**JAVA**

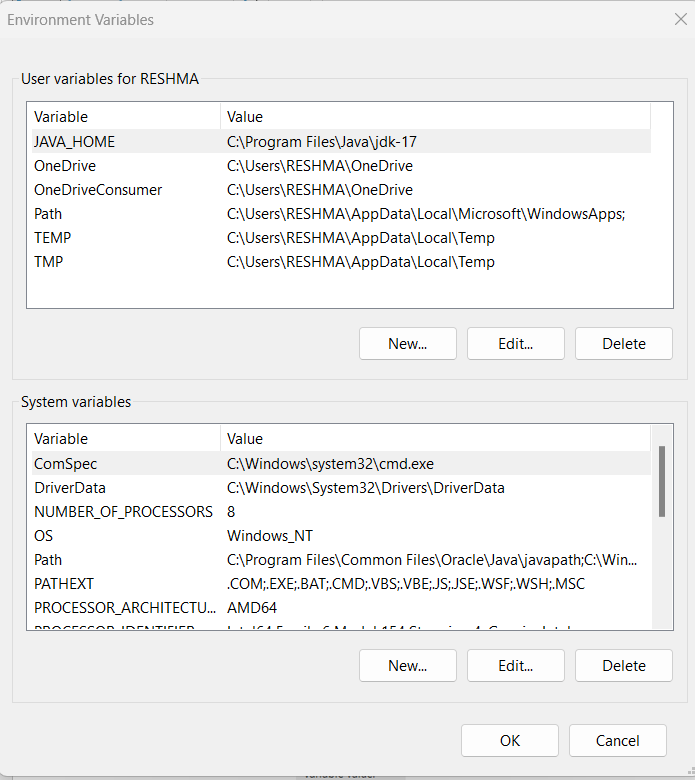
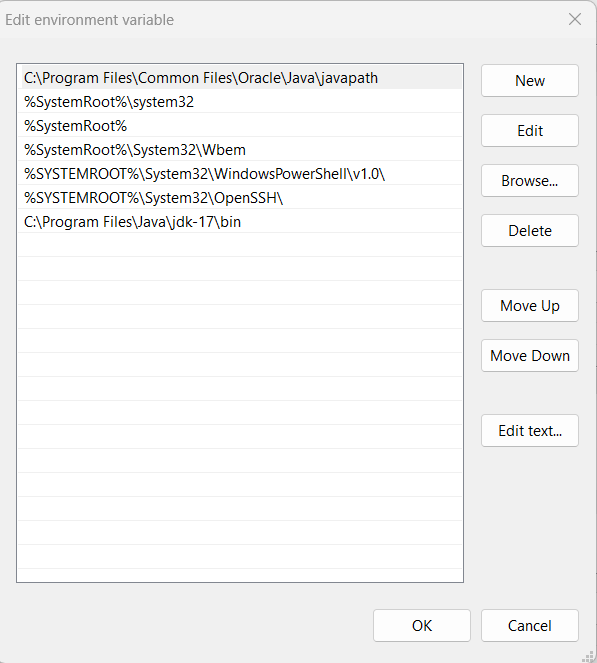
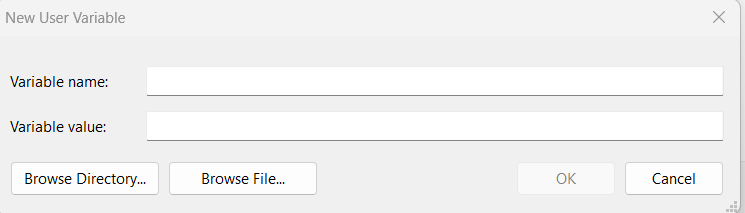
# How to setup JDK

## Jdk 17 installation & Configuration

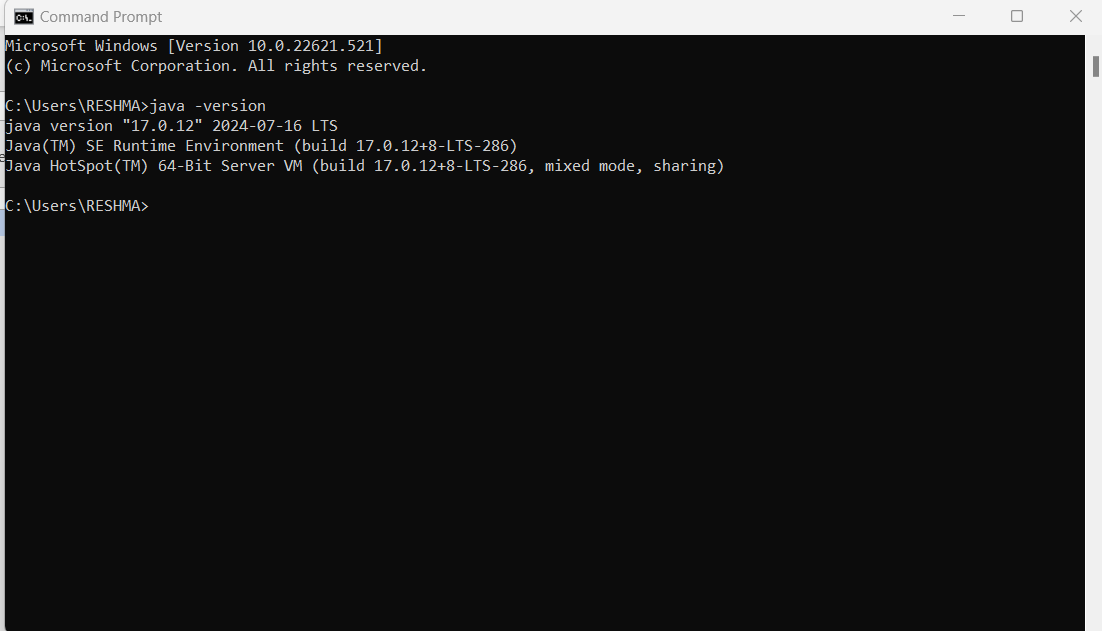
1. Download jdk17

* Windowsx64 is to be used for 64-bit PC. Once installed the kit needs to be configured with the PC using the steps below.

1. Run it as administrator
2. After installation set up path

* C. → Program Files → Java → jdk-17 (copy the path)
* Right click on This PC → Properties →Advance System Settings → Environment Variables
* In User Variables, New → {Variable name: JAVA\_HOME & Variable Value: Paste the path till Jdk-17 file} → Ok
* In System variables, Path → Edit → New →Pate the path till bin

1. To check if it is installed properly
2. Open Command prompt → Type “java -version” →enter

We can see the current version of jdk (if there is any issue check the path)

## Jdk -Java Development Kit

* Development Environment for running java programs.
* Consists of set of libraries

# How to use Eclipse?

1. Create new java project

File → New → java Project

Others →java →java project

1. Create a package inside src folder

Right click on src → new →package

1. Create a class inside the package & write the program

Right click on package → new →class

# What is java?

* Programming language developed by **Sun Microsystems** in **1995**
* Initially it is called **Oak**
* Father → **James Gosling**

# History

* It is developed for **interactive television** but it is too advanced technology for digital cable television industry at that time
* Java team members → **Green Team**
* Used for → **Internet Programming**

**Mobile Devices**

**Games**

**E-business solutions** etc.

# Features

## Simple

* **Simple** Syntax.
* No **Explicit** **pointers** (used for storing Address of variables).
* No **Operator Overloading.**
* No need to remove Unreferenced objects (i.e. no longer in use objects)
* **Automatic Garbage Collection** → process that **automatically** manages the memory by **identifying & removing unused /unreferenced objects** from the heap memory

→ Done by **JVM**

## Object -Oriented

* Object **→** Entity with **State** & **Behaviour**
* **Everything** in java is an object.
* OOPs Concepts

1. **Object**
2. **Class**
3. **Inheritance**
4. **Polymorphism**
5. **Abstraction**
6. **Encapsulation**

* **Object-Oriented Programming** → **methodology or paradigm** to design a program using **classes** and **objects**.

→ It **simplifies** software development and maintenance.

## Platform Independent



* Platform → **S/w** or **H/w** in which program runs
* **Only Requirement** →JDK
* **Compiler** compiles the source code i.e. .java file to find the syntax error and **converts** it into **byte code** i.e. .class file and stored it in the file
* Always run the byte code which is **platform independent code**
* It is **write once run anywhere (WORA)**
* Byte Code → **intermediate code** generated by the compiler after the **compilation** of source code

→This intermediate code **makes** Java a platform-independent language.

## Portable

* Java provides **S/W based** platform
* Because it can be run on any device that has a **Java Virtual Machine (JVM)** without needing to be recompiled.
* This is due to Java's "write once, run anywhere" principle, which allows Java code to be compiled into bytecode that is independent of the platform

## Secured

* **No pointers**.
* Java run **inside a virtual machine** i.e. JVM

## Robust

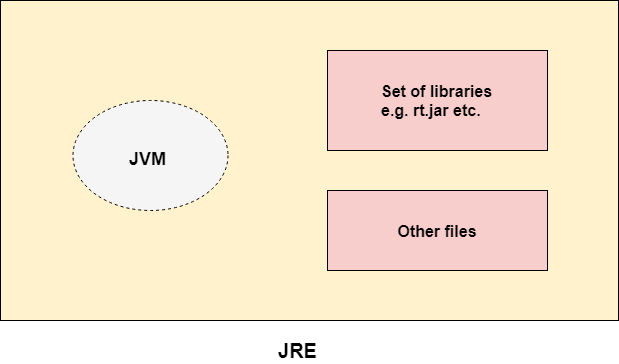
* **Strong memory** management.
* Provides **Automatic Garbage Collection**
* **Lack of pointers** avoid security issues.

## Architectural Neutral

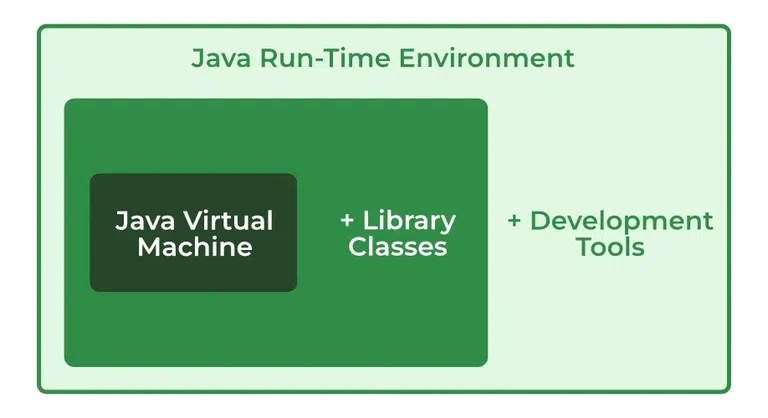
* run on **any platform or operating system** without changing the code
* **no implementation dependent** features

# JVM, JRE & JDK

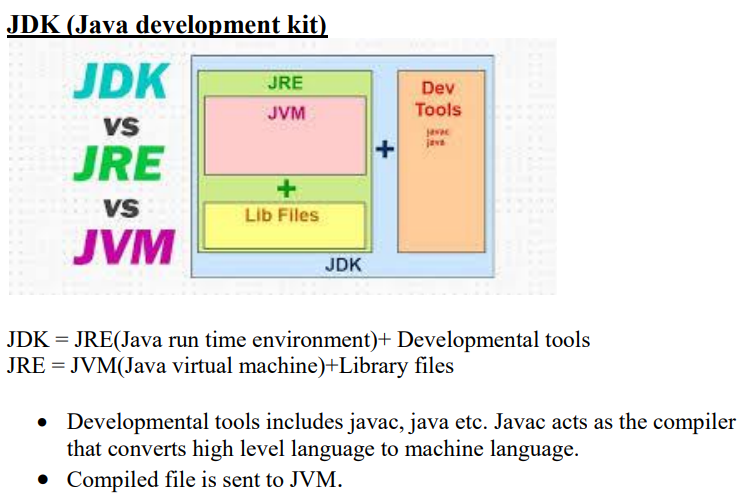
## JVM

* **Java Virtual Machine**
* Doesn’t physically exist
* virtual machine that allows Java programs **to run on different OS and platforms**
* part of the **Java Runtime Environment (JRE)**
* Performs
* **Loads** code
* **Verifies** code
* **Executes** code
* Provides **runtime environment**

## JRE

* Provides **run time environment**
* Part of **JDK**
* It is the **implementation of JVM**, physically exists.
* **software layer** that allows Java programs to **run** on a computer.
* Contains set of **libraries & other files** that needed during the run time
* JVM + library Files

## JDK

* **Java Development kit**
* **JRE + Development** tools
* **software development environment** for building Java applications, applets, and components.
* It includes **tools and utilities** for developing and testing Java programs
* **Javac** acts as the **compiler that converts** high level language to machine language.

# Packages

* A java package is a group of **similar types** of classes, interfaces and sub packages.
* 2 types:

1. **Built in** package
2. **User defined** package.

* **Advantage** of package:

1**. Categorise** the classes so that they are easily maintained.

2. Provides **access protection**

3. **Removes name collision**

* ‘**Package**’ keyword is used to **create a package**

E.g.: - package student;

* To **access package** from another package we have the following methods

1. import package name. \*;

2. import package name .classname;

* Example of built-in packages are **java, lang, awt, javax, swing, net, io, util, sql** etc.

# Class

* **Template/Blueprint** of objects
* **Group of objects** which have **common properties**.
* Syntax – **Set of rules**
* Keyword – ‘**class’**
* Syntax: -

class ClassName

{

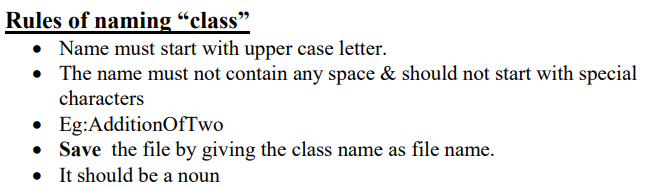
// set of codes

}

E.g.: class Addition

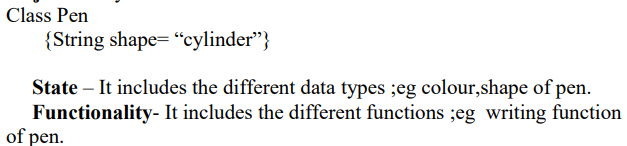
{

//Addition Operation

 }

# Object

* **Instance of a class**.
* An entity that has **state & behaviour**.
* Can be **physical or logical** entity i.e. real-world entity.
* State **→** Represents the **data** of an object i.e. variables
* Behaviour **→** Represents the **functionality** of an object i.e. methods.



# Java Program: to Print “Hello World”

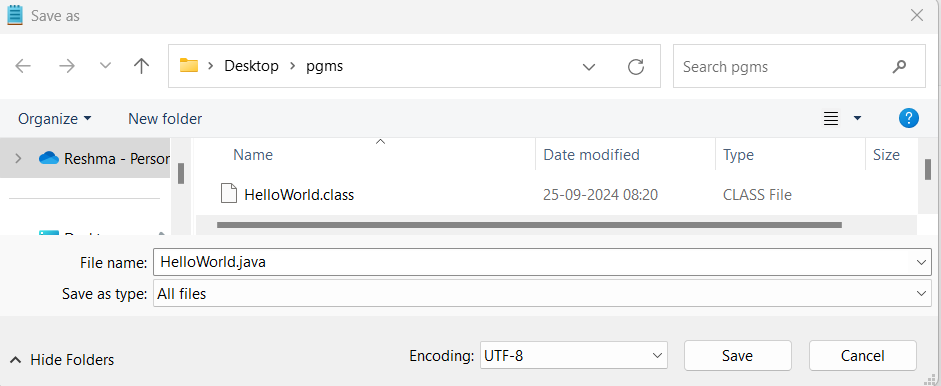
* **Execution starts** at main ()

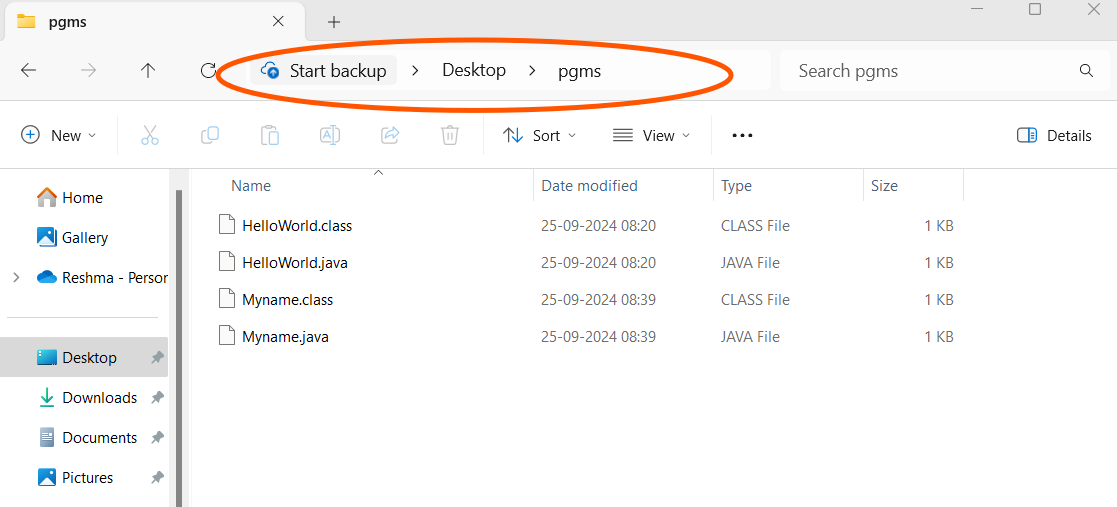
public static void main (String args []) →Method Signature

{

}

* Printing statement: - **System.out.println();**
* Statements should **end** with ‘**;**’ (semicolon).
* To Save a file: File → Save as

 File name → class name.java type → All files or java

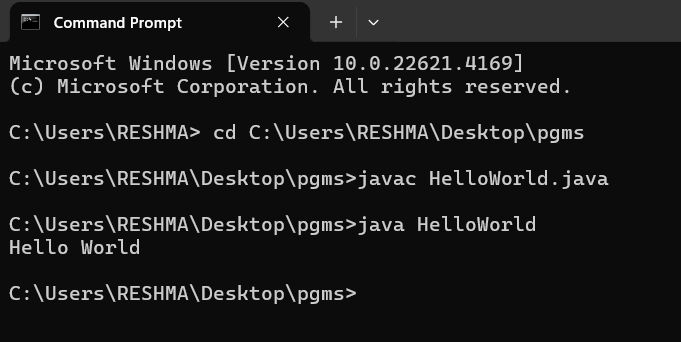
* In Command Prompt

To go to that file location:

1. open that file then type at address path as ‘cmd’

2. copy that location and paste it at cmd

i.e. cd \_ address location



* To compile → javac classname.java
* To run → java classname
* Compile time Error → Errors occurred @compile time mainly syntax errors
* Run time Errors → Errors occurred @run time mainly logical errors

# Data Types

* Specifies the **different sizes & values** that can be stored in a variable.
* Variable → to **store** any values
* 2 Types

1. **Primitive**
2. **Non-Primitive/Wrapper** **Class**

## Non-Primitive Data Types

* Also called **Wrapper class** or **Predefined Classes**
* Class that **encapsulates data types**, allowing them to be used **to create object instances and methods** in other classes
* Byte Short Integer Float

Character Boolean String Array

Interface Object

## Primitive Data Type

* **8** types
* byte - 1 byte short - 2-byte int - 4 byte

long - 8-byte char - 2-byte float - 4 byte

Boolean – true/false double - 8 byte

* int a = 10;

int a →**Declaration** part

10; → **Initialization** part

# Arithmetic Operations

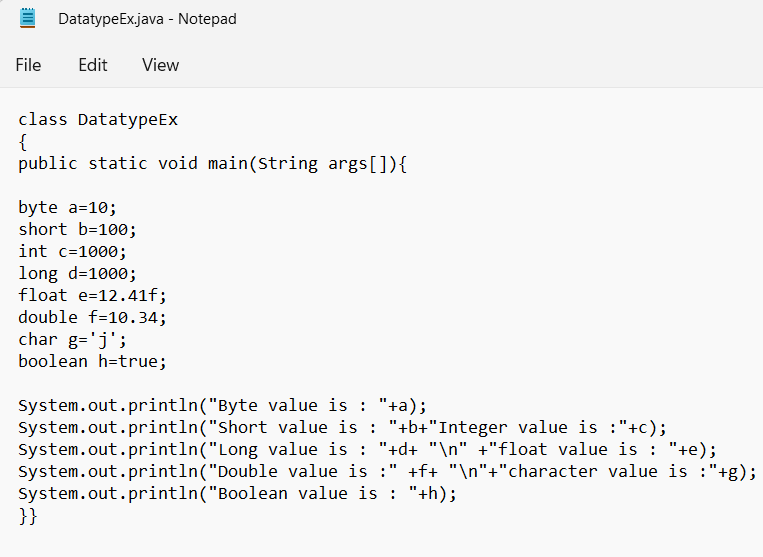
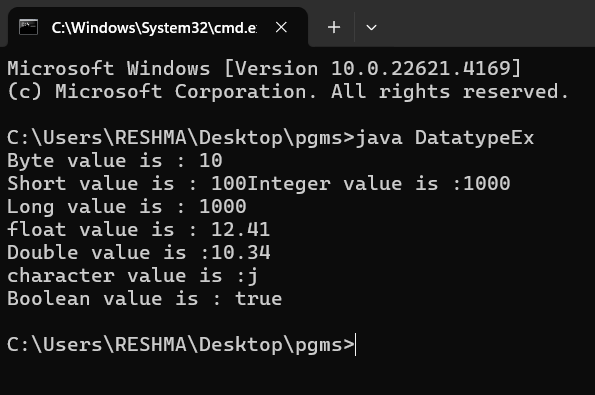
* + → Addition

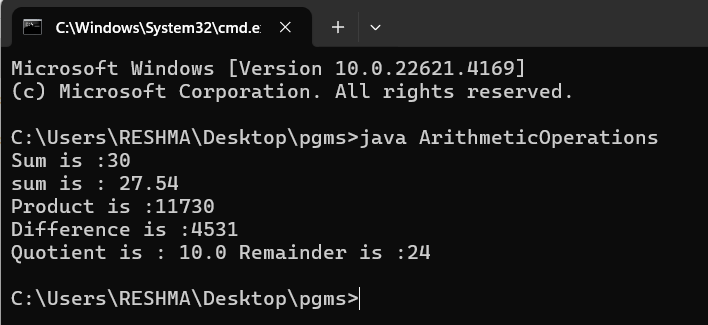
- → Subtraction

\* → Multiplication

/ → Quotient

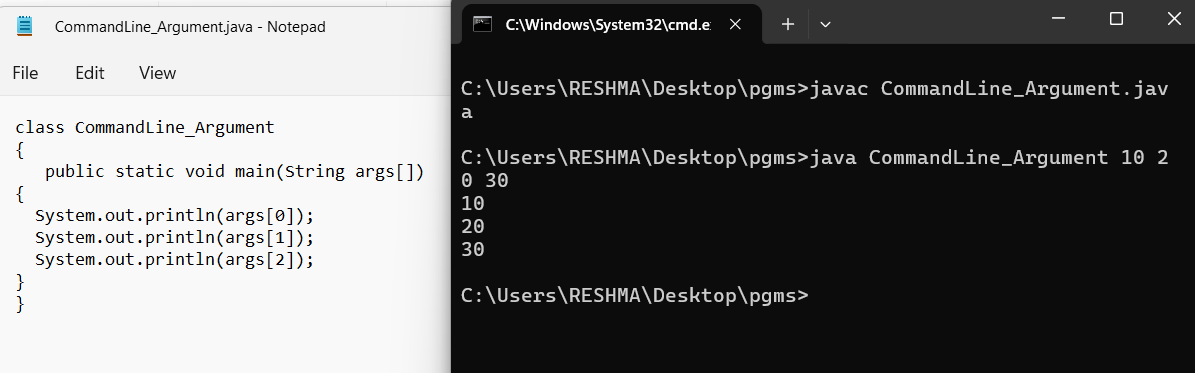
% → Remainder



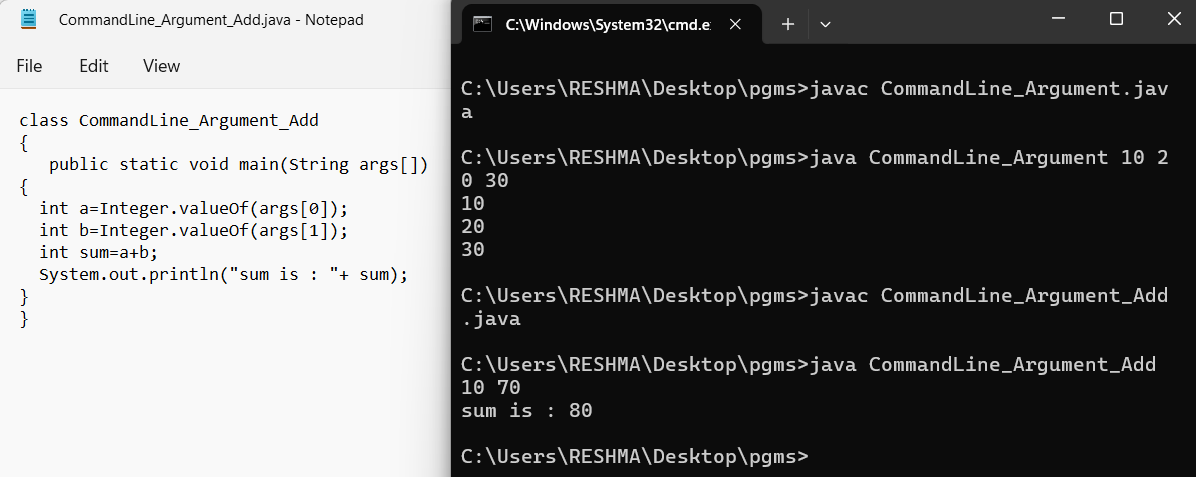


# Command Line Argument

* An argument is a value **passed** to a function when the **function is called**
* Arguments that are **passed** @ the **time of running** of java program.
* Arguments that are giving **during execution**
* The arguments **passed from the console** can be received in the java program and it can be used as an **input**. So, it provides a convenient way to **check the behaviour** of the program for the different values.
* In eclipse 🡪 run as 🡪 run configuration 🡪 in arguments option type the values



## String To Integer

* **Integer.valueOf(val);**
* **Integer.parseInt(val);**
* Both produces same output the **only difference** is that **valueOf ()** returns an **Integer object** while **Integer. parseInt ()** returns a **primitive int**.

# Variables

* Used to **store the data values** during java program execution.

E.g.: int a=10;

* **Naming Convention**:

1. Should **start** with **lowercase**

E.g.: id, name. Roll etc.

1. Should **not start** with **special characters**

E.g.: $ # % @ etc.

1. **1st letter** of 1st word should be **lowercase** & 1st letter of **2nd words** should be **uppercase**

E.g.: firstName, rollNo etc.

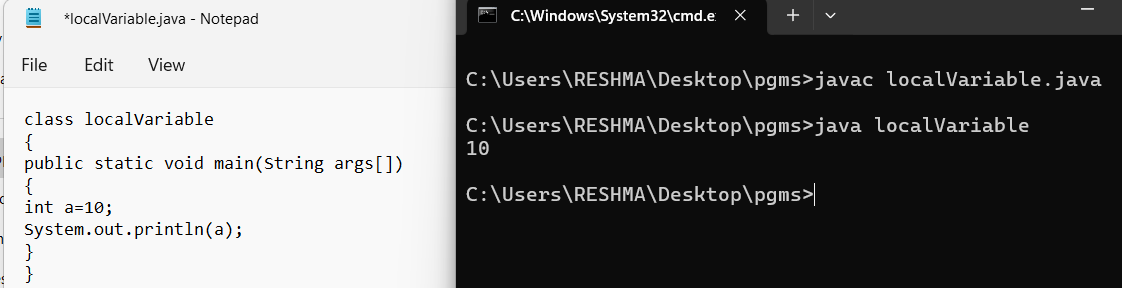
1. Avoid using 1-character variables

E.g.: a, v, e etc.

## Types of Variables

1. Local Variable
2. Instance Variable
3. Static Variable

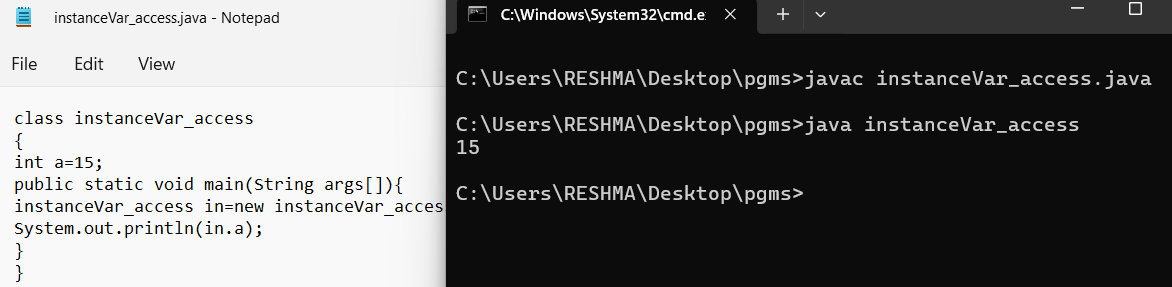
### Local Variable

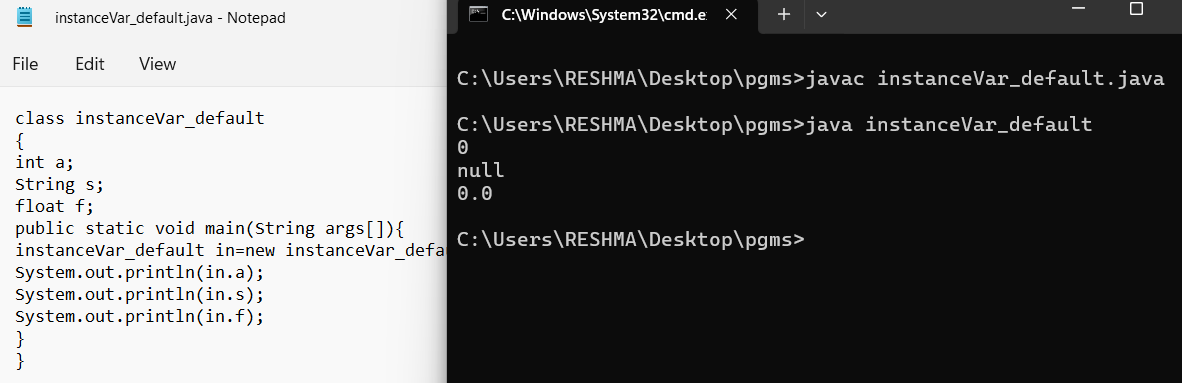
* Variables that are defined **within a block/method/constructor**
* **temporary** variable that is **only available within** a specific function or block of code
* **Scope** → exists **only** **within** the block/method/constructor in which it is defined.
* **Declaration** → Local variables are declared **in a block, loop statement, or function body**. The declaration includes the **variable type, name, and optional initialization**.
* **Initialization** → Local variables **do not have a default value**, so they **must be declared and initialized** before they are first used.
* **Creation and destruction** → Local variables are created when the code section is **executed** and destroyed when it **finishes**.
* **Visibility** → Local variables are **only visible within** the method, constructor, or block that they are declared in
* **Use** → Local variables are **useful** when data is only needed within a specific expression

### Instance/Non-static Variable

* Variables that are defined **within a class but outside** the block/method /constructor.
* **Scope** → **within** the class
* **Initialization** → is **not mandatory**
* **Modification** → Instance variables can be **changed directly**, provided access restrictions are set.
* **Value** → **Each instanc**e of a class has **its own copy** of the instance variables.
* **Lifetime** → Instance variables live in **memory for the lifetime of the object** they are owned by.
* **Destruction** → Instance variables are destroyed when the **object is destroyed**.
* It can be **accessed only by creating objects**
* Access → Instance variables can be accessed and modified by any method or constructor in the class.
* Object creation →

Class\_Name object\_name = new Class\_Name ();

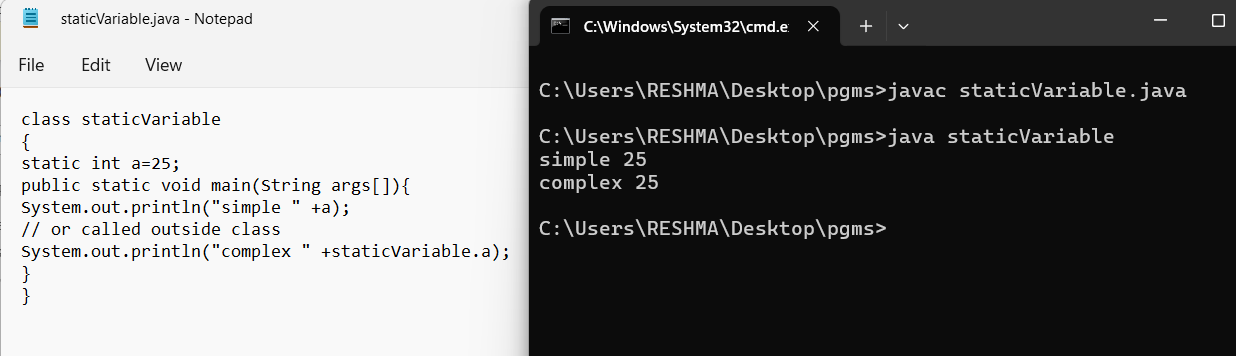


* If we don’t give values, it will take default values
* **Default value** → Instance variables always **get a default value**.
* **Re-initialization** → Instance variables **cannot be re-initialized** directly within a class, but they can be re-initialized inside methods or constructors.
* Instance variables are different from class variables, which can only have **one value at any one time**, **shared between all instances**

### Static Variable

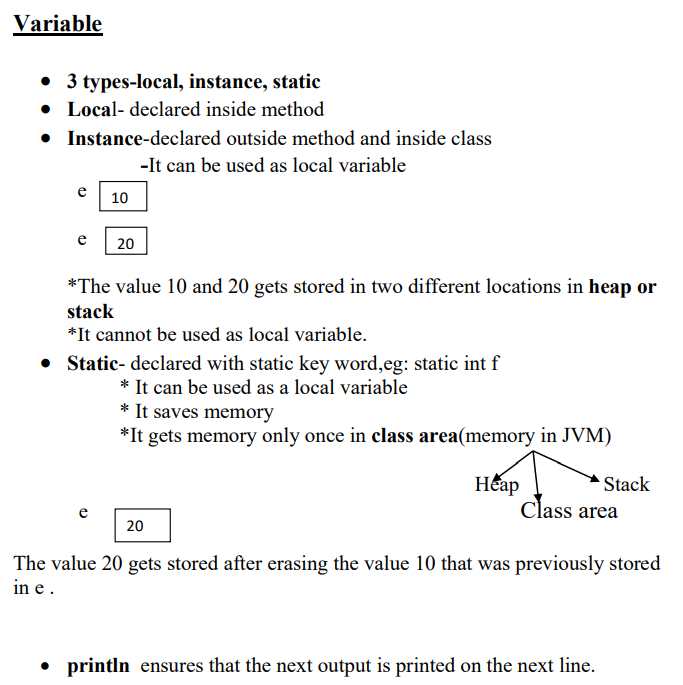
* Variables that are defined **within a class** with a ‘**static**’ keyword **but outside** the block/method/constructor.

**static variable\_name**;

* declared using the **static keyword** and are **shared among all instances/objects** of the class.
* **Scope** → **within** the class
* **Initialization** → is **not** mandatory
* **Memory efficiency** → Static variables are **allocated memory only once** when the **class is loaded**.
* **Global access** → Static variables are like global variables and are **available to all** methods.
* **Object independence** → Static variables are produced at the **program's start** and destroyed at its **conclusion**.
* **How to access** → Static variables can only be accessed directly with the class name.
* **Can be accessed** by → variable\_name OR

class\_name. variable\_name [← in case of outside class]

* **Common properties** → Static variables are often used **to store common properties** **shared by the class objects**.

****For example, the name of a department for a college class.

# Reading input from Keyboard

* Using **Scanner** class
* **Creating object** of Scanner class
* **Scanner sc=new Scanner (System.in)** ← package: **Java.util. scanner**
* Integer → nextInt ()

Char → next (). charAt(0) -- Changed to charAt(0) to get the first character

Float → nextFloat()

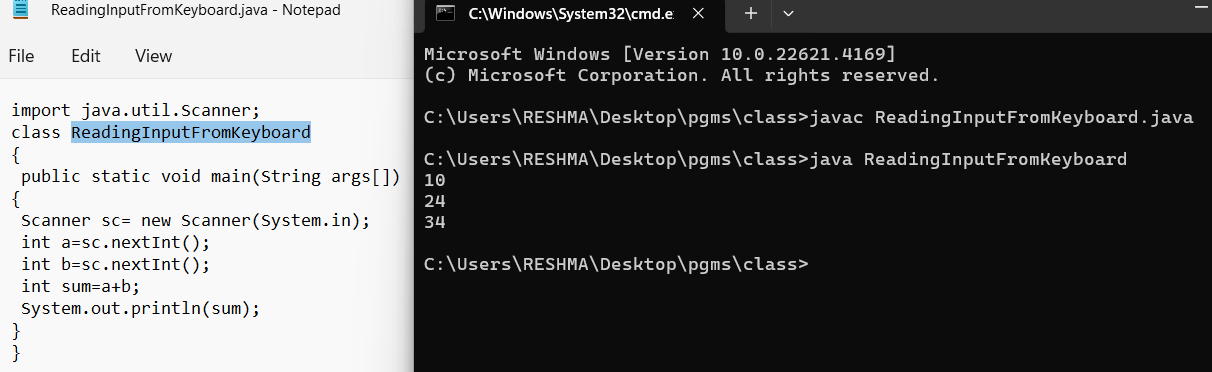
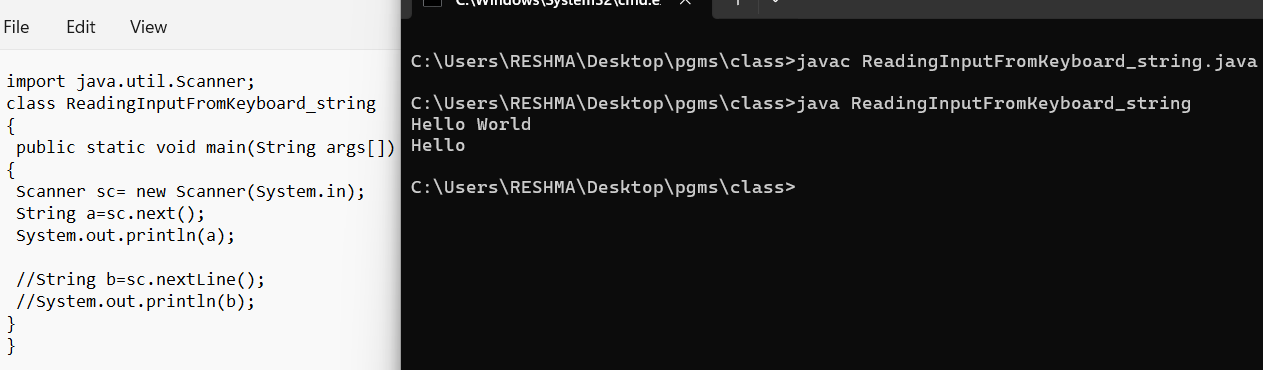
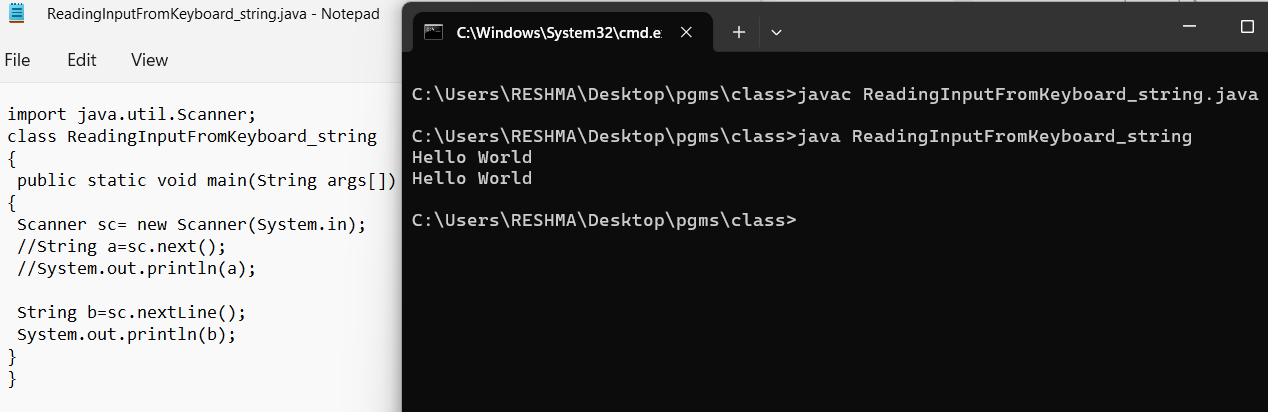
Double → nextDouble()

Long → nextLong()

Boolean → nextBoolean()

String → next() ---only reads till space i.e. it stops at space

nextLine () ---reads the entire line including space

* Character.toLowerCase(s)) -- Convert ‘s’ to lowercase character
* For integers:
* For string:

# Decision Making in Java

## Simple if statement

Syntax: if(condition)

{

True //Executes if condition is true

}

## If…...else statement

Syntax: if(condition)

{

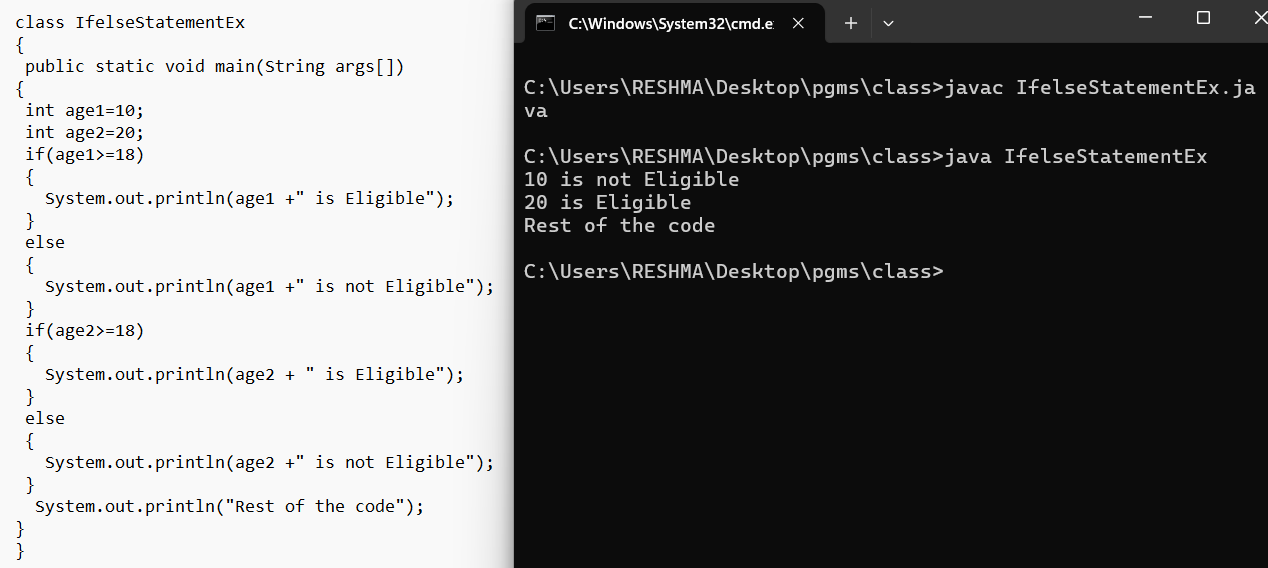
True //Executes if condition is true

}

else

{

False //Executes when condition is false

}

## If…. else if…...else statement

Syntax: if(condition1)

{

//Executes if condition is true

}

else if(condition2)

{

//Executes when condition1 is false & condition2 is true

}

Else

{

//Executes when both condition is false

}



## switch statement

Syntax: switch (Expression)

{

case value1:

Statement1;

break;

case value2:

Statement1;

break;

.

.

.

case valueN:

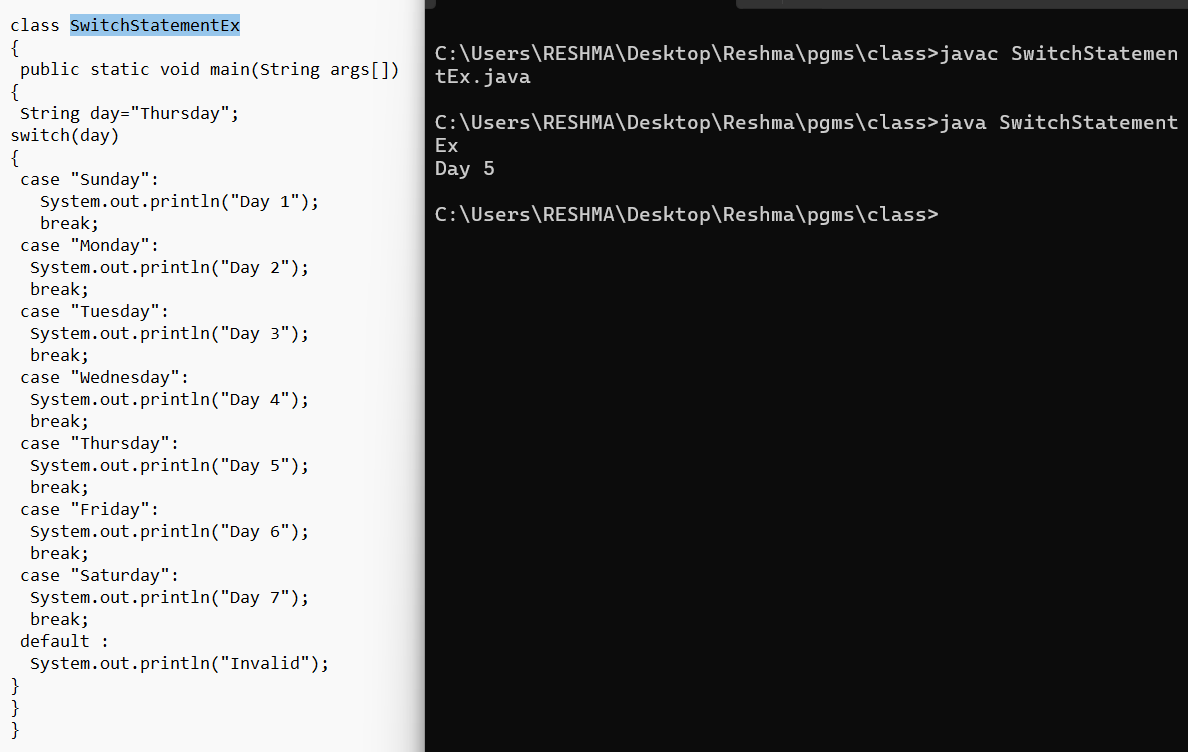
StatementN;

break;

default:

Statement;

}



# Looping Statement

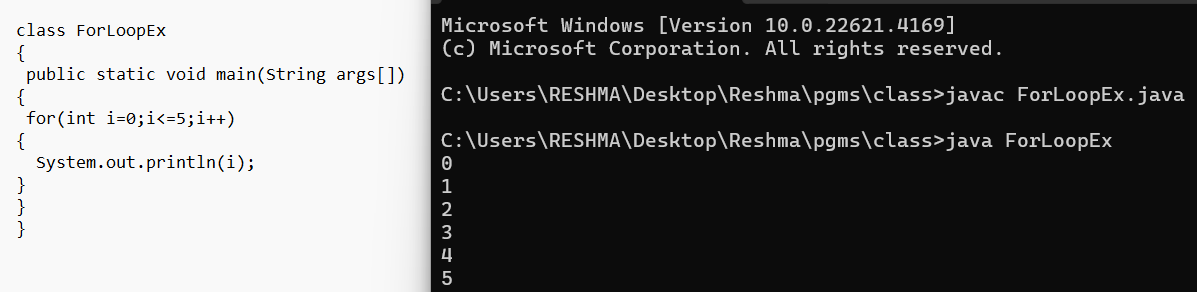
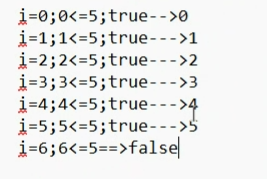
1. for loop
2. while loop
3. do while loop
4. for each

## For loop

* **Entry** controlled loop
* Syntax: for (initialization; condition; increment/decrement)

{

// Code to be executed

}

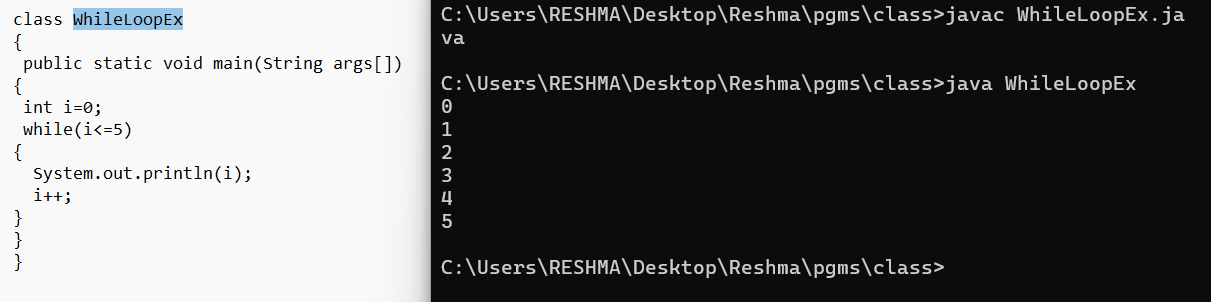
## While loop

* **Entry** controlled loop
* Syntax: while(condition)

{

//Code to be executed

//increment/decrement

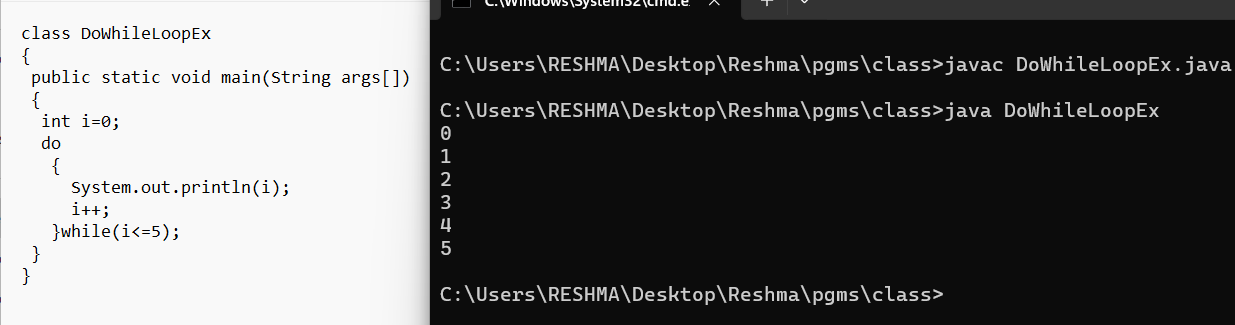
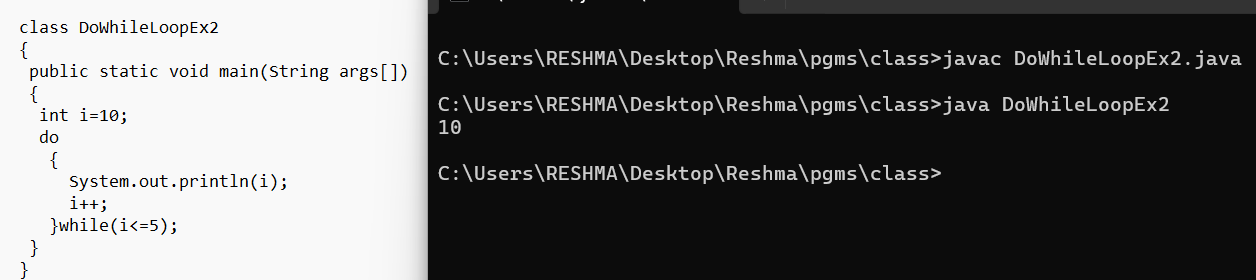
 }

## do…. while Loop

Syntax: do

{

//code to be executed

} while(condition);

* **Executes once** even if the **condition is false**
* It runs the loop once **without checking the condition** due to the **usage of “do**” initially
* A **For loop** is used when the **number of iterations** is known. A **While loop** runs as long as a **condition is true**. A **Do-While loop** runs **at least once** and then **continues** if a condition is **true**
* **For** & **while** Loop → **entry-**controlled loop

**do while** loop → **exit** controlled loop

* In an **entry-controlled** loop, the **test condition is checked before the loop body** is executed. If the condition is **true**, the loop body is **executed**, and if the condition is **false**, the loop body is **not executed**
* In an **exit-controlled** loop, the **test condition is evaluated after the loop body**, so the loop body is **always executed at least once**

## For each loop

* Syntax : for (data\_type variable : array\_name)   
   {   
   statements using var;  
   }
* Eg:int arr[]={3,45,67,89};

For(int i:arr)

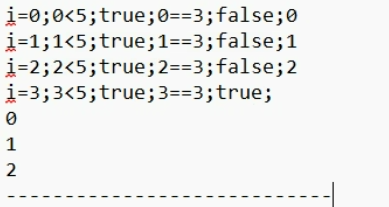
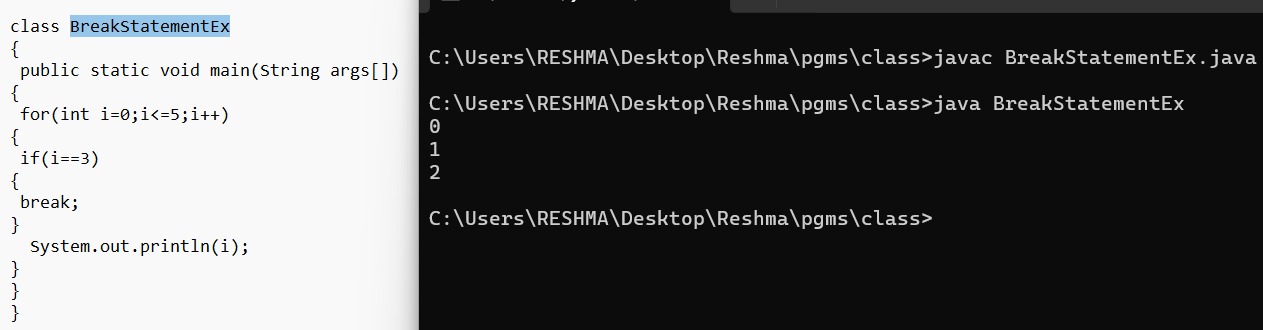
{

System.out.println(i);

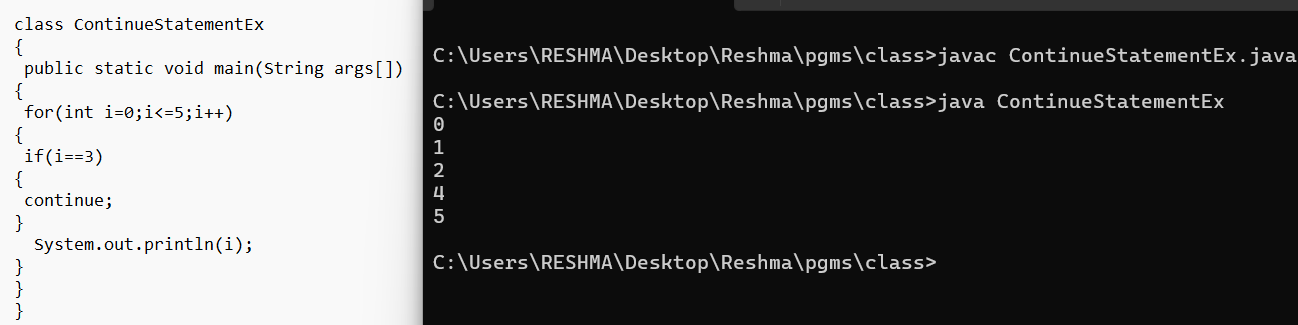
}

# Branching Statements in Java

## 1.Break

* Used to **terminate from a loop** immediately
* Or break statement **stops the iteration once** **and for all** and hence **no more execution** occurs.

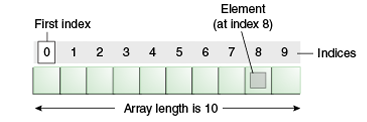
## 2. Continue

* Used to **skip the current iteration** of a loop.
* continue is used to **skip the remaining statements** in the loop and to **continue with the next iteration.**

# Java Comments

* **Comments** can be used to **explain Java** code, and to make it **more readable**. It can also be used to **prevent execution when testing alternative code**
* **//** → **Single** Line Comment
* **/\* …………………………\*/** → **Multi line** Comment

# Array

* **Collection of Similar** data types of elements.
* ****used to **store multiple values in a single variable**, instead of declaring separate variables for each value
* Elements of an array are stored in the **continuous memory location.**
* Can store **only a fixed set** of elements i.e. **not dynamic**
* **Index based** – 1st element @0Th position & 2nd element @1st position and so on.
* **Adv** → **Code Optimization --**We can retrieve or sort data efficiently.

**Random Access --** We can get any data located at an index position.

* **Disadv → Size limit**
* **ArrayIndexOutOfBoundsException**: The ArrayIndexOutOfBoundsException occurs whenever we are **trying to access** any item of an array at an **index which is not present** in the array. In other words, the index may be **negative or exceed the size** of an array.

## Types of an array

1. **Single** Dimensional
2. **Multi**-Dimensional

### Single Dimensional Array

* **Declaration**:

Syntax: **datatype arrayname[];**  int a[];

**datatype[] arrayname;** int[] a;

**datatype []arrayname;** int []a;

* **Instantiation**:

Syntax: **datatype arrayname[]=new datatype[size];**

E.g.: - int arr[]=new int[5];

* **Initialization:**

Syntax: **arrayname[index]=value**;

E.g.: - Int arr[]=new int[6];

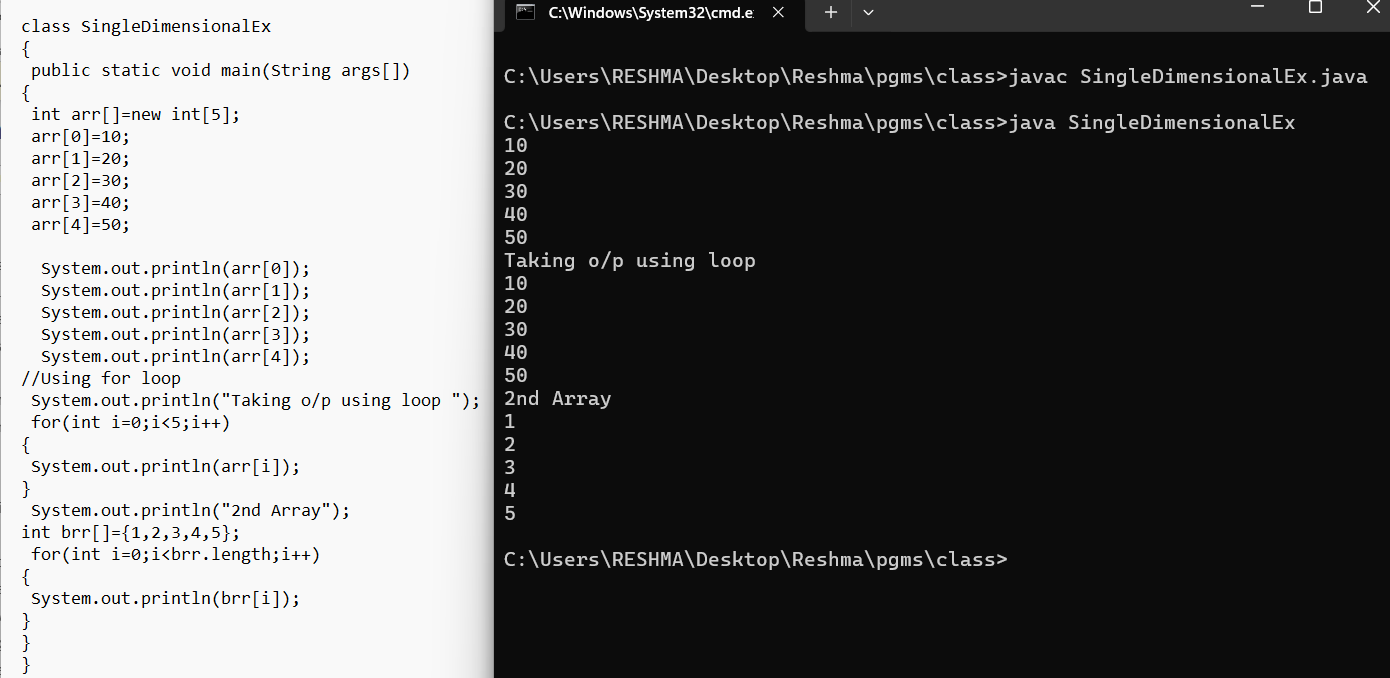
arr[0]=10;

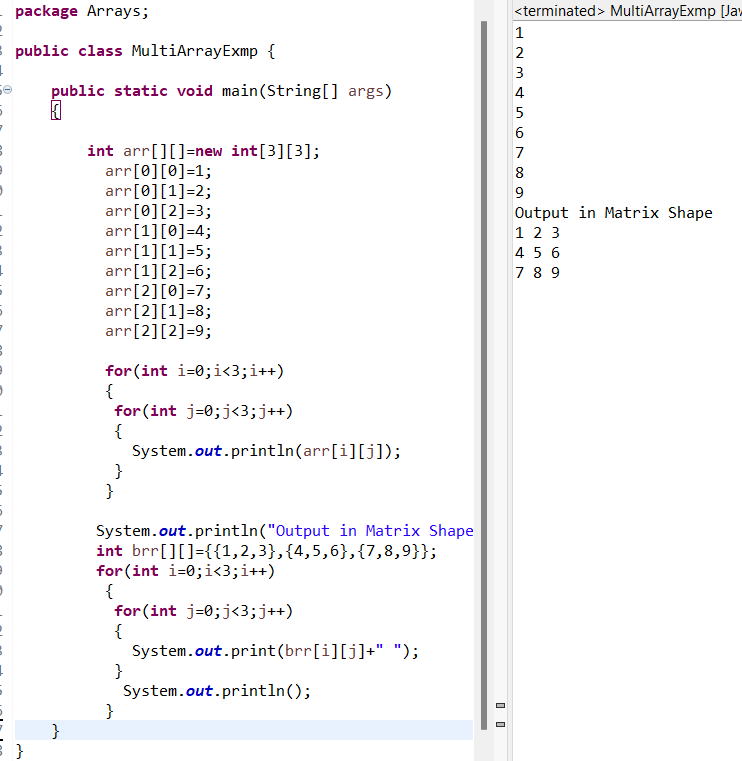
* Or we can do these in single line

Syntax: datatype arrayname[]={value1,value2,……..}

* **Declaration, instantiation &initialisation together**

int a[]={2,4,10}

* **arrayname. length** → gives the **size** of an array



## Multi-Dimensional Array

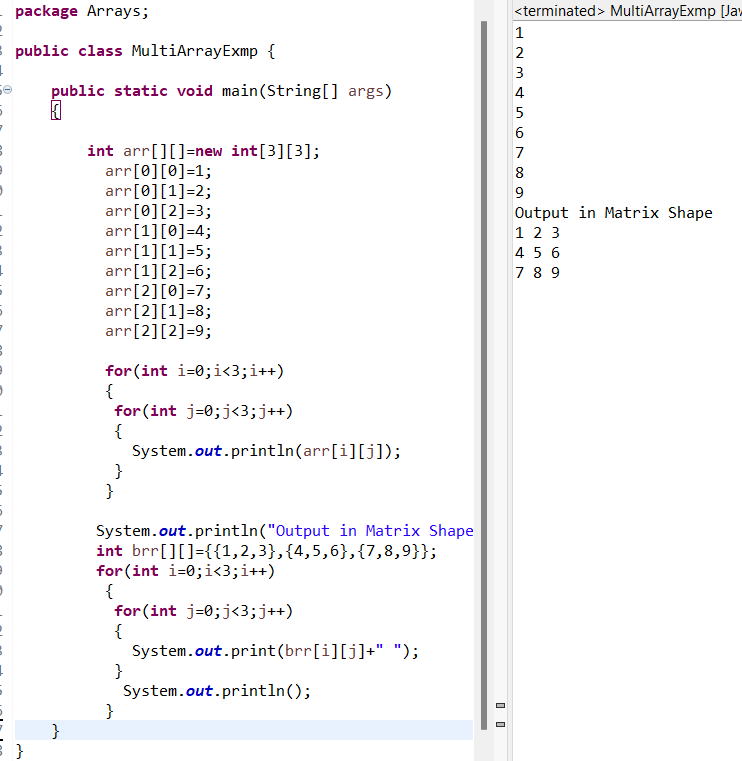
* Syntax: **datatype arrayname[][]=**

**new datatype[row][col];**

E.g.: int arr[][]=new int [3][3];

* **datatype arrayname[][] ={{val1,val2,..}**

**,{val1,val2,..},{.},.};**

 E.g.: - int arr[][]={{1,2,3},{4,5,6},{7,8,9}};

* Here we use 2 loops,

**1st -- row** & **2nd – column**

# Methods

* Set of **codes grouped together** to perform **particular task/operation**
* Executed **only when** we **call/invoke** it
* Syntax: Without Parameter :

**access\_specifier return\_type methodName()**

{

//statements;

}

With Parameter :

**access\_specifier return\_type methodName(datatype param1,……….)**

{

//Statements;

}

* **access\_specifier** → Describe the **accessibility or visibility** of a method.
* **return\_type** → Define the **type of returning value**.

void: - no return

E.g.: - public void run()

{

//statements

}

* Naming Convention:

1. Single word – starts with **lowercase**

Multiple words – **1st word** starts with **lowercase**

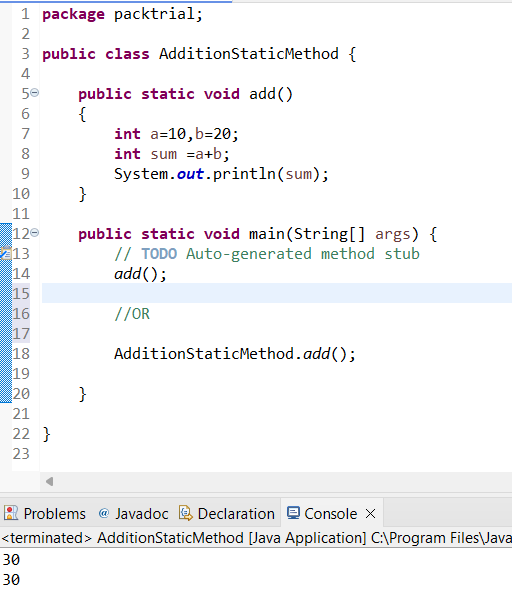
**2nd word** starts with **Uppercase**

E.g.: - run(), runOver().run\_Above()

2. The name **must not contain** any **space** & should not start with **special characters**

3. It should be a **verb** eg:main(),calculate()

## Types of Method

1. **Static** Method
2. **Instance** Method

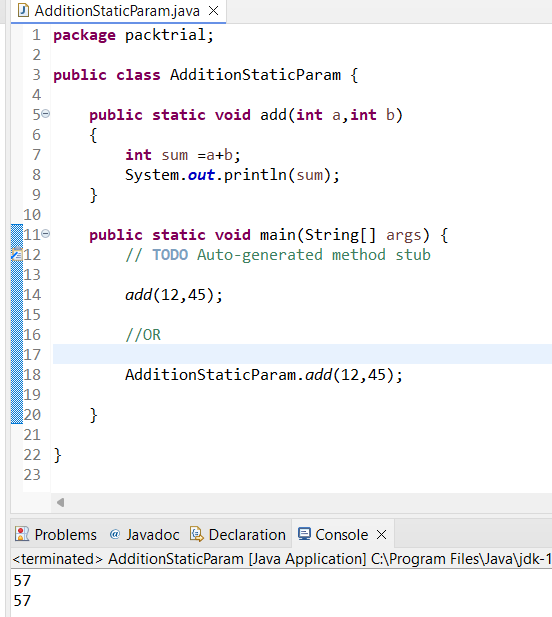
### Static Method

* Use ‘static keyword’
* Syntax: - Without Parameter :-

access\_specifier static return\_type methodName()

{

//Statements

}

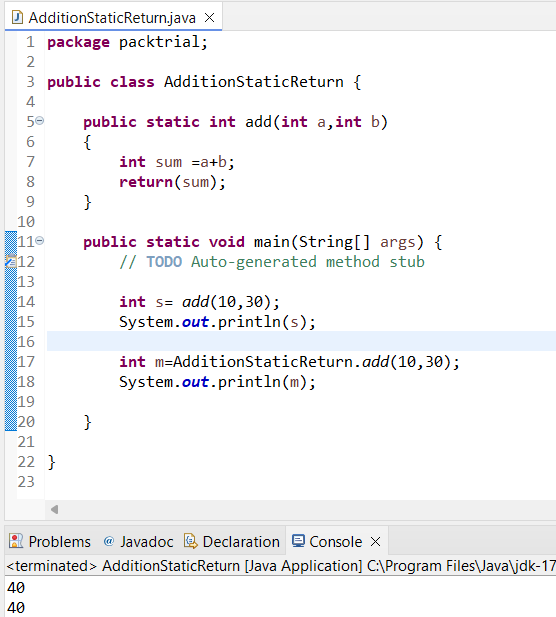
With Parameters: -

access\_specifier static return\_type methodName(datatype param1,….)

{

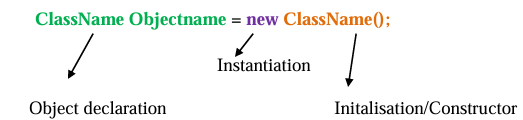
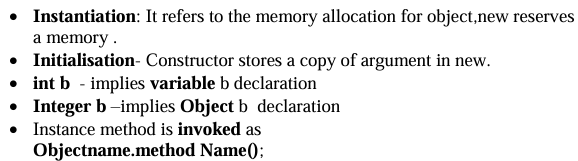
//statements;

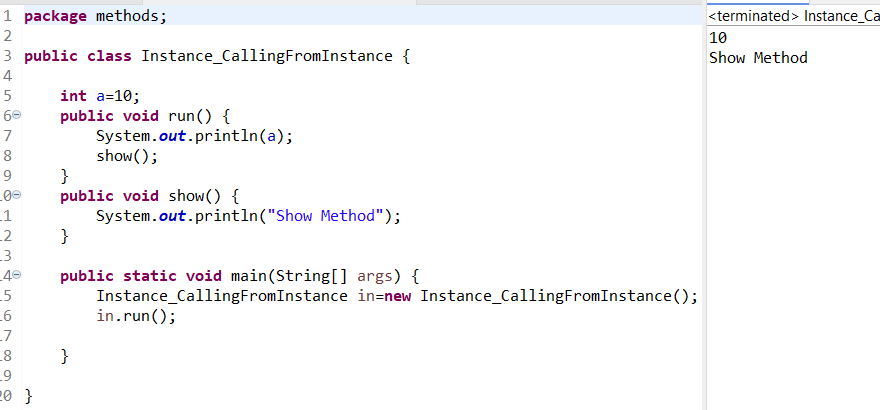
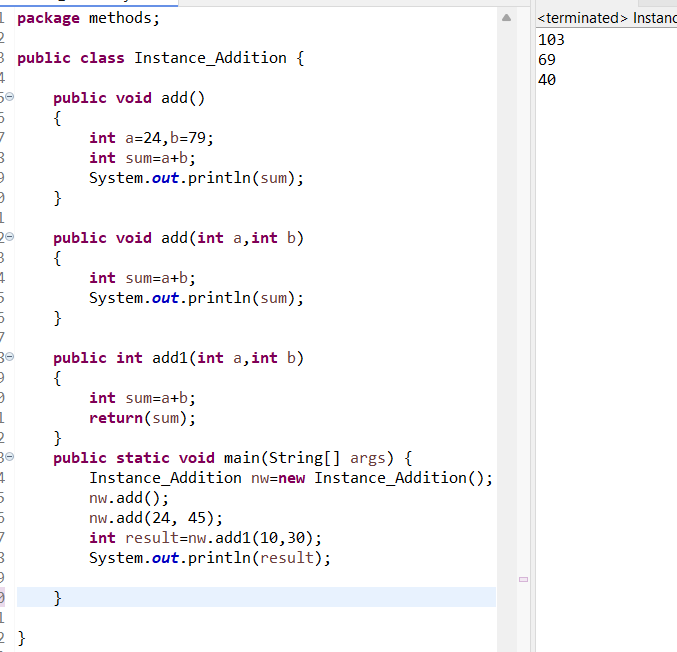
}

 With Return type :-

return\_type as int,float,String,Double …

## Instance Method

* methods that **require an object** of its class to be created before it can be called.
* To invoke an instance method, we have to **create an Object** of the class in which the method is defined.
* **Creation of object** needs to follow the syntax given below:



* **Calling a Static variable & Static method**

ClassName.VariableName / VariableName

ClassName.MethodName() / MethodName()

* **Calling an instance variable & Instance method from Static Context**

Create object, then object.VariableName

Create object, then object.MethodName()

* **Instance variable & instance method from Instance Context**

VariableName

MethodName()

# Method Overloading with Static Methods

* A class with **multiple methods** having **same name** but **different parameters**.
* 3 ways to achieve

1. By changing the **no of arguments**
2. By changing the **datatypes**
3. By changing the **order of parameters**

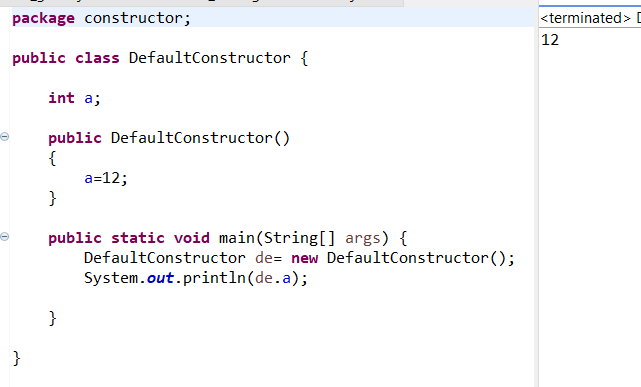
## By changing the no. of arguments

## By Changing the datatype

## By Changing the order of Parameters

# Constructor

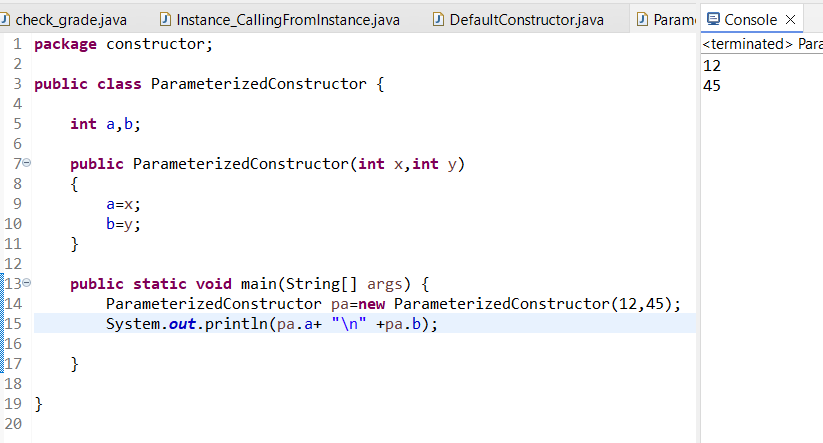
* used to **set initial values** for object attributes and **initialize instance variables**
* Must have **same name as class**.
* **cannot** have a **return type**
* Invoked at the time of **object creation** or **instance creation**.
* Constructors are **automatically called** when an object is created **using the new keyword**
* **cannot** be **static, volatile, or final** i.e. non-access-modifiers.
* **2 types**: 1. **Default** constructor

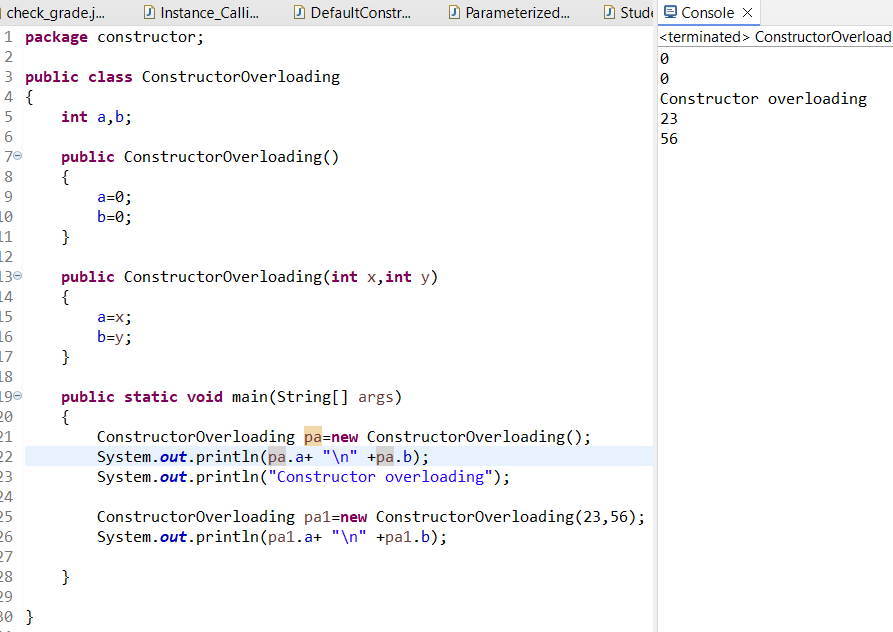
 2.**Parameterized** constructor

## Default Constructor

* Syntax: **Access\_modifier Class\_name()**
* a constructor that is **automatically generated by the compiler** when no constructors are defined for a class
* initializes an object with **default values**, such as 0 for integers and null for objects
* also known as a **no-argument constructor**
* if we write **any constructor** (i.e. default or parameterized), then the complier **will not create** a default constructor.

## Parameterized constructor

* Syntax : **Access\_modifier Class\_name(parameters)**
* a constructor that **accepts one or more parameters** to initialize an object's instance variables



# Constructor Overloading

* Constructor overloading occurs when a **class has multiple constructors** with the **same name** but **different parameters**
* A class can have multiple constructors that **differ in the number and/or type** of their **parameters**.

# Difference between Constructor & Method

|  |  |
| --- | --- |
| **Method** | **Constructor** |
| A **set of statements** that **performs specific task** with and without returning value to the caller is known as method. | A block of code that **initialize at the time of creating a new object** of the class is called constructor. |
| used to **reuse the code** | used for **initializing the object**. |
| A method is **called by** the **programmer**. | It is **implicitly invoked** by the system. |
| **Method calls** are responsible for invoking methods. | The **new keyword** plays an important role in invoking the constructor. |
| it has a **return type**. | It has **no return type**. |
| We can use **any name** for the method name | The constructor name will always be the **same as the class name.** |
| A class can also have **more than one method with the same name** but **different in arguments and datatypes**. | A class can have **more than one parameterized constructor**. But constructors should have **different parameters**. |
| Sub-class can **inherit the method** of the parent class. | Sub-class **cannot inherit** parent class constructor. |

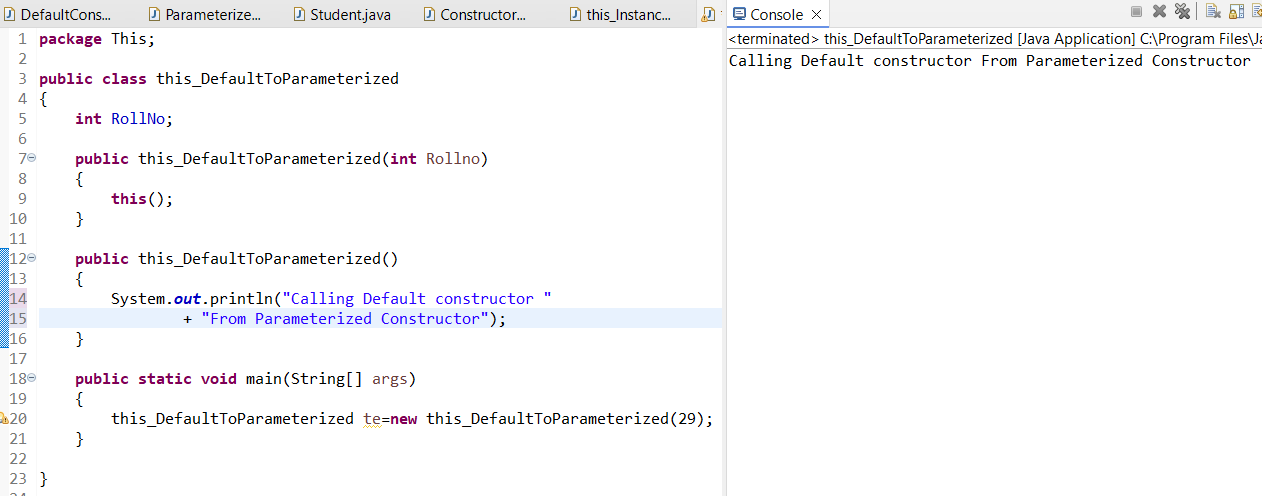
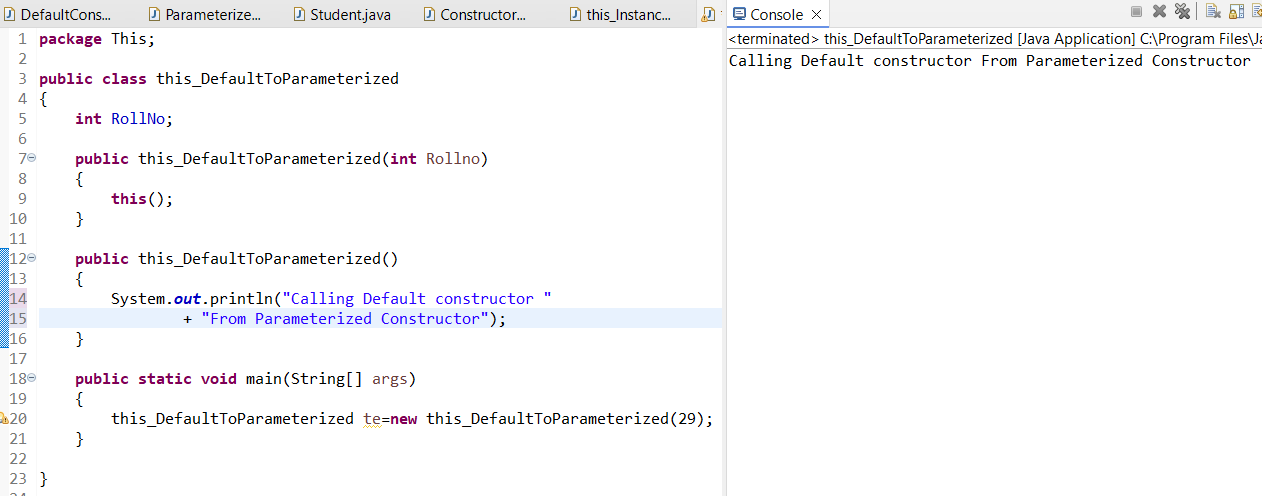
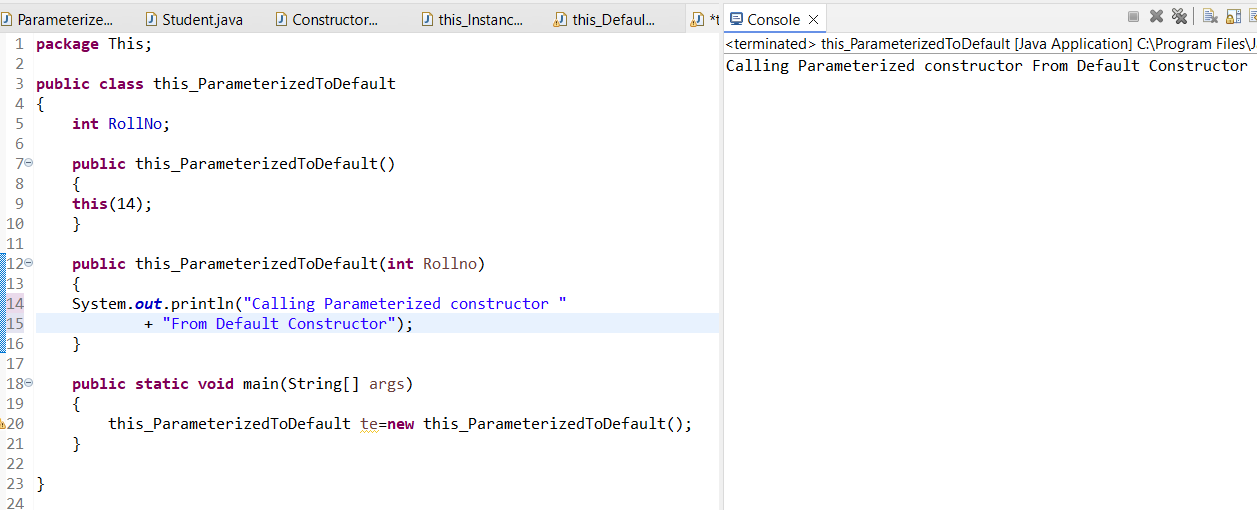
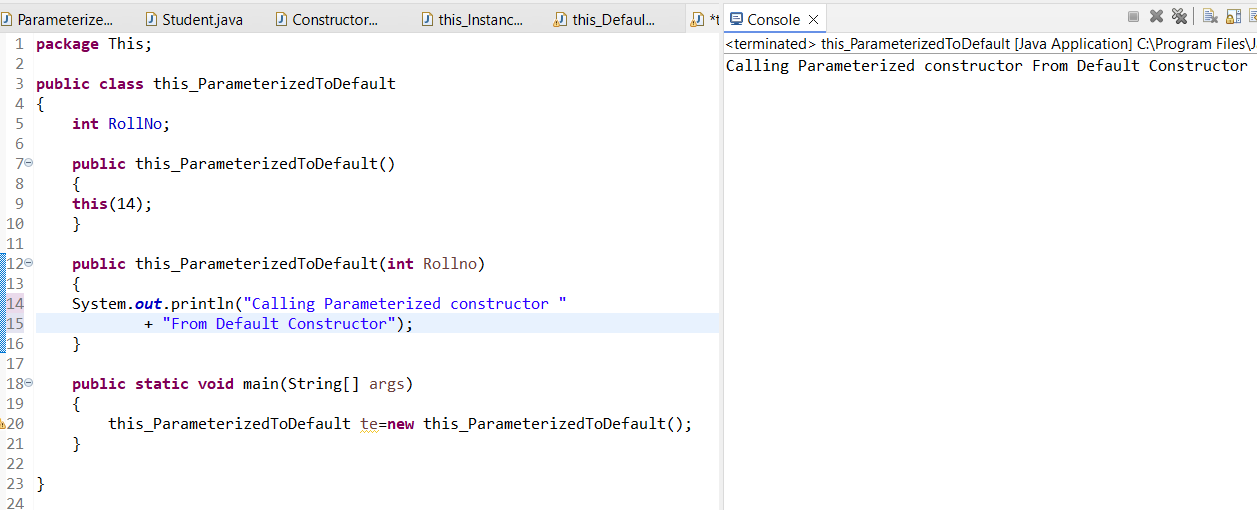
# Static & Instance variable memory allocation using constructor

# ‘this’ Keyword

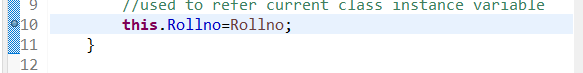
* this is a **reference variable** that refers to the **current class object**.
* **3 uses**:

1. Used to refer current class **instance variable –**

* when the parameter & instance variable has same name

1. Used to invoke current class **instance method**
2. Used to invoke current class **constructor**

# Debugging

* Debugging in Java involves **identifying, isolating, diagnosing, resolving**, and **verifying problems** in your code
* Here the **starting point** is called **break point**
* Steps:

1. Double tap on the line where we need to start the debugging process
2. Click on the debug icon to start

1 → Skip all Breakpoints: Ctrl+Alt+b

2 → Resume: F8

3 → Suspend

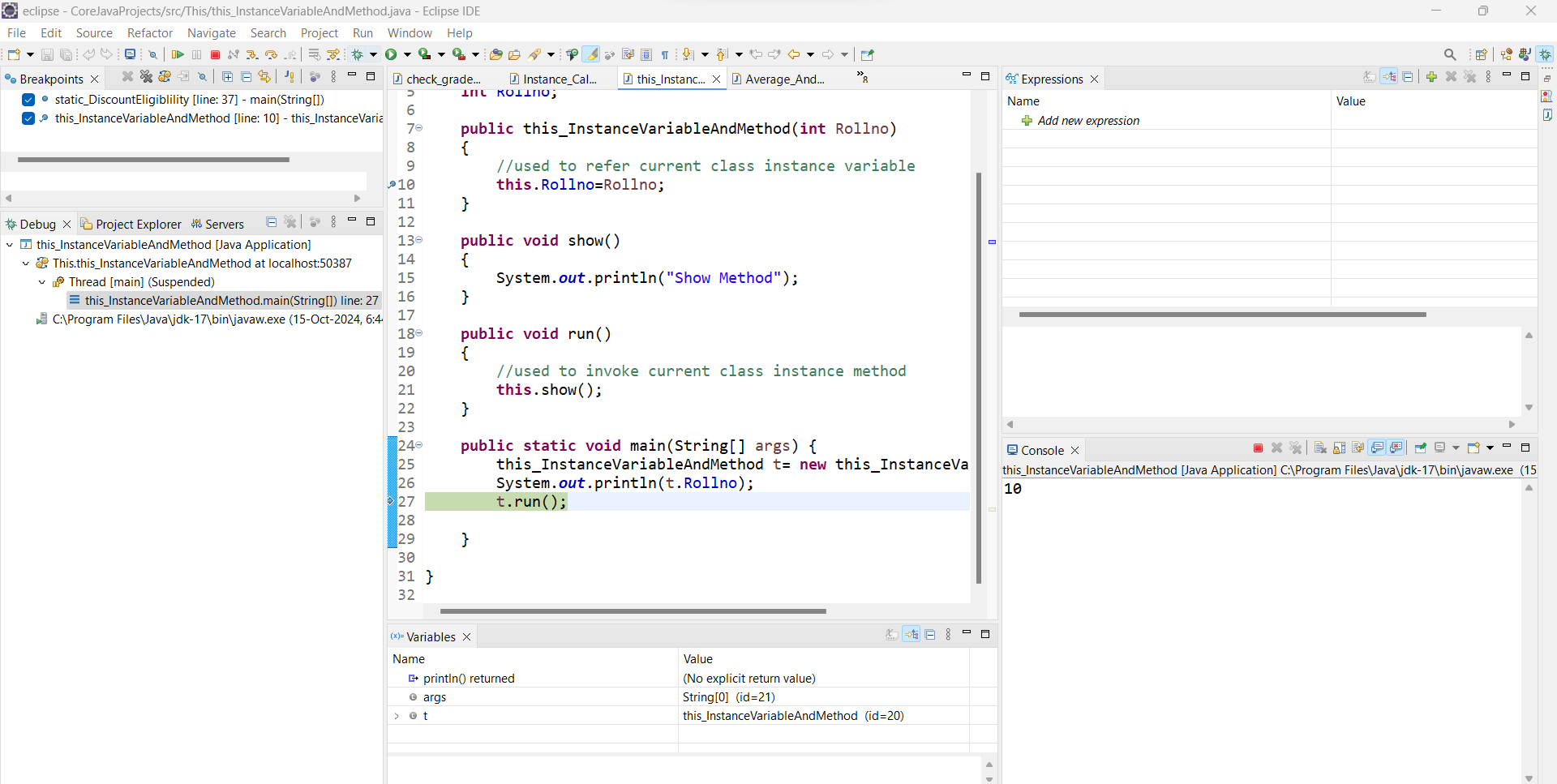
4 → Terminate: Ctrl+F2

5 →Disconnect:

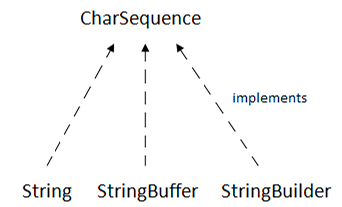
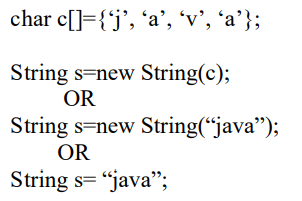
6 → Step into: F5 -- to go inside a method

7 →Step Over: F6 – to skip the inside of a method and goes to next line

8 → Step Return: F7 – same as Undo



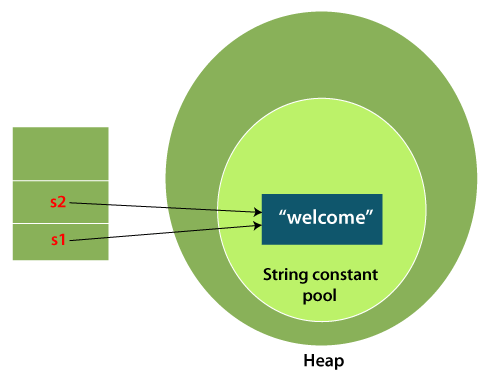
# Java String: Non primitive data type

* String is basically an **object** that represents **sequence of char values**.
* A String variable contains **a collection of characters surrounded by double quotes**
* We can create strings in java by using these three classes.
* The **CharSequence interface** is used to represent the sequence of characters - String, StringBuffer and StringBuilder classes implement it.
* **Two ways** to create string object:

1. By **new Keyword**

2. By **String Literal**

## By String Literal

* String s1=” Welcome”;

String s2=” Welcome”;

* Here **only one object** will be created. Firstly, JVM will **not find** any string object with the **value "Welcome"** in **string constant pool** that is why it **will create** a new object. After that it will **find the string** with the value "Welcome" in the pool, it **will not create** a new object but **will return the reference to the same instance**.
* **String objects** are stored in a **special memory** area known as the "**string constant pool**"
* Each time **you create a string literal**, the JVM **checks** the "**string constant pool" first**. If the string **already exists** in the pool, a **reference to the pooled instance is returned**. If the string **doesn't exist** in the pool, a **new string instance is created and placed** in the pool.
* Strings are **Immutable** i.e. **once it created, it cannot be changed or modified**.
* **Adv** → **memory Efficient**
* **Issue** →if we **change** value of any 1 string, it will **affect the other one** too. To **solve** this the java String are immutable.

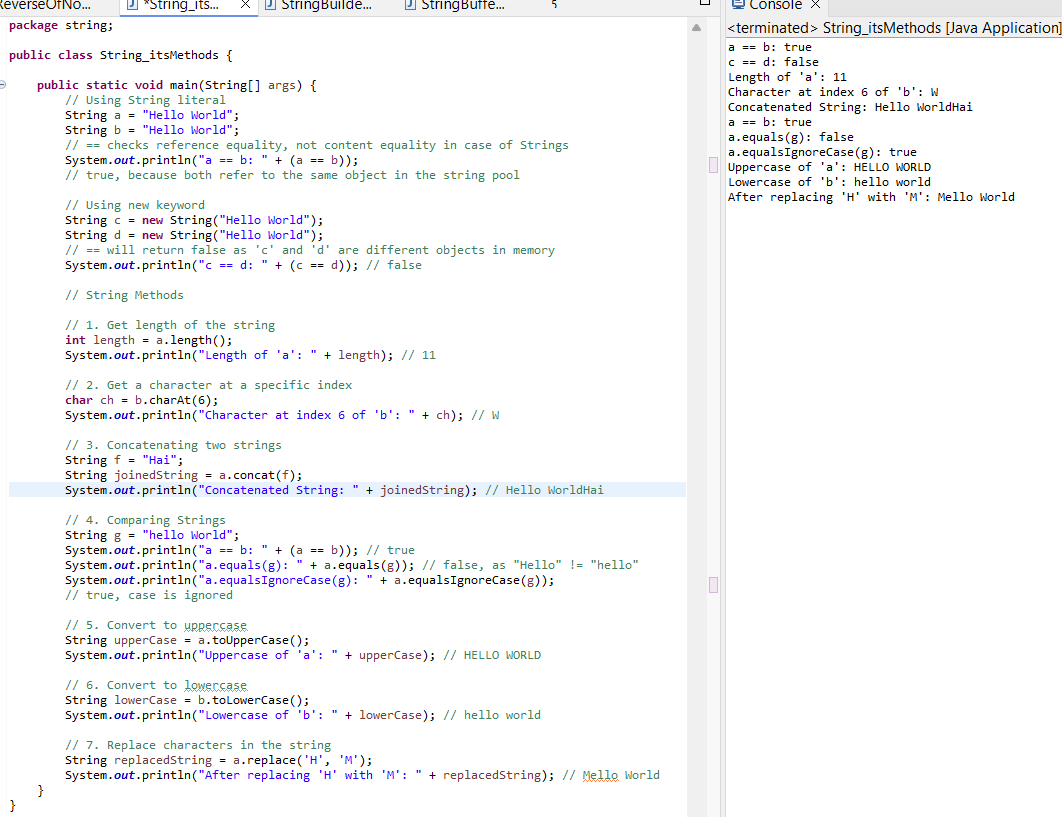
## By new Keyword

* String a =new String(“java”);

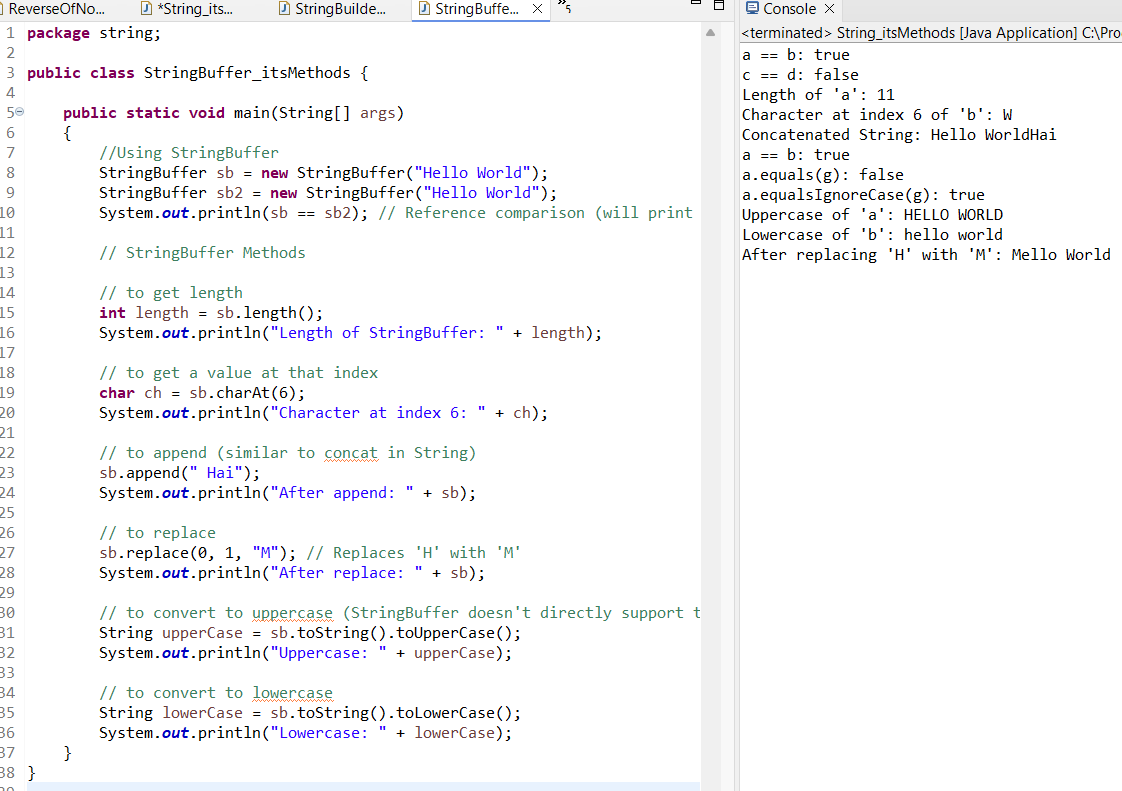
String a1=new String(“java”);

* Here, the JVM will **not check** if there any value exits or not. It **immediately creates** a new String object

## Difference between String & String buffer



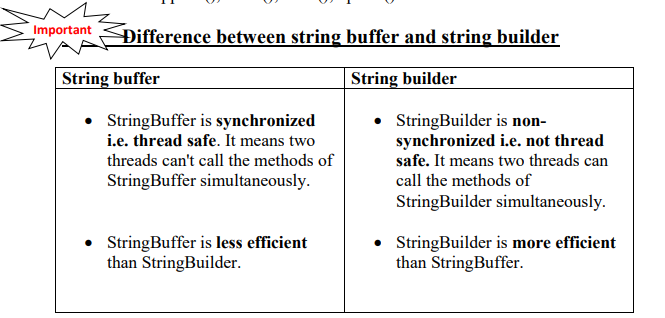
## String Buffer

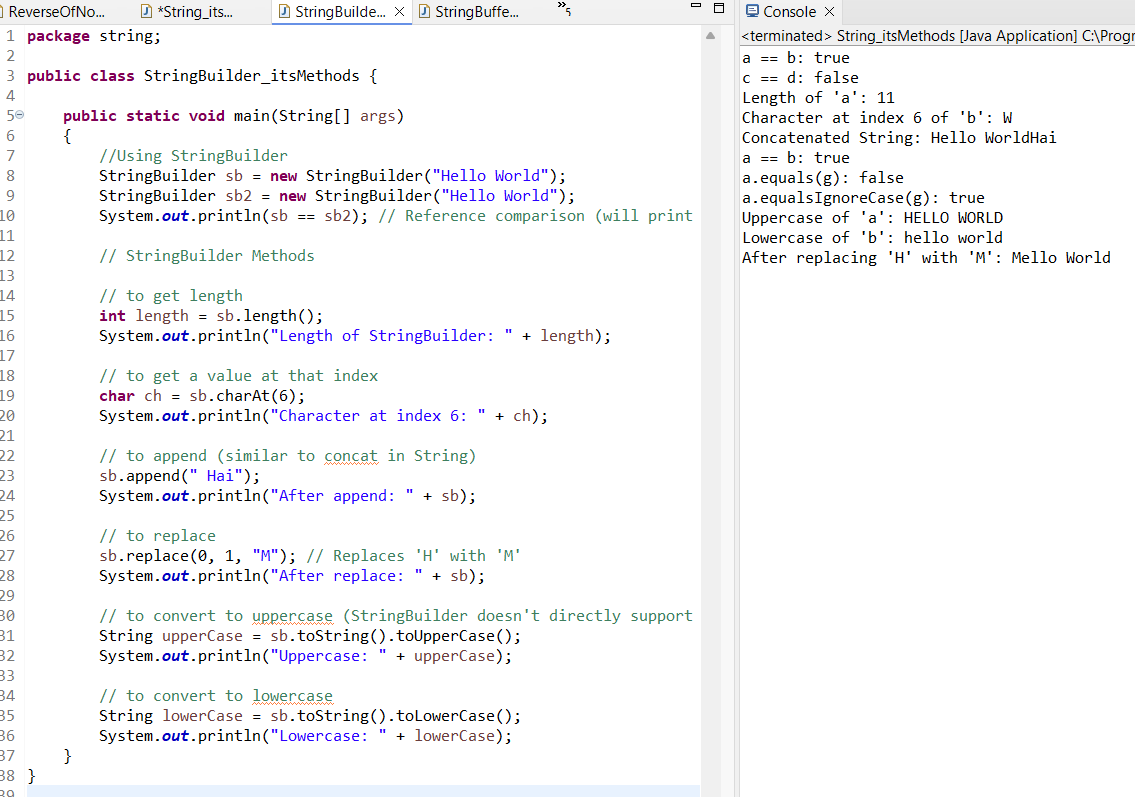
* Syntax: **String Buffer s=new String Buffer("Hello World”);**
* **Java String Buffer class** is used to **create mutable (modifiable) string**.
* Java String Buffer class is **thread-safe** i.e. multiple threads cannot access it simultaneously and synchronised

## String Builder

* Syntax: **StringBuilder s=new StringBuilder ("Hello World”);**
* **Java StringBuilder class** is used to **create mutable (modifiable) string**.
* The Java StringBuilder class is **same as String Buffer** class except that it is **non-synchronized** and **not thread safe**.

## Difference between String Builder& String buffer



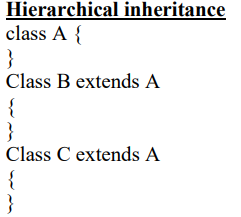
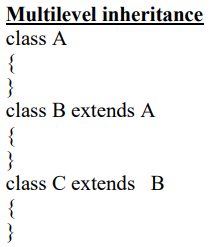
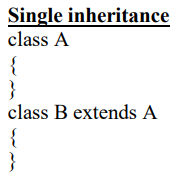


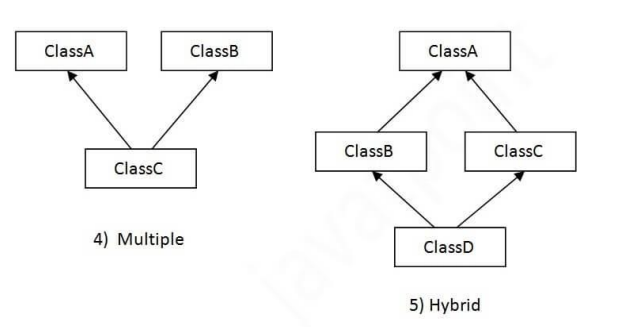
# Inheritance

* It is the **mechanism** in which **one class acquires** the **property of another class** i.e. new classes are built based on the existing classes.
* **one object** acquires all the **properties and behaviour of the parent object**.
* We can achieve inheritance using “**extends**” keyword
* We can reuse **fields(variable) and methods** of existing class (parent class).
* Inheritance represents **IS-A relationship** (**parent-child relationship**)
* **Use** of inheritance - **code reusability** & **method overriding**
* **3 types** –

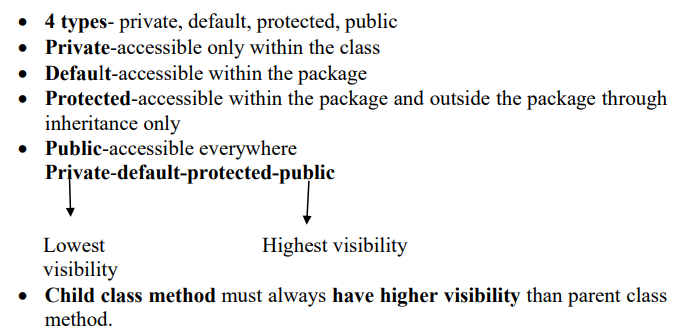
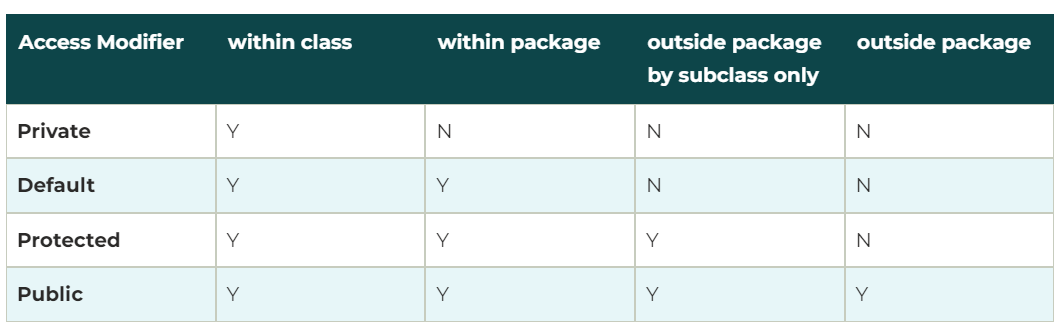
1. **Single** inheritance
2. **multilevel** inheritance
3. **hierarchal** inheritance

## Types of Inheritance

* Basically **3 types** supported by java -single, multilevel, hierarchal.
* **Hybrid and multiple** inheritance are **not supported in java** through **class** and is **supported only through interface**.



# Access Modifiers

* **Access modifiers** in Java specifies the **accessibility or scope** of a field, method, constructor, or class.
* ****

# ‘super’ Keyword

* Super is a **reference variable** used to refer **immediate parent class object**
* Super is **added** in each **constructor automatically by the compiler** if there is **no super() or this()**
* Super() must always be included as the **first statement** in method or constructor.
* **3 uses** :

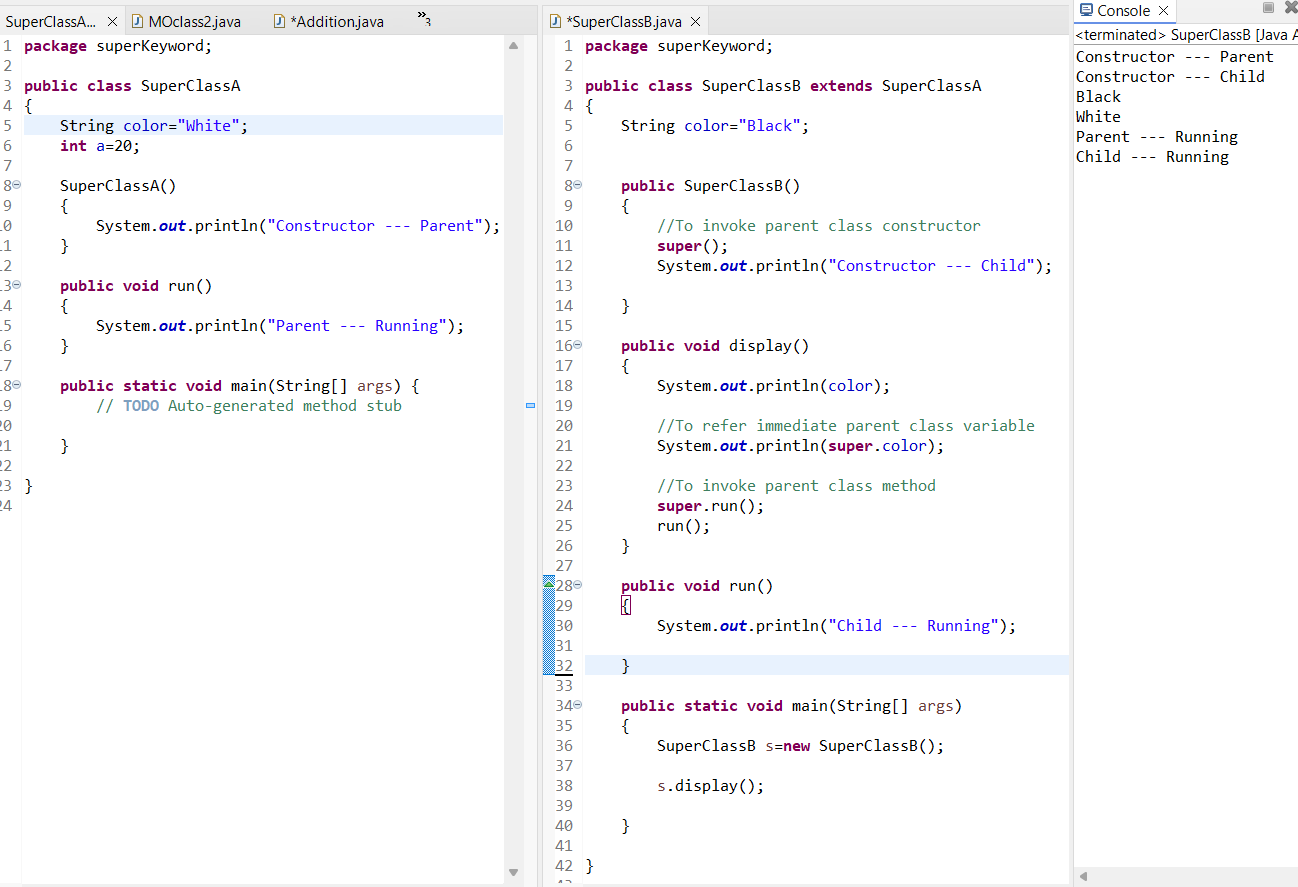
1. Super is used to refer immediate parent class **instance variable**

* when the **same variable** is used in the **parent and child** class as **instance variable.**
* syntax -- **super. Instance\_variable**

1. Super can be used to invoke parent class **method**:

* Syntax -- **super. Method\_name**
* When the child class has the same method as the parent class and the **method is overridden**, super keyword can be **used to invoke parent class method.**

1. Super is used to invoke the parent class **constructor**:

* syntax -- **super (constructor argument)**

# Difference between super() & this()

# Polymorphism

* Polymorphism is the concept in which a **single action** can be performed in **different ways**.
* **2 types** of polymorphism

1. **Compile time** polymorphism → Method **Overloading**

* also known as **static polymorphism** or **early binding**
* Type of polymorphism that occurs **during the compilation process**.
* It's achieved through **method or function overloading**, which allows **multiple methods or functions** to have the **same name** but **different arguments, signatures, or return type**

1. **Run time** polymorphism → Method **Overriding**

* also known as **dynamic or late binding polymorphism**
* Programming technique where the **compiler resolves an object during runtime** and **determines which function to associate** with it.
* It's implemented through **virtual functions and method overriding**

## Method Overloading

* Compile time polymorphism
* Same method name with different parameters.

## Method Overriding

* Process in which a **child class** will have the **same method** as declared in the **parent class** i.e. the **method name, number of arguments and data type** of the argument **remains similar** to the parent class.
* Uses - **allows a subclass to provide its own implementation of a method** that is already defined in its parent class
* Rules: -

1. The method must have the **same name** as in the parent class
2. The method must have the **same parameter** as in the parent class.
3. There must be an **IS-A relationship (inheritance**).

* **Static method** **canno**t be overridden since it is **invoked by class** and **not by object**. Hence static method cannot be inherited
* **Private and final method** **cannot** be overridden as they are **local to the class**
* **Java main method cannot** be overridden but **can be overloaded** provided a **string argument is passed.**
* **@Override annotation**
* is used **over methods re-defined in a chid class** which might be defined in the parent class
* Child class will have the **same method** as declared in the parent class i.e. the method name, number of arguments and data type of the argument remains similar to the parent class.
* **Uses** of override annotation

1. Improves **code readability**

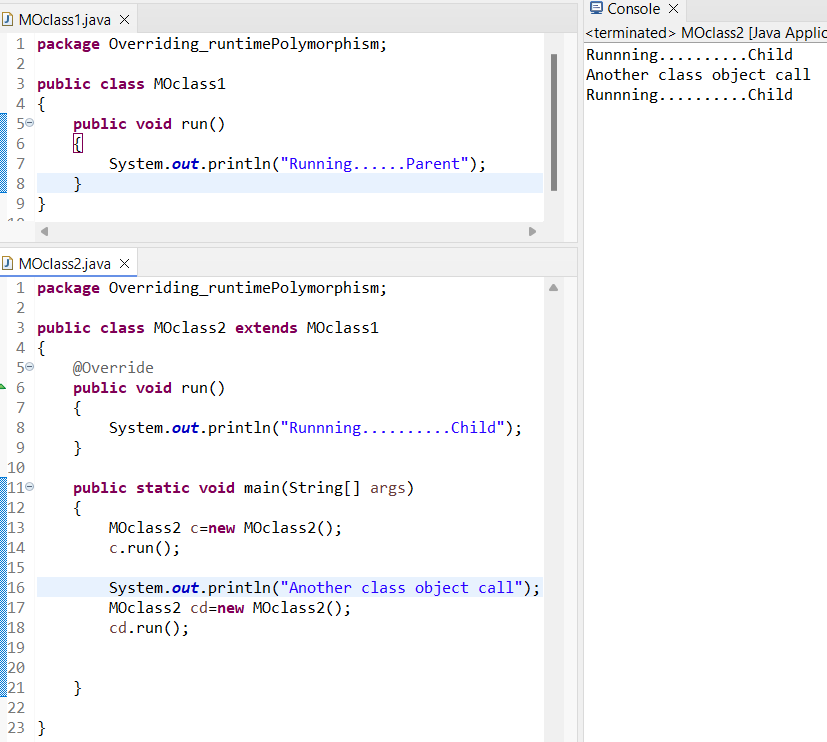
* One can **easily figure out** that this method also exists in a super class.

1. Ensures **perfect** overriding

* Applying @Override annotation **forces the compiler to check** if the arguments of the overridden method (number and their type) **match with that of super class**
* Also, whether its return type and access modifier match with that of super class.

1. **Prevents code break** due to maintenance activities.

* Suppose a method defined in a super class is overridden in a sub-class. Tomorrow, the method signature of super class changes to include one more argument.
* If the sub-class method has @Override annotation, you will get **compiler error** and hence **code can be corrected**. In absence of this annotation, again there will be **no compile time error** and the **method in sub-class** **will be a separate method which may result in runtime errors**.

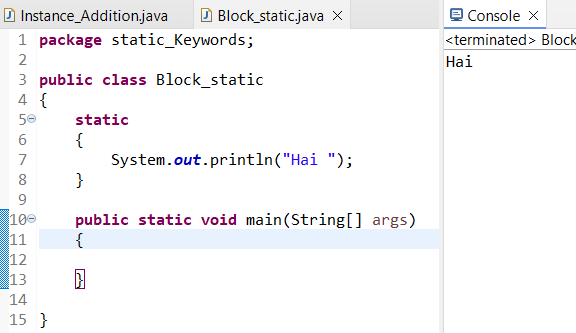


# Difference Between Method Overloading & Method Overriding

# “static” Keyword

* Used for **memory management**.
* Used to **share** the **same variable or method** of a given class
* The static keyword **belongs to the class** rather than an instance of the class.
* It is a **non-access modifier** in Java
* **characteristics** of the static keyword in Java:

1. **Shared memory allocation**: Static variables and methods are allocated **memory space only once** during the execution of the program. This memory space is **shared among all instances** of the class, which makes static members useful for maintaining **global state or shared functionality**.
2. **Accessible without object instantiation:** Static members can be accessed **without** the need to create an **instance of the class**. This makes them useful for **providing utility functions and constants** that can be used across the entire program.
3. **Associated with class, not objects:** Static members are associated with the class, not with individual objects. This means that **changes** to a static member are **reflected in all instances** of the class, and that you can access static members using the class name rather than an object reference.
4. **Cannot access non-static members:** Static methods and variables cannot access non-static members of a class, as they are **not associated** with any particular instance of the class.
5. **Can be overloaded, but not overridden**: Static methods can be overloaded, which means that you can define multiple methods with the same name but different parameters. However, they cannot be overridden, as they are associated with the class rather than with a particular instance of the class.

* It is used with

1. Blocks
2. Variables
3. Methods
4. Classes

## Static block

* it is used to **initialise a static variable** and is **executed before main method** at the **time of loading.**

## Static Variable

* It gets the **memory only once** in the class area.
* It can be used **to refer common property of object** such as college name etc

### **Difference between instance and static variable -🡪**

## Static method

* It gets **invoked by the class** and it **can access static variable** and **change** its value.
* **Restriction** on static method

1. It **cannot use non static /instance variable** or **call non static method** directly.
2. **this and super cannot** be used in static context.

# “final” Keyword

* used to **restrict the user** from editing the data.
* It can be used **along with variable, method and class**

## Final variable

* value will be **constant** and **cannot be changed**.
* A final variable which is **not initialise** is called a **blank final variable** and needs to be **initialised in a constructor or static block** in the **case of static method**

## Final method

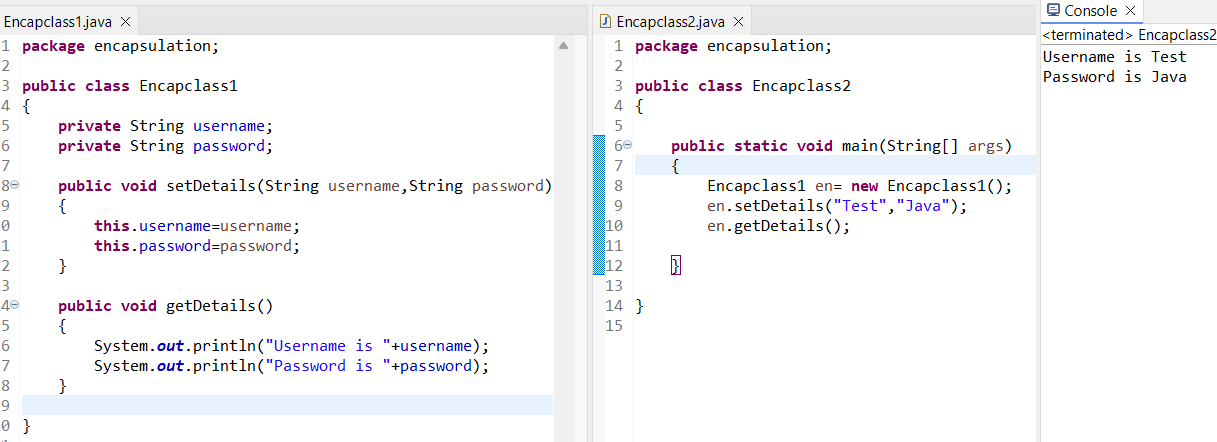
* If a method uses the **keyword final** before it then it **cannot be overridden**.
* If there are **2 methods** - **void run()** and **final void run()** it will result in **compile time error**.

## Final class

* If a class uses the keyword final before it then it **cannot be extended by inheritance.**
* **A constructor cannot** be final as it **is never inherited**.
* Example public static final int p=2; static final public final int p=2; instance final

# Difference Between static & final

# Encapsulation

* It is the mechanism of **wrapping code(methods) and data(variables) together** as a **single unit.**
* A **fully encapsulated class** is created by the following steps
* declare **all its variable** as **private**
* use **setter** and **getter methods** to **set(modify)** and **get(view)the data** in it.
* **Advantages**
* By using a getter or setter method it can be made as **read only or write only class**
* **Data hiding**: Other class will **not be able to access** the data through private variables and hence it acts as **protective shield**.
* **Control over data**: Setter method gives control over the data **as conditional statements can be used within it to select** specific data values.
* **Loose coupling** can be achieved by encapsulation.

# Abstraction

* Process of **hiding the internal details** and showing **only the functionality** to the user
* **2 ways** to achieve abstraction: -

1. **Abstract class**
2. **Interface**

## Abstract Class

* A class declared with **abstract keyword** is known as abstract class.
* It provides **0-100% data hiding** based on the methods used within the class
* The **methods** of an abstract class can be **abstract or non-abstract** methods.
* It provides **100% data hiding** if **all the methods are abstract**.
* Abstract class **cannot** **be instantiated**.
* **Abstract methods can** be defined **only in abstract class**.
* Abstract class will have **constructor and static** method.
* The **definition** of all the abstract method **must be provided** in the **extended class which is not abstract**.
* The abstract method will just include the **signature of the method without its definition**
* Abstract class **can have final methods**
* ‘**extends**’ keyword to call the abstract class in another class
* Syntax: **- public abstract class Class\_Name**

{

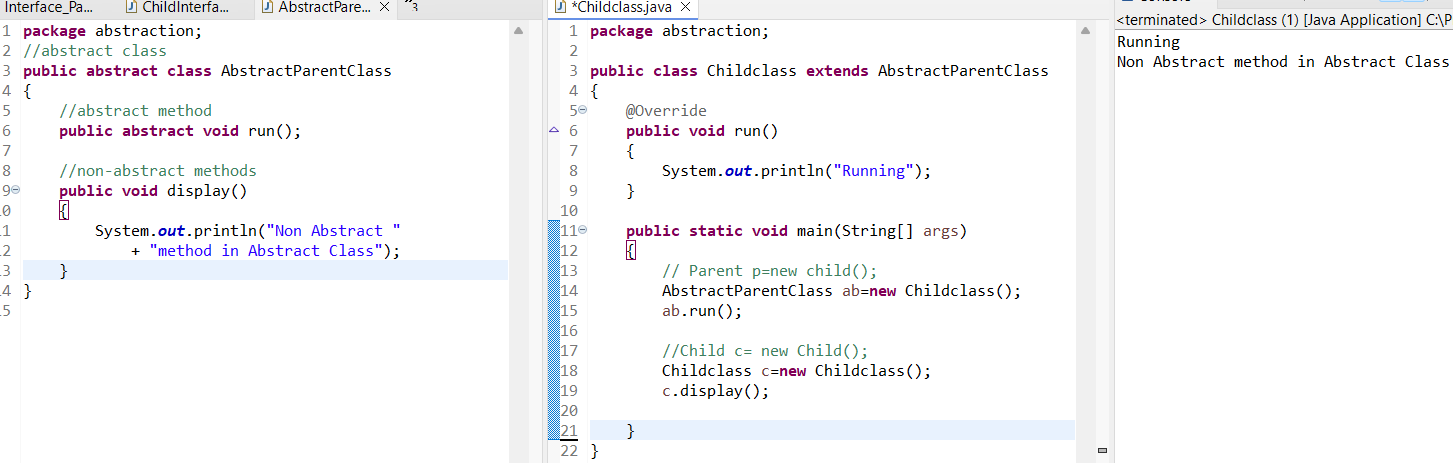
**public abstract return\_type Method\_name();** //Abstract Method

public return\_type Method\_name() //Non abstract Method

{

System.out.println(“Non abstract Method”);

}

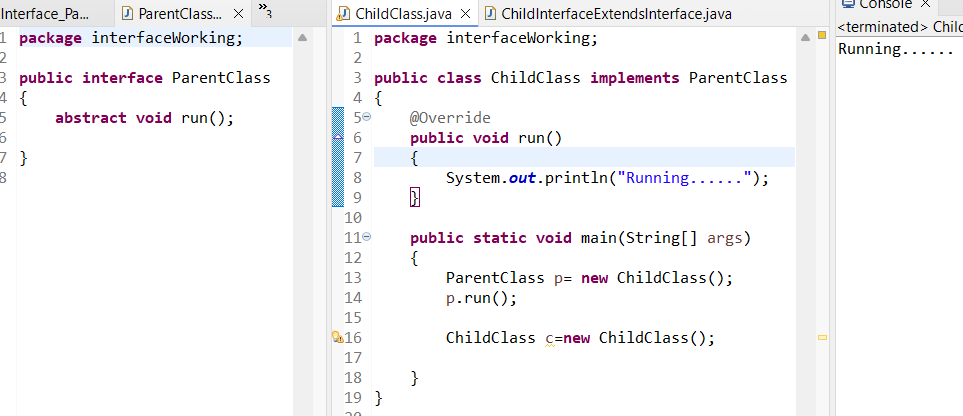
 }

## Interface

* ‘**implements**’ keyword to call the interface in another class
* Syntax: - **public interface Interface\_Name**

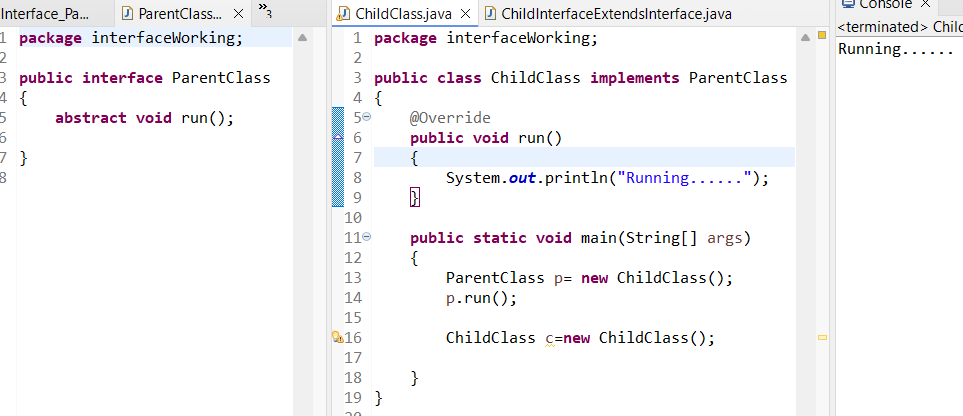
{

**public abstract return\_type Method\_name();**  //Abstract Method



}

* It is a **blueprint of class.**
* In interface **all methods** will be **public &abstract without method body**
* **Variables** will be **public static and final by default**.
* It does **not include instantiation** and **constructor**.
* \**Java 8** **supports defaul**t and static methods & **Java 9** supports **private method**s also in interface.
* **Uses** of interface

1. To achieve **100% data hiding** since all methods are abstract here.
2. It helps to **achieve multiple inheritance**.
3. It helps **to achieve loose coupling**.

(Loose coupling: It means reducing dependencies of a class that use different class directly)

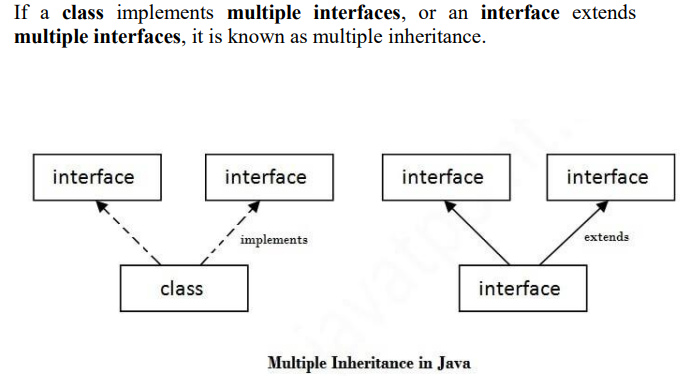
## Relationship between class & interface

* Class can be **either abstract or nonabstract** in the above figure. In both case the keyword “extends” and “implements” used in figure holds
* A class can **only extend one class**, but **it can implement multiple interfaces**.

|  |  |
| --- | --- |
| **Class** | **Interface** |
| Keyword 🡪 “**class**” | Keyword 🡪 “**interface**” |
| A class is a **blueprint for objects** that have methods and attributes. It **describes the behaviour** of an object and **stores data** | An interface is a **template** that **defines a group of related methods** with **empty bodies**. |
| **can be** **instantiated** i.e., objects of a class can be created. | **cannot be** instantiated i.e. objects cannot be created. |
| **implements** the **methods defined** by the **interface** | **does not implement methods**. |
| **do not support multiple** inheritance. | **supports** multiple inheritance |
| It can be **inherited** from **another class**. | It **cannot inherit** **a class**. |
| It can be inherited by another class using the keyword ‘**extends**’. | It can be **inherited by a class** by using the keyword ‘**implements**’ and it can be **inherited by an interface** using the keyword ‘**extends**’. |
| It **can** contain **constructors**. | It **cannot** contain constructors. |
| It **cannot** contain **abstract methods**. | It **contains** abstract methods **only**. |
| Variables and methods in a class can be declared using **any access specifier** (public, private, default, protected). | All variables and methods in an interface are declared **as public**. |
| Variables in a class can be **static, final, or neither.** | All variables **are static and final.** |

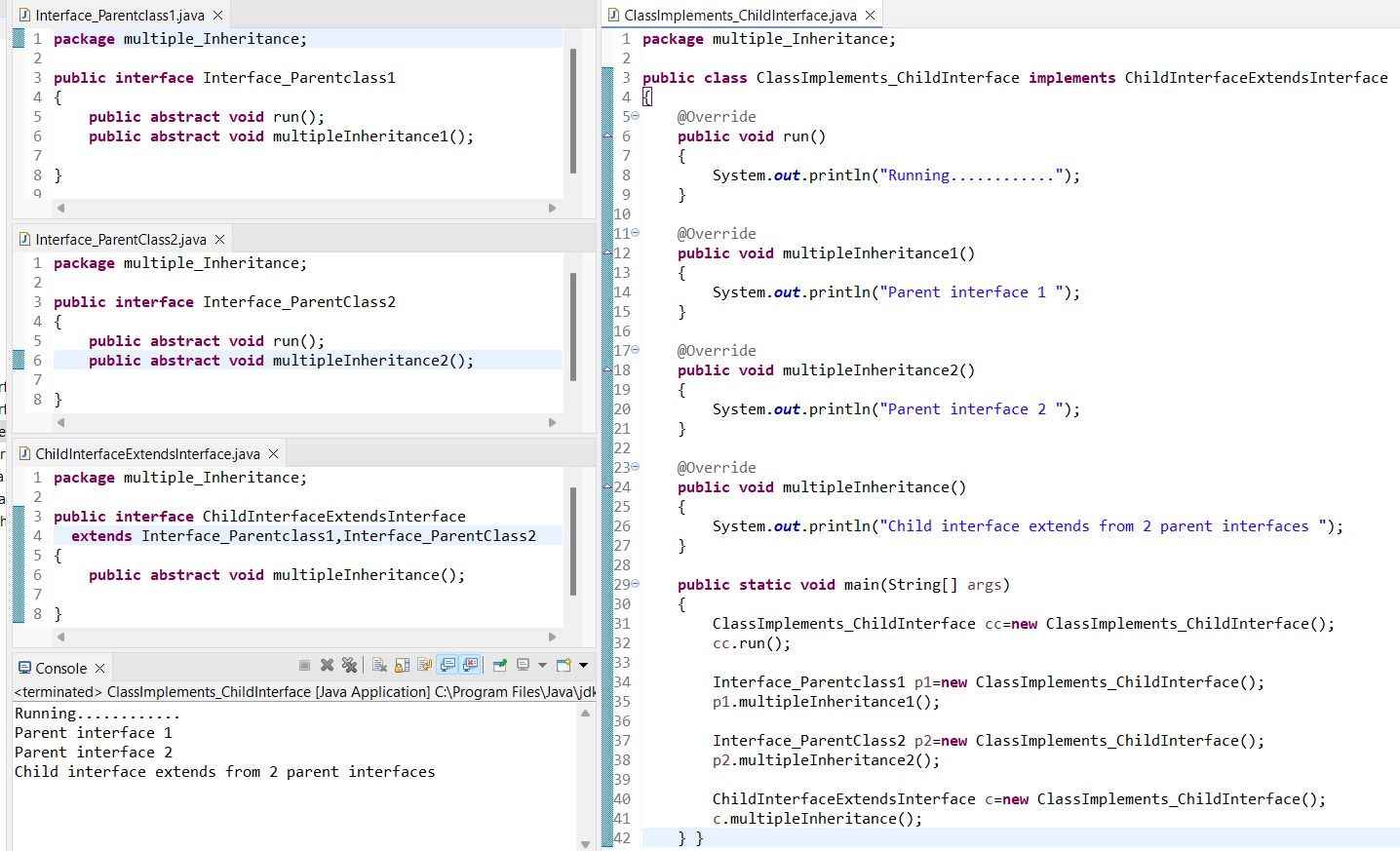
# Multiple Inheritance

* If a **class implements multiple interfaces**, or an **interface extends multiple interfaces**, it is known as multiple inheritance.

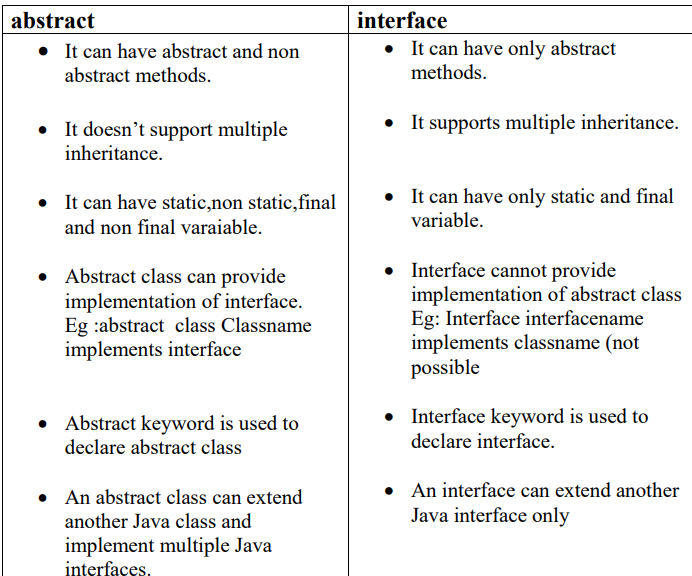
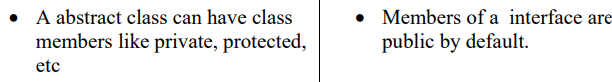


## Class implements multiple interfaces

## Interface extends multiple interfaces



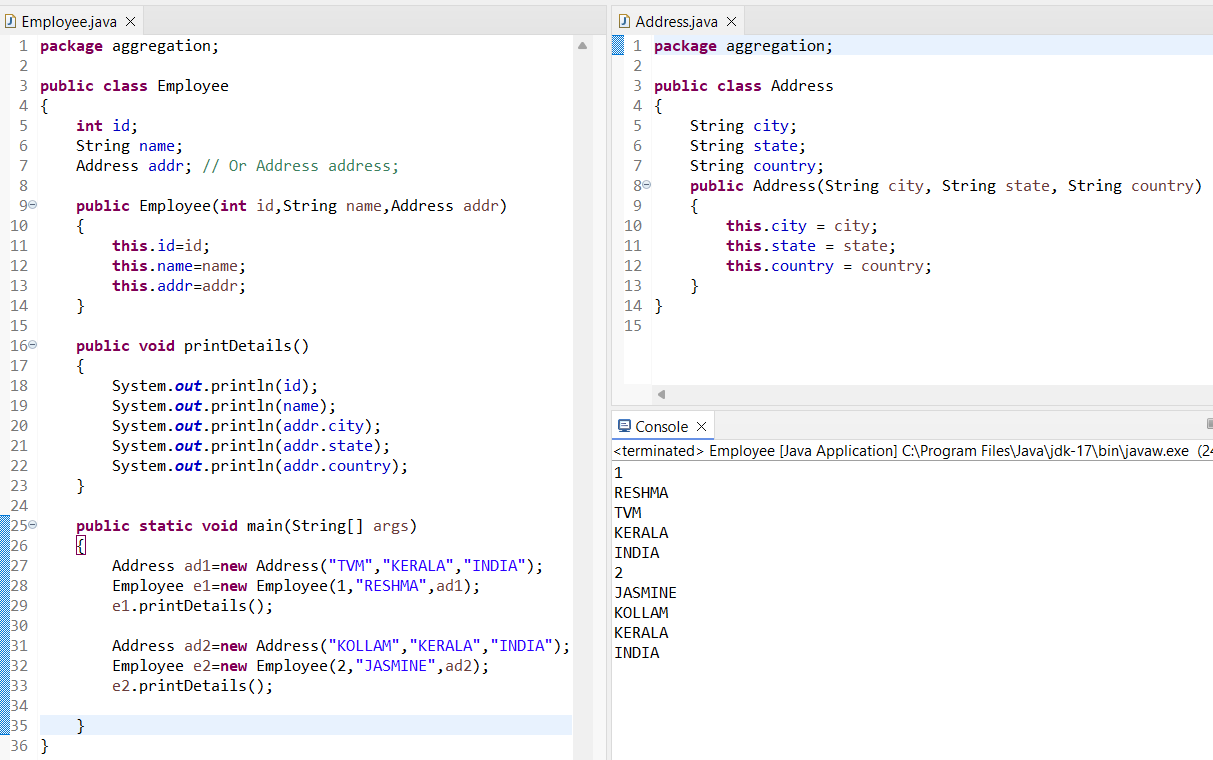
# Difference between abstract class and interface



# Difference between abstract and final

# Aggregation

* If a class have an **entity reference of another class** it is called **aggregation**
* **Use** of aggregation - **code reusability**.
* It represents **HAS-A** relationship.
* Wherever a method needs to be used **an object will be created** for the specific method with **reference to its class**



# Hierarchy of Java Collection frameworkCollection

* Java Collection means a **single unit of objects**.
* Package : **Java.util.**
* Collection in Java is a **framework** that provides an **architecture to store and manipulate** the group of objects.
* It has **Interfaces and its implementations**, i.e., classes & Algorithm
* The Collection interface is the **interface which is implemented by all the classes** in the collection framework.
* It declares the **methods that every collection** will have. In other words, we can say that the **Collection interface builds the foundation on which the collection framework depends.**

## Iterable Interface

* The Iterable interface is the **root interface** for **all the collection classes.**
* The Collection interface **extends the Iterable interface** and therefore all the subclasses of Collection interface also **implement the Iterable interface**.
* It contains **only one abstract method**. i.e.,

# List Interface

* list interface **extends** the **collection interface**.
* A list is used to **store ordered collection of data** and it **may contain duplicates**.
* Lists are **ordered collections** of data.
* Their ordered nature ensures that **their insertion order** remains intact.
* They can **be indexed and accessed** based on their positions.
* The list interface is **implemented** by **LinkedList, ArrayList, Vectors and Stack classes**.
* Syntax : **List<Data-Type> linkedlist = new LinkedList<Data-Type>();**

**List<Data-Type> arraylist = new ArrayList<Data-Type>();**

**List<Data-Type> vector = new Vector<Data-Type>();**

## ArrayList

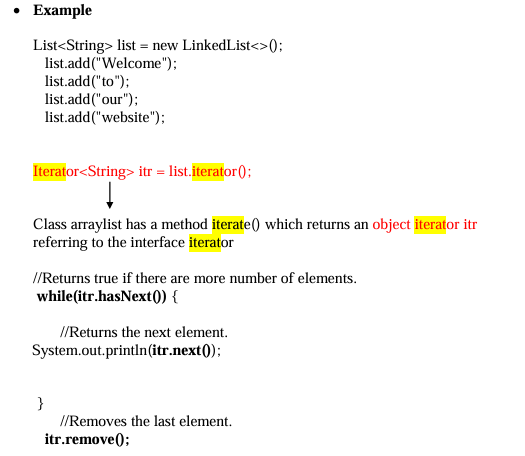
* ArrayList is **similar to Arrays**.
* They are also called **dynamic arrays**. That means it **does not have a fixed size**. Its size can be **increased or decreased** if elements are added or removed.
* It **implements** **the List Interface**.
* Since the ArrayList **cannot** be used for **primitive data types** like int, char, etc. , we need to **use a wrapper class**(Integer,String etc)
* ArrayList maintains the **insertion order** internally.
* can have the **duplicate elements**
* allows **random access** because the array works on an **index basis**.
* Normally we deal with generic collections
* Syntax :- **ArrayList<Data-Type> arraylist1 = new ArrayList<Data-Type>();**

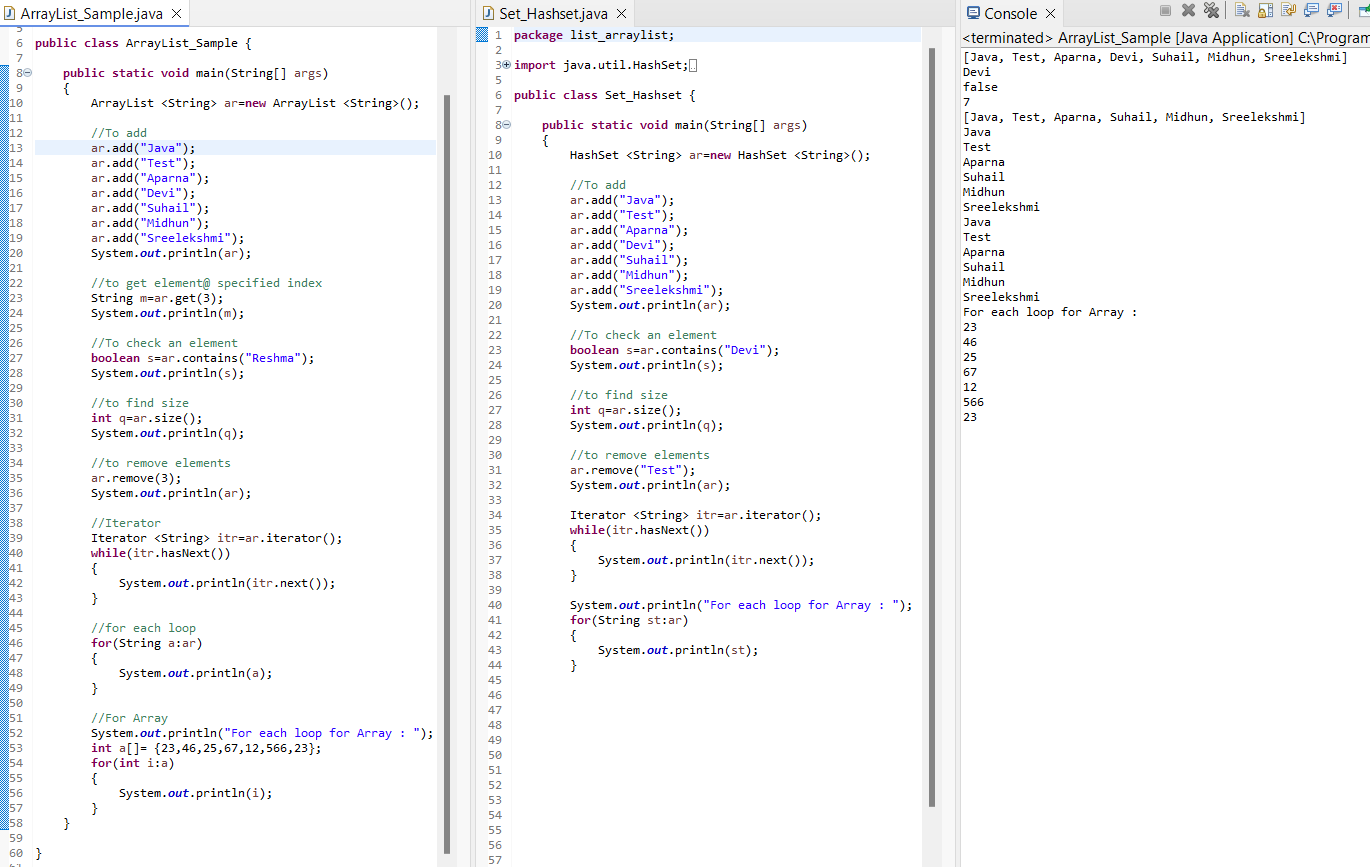
### Difference between Generic & non-generic ArrayList

## LinkedList

* **linked-list data structure**.
* Linked List is **a linear data structure** where the **elements** are called as **nodes**.
* Here, **each node** has **two fields- data and next**.
* **Data** stores the **actual piece of information** and **next points** to the **next node**. 'Next' field is actually the **address of the next node**.
* Elements are **not stored in a contiguous memory**, so **direct access** to that element is **not possible.**
* **LinkedList** uses **Doubly Linked List** to **store** its elements while **ArrayList** internally uses a **dynamic array** to **store** its elements.
* LinkedList is **faster in the manipulation** of data as it is **node-based** which makes it unique.
* Syntax : **LinkedList<Data-Type> linkedlist = new LinkedList<Data-Type>();**

# Iterator

* Iterator is an **object** that can **be used to loop through collections**, like ArrayList and HashSet.
* It is called an "iterator" because "**iterating**" is the technical term for looping
* To use an Iterator, you must import it from the **java.util package**.
* Interface with methods **hasNext(),next(),remove()**
* hasNext() - Returns a true value if the **more number of elements** are encountered during iteration.
* next() - Returns the **next specified element** during the iteration.
* remove() - Removes the last element from the collection as provided by the iterator.



# Set Interface

* Set Interface in Java is present in **java.util package**.
* It **extends** the **Collection interface**.
* It represents the **unordered set of elements** which **doesn't allow** us to store the **duplicate** items. We can store **at most one null** value in Set.
* Set is implemented by **HashSet, LinkedHashSet, and TreeSet**.
* Syntax : **Set<data-type> s1 = new HashSet<data-type>();**

## HashSet

* HashSet class **implements Set Interface**.
* It represents the collection that **uses a hash table for storage**.
* The hash table stores the values in an unordered method with the **help of hashing mechanism**
* Syntax : **HashSet<String> set=new HashSet<String>();**

# Difference Between List and Set interface in Java

|  |  |
| --- | --- |
| **List Interface** | **Set Interface** |
| * Allows Duplicate values | * Can’t have duplicate values |
| * Can be indexed & positionally accessed | * Cannot be indexed & positionally accessed |
| * Multiple null elements can be stored | * Only one null element can be stored at a time |
| * Insertion order is mainatained | * Doesn’t maintain Insertion order |

# Difference Between ArrayList & LinkedList

|  |  |
| --- | --- |
| **ArrayList** | **LinkedList** |
| * uses a **dynamic array** to store the elements. | * uses a **doubly linked list** to store |
| * **Manipulation** with ArrayList is **slow** because it internally uses an array. If any element is removed from the array, all the other elements **are shifted** in memory. | * Manipulation with LinkedList is **faster** than ArrayList because it uses a doubly linked list, so **no bit shifting** is required in memory. |
| * can **act as a list** only because it **implements List only.** | * LinkedList class can **act as a list and queue** both because it **implements List and Deque interfaces**. |
| * **better** for **storing** and **accessing** data. | * **better** for **manipulating** data. |
| * **contiguous** memory location | * **not contagious** memory location |
| * Generally, when an ArrayList is initialized, a **default capacity of 10** is assigned | * **no case of default capacity** in a LinkedList. In LinkedList, **an empty list** is created when a LinkedList is **initialized.** |
| * ArrayList is a **resizable array.** | * LinkedList **implements the doubly linked list** of the list interface |

# hierarchy of exception handlingException

* **Exception** is an **abnormal condition**.
* In Java, an exception is an event that **disrupts the normal flow** of the program. It is an **object** which is **thrown at runtime**.
* It **can be handled**
* Exception is **a class**
* **Advantage** 🡪 **to maintain the normal flow of the application**
* The **java.lang.Throwable class** is the root class of **Java Exception hierarchy** inherited by **two subclasses**: **Exception** and **Error**.

## Types of Java Exception

* Mainly **two types** of exceptions:

1. **Checked**
2. **Unchecked**

* An **error** is considered as the **unchecked exception**.
* **Error** is **irrecoverable**
* However, according to Oracle, there are **three types of exceptions** namely:

## hierarchy of exception handlingChecked Exception

* Also called **Compile time Exception**
* **Checked exceptions** are the ones that are checked **during compilation time**.
* Exceptions that must be **either caught or declared** in the method signature using the **throws keyword**.
* These exceptions are typically used to **handle expected error scenarios** that a program can recover from.

### Types of Checked Exception

1. ClassNotFoundException :

* Occurs when a Java virtual machine **(JVM) tries to load a class** but **fails** because it is **unable to find the classpath**.

1. InterruptedException :

* Occurs when a **thread is interrupted** while **waiting or sleeping or occupied in some task**. During this, a **method interrupt()** is been **called by the code** in the thread which **blocks the I/O operations**.

1. IOException :

* It is an **input/output exception** and it occurs whenever an **input or output operation fails or is interpreted** in java.
* For instance, when you try to read a file that does not exist, it will throw an IOException.

1. SQLException :

* This exception provides **detail about SQL database error** if it occurs.
* The error can be **SQL syntax or SQL driver.**

1. FileNotFoundException :

* It occurs when a **file** that we are **trying to find is unavailable** in the directory.
* It is **available in class java.io** and is a checked exception because it is **thrown by the constructor RandomAccessFile at run time**.

## Unchecked Exception

* Also called **Run time Exception**
* An unchecked exception is something that has **gone wrong with the program** and is **unrecoverable**.
* It occurs at the **time of execution**
* It includes **bugs**, **improper usage of API**, and **syntax** or **logical error** in programming.

### Types of Unchecked Exception

1. Arithmetic exception :

* Type of **unchecked error in code** that is thrown whenever there is a **wrong mathematical or arithmetic calculation** in the code, especially during run time.
* For instance, when the **denominator in a fraction is zero**, the arithmetic exception is **thrown**.

1. NullPointerException :

* It is thrown when a **null value is assigned to a reference object** and the program **tries to use** that null object.

1. ArrayIndexOutOfBoundsException :

* It occurs when we **access an array with an invalid index**.
* This means that either the index value is **less than zero or greater** than that of the **array’s length**.

1. NumberFormatException :

* It occurs when we are trying to **convert a string to an int or other numeric value**.
* This exception is thrown in cases when it is **not possible to convert** a string to other numeric types.

1. InputMismatchException :

* It occurs when an **input provided by the user is incorrect**.
* The type of incorrect input can be **out of range or incorrect data type**.

1. IllegalStateException :

* It occurs when a **method of a code is triggered or invoked** at the **wrong time**.
* This exception is **used to give a signal** that the method is invoked at the wrong time.

# Exception Handling

* The **Exception Handling** in Java is one of the powerful **mechanism to handle** the **runtime errors** so that the **normal flow** of the application can be **maintained.**

1. Java try-catch block

* **try block** is used to **enclose** the code that might **throw an exception**
* It must be used **within the method**
* If an **exception occurs** at the particular statement of try block, the **rest** of the block code **will not execute**. So, it is better **not to keep** the code in try block that **will not throw an exception**.
* try block must be **followed by** either **catch or finally** block.
* **catch** block is used to **handle the Exception** by declaring the **type of exception** within the **parameter**
* The catch block **must be used after** the **try block only**
* A try statement can have **multiple catch blocks**.
* A try catch block comes only **within main method**.
* JVM has a **default try catch** block **which handles exception** in case the **exception is not handled**
* Syntax : try

{

//code that may throw an exception

}

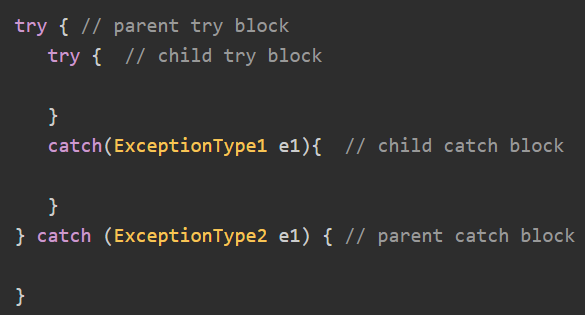
catch(Exception\_class\_Name ref)

{

}



1. Multiple catch block

* At a time **only one exception** occurs and at a time **only one catch block** is executed.
* All catch blocks must be **ordered** from **most specific to most general**, i.e. catch for ArithmeticException must come before catch for Exception,else it results in compile time error
* In above case there are **two exceptions within the try block**, the exception which is **listed first** in the catch block **alone will be handled**.
* In case there are **two exceptions** and both are **not listed** in catch block then the catch block with the **general class exception gets invoked**.

1. Java nested try block

* Sometimes a situation may arise where a **part of a block** may cause **one error** and **the entire block** itself may cause **another error**. Here we make use of nested try block.

1. Java finally block

* **finally block** is a block that is used to **execute important code** such as closing connection, stream etc
* Finally **ensures mandatory execution** of code within it.
* Java finally block is always **executed whether exception is handled or not**.
* Java finally block **follows try or catch block**
* If you **don't handle exception**, before terminating the program, **JVM executes finally block(if any)**.
* **For each try** block there can be **zero or more catch** blocks, but **only one finally** block.

1. Java throw keyword

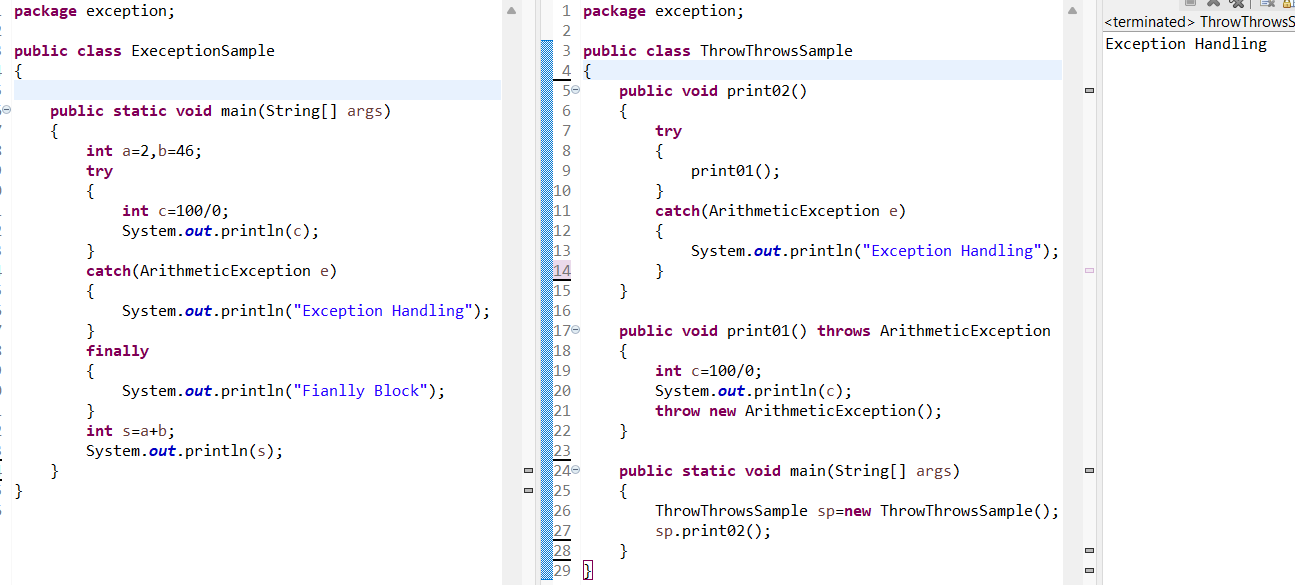
* **throw keyword** is used to explicitly **throw an exception**
* Followed by an **instance**
* It can be used to throw **checked or unchecked** exception
* Syntax :- **throw new exception**();
* Used **within method**
* It is used mainly for **custom exceptions(exceptions created by us)**

1. Java throws keyword

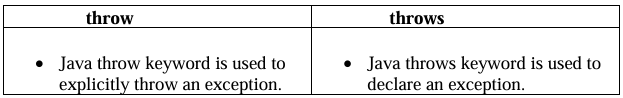
* It is used to **declare an exception** in the **method signature**
* It gives **an information to the programmer** that there **may occur** an exception
* It is mainly **used to handle** checked exception as unchecked exception handling is programmers responsibility
* Exception is **thrown from method to main** and **main to JVm**
* When it is thrown from method, try catch block within **main should handle** it.
* When **thrown from main** ,**JVM handles** it by using the **default try catch block**
* Syntax :- **returntype methodname() throws exceptionclassname**

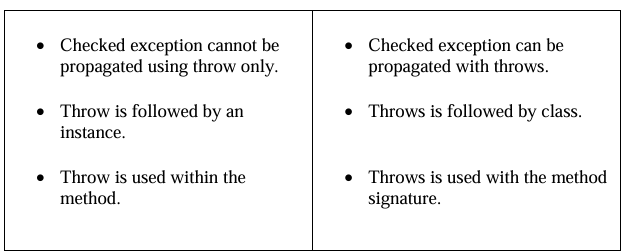
**{**

**//method code**

** }**

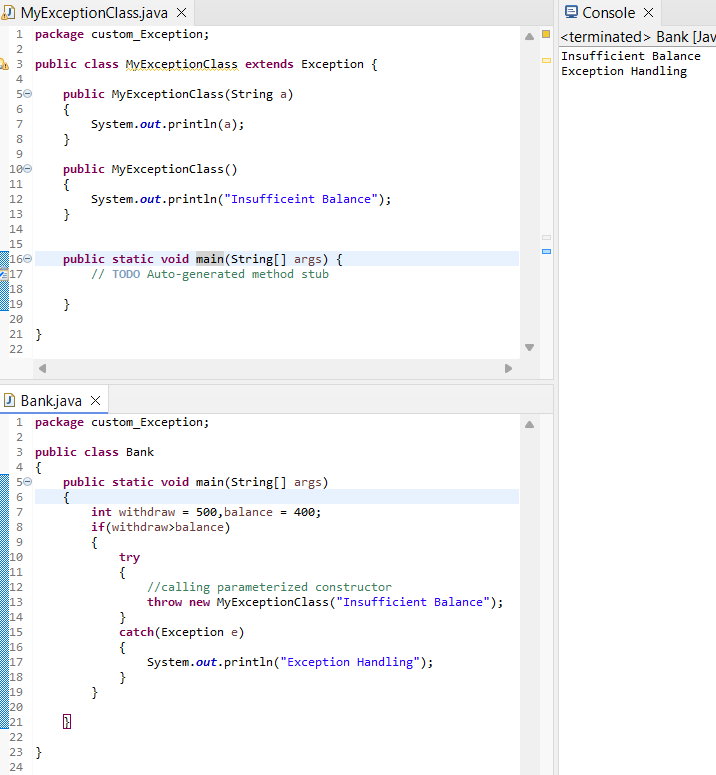
# Difference between throw and throws



****

# Custom Exception

* custom exception refers to the **creation of your own exception** to **customize** an exception according to the needs.
* The custom exceptions are **derived** from the **Exception** class.
* Syntax :- **class MyException extends Exception { }**



# Difference between final, finally and finalized

# Git hub

* **version control system** widely used for **tracking changes in source code** during software development.
* A **Git repository** is a **virtual storage** of your project. It allows you to **save versions** of your code, which you can **access when needed**

## How to generate Tokens

* **Personal access tokens** are an **alternative to using passwords** for authentication to GitHub when using the GitHub API or the command line.
* Personal access tokens are intended to **access GitHub resources** on behalf of yourself
* Creation: -

**click on Profile → Settings →Developer settings → Personal access Tokens(classic) →**

**Token(classic) →Generate new Tokens(classic)**

Type note name – note

Set Expiration – no expiration

Select required scope –select all scope **→ Generate Token**

**Copy the URL it can we used in turn of password while connecting to local**

( ghp\_q9C1HTxO9w2CkR8q0u9xqAYKsUzuch3vE13N )

## Create Repository

In Home **→ Create a new repository**

Repository Name: Test

Public/private: public

Description: my new repo

Add ReadMe. File **→ Create Repository**

## Creating new branch

**Click on the drop down on Main**

Type name: test **→Create branch**

## Cloning a Project

**Click on the drop down on Main → Copy the Https URL**

In eclipse:

**Window → Show View → Other → Git → Git Repository**

In git repositories tab of eclipse:

**Clone a Git repository →Clone URL → Select Branch → Finish**

To add a project to repo:

**Right click on project → Team →Share Project →Choose Repository**

**from the drop down → Finish**

To Commit & Push

**Select on Project → Window →Show View → Git →Git saging →drag**

**& place or add all (++)**

Staged -- code that are needed to push

Unstaged – All projects on local but not in git

**Type commit message →Commit & Push → username & password →Log in**

Compare & Pull request (from branch to main branch)

To pull – from git to local or from a branch to main branch

**Create pull request** **→ Assignees: assign to yourself → Merge**

**pull request → Confirm Merge**

Pull request (from git to local)

**Right click on project → Team →Pull**

# Maven

* Apache Maven is a **software project management and comprehension tool** that helps with **project management** and **comprehension**.
* Based on **POM (project object model)**
* **Used** to **build**, **publish**, and **deploy** projects
* Mostly used for the **java projects to build web application** packages.

## POM Files:

* Project Object Model (POM) Files are **XML file** that contains **information related** to the **project and its configuration** such as **dependencies, source directory, plugin, goals** etc, **used by** Maven **to build the project**.
* when you should execute a maven command you give maven a POM file to execute the commands.
* Maven reads pom.xml file to accomplish its configuration and operations.

## Dependencies and Repositories:

* **Dependencies** are **external Java libraries** required for Project
* **Repositories** are **directories of packaged JAR files**.
* **3 types** of Repositories**:**

1. **Local**
2. **Central**
3. **Remote**

* **Local repository**: A directory on the **computer** where Maven **runs those caches remote downloads** and contains **temporary build artifacts**.
* **Remote repository**: A **collection of projects** from which Maven populates the local repository. Or it’s a **repository of a remote server**
* **Central Repository**: A repository that contains a **large number of commonly used libraries**.

Mainly web.

* On the **1st run command** it checks **for local repositories & dependencies** If the **dependencies are not found** in the local Maven repository, Maven downloads them **from a central Maven repository** and **puts them** in your **local repository**.

## Build Life cycles

* A build life cycle consists of a **sequence of build phases**, and **each build phase** consists of a **sequence of goals**.
* Build --- Clean --- Install

## What is Maven Used For?

1. We can easily **build a project** using maven.
2. We can **add jars and other dependencies** of the project easily using the help of maven.
3. Maven **provides project information** (log document, dependency list, unit test reports, etc.)
4. Maven is very helpful for a project **while updating the central repository of JARs and other dependencies**
5. with the help of Maven, we can build **any number of projects into output types** like the JAR, WAR, etc **without doing** any **scripting**.
6. Using maven, we can **easily integrate our project with a source control system (such as Subversion or Git)**.
7. Maven also helps in **managing the project's build lifecycle**, including tasks like **compiling, testing, packaging**, and **deploying** the code.
8. Maven provides a **standard project structure**, making it easy for developers **to understand the layout** of the project and **locate specific files.**
9. Maven supports **multi-module projects**, allowing developers to **work on multiple related projects simultaneously** and **manage their dependencies efficiently**.
10. **Maven plugins** can be used to add **additional functionality to the build process**, such as code coverage analysis, static code analysis, and more.

## Creation

* **File → New → Other → Maven →Maven Project**

**Type any GroupId & ArtifactId → Finish**

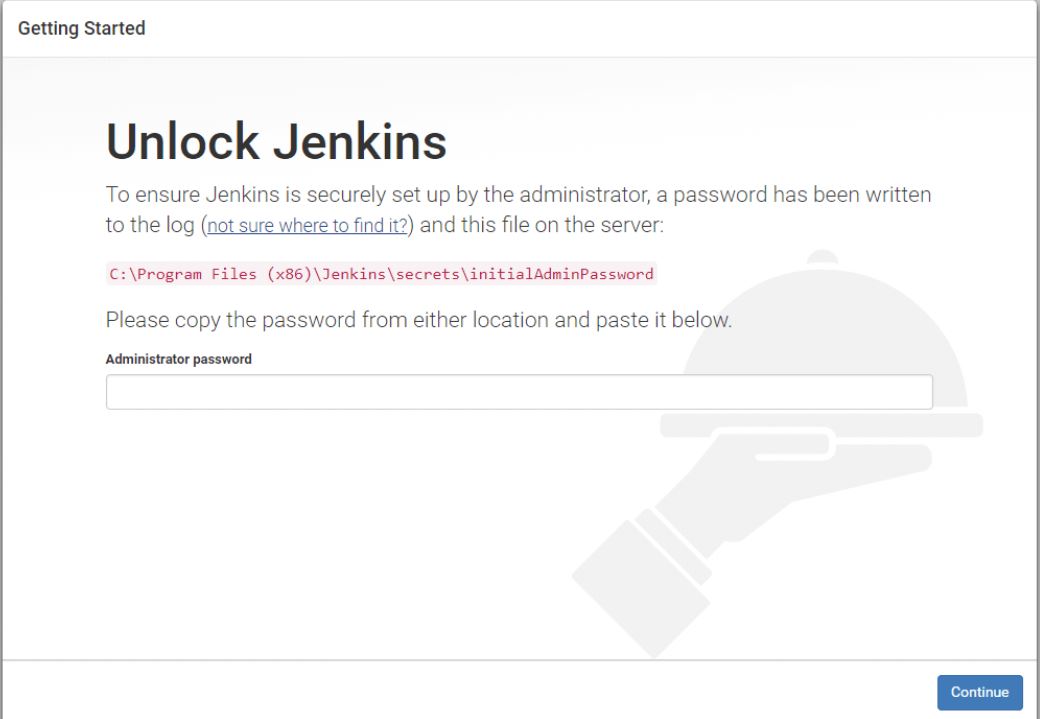
(Used to identify the project)

# Jenkins

* **Jenkins** is a tool that is used for **automation**.
* It is mainly an **open-source server** that allows all the developers to **build, test** and **deploy software**.
* It is **written** in **Java** and **runs** on **java only**.
* By using Jenkins we can make a **continuous integration of projects(jobs)** or **end-to-endpoint automation**
* Jenkins facilitates the **automation of several stages** of the **software development lifecycle**, including **application development, testing, and deployment**.
* **Continuous delivery (CD) and integration (CI) pipelines** can be created and managed with Jenkins.
* The development, testing, and deployment of software applications are **automated using CI/CD pipelines**.
* jenkins is **flexible**.
* Can add the **n no.of plugins** you want to add to the jenkins.
* Can **automate the proceses of CI/CD pipelines** of all the projects.
* **CI & CD** is the **practice** of **automating the integration of code changes** from **multiple developers into a single codebase**. It is a **software development practice** where the developers **commit** their work frequently to the **central code repository** (Github or Stash). Then there are **automated tools that build** the newly committed code and do a code review, etc as required upon integration.
* In computing, a **pipeline** is a **set of stages or processes** linked together to form a processing system. **Each stage** in the pipeline takes an **input, processes** it in accordance with a set of rules, and then **sends the outputs** to the stage that follows. Frequently, the **pipeline’s overall output** is its **final step’s output.**

## **Installation of Jenkins in windows:**

1. Before installing Jenkins you should have **already installed java** on your device
2. Download Jenkins 2.482 for: <https://www.jenkins.io/download/>
3. After download, open the file and install it.
4. Browse to http://localhost:8080 (or whichever port you configured for Jenkins when installing it) and wait until the **Unlock Jenkins** page appears.



1. The initial Administrator password should be found under the location provided in the screen.
2. Copy that address location and paste it on the search bar of the system,then enter.A file named ‘initialAdminPassword’ will be open(open with notepad) copy the content of the file ,that will be the default password( for one time use only) & paste it at the Administrator password menu of the Jenkins page.
3. 🡪 Continue 🡪 Install suggested Plugins
4. Set Username,Password,Fullname & email-id

Username : Reshma Treesa

Password : Reshma@Treesa

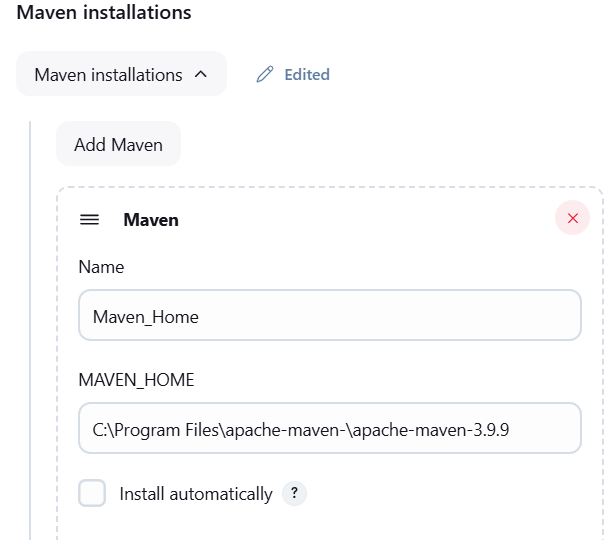
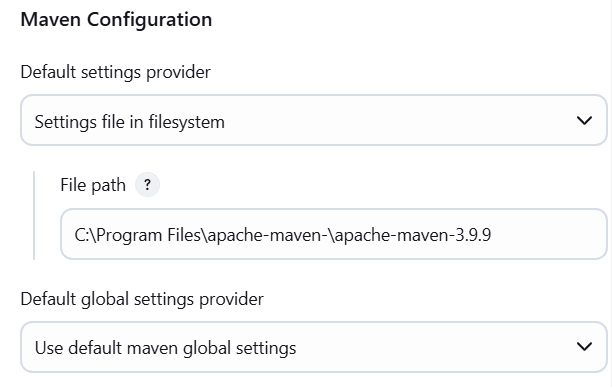
## Strating with Jenkins

* Jenkins uses **projects** (also known as "**jobs")** to perform its work. Projects are **defined and run** by Jenkins users
* New item 🡪 to create new project file

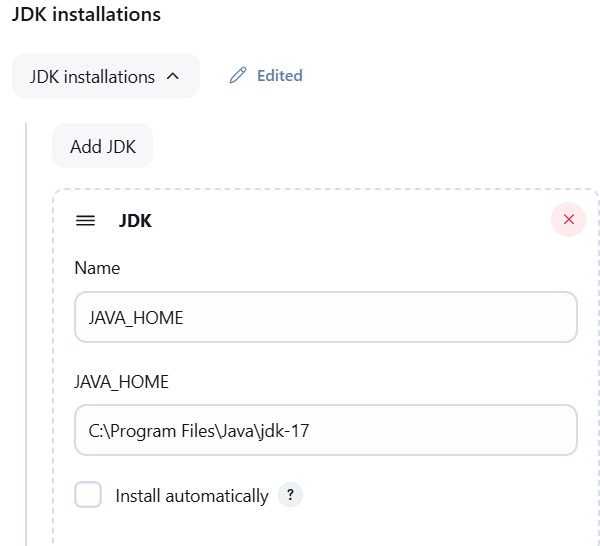
Manage Jenkins 🡪 to set up the related paths required for the project file.

* To add maven project dependencies :-

1. to add maven file

 Manage Jenkins 🡪 Tools 🡪 In maven Configuration/In Maven installation

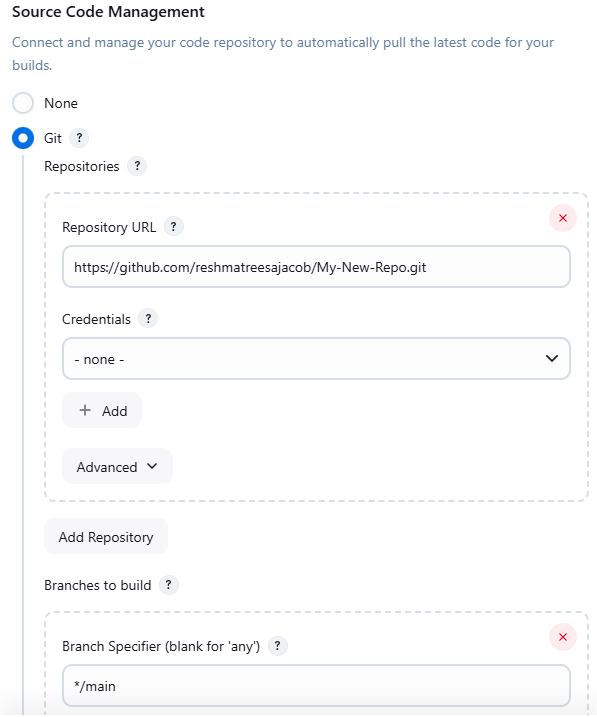
1. To add jdk path



Path 🡪 copy path from the environment variable tab.

1. Then save

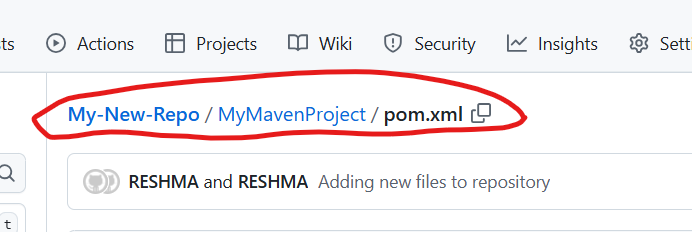
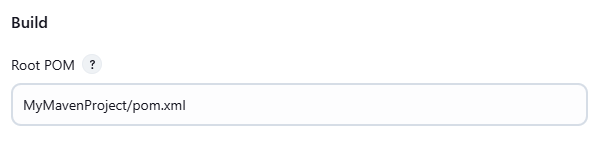
* To create a maven project file :

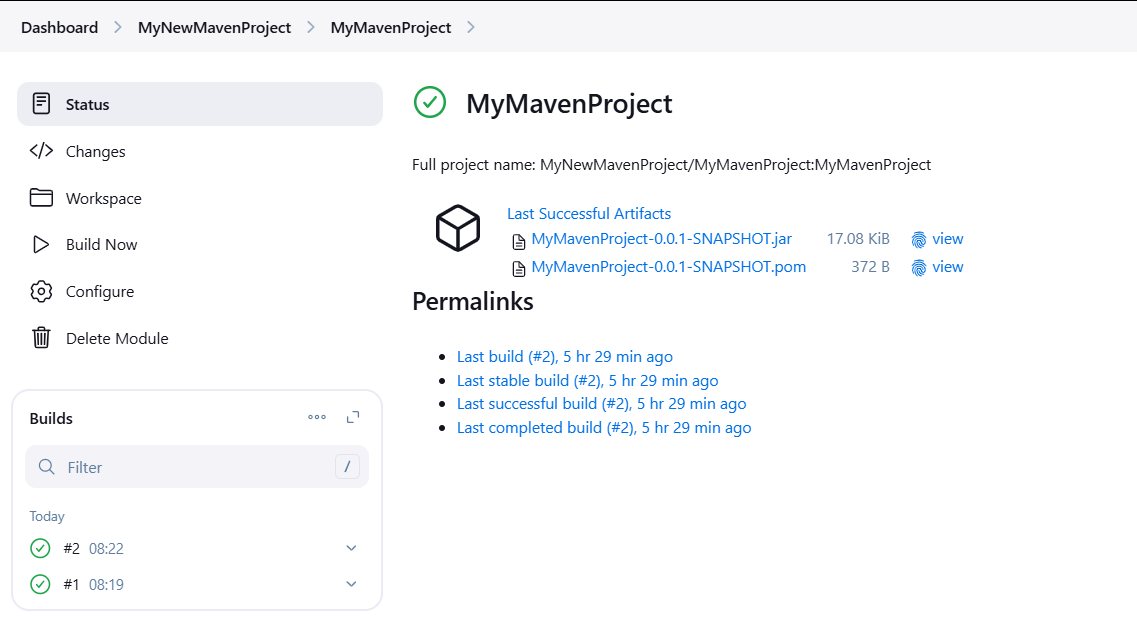
New item 🡪 Type Item name (here MyNewMavenProject) 🡪 select Maven Project 🡪 Ok

In configure Page –

Source Code Management : Type repo URl from git(code 🡪https -copy that URL & paste here) 🡪 change main branch name as in git repo (here main)

Build :

Root POM 🡪 copy the POM.xml file path of git



Then Save.

Status 🡪 Current Status

Workspace 🡪 Contents of Project file

Build now 🡪 to build

Configure 🡪 to change settings

# Read Excel File in Java

* In Java, reading excel file is not similar to read word file because of cells in excel file.
* JDK **does not provide direct API** to read or write Microsoft Excel or Word document. We have to rely on the **third-party library** that is **Apache POI**.
* **Apache POI (Poor Obfuscation Implementation)** is a Java API for **reading and writing** Microsoft Documents in both formats **.xls** and **.xlsx**. It contains **classes and interfaces**. The Apache POI library provides **two implementations for reading** excel files:

1. **HSSF (Horrible SpreadSheet Format)** Implementation: It denotes an API that is working with Excel 2003 or earlier versions.
2. **XSSF (XML SpreadSheet Format)** Implementation: It denotes an API that is working with Excel 2007 or later versions.

## Interfaces and Classes in Apache POI

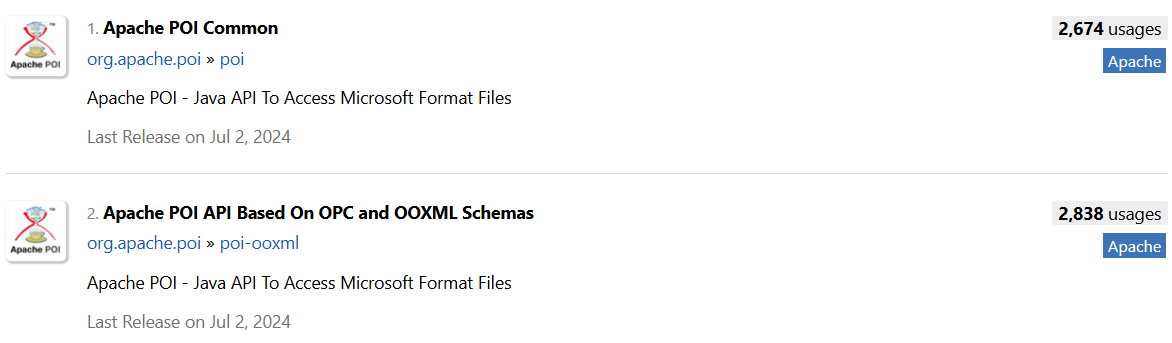
* Interfaces

1. **Workbook:** It represents an **Excel Workbook**. It is an interface implement by **HSSFWorkbook** and **XSSFWorkbook**.
2. **Sheet:** It is an interface that represents an **Excel worksheet**. A sheet is a **central structure of a workbook**, which represents a **grid of cells**. The Sheet interface extends **java.lang.Iterable**.
3. **Row:** It is also an interface that represents the **row** of the spreadsheet. The Row interface extends **java.lang.Iterable**. There are **two concrete** classes: **HSSFRow** and **XSSFRow**.
4. **Cell:** It is an interface. It is a high-level representation of a **cell** in a row of the spreadsheet. **HSSFCell** and **XSSFCell** implement Cell interface.

* Classes

1. **XLS Classes**
   * + 1. **HSSFWorkbook:** It is a class representing the **XLS file**.
       2. **HSSFSheet:** It is a class representing the **sheet** in an XLS file.
       3. **HSSFRow:** It is a class representing a **row** in the sheet of XLS file.
       4. **HSSFCell:** It is a class representing a **cell in a row** of XLS file.
     1. **XLSX Classes**
        1. **XSSFWorkbook:** It is a class representing the **XLSX file**.
        2. **XSSFSheet:** It is a class representing the **shee**t in an XLSX file.
        3. **XSSFRow:** It is a class representing a **row** in the sheet of XLSX file.
        4. **XSSFCell:** It is a class representing **a cell in a row** of XLSX file.

## To read data from XLS file

* To read we need to add apache poi & poi-ooxml dependency (Or JXL)
* Go to maven central repository : <https://mvnrepository.com/> Then select apache poi select 3.7 then copy the pom.xml file code and paste it in the pom.xml file in local maven project file.Same for apache

Poi based on ooxml.

* Add these 2 dependencies code inside <dependencies>

</ dependencies >

