

Objects & Classes

Liang, Introduction to Java Programming, Tenth Edition, (c) 2015 Pearson Education, Inc. All



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.
- ❖ An object has a unique **identity**, **state**, and **behaviors**.
 - The **state** of an object consists of a set of *data fields* (also known as **properties**) with their current values.
 - The **behavior** of an object is defined by a set of **methods**.

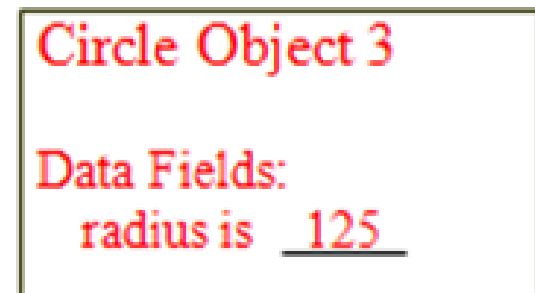
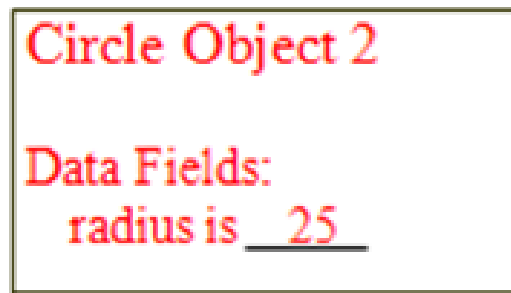
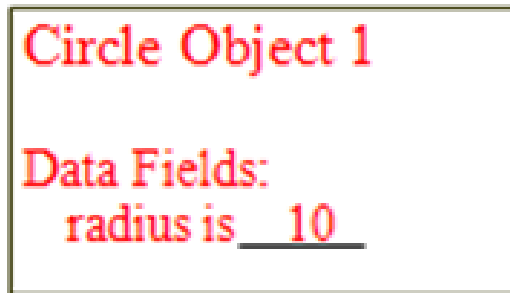
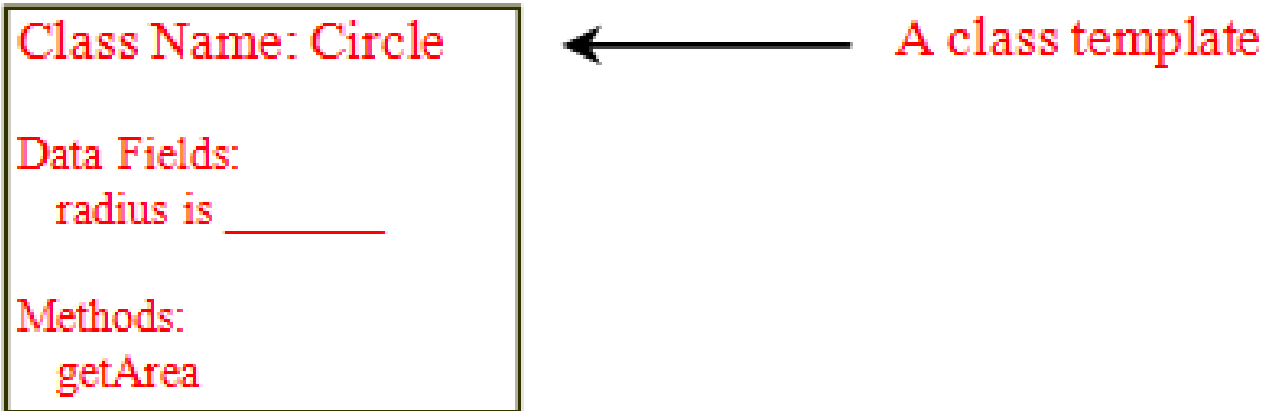


Objects and Classes

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



Objects and Classes cont.



Three objects of
the Circle class



Circle Class

```
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;  
    }  
}
```

← **Data field**

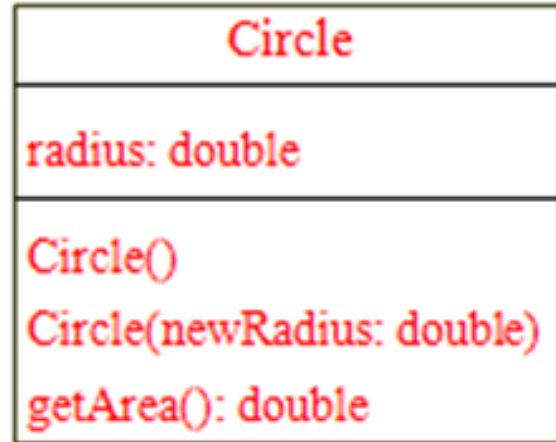
← **Constructors**

← **Method**

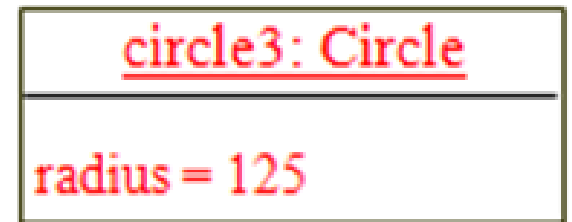
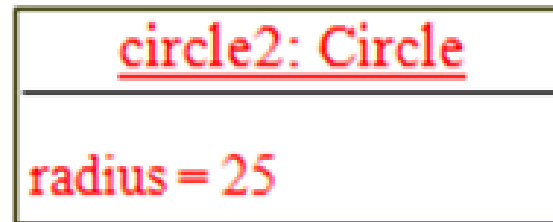
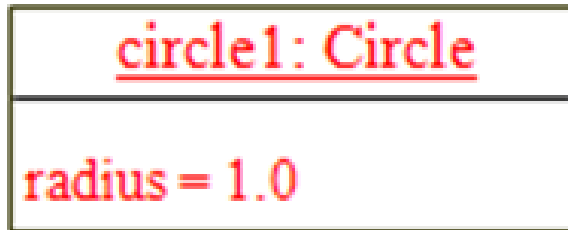


UML Class Diagram

UML Class Diagram



- ← Class name
- ← Data fields
- ← Constructors and methods



UML notation
for objects



Constructors

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

```
Circle() {  
}
```

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



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 maybe defined **without** constructors.
- ❖ In this case, a **no-arg constructor** with an empty body is **implicitly** declared in the class.
- ❖ This constructor, called a **default constructor**, is provided **automatically**

ONLY IF *no constructors are explicitly defined in the class.*



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;

Example:

Circle myCircle;




Declaring/Creating Objects in a Single Step

ClassName objectRefVar = new ClassName();

Example:

Assign object reference Create an object

`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()`**



Reference Data Fields

- ❖ The data fields can be of reference types.
 - If a data field of a **reference** type does not reference any object, the data field holds a special literal value, **null**.
 - For example, the following **Student** class contains a data field **name** of the **String** type.

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



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

'\u0000' for a *char* type

❖ However, **Java assigns NO default value to a local variable inside a method.**



Example

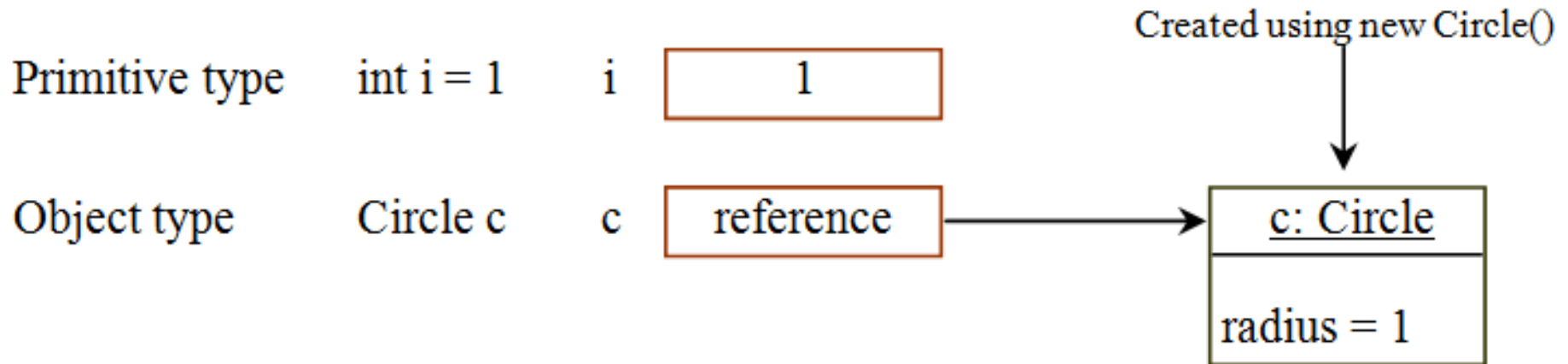
- ❖ Java assigns **no** default value to a local variable inside a method.

```
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);  
    }  
}
```

Compilation error: **variables not initialized**



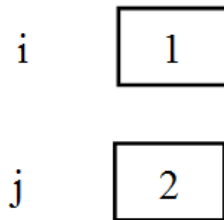
Differences between Variables of Primitive Data Types and Object Types



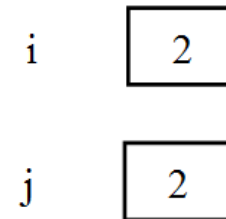
Copying Variables of Primitive Data Types and Object Types

Primitive type assignment $i = j$

Before:

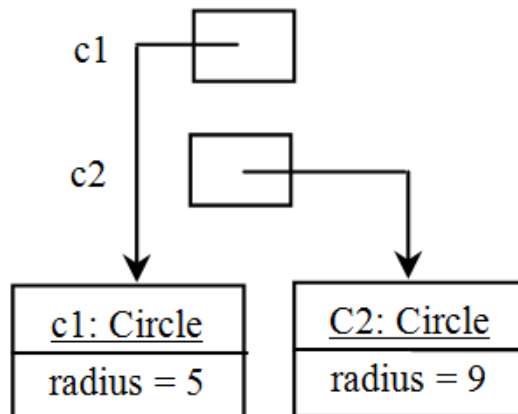


After:

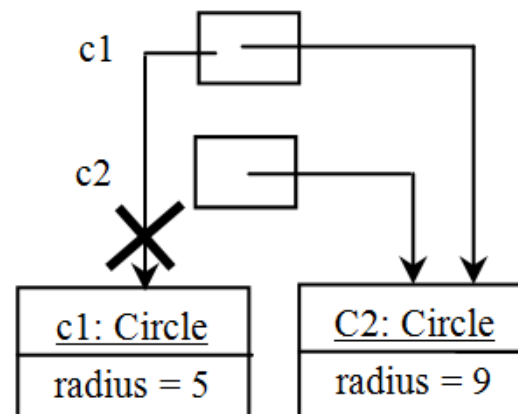


Object type assignment $c1 = c2$

Before:



After:



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 **Date** Class

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

The + sign indicates
public modifier



java.util.Date
+Date()
+Date(elapseTime: long)
+toString(): String
+getTime(): long
+setTime(elapseTime: long): void

Constructs a Date object for the current time.

Constructs a Date object for a given time in milliseconds elapsed since January 1, 1970, GMT.

Returns a string representing the date and time.

Returns the number of milliseconds since January 1, 1970, GMT.

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(date.toString());
```

▪ displays a string like:

Mon Nov 04 19:50:54 IST 2013



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 (exclusive).
+nextLong(): long	Returns a random long value.
+nextDouble(): double	Returns a random double value between 0.0 and 1.0 (exclusive).
+nextFloat(): float	Returns a random float value between 0.0F and 1.0F (exclusive).
+nextBoolean(): boolean	Returns a random boolean value.



Instance Variables, and Methods

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



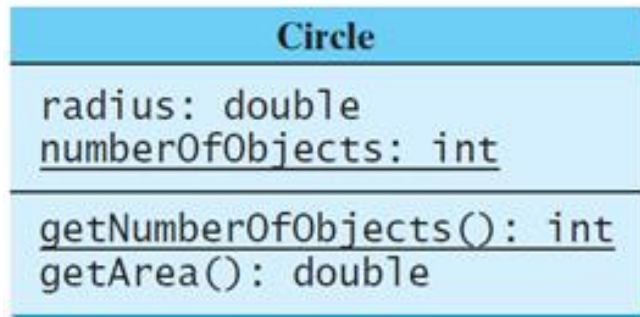
Static Variables, Constants, and Methods

- ❖ **Static variables** 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.



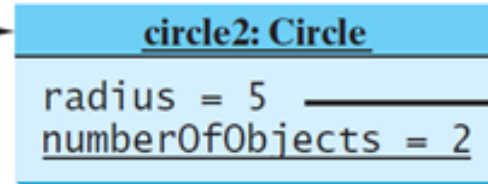
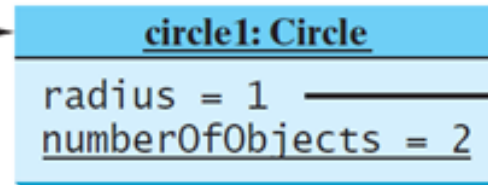
Static

UML Notation:
underline: static
variables or methods



instantiate

instantiate



After two Circle
Objects were created,
numberOfObjects
is 2.

Memory

1

radius

2

numberOfObjects

5

radius



Static Variable

- ❖ It is a variable which belongs to the **class** and not to the **object (instance)**.
- ❖ Static variables are **initialized only once**, at the start of the execution. These variables will be initialized first, before the initialization of any instance variables.
- ❖ A **single copy** to be shared by all instances of the class.
- ❖ A static variable can be **accessed directly** by the **class name** and doesn't need any object.



Syntax : ***<class-name>.<static-variable-name>***

Static Method

- ❖ It is a method which **belongs to the class** and **not** to the **object** (instance).
- ❖ A **static method can access only static data**. It can not access non-static data (instance variables).
- ❖ A **static method can call only other static methods** and can not call a non-static method from it.
- ❖ A static method can be **accessed directly** by the **class name** and doesn't need any object.

Syntax : ***<class-name>.<static-method-name>***

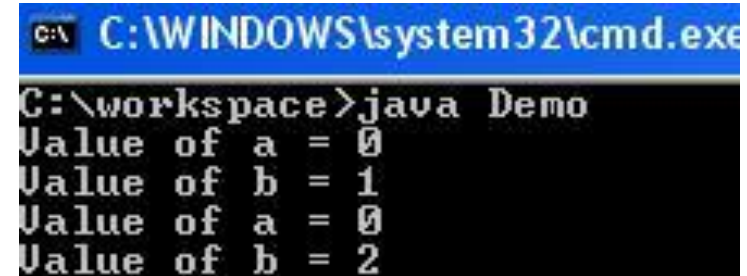
- ❖ A static method cannot refer to “**this**” or “**super**” keywords in anyway.



main method is static, since it must be accessible for an application to run, before any instantiation takes place.

Static example

```
1  class Student {
2  int a; //initialized to zero
3  static int b; //initialized to zero only when class is loaded
4
5  Student(){
6  //Constructor incrementing static variable b
7  b++;
8  }
9
10 public void showData(){
11     System.out.println("Value of a = "+a);
12     System.out.println("Value of b = "+b);
13 }
14 //public static void increment(){
15 //a++;
16 //}
17
18 }
19
20 class Demo{
21     public static void main(String args[]){
22         Student s1 = new Student();
23         s1.showData();
24         Student s2 = new Student();
25         s2.showData();
26         //Student.b++;
27         //s1.showData();
28     }
29 }
```

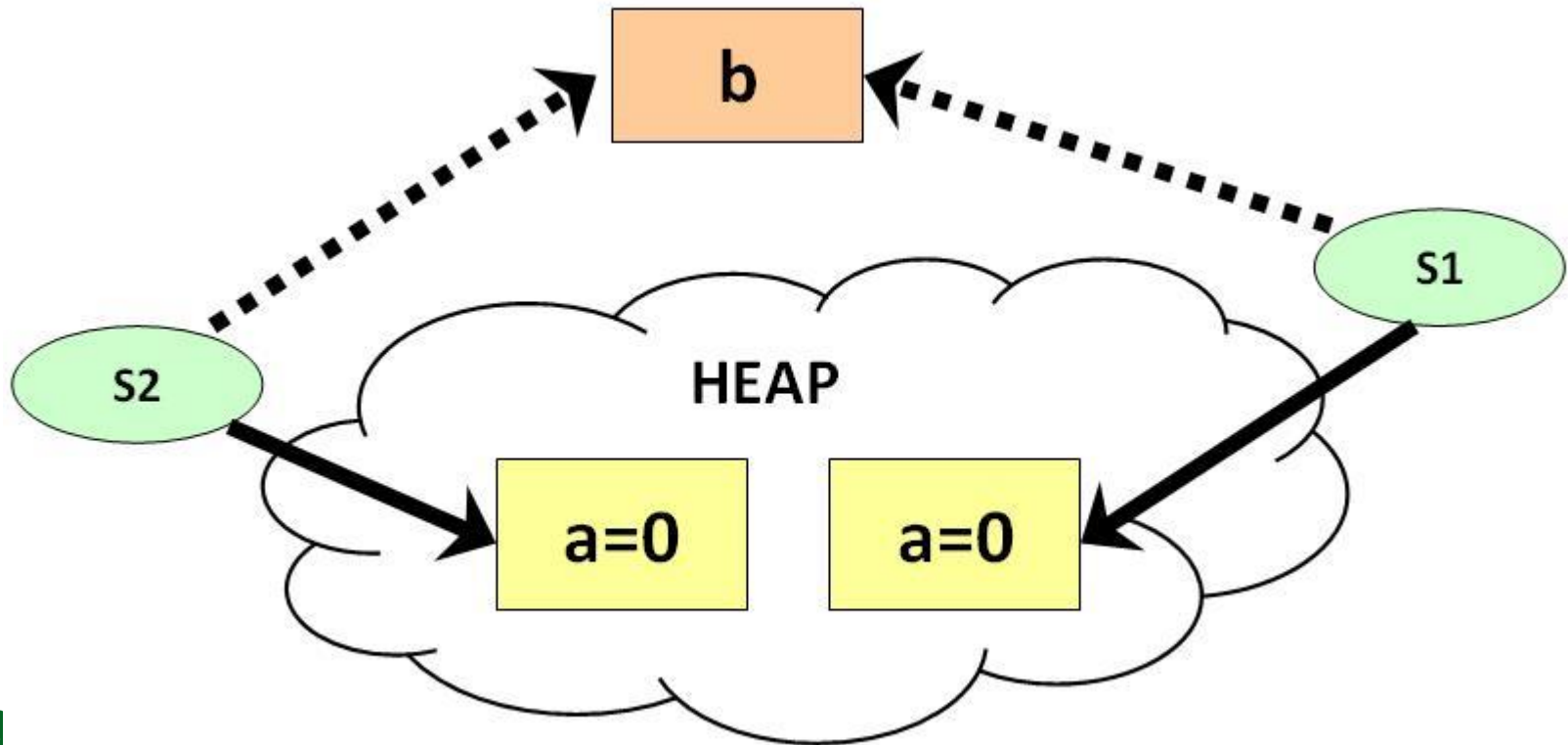


```
C:\WINDOWS\system32\cmd.exe
C:\workspace>java Demo
Value of a = 0
Value of b = 1
Value of a = 0
Value of b = 2
```



Static example cont.

❖ Following diagram shows , how reference variables & objects are created and static variables are accessed by the different instances.



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 by the declaring class.
- ❖ The **get** and **set** methods are used to read and modify private properties.



```

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**.



The **public** modifier enables **unrestricted access**.

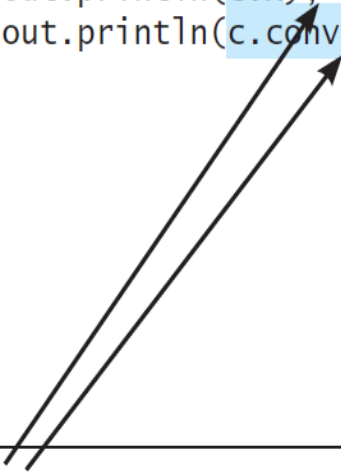
NOTE

❖ 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).

```
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`.

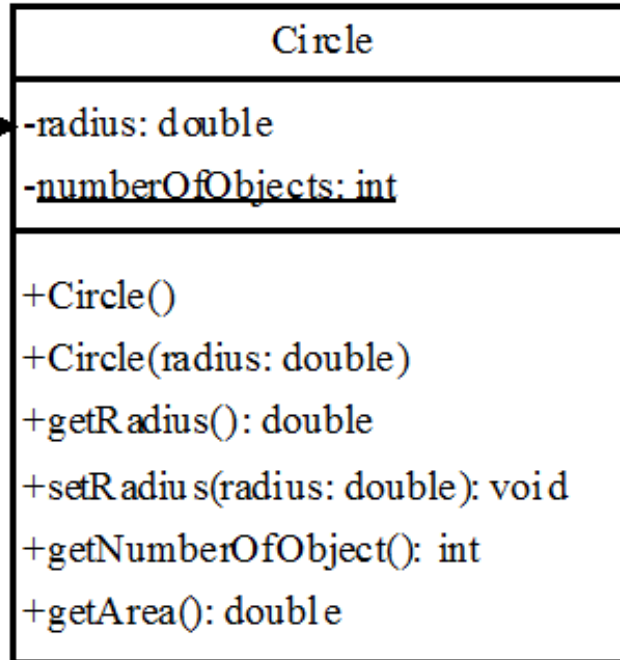
```
public class Test {  
    public static void main(String[] args) {  
        C c = new C();  
        System.out.println(c.x);  
        System.out.println(c.convert());  
    }  
}
```



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



Example of Data Field Encapsulation



The - sign indicates **private** modifier →

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.



Overloading Methods and Constructors

- ❖ In a class, there can be **several methods with the same name**. However they **must** have **different signature**.
- ❖ The signature of a method is comprised of its ***name***, its ***parameter types*** and the ***order of its parameter***.
- ❖ The signature of a method is **not** comprised of its ***return type*** nor ***its visibility*** nor its ***thrown exceptions***.



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 TestPassObject {
    public static void main(String[] args) {
        Circle myCircle = new Circle(1);
        // Print areas for radius 1, 2, 3, 4, and 5.
        int n = 5;
        printAreas(myCircle, n);
        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( Circle 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--;
        }
    }
}
```



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.

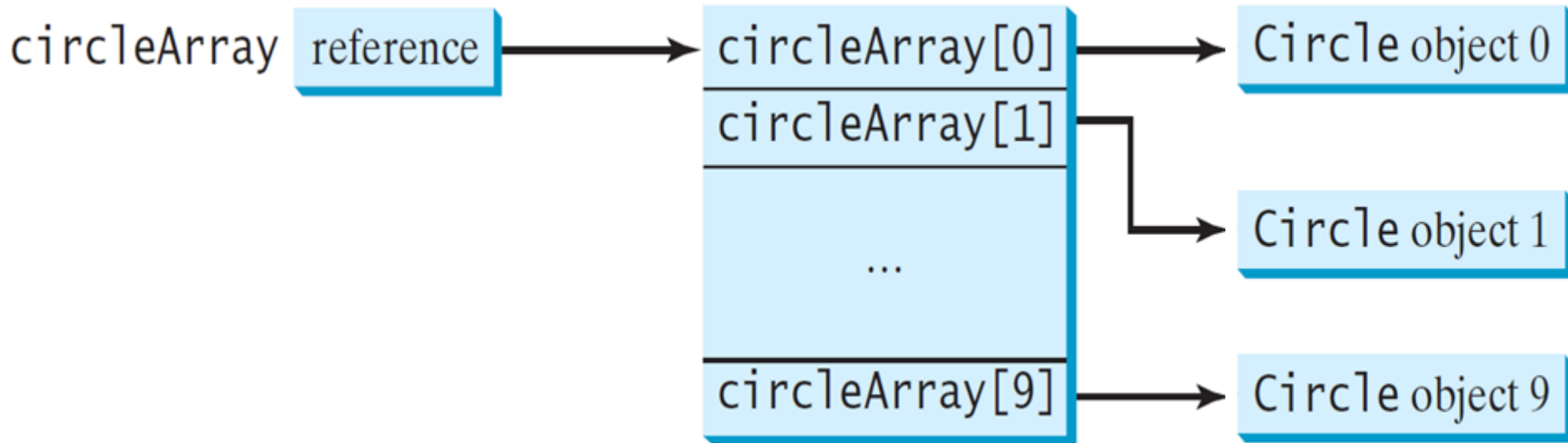
circleArray references to the entire array.

circleArray[1] references to a Circle object.



Array of Objects

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



```
circleArray[0] = new Circle();  
circleArray[1] = new Circle();  
:  
circleArray[9] = new Circle();
```



Immutable Objects and Classes

❖ If the contents of an object (instance) **can't** be changed once the object is created, the object is called an ***immutable object*** and its class is called an ***immutable class***.



Immutable Objects and Classes

❖ If you delete the **set** method in the **Circle** class, the class would be **immutable** because **radius** is private and cannot be changed without a **set** method.

```
public class Circle {  
    private double radius = 1;  
  
    public double getArea() {  
        return radius * radius * Math.PI;  
    }  
  
    public void setRadius(double r) {  
        radius = r;  
    }  
}
```



Immutable Objects and Classes

❖ A class with all **private** data fields and without **mutators** is not necessarily immutable.

❖ For example, the following class **Student** has all **private** data fields and no **mutators**, but it is mutable!!!



Example

```
import java.util.Date;
public class Student {
    private int id;
    private Date birthDate;

    public Student(int ssn, Date newBD) {
        id = ssn;
        birthDate = newBD;
    }

    public int getId() { return id; }

    public Date getBirthDate() { return birthDate; }
}
```

```
public class Test {
    public static void main(String[] args) {
        java.util.Date bd = new java.util.Date();
        Student student = new Student(111223333, bd);
        java.util.Date date = student.getBirthDate();
        date.setMonth(5); // Now the student birthdate is changed!
    }
}
```



What Class is **Immutable**?

- ❖ For a class to be immutable:
 - It must mark all data fields **private**.
 - Provide **no mutator** methods.
 - No accessor methods that would return a reference to a mutable data field object.



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 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

❖ What is the output?

```
public class A{  
    int year = 2014; // instance variable  
  
    void p() {  
        System.out.println("Year: " + year);  
        int year = 2015; // local variable  
        System.out.println("Year: " + year);  
    }  
}
```



The **this** Keyword

- ❖ The **this** keyword is the name of a **reference** that refers to an **object itself**.
- ❖ One common use of the **this** keyword is reference a class's *hidden data fields*.
- ❖ Another common use of the **this** keyword 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;  
  
    void setI(int i) {  
        this.i = i;  
    }  
  
    static void setK(double k) {  
        F.k = k;  
    }  
}
```

Suppose that `f1` and `f2` are two objects of `F`.
`F f1 = new F(); F f2 = new 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`



Calling **Overloaded** Constructor

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

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

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

```
    public Circle() {  
        this(1.0);  
    }
```

this is used to invoke another constructor

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

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

