BIRZEIT UNIVERSITY

COMPUTER SCIENCE DEPARTMENT FACULTY OF ENGINEERING AND TECHNOLOGY

ADVANCED PROGRAMMING COMP231

Lecturer :Farid Mohammad

Object Oriented

Constructing Objects Using Constructors

<table>
<tr>
<td>

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

■ **A constructor must have the same name as the class itself.**

■ **Constructors do not have a return type—not even void.**

■ **Constructors are invoked using the new operator when an object is created.**

Constructors play the role of initializing objects.

</td>
<td>

// Create a Main class

```
public class Main {

  int x;  // Create a class attribute



  // Create a class constructor for the Main class

  public Main() {

    x = 5;  // Set the initial value for the class attribute x

  }



  public static void main(String[] args) {

    Main myObj = new Main(); // Create an object of class Main (This will call the constructor)

    System.out.println(myObj.x); // Print the value of x

  }

}
```

// Outputs 5

</td>
</tr>
<tr>
<td>

Also note that the constructor is called when the object is created.

</td>
<td></td>
</tr>
</table>

All classes have constructors by default: if you do not create a class constructor yourself, Java creates one for you. **However, then you are not able to set initial values for object attributes.**	

Constructor Parameters

Constructors can also take parameters, which is used to initialize attributes.

The following example adds an `int y` parameter to the constructor. Inside the constructor we set x to y (x=y).

When we call the constructor, we pass a parameter to the constructor (5), which will set the value of x to 5:

```
public class Main {
  int x;

  public Main(int y) {
    x = y;
  }

  public static void main(String[] args) {
    Main myObj = new Main(5);
    System.out.println(myObj.x);
  }
}
```

// Outputs 5

You can have as many parameters as you want:

```
public class Main {
  int modelYear;
  String modelName;
```

<pre> public Main(int year, String name) { modelYear = year; modelName = name; } public static void main(String[] args) { Main myCar = new Main(1969, "Mustang"); System.out.println(myCar.modelYear + " " + myCar.modelName); } }</pre>	

9.5 What are the differences between constructors and methods?

9.6 When will a class have a default constructor?

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.** **They can be accessed via reference variables.**	
Reference Variables and Reference Types **Objects are accessed via the object's *reference variables*,** **which contain references to the objects.** **Such variables are declared using the following syntax:** **ClassName objectRefVar;** **example to declare a reference:** **Circle myCircle;**	

to create the object:

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

Accessing an Object’s Data and Methods

In OOP terminology, an object’s member refers to its data fields and methods.

After an object is created, its data can be accessed and its methods can be invoked using the *dot operator* (`.`),

also known as the *object member access operator*:

- `objectRefVar.dataField` **references a data field in the object.**

- `objectRefVar.method(arguments)` **invokes a method on the object.**

For example, `myCircle.radius` **references the radius in** `myCircle`**, and** `myCircle`

`.getArea()` **invokes the** `getArea` **method on** `myCircle`**. Methods are invoked as operations**

on objects.

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

Differences between Variables of Primitive Types and Reference Types

Every variable represents a memory location that holds a value.

When you declare a variable,

you are telling the compiler what type of value the variable can hold.

For a variable of a primitive type, the value is of the primitive type.

For a variable of a reference type, the value is a reference to where an object is located.

For example, the value of int variable i is int value 1,

and the value of Circle object c holds a reference to where the contents of the Circle object are stored in memory.

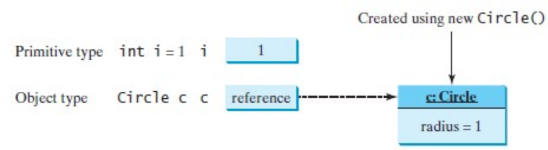


FIGURE 9.7 A variable of a primitive type holds a value of the primitive type, and a variable of a reference type holds a reference to where an object is stored in memory.

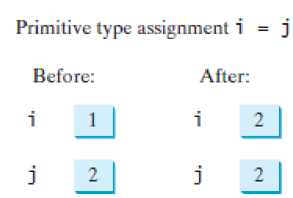


FIGURE 9.8 Primitive variable j is copied to variable i.

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. As shown in Figure 9.8, the assignment statement i = j copies the contents of j into i	

9.7 Which operator is used to access a data field or invoke a method from an object?

9.8 What is an anonymous object?

9.9 What is `NullPointerException`?

9.10 Is an array an object or a primitive type value? Can an array contain elements of an

object type? Describe the default value for the elements of an array.

9.11 What is wrong with each of the following programs?

```
public class ShowErrors {
  public static void main(String[] args) {
    ShowErrors t = new ShowErrors(5);
  }
}
```

(a)

```
public class ShowErrors {
  public static void main(String[] args) {
    ShowErrors t = new ShowErrors();
    t.x();
  }
}
```

(b)

```
public class ShowErrors {
  public void method1() {
    Circle c;
    System.out.println("What is radius "
      + c.getRadius());
    c = new Circle();
  }
}
```

(c)

```
public class ShowErrors {
  public static void main(String[] args) {
    C c = new C(5.0);
    System.out.println(c.value);
  }
}

class C {
  int value = 2;
}
```

(d)

Static Variables, Constants, and Methods

A static variable is shared by all objects of the class. *A static method cannot access instance members of the class.* *class Circle{* *float radius;* *Circle(float r){* *radius=r;* *}* *}* `Circle circle1 = new Circle();` `Circle circle2 = new Circle(5);` `Whats the value of  circle1.radius`	*class Circle{* *static float radius;* *Circle(float r){* *radius=r;* *}* *}* `Circle circle1 = new Circle();` `Circle circle2 = new Circle(5);` `Whats the value of  circle1.radius`

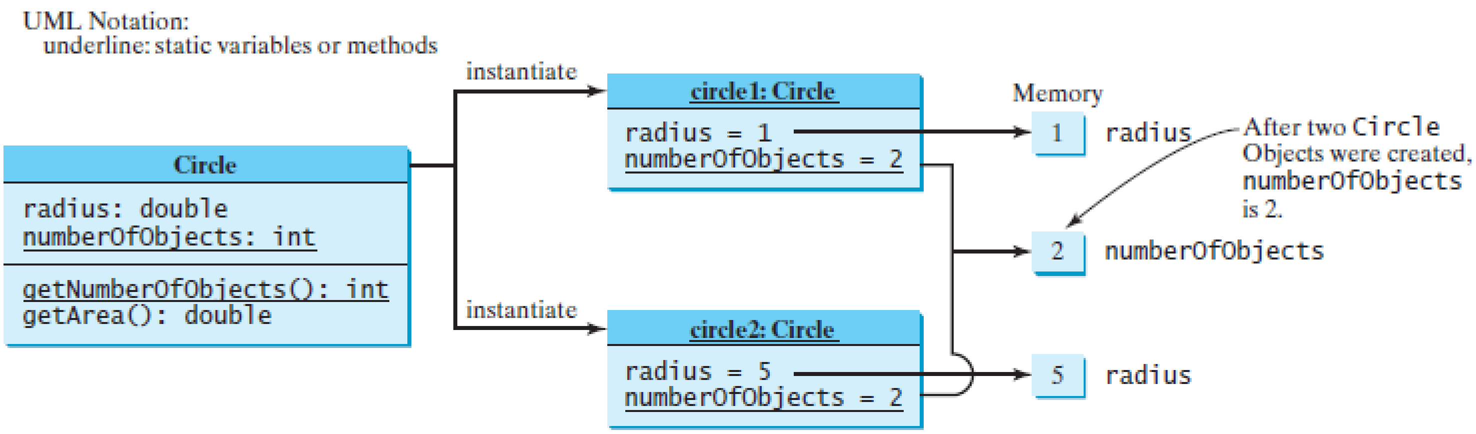


FIGURE 9.13 Instance variables belong to the instances and have memory storage independent of one another. Static variables are shared by all the instances of the same class.

How to know number of objects created

```
public class Circle {
/** The radius of the circle */
double radius;

/** The number of objects created */
static int numberOfObjects = 0;

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

/** Return numberOfObjects */
static int getNumberOfObjects() {
    return numberOfObjects;
}

}
```

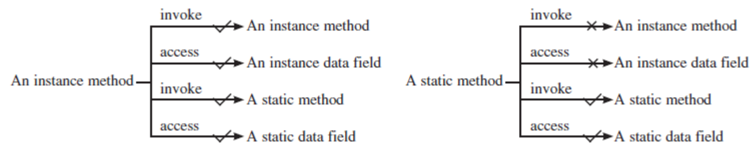
```
static int numberOfObjects = 0;
```

Static variable: only one place in memory for all objects

```
static int getNumberOfObjects() {
```

Static method
No object needed
Circle.getNumberOfObject

```
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 " +
 Circle.numberOfObjects);

 // Create c1
 Circle c1 = new Circle();

// 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
Circle c2 = new Circle(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 + ")");
}
}
```




```
Before creating objects
The number of Circle objects is 0
After creating c1
c1: radius (1.0) and number of Circle objects (1)
After creating c2 and modifying c1
c1: radius (9.0) and number of Circle objects (2)
c2: radius (5.0) and number of Circle objects (2)
```

D

Design Guide

How do you decide whether a variable or a method should be an **instance** one or a **static** one?

A variable or a method that is dependent on a specific instance of the class should be an instance variable or method.

A variable or a method that is not dependent on a specific instance of the class should be a static variable or method.

For example, **every circle has its own radius**, so the radius is dependent on a specific circle. Therefore, **radius** is an instance variable of the **Circle** class.

Since the **getArea** method is dependent on a specific circle, it is an instance method.

None of the methods in the **Math** class, such as **random**, **pow**, **sin**, and **cos**, is dependent on a specific instance. Therefore, these methods are static methods.

The **main** method is static and can be invoked directly from a class.

Caution

It is a common design error to define an instance method that should have been defined as static. For example, the method **factorial(int n)** should be defined as static, as shown next, because it is **independent** of any specific instance.

```java
public class Test {
  public int factorial(int n) {
    int result = 1;
    for (int i = 1; i <= n; i ++)
      result *= i;

    return result;
  }
}
```

(a) Wrong design

```java
public class Test {
  public static int factorial(int n) {
    int result = 1;
    for (int i = 1; i <= n; i++)
      result *= i;

    return result;
  }
}
```

(b) Correct design

Visibility Modifiers

Visibility modifiers can be used to specify the visibility of a class and its members. **1- You can use the public visibility modifier for classes, methods, and data fields to denote that they can be accessed from any other classes.** **2- 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*.**	
3- Private: **Methods and data fields accessible only from within its own class.** **4- The protected modifier will be introduced in Section 11.14, The protected Data and Methods.**	

C1 can be accessed from a class C2 in the same package and from a class C3 in a different package

```
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();
  }
}
```

FIGURE 9.14 The private modifier restricts access to its defining class, the default modifier restricts access to a package, and the public modifier enables unrestricted access.

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

As shown in Figure 9.15, C1 can be accessed from C2 but not from C3.

```
package p1;

class C1 {
  ...
}
```

```
package p1;

public class C2 {
  can access C1
}
```

```
package p2;

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

FIGURE 9.15 A nonpublic class has package-access.

As shown in Figure 9.16a, an object c of class C can access its private members, because c is defined inside its own 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;
  }
}
```

(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.

FIGURE 9.16 An object can access its private members if it is defined in its own class.

Data Field Encapsulation

Making data fields private protects data and makes the class easy to maintain. The data fields `radius` and `numberOfObjects` in the `CircleWithStaticMembers` class in Listing 9.6 can be modified directly (e.g., `c1.radius = 5` or `CircleWithStaticMembers.numberOfObjects = 10`). This is not a good practice—for two reasons: ■ First, data may be tampered with. For example, `numberOfObjects` is to count the number of objects created, but it may be mistakenly set to an arbitrary value (e.g., `CircleWithStaticMembers.numberOfObjects = 10`). ■ Second, the class becomes difficult to maintain and vulnerable to bugs. Suppose you want to modify the `CircleWithStaticMembers` class to ensure that the radius is nonnegative after other programs have already used the class. You have to change not only the `CircleWithStaticMembers` class but also the programs that use it, because the clients may have modified the radius directly (e.g., `c1.radius = -5`). 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*.**	

```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.

FIGURE 9.16 An object can access its private members if it is defined in its own class.

The **this** Reference

The keyword **this** *refers to the object itself.* *It can also be used inside a constructor to* invoke another constructor of the same class. The **this** *keyword* is the name of a reference that an object can use to refer to itself. You can use the **this** keyword to reference the object's instance members. For example, the following code in (a) uses **this** to reference the object's **radius** and invokes its **getArea()** method explicitly. The **this** reference is normally omitted, as shown in (b). However, the **this** reference is needed to reference hidden data fields or invoke an overloaded constructor.	

```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() ;
  }
}
```

(a)

Equivalent

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

  ...

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

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

(b)

9.14.1 Using **this** to Reference Hidden Data Fields

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 simply by using the **ClassName.staticVariable** reference. A hidden instance variable can be accessed by using the keyword **this**, as shown in Figure 9.21a.

```
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
}
```

(a)

```
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
```

(b)

FIGURE 9.21 The keyword **this** refers to the calling object that invokes the 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**, as shown in Figure 9.21b.

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.

Using **this** to Invoke a 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.

The line `this(1.0)` in the second constructor invokes the first constructor with a `double` value argument.

9.32 Describe the role of the `this` keyword.

9.33 What is wrong in the following code?

```
public class C {
private int p;

public C() {
System.out.println("C's no-arg constructor invoked");
this(0);
}

public C(int p) {
p = p;
}

public void setP(int p) {
p = p;
}
}
```

9.34 What is wrong in the following code?

```
public class Test {
private int id;
public void m1() {
this.id = 45;
}
public void m2() {
Test.id = 45;
}
}
```

Immutable Objects and Classes

You can define immutable classes to create immutable objects. *The contents of immutable objects cannot be changed.*	
<pre>public class Student { private int id; private String name; private java.util.Date dateCreated; public Student(int ssn, String newName) { id = ssn; name = newName; dateCreated = new java.util.Date(); } public int getId() { return id; } public String getName() { return name; } public java.util.Date getDateCreated() { return dateCreated; } }</pre>	