

# **JAVA @11**

Access Specifier,  
Constructors, Methods

# Objective

After completing this session you will be able to understand,

- Access Modifiers
- Encapsulation
- Method overloading
- Static keyword
- Constructor
- Constructor chaining

# Access Modifier

In my Facebook profile page, I do not wish to share some personal information like my 'age'. But, I would like to share my 'email' with my friends and I would like everyone to be able to access my 'school/university' details. How do I do this?



Facebook application has features where users can select what level of access to be provided to their profile information.

FacebookProfile

**No access:** age

**Private access:** emailId

**Public access:** school/college

# How is access level defined in Java?

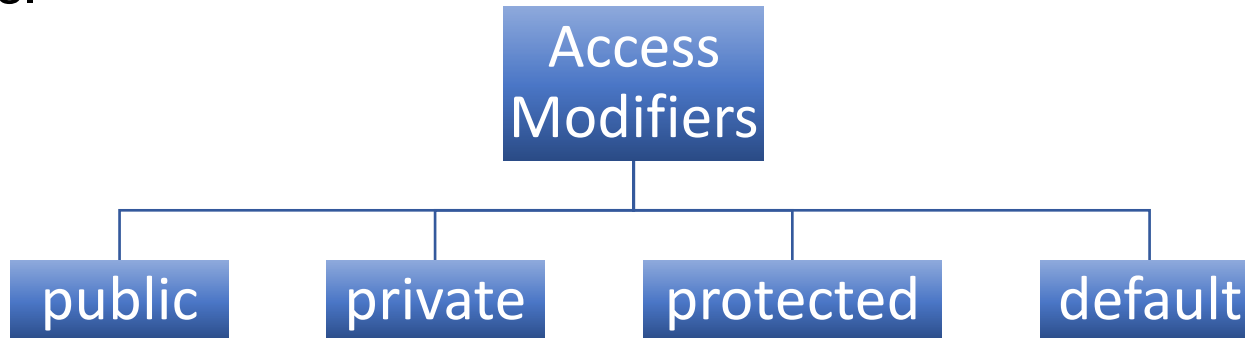
Similarly when developing java application developers needs to secure their methods/variables by defining different access level.

This is achieved using “Access Modifiers”

# What is Access Modifier?

What are access modifiers?

The scope of a variable/methods can be defined by using access modifiers.



Analogy to the Facebook example:

- School details can be made **public** (can be accessed by any class)
- Age can be made **private** (to be accessed only by the same class)
- Email Id can be made **default** (to be accessed only by the classes in same package)

# Public access

What is public access?

**Public** access specifies that the class members (variables or methods) are accessible to **anyone**, both inside and outside the class and outside of the package.

Syntax: `public <methods/variable name>`

Example:

public variable:

```
public int x = 0;
```

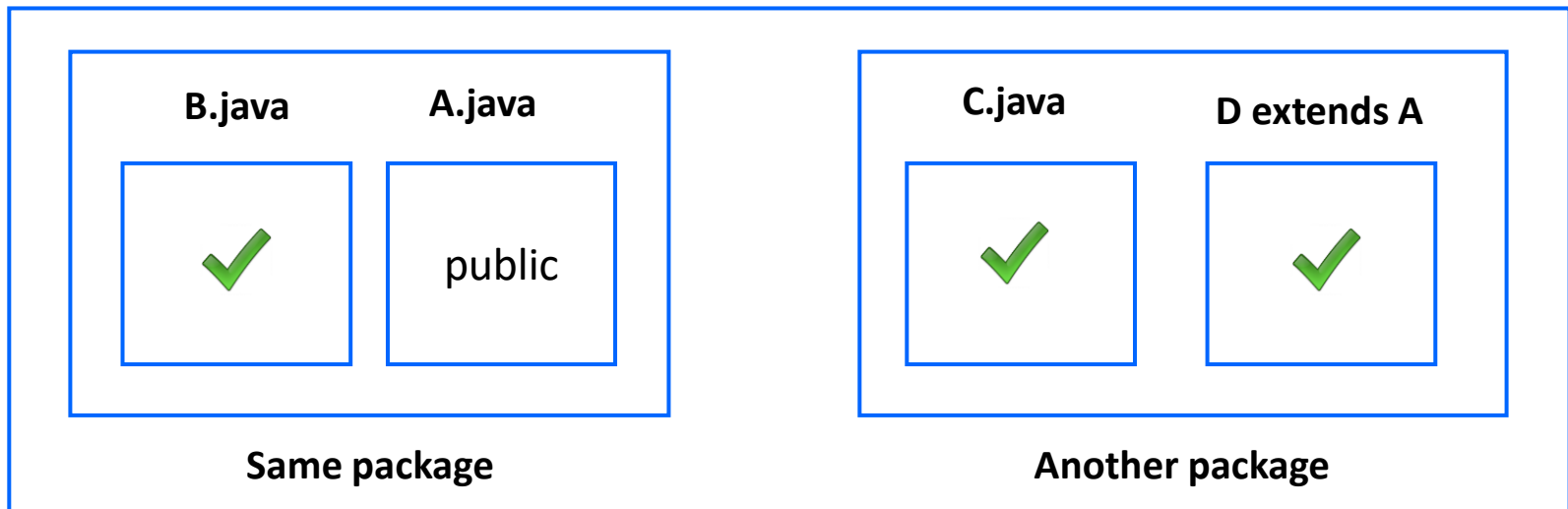
public methods:

```
public addNumbers(int a, int b){  
    //code block  
}
```

# public access

Illustration for public access:

- Assume there are 4 java classes A, B, C and D.
- Classes A & B are in the same package where as C & D is in a different package. D is a subclass from A.
- Class A has a public variable. Class B will be able to access the public variable in class A.
- Class C & D will be able to access the public variable in class A



# Private access

What is private access?

**Private** access specifies that the class members (variables or methods) are only accessible by the class in which they are defined. Cannot be accessed by any other class.

Syntax: `private <methods/variable name>`

Example:

private variable:

```
private int x = 0;
```

private methods:

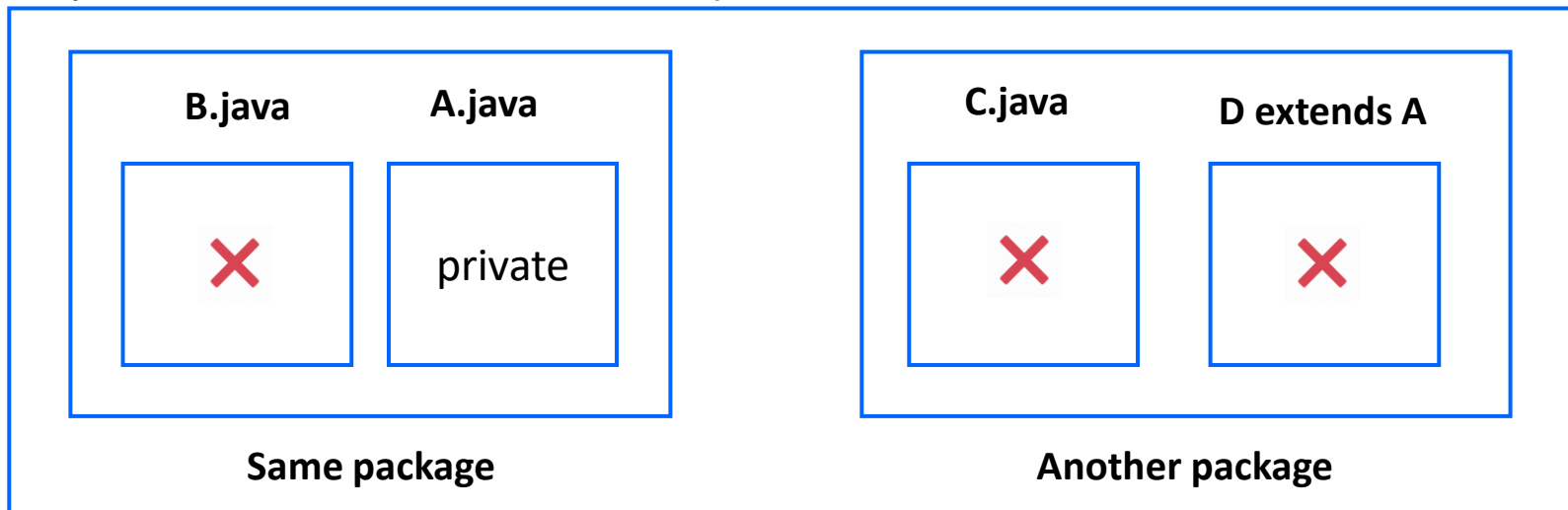
```
private addNumbers(int a, int b){  
    //code block  
}
```



# private access

Illustration for protected access:

- Class A has a **private** variable.
- Class B will **not** be able to access the private variable in class A
- Class C will **not** be able to access the private variable in class A
- Class D will not be able to access the private since it is a subclass from class A
- Only methods in class A can access the private variable in class A.



# Protected access

What is protected access?

**Protected** access specifies that the class members (variables or methods) are accessible to only the methods in that **class**, classes from **same** package and the **subclasses** of the class. The subclass can be in any package.

Syntax: `protected <methods/variable name>`

Example:

protected variable:

```
protected int x = 0;
```

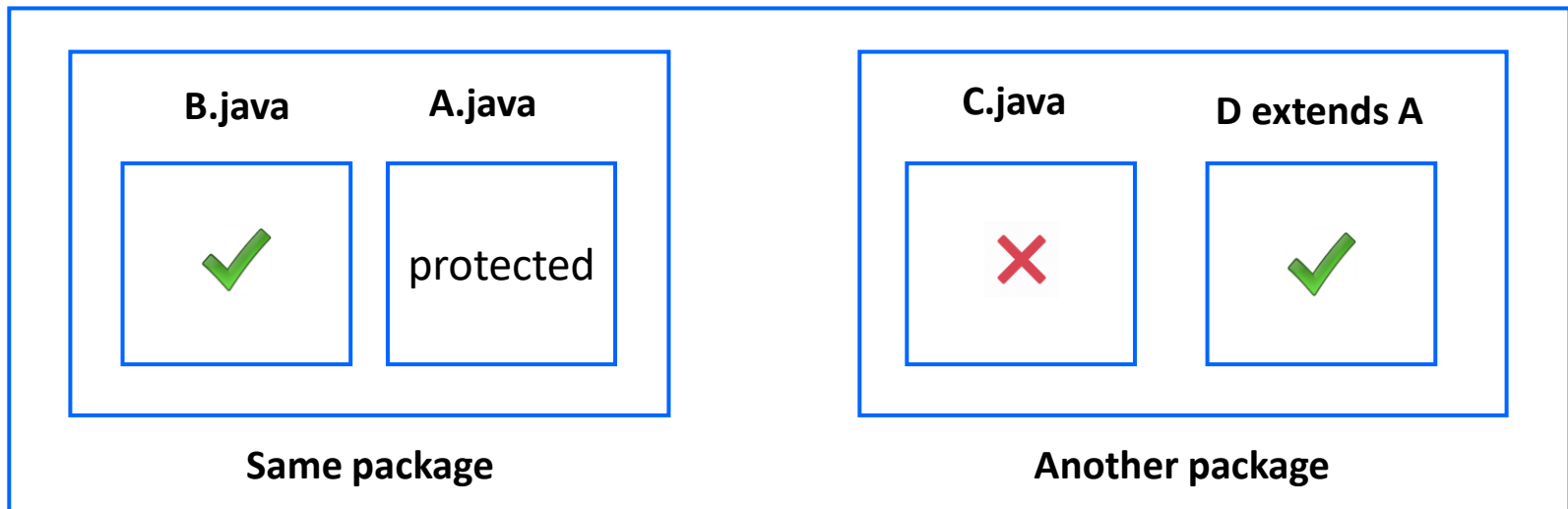
protected methods:

```
protected addNumbers(int a, int b){  
    //code block  
}
```

# Protected access

Illustration for protected access:

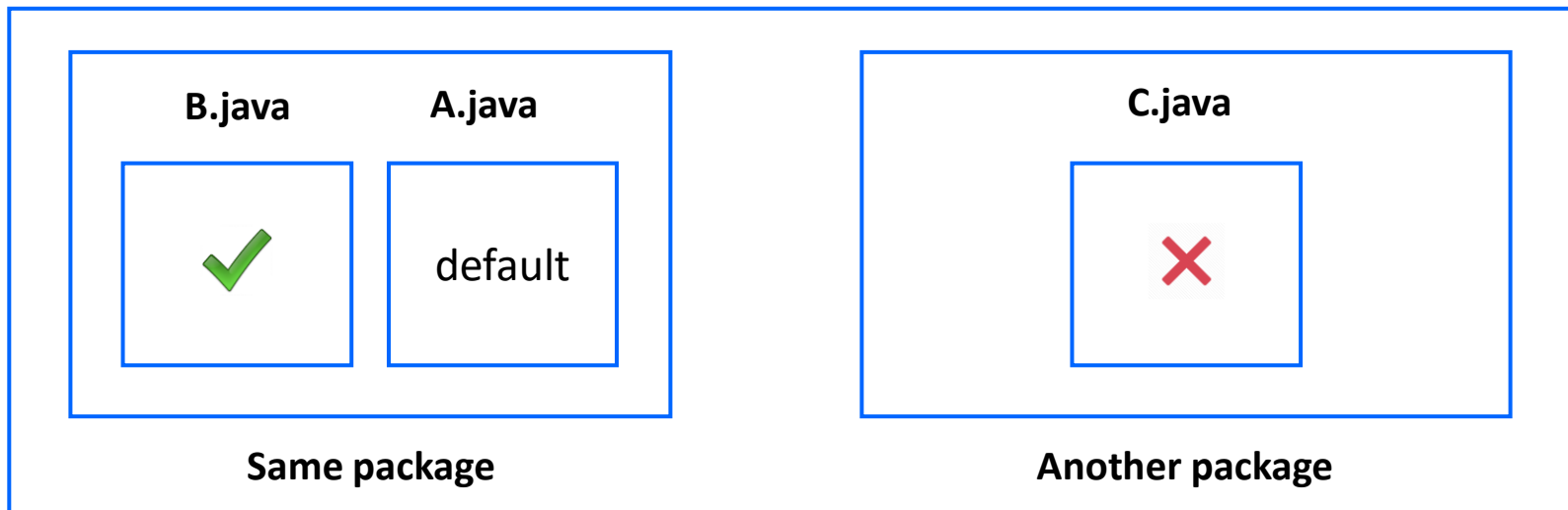
- Class A has a protected variable.
- Protected variable in class A can be accessed by class B in the same package.
- It can be accessed by class D since class D is a subclass of class A
- It cannot be accessed by class C since class C is in a different package and is not a sub class of class A.



# Default access

What is default access?

**Default/No** access specifies that only classes in the **same package** can have access to the variables and methods of the other class. No keyword is required for the default modifier and it is applied in the absence of an access modifier.

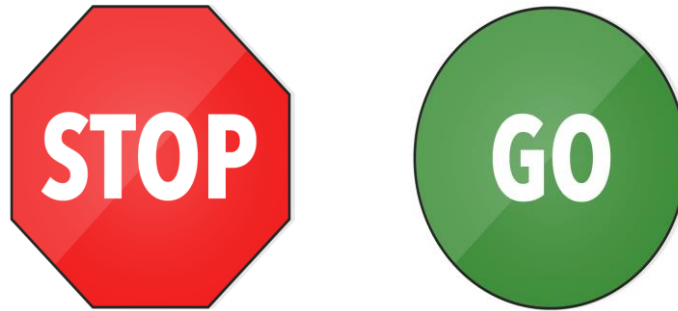


# Modifiers in a nutshell

<i>Access Modifier</i>	<i>Same Class</i>	<i>Same Package</i>	<i>Subclass</i>	<i>Other packages</i>
public	Y	Y	Y	Y
protected	Y	Y	Y	N
No access modifier	Y	Y	N	N
private	Y	N	N	N

*public* and *private* are the commonly used access specifiers in projects

# Time To Reflect



Trainees to reflect the following topics before proceeding.

- Why do you need access modifiers?
- What are the types of access modifiers?
- A method declared protected can it be accessed from other class residing in another package?
- A variable declared private can it be accessed from other class residing in the same package?
- What is the access specifiers used to prevent sub classes accessing the methods and variables?

# Methods

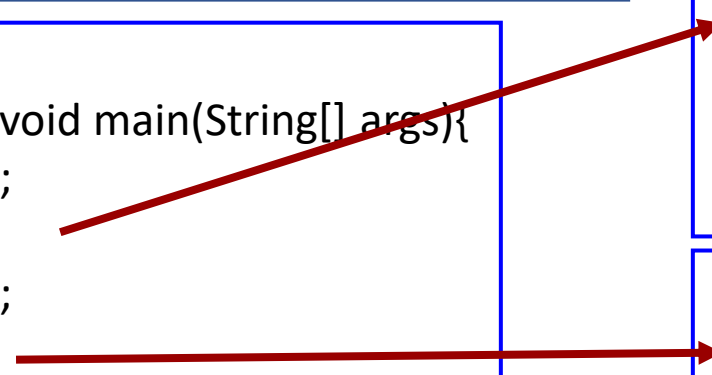
What is methods?

A method is a set of statements to perform a desired functionality which can be included inside a java class.

This set of statements can be called (invoked) at any point in the program by using the method name.

**Example:** main method in class A invokes method “add” in class B and method “multiply” in class C

```
class A{  
    public static void main(String[] args){  
        statement1;  
        add();  
        statement2;  
        multiply();  
    }  
}
```



The diagram illustrates the method invocation process. Two red arrows originate from the code in class A. The first arrow points from the **add();** line to the **add()** method definition in class B. The second arrow points from the **multiply();** line to the **multiply()** method definition in class C.

```
class B{  
    public void add(){  
        //code block  
    }  
}
```

```
class C{  
    public void multiply(){  
        //code block  
    }  
}
```

# Method declaration

How to declare a method?

Syntax:

```
<modifier><returnType><methodName>(<parameter-list>){  
    <statements>*  
}
```

Example:

```
public int add(int x, int y){  
    int sum = x+y;  
    return sum;  
}
```



# Encapsulation

What is encapsulation?



The actual medicine is hidden inside the capsule. The patient is not aware of the contents of the medicine.

# Encapsulation

What is Encapsulation?

- Encapsulation is one of the fundamental OOP concepts.
- It is the protective barrier that prevents the data in a class from being directly accessed by the code outside the class.
- Access to the data and code is tightly controlled by using an interface.

Example: In facebook people don't share their age information.

# How is Encapsulation achieved?

How is Encapsulation done?

- The fields in class are made ***private*** so that it cannot be accessed by anyone outside the class.
- Hence encapsulation is also called “***Data Hiding***”.
- The fields can be accessed only by using the methods in the class.
- Encapsulated data is accessed using the “Accessor (getting)” and “Mutator (setter)” methods.
  - **Accessor** – methods to retrieve the hidden data.
  - **Mutators** – methods to change hidden data.

# When encapsulation used?

**Encapsulation** are used to create **value** or **transfer** objects.

Transfer objects are used to logically bundle the related data and transferring it to another objects.

**Example: EmployeeVO** – variable like **employeeid**, **salary** will be encapsulated and exposed using getters/setters.

**VO** stands for Value Object.

# Exercise - Encapsulation

Let us look at an example of encapsulation,

In our **Facebook** profile, we would like to hide the **age**, **contactNo** and **maritalStatus** information to the external world. This can be done using encapsulation where the fields are made private and can be accessed only by the accessor and mutator methods.

```
public class Facebook {  
    private int age;  
    private long contactNo;  
    private String maritalStatus;  
    public int getAge() {  
        return age;  
    }  
    public void setAge(int age) {  
        this.age = age;  
    }  
    public long getContactNo() {  
        return contactNo;  
    }  
    public void setContactNo(long contactNo) {  
        this.contactNo = contactNo;  
    }  
    public String getMaritalStatus() {  
        return maritalStatus;  
    }  
    public void setMaritalStatus(String maritalStatus) {  
        this.maritalStatus = maritalStatus;  
    }  
}
```

The public methods are access points to the fields of this class. Hence, any class that wants to access the variables should access them through these getters and setters

# Exercise - Encapsulation

The variables of the EncapsDemo can be accessed as follows,

```
public class EncapsDemo {  
    public static void main(String[] args) {  
        Facebook fb = new Facebook();  
        fb.setAge(21);  
        fb.setContactNo(9943155523L);  
        fb.setMaritalStatus("Single");  
  
        System.out.println("Marital status: "+fb.getMaritalStatus()+  
                           " Age: "+fb.getAge()+  
                           " Contact Number: "+fb.getContactNo());  
    }  
}
```

The fields are set using the setter (Mutator) methods.

The values are got using the getter (Accessor) methods.

# Returning values from a method

How to return from a method?

- A method returns a value using the **return** keyword.

Syntax:

```
return <return Value>;
```

Where,

- **<return Value>** denotes the variable whose value needs to be returned.
- The datatype of the “**return Value**” must match the data type specified in the method declaration.  
*i.e.* You cannot return an integer variable value from a method which is declared to return a boolean.
- **Returning control:** If the method execution needs to be stopped and the control needs to be sent back to the calling method simply use the return keyword without the variable.

Syntax: `return;`

# Return from a method

## Multiple Return Statements:

- A method can have multiple return statements.
- You can use constants as return values instead of variables below is an example of a code having multiple return statements.

```
public String getNumberInWords(int number) {  
    String defaultNumber = "zero";  
    if (number == 1) {  
        return "one";  
    } else if (number == 2) {  
        return "two";  
    }  
    return defaultNumber;  
}
```



A constant value is being returned



Returning a variable value

**NOTE:** It is **NOT** a good practice to have multiple return statements from a method. Let us look at a solution to fix this best practice violation in the next slide.



# Return from a method

**Best Practice:** A method should have only one return statement.

```
public String getNumbeInWords(int number) {  
    String defaultNumber = "zero";  
    if (number == 1) {  
        defaultNumber = "one";  
    } else if (number == 2) {  
        defaultNumber = "two";  
    }  
    return defaultNumber;  
}
```



Value stored in a variable and  
returned from a single point

# Exercise for Methods.

**Let us all develop a program with a method invocation.**

1. Create a Java class "Circle.java" with a method "calculateArea"
2. This method should accept radius as an argument, calculate the area of circle and return the result.
3. The main method should invoke the Circle object "calculateArea" method by passing a value for the radius, say 12.5
4. The main method should also print the area of circle (result variable)

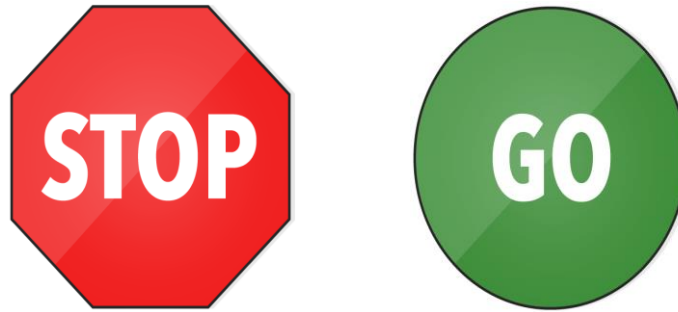
# Exercise - Solution

Solution:

```
public class Circle {  
    public static double calculateArea(double radius) {  
        return (Math.PI*radius*radius);  
    }  
    public static void main(String[] args) {  
        double result = calculateArea(12.5);  
        System.out.println("Area of Circle is "+result);  
    }  
}
```

---

# Time To Reflect



Trainees to reflect the following topics before proceeding.

- What is a method?
- What are the building block elements of the method?
- What is encapsulation?
- How is a variable encapsulated?
- How can we transfer the execution control to the calling method?
- Can we have multiple return statements from a method?

# Method Overloading

What is method overloading?

Let us recollect what we learnt in the Introduction to OOPs session.



```
Implementation #1  
makeSound(){  
    //default parameter  
    Bark woof woof  
}
```

```
Implementation #2  
makeSound(injured){  
    //Added input parameter  
    Make a whining sound  
}
```

Yes, You got It



**If not, go to the next slide for the definition !!**

# Method Overloading

What is method overloading?

- Two different versions of the same method available in the same class.
- This is done by either changing the input parameter or return type.

Example: The method “**add**” has been overloaded below,

```
void add() {  
  
    System.out.println("No parameters for this method");  
}  
  
void add(int a, int b, int c) {  
  
    int sum = a + b + c;  
    System.out.println("Sum of a+b+c is " + sum);  
}  
  
void add(String s1, String s2) {  
  
    String s = s1 + s2;  
    System.out.println(s);  
}
```

# Exercise – Method Overloading

Let us all create 4 overloaded methods for “test()” and invoke all versions of the overloaded methods.!!

```
public class OverloadDemo {  
  
    void test() {  
        System.out.println("No parameters");  
    }  
  
    // Overload test for one integer parameter.  
    void test(int a) {  
        System.out.println("a: " + a);  
    }  
  
    // Overload test for two integer parameters.  
    void test(int a, int b) {  
        System.out.println("a and b: " + a + " " + b);  
    }  
  
    // overload test for a double parameter  
    double test(double a) {  
        System.out.println("double a: " + a);  
        return a * a;  
    }  
}
```

# Exercise – Method Overloading

Create another class Overload .java which has a main method to call the overloaded methods in OverloadDemo.java

```
public class Overload {  
    public static void main(String args[]) {  
        OverloadDemo ob = new OverloadDemo();  
        double result;  
        // call all versions of the overloaded method test()  
        ob.test();  
        ob.test(10);  
        ob.test(10, 20);  
        result = ob.test(123.2);  
        System.out.println("Result of ob.test(123.2): " + result);  
    }  
}
```



# static keyword

Static keyword:

The static keyword is used before a method or variable (similar to an access modifier).

Examples:

```
variable: private static int x = 0;  
method: public static add(){  
        // some code here  
}
```

**Dictionary Definition of static:** Changeless, motionless

# static keyword

What is a static member?

- Static variables are global variables, which is shared by all the instances of the class.
- Static means that there will only be only one instance of that object in the class.
- It is not required to create an instance of the object to access the static method or variable.

## **Where is it used?**

Static variables are used in singleton implementation. You will learn more about this during the design pattern session.

# Exercise – static variable

Let us all understand static variables using an example. Develop the code and look at out static is different from a normal variable.

```
public class StaticVarDemo {  
    private int x = 5;  
    private static int y = 3;  
    public StaticVarDemo() {  
        x++;  
        y+=7;  
        System.out.println("X is "+x+" and Y is "+y);  
    }  
    public static void main(String[] args) {  
        StaticVarDemo s1 = new StaticVarDemo();  
        StaticVarDemo s2 = new StaticVarDemo();  
        StaticVarDemo s3 = new StaticVarDemo();  
    }  
}
```

- The output of the program is below,

```
X is 6 and Y is 10  
X is 6 and Y is 17  
X is 6 and Y is 24
```

Reason:

- Before s1 is created, X = 5 and Y = 3;
- When s1 is created, X = 6 and Y = 10 (as per logic inside constructor)
- When s2 is created, Y = 17 since the value of Y does not change with every instance. The value of Y is shared across all instances.
- But X = 6 (since a new object of X is created and incremented by 1.
- Similarly, when s3 is created, X = 6 and Y = 24

# What is a static method?

What is a static method?

A "static" method belongs to the class, not to any particular instance. In other words, static methods can be invoked using the class name

## Where is it used?

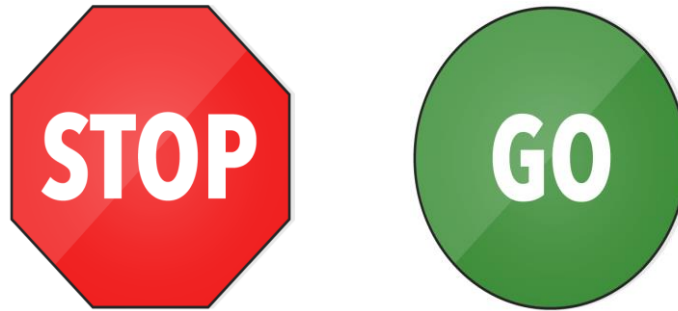
Static methods are used in singleton implementation. You will learn more about this during the design pattern session.

# Exercise – static method

Let us develop the following code to get accustomed to static method declaration and invocation. The static method `printMessage()` within class and `printData()` can be accessed directly using the class name.

```
class Test{
    public static void printData() {
        System.out.println("This is static other class.");
    }
}
public class StaticMethodDemo {
    public static void printMessage() {
        System.out.println("This is static within class.");
    }
    public static void main(String[] args) {
        printMessage();
        Test.printData();
    }
}
```

# Time To Reflect



Trainees to reflect the following topics before proceeding.

- What is method overloading?
- What are the ways by which you can overload methods?
- What is a static variable?
- How is a static variable invoked?

# Constructor

What is a constructor?

A constructor is a special method used for creating an object of a class and initializing its instance variables

- Constructors should have the same name as the class
- It should not have any return, not even void.
- It is not mandatory for the developer to write a constructor.
- If constructor is not defined by developer java will use the default constructor to create objects.



# Default Constructor

Default Constructor (no-args constructor):

- A constructor with no parameters
- If the class does not specify any constructors, then an implicit default constructor is automatically called by the JVM.

Syntax:

```
public class Employee {  
    public Employee() {  
        //code block  
    }  
}
```



**Default  
Constructor**

# Overloading Constructors

What is overloaded constructor?

A default constructor with one or more arguments is called a overloaded constructor

How to overload constructors?

Step 1: Create a method with the same name as the class name

Step 2: Do not provide any return type for the method created

Step 3: Add required number of arguments for the constructor (method) to create an overloaded constructor

Step 4: Any number of overloaded constructors can be created for a class.

# Exercise – How to overload constructor?

Lets all create this class and overload the constructors as illustrated.

```
public class Employee {  
    private int empId;  
    private String empName;  
    private String desig;  
    public Employee() {  
        System.out.println("Default constructor.");  
    }  
    public Employee(int empId) {  
        this.empId = empId;  
    }  
    public Employee(int empId, String empName) {  
        this.empId = empId;  
        this.empName = empName;  
    }  
    public Employee(int empId, String empName, String desig) {  
        this.empId = empId;  
        this.empName = empName;  
        this.desig = desig;  
    }  
    public static void main(String[] args) {  
        Employee e1 = new Employee(1201);  
        Employee e2 = new Employee(1202, "Ram");  
        Employee e3 = new Employee(1203, "Hari");  
  
        System.out.println("Id of First employee "+e1.empId);  
        System.out.println("Id of Second employee "+e2.empId);  
        System.out.println("Id of Third employee "+e3.empId);  
    }  
}
```

Default Constructor

Overloaded Constructors

Objects instantiated using different constructors

# “this” Reference

## The “this” keyword:

- “**this**” refer to the **current object** instance itself.
- It can be used for only instance variables and not for static or class variables
- It is used for also invoking the overloaded constructors.
- It is used to access the instance variables shadowed by the parameters in methods.  
Typically used with the encapsulated field.

Example: A method **calculateTax** has a parameter *salary*, also the class has the instance variable *salary*. In the method if we refer using “**this**” keyword it means we are referring to instance variable.

Syntax:

**this**.<name of instance variable>

**this**.<constructor Name><arguments>

# “this” reference - example

```
public class Shape {  
    private int x=0;  
    private int y=0;  
    public Shape() {  
        //code block  
    }  
    public Shape(int x,int y) {  
        this.x=x;  
        this.y=y;  
    }  
}
```

The member variables  
are being referred  
using the this key word.

# Chaining Constructor call using “this()”

## Chaining Constructor Calls:

- Constructor calls can be chained, which mean you can call another constructor from a constructor of the same class.
- You can use “this()” for invoking constructors from other constructor of a class.

There are a few things to remember when using this() method constructor call:

- When using “this()” constructor call, it must occur as the first statement in a constructor.
- The “this()” call can then be followed by any other statements.

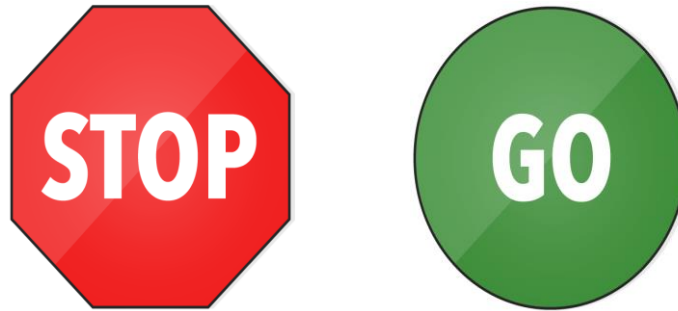
# “this()” Constructor call

Let us example of constructor chaining using “this()” reference.

```
public class Employee {  
    private String empName;  
    public Employee() {  
        this("Hari");  
    }  
    public Employee(String empName) {  
        this.empName = empName;  
    }  
    public static void main(String[] args) {  
        Employee e1 = new Employee();  
        System.out.println("Name of the employee is "+e1.empName);  
    }  
}
```

The overloaded constructor is being invoked by the default constructor using this()

# Time To Reflect



Trainees to reflect the following topics before proceeding.

- What is a constructor?
- How to overload constructor?
- Can a constructor have return value?
- What is chaining of constructors? How can it be implemented?



Thank you

*You have successfully completed*  
**Access Specifier,  
Constructors &  
Methods**