Prerequisite:

- **High level language**
- **Low level language**
- **Different type of High-level language (procedural, functional, OOPS)**

JAVA ☕

Ok here we learn about the JAVA programming **but why are you used coffee symbol 🤔?**

History of JAVA:

1. **Green Team and the Birth of Java**:
    - In **June 1991**, a small team of Sun engineers, known as the **Green Team**, initiated the **Java language project**. Their goal was to create a language for digital devices like set-top boxes and televisions.
    - Initially, Java was designed for **embedded systems** in electronic appliances.
    - The project started with the name **"Greentalk"**, and the file extension was .gt.
    - Later, it was renamed to **"Oak"** as part of the **Green** project.
2. **Why "Oak"?**:
    - The name **"Oak"** symbolized strength and resilience, much like the oak tree, which is a national symbol in several countries.
    - However, in 1995, it was renamed to **"Java"** due to trademark conflicts with Oak Technologies.
3. **Choosing the Name "Java"**:
    - The team brainstormed various names, including "dynamic," "revolutionary," "Silk," "jolt," and "DNA."
    - They sought a name that reflected the essence of the technology: **revolutionary, dynamic, lively, cool, unique, and easy to spell**.
    - **James Gosling**, the father of Java, chose the name **"Java"** while sipping coffee near his office.

- Interestingly, **Java** is an island in Indonesia where the first coffee was produced (known as **Java coffee**).

**This is the reason why I'm used this coffee symbol

What is JAVA?

JAVA is a class based, heigh-level, object-oriented programming language developed by "JAMES GOSLING" and his friend in the year 1995.

NOTE:

- The first version of java (JDK 1.0) was released on the year JAN-23rd 1996 by "**SUN MICROSYSTEM**"
- Latest version of java (JDK 22) 2024 by "oracle"

But the question is we are already done C++ and but why we learn JAVA??

Java is one of the most popular programming languages in the world, and for good reason. It is a powerful, versatile, and secure language that can be used to develop a wide variety of applications.

** Features of JAVA

1. **Platform Independence (Write Once, Run Anywhere)**:
   - Java code is compiled into an **intermediate form called bytecode**. This bytecode can run on any platform with a **Java Virtual Machine (JVM)**.
   - JVM interprets the bytecode, making Java applications **platform-independent**.
2. **Object-Oriented Programming (OOP)**:
   - Java follows the principles of OOP, including **encapsulation, inheritance, and polymorphism**.
   - It allows developers to create modular, reusable, and maintainable code.
3. **Strongly Typed Language**:

    - Java enforces strict type checking during compilation.
    - This helps catch errors early and ensures type safety.
4. **Automatic Memory Management (Garbage Collection)**:
    - Java manages memory automatically through **garbage collection**.
    - Developers don't need to explicitly free memory; the JVM handles it.
5. **Rich Standard Library (Java API)**:
    - Java provides a comprehensive set of classes and methods in its **Standard Library**.
    - It covers areas like I/O, networking, data structures, and more.
6. **Exception Handling**:
    - Java has a robust exception handling mechanism.
    - Developers can catch and handle exceptions gracefully.
7. **Multithreading and Concurrency**:
    - Java supports multithreading, allowing developers to create **concurrent** applications.
    - The java.util.concurrent package provides tools for managing threads.
8. **Security and Safety**:
    - Java emphasizes security.
    - The JVM ensures that untrusted code doesn't harm the system.

First program in java

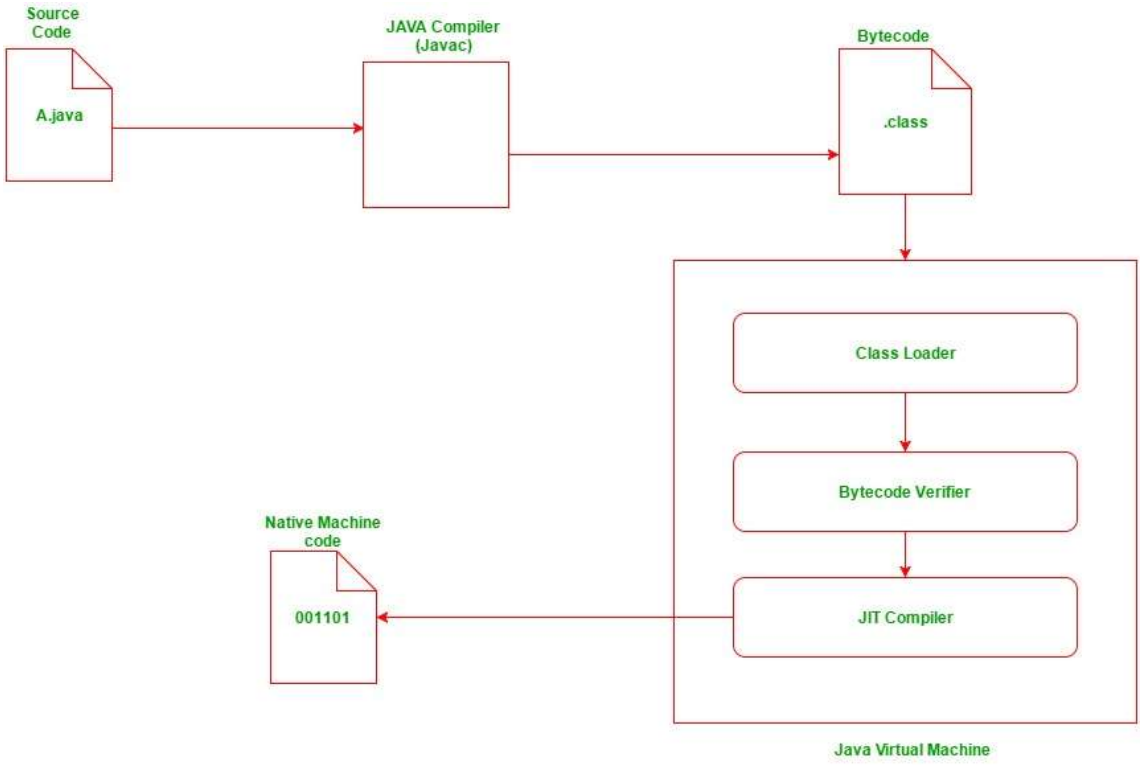
Now how to execute this line of code??

```
class Test
{
    public static void main(String []args)
    {
        System.out.println("My First Java Program.");
    }
}
```

Execution process:

1. Save class name (Test).java →save
2. javac Test.java →
3. java class name (Test)

Now, how to execute .java file internally 🤔 🤔?



Important Terms: JDK, JRE, JVM and JIT

1. **JDK (Java Development Kit)**:
    - The **JDK** is a **software development environment** used for creating and running Java applications.
    - It includes:
        - **Development Tools**: These tools provide an environment for writing and compiling Java programs.
        - **JRE (Java Runtime Environment)**: Necessary for executing Java programs.
    - Developers use the JDK to write, compile, and package Java code.
2. **JRE (Java Runtime Environment)**:
    - The **JRE** is an **installation package** that allows you to **run** Java programs on your machine.
    - It includes:
        - **Java Virtual Machine (JVM)**: Responsible for executing Java bytecode.
        - **Core classes** required for running Java applications.
        - **Supporting files** needed for proper execution.
    - End-users who only want to run Java programs use the JRE.

3. **JVM (Java Virtual Machine)**:
    - The **JVM** is a crucial part of both the JDK and JRE.
    - It serves as an **interpreter** for Java programs.
    - Key points about JVM:
        - It executes Java programs **line by line**.
        - JVM is responsible for converting Java bytecode into machine-specific instructions.
        - Whenever you run a Java program, an instance of JVM is created.
4. **JIT (Just-In-Time Compiler)**:
    - The **JIT** is a dynamic component within the JVM.
    - It **optimizes** the execution of Java programs.
    - How it works:
        - When a Java program starts executing, the JIT compiles parts of the bytecode into native machine code.
        - This compilation happens **on the fly**, improving performance during runtime.

Hope you understand how JVM works Internally

```
class Test
{
    public static void main(String []args)
    {
        System.out.println("My First Java Program.");
    }
}
```

But above we write a JAVA code, right?

Here you use class Test or something code What is the meaning about this line of code??

Now we learn rules or syntax to write JAVA code ☕

1. **Class Definition:**

```
class Test {
```

```
        //statement
    }
```

This line uses the keyword **class** to declare that a new class is being defined.

And contain **Test** It is an identifier that is the name of the class and define definition of class with the help of **{curly brace}**

2. **main Method**

```
public static void main(String []args)
```

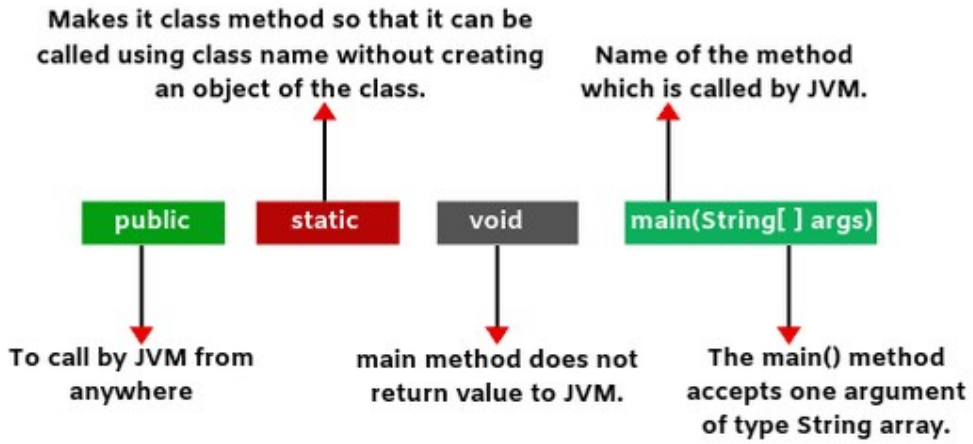


Fig: Java main method

3. **System.out.println("My First Java Program.");**

   This line outputs the string "**`My First Java Program.`**" followed by a new line on the screen. Output is accomplished by the built-in println( ) method. The **System** is a predefined class that provides access to the system and **out** is the variable of type output stream connected to the console.

Hope you understand how to write java code

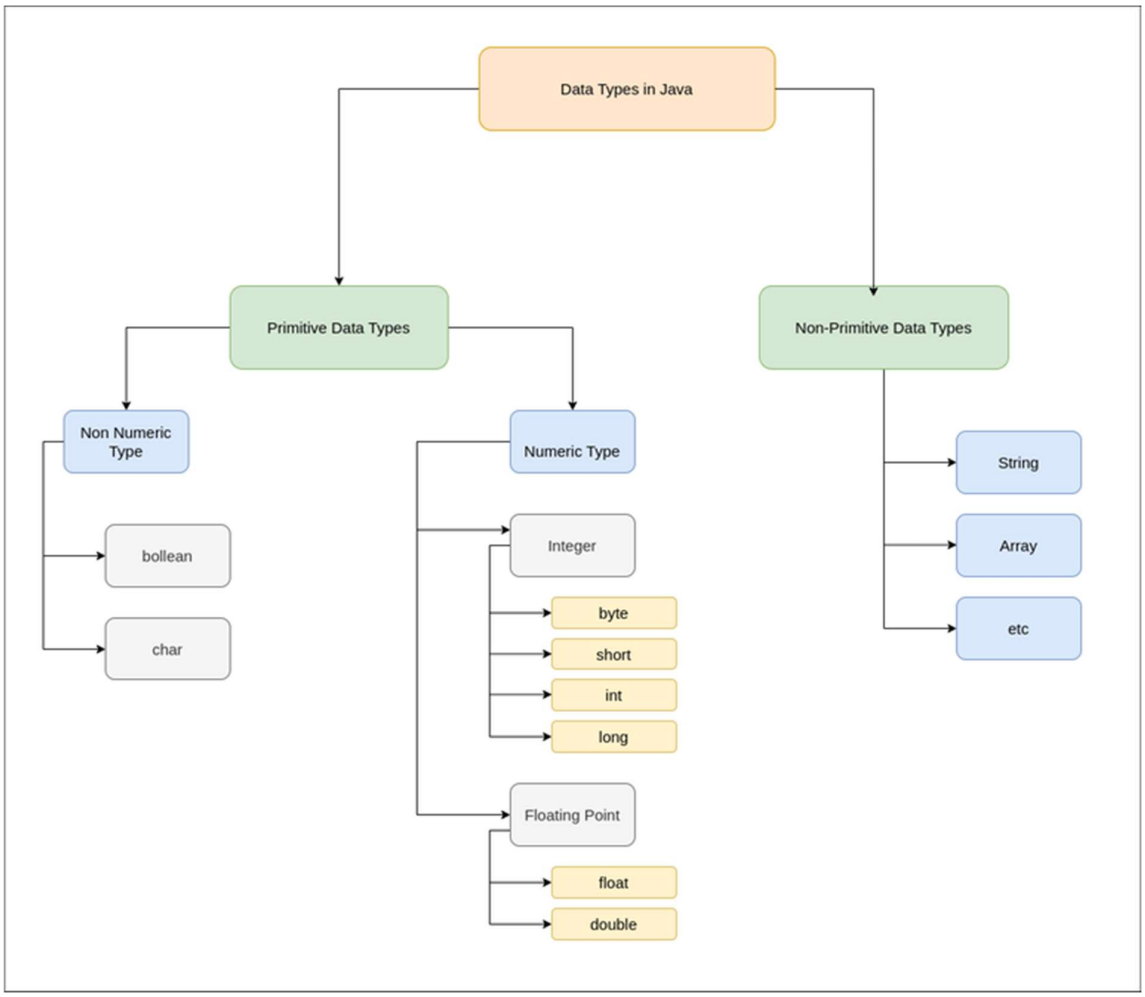
JAVA Data Types

Java is statically typed and also a strongly typed language because, in Java, each type of data (such as integer, character, hexadecimal, packed decimal, and so forth) is predefined as part of the programming language and all constants or variables defined for a given program must be described with one of the Java data types.

What is DATA type?

Datatype specify the different sizes and values that can be stored in the variable. Java has two categories in which data types are segregated.

1. **Primitive Data Types:**
    - These are the fundamental building blocks in Java. They include:
        - byte: An 8-bit integer that stores whole numbers from -128 to 127.
        - short: A 16-bit integer for whole numbers ranging from -32,768 to 32,767.
        - int: A 32-bit integer capable of storing whole numbers from -2,147,483,648 to 2,147,483,647.
        - long: An 64-bit integer that accommodates whole numbers from -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807.
        - float: A 32-bit floating point type for fractional numbers (approximately 6 to 7 decimal digits).
        - double: A 64-bit floating point type for more precise fractional numbers (approximately 15 decimal digits).
        - char: A 16-bit Unicode character representing a single letter or symbol.
        - boolean: A 1-bit type that stores true or false values.
2. **Non-Primitive Data Types:**
    - **These are more complex and include:**
        - **String: Represents sequences of characters (text).**
        - **Arrays: Collections of elements of the same type.**
        - **Classes: User-defined data types that encapsulate data and methods.**
        - **Interfaces: Blueprint for classes to implement.**



What is type casting in java?

➔ Converting one datatype to another datatype is called type casting.

Two type of type casting in java

1. **Implicit type casting**
2. **Explicit type casting**

Implicit type casting:

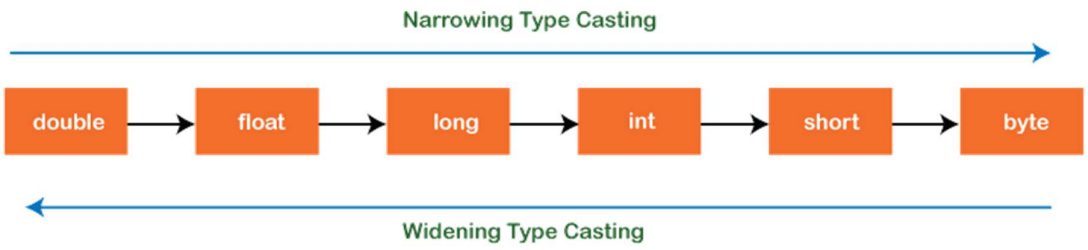
it is automatically performed by the compiler

Explicit type casting:

By default, the compiler doesn't allow the explicit type casting.

For example: double x=10.5;

int y=(int)x;



Type Casting in Java

variable

variable is the name of the memory location.

In other word we can say it is used define name which is given by user. Variable can store on type of value.

Ex:- int a=10

Here a is the variable.

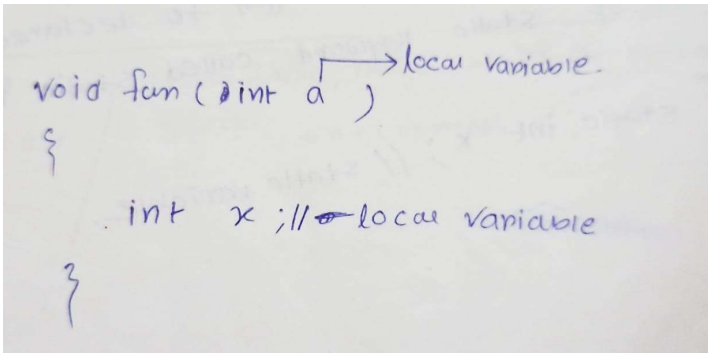
Type:

Three types of variables are in java there are

Local variable:

A variable which is declared inside the body of the method or method parameter call local variable.

Syntax:



Instance variable:

A variable which is declared inside the class but outside of all the methods call instance variable.

- As instance variables are declared in a class, these variables are created when an **object of the class is created** and destroyed when the object is destroyed.
- Unlike local variables, we may use access specifiers for instance variables. If we do not specify any access specifier, then the default access specifier will be used.
- Initialization of an instance variable is not mandatory. Its default value 0.
- Instance variables can be accessed only by creating objects.

Static variable:

A variable written to declared with the help of static keyword call static variable.

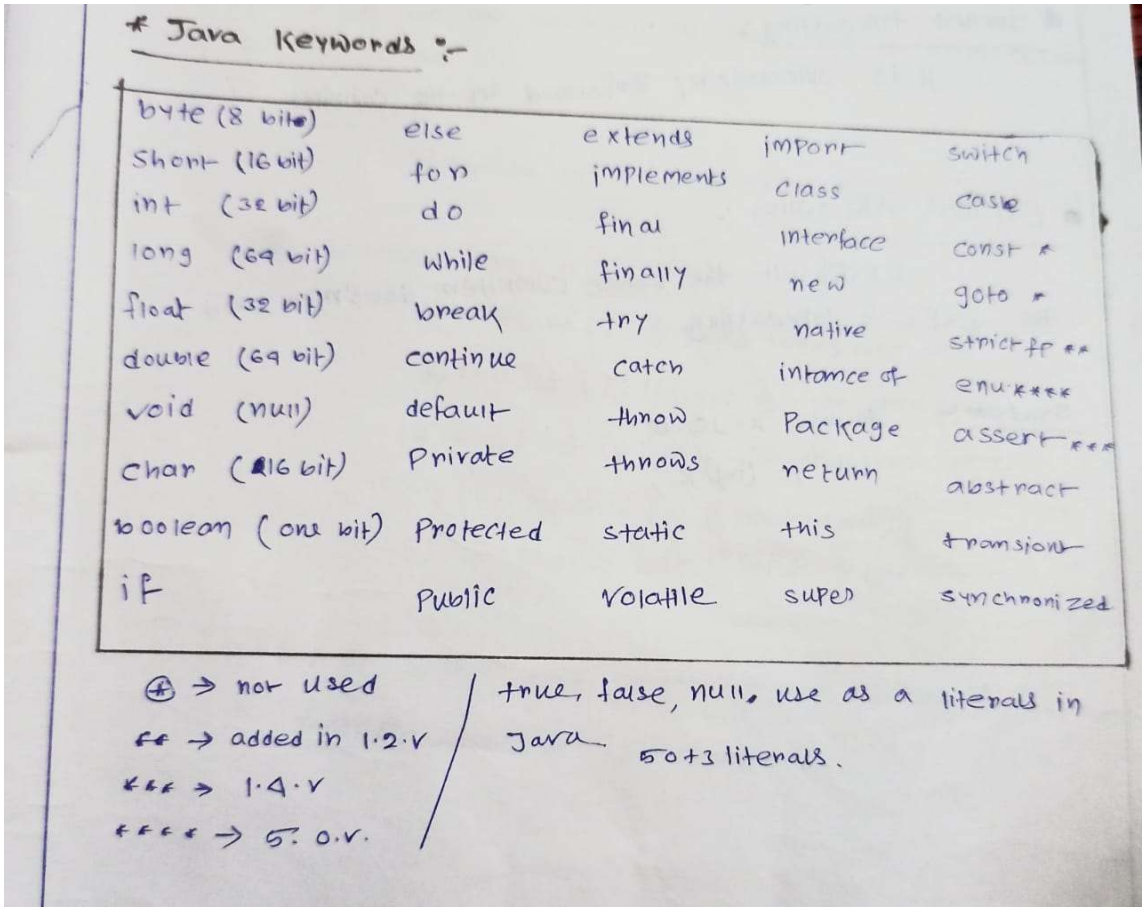
Syntax: static int x;

Keyword

Keyword are the reserved word whose meaning is already define in the java compiler.

Note:

- We can't use keyword for our personal use.
- Keyword are the case-sensitive.



* Java Keywords :-

byte (8 bits)	else	extends	import	switch
Short (16 bit)	for	implements	class	case
int (32 bit)	do	final	interface	const *
long (64 bit)	while	finally	new	goto *
float (32 bit)	break	try	native	strictfp **
double (64 bit)	continue	catch	instance of	enu****
void (null)	default	throw	Package	assert***
char (16 bit)	Private	throws	return	abstract
boolean (one bit)	Protected	static	this	transient
if	Public	volatile	super	synchronized

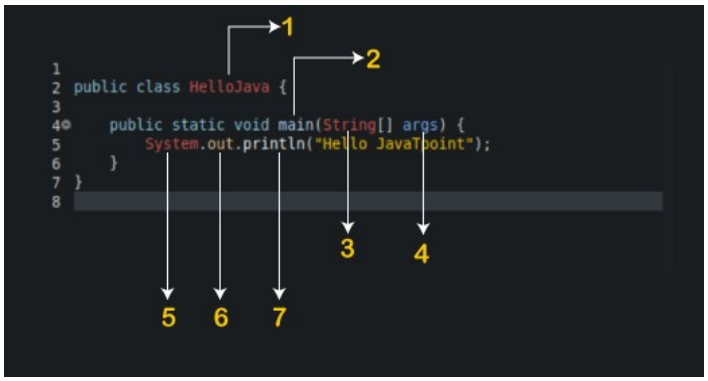
~~if~~ → not used
~~ff~~ → added in 1.2.v
~~***~~ → 1.4.v
~~*****~~ → 5.0.v.

true, false, null, use as a literals in Java — 50+3 literals.

Identifier

In java, an identifier is the name of the variable, method, class, package or interface that is used for the purpose of identification.

--- In java an identifier can be a class name, method name, variable name or label.



Here

HelloJava → class name

main →method name

String [] args →array

System.out.println()→ object/variable/method

Rules For Defining Java Identifiers

- The only allowed characters for identifiers are all alphanumeric characters**([A-Z],[a-z],[0-9]), '$'(dollar sign) and '_' (underscore).**For example "geek@" is not a valid Java identifier as it contains a '@' a special character.
- Identifiers should **not** start with digits(**[0-9]**). For example "123geeks" is not a valid Java identifier.
- Java identifiers are **case-sensitive**.
- There is no limit on the length of the identifier but it is advisable to use an optimum length of 4 – 15 letters only.

- **Reserved Words** can't be used as an identifier. For example, "int while = 20;" is an invalid statement as a while is a reserved word. There are **53** reserved words in Java.

Token

Token is the smallest element of a program that is identify by a compiler.

- Every java statement and expression are created using token.

List of Token:

1. Keyword
2. Identifier
3. Operator
4. Separator
5. Literals

JAVA operators

Operator Type	Category	Precedence	Associativity
Unary	**postfix**	**a++, a--**	**Right to left**
	prefix	**++a, --a, +a, -a, ~, !**	**Right to left**
Arithmetic	**Multiplication**	***, /, %**	**Left to Right**
	Addition	**+, -**	**Left to Right**
Shift	**Shift**	**<<, >>, >>>**	**Left to Right**
Relational	**Comparison**	**<, >, <=, >=, instanceOf**	**Left to Right**
	equality	**==, !=**	**Left to Right**
Bitwise	**Bitwise AND**	**&**	**Left to Right**
	Bitwise exclusive OR	**^**	**Left to Right**
	Bitwise inclusive OR	**|**	**Left to Right**
Logical	**Logical AND**	**&&**	**Left to Right**
	Logical OR	**||**	**Left to Right**
Ternary	**Ternary**	**? :**	**Right to Left**
Assignment	**assignment**	**=, +=, -=, *=, /=, %=, &=, ^=, |=, <<=, >>=, >>>=**	**Right to Left**

Input & output in JAVA

How to Take Input From User in Java?

Java brings various Streams with its I/O package that helps the user perform all the Java input-output operations. These streams support all types of objects, data types, characters, files, etc. to fully execute the I/O operations. Input in Java can be with certain methods mentioned below in the article.

Methods to Take Input in Java

There are **two ways** by which we can take Java input from the user or from a file

- BufferedReader Class
- Scanner Class

1. Using BufferedReader Class for String Input In Java

It is a simple class that is used to read a sequence of characters. It has a simple function that reads a character another read which reads, an array of characters, and a readLine() function which reads a line.

InputStreamReader() is a function that converts the input stream of bytes into a stream of characters so that it can be read as BufferedReader expects a stream of characters. BufferedReader can throw checked Exceptions.

2. Using Scanner Class for Taking Input in Java

It is an advanced version of BufferedReader which was added in later versions of Java. The scanner can read formatted input. It has different functions for different types of data types.

- The scanner is much easier to read as we don't have to write throws as there is no exception thrown by it.
- It was added in later versions of Java
- It contains predefined functions to read an Integer, Character, and other data types as well.

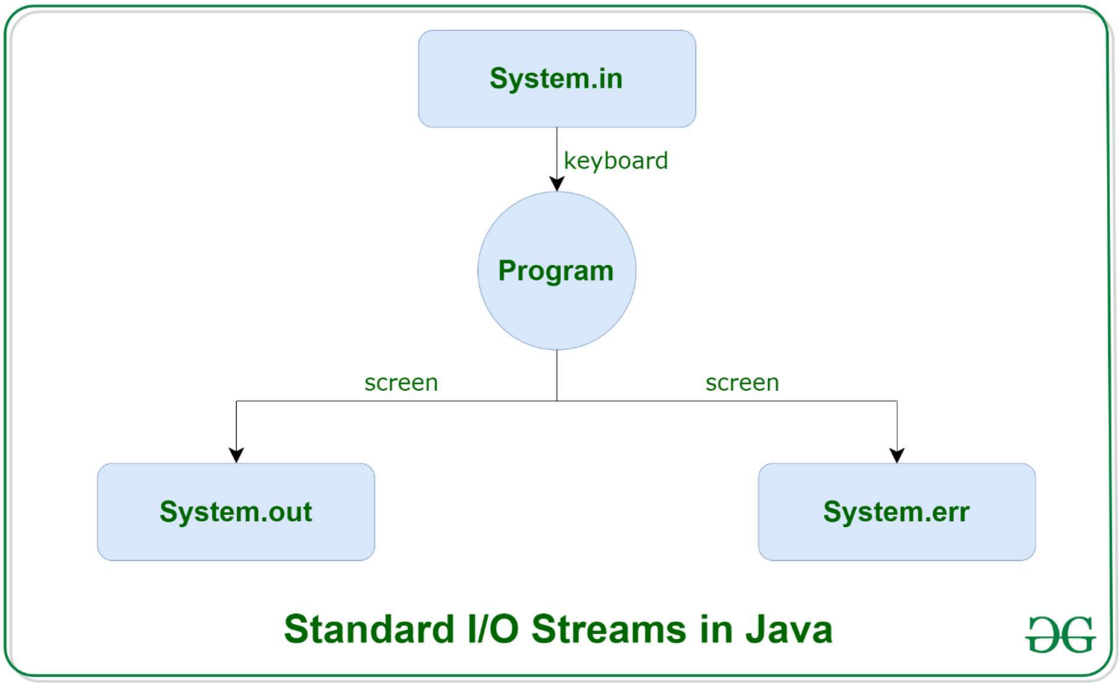
```
Scanner sc = new Scanner(System.in);
```

Now back to the deep knowledge about of I/O

This is GFG article.

Java brings various Streams with its I/O package that helps the user to perform all the input-output operations. These streams support all the types of objects, data-types, characters, files etc to fully execute the I/O operations.

Before exploring various input and output streams lets look at **3 standard or default streams** that Java has to provide which are also most common in use:



1. System.in**:** This is the **standard input stream** that is used to read characters from the keyboard or any other standard input device.

2. System.out: This is the **standard output stream** that is used to produce the result of a program on an output device like the computer screen.
   Here is a list of the various print functions that we use to output statements:

   - print(): This method in Java is used to display a text on the console. This text is passed as the parameter to this method in the form of String. This method prints the text on the console and the cursor remains at the end of the text at the console. The next printing takes place from just here.
     Syntax:
     ```
     System.out.print(parameter);
     ```
     Example:
     ```
     // Java code to illustrate print()
     import java.io.*;

     class Demo_print {
         public static void main(String[] args)
         {

             // using print()
             // all are printed in the
             // same line
             System.out.print("GfG! ");
             System.out.print("GfG! ");
             System.out.print("GfG! ");
         }
     }
     ```
     Output:
     ```
     GfG! GfG! GfG!
     ```
   - println(): This method in Java is also used to display a text on the console. It prints the text on the console and the cursor moves to the start of the next line at the console. The next printing takes place from the next line.
     Syntax:
     ```
     System.out.println(parameter);
     ```
     Example:

```java
// Java code to illustrate println()

import java.io.*;

class Demo_print {
    public static void main(String[] args)
    {

        // using println()
        // all are printed in the
        // different line
        System.out.println("GfG! ");
        System.out.println("GfG! ");
        System.out.println("GfG! ");
    }
}
```

Output:

```
GfG!
GfG!
GfG!
```

- printf(): This is the easiest of all methods as this is similar to printf in C. Note that System.out.print() and System.out.println() take a single argument, but printf() may take multiple arguments. This is used to format the output in Java.

3. **System.err:** This is the **standard error stream** that is used to output all the error data that a program might throw, on a computer screen or any standard output device.

   This stream also uses all the 3 above-mentioned functions to output the error data:

   - print()
   - println()
   - printf()

Flow Control in JAVA

A programming language used to control the statements to control the flow of execution of a program based on certain conditions. These are use to cause the flow of execution to advanced and branch based on change to the state of a program.

JAVA selection statement:

- **if**
- **nested if**
- **if else**
- **if-else-if**
- **switch**
- **JUMP (break, continue, return)**

1. **if:** if statement is the simplest decision-making statement. It is used to decide whether a certain statement or block of statements will be executed or not i.e if a certain condition is true then a block of statements is executed otherwise not.

```
if(condition)
{
   // Statements to execute if
   // condition is true
}
```

Here, the **condition** after evaluation will be either true or false. if statement accepts Boolean values – if the value is true then it will execute the block of statements under it.
If we do not provide the curly braces '{'and '}' after **if (condition)** then by default if statement will consider the immediate one statement to be inside its block. For example,

```
if(condition) //Assume condition is true
   statement1; //part of if block
   statement2; // separate from if block


// Here if the condition is true
// if block will consider statement1 as its part and executes in only true condition
// statement2 will be separate from the if block so it will always executes whether the condition is true or false.
```

2. **if-else**: The if statement alone tells us that if a condition is true it will execute a block of statements and if the condition is false it won't. But what if we want to do something else if the condition is false? Here comes the else statement. We can use the else statement with the if statement to execute a block of code when the condition is false.

Syntax:

```
if (condition)
{
    // Executes this block if
    // condition is true
}
else
{
    // Executes this block if
    // condition is false
}
```

3. **nested-if:** A nested if is an if statement that is the target of another if or else. Nested if statements mean an if statement inside an if statement. Yes, java allows us to nest if statements within if statements. i.e, we can place an if statement inside another if statement.

Syntax:

```
if (condition1)
{
   // Executes when condition1 is true
   if (condition2)
   {
      // Executes when condition2 is true
   }
}
```

4. **if-else-if ladder:** Here, a user can decide among multiple options. The if statements are executed from the top down. As soon as one of the conditions controlling the if is true, the statement associated with that 'if' is executed, and the rest of the ladder is bypassed. If none of the conditions is true, then the final else statement will be executed. There can be as many as 'else if' blocks associated with one 'if' block but only one 'else' block is allowed with one 'if' block.

```
if (condition)
    statement;
else if (condition)
    statement;
.
.
else
    statement;
```

5. **switch-case:** The switch statement is a multiway branch statement. It provides an easy way to dispatch execution to different parts of code based on the value of the expression.

Syntax:

```
switch (expression)
{
  case value1:
    statement1;
```

```
    break;
  case value2:
    statement2;
    break;
  .
  .
  case valueN:
    statementN;
    break;
  default:
    statementDefault;
}
```

JUMP

jump: Java supports three jump statements: **break, continue** and **return**. These three statements transfer control to another part of the program.

1. **Break:** In Java, a break is majorly used for:
    - Terminate a sequence in a switch statement (discussed above).
    - To exit a loop.
    - Used as a “civilized” form of goto.

2. **Continue:** Sometimes it is useful to force an early iteration of a loop. That is, you might want to continue running the loop but stop processing the remainder of the code in its body for this particular iteration. This is, in effect, a goto just past the body of the loop, to the loop’s end. The continue statement performs such an action.

3. **Return:** The return statement is used to explicitly return from a method. That is, it causes program control to transfer back to the caller of the method.

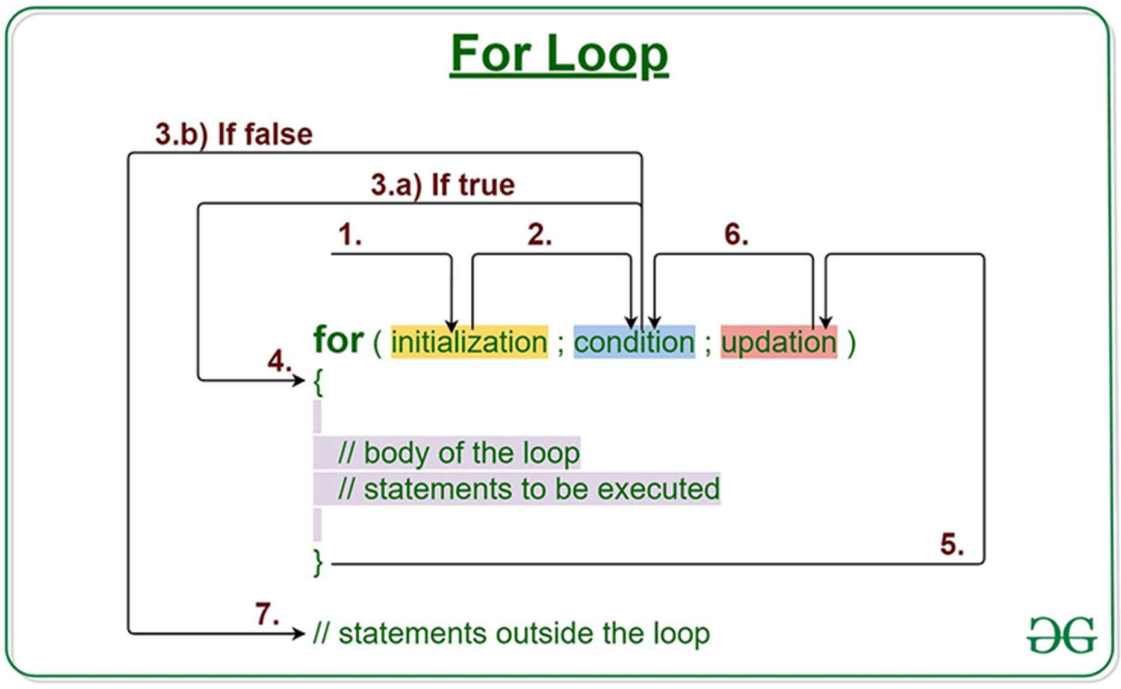
Looping statement

Looping in programming languages is a feature. Whenever we have to repeat certain statement, several times is called loop.

Types:

1. For loop
2. While loop
3. Do-while
4. For-each loop

For loop



Syntax:

```
for (initialization expr; test expr; update exp)
{
     // body of the loop
     // statements we want to execute
}
```

Parts of Java For Loop

Java for loop is divided into various parts as mentioned below:

- Initialization Expression
- Test Expression
- Update Expression
-

1. Initialization Expression

In this expression, we have to initialize the loop counter to some value.

Example:

```
int i=1;
```

2.Test Expression

In this expression, we have to test the condition. If the condition evaluates to true then, we will execute the body of the loop and go to the update expression. Otherwise, we will exit from the for a loop.

Example:

```
i <= 10
```

3.Update Expression:

After executing the loop body, this expression increments/decrements the loop variable by some value.

Example:

```
i++;
```

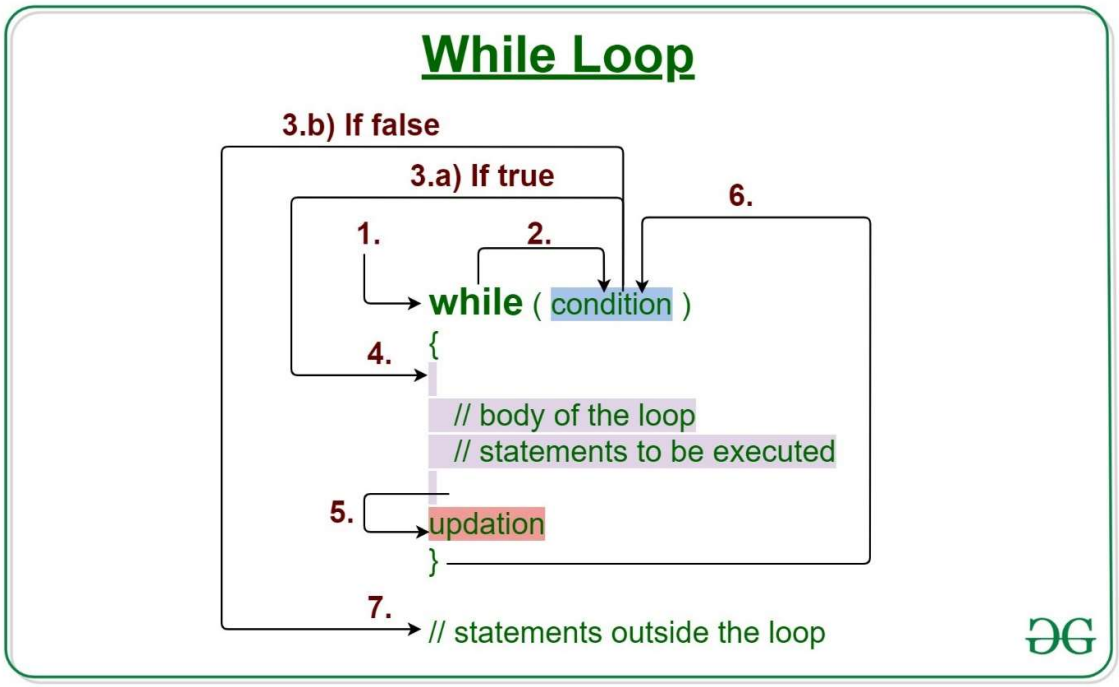
How does a For loop work?

1. Control falls into the for loop. Initialization is done
2. The flow jumps to Condition
3. Condition is tested.
    - If the Condition yields true, the flow goes into the Body
    - If the Condition yields false, the flow goes outside the loop
4. The statements inside the body of the loop get executed.
5. The flow goes to the Updation
6. Updation takes place and the flow goes to Step 3 again
7. The for loop has ended and the flow has gone outside.

```java
class Test {
    public static void main(String[] args)
    {
        for (int i = 1; i <= 10; i++) {
            System.out.println(i);
        }
    }
}
```

while loop

Java while loop is a control flow statement that allows code to be executed repeatedly based on a given Boolean condition. The while loop can be thought of as a repeating if statement. While loop in Java comes into use when we need to repeatedly execute a block of statements. The while loop is considered as a repeating if statement. If the number of iterations is not fixed, it is recommended to use the while loop.



Syntax:

```
while (test_expression)
{
   // statements

   update_expression;
}
```

Parts of Java While Loop

The various parts of the While loop **are:**
1. Test Expression: In this expression, we have to test the condition. If the condition evaluates to true then we will execute the body of the loop and go to update expression. Otherwise, we will exit from the while loop.

Example:

```
i <= 10
```

2. Update Expression: After executing the loop body, this expression increments/decrements the loop variable by some value.

Example:

```
i++;
```

How Does a While loop execute?

1. Control falls into the while loop.
2. The flow jumps to Condition
3. Condition is tested.
    - If Condition yields true, the flow goes into the Body.
    - If Condition yields false, the flow goes outside the loop
4. The statements inside the body of the loop get executed.
5. Updation takes place.
6. Control flows back to Step 2.
7. The while loop has ended and the flow has gone outside.

// Java program to illustrate while loop.

```
class whileLoopDemo {
	public static void main(String args[])
	{
		// initialization expression
		int i = 1;
		// test expression
		while (i < 6) {
			System.out.println("Hello World");
			// update expression
			i++;
		}
	}
}
```

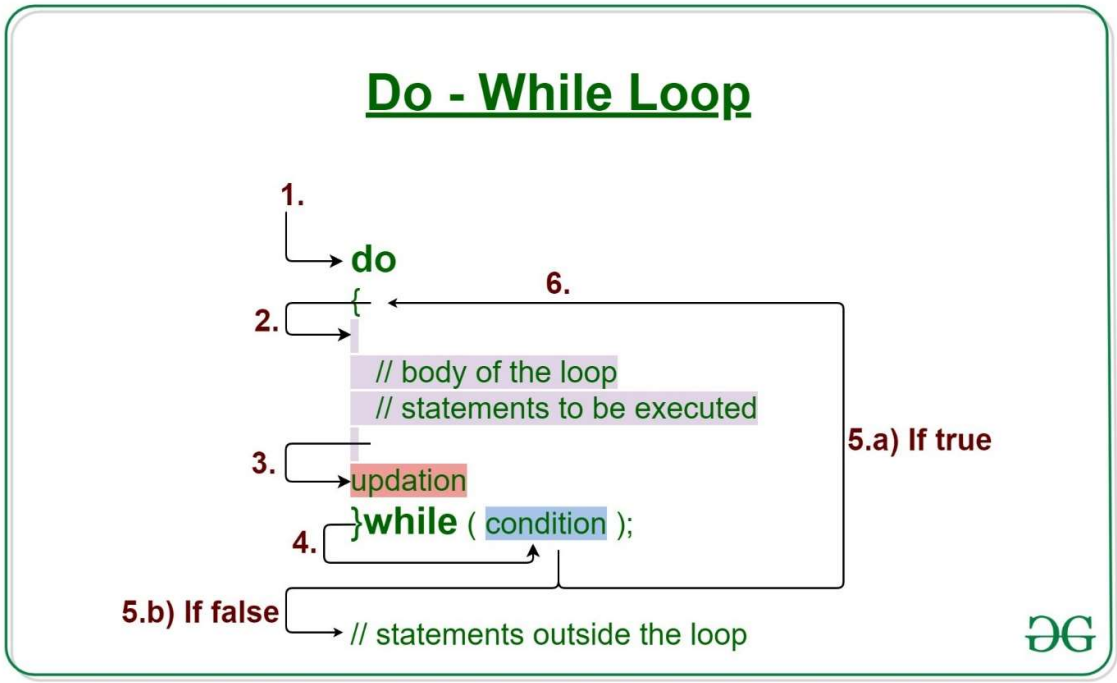
do-while loop with Examples

Loops in Java come into use when we need to repeatedly execute a block of statements. Java **do-while loop** is an **Exit control loop**. Therefore, unlike ***for*** or ***while*** loop, a do-while check for the condition after executing the statements of the loop body.

Syntax:

```
do
{
    // Loop Body
    Update_expression
}
```

```
// Condition check
while (test_expression);
```



Components of do-while Loop

A. Test Expression: In this expression, we have to test the condition. If the condition evaluates to true then we will execute the body of the loop and go to update expression. Otherwise, we will exit from the while loop. For example:

```
i <= 10
```

B. Update Expression: After executing the loop body, this expression increments/decrements the loop variable by some value. For example:

```
i++;
```

Execution of do-While loop

1. Control falls into the do-while loop.
2. The statements inside the body of the loop get executed.
3. Updation takes place.
4. The flow jumps to Condition
5. Condition is tested.
    1. If Condition yields true, go to Step 6.
    2. If Condition yields false, the flow goes outside the loop

6. The flow goes back to Step 2.

For-each loop in Java

For-each is another array traversing technique like for loop, while loop, do-while loop introduced in Java5.

- It starts with the keyword **for** like a normal for-loop.
- Instead of declaring and initializing a loop counter variable, you declare a variable that is the same type as the base type of the array, followed by a colon, which is then followed by the array name.
- In the loop body, you can use the loop variable you created rather than using an indexed array element.

Hope you all understood about control flow

JAVA String

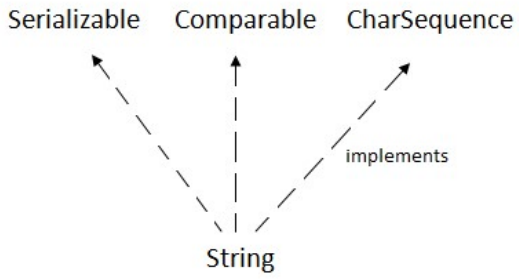
In Java, string is basically an object that represents sequence of char values. An array of characters works same as Java string. For example:

```
char[] ch={'j','a','v','a','t','p','o','i','n','t'};
String s=new String(ch);
```

is same as:

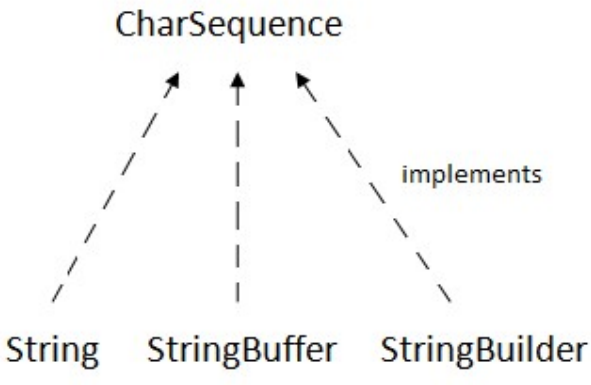
```
String s="javatpoint";
```

The java.lang.String class implements *Serializable*, *Comparable* and *CharSequence* interfaces.



CharSequence Interface

The CharSequence interface is used to represent the sequence of characters. String, String Buffer and StringBuilder classes implement it. It means, we can create strings in Java by using these three classes.



The Java String is immutable which means it cannot be changed. Whenever we change any string, a new instance is created. For mutable strings, you can use StringBuffer and StringBuilder classes.

We will discuss immutable string later. Let's first understand what String in Java is and how to create the String object.

What is String in Java?

Generally, String is a sequence of characters. But in Java, string is an object that represents a sequence of characters. The java.lang.String class is used to create a string object.

How to create a string object?

There are two ways to create String object:

1. By string literal
2. By new keyword

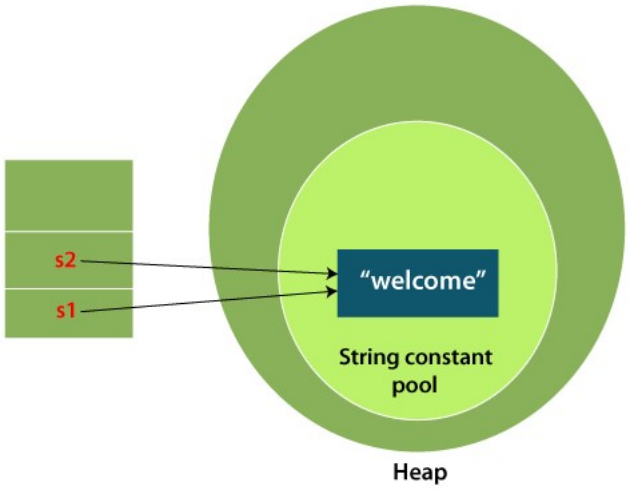
1) String Literal

Java String literal is created by using double quotes. For Example:

```
1. String s="welcome";
```

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. For example:

```
1. String s1="Welcome";
2. String s2="Welcome";//It doesn't create a new instance
```



In the above example, 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.

Why Java uses the concept of String literal?

To make Java more memory efficient (because no new objects are created if it exists already in the string constant pool).

2) By new keyword

1. String s=**new** String("Welcome");
   //creates two objects and one reference variable

In such case, JVM will create a new string object in normal (non-pool) heap memory, and the literal "Welcome" will be placed in the string constant pool. The variable s will refer to the object in a heap (non-pool).

Immutable String in Java

A String is an unavoidable type of variable while writing any application program. String references are used to store various attributes like username, password, etc. In Java, **String objects are immutable**. Immutable simply means unmodifiable or unchangeable.

Once String object is created its data or state can't be changed but a new String object is created.

Let's try to understand the concept of immutability by the example given below:

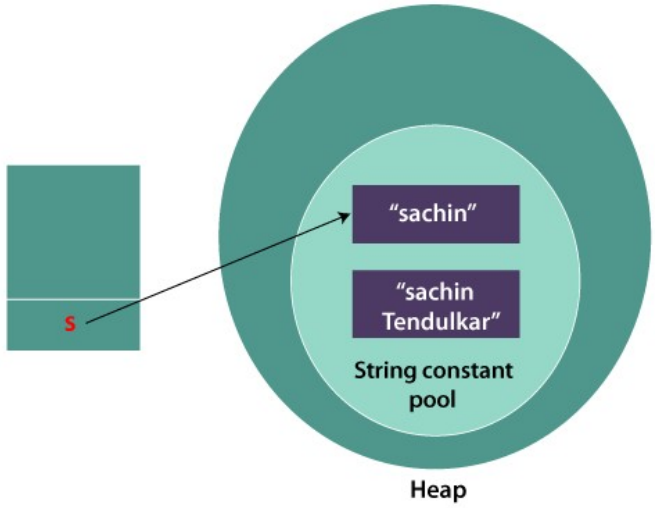
Testimmutablestring.java

```
class Testimmutablestring{
 public static void main(String args[]){
   String s="Sachin";
   s.concat(" Tendulkar");//concat() method appends the string at the end
   System.out.println(s);//will print Sachin because strings are immutable objects
 }
}
```

Output:

```
Sachin
```

Now it can be understood by the diagram given below. Here Sachin is not changed but a new object is created with Sachin Tendulkar. That is why String is known as immutable.



As you can see in the above figure that two objects are created but ***s*** reference variable still refers to "Sachin" not to "Sachin Tendulkar".

But if we explicitly assign it to the reference variable, it will refer to "Sachin Tendulkar" object.

For example:

Testimmutablestring1.java

```
class Testimmutablestring1{
public static void main(String args[]){
String s="Sachin";
s=s.concat(" Tendulkar");
System.out.println(s);
}
}
```

Output:

```
Sachin Tendulkar
```

In such a case, s points to the "Sachin Tendulkar". Please notice that still Sachin object is not modified.

String methods

Java String charAt()

The **Java String class charAt()** method returns *a char value at the given index number*.

The index number starts from 0 and goes to n-1, where n is the length of the string. It returns **StringIndexOutOfBoundsException,** if the given index number is greater than or equal to this string length or a negative number.

Syntax

```
public char charAt(int index)
```

```
public class CharAtExample{
public static void main(String args[]){
String name="javatsupper";
char ch=name.charAt(4);//returns the char value at the 4th index
System.out.println(ch);
}}
```

String compare

We can compare String in Java on the basis of content and reference.

It is used in **authentication** (by equals() method), **sorting** (by compareTo() method), **reference matching** (by == operator) etc.

There are three ways to compare String in Java:

1. By Using equals() Method
2. By Using == Operator
3. By compareTo() Method

equals() Method

The String class equals() method compares the original content of the string. It compares values of string for equality. String class provides the following two methods:

Using == operator

The == operator compares references not values.

compareTo() method

The String class compareTo() method compares values lexicographically and returns an integer value that describes if first string is less than, equal to or greater than second string.

Suppose s1 and s2 are two String objects. If:

- **s1 == s2** : The method returns 0.
- **s1 > s2** : The method returns a positive value.
- **s1 < s2** : The method returns a negative value.

String Concatenation in Java

String Concatenation by + (String concatenation) operator

Java String concatenation operator (+) is used to add strings. For Example:

TestStringConcatenation1.java

```
class TestStringConcatenation1{
 public static void main(String args[]){
   String s="Sachin"+" Tendulkar";
   System.out.println(s);//Sachin Tendulkar
 }
}
```

- ## Concatenation by concat() method

The String concat() method concatenates the specified string to the end of current string. Syntax:

1. **public** String concat(String another)

Substring in Java

A part of String is called **substring**. In other words, substring is a subset of another String. Java String class provides the built-in substring() method that extract a substring from the given string by using the index values passed as an argument. In case of substring() method startIndex is inclusive and endIndex is exclusive.

Suppose the string is "**computer**", then the substring will be com, compu, ter, etc.

String endsWith()

The **Java String class endsWith()** method checks if this string ends with a given suffix. It returns true if this string ends with the given suffix; else returns false.

Signature

The syntax or signature of endsWith() method is given below.

```
1. public boolean endsWith(String suffix)
```

getBytes() Method Example

The parameterless getBytes() method encodes the string using the default charset of the platform, which is UTF - 8. The following two examples show the same.

FileName: StringGetBytesExample.java

```
public class StringGetBytesExample{
public static void main(String args[]){
String s1="ABCDEFG";
byte[] barr=s1.getBytes();
for(int i=0;i<barr.length;i++){
System.out.println(barr[i]);
}
}}
```

Output:

```
65
66
67
68
69
70
71
```

String indexOf()

The **Java String class indexOf()** method returns the position of the first occurrence of the specified character or string in a specified string.

String isEmpty()

The **Java String class isEmpty()** method checks if the input string is empty or not. Note that here empty means the number of characters contained in a string is zero.

```
public class IsEmptyExample{
public static void main(String args[]){
String s1="";
String s2="shuvra";

System.out.println(s1.isEmpty());
System.out.println(s2.isEmpty());
}}
```

Output:

```
true
false
```

Java StringBuffer Class

Java StringBuffer class is used to create mutable (modifiable) String objects. The StringBuffer class in Java is the same as String class except it is mutable i.e. it can be changed.

Important methods of StringBuffer class

Modifier and Type	Method	Description
public synchronized StringBuffer	append(String s)	It is used to append the specified string with this string. The append() method is overloaded like append(char), append(boolean), append(int), append(float), append(double) etc.
public synchronized StringBuffer	insert(int offset, String s)	It is used to insert the specified string with this string at the specified position. The insert() method is overloaded like insert(int, char), insert(int, boolean), insert(int, int), insert(int, float), insert(int, double) etc.
public synchronized StringBuffer	replace(int startIndex, int endIndex, String str)	It is used to replace the string from specified startIndex and endIndex.
public synchronized StringBuffer	delete(int startIndex, int endIndex)	It is used to delete the string from specified startIndex and endIndex.
public synchronized StringBuffer	reverse()	is used to reverse the string.
public int	capacity()	It is used to return the current capacity.
public void	ensureCapacity(int minimumCapacity)	It is used to ensure the capacity at least equal to the given minimum.
public char	charAt(int index)	It is used to return the character at the specified position.

public int	length()	It is used to return the length of the string i.e. total number of characters.
public String	substring(int beginIndex)	It is used to return the substring from the specified beginIndex.
public String	substring(int beginIndex, int endIndex)	It is used to return the substring from the specified beginIndex and endIndex.

What is a mutable String?

A String that can be modified or changed is known as mutable String. StringBuffer and StringBuilder classes are used for creating mutable strings.

Java StringBuilder Class

Java StringBuilder class is used to create mutable (modifiable) String. The Java StringBuilder class is same as StringBuffer class except that it is non-synchronized. It is available since JDK 1.5.

Important methods of StringBuilder class

Method	Description
public StringBuilder append(String s)	It is used to append the specified string with this string. The append() method is overloaded like append(char), append(boolean), append(int), append(float), append(double) etc.
public StringBuilder insert(int offset, String s)	It is used to insert the specified string with this string at the specified position. The insert() method is overloaded like insert(int, char), insert(int, boolean), insert(int, int), insert(int, float), insert(int, double) etc.
public StringBuilder replace(int startIndex, int endIndex, String str)	It is used to replace the string from specified startIndex and endIndex.
public StringBuilder delete(int startIndex, int endIndex)	It is used to delete the string from specified startIndex and endIndex.
public StringBuilder reverse()	It is used to reverse the string.

public int capacity()	It is used to return the current capacity.
public void ensureCapacity(int minimumCapacity)	It is used to ensure the capacity at least equal to the given minimum.
public char charAt(int index)	It is used to return the character at the specified position.
public int length()	It is used to return the length of the string i.e. total number of characters.
public String substring(int beginIndex)	It is used to return the substring from the specified beginIndex.
public String substring(int beginIndex, int endIndex)	It is used to return the substring from the specified beginIndex and endIndex.

OOPs concept

What is OOPs?

Oops stands for **Object Oriented Programming Language,** the main purpose of oops is to deal with the real-world entity using programming language.

Oops features:

- **class**
- **object**
- **Inheritance**
- **Polymorphism**
- **Encapsulation**
- **Abstraction**

Now it’s time to jump real world

Class And Object

Class Definition:

- A class defines the **state** (attributes or fields) and **behaviour** (methods) of an object.
- For instance, consider a **Bicycle** class. It might have a field like gear (representing the gear number) and a method like braking () (to handle braking).

- Example class definition:

Java

```
class Bicycle {
   // State (field)
   private int gear = 5; // Default gear value

   // Behavior (method)
   public void braking () {
      System.out.println("Working of Braking");
   }
}
```

Creating Objects (Instances):

- An **object** is an instance of a class. For example, if Bicycle is the class, then MountainBicycle, SportsBicycle, and TouringBicycle can be considered objects of that class.
- To create an object, use the new keyword along with the class constructor (which has the same name as the class).
- Example object creation:

Java

```
Bicycle sportsBicycle = new Bicycle();
Bicycle touringBicycle = new Bicycle ();
```

Accessing Members of a Class:

- You can use the object's name (e.g., sportsBicycle) along with the dot (.) operator to access fields and methods of the class.
- For instance, you can set the gear value or call the braking () method:

Java

```
sportsBicycle.gear = 3; // Set gear to 3
sportsBicycle.braking(); // Call the braking method
```

Constructor

Definition:

Constructor is a special type of method whose name is same as class name.

Note:

1. The main purpose of constructor is to initialized the object.
2. Every java class has a constructor
3. A constructor is an automatically called at the time 0f object creation.
4. A constructor never contains any return type (including void).

Code

```java
// Java Program to demonstrate
// Constructor
// Driver Class
class Cons {

    // Constructor
    Cons ()
    {
        System.out.println("Constructor Called");
    }

    // main function
    public static void main(String[] args)
    {
        Cons geek = new Cons ();
    }
}
```

Types of constructors

1. **Default constructor**
2. **Parameterized constructor**
3. **Copy constructor**

Default constructor

Definition:

A constructor which does not have any parameter is called default constructor.

Syntax:

```
Class A {

  A () //default constructor

}
```

Parameterized constructor

A constructor through which we can pass one or more parameter is called parameterized constructor.

Syntax:

```
class A {
    A (String b) {
    }
}
```

Code

```java
// Java Program for Parameterized Constructor
class Geek {
    // data members of the class.
    String name;
    int id;
    Geek (String name, int id)
    {
        this.name = name;
        this.id = id;
    }
}
class GFG {
    public static void main (String [] args)
    {
        // This would invoke the parameterized constructor.
        Geek geek1 = new Geek ("Avinash", 68);
        System.out.println("GeekName:" + geek1.name
                           + " and GeekId:" + geek1.id);
    }
}
```

Copy constructor

Definition:

Whenever we pass object reference to the constructor then it's called copy constructor.

Syntax:

class A {

A (A ref) {

}

}

Code

```
// Java Program for Copy Constructor
class Geek {
    // data members of the class.
    String name;
    int id;

    // Copy Constructor
    Geek (Geek obj2)
    {
        this.name = obj2.name;
        this.id = obj2.id;
    }
}
class GFG {
    public static void main (String [] args)
    {
        // This would invoke the copy constructor.
        Geek geek2 = new Geek(geek1);
        System.out.println(
            "Copy Constructor used Second Object");
        System.out.println("GeekName :" + geek2.name
                           + " and GeekId :" + geek2.id);
    }
}
```

Constructor Overloading

Whenever we have more than constructor in our class then it's called constructor overloading.

Code:

```
public class ConstructorEx {
   int a; // a=10 //using this keyword
   int b; // b=20 //using this keyword
   ConstructorEx() {   /*  constructor */  /* default constructor */
      System.out.println("Hello i'm constructor");
   }
   ConstructorEx(int c, int d){ /* parameterized constructor */
      System.out.println("Hello i'm parameterized constructor");
      this.a=c;
      this.b=d;
   }
   ConstructorEx(int a, int b, char c){  /* parameterized constructor */
      System.out.println("Hello i'm parameterized constructor");
   }
   ConstructorEx(int a, int b, char c, char d){  /* parameterized constructor */
      System.out.println("Hello i'm parameterized constructor");
   }
   public static void main(String[] args) {
      ConstructorEx constructorEx1 = new ConstructorEx(); // 1.
```

```
        ConstructorEx constructorEx2 = new ConstructorEx(10,20); //  2.
        System.out.println(constructorEx2.a); // 10
        System.out.println(constructorEx2.b); // 20
        ConstructorEx constructorEx3 = new ConstructorEx(10,20,'3'); // 3.
        ConstructorEx constructorEx4 = new ConstructorEx(10,20,'3','c'); // 4.
    }
}
```

Polymorphism

Polymorphism is the combination of two Greek word one is poly means "many" and another is morphism means "from".

Whose meaning is "same object having different behaviour".

Types of polymorphism

In java polymorphism is mainly divide into two types:

1. Compile time polymorphism / static polymorphism
2. Runtime polymorphism / Dynamic polymorphism

Compile time polymorphism:

A polymorphism which is achieved by compile time called compile time polymorphism/early binding.

----- **method overloading** is an example of compile time polymorphism.

Method overloading:

Whenever a class contain more then one method with same name and different parameter called method overloading.

Syntax:

Return_type method_name(p1)

Return_type method_name (p1, p2)

Runtime polymorphism:

A polymorphism which is exist at the time of execution of program is called runtime polymorphism.

Method Overriding:

whenever we writing method in super class and sub class in such a way that the method name parameter must be same called method overriding.

Inheritance

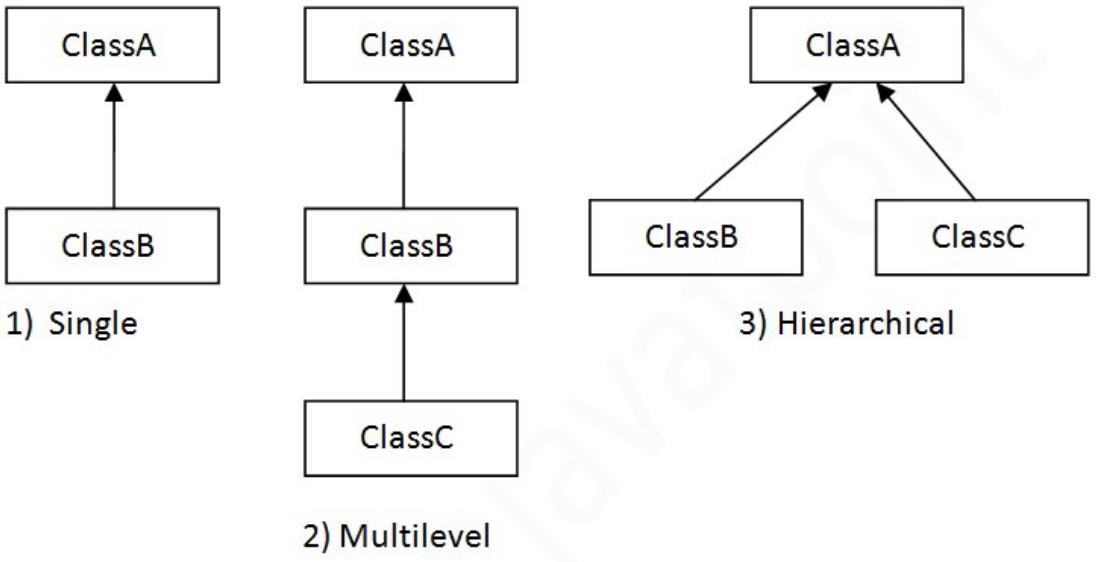
When we construct a new class from existing class in such a way that the new class access all the features & properties of existing class called inheritance.

Note:

1. In java extends keyword is used to perform inheritance.
2. It provides code reusability
3. We can't access private member of class through inheritance
4. A subclass contains all the features of super class so, we should create of sub class.
5. Method overriding only possible through inheritance.

Types of inheritance in java

On the basis of class, there can be three types of inheritance in java: single, multilevel and hierarchical.



Single Inheritance Example

When a class inherits another class, it is known as a *single inheritance*. In the example given below, Dog class inherits the Animal class, so there is the single inheritance.

File: TestInheritance.java

```
1. class Animal{
2. void eat(){System.out.println("eating...");}
3. }
4. class Dog extends Animal{
5. void bark(){System.out.println("barking...");}
6. }
7. class TestInheritance{
8. public static void main(String args[]){
9. Dog d=new Dog();
10. d.bark();
11. d.eat();
12. }}
```

Output:

```
barking...
eating...
```

Multilevel Inheritance Example

When there is a chain of inheritance, it is known as *multilevel inheritance*. As you can see in the example given below, BabyDog class inherits the Dog class which again inherits the Animal class, so there is a multilevel inheritance.

File: TestInheritance2.java

```
1. class Animal{
2. void eat(){System.out.println("eating...");}
3. }
4. class Dog extends Animal{
5. void bark(){System.out.println("barking...");}
6. }
7. class BabyDog extends Dog{
```

```
8.  void weep(){System.out.println("weeping...");}
9.  }
10. class TestInheritance2{
11. public static void main(String args[]){
12. BabyDog d=new BabyDog();
13. d.weep();
14. d.bark();
15. d.eat();
16. }}
```

Output:

```
weeping...
barking...
eating...
```

Hierarchical Inheritance Example

When two or more classes inherits a single class, it is known as *hierarchical inheritance*. In the example given below, Dog and Cat classes inherits the Animal class, so there is hierarchical inheritance.

File: TestInheritance3.java

```
1.  class Animal{
2.  void eat(){System.out.println("eating...");}
3.  }
4.  class Dog extends Animal{
5.  void bark(){System.out.println("barking...");}
6.  }
7.  class Cat extends Animal{
8.  void meow(){System.out.println("meowing...");}
9.  }
10. class TestInheritance3{
11. public static void main(String args[]){
12. Cat c=new Cat();
13. c.meow();
14. c.eat();
15. //c.bark();//C.T.Error
16. }}
```

Output:

```
meowing...
eating...
```

Why multiple inheritance is not supported in java?

To reduce the complexity and simplify the language, multiple inheritance is not supported in java.

Consider a scenario where A, B, and C are three classes. The C class inherits A and B classes. If A and B classes have the same method and you call it from child class object, there will be ambiguity to call the method of A or B class.

Since compile-time errors are better than runtime errors, Java renders compile-time error if you inherit 2 classes. So whether you have same method or different, there will be compile time error.

```
1. class A{
2. void msg(){System.out.println("Hello");}
3. }
4. class B{
5. void msg(){System.out.println("Welcome");}
6. }
7. class C extends A,B{//suppose if it were
8.
9.  public static void main(String args[]){
10.  C obj=new C();
11.  obj.msg();//Now which msg() method would be invoked?
12. }
13. }
```

Output

```
Compile Time Error
```

\

Encapsulation in Java

Encapsulation in Java is a *process of wrapping code and data together into a single unit*, for example, a capsule which is mixed of several medicines.


Capsule

We can create a fully encapsulated class in Java by making all the data members of the class private. Now we can use setter and getter methods to set and get the data in it.

The **Java Bean** class is the example of a fully encapsulated class.

Simple Example of Encapsulation in Java

Let's see the simple example of encapsulation that has only one field with its setter and getter methods.

File: Student.java

```
1. //A Java class which is a fully encapsulated class.
2. //It has a private data member and getter and setter methods.
3. package com.javatpoint;
4. public class Student{
5. //private data member
6. private String name;
7. //getter method for name
8. public String getName(){
9. return name;
10. }
11. //setter method for name
12. public void setName(String name){
13. this.name=name
14. }
15. }
```

File: Test.java

```java
//A Java class to test the encapsulated class.
package com.javatpoint;
class Test{
public static void main(String[] args){
//creating instance of the encapsulated class
Student s=new Student();
//setting value in the name member
s.setName("vijay");
//getting value of the name member
System.out.println(s.getName());
}
}
```

Output:

```
vijay
```

Abstraction in Java

Abstraction is a process of hiding the implementation details and showing only functionality to the user.

Another way, it shows only essential things to the user and hides the internal details, for example, sending SMS where you type the text and send the message. You don't know the internal processing about the message delivery.

Abstraction lets you focus on what the object does instead of how it does it.

Ways to achieve Abstraction

There are two ways to achieve abstraction in java

1. Abstract class (0 to 100%)
2. Interface (100%)

1. Abstract class:

A class which contains the abstract keyword in its declaration is called abstract class.

Note:

- We can't create object for abstract class
- It may or may not contain any abstract method.
- It can have abstract & non-abstract methods
- To use an abstract class, you have to inherit it form sub class.

Code: Github.com/shuvra442

Abstract Method:

A method which contains the abstract keyword in its declaration is called abstract method.

Note:

- Any class that contains one or more abstract methods must also be declared abstract.
- If a class contains an abstract method it needs to be abstract and vice versa is not true.
- If a non-abstract class extends an abstract class, then the class must implement all the abstract methods of the abstract class else the concrete class has to be declared as abstract as well.
- The following are various **illegal combinations** of other modifiers for methods with respect to abstract modifiers:
    - final
    - abstract native
    - abstract synchronized
    - abstract static
    - abstract private
    - abstract strict

Code: Github.com/shuvra442

Interface

Interface is like a class, which contains only abstract method.

To archive interface java provides keyword called "**implements**".

Note:

- Interface variable is by default public+ static + final.
- Interface variable is by default public + abstract.
- Interface method must be overridden inside the implement's classes.
- Interface noting but deals between client & developer.

Code: Github.com/shuvra442

Exceptions in Java

Exception Handling in Java is one of the effective means to handle runtime errors so that the regular flow of the application can be preserved. Java Exception Handling is a mechanism to handle runtime errors such as ClassNotFoundException, IOException, SQLException, RemoteException, etc.

What are Java Exceptions?

In Java, Exception is an unwanted or unexpected event, which occurs during the execution of a program, i.e. at run time, that disrupts the normal flow of the program's instructions. Exceptions can be caught and handled by the program. When an exception occurs within a method, it creates an object. This object is called the exception object. It contains information about the exception, such as the name and description of the exception and the state of the program when the exception occurred.

Major reasons why an exception Occurs

- Invalid user input
- Device failure
- Loss of network connection
- Physical limitations (out-of-disk memory)
- Code errors
- Opening an unavailable file

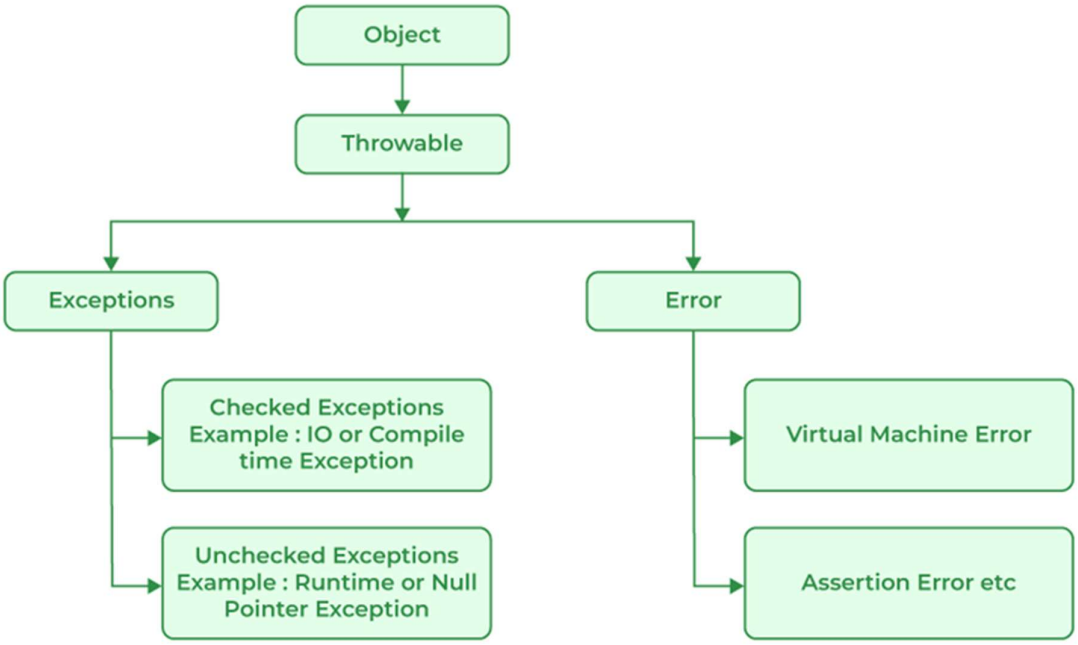
Difference between Error and Exception

Let us discuss the most important part which is the **differences between Error and Exception** that is as follows:

- **Error:** An Error indicates a serious problem that a reasonable application should not try to catch.
- **Exception:** Exception indicates conditions that a reasonable application might try to catch.

Exception Hierarchy

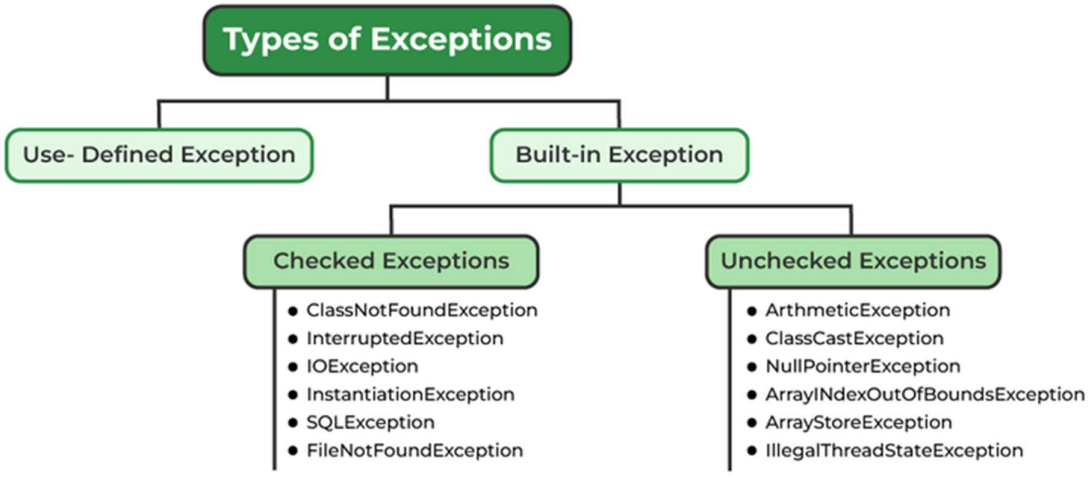
All exception and error types are subclasses of the class **Throwable**, which is the base class of the hierarchy. One branch is headed by **Exception**. This class is used for exceptional conditions that user programs should catch. NullPointerException is an example of such an exception. Another branch, **Error** is used by the Java run-time system(JVM) to indicate errors having to do with the run-time environment itself(JRE). StackOverflowError is an example of such an error.



Java Exception Hierarchy

Types of Exceptions

Java defines several types of exceptions that relate to its various class libraries. Java also allows users to define their own exceptions.



Exceptions can be categorized in two ways:

1. **Built-in Exceptions**
    - Checked Exception
    - Unchecked Exception

2. **User-Defined Exceptions**

Let us discuss the above-defined listed exception that is as follows:

1. Built-in Exceptions

Built-in exceptions are the exceptions that are available in Java libraries. These exceptions are suitable to explain certain error situations.

- **Checked Exceptions:** Checked exceptions are called compile-time exceptions because these exceptions are checked at compile-time by the compiler.

- **Unchecked Exceptions:** The unchecked exceptions are just opposite to the checked exceptions. The compiler will not check these exceptions at compile time. In simple words, if a program throws an unchecked exception, and even if we didn't handle or declare it, the program would not give a compilation error.

2. User-Defined Exceptions:

Sometimes, the built-in exceptions in Java are not able to describe a certain situation. In such cases, users can also create exceptions, which are called 'user-defined Exceptions'.

The ***advantages of Exception Handling in Java*** are as follows:

1. Provision to Complete Program Execution
2. Easy Identification of Program Code and Error-Handling Code
3. Propagation of Errors
4. Meaningful Error Reporting
5. Identifying Error Types

Why Does an Exception occur?

An exception can occur due to several reasons like a Network connection problem, Bad input provided by a user, Opening a non-existing file in your program, etc

Blocks and Keywords Used for Exception Handling

1. try in Java

The **try** block contains a set of statements where an exception can occur.

```
try
{
    // statement(s) that might cause exception
}
```

2. catch in Java

The catch block is used to handle the uncertain condition of a try block. A try block is always followed by a catch block, which handles the exception that occurs in the associated try block.

```
catch
{
   // statement(s) that handle an exception
   // examples, closing a connection, closing
   // file, exiting the process after writing
   // details to a log file.
}
```

3. throw in Java

The throw keyword is used to transfer control from the try block to the catch block.

Below is the implementation of the above approach:

- Java

```
// Java program that demonstrates the use of throw

class ThrowExcep {

    static void help()

    {
```

```
        try {
            throw new NullPointerException("error_unknown");
        }
        catch (NullPointerException e) {
            System.out.println("Caught inside help().");
            // rethrowing the exception
            throw e;
        }
    }

    public static void main(String args[])
    {
        try {
            help();
        }
        catch (NullPointerException e) {
            System.out.println(
                "Caught in main error name given below:");
            System.out.println(e);
        }
    }
}
```

Output

```
Caught inside help().
Caught in main error name given below:
java.lang.NullPointerException: error_unknown
```

4. **throws in Java**

The **throws** keyword is used for exception handling without try & catch block. It specifies the exceptions that a method can throw to the caller and does not handle itself.
Below is the implementation of the above approach:

- Java

```
// Java program to demonstrate working of throws

class ThrowsExecp {


    // This method throws an exception

    // to be handled

    // by caller or caller

    // of caller and so on.

    static void fun() throws IllegalAccessException

    {

        System.out.println("Inside fun(). ");

        throw new IllegalAccessException("demo");

    }


    // This is a caller function

    public static void main(String args[])

    {

        try {

            fun();

        }

        catch (IllegalAccessException e) {
```

```
            System.out.println("caught in main.");
        }
    }
}
```

Output

```
Inside fun().
caught in main.
```

5. **finally in Java**

It is executed after the catch block. We use it to put some common code (to be executed irrespective of whether an exception has occurred or not) when there are multiple catch blocks.
An example of an exception generated by the system is given below:

```
Exception in thread "main"
java.lang.ArithmeticException: divide
by zero at ExceptionDemo.main(ExceptionDemo.java:5)
ExceptionDemo: The class name
main:The method name
ExceptionDemo.java:The file name
java:5:line number
```

Below is the implementation of the above approach:

- Java

```
// Java program to demonstrate working of try,
// catch and finally

class Division {
    public static void main(String[] args)
    {
```

```
        int a = 10, b = 5, c = 5, result;
        try {
            result = a / (b - c);
            System.out.println("result" + result);
        }

        catch (ArithmeticException e) {
            System.out.println("Exception caught:Division by zero");
        }

        finally {
            System.out.println("I am in final block");
        }
    }
}
```

Output

```
Exception caught:Division by zero
I am in final block
```

Differences between `throw` and `throws`:

throw	throws
Used to throw an exception for a method	Used to indicate what exception type may be thrown by a method

Cannot throw multiple exceptions	Can declare multiple exceptions
Syntax: • `throw` is followed by an object (new *type*) • used inside the method	**Syntax:** • `throws` is followed by a class • and used with the method signature

Code: Github.com/shuvra442

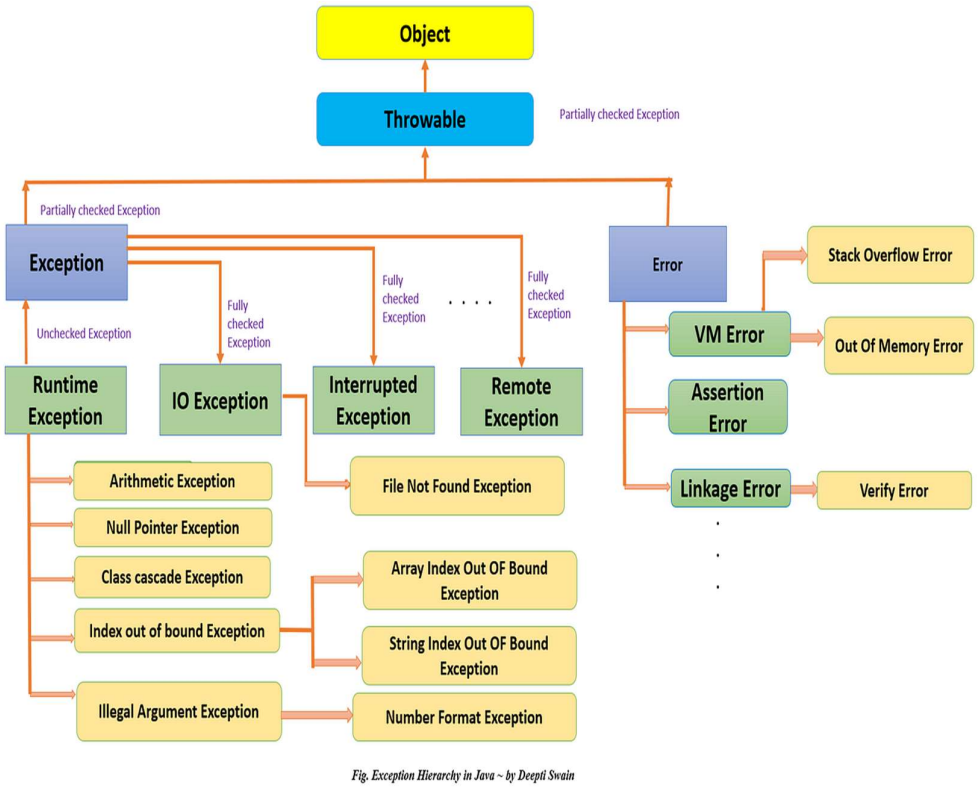
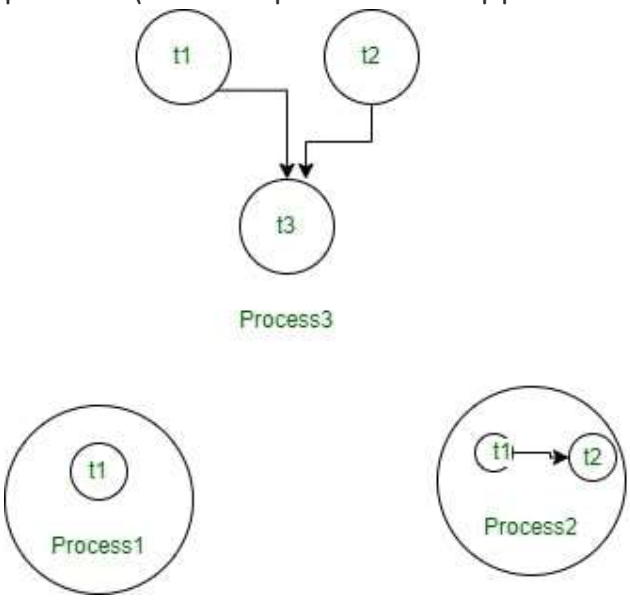


Fig. Exception Hierarchy in Java ~ by Deepti Swain

JAVA Threads

Typically, we can define threads as a subprocess with lightweight with the smallest unit of processes and also has separate paths of execution. The main advantage of multiple threads is efficiency (allowing multiple things at the same time. For example, in MS Word. one thread automatically formats the document while another thread is taking user input. Another advantage is quick response, if we use multiple threads in a process and if a thread gets stuck due to lack of resources or an exception, the other threads can continue to execution, allowing the process (which represents an application) to continue to be responsive.



As we can observe in, the above diagram a thread runs inside the process and there will be context-based switching between threads there can be multiple processes running in OS, and each process again can have multiple threads running simultaneously. The Multithreading concept is popularly applied in games, animation...etc.

The Concept Of Multitasking

To help users Operating System accommodates users the privilege of multitasking, where users can perform multiple actions simultaneously on the machine. This Multitasking can be enabled in two ways:

1. **Process-Based Multitasking**
2. **Thread-Based Multitasking**

Thread-Based Multitasking

As we discussed above Threads are provided with lightweight nature and share the same address space, and the cost of communication between threads is also low.

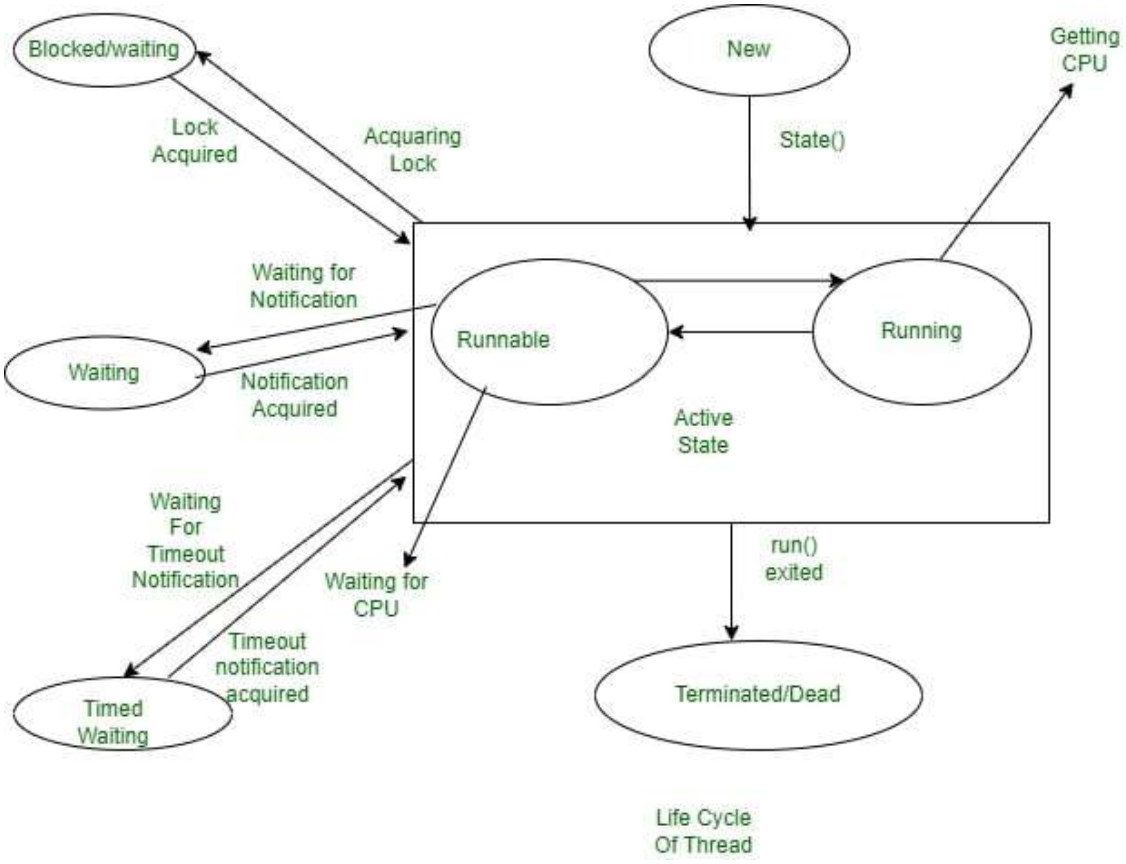
Why Threads are used?

Now, we can understand why threads are being used as they had the advantage of being lightweight and can provide communication between multiple threads at a Low Cost contributing to effective multi-tasking within a shared memory environment.

Life Cycle Of Thread

There are different states Thread transfers into during its lifetime, let us know about those states in the following lines: in its lifetime, a thread undergoes the following states, namely:

1. New State
2. Active State
3. Waiting/Blocked State
4. Timed Waiting State
5. Terminated State



We can see the working of different states in a Thread in the above Diagram, let us know in detail each and every state:

1. New State

By default, a Thread will be in a new state, in this state, code has not yet been run and the execution process is not yet initiated.

2. Active State

A Thread that is a new state by default gets transferred to Active state when it invokes the start () method, his Active state contains two sub-states namely:

- **Runnable State:** In This State, The Thread is ready to run at any given time and it's the job of the Thread Scheduler to provide the thread time for the runnable state preserved threads. A program that has obtained Multithreading shares slices of time intervals which are shared between threads hence, these threads run for some short span of time and wait in the runnable state to get their schedules slice of a time interval.
- **Running State:** When The Thread Receives CPU allocated by Thread Scheduler, it transfers from the "Runnable" state to the "Running" state. and after the expiry of its given time slice

session, it again moves back to the “Runnable” state and waits for its next time slice.

3. Waiting/Blocked State

If a Thread is inactive but on a temporary time, then either it is a waiting or blocked state, for example, if there are two threads, T1 and T2 where T1 needs to communicate to the camera and the other thread T2 already using a camera to scan then T1 waits until T2 Thread completes its work, at this state T1 is parked in waiting for the state, and in another scenario, the user called two Threads T2 and T3 with the same functionality and both had same time slice given by Thread Scheduler then both Threads T1, T2 is in a blocked state. When there are multiple threads parked in a Blocked/Waiting state Thread Scheduler clears Queue by rejecting unwanted Threads and allocating CPU on a priority basis.

4. Timed Waiting State

Sometimes the longer duration of waiting for threads causes starvation, if we take an example like there are two threads T1, T2 waiting for CPU and T1 is undergoing a Critical Coding operation and if it does not exist the CPU until its operation gets executed then T2 will be exposed to longer waiting with undetermined certainty, In order to avoid this starvation situation, we had Timed Waiting for the state to avoid that kind of scenario as in Timed Waiting, each thread has a time period for which sleep() method is invoked and after the time expires the Threads starts executing its task.

5. Terminated State

A thread will be in Terminated State, due to the below reasons:

- Termination is achieved by a Thread when it finishes its task Normally.
- Sometimes Threads may be terminated due to unusual events like segmentation faults, exceptions...etc. and such kind of Termination can be called Abnormal Termination.
- A terminated Thread means it is dead and no longer available.

What is Main Thread?

As we are familiar, we create Main Method in each and every Java Program, which acts as an entry point for the code to get executed by JVM, similarly in this Multithreading Concept, Each Program has one Main Thread which was provided by default by JVM, hence whenever a

program is being created in java, JVM provides the Main Thread for its Execution.

How to Create Threads using Java Programming Language?

We can create Threads in java using two ways, namely:

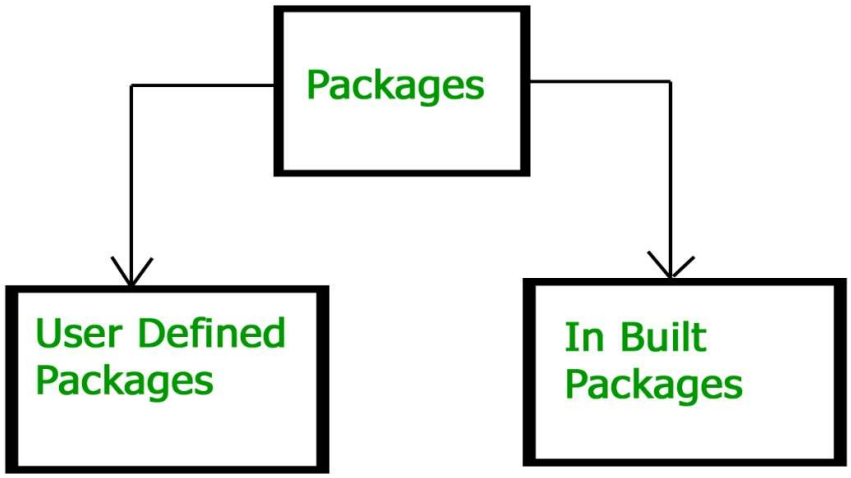
1. Extending Thread Class
2. Implementing a Runnable interface

Code: Github.com/shuvra442

Package

Package in Java is a mechanism to encapsulate a group of classes, sub packages and interfaces.

Types of packages:



Built-in Packages

These packages consist of a large number of classes which are a part of Java **API**.Some of the commonly used built-in packages are:

1. **java.lang:** Contains language support classes(e.g classes which defines primitive data types, math operations). This package is automatically imported.
2. **java.io:** Contains classes for supporting input / output operations.
3. **java.util:** Contains utility classes which implement data structures like Linked List, Dictionary and support ; for Date / Time operations.
4. **java.applet:** Contains classes for creating Applets.
5. **java.awt:** Contain classes for implementing the components for graphical user interfaces (like button , ;menus etc). 6)
6. **java.net:** Contain classes for supporting networking operations.

User-defined packages: These are the packages that are defined by the user. First, we create a directory **myPackage** (name should be same as the name of the package). Then create the **MyClass** inside the directory with the first statement being the **package names.**

Some important keywords in JAVA

1. **abstract:** It is a non-access modifier applicable for classes and methods. It is used to achieve abstraction. For more, refer to abstract keyword in java
2. **enum:** It is used to define enum in Java
3. **instanceof:** It is used to know whether the object is an instance of the specified type (class or subclass or interface).
4. **private:** It is an access modifier. Anything declared private cannot be seen outside of its class.
5. **protected:** If you want to allow an element to be seen outside your current package, but only to classes that subclass your class directly, then declare that element protected.
6. **public:** Anything declared public can be accessed from anywhere. For more on Access Modifiers, refer to Access Modifiers in Java

7. **static:** It is used to create a member(block, method, variable, nested classes) that can be used by itself, without reference to a specific instance. For more, refer static keyword in java

Super Keyword in Java

The **super keyword in Java** is a reference variable that is used to refer to parent class when we're working with objects. You need to know the basics of Inheritance and Polymorphism to understand the Java super keyword.
The Keyword **"super"** came into the picture with the concept of Inheritance. In this article, we gonna covers all about super in Java including definitions, examples, Uses, Syntax, and more.

Java this Keyword

Definition and Usage

The this keyword refers to the current object in a method or constructor.

The most common use of the this keyword is to eliminate the confusion between class attributes and parameters with the same name (because a class attribute is shadowed by a method or constructor parameter). If you omit the keyword in the example above, the output would be "0" instead of "5".

this can also be used to:

- Invoke current class constructor
- Invoke current class method
- Return the current class object
- Pass an argument in the method call
- Pass an argument in the constructor call

Java final Keyword

Definition and Usage

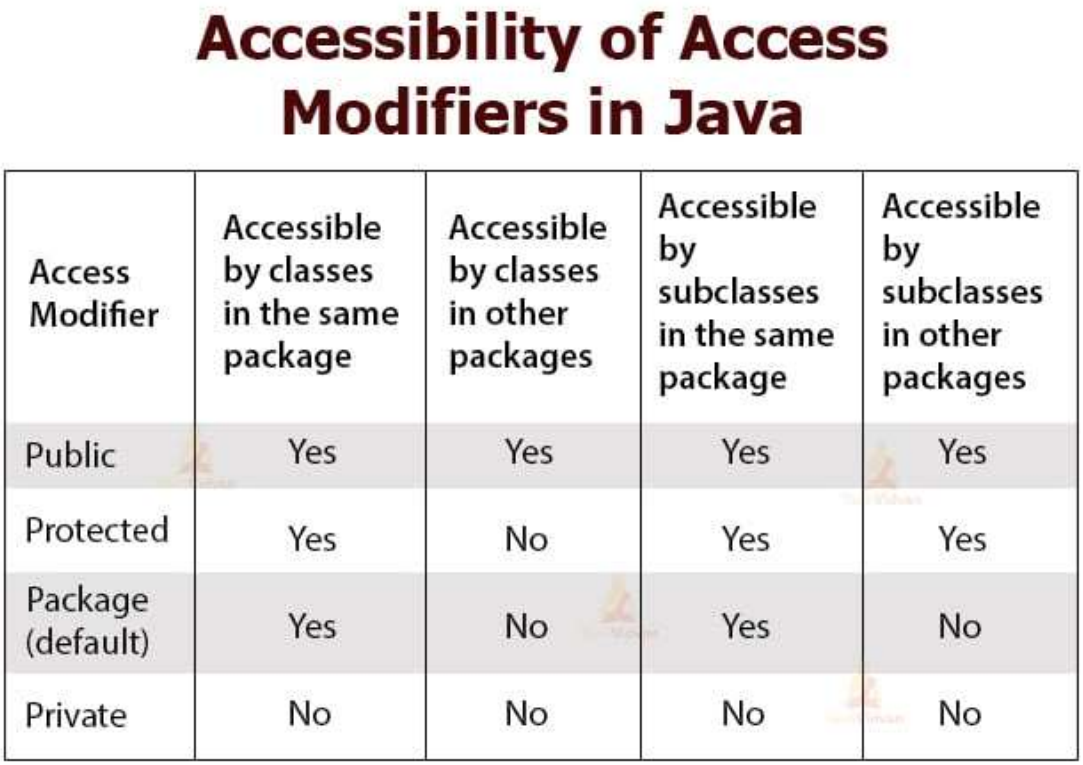
The final keyword is a non-access modifier used for classes, attributes and methods, which makes them non-changeable (impossible to inherit or override).

The final keyword is useful when you want a variable to always store the same value, like PI (3.14159...).

The final keyword is called a "modifier".

Access modifier

ss



Access Modifier	Accessible by classes in the same package	Accessible by classes in other packages	Accessible by subclasses in the same package	Accessible by subclasses in other packages
Public	Yes	Yes	Yes	Yes
Protected	Yes	No	Yes	Yes
Package (default)	Yes	No	Yes	No
Private	No	No	No	No