JAVA-Selenium

Java  
====  
-  Java is a high-level, object-oriented programming   
  language.   
  
-  This language is very easy to learn and widely used.   
  
-  It is known for its platform independence, reliability and security. It follows one principle, that is:  
  "Write Once, Run Anywhere"(WORA) principle.

Java's origin  
-------------

-  Java's history begins in 1991 at Sun Microsystems, where a team led by "James Gosling" aimed to create a new programming language for digital devices.

-  Initially called Oak, it was later renamed Java, inspired by Indonesian coffee beans.   
  
-  The language's core principle of "Write Once,   
  Run Anywhere" (platform independence) fueled its   
  rapid adoption, especially with the rise of the   
  internet.

Java Versions  
==============  
1991-1994: The Green Project and Oak. Sun Microsystems'   
"Green Team" (including Gosling, Mike Sheridan,   
and Patrick Naughton) initiated the project to develop   
a language for embedded systems, focusing on consumer   
electronics. The initial language was called Oak,   
named after an oak tree outside Gosling's office.

1995: Java's Public Debut.   
Sun Microsystems officially   
launched Java, emphasizing its platform-independent   
nature. The "HotJava" browser was introduced to showcase   
Java applets.

1996: First Version (1.0). The initial version of Java, 1.0,  
was released.

1997: Standardization.   
Java was standardized by ISO and ANSI, solidifying its   
position as a reliable language.

1999: Java 2 Platform (J2SE, J2EE, J2ME). Java was divided   
into three editions to cater to different needs.

2004: Java 5 and Enhancements. Java 5 introduced significant   
language enhancements, including generics and other   
features.

2006: Open Sourcing.   
Sun Microsystems made Java open source by releasing   
the OpenJDK.

2010: Oracle Acquisition.   
Oracle acquired Sun Microsystems,taking over Java's   
development and future.

2014: Java 8 and Modern Features.   
Java 8 brought in features like lambda expressions and a new Date and Time   
API.

2017: Java 9 and Modularization.   
Java 9 introduced the   
module system (JPMS) for modularizing applications.

Continued Evolution: Java continues to evolve with regular   
releases, incorporating new features and improvements.

Features of Java:  
=================  
1.  Simple Syntax  
  Java syntax is very straightforward and very easy to   
  learn. Java removes complex features like pointers   
  and multiple inheritance, which makes it a good   
  choice for beginners.    
  
2.  Object Oriented  
  Java is a pure object-oriented language. It supports   
  core OOP concepts like:  
  1.  Class - It is a bluprint of the object.  
        - It is a collection of "SIMILAR" objects.  
    Car(CLASS) - BMW,Audi,Mercedez,Aston Martin(OBJECTS)  
    HONDA(Class) - HONDA CRV,HONDA CIVIC,HONDA ACCORD,  
             HONDA CITY.    
      (VEHICLES)  
      Truck,Bus,Trailer,  
       - It is a logical construct.  
  
  2.  Objects -   It is a physical construct.It is an   
          instance of the class.  
  
  3.  Inheritance - A CHILD inherits the properties or behavoiur  
           from their PARENTS.  
  
  4.  Encapsulation - Wrapping of data into a single unit and   
            that unit has a name.  
  
  5.  Abstraction - Hiding unecessary data from the user so that  
            he/she should be concentrate on the logic.  
            printf("Hello");  
  
  6.  Polymorphism - 1 single entity can have many forms  
            POLY(Many) + MORPHISM(Forms)  
  
3.  Platform Independent  
  Java is platform-independent because of Java Virtual   
  Machine (JVM).

-  When we write Java code, it is first compiled by the   
  compiler and then converted into bytecode(intermediate code)  
  (which is platform-independent).  
-  This byte code can run on any platform which has   
  JVM installed.    
  
  Source code          ByteCode  
  abc.java----->javac------abc.class----java abc----output  
  
  =======================================================  
  JVM - Java Virtual Machine - It is PLATFORM DEPENDENT.  
  ========================================================  
  
  JVM            JVM          JVM  
  WINDOWS          LINUX        MAC  
  javac abc.java      java abc      java abc    
  (WORA)  
  
4.  Interpreted  
-  Java code is not directly executed by the computer.  It is first compiled into bytecode. This byte code is then understand by the JVM. This enables Java to run on any platform without rewriting code.    
  
5.  Scalable  
-  Java can handle both small and large-scale applications.  
  Java provides features like multithreading and   
  distributed computing that allows developers to   
  manage loads more easily.    
  
6.  Portable  
-  When we write a Java program, the code first get   
  converted into bytecode and this bytecode does not   
  depend on any operating system or any specific computer.   
  We can simply execute this bytecode on any platform with   
  the help of JVM. Since JVMs are available on all the OS's   
  and that's why we can run the same Java program on   
  different platforms.  
  
7.  Secured and Robust  
-  Java is a reliable programming language because it   
  can catch mistakes early while writing the code and   
  also keeps checking for errors when the program is   
  running. It also has a feature called exception   
  handling that helps deal with unexpected problems   
  smoothly.    
  
8.  Memory Management  
-  Memory management in Java is automatically handled   
  by the Java Virtual Machine (JVM).

-  Java garbage collector reclaim memory from objects that   
  are no longer needed.  
  
-  "Memory for objects are allocated in the "HEAP".

-  Method calls and local variables are stored in the   
  STACK.    
  
9.  High Performance  
-  Java is faster than old interpreted languages. Java   
  program is first converted into bytecode which is   
  faster than interpreted code.     
  
10.  Multithreading  
-  Multithreading in Java allows multiple threads to run   
  at the same time.  
-  It improves CPU utilization and enhancing performance   
  in applications that require concurrent task execution.    
  
11.  Rich Standard Library  
-  Java provides various pre-built tools and libraries   
  which is known as Java API. Java API is used to cover   
  tasks like file handling, networking, database connectivity   
  (JDBC), security, etc.     
  
12.  Functional Programming Features  
-  Since Java 8, the language has introduced functional   
  programming features such as:    
  1.  lambda expression  
  2.  Stream API  
  3.  Functional Interfaces  
  
13.  Integration with Other Technologies  
-  Java can easily work with many languages and tools as   
  well. For example, Java can connect with C and C++   
  with the help of Java Native Interface (JNI).   
  Java is very popular for building websites and   
  webservices like RESTful & SOAP.    
  
14.  Support for Mobile and Web Application  
-  Java offers support for both web and mobile applications.

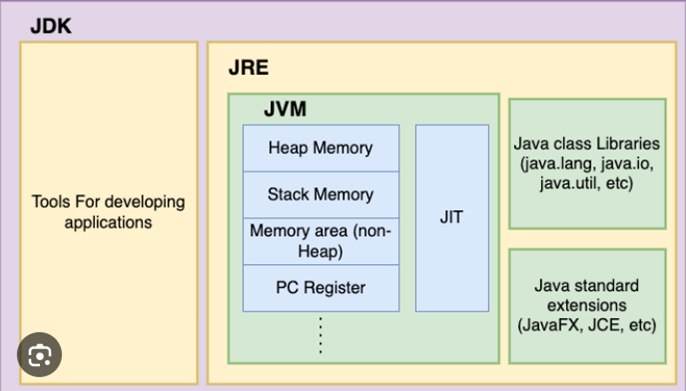
15.  Documentation and Community Support  
-  Java provide documentation which includes guides,   
  API references, and tutorials for easy learning.   
  Java has a large and active global community   
  contributing to open-source projects, and resources.

=================  
JDK, JRE, and JVM are three core components of the Java   
ecosystem, each with distinct roles.   
-  The JDK (Java Development Kit) is a software   
  development kit used for creating Java applications.   
  JDK ----> CREATE JAVA APPS.  
  
-  The JRE (Java Runtime Environment) provides the   
  environment for running Java applications.  
  JRE ----> RUN JAVA APPS  
  
-  The JVM (Java Virtual Machine) is the engine that   
  executes Java bytecode, enabling platform independence.   
  JVM ----> EXECUTES BYTECODE

JDK (Java Development Kit):  
It's a kit for Java developers, providing tools like   
compilers, debuggers, and documentation generators.   
It includes the JRE, so when you install the JDK, you also   
get the environment to run Java programs.   
Essentially, it's what you need to develop Java code.

JRE (Java Runtime Environment):   
It provides the runtime environment for Java applications, including the JVM and necessary libraries.   
It's what you need to run Java applications, but not to develop them.   
It's platform-dependent, meaning you need a JRE specific to your operating system.

JVM (Java Virtual Machine):  
It's the core of the Java runtime environment, responsible for executing Java bytecode.   
It provides platform independence by abstracting away the   
underlying hardware and operating system.   
JVM - java abc(bytecode)  
==========================================================  
It interprets bytecode into machine-specific instructions, allowing Java code to run on different platforms.   
===========================================================  
It's a key component of the JRE.



Java Keywords  
==============  
-  In Java, keywords are predefined, reserved words that have specific meanings to the Java compiler and cannot be used as identifiers (e.g., variable names, method names,  
  class names).   
  
Below is the list of common Java keywords and their general purposes:  
=====================================================  
1.  Access Modifiers: public, private, protected, static,   
  final, abstract

2.  Class and Object related: class, interface, enum,   
  extends, implements, new, this, super, instanceof

3.  Data Types: byte, short, int, long, float, double,   
  boolean, char, void

4.  Control Flow: if, else, switch, case, default, for,   
  while, do, break, continue, return

5.  Exception Handling: try, catch, finally, throw, throws

6.  Package and Import: package, import

7.  Other: transient, volatile, strictfp, assert,   
  synchronized, native

8.  Keywords added in newer versions: var (Java 10),   
  record (Java 16), sealed, permits (Java 17)

// Comments are used to increase the readability of the program.  
//A Basic Java Program  
/\*  
Below is the basic   
program for the begiunners !!!  
\*/

// Define a class  
class MyClass{  
  
  //JVM Entry Point  
  public static void main(String[] args){  
  
    // Print the text  
    System.out.print("Hello Learners..");  
  }  
}

class DataType{  
  //String[] args - It is a command line argument  
  public static void main(String[] args){  
    System.out.println("args[0]: "+args[0]);  
    System.out.println("args[1]: "+args[1]);  
  
    System.out.println("Addition: "+(args[0]+args[1]));  
  
    // Convert the String to int data type   
    //using WRAPPER CLASS(Integer)  
    int num1 = Integer.parseInt(args[0]);  
    int num2 = Integer.parseInt(args[1]);  
    System.out.println(" New Addition is: "+(num1+num2));  
  
    //Convert the String to a double data type  
    double d1 = Double.parseDouble(args[2]);  
    double d2 = Double.parseDouble(args[3]);  
    System.out.println("Decimal Addition is: "+(d1+d2));  
  
    int x=234;  
    x=x+100;  
    System.out.println("X: "+x);  
  
    String y = "123";  
    y= y+100;  
    System.out.println("Y: "+y);  
  
    // Convert the integer to a String  
    System.out.println("New Str: "+String.valueOf(x)+200);  
    String str = String.valueOf(x);  
    str = str+100;  
    System.out.println("Str: "+str);  
  
    //Task  
    System.out.println("Java"+20+30);  
    System.out.println(20+30+"Java"+20+30);  
    //System.out.println("Args[4]: "+args[4]);  
  
  }  
}

class DataType{  
  //String[] args - It is a command line argument  
  //public static void main(String[] args){  
  
  // Variable argument   
  static public void main(String...args){  
  
    System.out.println("args[0]: "+args[0]);  
    System.out.println("args[1]: "+args[1]);  
  
    System.out.println("Addition: "+(args[0]+args[1]));  
  
    // Convert the String to int data type   
    //using WRAPPER CLASS(Integer)  
    int num1 = Integer.parseInt(args[0]);  
    int num2 = Integer.parseInt(args[1]);  
    System.out.println(" New Addition is: "+(num1+num2));  
  
    //Convert the String to a double data type  
    double d1 = Double.parseDouble(args[2]);  
    double d2 = Double.parseDouble(args[3]);  
    System.out.println("Decimal Addition is: "+(d1+d2));  
  
    int x=234;  
    x=x+100;  
    System.out.println("X: "+x);  
  
    String y = "123";  
    y= y+100;  
    System.out.println("Y: "+y);  
  
    // Convert the integer to a String  
    System.out.println("New Str: "+String.valueOf(x)+200);  
    String str = String.valueOf(x);  
    str = str+100;  
    System.out.println("Str: "+str);  
  
    //Task  
    System.out.println("Java"+20+30);  
    System.out.println(20+30+"Java"+20+30);  
    //System.out.println("Args[4]: "+args[4]);  
  
    System.out.println("Welcome");  
  }  
}

Garbage Collection  
===================  
-  Garbage Collection (GC) in Java is an automatic memory management process that frees up memory by identifying and reclaiming objects that are no longer referenced or needed by the program.   
This automated process is a key feature of the Java Virtual Machine (JVM) and helps prevent memory leaks and simplifies development by eliminating the need for manual memory deallocation.

How it works:   
=============

Object Creation and Heap Memory:  
----------------------------------  
When a Java program runs, objects are created and stored   
in the heap memory, which is a dedicated portion of memory   
managed by the JVM.

Identifying Unreachable Objects:  
--------------------------------  
The Garbage Collector continuously monitors the heap and   
identifies objects that are no longer reachable or   
referenced by any part of the running program.   
This means there are no active references (variables,   
data structures, etc.) pointing to these objects.

Reclaiming Memory:  
------------------  
Once an object is determined to be unreachable, the   
Garbage Collector reclaims the memory occupied by that   
object, making it available for new object allocations.

Automatic Process:  
------------------  
Unlike languages like C or C++ where developers explicitly   
manage memory allocation and deallocation,   
Java's GC handles this automatically. This reduces the   
risk of memory-related errors and allows developers to   
focus on application logic.

**JVM Architecture**

The **Java Virtual Machine (JVM)** is the engine that enables Java's **platform independence**, allowing Java programs to run on any device or operating system that has a JVM implementation.

Here's a breakdown of the **JVM architecture**:

**JVM Architecture Overview**

+-------------------------------------------+

|               Class Loader Subsystem      |

+-------------------------------------------+

↓

+-------------------------------------------+

|              Runtime Data Areas           |

|   (Method Area, Heap, Stack, PC Register, |

|    Native Method Stack)                   |

+-------------------------------------------+

↓

+-------------------------------------------+

|              Execution Engine             |

|   (Interpreter, JIT Compiler, GC, etc.)   |

+-------------------------------------------+

↓

+-------------------------------------------+

|         Native Interface (JNI)            |

+-------------------------------------------+

↓

+-------------------------------------------+

|         Native Method Libraries           |

+-------------------------------------------+

**Key Components Explained**

**1. Class Loader Subsystem**

* **Loads .class files** (bytecode) into memory.
* Performs:
  + **Loading**
  + **Linking** (verifying, preparing, resolving)
  + **Initialization**
* Ensures classes are loaded only once.

**2. Runtime Data Areas**

These are memory areas used during program execution:

**a. Method Area**

* Stores class-level data like:
  + Class structure
  + Field & method data
  + Static variables

**b. Heap**

* Stores **objects** and **instance variables**.
* Shared among all threads.

**c. Java Stack (per thread)**

* Stores:
  + Method frames
  + Local variables
  + Operand stacks
* Each thread has its own stack.

**d. Program Counter (PC) Register**

* Holds the **address of the current instruction** being executed per thread.

**e. Native Method Stack**

* Supports native methods written in **C/C++** (outside Java).
* Uses **JNI (Java Native Interface)**.

**3. Execution Engine**

Executes the bytecode.

**a. Interpreter**

* Reads and executes bytecode **line by line**.
* Slower but quick to start.

**b. JIT Compiler (Just-In-Time)**

* Converts bytecode to **native machine code**.
* Improves performance by caching frequently-used code.

**c. Garbage Collector**

* Automatically manages memory.
* Reclaims memory from unused objects in the heap.

**4. Native Interface (JNI)**

* Enables Java code to **interact with native applications** and libraries (written in C, C++, etc.).

**5. Native Method Libraries**

* System-level libraries (e.g., libc.so on Linux, msvcrt.dll on Windows).

**Summary**

| **Component** | **Role** |
| --- | --- |
| Class Loader | Loads .class files into JVM |
| Runtime Data Areas | Stores data, code, and objects |
| Execution Engine | Executes the program |
| JNI & Native Libs | Bridge between Java and native code |

------------------

Data types in java  
===================  
Java categorizes data types into two main groups:   
Primitive data types and non-primitive(or reference)data   
types.

==============================================================  
NOTE: We cannot create the OBJECT of any PRIMITIVE data type.  
===============================================================  
Due to the presence of PRIMITIVE data types,   
JAVA is not "TRUE" ObjectOrientedLanguage.

Primitive Data Types:  
---------------------  
These are fundamental data types that directly store values and are predefined in Java. There are eight primitive data types:

Integral Types:  
---------------  
1.  byte: 8-bit signed integer(1 byte) (-128 to 127).  
2.  short: 16-bit signed integer(2 bytes) (-32,768 to 32,767).  
3.  int: 32-bit signed integer(4 bytes) (-2,147,483,648 to 2,147,483,647). This is the default integer type.  
4.  long: 64-bit signed integer(8 bytes) (-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807). Used for very large integer values.

Floating-Point Types:  
---------------------  
1.  float: 32-bit(4 bytes) single-precision floating-point number.  
2.  double: 64-bit(8 bytes) double-precision floating-point number. This is the default floating-point type.

Character Type:  
---------------  
1.  char: 16-bit(2 bytes) Unicode character (stores a single character).

Boolean Type:  
-------------  
1.  boolean: Stores true or false values.

Non-Primitive (Reference) Data Types:  
=====================================  
These data types do not store the actual values directly   
but rather store references (memory addresses) to objects. They are created by the programmer and include:  
1.  Classes: User-defined blueprints for creating objects.

2.  Interfaces: Blueprints of a class that define methods   
  but do not implement them.

3.  Arrays: Collections of elements of the same data type.  
4.  Strings: A sequence of characters (a special class in   
  Java).

PATH and CLASSPATH  
==================  
-  The CLASSPATH specifies where the JVM looks for .class   
  files.  
-  PATH (a separate environment variable) specifies   
  where the operating system looks for executable   
  programs (like java and javac).

-  Using the -classpath option is generally preferred   
  over modifying the environment variable for better   
  isolation and project-specific control.

**Operators in Java**

In Java, **operators** are special symbols or keywords used to perform operations on variables and values. Java operators are broadly categorized into following types:

**1. Arithmetic Operators**

Used to perform basic arithmetic operations.

| **Operator** | **Description** | **Example (a = 10, b = 5)** | **Result** |
| --- | --- | --- | --- |
| + | Addition | a + b | 15 |
| - | Subtraction | a – b | 5 |
| \* | Multiplication | a \* b | 50 |
| / | Division | a / b | 2(Quotient) |
| % | Modulus (remainder) | a % b | 0(Remainder) |

**2. Relational (Comparison) Operators**

Used to compare two or more than 2 values.

| **Operator** | **Description** | **Example (a = 10, b = 5)** | **Result** |
| --- | --- | --- | --- |
| == | Equal to(Comparison) | a == b | false |
| != | Not equal to | a != b | true |
| > | Greater than | a > b | true |
| < | Less than | a < b | false |
| >= | Greater than or equal | a >= b | true |
| <= | Less than or equal | a <= b | false |

**3. Logical Operators**

Used to combine multiple conditions.

| **Operator** | **Description** | **Example a=10,b=5** | **Result** |
| --- | --- | --- | --- |
| && | Logical AND | a > 5 && b < 10 | true |
| || | Logical OR a>5 | || b>10 | true |
| ! | Logical NOT | !(a == b) | true |

**4. Assignment Operators**

Used to assign values to variables.

| **Operator** | **Description** | **Example** | **Equivalent to** |
| --- | --- | --- | --- |
| = | Assign | a = 10 | — |
| += | Add and assign | a += 5 | a = a + 5 |
| -= | Subtract and assign | a -= 5 | a = a - 5 |
| \*= | Multiply and assign | a \*= 5 | a = a \* 5 |
| /= | Divide and assign | a /= 5 | a = a / 5 |
| %= | Modulus and assign | a %= 5 | a = a % 5 |

**5. Unary Operators**

Operate on a single operand.

| **Operator** | **Description** | **Example** | **Result** |
| --- | --- | --- | --- |
| + | Unary plus | +a | +10 |
| - | Unary minus | -a | -10 |
| ++ | Increment | a++ / ++a | 11 |
| -- | Decrement | a-- / --a | 9 |
|  |  |  |  |

**6. Bitwise Operators**

Operate on bits.

| **Operator** | **Description** | **Example (a = 5, b = 3)** | **Result** |
| --- | --- | --- | --- |
| & | Bitwise AND | a & b | 1 |
| ` | ` | Bitwise OR | `a |
| ^ | Bitwise XOR | a ^ b | 6 |
| ~ | Bitwise complement | ~a | -6 |
| << | Left shift | a << 1 | 10 |
| >> | Right shift | a >> 1 | 2 |
| >>> | Unsigned right shift | a >>> 1 | 2 |

**7. Ternary Operator**

A shorthand for if-else.

int result = (a > b) ? a : b; if(a>b){ sout(a);}else{sout(b);

This returns the greater of a and b.

**8. Instanceof Operator**

Checks if an object is an instance of a specific class.

if (obj instanceof String)

                                                                          ###########

class OpsDemo{  
  public static void main(String[] args){  
    int a=10,b=5;  
    System.out.println("Arithmetic Operators");  
    System.out.println("--------------------");  
  
    System.out.println("Addition: "+(a+b));  
    System.out.println("Subtraction: "+(a-b));  
    System.out.println("Multiplication: "+(a\*b));  
    System.out.println("Division: "+(a/b));  
    System.out.println("Modulus: "+(a%b));  
  
    System.out.println();  
    System.out.println("Relational Operators");  
    System.out.println("--------------------");  
    System.out.println("if a==b: "+(a==b));  
    System.out.println("if a!=b: "+(a!=b));  
    System.out.println("if a>b: "+(a>b));  
    System.out.println("if a<b: "+(a<b));  
    System.out.println("if a>=b: "+(a>=b));  
    System.out.println("if a<=b: "+(a<=b));  
  
    System.out.println();  
    System.out.println("Logical Operators");  
    System.out.println("--------------------");  
    System.out.println("Logical AND &&: "+(a<5 && b<10));  
    System.out.println("Logical OR ||: "+(a>5 || b<10));  
    System.out.println("Logical NOT !: "+!(a==b));  
  
    System.out.println();  
    System.out.println("Assignment Operators");  
    System.out.println("--------------------");  
    a +=20; //a = 30  
    a -=10; // a = 20   
    a \*=30; // a = 600  
    a /=50; // a = 12  
    a %=8; // a = 4  
    System.out.println("A: "+a);  
  
    System.out.println();  
    System.out.println("Unary Operators");  
    System.out.println("--------------------");  
    System.out.println("A: "+(-a));  
    a++; // post increment  
    ++a; // pre increment  
    ++a;  
    ++a;  
    a++; // a=9  
    System.out.println("New value of A: "+(a));  
    int c;  
    c= ++a; // a++; c=a;  
    a++;  
    ++a;  
    c=a++; // c=a; a++;  
    ++a;  
    c=a; //c =14  
    System.out.println("C: "+c);  
  
    c= --a; // a=13; c=13  
    --a; //a=12  
    a--; //a=11  
    c= a--; //c=11; a=10  
    a++; //a=11  
    c=a++; //c=11; a=12  
    System.out.println("New C: "+c);

**Bitwise Operators**

Bitwise operators are used to perform operations on **binary representations** of integers. These operators work at the **bit level**, meaning they directly manipulate individual bits of numbers.

| **Operator** | **Name** | **Symbol** | **Description** |
| --- | --- | --- | --- |
| AND | Bitwise AND | & | Sets each bit to 1 if both bits are 1 |
| OR | Bitwise OR | | | Sets each bit to 1 if one of the bits is 1 |
| XOR | Bitwise XOR | ^ | Sets each bit to 1 if only one of the bits is 1 |
| NOT | Bitwise NOT | ~ | Inverts all the bits (1 becomes 0, and 0 becomes 1) |
| << | Left Shift | << | Shifts bits to the left, filling with 0s on the right       a=5 00001010 a<<1 |
| >> | Right Shift | >> | Shifts bits to the right, discarding rightmost bits |

**Example (Using 8-bit representation)**

Let’s use a = 5 and b = 3.

a = 5     -> 00000101

b = 3     -> 00000011

* a & b → 00000001 → **1**
* a | b → 00000111 → **7**
* a ^ b → 00000110 → **6**
* ~a → 11111010 → **-6** (in two's complement form)
* a << 1 → 00001010 → **10**
* a >> 1 → 00000010 → **2**

System.out.println();  
    System.out.println("Bitwisw Operators");  
    System.out.println("--------------------");  
    a=5; //00000101  
            1  
      //11111010 ---> 1's complement  
      //11111011 ---> 2's complement      
    b=3;  
    System.out.println("a bitwise b(a&b): "+(a&b));  
    System.out.println("a bitwise b(a|b): "+(a|b));  
    System.out.println("a bitwise b(a^b): "+(a^b));  
    System.out.println("a leftshift by 1 bit(a<<1): "+(a<<1));  
    System.out.println("a rightshift by 1 bit(a>>1): "+(a>>1));  
    System.out.println("a 1's complement: "+(~a));  
  }  
}    
  
  
  }  
}

 class BitwiseOp{  
  public static void main(String[] args){  
    int a=5; // 00000101  
    int onescomplement = ~a; //11111010   
    System.out.println("Value of a is: "+a);  
  
    //1's complement  
    System.out.println("a 1's complement: "+onescomplement);  
  
    //2's complement  
    int twoscomplement = onescomplement+1;  
    System.out.println("a 2's complement: "+twoscomplement);  
  
  }  
}

Control Flow

**Control flow statements**

**Control flow statements** in Java are instructions that determine the **order in which code is executed** in a program. They enable the program to make decisions, repeat tasks, or jump to specific parts of the code based on certain conditions.

* Java has **three types** of control flow statements:

1. **Decision Making Statements** (**Make choices**) e.g., if, switch
2. **Looping Statements (Repeat actions)** e.g., for, while, do…while, enchanced for loop
3. **Jump Statements (Control execution)** e.g., break, continue, return

**1. Decision-Making Statements**

These are used to execute code conditionally.

**if Statement**

if (condition) {

   // code executes if condition is true

}

**if-else Statement**

if (condition) {

   // executes if true

} else {

   // executes if false

}

**if-else-if Ladder**

if (condition1) {

   // code

} else if (condition2) {

   // code

} else {

   // default code

}

**switch Statement**

Efficient alternative to many if-else blocks (for discrete values).

switch (expression) {

   case value1:

       // code

       break;

   case value2:

       // code

       break;

   default:

       // code

}

**2. Looping Statements**

Used to execute a block of code repeatedly.

**for Loop**

for (int i = 0; i < 5; i++) {

  System.out.println(i);

}

**while Loop**

int i = 0;

while (i < 5) {

  System.out.println(i);

   i++;

}

**do-while Loop**

Always runs the loop body at least once.

int i = 0;

do {

  System.out.println(i);

   i++;

} while (i < 5);

**Enhanced for-each Loop (for arrays/collections)**

int[] numbers = {1, 2, 3};

for (int n : numbers) {

  System.out.println(n);

}

**3. Jump Statements**

Used to transfer control to other parts of the code.

**break**

* Exits the current loop or switch.

for (int i = 0; i < 5; i++) {

   if (i == 3) break;

  System.out.println(i);

}

**continue**

* Skips the current iteration.

for (int i = 0; i < 5; i++) {

   if (i == 2) continue;

  System.out.println(i);

}

**return**

* Exits from the method.

public int add(int a, int b) {

   return a + b;    }

class CFlow{  
  public static void main(String[] args){  
    //if statement  
    int x=20;  
    int y=30;  
  
    if(x==20){  
      System.out.println("Valid Value: "+x);  
    }  
    else{  
      System.out.println("Invalid Value");  
    }  
  
    //Example 2  
    //if(x==20 && y==40){  
    if(x>=20 && x<=40){  
  
      System.out.println("Valid Value: "+x+","+y);  
    }  
    else{  
      System.out.println("Invalid Value Again");  
    }  
  
    //Example 3 ---if...elseif Ladder  
    //x=20 and y=30  
    if(x<10){  
      System.out.println("Valid X");  
    }  
    else if(y>=30){  
      System.out.println("Valid y");  
    }  
    else{  
      System.out.println("Invalid Values of x and y");  
    }  
  
    //Example 4  
    //Independent if statements  
    if(x==20){  
      System.out.println("Valid X");  
    }  
    if(y>=30){  
      System.out.println("Valid y");  
    }  
    if(x==y){  
      System.out.println("valid Values of x and y");  
    }  
    else{  
      System.out.println("XXXXXXXXXXXX");  
    }        
  
    String ch=null;  
    switch(ch){  
      case "n":{  
        System.out.println("No");  
        break;  
      }  
      //null --->It is an undetermined value.  
      // You cannot determine the value of null.  
      case null:{  
        System.out.println("Big NO");  
        break;  
      }  
      case "Y":  
      case "y":{  
        System.out.println("Yes");  
        break;  
      }  
      default:{  
        System.out.println("No Choice");  
        break; //optional   
      }  
    }  
      //if statement using with switch statement  
      String str="Jan";  
      if(str==null){  
        System.out.println("NULL");  
      }  
      else{  
        switch(str){  
          case "Jan":{  
            System.out.println("Jan");  
            break;  
          }  
          case "Feb":{  
            System.out.println("Feb");  
            break;  
          }  
          case "Mar":{  
            System.out.println("Mar");  
            break;  
          }  
        }  
      }  
  
      //Nested switch statement  
      int m=1;  
      int num=100;  
      switch(m){  
          case 1:{  
            switch(num){  
              case 100:{  
                System.out.println("100");  
                break;  
              }  
              case 200:{  
                System.out.println("200");  
                break;  
              }  
            }  
          break;  
          }  
          default:{  
            System.out.println("Invalid statement");  
          }  
      }  
  }  
  }

Arrays in Java

An array in Java is a **container object** that holds a fixed number of values of the same data type. It is used to store multiple values in a single variable instead of declaring separate variables for each value.

**Key Features of Arrays in Java**

* Fixed in size (cannot change after creation)
* Stores elements of the same type (e.g., all int or all String)
* Index-based (starting from 0)
* Can be single-dimensional or multi-dimensional (like 2D arrays)

**Declaration of Arrays**

// Declaration

int[] numbers;      // preferred

int numbers[];      // also valid

// Initialization

numbers = new int[10]; // creates an array of size 10 (default values: 0)

**Declaration + Initialization Together**

int[] numbers = new int[10];         // with default values (0s)

int[] marks = {90, 80, 70, 60};     // directly assigning values

**Accessing Elements**

System.out.println(marks[0]);   // prints 90

marks[2] = 75;                  // updates index 2 (70 -> 75)

**Looping Through Arrays**

**Using for loop:**

for (int i = 0; i < marks.length; i++) {

  System.out.println(marks[i]);

}

**Using for-each loop:**

for (int mark : marks) {

  System.out.println(mark);

}

**Multi-Dimensional Arrays (2D Array Example)**

int[][] matrix = {

   {1, 2, 3},

   {4, 5, 6}

};

System.out.println(matrix[0][1]);  // prints 2

**Common Array Operations**

| **Operation** | **Example** |
| --- | --- |
| Length of array | arr.length |
| Sorting (using Arrays) | Arrays.sort(arr); (from java.util) |
| Copying arrays | Arrays.copyOf(arr, newLength) |
| Searching | Arrays.binarySearch(arr, key) |

**Arrays class**

The Arrays class in Java is a utility class provided in the “java.util” package. It contains a collection of static methods for manipulating arrays (such as sorting, searching, comparing, and filling). It works with both primitive and object arrays.

**Importing the Arrays Class**

import java.util.Arrays;

Commonly Used Methods

**1. Sorting**

int[] arr = {5, 2, 9, 1};

Arrays.sort(arr);  // arr is now {1, 2, 5, 9}

**2. Binary Search**

The array must be sorted before using binarySearch.

int[] arr = {1, 2, 5, 9};

int index = Arrays.binarySearch(arr, 5);  // returns index: 2

**3. Filling**

int[] arr = new int[5];

Arrays.fill(arr, 6);  // arr becomes {6,6,6,6,6}

**4. Comparing Arrays**

int[] a = {1, 2, 3};

int[] b = {1, 2, 3};

boolean result = Arrays.equals(a, b);  // true

**5. Converting to String**

int[] arr = {1, 2, 3};

System.out.println(Arrays.toString(arr));  // prints [1, 2, 3]

**6. Copying Arrays**

int[] original = {1, 2, 3};

int[] copy = Arrays.copyOf(original, 5);  // {1, 2, 3, 0, 0}

**7. Parallel Sort (Java 8+)**

int[] arr = {5, 2, 1, 9};

Arrays.parallelSort(arr); // like sorting, but uses multithreading

**8. Deep Equality for Multidimensional Arrays (Comparing arrays)**

int[][] a = {{1, 2}, {3, 4}};

int[][] b = {{1, 2}, {3, 4}};

System.out.println(Arrays.deepEquals(a, b));  // true

**Example Program:**

import java.util.Arrays;

public class ArrayExample {

   public static void main(String[] args) {

       int[] nums = {5, 3, 8, 1};

      Arrays.sort(nums);

       for (int n : nums) {

          System.out.print(n + " ");

       }

   }

}

**Output:** 1 3 5 8

**Copy Array Example**

import java.util.Arrays;

public class CopyArrayExample {

   public static void main(String[] args) {

       int[] original = {10, 20, 30, 40, 50};

       // Copy first 3 elements

       int[] copy = Arrays.copyOf(original, 3);

       // Print copied array

      System.out.println("Copied Array: " + Arrays.toString(copy));

   }

}

**Output:**

Copied Array: [10, 20, 30]

**Example: Copy Full Array**

import java.util.Arrays;

public class CopyFullArray {

   public static void main(String[] args) {

       int[] original = {10, 20, 30, 40, 50};

       // Copy the entire array

       int[] copy = Arrays.copyOf(original, original.length);

      System.out.println("Original Array: " + Arrays.toString(original));

      System.out.println("Copied Array: " + Arrays.toString(copy));

   }

}

import java.util.Arrays;  
class JArray{  
  public static void main(String[] args){  
    //Declaration of an integer array  
    int[] numbers;  
    //Initialization of an integer array  
    numbers = new int[5];  
    System.out.println(numbers);  
  
    // 1st way   
    numbers[0]=444;  
    numbers[1]=4344;  
    numbers[2]=4464;  
    numbers[3]=4414;  
    numbers[4]=44554;  
  
    // Iterate the elements of an array using simple for loop  
    for(int i=0;i<numbers.length;i++){  
      System.out.println("numbers["+i+"]: "+numbers[i]);  
    }  
  
    // using enhanced for loop  
    for(int num: numbers){  
      System.out.println(num);  
    }  
  
    //To declare and initialize a single dim. array  
    int[] nums={2554,5467,77,87,8,86,6,67,6,7};  
    for(int num\_1: nums){  
      System.out.print(num\_1+",");  
    }  
    System.out.println();  
    //To sort an array using Arrays class  
    System.out.println("Sorted Array");  
  
    // To sort an array using Arrays class  
    Arrays.sort(nums);  
    //To display the sorted array....  
    System.out.println(Arrays.toString(nums));

    //BinarySearch using an Arrays class  
    int[] arr = {1, 2, 5, 9};  
    int index = Arrays.binarySearch(arr, 2);  // returns index: 2  
    System.out.println("The element is at index: "+index);  
  
    // To fill array with the val;ue using Arrays class   
    //"fill()" static method  
    int[] arr1 = new int[5]; // 1 row and 5 columns  
    Arrays.fill(arr1, 6);  // arr becomes {6,6,6,6,6}  
    for(int n: arr1){  
      System.out.println(n);  
    }  
  
    //Comparing arrays with the Arrays equals() method  
    int[] a = {1, 2, 3};  
    int[] b = {1, 2, 3};  
    boolean result = Arrays.equals(a, b);  // true  
    System.out.println(result);  
  
    //Copying an array using Arrys copyOf() method  
    int[] original = {1, 2, 3};  
    int[] copy = Arrays.copyOf(original, original.length);  // {1, 2, 3, 0, 0}  
    for(int cpy:copy){  
      System.out.println("Copy Array: "+cpy);  
    }  
  
    //ParallelSort using Arrays class  
    int[] psarr = {5, 2, 1, 9};  
    Arrays.parallelSort(psarr);  // like sorting, but uses multithreading  
    System.out.println(Arrays.toString(psarr));  
  
    //Multi-dimensional arrays  
    int[][] mdarr=new int[4][5]; // 4 rows 5 cols.  
  
    int[][] darr = {{1,2,3,4,6},  
            {2,5,8,9,3},  
            {2,6,7,8,5},  
            {8,9,0,4,1}  
             };  
    for(int row=0;row<4;row++){  
      for(int col=0;col<5;col++){  
        System.out.println("darr["+row+"]["+col+"]: "+darr[row][col]);  
      }          
    }  
  
    //Manually  
    mdarr[0][0]=33;  
  
    //Deep Equality for MD Arrays  
    int[][] a1 = {{1, 2}, {3, 4}};  
    int[][] b1 = {{1, 2}, {3, 4}};  
    System.out.println(Arrays.deepEquals(a1, b1));  // true

  }  
}

**String class in Java**

In Java, the String class is a **built-in class** that represents a **sequence of characters**.

It is part of the “java.lang” package, so we don’t need to import it explicitly.

**Key Features of the String Class**

* **Immutable**: Once a String object is created, its value **cannot be changed**.
* **Stored in String pool**: Java maintains a pool for strings to optimize memory usage.
* **Final class**: Cannot be extended (i.e., public final class String).
* **Implements**: Serializable, Comparable<String>, and CharSequence.

**Creating Strings**

String s1 = "Hello";                    // using string literal

String s2 = new String("Hello");       // using new keyword

Both create strings, but only the first uses the **string pool**.

**Common String Methods**

| **Method** | **Description** | **Example** |
| --- | --- | --- |
| length() | Returns string length | s.length() |
| charAt(index) | Returns character at index | s.charAt(0) → 'H' |
| substring(start, end) | Returns a substring | s.substring(0, 4) → "Hell" |
| equals(str) | Compares content | s.equals("Hello") |
| equalsIgnoreCase(str) | Case-insensitive comparison | s.equalsIgnoreCase("hello") |
| compareTo(str) | Lexicographic comparison | s.compareTo("World") |
| contains(seq) | Checks for substring | s.contains("ell") → true |
| toUpperCase() | Converts to uppercase | s.toUpperCase() |
| toLowerCase() | Converts to lowercase | s.toLowerCase() |
| trim() | Removes whitespace | " hello ".trim() → "hello" |
| replace(a, b) | Replaces characters | s.replace('H', 'J') → "Jello" |
| split(delimiter) | Splits string into array | "a,b,c".split(",") → ["a","b","c"] |
| indexOf(char) | Finds index of first occurrence | s.indexOf('e') |
| isEmpty() | Checks if string is empty | "".isEmpty() → true |

**Immutability Example**

String s = "Hello";

s.concat(" World");

System.out.println(s);  // Output: Hello (not Hello World)

//concat() does not change s, because strings are **immutable**.

To change it:

s = s.concat(" World");

System.out.println(s);  // Output: Hello World

**String Comparison**

String a = "Java";

//String b= “Java”;

String b = new String("Java");

System.out.println(a == b);         // false (different objects)

System.out.println(a.equals(b));    // true (same content)

**String Pool Concept**

String s1 = "Java";

String s2 = "Java";

System.out.println(s1 == s2);       // true (same pool reference)

Literal strings are stored in the **string pool** and reused.

**String Class Declaration (Simplified)**

public final class String implements java.io.Serializable,

                                   Comparable<String>,

                                   CharSequence {

   // many methods and fields

}

**StringBuffer Class in Java**

The StringBuffer class in Java is a part of the java.lang package and is used to create **mutable strings**. Unlike the String class, which is **immutable**, StringBuffer allows you to **modify the contents** of a string without creating a new object every time.

**Key Features of StringBuffer**

* **Mutable**: You can change the content (e.g., append, insert, delete).
* **Thread-safe**: All methods are **synchronized**, making it **safe for use in multi-threaded environments**.
* **More efficient** than String for repeated string manipulation (in single-threaded code, StringBuilder is faster).

**Creating a StringBuffer Object**

StringBuffer sb = new StringBuffer();               // empty buffer

StringBuffer sb2 = new StringBuffer("Hello");       // with initial string

**Common Methods of StringBuffer**

| **Method** | **Description** | **Example** |
| --- | --- | --- |
| append(String s) | Adds string to the end | sb.append(" World") |
| insert(int offset, String s) | Inserts string at a position | sb.insert(1, "Java") |
| replace(int start, int end, String s) | Replaces part of the string | sb.replace(0, 5, "Hi") |
| delete(int start, int end) | Deletes characters from start to end | sb.delete(0, 2) |
| reverse() | Reverses the string | sb.reverse() |
| toString() | Converts buffer to regular String | sb.toString() |
| length() | Returns current length | sb.length() |
| capacity() | Shows the current capacity (not length) | sb.capacity() |
| setCharAt(int index, char ch) | Changes a character at given index | sb.setCharAt(0, 'Y') |

**Example Code**

public class StringBufferDemo {

   public static void main(String[] args) {

       StringBuffer sb = new StringBuffer("Hello");

      sb.append(" World");      // Adds to end

       sb.insert(5, ",");         // Inserts at index 5

       sb.replace(0, 5, "Hi");    // Replaces "Hello" with "Hi"

       sb.delete(3, 5);           // Deletes characters

      sb.reverse();              // Reverses string

      System.out.println(sb);    // Output: "dlroW ,iH"

   }

}

**String vs StringBuffer vs StringBuilder**

| **Feature** | **String** | **StringBuffer** | **StringBuilder** |
| --- | --- | --- | --- |
| Mutability | Immutable | Mutable | Mutable |
| Thread-safe | No | Yes (synchronized) | No |
| Performance | Slower | Slower (thread-safe) | Faster (no sync) |
| Use case | When string never changes | Multi-threaded env | Single-threaded, fast changes |

class Impl{  
    public static void main(String[] args) {  
    String str="Welcome";  
    System.out.println("Length: "+str.length());  
    System.out.println("str position: "+str.charAt(3));  
    System.out.println("Substring: "+str.substring(0,3));  
    System.out.println("Compare 2 string: "+"Welcome".equals(str));  
    System.out.println("Compare 2 string: "+"welcome".equalsIgnoreCase(str));  
    System.out.println("CompareTo: "+"Welcome".compareTo(str));  
    System.out.println("str contains: "+str.contains("come"));  
    System.out.println("Uppercase: "+str.toUpperCase());  
    System.out.println("Lowercase: "+str.toLowerCase());  
    System.out.println("trim: "+"hello ".trim());  
    System.out.println("replace: "+str.replace("W","M"));  
    //String argument[] = str.split(",");  
    //System.out.println("split: "+argument);  
    System.out.println("Index: "+str.indexOf('c'));  
    System.out.println("isEmpty: "+str.isEmpty());

    //Immutable Example  
    String s = "Hello";  
    s.concat(" World");  
    System.out.println(s);   // Output: Hello (not Hello World)  
    //concat() does not change s,   
    //because strings are immutable.  
    s = s.concat(" World");  
    System.out.println(s);   // Output: Hello World

    //StringBuffer - mutable  
    StringBuffer sb = new StringBuffer("Hello");       // with initial string  
    sb.append("World");  
    System.out.println(sb);    
    sb.insert(1, "Java");  
    System.out.println(sb);    
    sb.replace(0, 5, "Hi");  
    System.out.println(sb);    
    sb.delete(0, 2);  
    System.out.println(sb);    
    sb.reverse();  
    System.out.println(sb);    
    System.out.println("capacity: "+sb.capacity());  
    sb.setCharAt(0, 'T');  
    System.out.println(sb);   
  
  
  
  }  
}