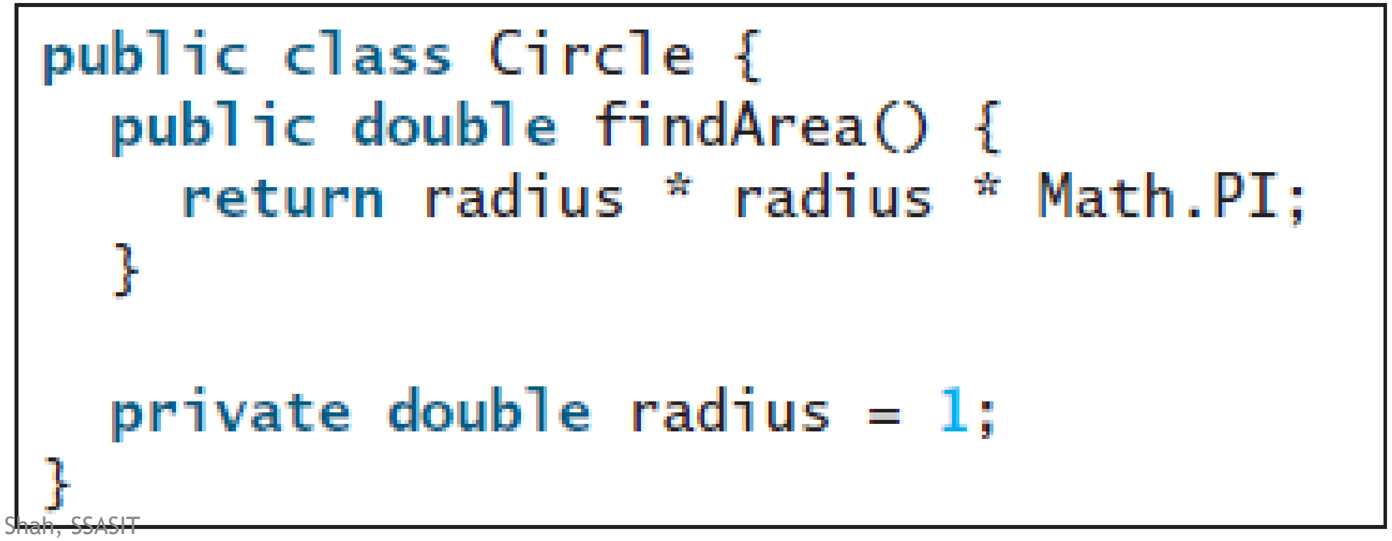
- *You can define immutable classes to create immutable objects*
- *contents of immutable objects cannot be changed*
- object whose contents cannot be changed once the object has been created
- We call such an object as *immutable object* and its class as *immutable class*
- String class, for example, is immutable
- If a class is immutable, then
  - all its data fields must be private
  - it cannot contain public setter methods for any data fields

Immutable Objects and Classes (Contd…)

- For a class to be immutable, it must meet the following requirements:
  - All data fields must be private.
  - There can’t be any mutator methods for data fields.
  - No accessor methods can return a reference to a data field that is mutable.

Scope of Variables

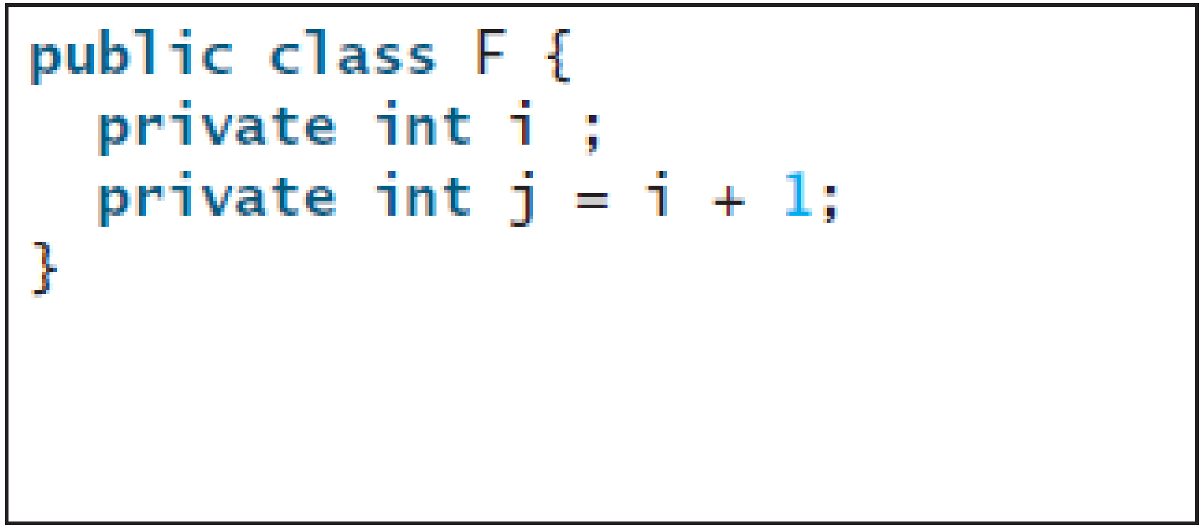
- *scope of instance and static variables is the entire class, regardless of where the variables are declared*
- Instance and static variables in a class are referred to as the *class's variables* or *data fields*
- variable defined inside a method is referred to as a *local variable*
- scope of a class's variables is the entire class
- A class's variables and methods can appear in any order in the class



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

  private double radius = 1;
}
```

Scope of Variables (Contd…)

- when a data field is initialized based on a reference to another data field, the other data field must be declared first



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

Scope of Variables (Contd…)

- declare a class’s variable only once
- you can declare the same variable name in a method many times in different nonnesting blocks
- 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*
-

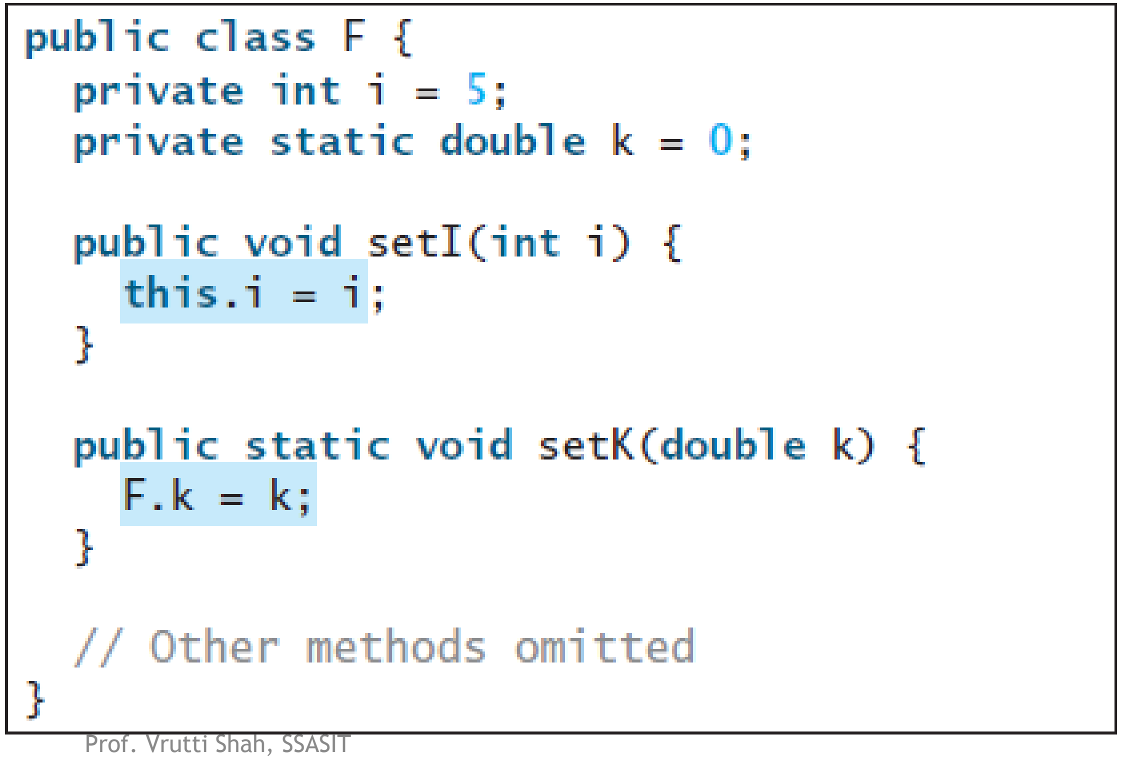
```
public class F {
  private int x = 0; // Instance variable
  private int y = 0;
  public F() {}
  public void p() {
  int x = 1; // Local variable
  System.out.println("x = " + x);
  System.out.println("y = " + y);
  }
  }
```

this Reference

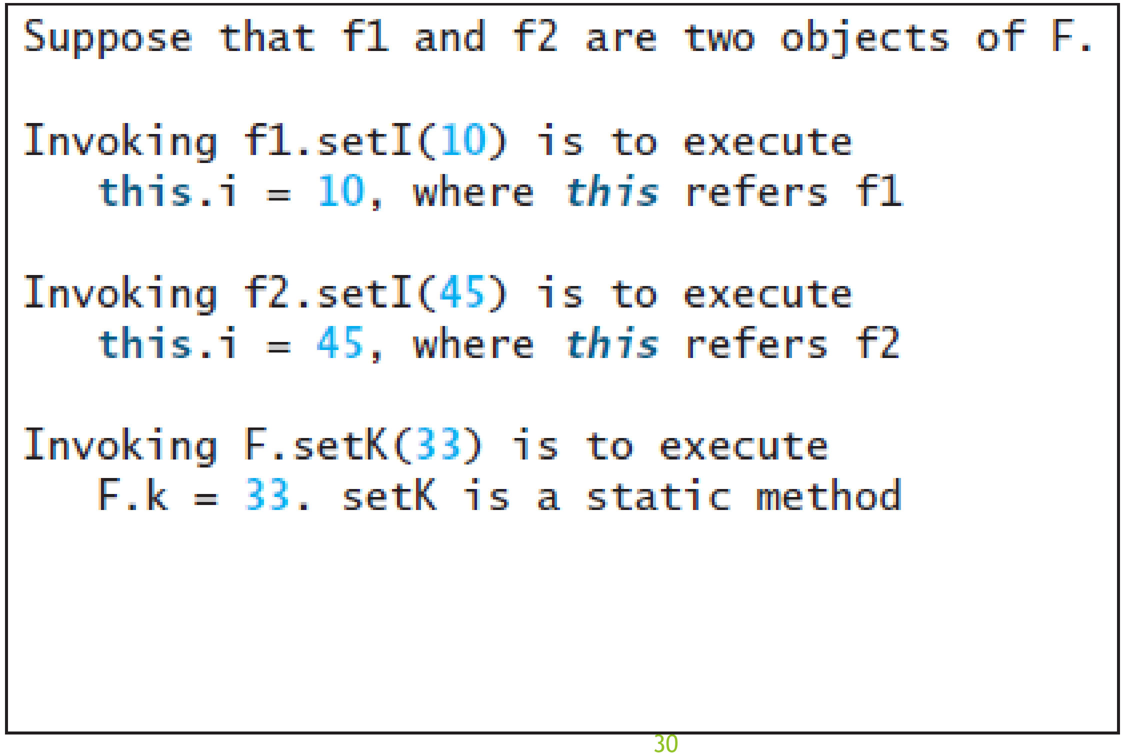
- *keyword* this *refers to the object itself*
- *It can also be used inside a constructor to invoke another constructor of the same class*
- this *keyword* is name of a reference that an object can use to refer to itself
- You can use this keyword to reference the object’s instance members

Using **this** to Reference Hidden Data Fields

- this keyword can be used to reference a class’s *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
```

Using **this** to Invoke a Constructor

- this keyword can be used to invoke another constructor of the same class

```
public class Circle {
  private double radius;
  public Circle(double radius) {
  this.radius = radius;
  }
  public Circle() {
  this(1.0);
  }
}
```