



SNS COLLEGE OF TECHNOLOGY



Coimbatore-35.

An Autonomous Institution

**Accredited by NBA – AICTE and Accredited by NAAC – UGC with ‘A++’ Grade
Approved by AICTE, New Delhi & Affiliated to Anna University, Chennai**

**DEPARTMENT OF COMPUTER SCIENCE ENGINEERING
COURSE CODE & NAME : 23CST205 - Object Oriented Programming Using Java**

II YEAR/ III SEMESTER

UNIT – II INTRODUCTION TO JAVA

Topic: BASICS OF JAVA PROGRAMMING- OVERLOADING



Java Method Overloading



Method Overloading

With **method overloading**, multiple methods can have the same name with different parameters:

Example

```
int myMethod(int x)
float myMethod(float x)
double myMethod(double x, double y)
```



Java Method Overloading



Consider the following example, which have two methods that add numbers of different type:

Example

```
static int plusMethodInt(int x, int y) {  
    return x + y;  
}  
  
static double plusMethodDouble(double x, double y) {  
    return x + y;  
}  
  
public static void main(String[] args) {  
    int myNum1 = plusMethodInt(8, 5);  
    double myNum2 = plusMethodDouble(4.3, 6.26);  
    System.out.println("int: " + myNum1);  
    System.out.println("double: " + myNum2);  
}
```

Output:

```
int: 13  
double: 10.559999999999999
```



Java Method Overloading



Instead of defining two methods that should do the same thing, it is better to overload one.

In the example below, we overload the `plusMethod` method to work for both `int` and `double` :

Example

```
static int plusMethod(int x, int y) {  
    return x + y;  
}  
  
static double plusMethod(double x, double y) {  
    return x + y;  
}  
  
public static void main(String[] args) {  
    int myNum1 = plusMethod(8, 5);  
    double myNum2 = plusMethod(4.3, 6.26);  
    System.out.println("int: " + myNum1);  
    System.out.println("double: " + myNum2);  
}
```

Output:

```
int: 13  
double: 10.559999999999999
```

Note: Multiple methods can have the same name as long as the number and/or type of parameters are different.



Java Scope



Java Scope

In Java, variables are only accessible inside the region they are created. This is called **scope**.

Method Scope

Variables declared directly inside a method are available anywhere in the method following the line of code in which they were declared:

Example

```
public class Main {  
    public static void main(String[] args) {  
  
        // Code here CANNOT use x  
  
        int x = 100;  
  
        // Code here can use x  
        System.out.println(x);  
    }  
}
```

Output:
100



Java Scope



Block Scope

A block of code refers to all of the code between curly braces `{}`. Variables declared inside blocks of code are only accessible by the code between the curly braces, which follows the line in which the variable was declared:

Example

```
public class Main {  
    public static void main(String[] args) {  
  
        // Code here CANNOT use x  
  
        { // This is a block  
  
            // Code here CANNOT use x  
  
            int x = 100;  
  
            // Code here CAN use x  
            System.out.println(x);  
  
        } // The block ends here  
  
        // Code here CANNOT use x  
  
    }  
}
```

Output:

100



Java Recursion



Java Recursion

Recursion is the technique of making a function call itself. This technique provides a way to break complicated problems down into simple problems which are easier to solve.

Recursion may be a bit difficult to understand. The best way to figure out how it works is to experiment with it.

Recursion Example

Adding two numbers together is easy to do, but adding a range of numbers is more complicated. In the following example, recursion is used to add a range of numbers together by breaking it down into the simple task of adding two numbers:



Java Recursion



Example

Use recursion to add all of the numbers up to 10.

```
public class Main {  
    public static void main(String[] args) {  
        int result = sum(10);  
        System.out.println(result);  
    }  
    public static int sum(int k) {  
        if (k > 0) {  
            return k + sum(k - 1);  
        } else {  
            return 0;  
        }  
    }  
}
```

Output:

55



Java Recursion



Example Explained

When the `sum()` function is called, it adds parameter `k` to the sum of all numbers smaller than `k` and returns the result. When `k` becomes 0, the function just returns 0. When running, the program follows these steps:

```
10 + sum(9)
10 + ( 9 + sum(8) )
10 + ( 9 + ( 8 + sum(7) ) )
...
10 + 9 + 8 + 7 + 6 + 5 + 4 + 3 + 2 + 1 + sum(0)
10 + 9 + 8 + 7 + 6 + 5 + 4 + 3 + 2 + 1 + 0
```

Since the function does not call itself when `k` is 0, the program stops there and returns the result.

