

```

class TestStatic {

    static int count=6;
    int var;

    public TestStatic(){
        count+=1;

    }

    public static void main(String args[]){
        TestStatic ts1= new TestStatic();
        ts1.var=3;
        TestStatic ts2= new TestStatic();
        ts2.var=5;
        TestStatic ts3= new TestStatic();
        ts3.var=7;
        System.out.println("count= "+ ts1.count + ", var= "+ ts1.var);
        System.out.println("count= "+ ts2.count + ", var= "+ ts2.var);
        System.out.println("count= "+ ts3.count + ", var= "+ ts3.var);
    }
}

```

```

count= 9, var= 3
count= 9, var= 5
count= 9, var= 7

```

# Example

```
class Employee{
```

```
    static int id;
```

```
    static String name;
```

**Static variables**

```
    Employee (int newId, String newName){
```

```
        id=newId;
```

```
        name=newName;
```

```
    }
```

```
    public static void main (String [] args){
```

```
        Employee e1= new Employee(123,"Ahamd");
```

```
        Employee e2= new Employee(456,"Yamen");
```

```
        Employee e3= new Employee(983,"Amir");
```

```
    }
```

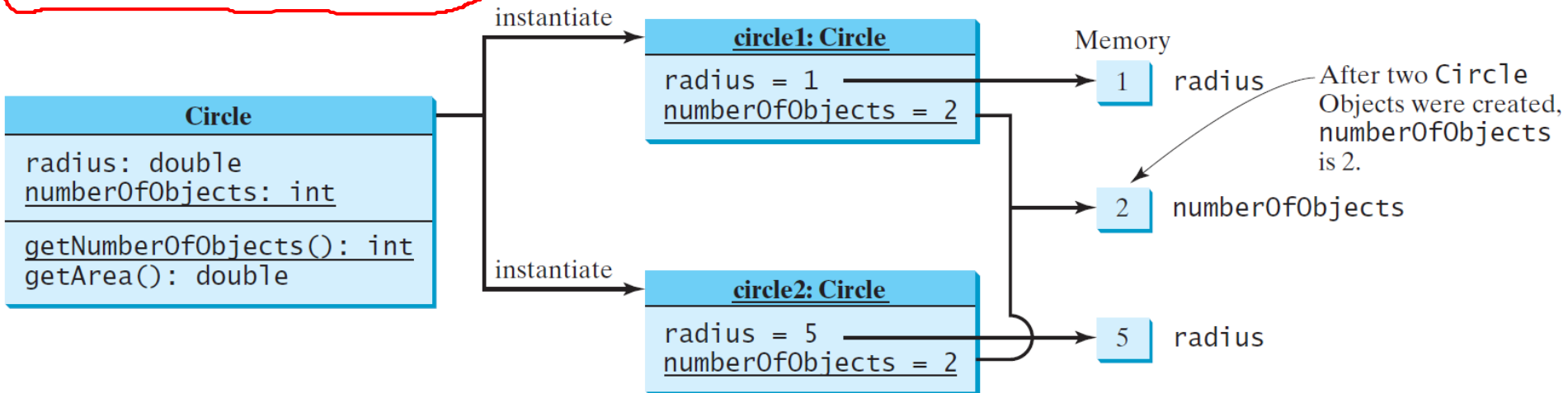
```
}
```

# Static Variables, Constants, and Methods, cont.

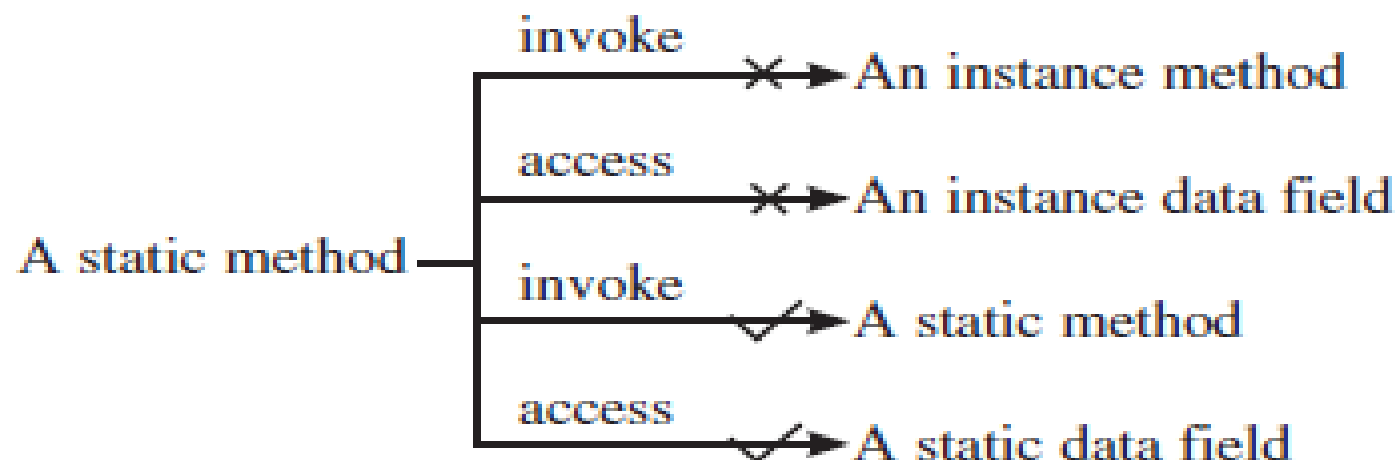
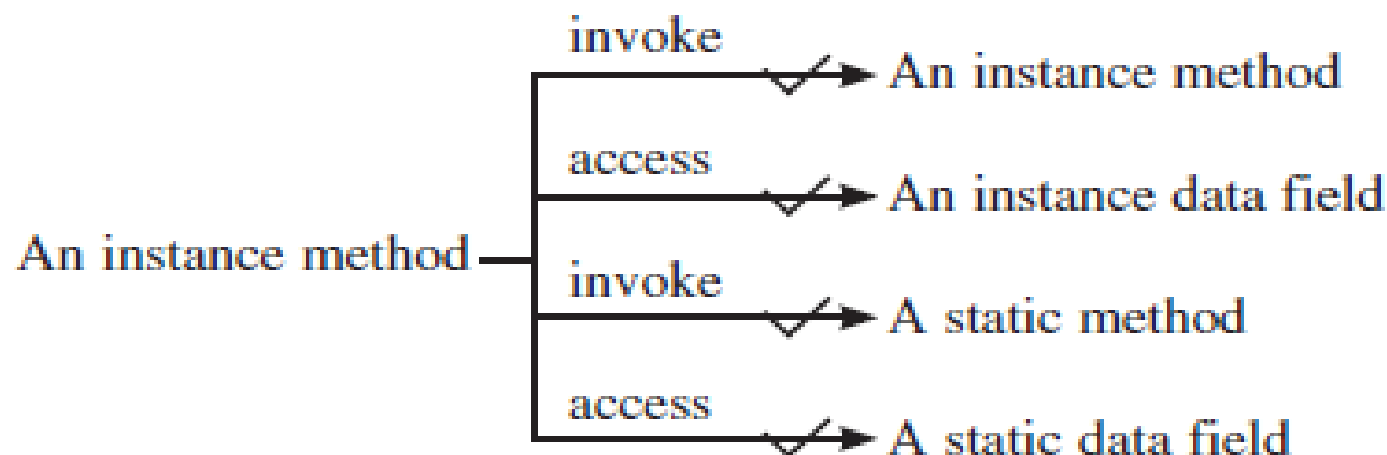
To **declare static variables, constants, and methods**, use the **static** modifier.

# Static Variables, Constants, and Methods, cont.

UML Notation:  
underline: static variables or methods



# Instance and Static and Methods



# 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.

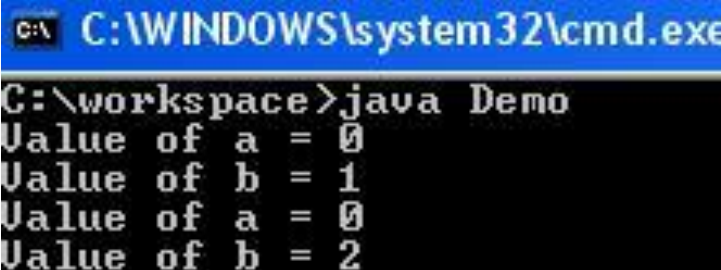
Syntax : <class-name>.<*static-method-name*>

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 }
```



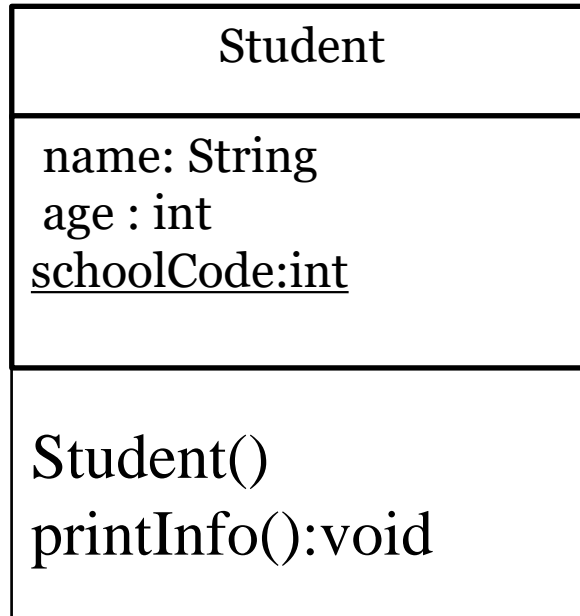
C:\WINDOWS\system32\cmd.exe

```
C:\workspace>java Demo
Value of a = 0
Value of b = 1
Value of a = 0
Value of b = 2
```

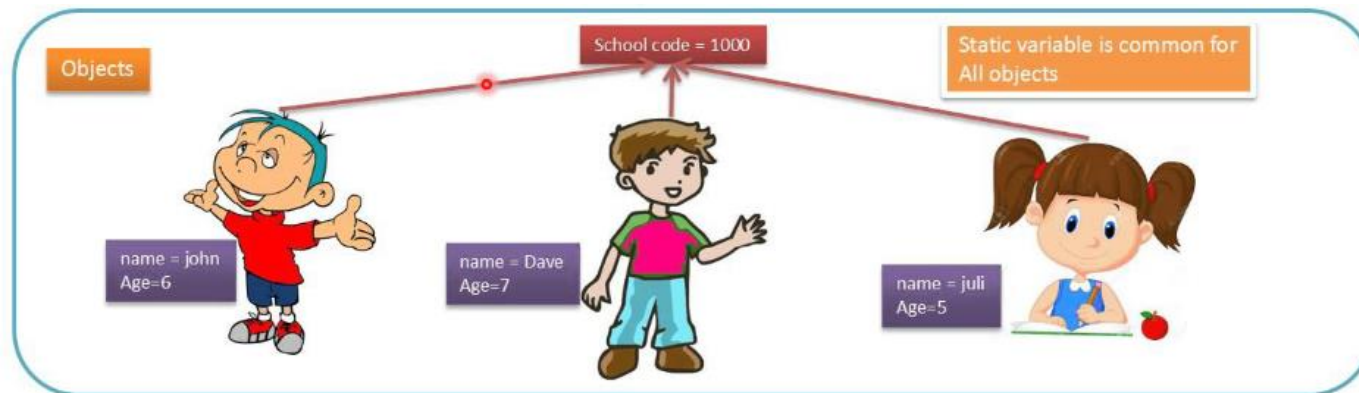


# Example

Create a class students based on the following UML.



Increase the school code  
Print the student information



```

class Student{
    String name;
    int age;
    static int schoolCode;

    Student (){
        schoolCode++;
    }

    void printInfo(){
        System.out.println (name + ", " + age + ", " + schoolCode );
    }

    public static void main(String [] args){

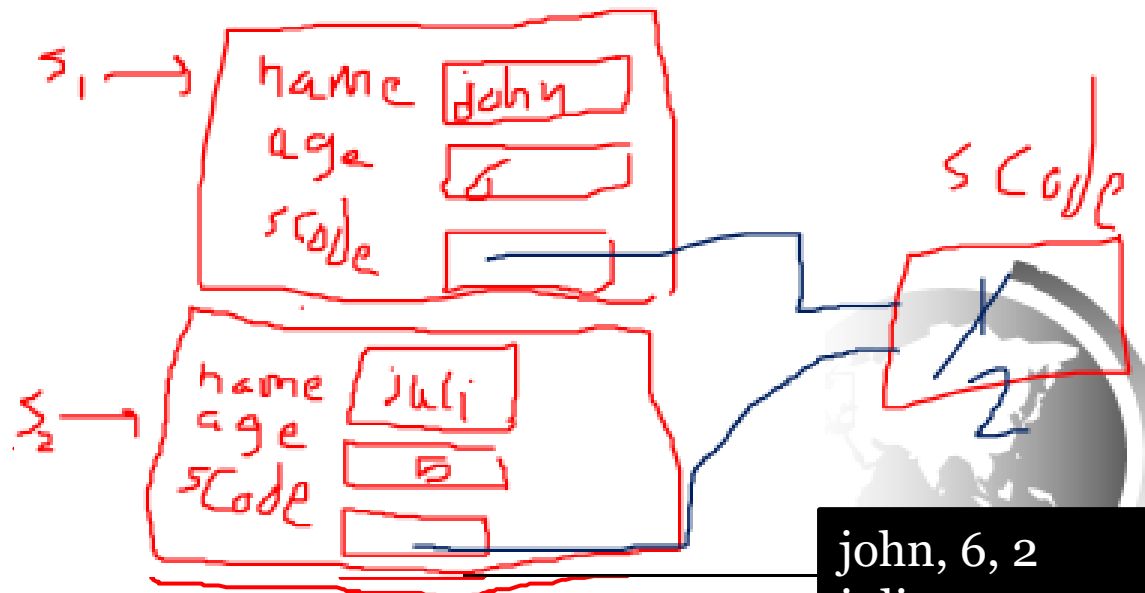
        Student s1=new Student();
        Student s2=new Student();
        s1.name="john";
        s1.age=6;

        s2.name="juli";
        s2.age=5;

        s1.printInfo();
        s2.printInfo();

    }
}

```



```
public class TV {  
  
    public int channel;  
    public int volumeLevel; // Accept any volume in the range [0,7]  
    public boolean on;  
  
}
```

```
public class Test {  
  
    public static void main(String[] args) {  
        TV obj = new TV();  
        obj.on = true;  
        obj.volumeLevel = 100;  
        System.out.println(obj.volumeLevel);  
    }  
}
```

There is no control  
over what is stored in  
its fields

100

```
public class TV {  
    private int channel;  
    private int volumeLevel;  
    private boolean on;  
    public int getChannel() {  
        return channel;  
    }  
    public void setChannel(int newChannel) {  
        if ((on==true) && (newChannel>=1 && newChannel<=9))  
            channel = newChannel;  
        else  
            System.out.println("Incorrect Input");  
    }  
    public int getVolumeLevel() {  
        return volumeLevel;  
    }  
    public void setVolumeLevel(int newVolumeLevel) {  
        if ((on==true) && (newVolumeLevel>=0 && newVolumeLevel<=7))  
            volumeLevel = newVolumeLevel;  
        else  
            System.out.println("Incorrect Input");  
    }  
    public boolean isOn() {  
        return on;  
    }  
    public void setOn(boolean turnOn) {  
        on = turnOn;  
    }  
}
```

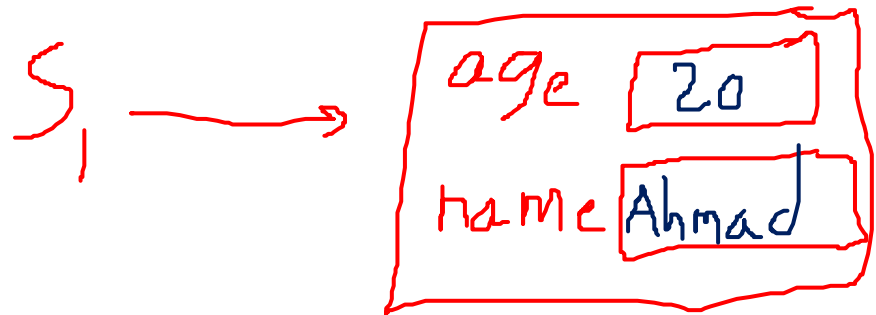
```
public class Test {  
  
    public static void main(String[] args) {  
        TV obj =new TV();  
        obj.setOn(true);  
        obj.setVolumeLevel(12);  
        obj.setVolumeLevel(3);  
        System.out.println(obj.getVolumeLevel());  
    }  
}
```

Incorrect Input  
3

A class can have total control over what is stored in its fields

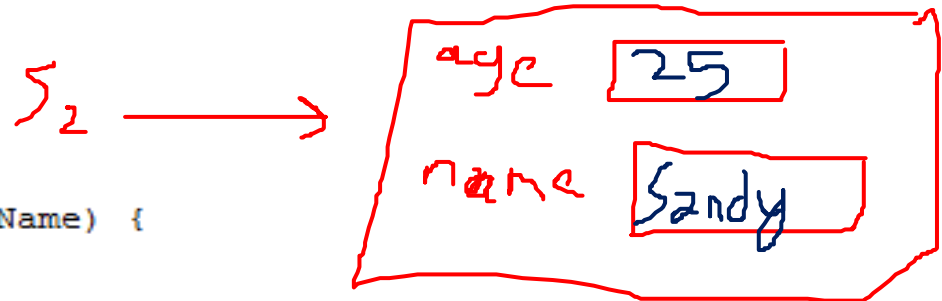
```
public class Student {
    private int age;
    private String name;

    public int getAge() {
        return age;
    }
}
```



```
    public void setAge(int newAge) {
        age = newAge;
    }
}
```

```
    public String getName() {
        return name;
    }
}
```



```
    public void setName(String newName) {
        name = newName;
    }
}
```

```
public static void main(String[] args) {
    Student s1 = new Student ();
    Student s2 = new Student ();
    s1.setAge(20);
    s1.setName("Ahmad");

    s2.setAge(25);
    s2.setName("Sandy");

    System.out.println(s1.getAge() + " , " + s1.getName());
    System.out.println(s2.getAge() + " , " + s2.getName());
}
```

20	,	Ahmad
25	,	Sandy

```
public class Vehicle {
```

```
    private int model;  
    private String name;  
    private String color;
```

```
    public int getModel() {  
        return model;  
    }
```

```
    public void setModel(int newModel) {  
        model = newModel;  
    }
```

```
    public String getName() {  
        return name;  
    }
```

```
    public void setName(String newName) {  
        name = newName;  
    }
```

```
    public String getColor() {  
        return color;  
    }
```

```
    public void setColor(String newColor) {  
        color = newColor;  
    }
```

```
public class TestVehicle {
```

```
    public static void main(String[] args) {  
        Vehicle obj =new Vehicle();  
        Vehicle obj2 =new Vehicle();  
        obj.setName("BMW");  
        obj.setColor("Black");  
        obj.setModel(2017);
```

```
        obj2.setName("BMW");  
        obj2.setColor("Red");  
        obj2.setModel(2016);
```

```
        System.out.println(obj.getName() + " " + obj.getColor() +  
                             " "+ obj.getModel());
```

```
        System.out.println(obj2.getName() + " " + obj2.getColor() +  
                             " "+ obj2.getModel());  
    }
```

```
}
```

```
BMW Black 2017  
BMW Red 2016
```

The following has an error:

class **TestVehicle**{ .....

public static void main (String [] args){

**Vehicle** obj=new **Vehicle**();

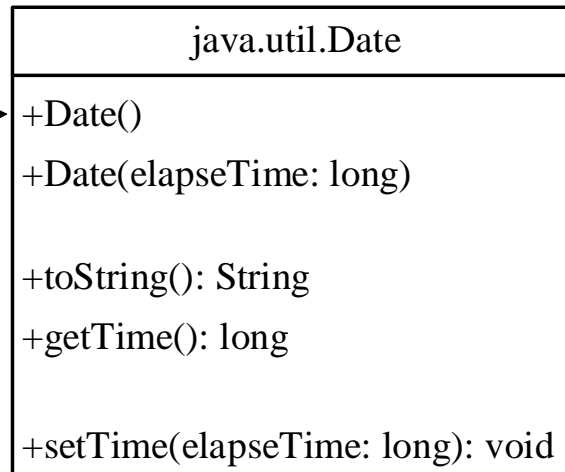
**Obj.model=2017; //private model**

....

# 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



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 Sun Mar 09 13:50:19 EST 2003.

```
Sun Mar 19 14:59:43 IST 2017
```

```
Sun Mar 19 13:00:22 UTC 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 (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.

# 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.

```
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) + " ");
```

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

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

# 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.*

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

Same Package P<sub>1</sub>

```
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**.

Same package P<sub>1</sub>

```
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**, and the **public** modifier **enables unrestricted access**.

```
package P1;
```

```
public class A {
```

```
    /*Default */
```

```
    static int x=5; //remove static and check
```

```
    /*Private*/
```

```
    private static int m=9; // You can use in the same class
```

```
    /*Public*/
```

```
    public static int w=7;
```

```
    public static void main (String []args){
```

```
        System.out.println("x (default) = " + x);
```

```
        System.out.println("m (private)= " + m);
```

```
        System.out.println("w (public) = " + w);
```

```
    }
```

```
}
```

```
package P1;
```

```
public class B {
```

```
    /*default*/
```

```
    static int x2=1; //remove static and check
```

```
    public static void main(String[] args) {
```

```
        System.out.println("x2 (default in package p1) = " + x2);
```

```
        System.out.println("A.x (default in package p1)= " + A.x);
```

```
        //System.out.println("A.m (private in package p1) = " + A.m);
```

```
    }
```

```
}
```

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.

```
package P1;
```

```
class C { // default class
```

```
    static int l=9; // default varaibale
```

```
    public static void main(String[] args) {
```

```
        System.out.println ("l= " + l);
```

```
    }
```

```
}
```

```
package P2;
```

```
import P1.A;
```

```
/*import P1.C;*/ //Error change visibility to public
```

```
public class Test {
```

```
    public static void main(String[] args) {
```

```
        System.out.println("A.w= " + A.w);
```

```
    }
```

```
    //System.out.println("A.x= " + A.x); //Error: Visibility of x is default, change to public
```

```
    //System.out.println("C.l= " + C.l); // Error : create a class C (C not visible in this package)
```

```
}
```

```
}
```

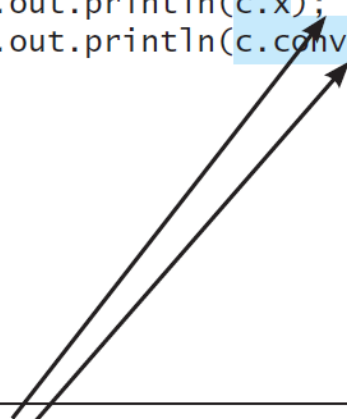
# NOTE

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`.



# Why Data Fields Should Be private?

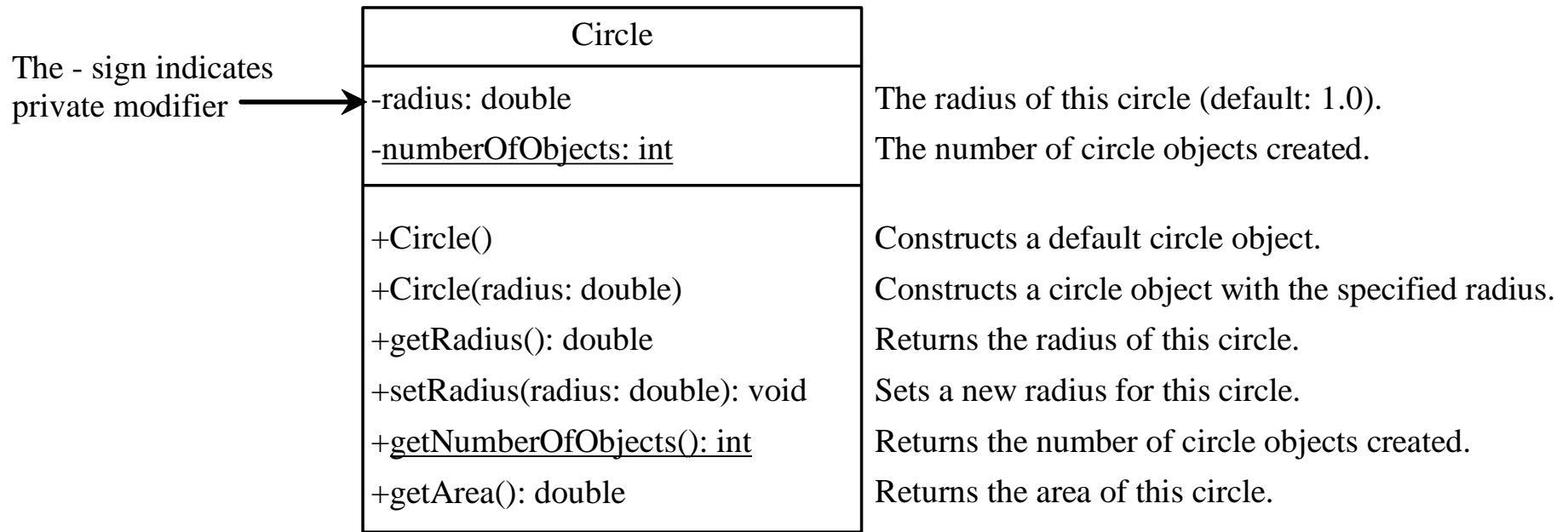


To protect data.



To make code easy to maintain.

# Example of Data Field Encapsulation



# 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)

```
class Apple {  
    public String color="red";  
}  
  
public class Main {  
    public static void main(String[] args) {  
        Apple apple = new Apple();  
        System.out.println(apple.color);  
  
        changeApple(apple);  
        System.out.println(apple.color);  
    }  
  
    public static void changeApple(Apple appleObj){  
        appleObj.color = "green";  
    }  
}
```

Output:

red  
green

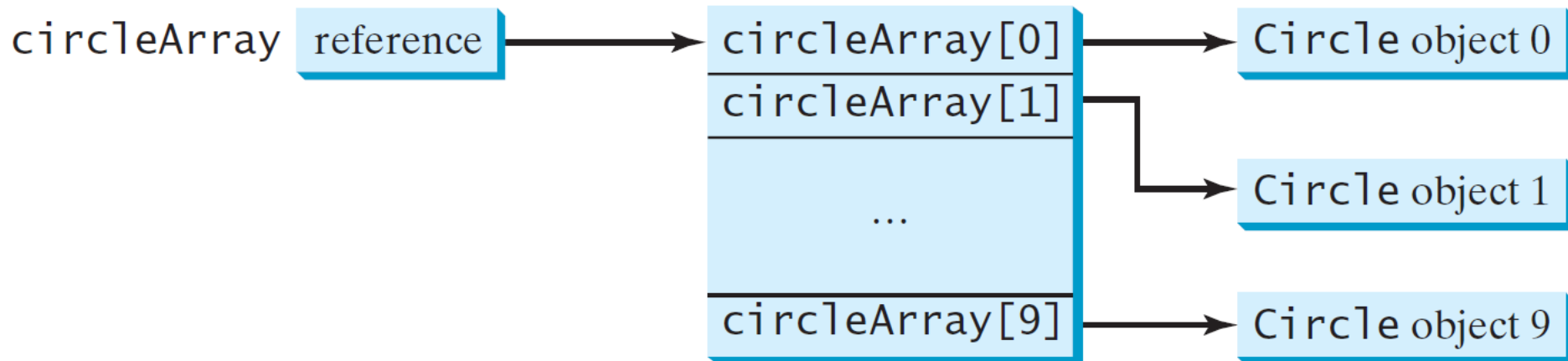
# Array of Objects

```
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, cont.

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



# Check Point

What is wrong in the following code?

```
1  public class Test {  
2      public static void main(String[] args) {  
3          java.util.Date[] dates = new java.util.Date[10];  
4          System.out.println(dates[0]);  
5          System.out.println(dates[0].toString());  
6      }  
7  }
```

(Line 4 prints null since dates[0] is null. Line 5 causes a NullPointerException since it invokes toString() method from the null reference.)

# Check Point

Suppose that the class F is defined in (a). Let f be an instance of F. Which of the statements in (b) are correct?

(a)	<code>System.out.println(f.i);</code>
<code>public class F {</code>	<code>Answer: Correct</code>
<code>    int i;</code>	
<code>    static String s;</code>	<code>System.out.println(f.s);</code>
	<code>Answer: Correct</code>
<code>    void imethod() {</code>	<code>f.imethod();</code>
<code>    }</code>	<code>Answer: Correct</code>
<code>    static void smethod() {</code>	
<code>    }</code>	<code>f.smethod();</code>
<code>}</code>	<code>Answer: Correct</code>
(b)	<code>System.out.println(F.i);</code>
<code>System.out.println(f.i);</code>	<code>Answer: Incorrect</code>
<code>System.out.println(f.s);</code>	<code>System.out.println(F.s);</code>
<code>f.imethod();</code>	<code>Answer: Correct</code>
<code>f.smethod();</code>	
<code>System.out.println(F.i);</code>	<code>F.imethod();</code>
<code>System.out.println(F.s);</code>	<code>Answer: Incorrect</code>
<code>F.imethod();</code>	
<code>F.smethod();</code>	<code>F.smethod();</code>
	<code>Answer: Correct</code>



# Check Point

Add the static keyword in the place of ? if appropriate.

```
public class Test {  
    int count;  
    public ? void main(String[] args) {  
        ...  
    }  
    public ? int getCount() {  
        return count;  
    }  
    public ? int factorial(int n) {  
        int result = 1;  
        for (int i = 1; i <= n; i++)  
            result *= i;  
        return result;  
    }  
}
```

Add static in the main method and in the factorial method because these two methods don't need reference any instance objects or invoke any instance methods in the Test class.

# Immutable Objects and Classes

- If the *contents of an object cannot be changed* once the object is created, the object is called an *immutable object* and its class is called an *immutable class*.
- A class with all private data fields and without mutators is **not necessarily immutable**.

# 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;  
    }  
}
```

# Example

```
public class Student {
    private int id;
    private BirthDate birthDate;

    public Student(int ssn,
        int year, int month, int day) {
        id = ssn;
        birthDate = new BirthDate(year, month, day);
    }

    public int getId() {
        return id;
    }

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

```
public class BirthDate {
    private int year;
    private int month;
    private int day;

    public BirthDate(int newYear,
        int newMonth, int newDay) {
        year = newYear;
        month = newMonth;
        day = newDay;
    }

    public void setYear(int newYear) {
        year = newYear;
    }
}
```

```
public class Test {
    public static void main(String[] args) {
        Student student = new Student(111223333, 1970, 5, 3);
        BirthDate date = student.getBirthDate();
        date.setYear(2010); // Now the student birth year is changed!
    }
}
```

# What Class is Immutable?

For a class to be **immutable**, it must mark all data fields private and provide no mutator methods and no accessor methods that would return a reference to a mutable data field object.

**Note:**

getter called accessor  
setter called mutator

# Check Point

- If a class contains only private data fields and no setter methods, is the class immutable?

No. It must also contain no get methods that would return a reference to a mutable data field object.

- Is the following class immutable?

```
public class A {  
    private int[] values;  
  
    public int[] getValues() {  
        return values;  
    }  
}
```

No, because values is a reference type.

# 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.

# 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;  
  
    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.

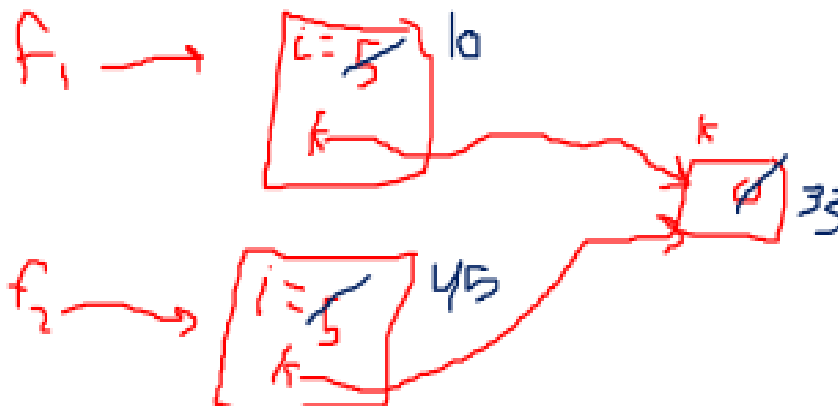
Invoking f1.setI(10) is to execute  
`this.i = 10`, where `this` refers to f1

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

# 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***.

# Calling Overloaded Constructor

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

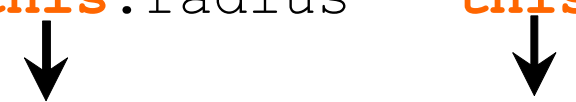
```
    public Circle(double radius) {  
        this.radius = radius;
```

 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

# Revision of the OOP basic concepts (Objects and Classes)