Anu Arora, Assistant Professor, GNA University

# JAVA METHODS & CONSTRUCTORS

Java Programming

# Methods in JAVA

- In Java, the word **method** refers to the same kind of thing that the word *function* is used for in other languages.
- Specifically, a **method** is a function that belongs to a class.
- In Java, *every* function belongs to a class.

# Functions/Methods

A **function** is a reusable portion of a program, sometimes called *procedure* or *subroutine*.

- Like a mini-program (or *subprogram*) in its own right
- Can take in special inputs (arguments)
- Can produce an answer value (return value)
- Similar to the idea of a *function* in mathematics

# Why write and use functions?

- Divide-and-conquer
  - Can breaking up programs and algorithms into smaller, more manageable pieces
  - This makes for easier writing, testing, and debugging
  - Also easier to break up the work for team development
- Reusability
  - Functions can be called to do their tasks anywhere in a program, as many times as needed
  - Avoids repetition of code in a program
  - Functions can be placed into libraries to be used by more than one "program

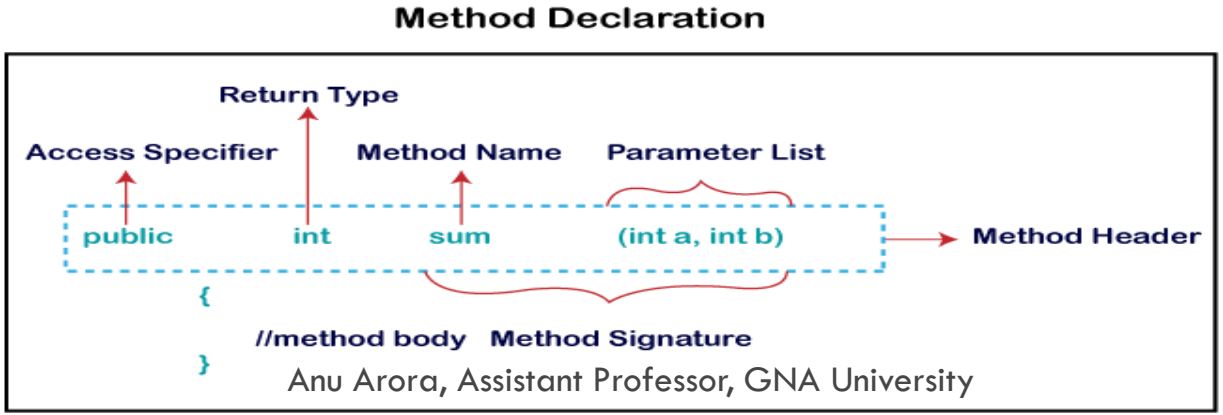
# major points of view

**With methods (functions), there are 2 major points of view**

- **Builder** of the method -- responsible for creating the *declaration* and the *definition* of the method (i.e. how it works)
- **Caller** -- somebody (i.e. some portion of code) that *uses* the method to perform a task

# Method Declaration

- The method declaration provides information about method attributes, such as visibility, return-type, name, and arguments.
- It has six components that are known as **method header**, as we have shown in the following figure.



# Method Signature & Access Specifier

- **Method Signature:**

  Every method has a method signature. It is a part of the method declaration. It includes the **method name** and **parameter list.**

- **Access Specifier:**

  Access specifier or modifier is the access type of the method. It specifies the visibility of the method.

# Types of access specifier

Java provides **four** types of access specifier:

- **Public:** The method is accessible by all classes when we use public specifier in our application.
- **Private:** When we use a private access specifier, the method is accessible only in the classes in which it is defined.
- **Protected:** When we use protected access specifier, the method is accessible within the same package or subclasses in a different package.
- **Default:** When we do not use any access specifier in the method declaration, Java uses default access specifier by default. It is visible only from the same package only.

# Terminology

- **Return Type:** Return type is a data type that the method returns. It may have a primitive data type, object, collection, void, etc. If the method does not return anything, we use void keyword.
- **Method Name:** It is a unique name that is used to define the name of a method. It must be corresponding to the functionality of the method. Suppose, if we are creating a method for subtraction of two numbers, the method name must be **subtraction().** A method is invoked by its name.
- **Parameter List:** It is the list of parameters separated by a comma and enclosed in the pair of parentheses. It contains the data type and variable name. If the method has no parameter, left the parentheses blank.
- **Method Body:** It is a part of the method declaration. It contains all the actions to be performed. It is enclosed within the pair of curly braces.

# Types of Methods

There are two types of methods in Java:

- Predefined Method
- User-defined Method

# Predefined Method

- In Java, predefined methods are the method that is already defined in the Java class libraries is known as predefined methods.
- It is also known as the **standard library method** or **built-in method**.
- We can directly use these methods just by calling them in the program at any point.
- Some pre-defined methods are **length()**, **equals()**, **pow()**, **sqrt()**, etc

# User-defined Method

- The method written by the user or programmer is known as **a user-defined** method.
- These methods are modified according to the requirement.

# Creating Method-Method I

Considering the following example to explain the syntax of a method −

**Syntax:** public static int methodName(int a, int b) { // body } Here,

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

# Creating Method-Method II

Method definition consists of a method header and a method body.

**Syntax:** 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.

# Calling a Method/Function in Java

- To access or to use a method, we need to call it. A function is called (or invoked, or executed) by providing the function name, followed by the parameters being enclosed within parentheses.
- When the program invokes any method, the program control automatically transfers to the function, the statements in the function body are executed, and then the control again returns to the calling code in any of the 3 situations:
  I. It completes all the statements in the method.
  II. It reaches a return statement of the method.
  III. If the method throws an exception.

# Actual and Formal Parameters

- The parameters that appear in the function definition are called **formal parameters.**
- The parameters that appear in the function call statement are called **actual parameters.**

# Example

```
public class MethodDemo
{
//function definition
public int getArea(int x, int y)
//x and y are formal parameters
{
return x * y;
}
```

```
public static void main(String args[])
{
int length = 10, width = 5, area =0 ;
MethodDemo demo = new MethodDemo();
//Calling a function
area = demo.getArea(length, width);
//length and width are actual parameters
System.out.println("The Area is: " +area);
}
}
```

# Arguments to Functions/Methods in Java

When you pass arguments to functions, you can pass any value of a legal Java data type. That is, arguments to functions can be:

- **Primitive data types:** char, byte, short, int, long, float, double, boolean.
- **Reference data types:** objects or arrays.

# Categories of function definition/call

- Without Parameter without Return type
- Without Parameter with Return type
- With Parameter without Return type
- With Parameter with Return type

# Call by Value and Call by Reference

We can invoke or call a function in two manners:

- **Call by Value**
- **Call by Reference.**

Basically, these two ways of invoking functions are also known as **Pass by Value** and **Pass by Reference**, because they depict the way of passing arguments to functions.

# Pass By Value(Call by Value)

- In the call by value method, the value of actual parameters gets copied into the formal parameters, that is, the function creates its own copy of argument values and then uses them.
- In the call by value, the changes are not reflected back to the original values.

# Example

```
public class CallByValue
{
public static int change( int a)
{
a = 20;
System.out.println("Inside the method
    change(), value is now changed
    to " +a);
return a;
}
public static void main(String[] args)
{
int original = 10;
System.out.println("The original value
    is: " + original);
change(original);
System.out.println("The value after
    execution of function change() is:
    " + original);
}}
```

# Pass By Reference(Call by Reference)

- In the call by reference method, the called function creates a new set of variables and copies the value of arguments into them. Instead of passing a value to the function, we pass a reference to the original variable. The reference stores a memory location of a variable.
- In the call by reference, the called method does not create its own copy of original values rather it refers to the original values, by different names (references).
- In the call by reference method, the changes are reflected back to the original values.

# Example

```
public class CallByReference
{
public static int original = 7;
public static void change(
    CallByReference obj)
{
obj.original = 20;
System.out.println("The Value inside
    change method: " +obj.original);
}
public static void main(String[] args)
{
System.out.println("The initial value is: " +
    original);
//Creating a object or a reference
CallByReference object = new
    CallByReference();
//Passing a reference to the method
change(object);
System.out.println("The value after execution
    of function change() is:" + original);
}}
```

# Memory Allocation for Method Calls

- A stack is used to implement the method calls. A stack frame is created within the stack area whenever we call or invoke a method.
- After that, the local variables, the arguments passed to the method and value which is returned by this method, all are stored in this stack frame.
- This allocated stack frame gets deleted when the called method gets executed.

# Method/function Overloading in Java

- When there are two or more than two methods in a class that have the same name but different parameters, it is known as method overloading.
- Java allows a function to have the same name if it can distinguish them by their number and type of arguments.
  I. float divide(int a, int b){...}
  II. float divide( float x, float y){...}
  III. float divide (float a, int b) {...}

# Recursion

- Recursion is a process by which a function or a method calls itself again and again.
- This function that is called again and again either directly or indirectly is called the “recursive function”.

# Recursion

**Any method that implements Recursion has two basic parts:**

- Method call which can call itself i.e. recursive
- A precondition that will stop the recursion.

Note that a precondition is necessary for any recursive method as, if we do not break the recursion then it will keep on running infinitely and result in a stack overflow.

# Syntax

```
returntype methodname(){
//code to be executed
methodname();//calling same method
}
```

# Example

```
class Factorial
{ static int factorial( int n ) {
 if (n != 0)
// termination condition
return n * factorial(n-1);
// recursive call
else
return 1; }

public static void
main(String[] args) {
int number = 4, result;
result = factorial(number);
System.out.println(number +
" factorial = " + result); }
}
```

# Constructors in JAVA

- A constructor in Java is similar to a method that is invoked when an object of the class is created.
- Unlike Java methods, a constructor has the same name as that of the class and does not have any return type.

# How Constructors are Different From Methods in Java?

- Constructors must have the same name as the class within which it is defined while it is not necessary for the method in Java.
- Constructors do not return any type while method(s) have the return type or **void** if does not return any value.
- Constructors are called only once at the time of Object creation while method(s) can be called any number of times.

# When is a Constructor called?

- Each time an object is created using a **new()** keyword, at least one constructor (it could be the default constructor) is invoked to assign initial values to the **data members** of the same class.

# The rules for writing constructors

- Constructors are invoked implicitly when you instantiate objects.
- The two rules for creating a constructor are:
  The name of the constructor should be the same as the class.
  A Java constructor must not have a return type.
- If a class doesn't have a constructor, the Java compiler automatically creates a **default constructor** during run-time. The default constructor initializes instance variables with default values. For example, the int variable will be initialized to 0
- A constructor cannot be abstract or static or final.
- A constructor can be overloaded but can not be overridden.

# Types of Constructors in Java

Primarily there are two types of constructors in java:

- No-Argument Constructor
- Parameterized Constructor
- Default Constructor

# No-argument constructor

- Similar to methods, a Java constructor may or may not have any parameters (arguments).
- If a constructor does not accept any parameters, it is known as a no-argument constructor. For example,

```
private Constructor() { // body of the constructor }
```

```
class NamePrint {
    int num;
    String name;
      // this would be invoked while an object
    // of that class is created.

NamePrint()
{ System.out.println("Constructor called");
 }}
```

```
public class MainClass {
    public static void main(String[] args){
       // this would invoke default
 constructor.
       NamePrint np=new NamePrint();
         // Default constructor provides
 the default
       // values to the object like 0, null
       System.out.println(np.name);
       System.out.println(np.num);
}}
```

# Parameterized Constructor

- A constructor that has parameters is known as parameterized constructor.
- If we want to initialize fields of the class with our own values, then use a parameterized constructor.

# Example

```
class Main {
String languages;
// constructor accepting
//single value
Main(String lang) {
languages = lang;
System.out.println(languag
es + " Programming
Language"); }
public static void main(String[]
args) {
// call constructor by //passing
a single value Main obj1 =
new Main("Java");
Main obj2 = new
Main("Python");
Main obj3 = new Main("C");
} }
```

# Java Default Constructor

- If we do not create any constructor, the Java compiler automatically create a no-arg constructor during the execution of the program. This constructor is called default constructor.

# Example

```
class Main {
int a; boolean b;
public static void
main(String[] args) {
// A default
//constructor is called
Main obj = new Main();
System.out.println("Defau
lt Value:");
System.out.println("a =
" + obj.a);
System.out.println("b =
" + obj.b); } }
```

# Constructors Overloading in Java

- The constructor overloading can be defined as the concept of having more than one constructor with different parameters so that every constructor can perform a different task.
- Overloaded constructor is called based upon the parameters specified when new is executed

# Constructor Chaining

- In constructor chain, a constructor is called from another constructor in the same class this process is known as **constructor chaining.**
- It occurs through inheritance. When we create an instance of a derived class, all the constructors of the inherited class (base class) are first invoked, after that the constructor of the calling class (derived class) is invoked.

# Ways to achieve Constructor chianing

- **Within the same class:** If the constructors belong to the same class, we use **this**
- **From the base class:** If the constructor belongs to different classes (parent and child classes), we use the **super** keyword to call the constructor from the base class.

# Rules of Constructor Chaining

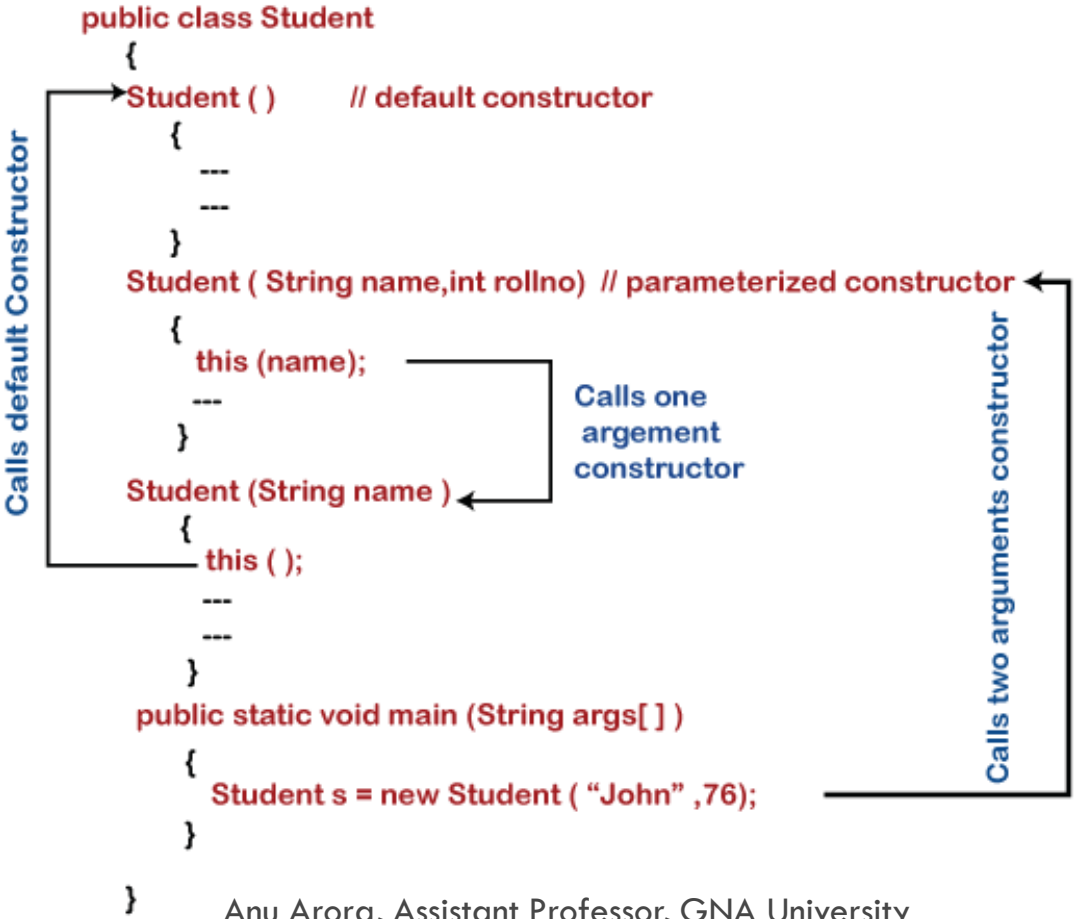
- An expression that uses **this** keyword must be the first line of the constructor.
- **Order** does not matter in constructor chaining.
- There must exist at least one constructor that does not use **this**

# Constructor Calling form another Constructor

The calling of the constructor can be done in two ways:

- **By using this() keyword:** It is used when we want to call the current class constructor within the same class.
- **By using super() keyword:** It is used when we want to call the superclass constructor from the base class.

## Constructor Chaining in Java



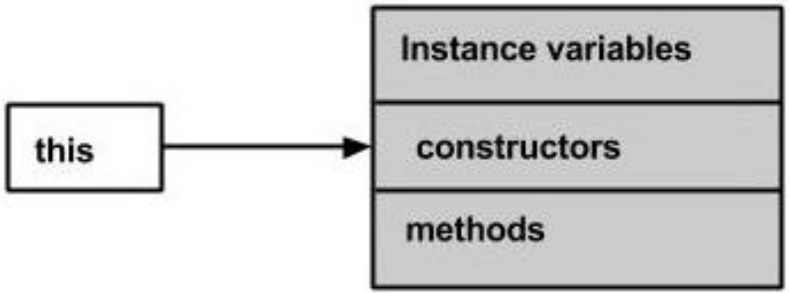
| Constructor Overloading | Constructor Chaining |
| --- | --- |
| 1. Constructor overloading allows a class to have more than one constructor that have the same name as that of the class but differ only in terms of number or type of parameters. | 1. Constructor chaining is a process of calling the one constructor from another constructor with respect to current object. |
| 2. Constructor overloading is done to construct object in different ways. | 2. Constructor chaining is done to call one constructor from another constructor. |
| 3. Constructor overloading is achieved by declaring more than one constructor with same name but different parameters in a same class. | 3. Constructor chaining is achieved by this() method. |
| 4. Constructor overloading is flexible which allows us to create object in different way. | 4. Constructor chaining is useful when we have many constructors in the class and want to use that constructor. |

# The this keyword

- **this** is a keyword in Java which is used as a reference to the object of the current class, with in an instance method or a constructor.
- Using *this* you can refer the members of a class such as constructors, variables and methods.
- **Note** – The keyword *this* is used only within instance methods or constructors

# this keyword

- In general, the keyword *this* is used to –
- Differentiate the instance variables from local variables if they have same names, within a constructor or a method.



# The finalize( ) Method

- It is possible to define a method that will be called just before an object's final destruction by the garbage collector. This method is called **finalize( )**, and it can be used to ensure that an object terminates cleanly.
- For example, you might use finalize( ) to make sure that an open file owned by that object is closed.
- To add a finalizer to a class, you simply define the finalize( ) method. The Java runtime calls that method whenever it is about to recycle an object of that class.
- Inside the finalize( ) method, you will specify those actions that must be performed before an object is destroyed.

# References


- E.Balaguruswamy, Programming with JAVA, A primer, 3e, TATA McGraw-Hill Company.
- https://www.javatpoint.com/operators-in-java
- https://www.geeksforgeeks.org/operators-in-java/
- https://www.tutorialspoint.com/
- https://abhiandroid.com/java/constructor-chaining.html