Methods in java

In this Lecture we Learn

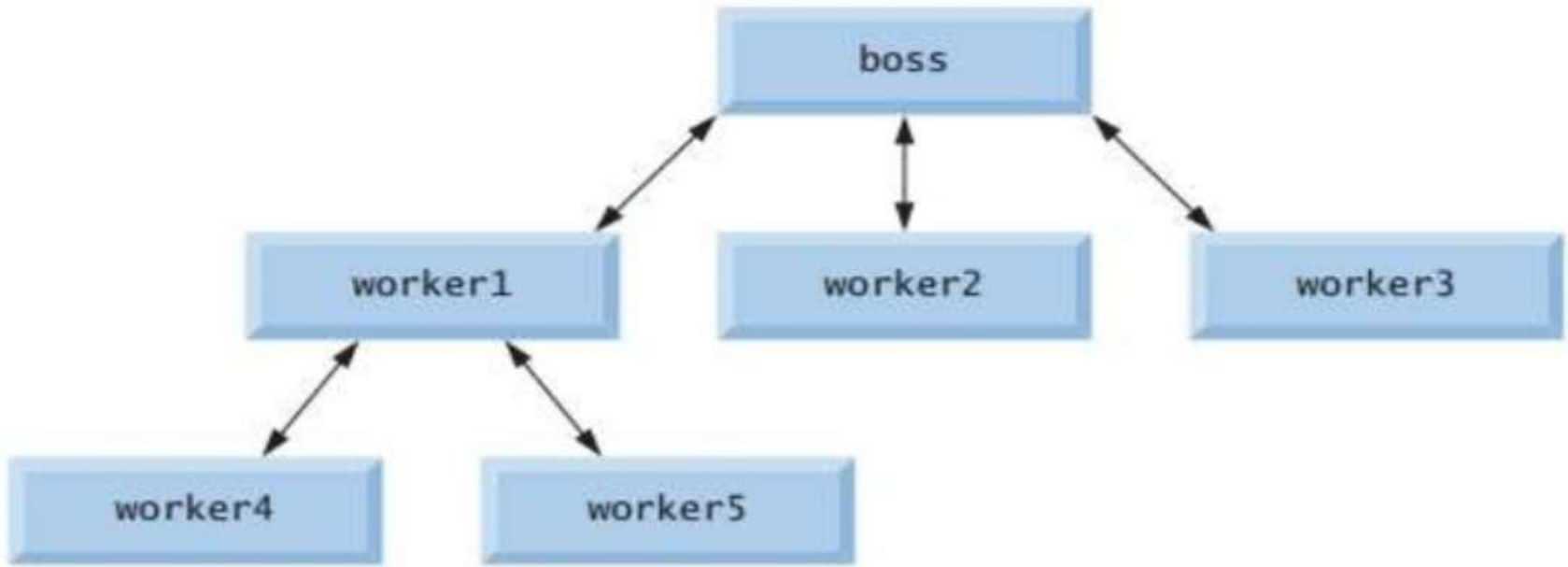
- **Methods in Java**

Divide And Conquer

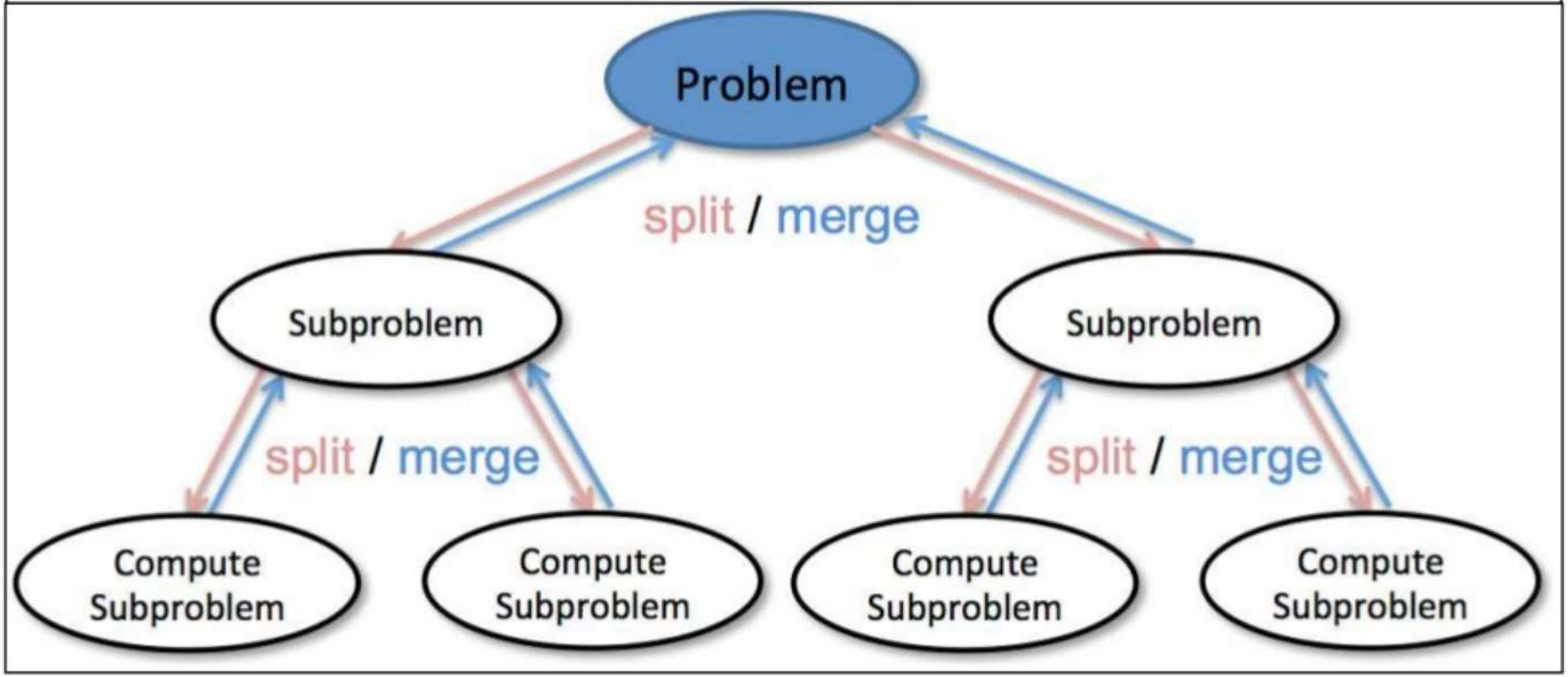
- The Top–down design approach is based on **dividing** the **main problem** into **smaller tasks which** may be **divided** into **simpler tasks**, then implementing **each simple** task by a subprogram or a **function**.
- This technique is called **divide and conquer**.
- **Example:** Construction of a house

Divide And Conquer

- Below figure shows the boss function communicating with several worker functions.
- The **boss function divides** the **responsibilities** among the worker functions, and worker1 acts as a " boss function" to worker4 and worker5.



Divide And Conquer



Methods are Used for Divide a Large Code into Modules

Introduction

- In the preceding Slides, we learned about such **methods** as System.**out**.println, Scanner Class methods like next(), nextInt().
- A **method** is a **collection** of **statements** that are **grouped** together to perform an **operation**.
- When you call the System.out.println method, for example, the system actually executes several statements in order to display a message.

Methods in Java

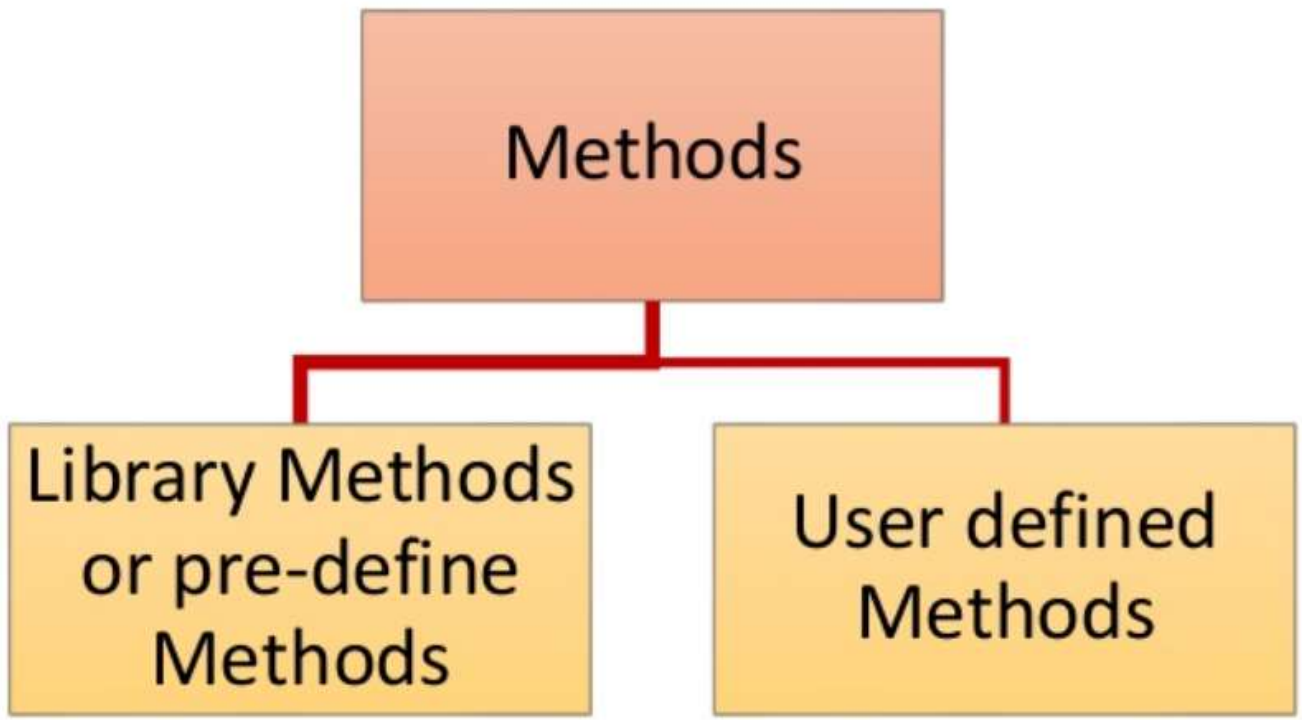
- A **method** is a **group** of **statements** that together **perform** a **specific task**. Every Java program has at least one function, which is main().
- **Why use Method ?**
- **Methods** are used for **divide** a **large code** into **module**, due to this we can **easily debug** and **maintain** the **code**. For example if we write a calculator programs at that time we can write every logic in a separate function (For addition sum(), for subtraction sub()). Any function can be called many times.

Advantages of Methods

- Methods **separate** the concept (what is done) from the **implementation** (how it is done).
- Methods make **programs easier** to understand.
- Methods can be called several times in the same program, allowing the **code to be reused**.
- Avoid **repeating of code**.
- Methods **reducing** code **complexity** and **increasing** its **maintainability**.

Type of Java Methods

- Java allows the use of both internal (user-defined) and external functions.



Pre-Defined Methods

- **Java** has a **library** of (pre-defined) classes and methods, organized in packages (e.g. java.util.Scanner).
- In order to use them, we "**import**" the packages (or classes individually).
- By default, the **java.lang** package is **automatically imported**, in any Java programs.
- In particular, classes String, Math and Character are included in java.lang.

Pre-Define-Main Method

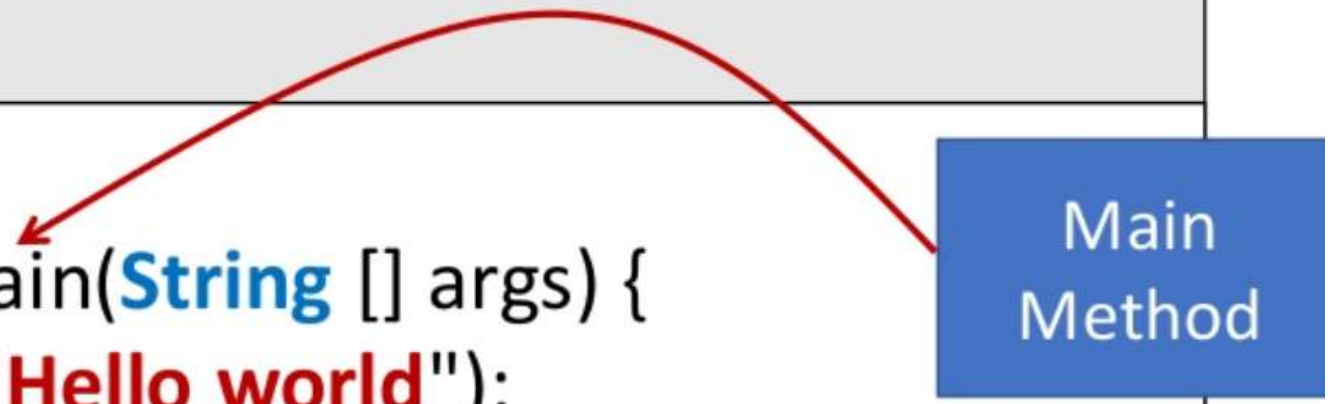
- **main()** is a special method where execution starts

```
public class Test {
public static void main(String [] args) {
System.out.println("Hello world");
    }
}
```

Pre-Define-Main Method

- **main()** is a special method where execution starts

```
public class Test {
public static void main(String [] args) {
System.out.println("Hello world");
    }
}
```



Pre-Defined Methods-Example

Method	Description
ceil(x)	Rounds x to the smallest integer not less than x
cos(x)	Trigonometric cosine of x (x in radians)
exp(x)	Exponential function e^x
fabs(x)	Absolute value of x
floor(x)	Rounds x to the largest integer not greater than x

To use these method, we **call the method** with the **class name(Math)**, followed by **.** and the **method name**, and appropriate **parameters** in parentheses.

Pre-Defined Methods-Example

```
14  public class Test {
15      public static void main(String[] args) {
16          double i=1;
17          double j=1.1;
18          double x=0;
19          double y=1.1;
20          System.out.println("The Exp of " + i + " is " + Math.exp(i));
21          System.out.println("The ceiling of " + j + " is " + Math.ceil(j));
22          System.out.println("The cos  " + x + " is " + Math.cos(x));
23          System.out.println("The floor of " + y + " is " + Math.floor(y));
24      }
25  }
```

test.Test

utput ×

JavaApplication5 (run) × | Test (run) #2 × | Debugger Console × | Test (run) ×

```
run:
The Exp of 1.0 is 2.718281828459045
The ceiling of 1.1 is 2.0
The cos  0.0 is 1.0
The floor of 1.1 is 1.0
```

Pre-Defined Methods-Example

```
public class Test {
   public static void main(String[] args) {
      double i=1;
      double j=1.1;
      double x=0;
      double y=1.1;
       System.out.println("The Exp of " + i + " is " + Math.exp(i));
       System.out.println("The ceiling of " + j + " is " + Math.ceil(j));
       System.out.println("The cos  " + x + " is " + Math.cos(x));
       System.out.println("The floor of " + y + " is " + Math.floor(y));
   }
}
```

Math Class

Method of Math Class

i is a Parameter of this Method

Output is:

The Exp of 1 is 2.718281
The ceiling of 1.1 is 2
The cos of 0 is 1

Limitations of Library Methods

- All predefined methods are contained limited task only that is for what purpose method is designed for same purpose it should be used.
- As a **programmer** we do **not** having any **controls** on **predefined method** implementation part is there in machine readable format.
- In implementation whenever a predefined method is not supporting user requirement then go for user defined method.

User Defined Method

- These Methods are created by programmer according to their requirement.
- For example suppose you want to create a method for add two number then you create a method with name **sum()** this type of method is called user defined method.
- Here if you don't like the name of the method, **sum()** then you can write any name as you want.
- You can write any name of user defined method as you like or according to your requirements.

Features of User Define Method in Java

- Using the user **defined method** model, we can set our own method as per **our requirement**.
- Method type may change as like return value types (int, char, double) method or only return type method like void.
- UDM (User Define Method) always accessed publicly, privately and protected formation.

Defining a Method

Syntax:

```
modifier returnType nameOfMethod (Parameter List)
{
  // method body
}
```

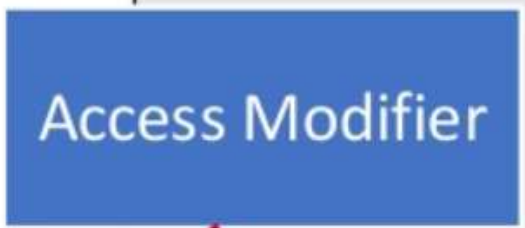
```
modifier returnType nameOfMethod (Parameter List)
{
  // method body
}
```

- The syntax shown above includes –
- **modifier** – It defines the access type of the method and it is optional to use.
- **returnType** – Method may return a value.
- **nameOfMethod** – This is the method name. The method signature consists of the method name and the parameter list.
- **Parameter List** – The list of parameters, it is the type, order, and number of parameters of a method. These are optional, method may contain zero parameters.
- **method body** – The method body defines what the method does with the statements.

User Define Method Example

```
public static int sum(int a, int b)
{
   // method body
}
```

User Define Method Example




```
public static int sum(int a, int b)
{
   // method body
}
```

User Define Method Example


```
public static int sum(int a, int b)
{
   // method body
}
```

User Define Method Example

```
public static int sum(int a, int b)
{
  // method body
}
```

- **public static** – modifier
- **int** – return type
- **sum** – name of the method
- **a, b** – formal parameters
- **int a, int b** – list of parameters

Body of the Method

- It consists of declaration and statements
- The **task of the method** is performed in the body of the method

```
void sum(int num1, int num2, int num3)
{
    int result = num1 + num2 + num3;
}
```

Method Calling

- In calling method write the name of the function and provide its arguments without data types
- In calling a method, return value type and data types of the arguments are not written

```
void sum(int num1, int num2, int num3)
{
    int result = num1 + num2 + num3;
}
```

Formal Parameters

```
sum(num1, num2, num3)
```

Actual Parameters

Example Add Two Integer Using Method

```
14    public class Test {
15        public static void main(String[] args) {
16            int z;
17            z=addition( 5 , 8);
18           System.out.println("Sum is: "+z);
19        }
20        public static int addition(int a, int b) {
21            int r;
22            r=a+b;
23            return r;
24        }
25
26    }
```

test.Test > main >

Output - Test (run)

```
run:
Sum is: 13
```

Example Add Two Integer Using Method

```
public class Test {
   public static void main(String[] args) {
      int z;
      z=addition( 5 , 8);
     System.out.println("Sum is: "+z);
   }
   public static int addition(int a, int b) {
      int r;
      r=a+b;
      return r;
   }
}
```

Calling of Method Passing two Arguments

Output is:
Sum is: 13

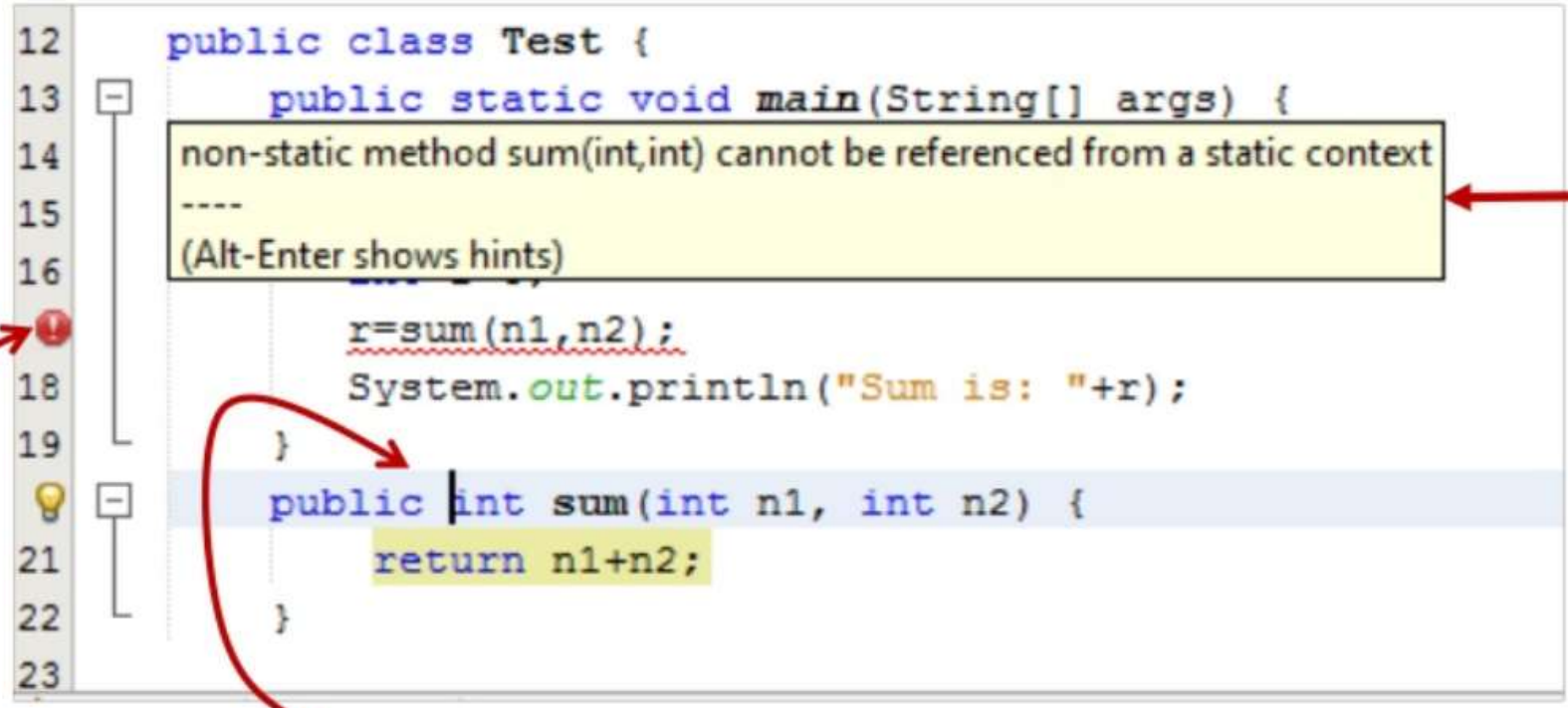
Addition Method return the sum of two Numbers

Method Addition of type public static int having two Parameters of type integer

The static keyword

- Java methods and variables can be declared static
- These exist **independent of any object**
- This means that a Class's
  - static methods can be called even if no objects of that class have been created and
  - static data is "shared" by all instances

Method Example without Static keyword



Compiler Error Message

Compiler Error

Non Statics Method Because we do use static keyword here

Problem Statement

- Write a Java program using function which accept two integers as an argument and return its sum. Call this function from main( ) and print the results in main( ).

Solution of Previous Problem

```
12  import java.util.Scanner;
13  public class Test {
14      public static void main(String[] vip) {
15      int n1; int n2; int r = 0;
16          Scanner input=new Scanner(System.in);
17          System.out.println("Please Enter First Number: ");
18          n1=input.nextInt();
19          System.out.println("Please Enter Second Number: ");
20          n2=input.nextInt();
21          r=sum(n1,n2,r);
22          System.out.println("Sum is: "+r);
23      }
24      public static int sum(int n1, int n2, int r) {
25                      r=n1+n2;
26          return n1+n2;
27      }
28  }
```

Output - Test (run)

```
run:
Please Enter First Number:
7
Please Enter Second Number:
3
Sum is: 10
```

Problem Statement

Write a Method which calculates the raise to power of a input base number up to given power number.

Solution of Previous Problem

```
14  import java.util.Scanner;
15  public class Test {
16      public static void main(String[] vip) {
17      double base; double power; double r ;
18          Scanner input=new Scanner(System.in);
19          System.out.println("Please Enter Base ");
20          base=input.nextInt();
21          System.out.println("Please Enter Power ");
22          power=input.nextInt();
23          r=raiseToPower(base,power);
24          System.out.println("The Result is: "+r);
25      }
26      public static double raiseToPower(double base, double power) {
27                  double result = 1.0;
28              for(int i=1; i<=power; i++) {
29                  result *= base;
30              }
31              return result;
32      }
33  }
```

test.Test > main > r >

Output - Test (run)

```
run:
Please Enter Base
2
Please Enter Power
5
The Result is: 32.0
```

Your Task..... ☺

- Write a Java program that lets the user **perform arithmetic operations** on two numbers. Your program must be menu driven, allowing the user to select the operation (+, -, *, or /) and input the numbers. Furthermore, your program must consist of following Methods:
- 1. **Method showChoice**: This Method shows the options to the user and explains how to enter data.
- 2. **Method add**: This Method accepts two number as arguments and returns sum.
- 3. **Method subtract**: This Method accepts two number as arguments and returns their difference.
- 4. **Method multiply**: This Method accepts two number as arguments and returns product.
- 5. **Method divide**: This Method accepts two number as arguments and returns quotient.

Java User-Defined Method Types

- Methods with no Argument and no return value
- Methods with no Argument but return value
- Methods with Argument but no return value
- Methods with Argument and return value

So For We only Learn about Methods having Arguments and return some values

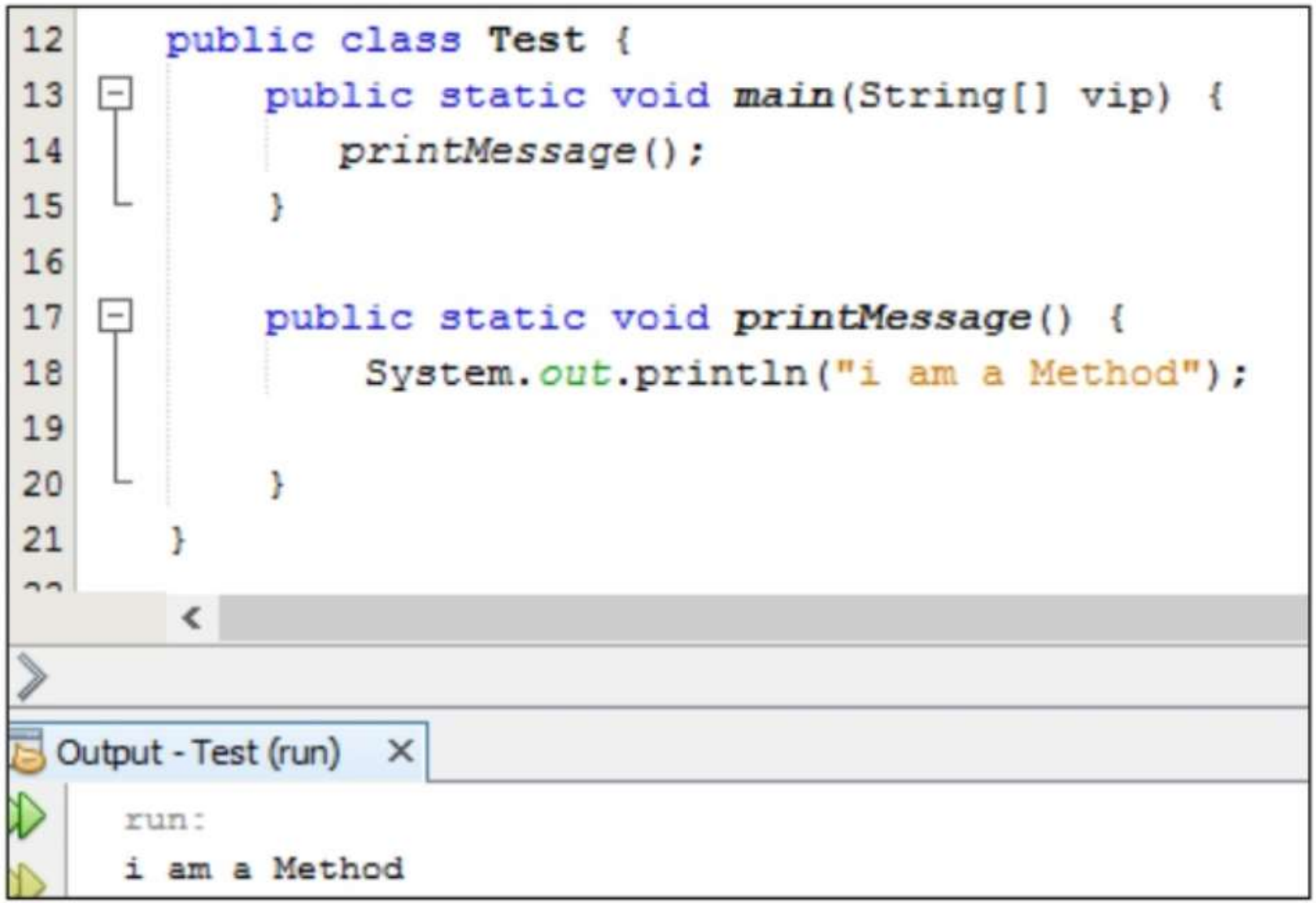
Methods with no Argument and no Return value

- The syntax shown below for Method:

```
type name ()
   {
      //statements
    }
```

- Requires the declaration to begin with a type. This is the type of the value returned by the function.
- But what if the function does not need to return a value? In this case, the type to be **used is void**

Methods With no Argument and no Return value



Please Note

- The * is a wildcard which means that you're importing all the classes in the particular package.

```
import java.util.*;
// Import all the classes in the util package.
```

- whereas in the second case, you've specified exactly which class you need to import.

```
import java.util.Scanner;
// import just the Scanner class from the util package and not others
```

Methods with Argument but no Return value

- **Note :**
  - Function accepts argument but it does not return a value back to the calling Program .
  - It is Single (One-way) Type Communication
  - Generally Output is printed in the Called function

Syntax is:

```
void methodName(argument1, argument2, …..)
{
    //body of method
}
```

Methods with Argument but no Return value

```
12  import java.util.*;
13  public class Test {
14      public static void main(String[] vip) {
15          double rad;double area;
16          Scanner input = new Scanner(System.in);
17          System.out.println("Please Enter Radius");
18          rad = input.nextDouble();
19          area(rad);
20      }
21      static void area(double rad)
22      {
23          double area;
24          area=Math.PI*rad*rad;
25          System.out.println("Area of Circle is: "+area);
26      }
27  }
```

test.Test > area > area

Output - Test (run)

```
run:
Please Enter Radius
3
Area of Circle is: 28.274333882308138
```

Methods with Argument but no Return value

```
import java.util.Scanner;
public class Test {
   public static void main(String[] vip) {
      double rad; double area;
      Scanner input = new Scanner(System.in);
      System.out.println("Please Enter Radius");
      rad = input.nextDouble();
      area(rad);
   }
   static void area(double rad)
   {
      double area;
      area=Math.PI*rad*rad;
      System.out.println("Area of Circle is: "+area);
   }
}
```

Output is:

Please Enter Radius
3
Area of Circle is: 28.2743338

Print output on screen

Methods with No Argument but Return Value

```
12  import java.util.Scanner;
13  public class Test {
14      public static void main(String[] vip) {
15          double result;
16          result=area();
17          System.out.println("Area of Circle is: "+result);
18      }
19      static double area()
20      {
21          double rad;double area;
22          Scanner input = new Scanner(System.in);
23          System.out.println("Please Enter Radius");
24          rad = input.nextDouble();
25          area=Math.PI*rad*rad;
26          return area;
27      }
28  }
```

Output - Test (run)

```
run:
Please Enter Radius
3
Area of Circle is: 28.274333882308138
```

Scope of Variable in Java

- **The scope**
  - determines where in the program the variable is accessible.
  - determines the lifetime of a variable or how long the variable can exist in memory.
  - The scope is determined by where the variable declaration is placed in the program.
- To simplify things, just think of the scope as anything between the curly braces {...}. The outer curly braces are called the outer blocks, and the inner curly braces are called inner blocks.

Scope of Variable in Java

- In simple term scope of variable mean visibility of the variable. variable can have:
  - class level scope
  - method level scope
  - block level scope
- Please note Global variables are globally accessible. Java does not support globally accessible variables due to following reasons:
  - The global variables breaks the referential transparency
  - Global variables creates collisions in namespace.

Scope of Variable in Java

- There is no concept of global variables in Java. Global variables are usually a design flaw. Your components should be self-contained and should not need any global state. Instead, use private static fields.
- We can achieve the same mechanism like this:
- To define Global Variable you can make use of static Keyword

```
public class Test
 {    public static int i;
      public static int j;
}
```

- Now we can access a and b from anywhere by calling

```
Example.i;
Example.j;
```

Scope of Variable in Java

Following diagram will give more clear understanding about scope of variable.

Scope of Variable in Java

Block level Scope:

- Block level scope mean variable is visible in that block only.
- Variables or integers declared inside block are used inside block .

- **Example:**

```
{
    int number=1;
    System.out.println("number is:"+number);
}
    System.out.println("number is:"+number);  //Error
```

Scope of Variable in Java

Block level Scope:

- Block level scope mean variable is visible in that block only.
- Variables or integers declared inside block are used inside block .

- **Example:**

This is a Block

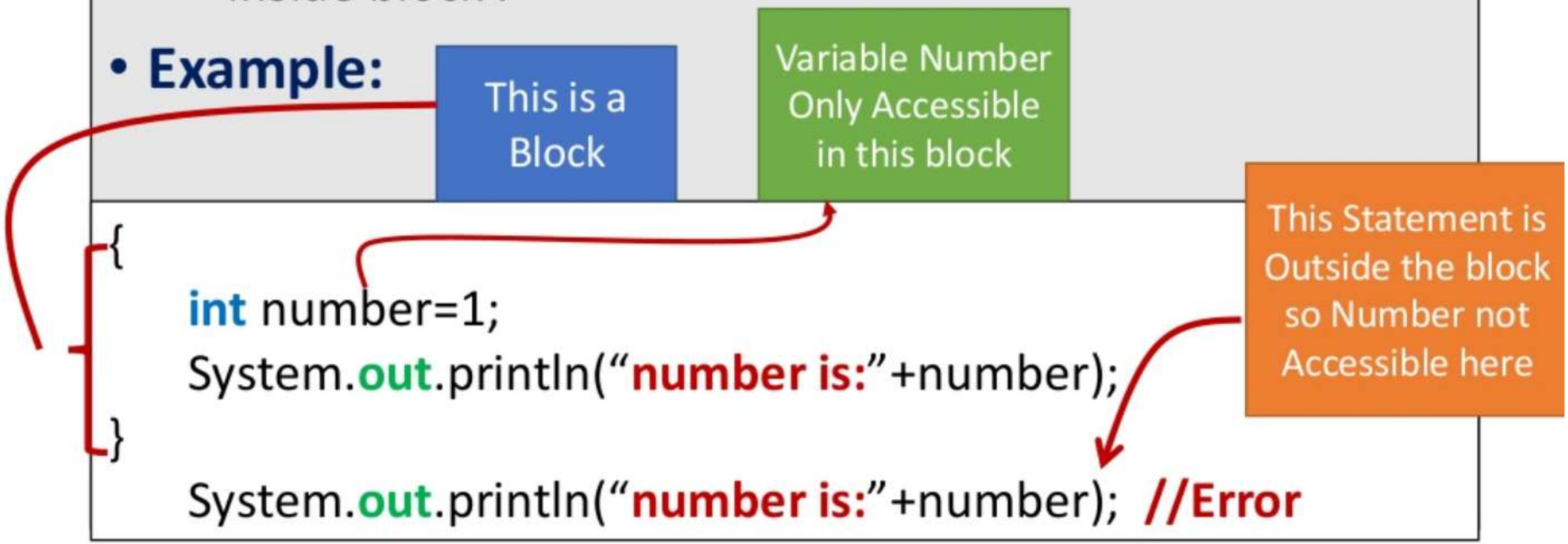
```
{  //block level scope start
    int number=1;
    System.out.println("number is:"+number);
} //block level scope ended
    System.out.println("number is:"+number);  //Error
```

Scope of Variable in Java

Block level Scope:

- Block level scope mean variable is visible in that block only.
- Variables or integers declared inside block are used inside block .

- **Example:**



Scope of Variable in Java

Method Level Scope

- Method level scope mean variable is visible on that method only.
- Declaring variables inside a function can be used in the whole function.

- **Example**

```
System.out.println("Result : "+number);   //Error
public static int square()
{
    int number = 0;
    return number * number;
}
```

Scope of Variable in Java

Global Level Scope

- Global scope mean variable is visible and accessible in entire program.
- Anything identified or declared outside of any function is visible to all functions in that file.

- **Example**

```
static int number=10;
   public static void main(String[] vip) {

    System.out.println("Result: "+number);  //ok
   }
```

Please Note

- We **cannot** define **different variables** with the **same name** inside the **same scope.**
- **Example:**

```
public class Test {
   public static void main(String[] vip) {

         int number=10;
         double number;    //Error
   }
}
```

Non-Overlapping (or disjoint) Scopes

- Disjoint scopes = 2 or more scopes that do *not* overlap with each other .
- **Example**

```
public class Test {
   public static void main(String[] vip) {
      {
         int number=10;
      }
         int number;  //ok
   }
}
```

Non-Overlapping (or disjoint) Scopes

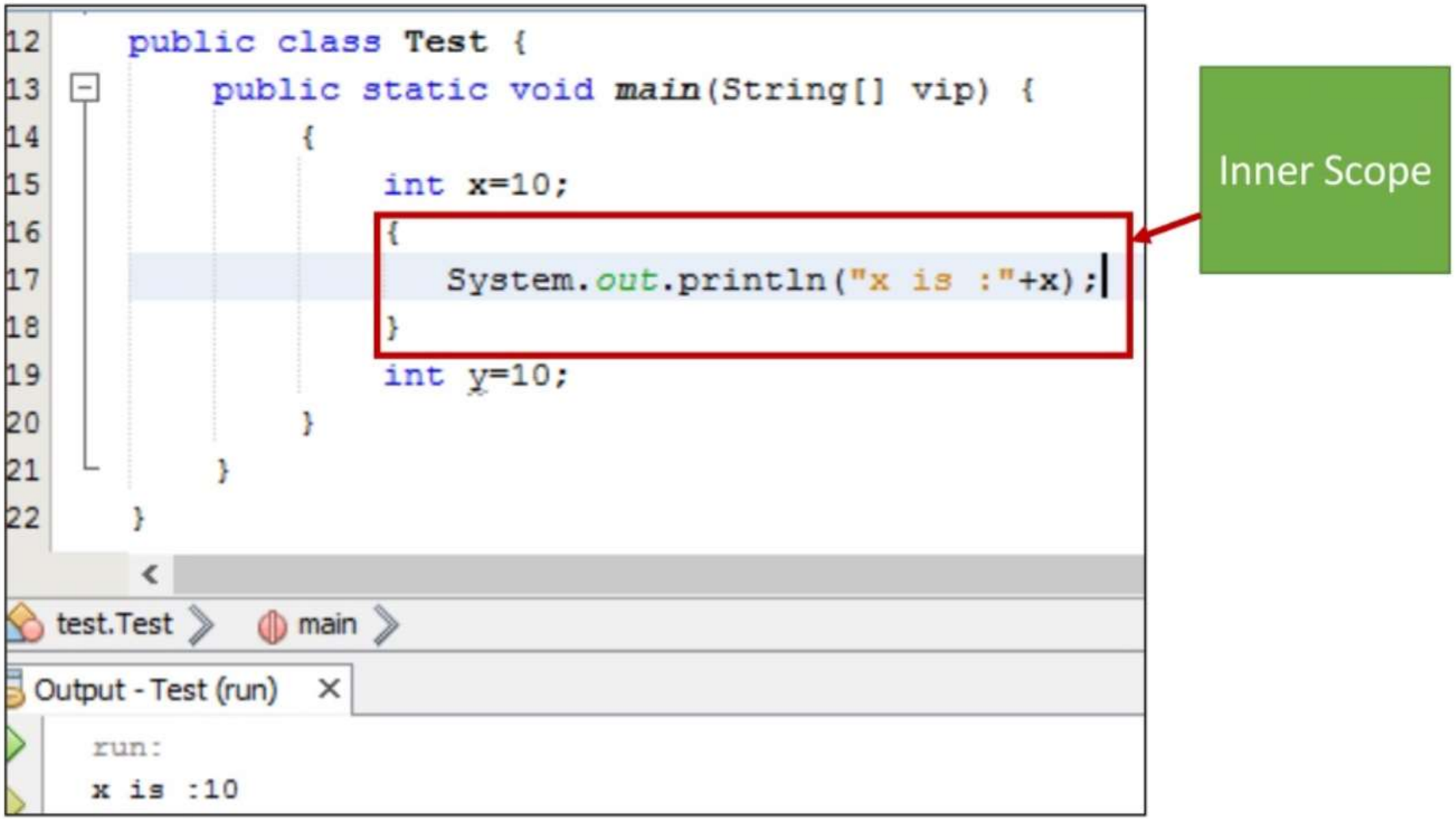
- Disjoint scopes = 2 or more scopes that do *not* overlap with each other .
- **Example**

```
public class Test {
   public static void main(String[] vip) {
      {
         int number=10;
      }
         int number;      //ok
   }
}
```

2 Different Variable with the Same Name in Different Scope

No Error here Because both Variable are in Different Scope

Scoping Rule for Nested Scope



Call-by-Value and Call-by-Reference

- There are two ways to pass an argument to a method:

- **call-by-Value** : In this approach copy of an argument value is pass to a method. Changes made to the argument value inside the method will have no effect on the arguments.
- **call-by-Reference** : In this reference of an argument is pass to a method. Any changes made inside the method will affect the argument value.

Call-by-Value and Call-by-Reference

Call By Value

- Passing a value held in the variable as an argument to the method.
- The value is copied to method parameters and changes takes place to that parameters.
- That is why changes made to the variable within the method had no effect on the variable that was passed.

Call-by-Value and Call-by-Reference

Call By Reference

- The Object is passed as an argument to the method.
- No copy of object is made here.
- Changes made in the method will be reflected in the original object

Example of Pass by Value in Java

- Suppose we have a method that is named "Receiving" and it expects an integer to be passed to it:

```
public void Receiving (int var)
{
  var = var + 2;
}
```

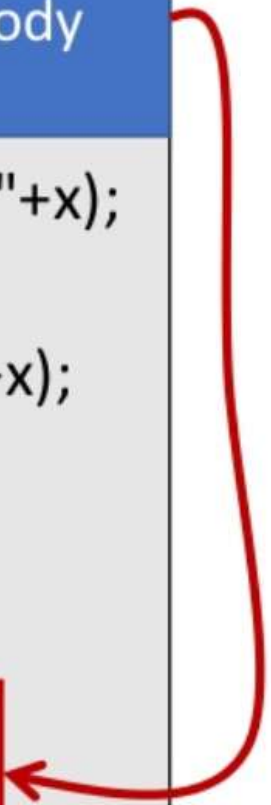
- Note that the "var" variable has 2 added to it. Now, suppose that we have some code which calls the method Receiving:

Explanation of Previous Program

```
class Test {
  public static void main ( String[] args ) {
    int x =3;
    System.out.println("Value of x before calling increment() is "+x);
    increment(x);
    System.out.println("Value of x after calling increment() is "+x);
  }

  public static void increment ( int a ) {
    System.out.println("Value of a before incrementing is "+a);
    a= a+1;
    System.out.println("Value of a after incrementing is "+a);
  }
}
```

Method Body



```
//create an object by passing in a name and age:
PersonClass variable1 = new PersonClass("Adil", 21);

PersonClass variable2;

// Both variable2 and variable1 now both name the same object
variable2 = variable1;

/*this also changes variable1, since variable 2 and variable1
  name the same exact object: */

variable2.set("Hina", 20);
System.out.println(variable1);
```

Output is:
Hina 20

- Let's just assume that System.out.println method above will print the name and age of a PersonClass object – even though this is not correct, it does keep the example simple and easy to understand. So, if we run the code above, the output will be:

Method Overloading in Java

- if a class have multiple methods by same name but different parameters, it is known as **Method Overloading**.
- **Why Method Overloading**
- Suppose we have to **perform addition** of given number but there can be any number of arguments, if we write method such as **a(int, int)**for two arguments, **b(int, int, int)** for three arguments then it is very **difficult** for you and other programmer to understand purpose or behaviors of method they can not identify **purpose of method**. So we use method overloading to easily figure out the program.
- For example two methods we can write sum(int, int) and sum(int, int, int) using method overloading concept.
- **Advantage of Method Overloading**
  - Method overloading increases the readability of the program.

Method Overloading in Java

- **Different Ways to Overload the Method**
- There are three ways to overload the method in java
  - By **changing number** of arguments or **parameters**
  - By **changing** the **data type**
  - By **changing Sequence** of Data type of **parameters**.
- **Method Overloading** is also known as **Static Polymorphism**.
- **Points to Note:**
  1. **Static Polymorphism** is also known as compile time binding or early binding.
  2. **Static binding** happens at compile time. Method overloading is an example of static binding where binding of method call to its definition happens at Compile time.

Method Overloading in Java

- **Different Ways to Overload the Method**
- There are three ways to overload the method in java
  - By **changing number** of arguments or **parameters**
  - By **changing** the **data type**
  - By **changing Sequence** of Data type of **parameters**.
- **Method Overloading** is also known as **Static Polymorphism**.
- **Points to Note:**
  1. **Static Polymorphism** is also known as compile time binding or early binding.
  2. **Static binding** happens at compile time. Method overloading is an example of static binding where binding of method call to its definition happens at Compile time.

Rules for Overloading

- Overloading can appear in the same class or a subclass
- Overloaded methods MUST change its number of argument or its type.
  - When declaring two or more methods in same name complier differentiate them by its arguments, so you cannot declare two methods with the same signature
- Overloaded methods CAN have different return type.
  - The compiler does not consider return type when differentiating methods, so it's legal to change return type of overloaded method
- Overloaded methods CAN change the access modifier.
  - Similarly overloaded method can have different access modifiers also.

Different Ways to Overload the Method

- **1.By Changing Number of Arguments or Parameters**
  - Overloaded method should have different argument

```
public class Test {
    public void add(int i, int j)
    {
    }
    public void add(int i)
    {
    }
}
```

Method Overloading by Changing Number of Arguments or Parameters

```
13    class Test
14            {
15            void sum(int a, int b)
16            {
17            System.out.println("Sum of Two Number: "+(a+b));
18            }
19            void sum(int a, int b, int c)
20            {
21            System.out.println("Sum of Three Number: "+(a+b+c));
22            }
23        public static void main(String args[])
24        {
25            Test obj=new Test();
26            obj.sum(10, 20);
27            obj.sum(10, 20, 30);
28        }
29    }
```

test.Test > main >

Output - Test (run)

```
run:
Sum of Two Number: 30
Sum of Three Number: 60
```

Method Overloading by Changing Number of Arguments or Parameters

```
class Test {
        void sum(int a, int b) {
        System.out.println("Sum of Two Number: "+(a+b));
        }
        void sum(int a, int b, int c) {
        System.out.println("Sum of Three Number: "+(a+b+c));
        }
   public static void main(String args[]) {
        Test obj=new Test();
        obj.sum(10, 20);
        obj.sum(10, 20, 30);
   }
}
```

Number of Parameters are Matched with Number of Arguments

Compiler can easily Differentiate which Method is to be called Depending upon the Number of Parameters

Different Ways to Overload the Method

- **2. Method Overloading by Changing data type of Arguments**
  - It's legal for Overloaded method to change only its type

```
public class Test {
    public void square(int i) {

    }
    public void square(float i) {

    }
}
```

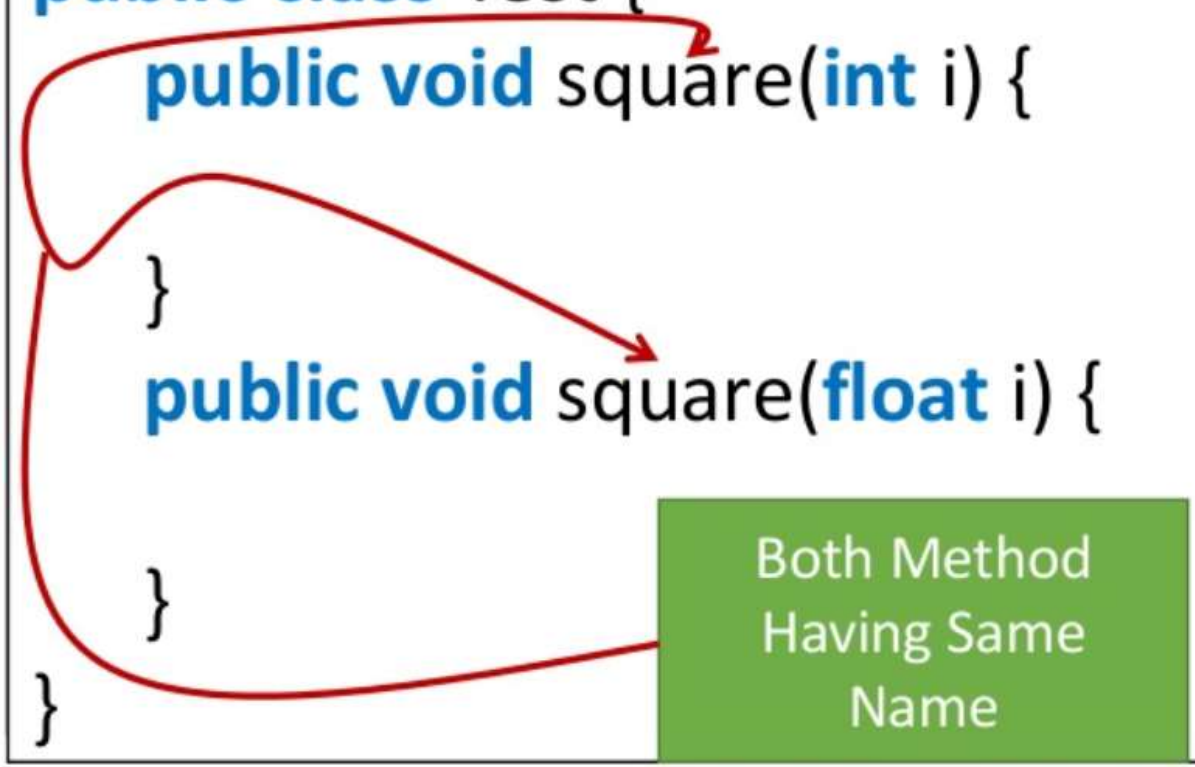
Different Ways to Overload the Method

- **2. Method Overloading by Changing data type of Arguments**
  - It's legal for Overloaded method to change only its type

```
public class Test {
    public void square(int i) {

    }
    public void square(float i) {

    }
}
```

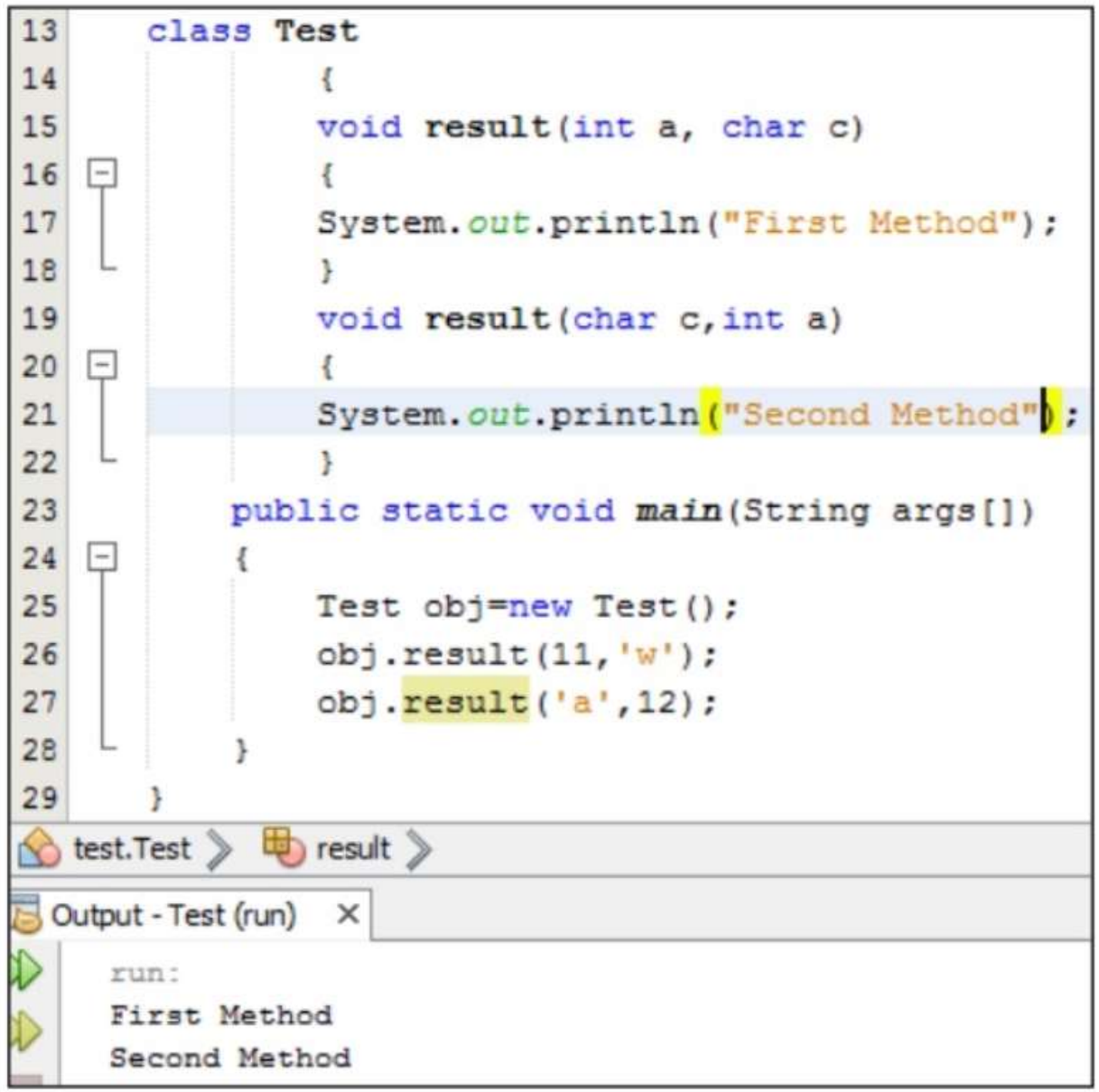


Different Ways to Overload the Method

- **3. Method Overloading By Changing the Sequence of Data type of Parameters.**

```
public class Test {
    public void result(int x, char c)
    {
    }
    public int result(char c, int x)
    {
    }
}
```

Method Overloading By Changing the Sequence of Data type of Parameters



```
13  class Test
14          {
15          void result(int a, char c)
16          {
17          System.out.println("First Method");
18          }
19          void result(char c,int a)
20          {
21          System.out.println("Second Method");
22          }
23      public static void main(String args[])
24      {
25          Test obj=new Test();
26          obj.result(11,'w');
27          obj.result('a',12);
28      }
29  }
```

test.Test > result >

Output - Test (run) ×

```
run:
First Method
Second Method
```

Method Overloading By Changing the Sequence of Data type of Parameters

```
class Test {
        void result(int a, char c) {
        System.out.println("First Method");
        }
        void result(char c,int a) {
        System.out.println("Second Method");
        }
   public static void main(String args[]) {
        Test obj=new Test();
        obj.result(11,'w');
        obj.result('a',12);
   }
}
```

Output is:

First Method

Second Method

Few Valid/Invalid Cases of Method Overloading

- **Case No 1:**

```
int myMethod(int a, int b, float c)

int myMethod(int var1, int var2, float var3)
```

- **Result:** Compile time error. Argument lists are exactly same. Both methods are having same number, data types and same sequence of data types in arguments.

Few Valid/Invalid Cases of Method Overloading

- **Case No 2:**

```
int myMethod(int a, int b)

int myMethod(float var1, float var2)
```

- **Result:** Perfectly fine. Valid case for overloading. Here data types of arguments are different.

Few Valid/Invalid Cases of Method Overloading

- **Case No 3:**

```
int myMethod(int a, int b)

int myMethod(int num)
```

- **Result:** Perfectly fine. Valid case for overloading. Here number of arguments are different.

Few Valid/Invalid Cases of Method Overloading

- **Case No 4:**

```
float myMethod(int a, float b)

float myMethod(float var1, int var2)
```

- **Result:** Perfectly fine. Valid case for overloading. Sequence of the data types are different, first method is having (int, float) and second is having (float, int).

Few Valid/Invalid Cases of Method Overloading

- **Case No 5:**

```
int myMethod(int a, int b)

float myMethod(int var1, int var2)
```

- **Result:** Compile time error. Argument lists are exactly same. Even though return type of methods are different, it is not a valid case. Since return type of method doesn't matter while overloading a method in Java.

Few Valid/Invalid Cases of Method Overloading

- **Case No 6:**

```
public int myMethod(long i)

private void myMethod(char c)
```

- Is this a Valid Case or not..????

Why Method Overloading is not Possible by Changing the return type of Method?

- In java, method overloading is not possible by changing the return type of the method because there may occur ambiguity.

- When two methods differ in their return type but not in parameters. Compiler cannot judge which is to be called depending on the return types; judges only on the parameter list.

Can We Overload Main() Method?

- Yes, We can overload main() method.
- A Java class can have any number of main() methods. But run the java program, which class should have main() method with signature as "public static void main(String[] args).
- If we do any modification to this signature, compilation will be successful. But, not run the java program. we will get the run time error as main method not found.

Example of Override Main() Method

```
public class mainclass {
    public static void main(String[] args) {
        System.out.println("Execution starts from Main()");
    }
    void main(int args){
        System.out.println("Override main()");
    }
    double main(int i, double d){
        System.out.println("Override main()");
        return d;
    }
}
```