



BIRZEIT UNIVERSITY

---

# Objects & Classes

Liang, Introduction to Java Programming and Data Structures,  
Twelfth Edition, (c) 2020 Pearson Education, Inc. All rights reserved.





By: Mamoun Nawahdah (Ph.D.)  
2022

## OO Programming Concepts

- ❖ **Object-Oriented Programming (OOP)** involves programming using objects.
- ❖ An **object** represents an entity in the 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**.



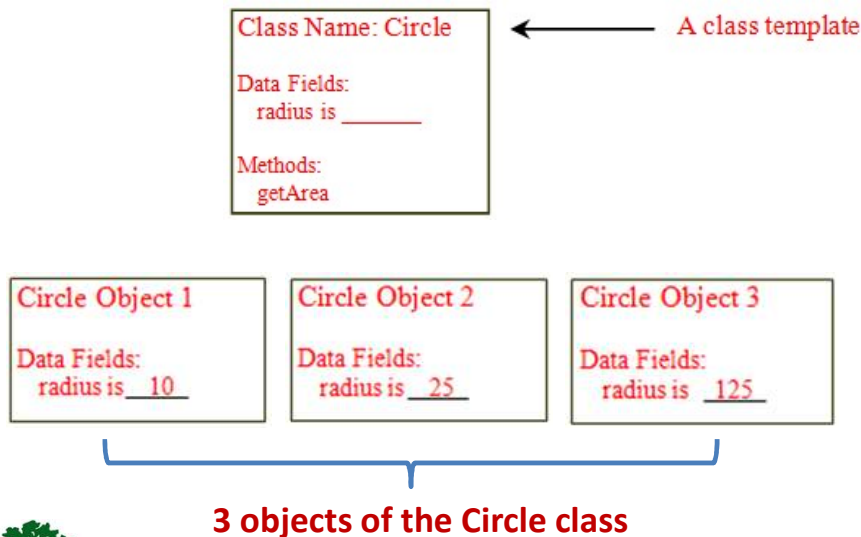
## Objects and Classes

- ❖ 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.



3

## Objects and Classes cont.



4

## 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

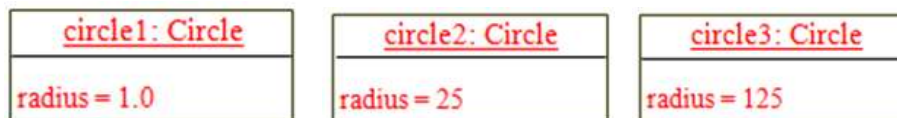
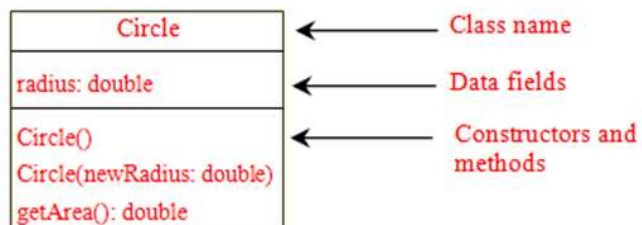
Constructors

Method

5

## UML Class Diagram

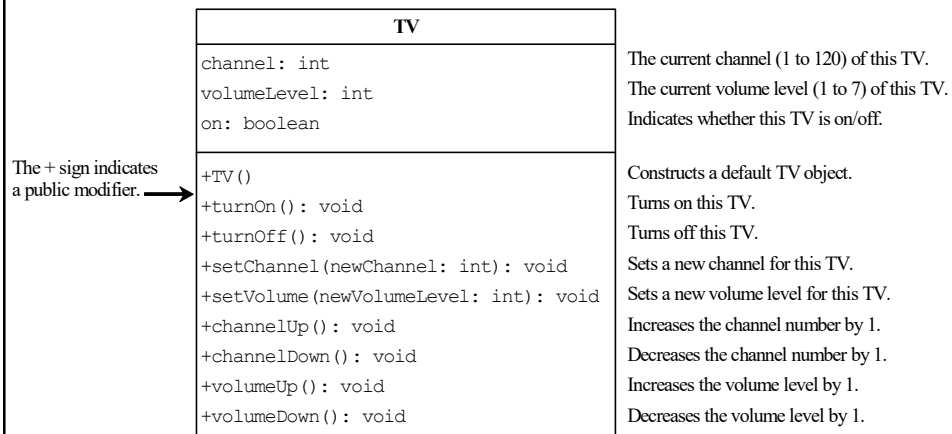
UML Class Diagram



UML notation for objects

6

## Example: TV Class



7

## Constructors

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

```

Circle() {
}

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



8

## 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.



9

## Creating Objects Using Constructors

---

**new ClassName();**

Example:

**new Circle();**

**new Circle(5.0);**



10

## 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.*



11

## 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;**



12

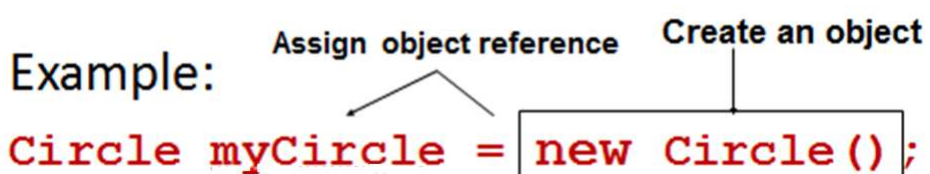
## Declaring/Creating Objects in a Single Step

```
ClassName objectRefVar = new ClassName();
```

Example:

Assign object reference      Create an object

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



13

## 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()**



14

## Trace Code

---

```
Circle myCircle = new Circle(5.0);
```

```
Circle yourCircle = new Circle();
```

```
yourCircle.radius = 100;
```

Declare myCircle

myCircle

no value



## Trace Code, cont.

---

```
Circle myCircle = new Circle(5.0);
```

```
Circle yourCircle = new Circle();
```

```
yourCircle.radius = 100;
```

myCircle

no value

: Circle

radius: 5.0

Create a circle



16



## Trace Code, cont.

Circle myCircle = new Circle(5.0);

Circle yourCircle = new Circle();

yourCircle.radius = 100;

myCircle reference value

Assign object reference  
to myCircle

<u>: Circle</u>
radius: 5.0



17

## Trace Code, cont.

Circle myCircle = new Circle(5.0);

Circle yourCircle = new Circle();

yourCircle.radius = 100;

myCircle reference value

<u>: Circle</u>
radius: 5.0

yourCircle no value

Declare yourCircle



18

## Trace Code, cont.

---

```
Circle myCircle = new Circle(5.0);
Circle yourCircle = new Circle();
yourCircle.radius = 100;
```


myCircle reference value

: Circle
radius: 5.0

yourCircle no value

: Circle
radius: 1.0

Create a new Circle object



19

## Trace Code, cont.

---

```
Circle myCircle = new Circle(5.0);
Circle yourCircle = new Circle();
yourCircle.radius = 100;
```


myCircle reference value

: Circle
radius: 5.0

yourCircle reference value

: Circle
radius: 1.0

Assign object reference to yourCircle



20

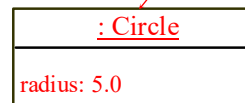
## Trace Code, cont.

```
Circle myCircle = new Circle(5.0);
```

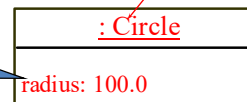
```
Circle yourCircle = new Circle();
```

```
yourCircle.radius = 100;
```

myCircle reference value



yourCircle reference value



Change radius in  
yourCircle



21

## 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'
}
```



22

## 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.**



23

## Example

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

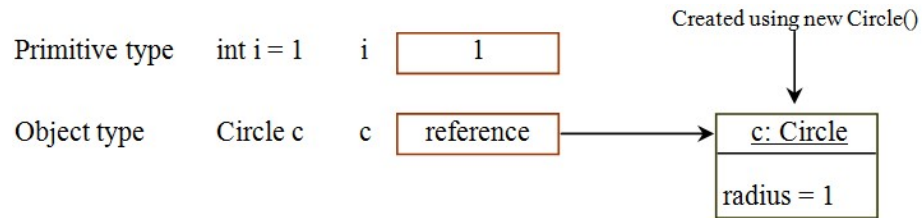
```
public class Test {
    public static void main(String[] args) {
        int x;    // x has no default value
        String y;    // y has no default value
        System.out.println("x is " + x);
        System.out.println("y is " + y);
    }
}
```



Compilation error: **variable not initialized**

24

## Differences between Variables of Primitive Data Types and Object Types



25

## Copying Variables of Primitive Data Types and Object Types

Primitive type assignment `i = j`

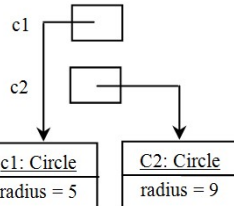
Before:

i    1j    2

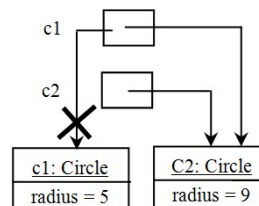
After:

i    2j    2Object type assignment `c1 = c2`

Before:



After:



26

## 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**.
- ❖ If you know that an object is no longer needed, you can **explicitly** assign **null** to a reference variable for the object.



27

## 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	
<p>The + sign indicates public modifier →</p>	<pre> +Date() +Date(elapseTime: long)  +toString(): String +getTime(): long  +setTime(elapseTime: long): void           </pre>
	<p>Constructs a Date object for the current time.</p> <p>Constructs a Date object for a given time in milliseconds elapsed since January 1, 1970, GMT.</p> <p>Returns a string representing the date and time.</p> <p>Returns the number of milliseconds since January 1, 1970, GMT.</p> <p>Sets a new elapse time in the object.</p>



28

## 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***



29

## 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.



30

## 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)
+distance(x: double, y: double): double
+distance(p: Point2D): double
+getX(): double
+getY(): double
+toString(): String
```

```
Constructs a Point2D object with the specified x- and y-coordinates.
Returns the distance between this point and the specified point (x, y).
Returns the distance between this point and the specified point p.
Returns the x-coordinate from this point.
Returns the y-coordinate from this point.
Returns a string representation for the point.
```



31

## Instance (**Object**) Variables, and Methods

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



32



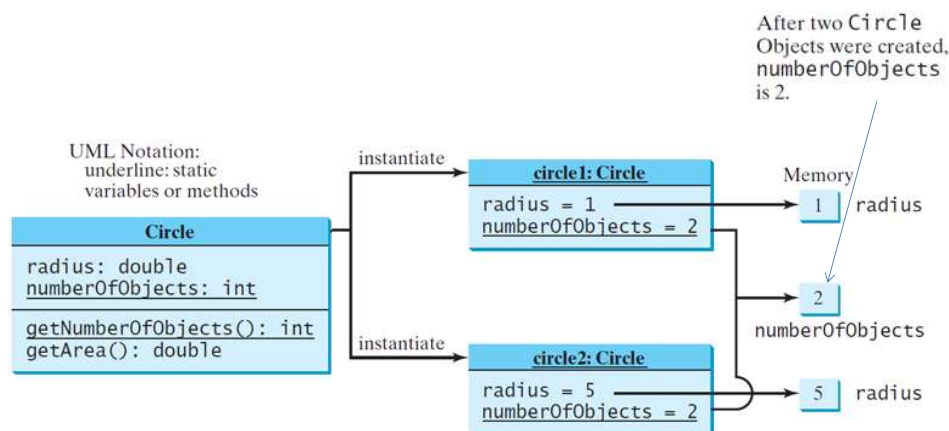
## Static Variables, Constants, and Methods

- ❖ **Static variables** are **shared** by all the instances (objects) of the class.
- ❖ **Static methods** are not tied to a specific instance (object).
- ❖ **Static constants** are **final** variables shared by all the instances (objects) of the class.
- ❖ To declare static *variables*, *constants*, and *methods*, use the **static** modifier.



33

## Static



34

## Static Variable

- ❖ It is a variable which belongs to the **class** and not to the **instance (object)**.
- ❖ Static variables are **initialized only once**, at the start of the execution.
  - Static 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 instance (**object**).
- ❖ 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.



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

## Static example

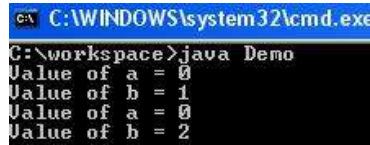
```

class Student {
    int a; //initialized to zero
    static int b; //initialized to zero only when class is loaded
    Student(){
        //Constructor incrementing static variable b
        b++;
    }

    public void showData(){
        System.out.println("Value of a = "+a);
        System.out.println("Value of b = "+b);
    }
    //public static void increment(){
    //a++;
    //}
}

class Demo{
    public static void main(String args[]){
        Student s1 = new Student();
        s1.showData();
        Student s2 = new Student();
        s2.showData();
        //Student.b++;
        //s1.showData();
    }
}

```



```

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

```

## Visibility Modifiers

❖ **By default** (when no visibility modifiers are used), the *class*, *variable*, or *method* can be accessed by any class in the same **package**.


👉 **public** modifier: The *class*, *data*, or *method* is visible to any class in any package.

👉 **private** modifier: The *data* or *method* can be accessed only by the declaring class.

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

<pre>package p1;  public class C1 {     public int x;     int y;     private int z;      public void m1() {     }     void m2() {     }     private void m3() {     } }</pre>	<pre>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();     } }</pre>	<pre>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();     } }</pre>
---	---	---

☞ 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.




39

---

<pre>package p1;  class C1 {     ... }</pre>	<pre>package p1;  public class C2 {     can access C1 }</pre>	<pre>package p2;  public class C3 {     cannot access C1;     can access C2; }</pre>
--	---	--

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



40

## 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`.



41

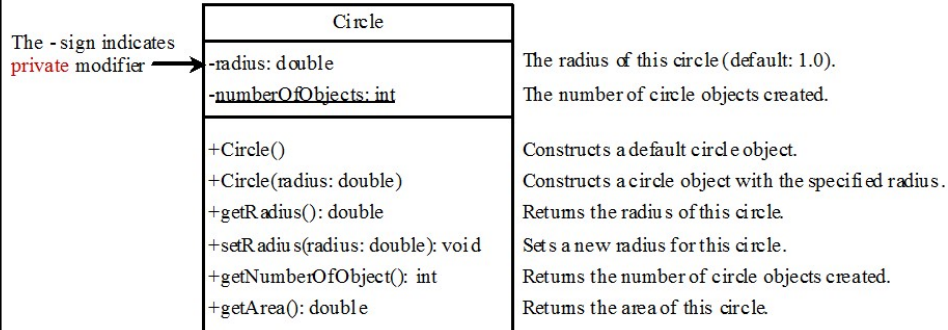
## Why Data Fields Should Be private?

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



42

## Example of Data Field Encapsulation



43

## Overloading Methods/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 reference** for reference type value:
  - The value is the **reference** to the object



45

## 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);
    }

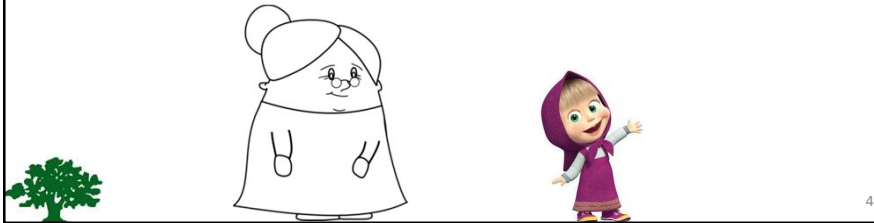
    /** Print a table of areas for radius */
    public static void printAreas(Circle c, int times) {
        System.out.println("Radius\t\tArea");
        while (times >= 1) {
            System.out.println(c.getRadius() + "\t\t" + c.getArea());
            c.setRadius(c.getRadius() + 1);
            times--;
        }
    }
}
```



46

## 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***.



47

## 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;
    }
}
```



48



## Immutable Objects and Classes

- ❖ A class with all **private data** fields and without **set** methods is not necessarily immutable.
- ❖ For example, the following class **Student** has all **private** data fields and no **set** methods, but it is mutable!!!



## Example

```
public class Student {
    private int id;
    private java.util.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 set** methods.
  - No **get** methods that would return a reference to a mutable data field object.



51

## Scope of Variables

- ❖ The scope of **instance** (object) 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.



52

## Scope of Variables

### ❖ What is the output?

```
public class A {
    int year = 2020; // instance variable

    void p() {
        System.out.println("Year: "+ year);
        int year = 2021; // local variable
        System.out.println("Year: "+ year);
    }
}
```

```
public class B {
    public static void main (String[] s) {
        A a = new A();
        a.p();
    }
}
```



## 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



55

## 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

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



56