

# Unit-I

# Syllabus

- Review of Object oriented concepts
- History of Java
- Java buzzwords
- JVM architecture
- Data types
- Variables
- Scope and life time of variables
- arrays
- operators
- control statements
- type conversion and casting,
- simple java program,
- constructors,
- methods,
- Static block,
- Static Data,
- Static Method String and String Buffer Classes,
- Using Java API Document

# Object oriented concepts

# OOPs

- Object-Oriented Programming is a paradigm that provides many concepts, such as **Abstraction, Encapsulation, Inheritance, Polymorphism**, etc.
- The programming paradigm where everything is represented as an object is known as a truly object-oriented programming language.
- **Smalltalk** is considered the first truly object-oriented programming language.
- Popular object-oriented languages are [Java](#), [C#](#), [PHP](#), [Python](#), [C++](#), [Kotlin](#) etc..

# Abstraction

- Providing the essential features without its inner details is called abstraction (or) hiding internal implementation is called Abstraction.
- We can enhance the internal implementation without effecting outside world.
- Abstraction provides security.
- A class contains lot of data and the user does not need the entire data.
- The advantage of abstraction is that every user will get his own view of the data according to his requirements and will not get confused with unnecessary data.

# Encapsulation

- Wrapping up of data (variables) and methods into single unit is called Encapsulation.
- Class is an example for encapsulation.
- Encapsulation can be described as a protective barrier that prevents the code and data being randomly accessed by other code defined outside the class.

# Inheritance

- Acquiring the properties from one class to another class is called inheritance (or) producing new class from already existing class is called inheritance.
- Reusability of code is main advantage of inheritance.
- In Java inheritance is achieved by using extends keyword.
- The properties with access specifier private cannot be inherited.

# Polymorphism

- The word polymorphism came from two Greek words ‘poly’ means ‘many’ and ‘morphos’ means ‘forms’.
- Thus, polymorphism represents the ability to assume several different forms.
- The ability to define more than one function with the same name is called Polymorphism



# OOPs Terms

- In object-oriented programming, a class is a programming language construct that is used as a blueprint to create objects.
- This blueprint includes attributes and methods that the created objects all share.
- Usually, a class represents a person, place, or thing - it is an abstraction of a concept within a computer program.
- Fundamentally, it encapsulates the state and behavior of that which it conceptually represents.
- It encapsulates state through data placeholders called member variables; it encapsulates behavior through reusable code called methods

# History of Java

- Initially Java was developed by James Gosling at Sun Microsystems (which is now a subsidiary of Oracle Corporation) and released first official version in 1995.
- He is also known as **Father of Java**
- The history of Java starts with the **Green Team**.
- **James Gosling** , **Mike Sheridan**, and **Patrick Naughton** initiated the Java language project in June 1991.

- Initially it was designed for small, embedded systems in electronic appliances like set-top boxes, TV's
- Firstly, it was called "**Greentalk**" by James Gosling, and the file extension was .gt.
- After that, it was called **Oak** and was developed as a part of the Green project.

# Why Java named "Oak"?

- Oak is a name of Tree (It's a symbol of strength)
- And it is national tree of many countries like the U.S.A., France, Germany, Romania, etc.

# Oak Tree



- Because of some trademark issues Oak was renamed as "**Java**" in 1995
- Java is **an island** of Indonesia where the **first coffee** was produced (called java coffee). It is a kind of espresso bean.
- Note : -JAVA is not an acronym
  - It is not extension of C++

# Symbol of JAVA





# Java Buzzwords

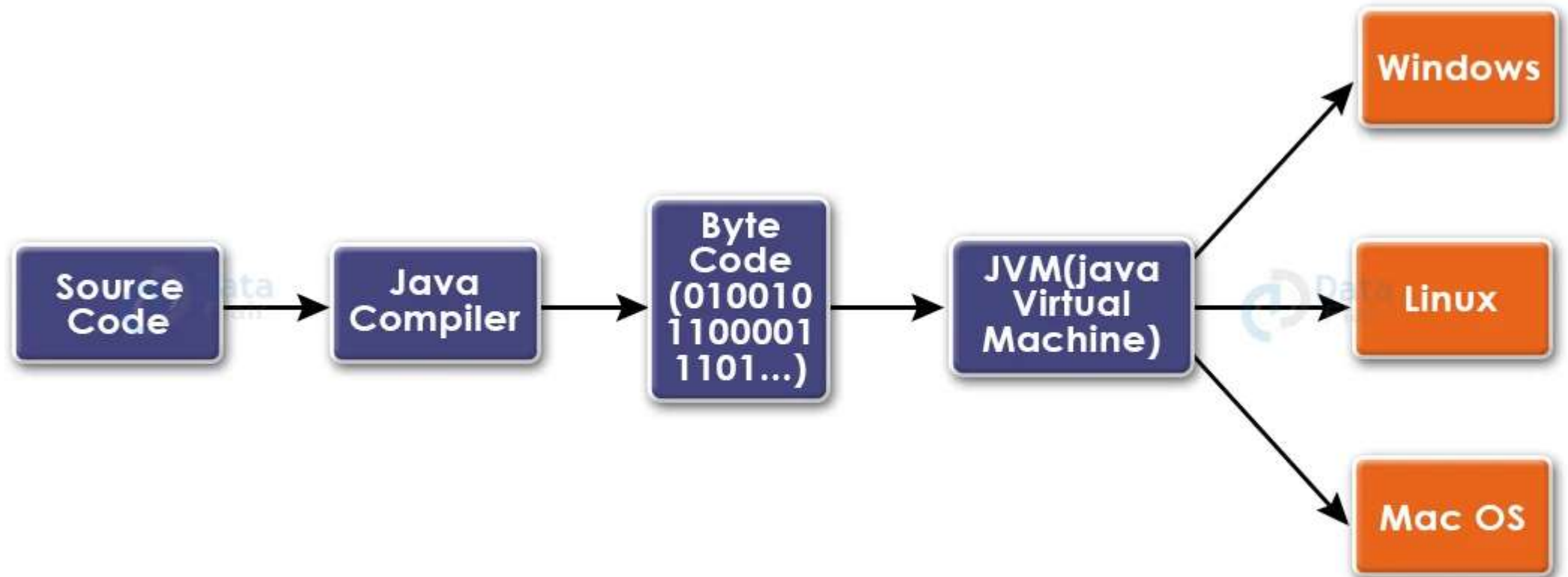
- Simple
- Platform independent
- Secure
- Portable
- Object-oriented
- Robust
- Multithreaded
- Architecture-neutral
- Interpreted
- High performance
- Distributed
- Dynamic

# Simple

- Java is a small and simple language.
- Java does not use pointers, pre-processor header files, goto statement and many other.
- It also eliminates operator overloading and multiple inheritance.
- Java inherits the C/C++ syntax and many of the object oriented features of C++.

# Platform Independent

- Compile the Java program on one OS (operating system) that compiled file can execute in any OS(operating system) is called Platform Independent Nature.
- The java is platform independent language.
- The java applications allows its applications compilation one operating system that compiled (.class) files can be executed in any operating system



**Java is platform-independent**

# Secure

- Security becomes an important issue for a language that is used for programming on Internet.
- Every time when you download a “normal program”, here is a risk of viral infection.
- When we use a java compatible web browser, we can safely download Java applets without fear of viral infection.
- Java achieves this protection by confining a Java program to the Java execution environment and not allowing it access to other parts of the computer.

# Portable

- Java programs can be easily moved from one computer system to another, anywhere and anytime.
- This is the reason why Java has become a popular language for programming on Internet.

# Object-Oriented

- Java is a true object oriented language.
- Almost everything in java is an object.
- All program code and data reside within objects and classes Java comes with an extensive set of classes, arranged in packages, that we can use in our programs by inheritance.
- The object model in java is simple and easy to extend.

# ROBUST

- Any technology if it is good at two main areas it is said to be ROBUST
  - ❖ Exception Handling
  - ❖ Memory Allocation
- JAVA is having very good predefined Exception Handling mechanism whenever we are getting exception we are having meaning full information.
- JAVA is having very good memory management system that is Dynamic Memory (at runtime the memory is allocated) Allocation which allocates and deallocates memory for objects at runtime



# Multithreaded

- Multithreaded means handling multiple tasks simultaneously.
- This means that we need not wait for the application to finish one task before beginning another.
- To accomplish this, java supports multithreaded programming which allows to write programs that do many things simultaneously.

# Architecture-Neutral

- A central issue for the Java designers was that of code longevity and portability.
- One of the main problems facing programmers is that no guarantee exists that if you write a program today, it will run tomorrow—even on the same machine.
- Operating system upgrades, processor upgrades, and changes in core system resources can all combine to make a program malfunction.

**Ex:**

- In C programming, **int** data type occupies 2 bytes of memory for 32-bit architecture and 4 bytes of memory for 64-bit architecture.
- But in java, int occupies 4 bytes of memory for both 32 and 64 bit architectures.
- Java Virtual Machine solves this problem. The goal is “**write once; run anywhere, any time, forever.**”

# Interpreted and High Performance

- Java performance is impressive for an interpreted language, mainly due to the use of byte code.
- This code can be interpreted on any system that provides a JVM.
- Java was designed to perform well on very low power CPUs.

# Distributed

- Java is designed for the distributed environment of the Internet, because it handles TCP/IP protocols.
- In fact, accessing a resource using a URL is not much different from accessing a file.
- The original version of Java (Oak) included features for intra address-space messaging.

- This allowed objects on two different computers to execute procedures remotely.
- Java revived these interfaces in a package called Remote Method Invocation (RMI).
- This feature brings an unparalleled level of abstraction to client/server programming.

# Dynamic

- Java programs carry with them substantial amounts of run-time type information that is used to verify and resolve accesses to objects at run time.
- Java is dynamic in nature because of its support for dynamic loading of classes at runtime, allowing programs to dynamically link in new class definitions.
- Allows for the dynamic resolution of method calls

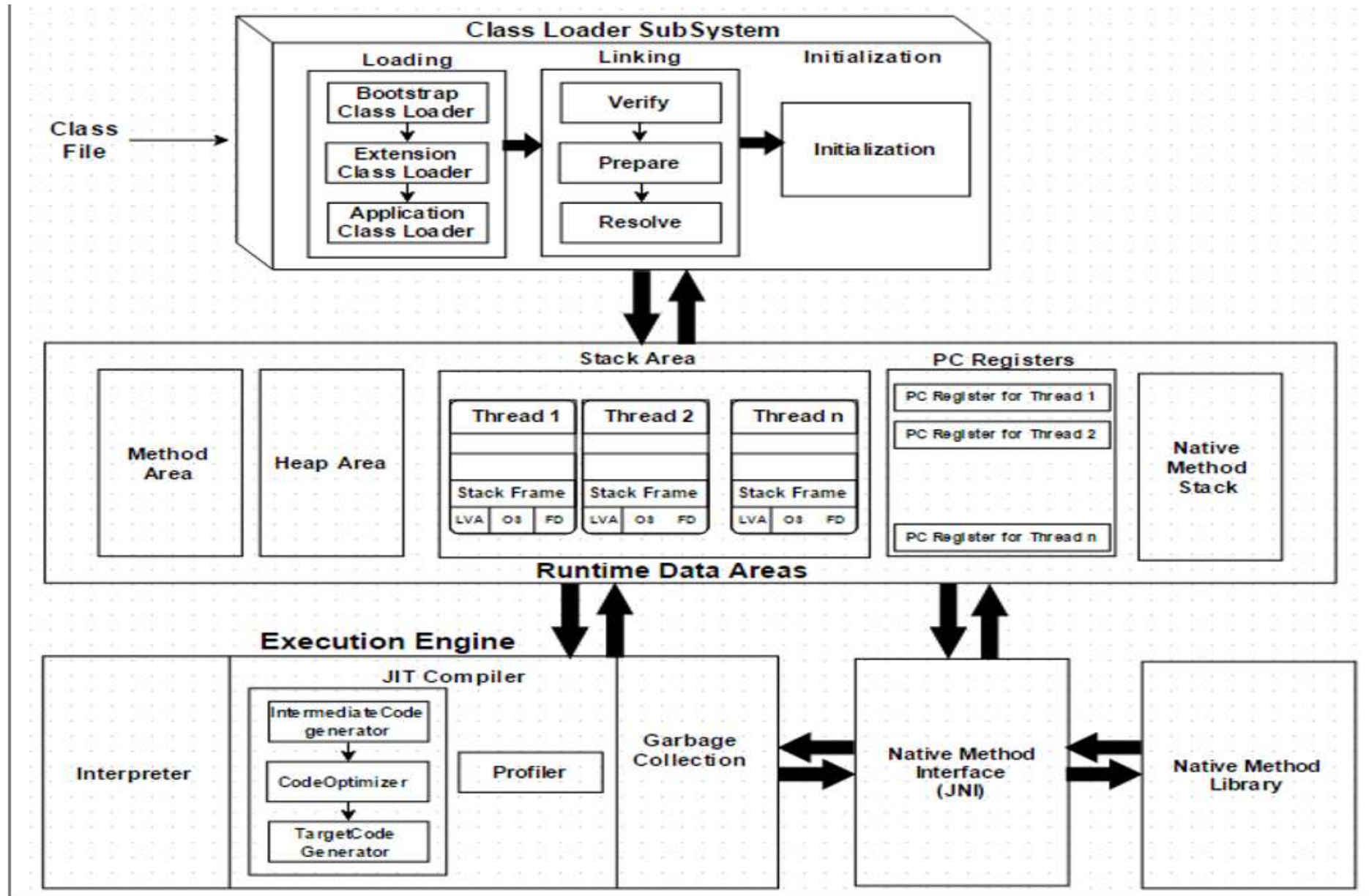
# JVM Architecture



# What Is the JVM?

- A **Virtual Machine** is a software implementation of a physical machine.
- Java was developed with the concept of **WORA** (*Write Once Run Anywhere*), which runs on a **VM**.
- The **compiler** compiles the Java file into a Java **.class** file, then that .class file is input into the JVM, which loads and executes the class file.

# JVM Architecture



As shown in the above architecture diagram, the JVM is divided into three main subsystems:

- ❖ Class Loader Subsystem
- ❖ Runtime Data Area
- ❖ Execution Engine

# 1. ClassLoader Subsystem

- Java's dynamic class loading functionality is handled by the ClassLoader subsystem.
- It loads, links, and initializes the class file when it refers to a class for the first time **at runtime**, not compile time.

## a) Loading

There are three class loaders in JVM i.e, BootStrap ClassLoader, Extension ClassLoader, and Application ClassLoader

- **BootStrap ClassLoader** : Responsible for loading classes from the bootstrap classpath, nothing but **rt.jar**. Highest priority will be given to this loader.
- **Extension ClassLoader** : Responsible for loading classes which are inside the ext folder (jre\lib).
- **Application ClassLoader** : Responsible for loading Application Level Classpath, path mentioned Environment Variable, etc.

## b) Linking

- **Verify** : Bytecode verifier will verify whether the generated bytecode is proper or not if verification fails we will get the verification error.
- **Prepare** : For all static variables memory will be allocated and assigned with default values.
- **Resolve** : All symbolic memory references are replaced with the original references from Method Area.

## c) Initialization

- This is the final phase of ClassLoading.
- Here, all static variables will be assigned with the original values.
- And the static block will be executed.

## 2. Runtime Data Area

- The Runtime Data Area is divided into five major components
- **Method Area** : All the class-level data will be stored here, including static variables. There is only one method area per JVM, and it is a shared resource.
- **Heap Area** : All the Objects and their corresponding instance variables and arrays will be stored here. There is also one Heap Area per JVM.

### Note:

Since the Method and Heap areas share memory for multiple threads, the data stored is not thread-safe.



# Stack Area:

- For every thread, a separate runtime stack will be created.
- For every method call, one entry will be made in the stack memory which is called Stack Frame.
- All local variables will be created in the stack memory.
- The stack area is thread-safe since it is not a shared resource.
- The Stack Frame is divided into three sub entities:
  - ❖ **Local Variable Array**
  - ❖ **Operand stack**
  - ❖ **Frame data**

- **PC Registers** : Each thread will have separate PC Registers, to hold the address of current executing instruction once the instruction is executed the PC register will be updated with the next instruction.
- **Native Method stacks** : Native Method Stack holds native method information. For every thread, a separate native method stack will be created.

### 3. Execution Engine

- The bytecode, which is assigned to the **Runtime Data Area**, will be executed by the Execution Engine.
- The Execution Engine reads the bytecode and executes it piece by piece.
- **a) Interpreter** :The interpreter interprets the bytecode faster but executes slowly. The disadvantage of the interpreter is that when one method is called multiple times, every time a new interpretation is required.

- **b) JIT Compiler** :The JIT Compiler neutralizes the disadvantage of the interpreter.
- The Execution Engine will be using the help of the interpreter in converting byte code, but when it finds repeated code it uses the JIT compiler, which compiles the **entire bytecode** and changes it to native code.
- This native code will be used directly for repeated method calls, which **improves the performance** of the system.

# Data types

# What is data type ?

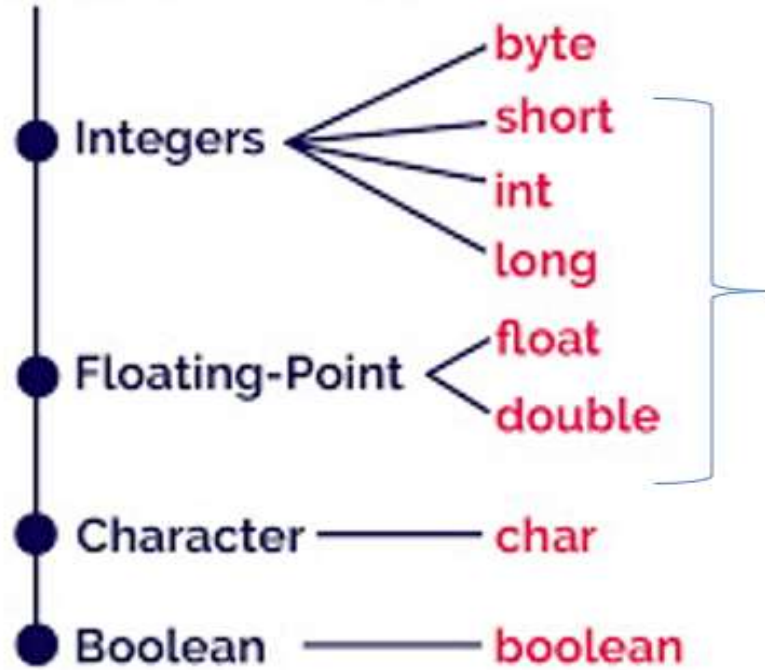
- Data types are used to represent the type of the variable and type of the expression.
- Java is **Strictly typed** / **Strongly typed** / **Statically typed**

# Java is a Strongly Typed

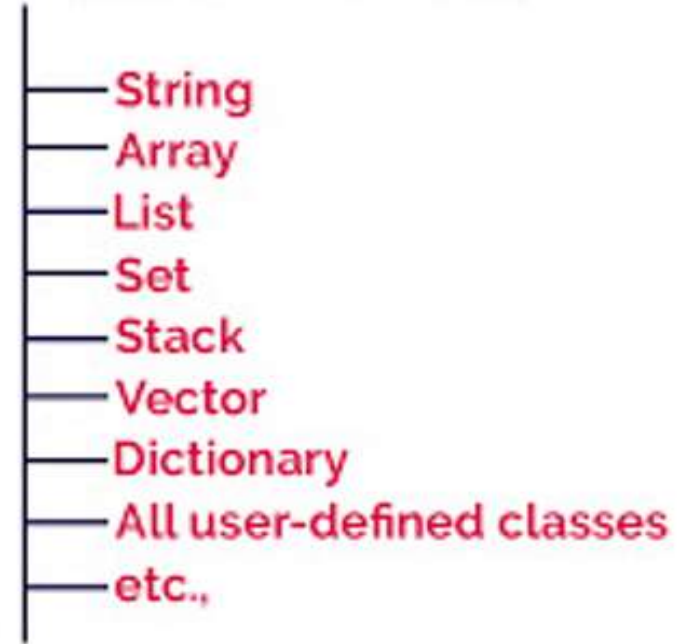
- Every variable has a type
- Every expression is a type
- All assignments are checked for type compatibility at compile time

## Data Types in java

### Primitive Data Types



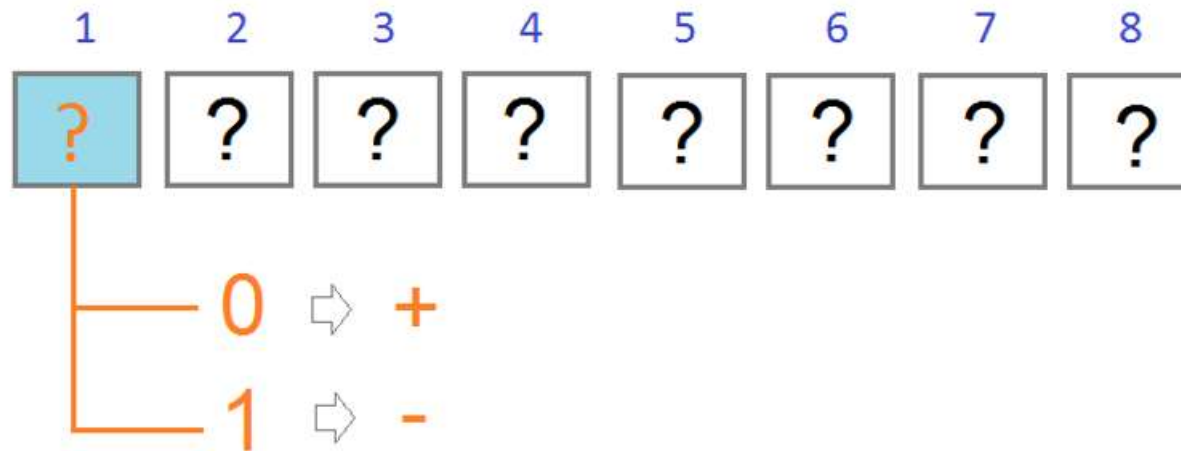
### Non-primitive Data Types





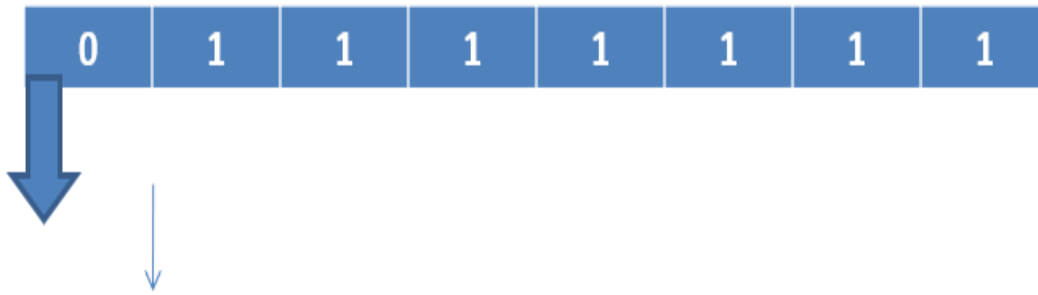
# Byte

- A byte is composed of 8 consecutive **bits** in the memory of computer. Each **bit** is a binary number of 0 or 1.



# Range

➤ Max value=127

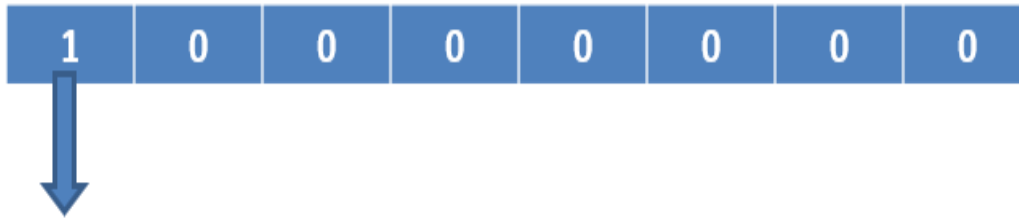


MSB=0----> +ve value

Value->  $1+2+4+8+16+32+64=127$

# Range:

➤ Min value=-128



MSB=1----> -ve value

Value→ All negative values will be calculated in 2's complement form

0000000--->1111111

+1

-----

10000000 ==>-128

# Application area

- Used to store data into Files
- Network

# short data type

- The short data type is a 16-bit signed two's complement integer.
- Rarely used data type.
- **Size:** 2 bytes
- **Range:** -32,768 to 32767.
- **Default value:** 0

# int Data type

- int data type is the preferred data type when we create variables with a numeric value.
- It will take 32 bits or 4bytes of memory.
- **Range:**-2147483648 to 2147483647
- **Default value:** 0

# long Data type

- Used when int is not large enough to hold the value, it has wider range than int data type.
- **Size** :8 bytes or 64 bits
- **Range**:-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
- **Default value**: 0

# float Data type

- A float data type is a **single precision** number format that occupies 4 bytes or 32 bits of memory.
- A float data type in java stores a decimal value with 6-7 total digits of precision.
- Ex: float f=10.25212121212f
- But it will take only **10.252121**.
- **Range:** 3.4e-038 to 3.4e+038
- **Default value: 0.0**



# double Data type

- The double data type is a **double-precision** 64-bit IEEE 754 floating points.
- In Java any floating value by default it's a **double** type
- 12-13 digits precisions it will take.
- EX: double d=10.2521212121213434332222;
- Output: **10.252121212121343**

# char Data type

- The char data type is a single 16-bit Unicode character.
- Ex: `char c='A';`
- Unsigned data type.
- **Range:** 0 to 65535
- **Default value:** `\u0000`

# Why 16 bits for char type?

- In C/C++ uses only ASCII characters and to represent all ASCII characters 8-bits is enough.
- Java uses Unicode system, to represent Unicode system 8 bit is not enough.
- Unicode=ASCII + Other language symbols.

# Boolean Data type

- The Boolean data type has only two possible values: true and false.
- Use this data type for simple flags that track true/false conditions.
- This data type represents one bit of information, but its “size” isn’t something that’s precisely defined.

- Default value is **false**
- **EX:**
- **boolean b=true;**
- **boolean b=0; ➔ Not allowed**

# Identifiers

- An **identifier** is a name used to identify entities like a variable, methods, classes and interfaces etc.

(Or)

- Any name in the java program like variable name, class name, method name, interface name is called identifier.

**Ex:**

class Test	→	Test	identifier
{			
void add()	→	add	identifier
{			
int a=10;	→	a	identifier
int b=20;	→	b	identifier
}			
}			

# Rules to declare identifiers

- Java identifiers should not start with numbers, it may start with alphabet symbol and underscore symbol and dollar symbol.
- An identifier should not contains symbols like + , - , . , @ , # . Only allowed special symbols are \_ and \$
- Duplicate identifiers are not allowed.

**Ex:**

```
class Test
{
    void add()
    {
        int a=10;
        int a=20;
    }
}
```

→ An identifier should not be duplicated.

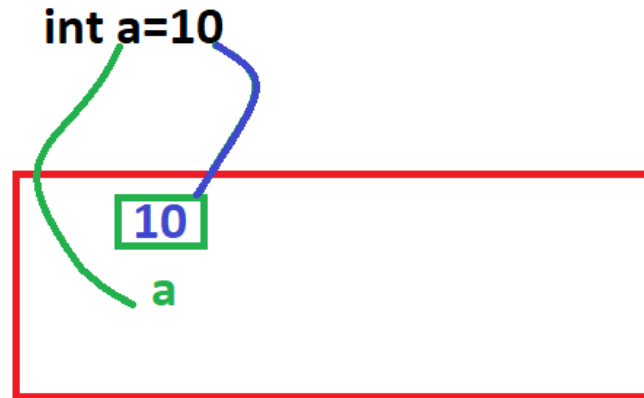
- In the java applications it is possible to declare all the predefined class names and predefined interfaces names as a identifier. But it is not recommended to use.
- Keywords are not allowed to use as an identifier.
- There is no length limit to define an identifier but it recommended to use short and meaning full names.



# Variables

# What is a variable??

- A variable is a name of the memory location. It is the small unit of storage in a program.



- The value stored in a variable can be varied during program execution.

➤ **Syntax:**

datatype variable\_name= value;

**Ex:** int a= 10;

char c='A';

**Note:**

➤ **Java is strongly typed.** So before using any variable in it is mandatory to declare with specific data type

# Naming convention of a variable

➤ As per documentation all user variables are small case.

➤ All constant variables should be in uppercase letter

**Ex: final double PI=3.141592653589793238;**

**final int DATABASE\_VERSION=1;**

➤ **Note:**

Above rules are optional. But it is highly recommended to follow Java coding standards

# Types of Variables

- There are three types of variables in Java:
  - ❖ Local Variables
  - ❖ Instance Variables
  - ❖ Static Variables

# Local variables

- The variables which are declared inside a method or inside a block or inside a constructor are called local variables.

Ex: class Student

```
{
    public void studentInfo()
    {
        // local variables
        String name="Ramu";
        int age = 26;
        System.out.println("Student name : " + name);
    }
}
```

- These variables are created when the block is entered or the function is called and destroyed after exiting from the block or when the call returns from the function.
- Hence The scope of local variables are inside a **method** or inside a **constructor** or inside a **block**.
- JVM wont provide any initial values for local variables. So initialization of Local Variable is Mandatory. If not it will generate compile time error.

- Access modifiers (public , private , protected , default) are not allowed for local variables.
- Local variables will be stored in **stack** area.



### Example:

```
public class Test
{
    public static void main(String[] args)
    {
        Test t=new Test();
        t.m1();
    }
    public void m1()
    {
        //Declaring variable inside method..
        int i=10;
        System.out.println(i);
    }
    public Test()
    {
        //Declaring variable inside constructor..
        int j=20;
        System.out.println(j);
    }
    {
        //Declaring variable inside block..
        int k=30;
        System.out.println(k);
    }
}
```

# Instance Variables

- Instance variables are declared inside class and outside of methods or constructor or block.

Ex: class A

```
{  
    int a;    //instance variable  
    public static void main(String[] args)  
    {  
    }  
}
```

- Instance variables are created when an object of the class is created and destroyed when the object is destroyed.
- We are able to access instance variables only inside the class any number of methods.

- Initialization of Instance Variable is **not mandatory** JVM automatically allocates default values.
- Unlike static variable, instance variables have their own separate copy i.e, if any changes done in instance variable that will not reflect on other objects.
- Instance variables are not allowed inside static area directly. But using object it will allow.

# Static variables

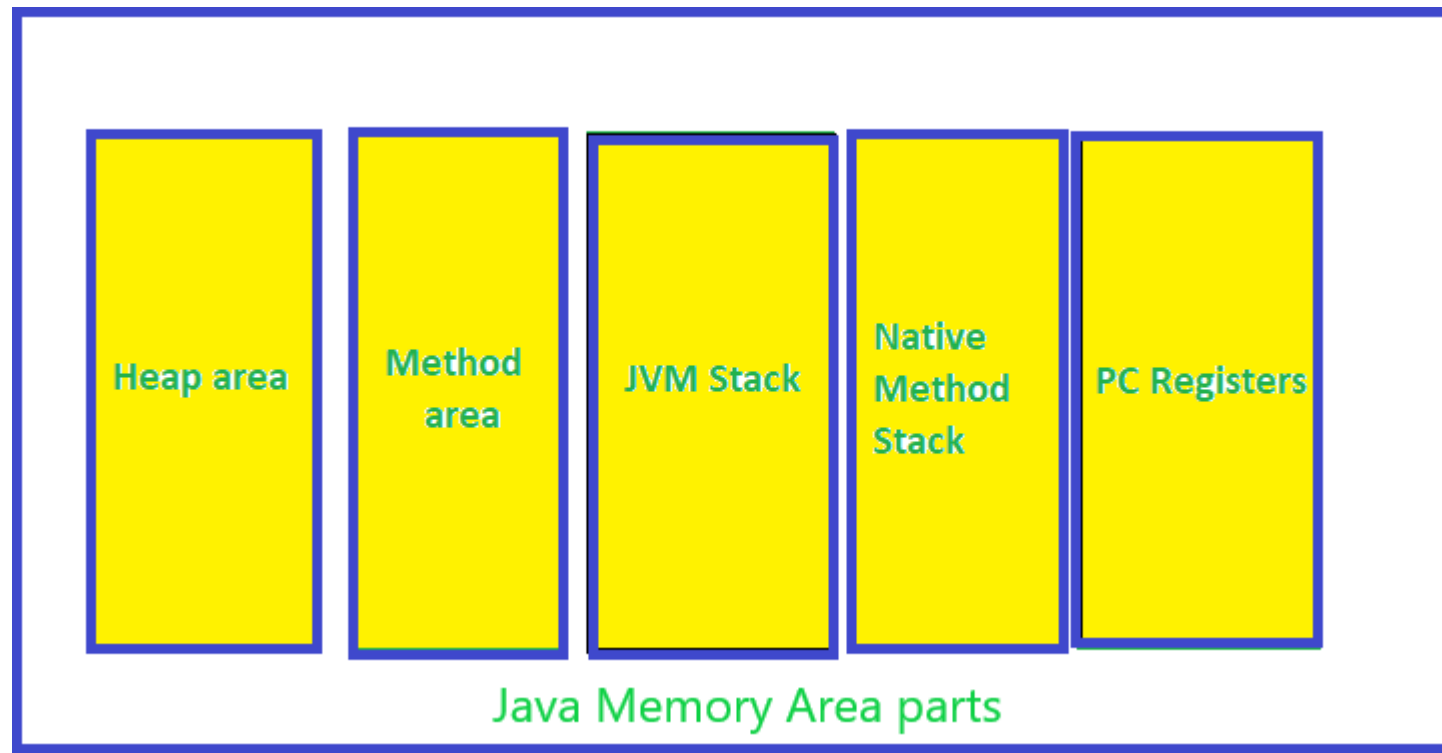
- If any variable declared inside class and outside methods with “**static**” keyword is called static variable.

Ex: 

```
public class Test
{
    static int a=10;
    public static void main(String[] args)
    {
    }
}
```

- **static** variables are also called as **class variable** because they are associated with the class and common for all the instances of the class.

- **static** variables are created at class loading time and destroyed at class unloading.
- All static variables will be stored within **method** area.



- Static variables are can be accessed from any area(instance or static) directly.
- Static variables can be accessed by using objects and using class name.
- Initialization of static variables is not mandatory

# Arrays

# What is an Array?

- An array is a container object that holds a fixed number of values of a single type.
- In Java, all arrays are dynamically allocated. (discussed below)
- Arrays are stored in contiguous memory [consecutive memory locations].
- Since arrays are objects in Java, we can find their length using the object property length. This is different from C/C++, where we find length using sizeof.



- The variables in the array are ordered, and each has an index beginning with 0.
- No –ve index concept in Java.
- Java array can also be used as a static field, a local variable, or a method parameter.
- The size of an array must be specified by int ,short , byte, and char value and not long.
- Maximum size of an Array is 2147483647.
- The direct superclass of an array type is Object.
- The size of the array cannot be altered(once initialized).

40	55	63	17	22	68	89	97	89
0	1	2	3	4	5	6	7	8

**<- Array Indices**

**Array Length = 9**

**First Index = 0**

**Last Index = 8**

# How to create an Array:

## Syntax:

Type[] arrayName;

- Data type can be primitive types like int, char, float,..(or) class type.
- Array name can be any valid identifier.

Ex:   int a[];

String s[];

Student student[];

# How to initialize an Array:

## Approach 1:-

### ➤Syntax:

Type variable\_name[]={element1, elem2, element3, element4,..etc};

Ex: int a[]={ 10,20,30,40};

## Approach 2:-

- In this approach we can create an object to the Array.
- Initialization is Mandatory.
- Allowed data types to specify size of an Array are byte, short, char, int only.
- **Syntax:**

Type Variable\_name[]=new Type[Initialization];

Ex: `int[] a=new int[4];`

`a[0]=10;`

`a[1]=20;`

`a[2]=30;`

`a[3]=40;`

# Traversing array using for loop

**Ex:**

```
public class Array
{
    public static void main(String[] args) {
        int a[]={10,20,30,40};
        for (int i=0;i<a.length;i++)
        {
            System.out.println(a[i]);
        }
    }
}
```

Output:

10  
20  
30  
40

# Traversing array using for each loop

Ex:

```
public class Array
{
    public static void main(String[] args) {
        int a[]={10,20,30,40};
        for (int i:a)
        {
            System.out.println(i);
        }
    }
}
```

Output:

```
10
20
30
40
```





# Operators

➤ **Operator** in java is a symbol that is used to perform operations.

Example: +, -, \*, / etc.

➤ There are many types of operators in java which are given below:

- ❖ Unary Operators

- ❖ Arithmetic Operators

- ❖ Shift Operators

- ❖ Bitwise Operators

- ❖ Logical Operators

- ❖ Relational Operators

- ❖ Assignment Operators.

- ❖ Ternary Operators

Operator Type	Category	Precedence
Unary	postfix	<i>expr++ expr--</i>
	prefix	<i>++expr --expr +expr -expr ~ !</i>
Arithmetic	multiplicative	<i>* / %</i>
	additive	<i>+ -</i>
Shift	shift	<i>&lt;&lt; &gt;&gt; &gt;&gt;&gt;</i>
Relational	comparison	<i>&lt; &gt; &lt;= &gt;= instanceof</i>
	equality	<i>== !=</i>
Bitwise	bitwise AND	<i>&amp;</i>
	bitwise exclusive OR	<i>^</i>
	bitwise inclusive OR	<i> </i>
Logical	logical AND	<i>&amp;&amp;</i>
	logical OR	<i>  </i>
Ternary	ternary	<i>? :</i>
Assignment	assignment	<i>= += -= *= /= %= &amp;= ^=  = &lt;&lt;= &gt;&gt;= &gt;&gt;&gt;=</i>

# Operators and Operands

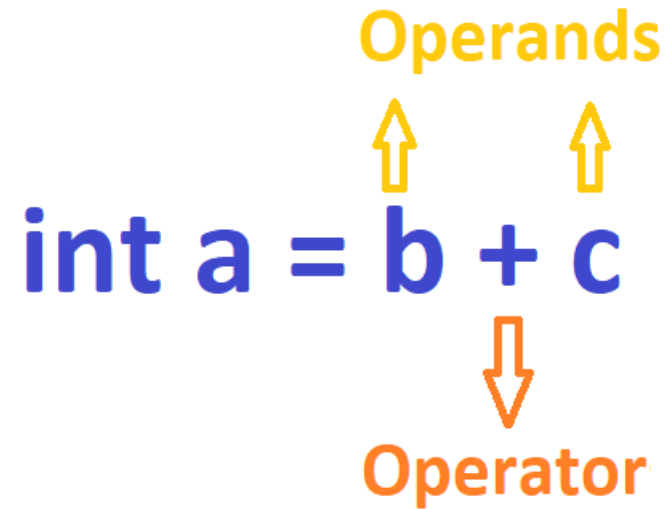
Operands

↑      ↑

**int a = b + c**

↓

Operator



# Unary operators

- The unary operators require only one operand;
- They perform various operations such as incrementing/decrementing a value by one, negating an expression, or inverting the value of a boolean.

Ex: int a=10;

a++;

++a;

Ex: int a=2;

~a;

# Pre Increment and Post Increment

## Pre Increment:

- When placed before the variable name, the operand's value is incremented instantly.

Ex: int a=10;

**++a; //11**

# Post Increment:

- The value of the operand is incremented but the previous value is retained temporarily until the execution of this statement and it gets updated before the execution of the next statement.

Ex: `int a=10;`

`a++; // 10`

**Ex:**

```
public class Ex2
{
    public static void main(String[] args)
    {
        int x=10;
        int y=20;
        int z=++x+y++;
        int a=++x+x++;
        int b=++y+x++;
        System.out.println(z);
        System.out.println(a);
        System.out.println(b);
    }
}
```

Output :

31

24

35



# Arithmetic Operators

- The Java programming language provides operators that perform addition, subtraction, multiplication, and division.
- The only symbol that might look new to you is "%", which divides one operand by another and returns the remainder as its result.

Operator	Use	Description
+	op1 + op2	Adds op1 and op2; also used to concatenate strings
-	op1 - op2	Subtracts op2 from op1
*	op1 * op2	Multiplies op1 by op2
/	op1 / op2	Divides op1 by op2
%	op1 % op2	Computes the remainder of dividing op1 by op2

# Shift Operators

The Java programming language also provides operators that perform bitwise and bit shift operations on integral types.

- ❖ Right shift(>>)

- ❖ Left shift(<<)

## Right shift(>>):

- Shifts the bits of the number to the right and fills 0 on voids left as a result.
- The leftmost bit depends on the sign of initial number. Similar effect as of dividing the number with some power of two.

**Ex:** `int a=10;`

`System.out.println(10>>1);`

Output: **5**

```
➤ int a=10;
```



# Left shift(<<):

Shifts the bits of the number to the left and fills 0 on voids left as a result.

**Ex:** int a=10;

System.out.println(a<<1);

Output:20

int a=10;



# Bitwise Operators:

Operator	Use	Operation
&	op1 & op2	Bitwise AND if both operands are numbers; conditional AND if both operands are boolean
	op1   op2	Bitwise OR if both operands are numbers; conditional OR if both operands are boolean
^	op1 ^ op2	Bitwise exclusive OR (XOR)
~	~op2	Bitwise complement

- When its operands are numbers, the **&** operation performs the bitwise **AND** function on each parallel pair of bits in each operand.
- The **AND** function sets the resulting bit to 1 if the corresponding bit in both operands is 1, as shown in the following table.

Bit in op1	Corresponding Bit in op2	Result=op1&op2
0	0	0
0	1	0
1	0	0
1	1	1



➤ **EX:**

1101 //13

&

1100 //12

-----

1100 //12

➤ *Inclusive or* means that if either of the two bits is 1, the result is 1. The following table shows the results of an *inclusive or* operation.

Bit in op1	Corresponding Bit in op2	Result=op1   op2
0	0	0
0	1	1
1	0	1
1	1	1

- *Exclusive or* means that if the two operand bits are different the result is 1; otherwise the result is 0.
- The following table shows the results of an *exclusive or* operation.

Bit in op1	Corresponding Bit in op2	Result=op1^op2
0	0	0
0	1	1
1	0	1
1	1	0

# Logical Operators

- Logical operators are used to check whether an expression is true or false. They are used in decision making.

Operator	Example	Meaning
&& (Logical AND)	expression1 && expression2	true only if both expression1 and expression2 are true
(Logical OR)	expression1    expression2	true if either expression1 or expression2 is true
! (Logical NOT)	!expression	true if expression is false and vice versa

**Ex:**

```
public class LogicalOpr {  
    public static void main(String[] args) {  
  
        // && operator  
        System.out.println((5 > 3) && (8 > 5)); // true  
        System.out.println((5 > 3) && (8 < 5)); // false  
  
        // || operator  
        System.out.println((5 < 3) || (8 > 5)); // true  
        System.out.println((5 > 3) || (8 < 5)); // true  
        System.out.println((5 < 3) || (8 < 5)); // false  
  
        // ! operator  
        System.out.println(!(5 == 3)); // true  
        System.out.println(!(5 > 3)); // false  
    }  
}
```

# Relational Operators

➤ Relational operators are used to check the relationship between two operands.

Operator	Description	Example
==	Is Equal To	3 == 5 returns <b>false</b>
!=	Not Equal To	3 != 5 returns <b>true</b>
>	Greater Than	3 > 5 returns <b>false</b>
<	Less Than	3 < 5 returns <b>true</b>
>=	Greater Than or Equal To	3 >= 5 returns <b>false</b>
<=	Less Than or Equal To	3 <= 5 returns <b>true</b>

**Ex:**

```
public class RelationalOpr {  
  
    public static void main(String[] args) {  
        // create variables  
        int a = 7, b = 11;  
        System.out.println("a is " + a + " and b is " + b);  
  
        // == operator  
        System.out.println(a == b);    // false  
  
        // != operator  
        System.out.println(a != b);    // true  
  
        // > operator  
        System.out.println(a > b);    // false  
  
        // < operator  
        System.out.println(a < b);    // true  
  
        // >= operator  
        System.out.println(a >= b);    // false  
  
        // <= operator  
        System.out.println(a <= b);    // true  
    }  
}
```

# Assignment Operators

- Assignment operators are used in Java to assign values to variables.

Ex:

```
int age;
```

```
age = 5;
```

- Here, = is the assignment operator.
- It assigns the value on its right to the variable on its left. That is, 5 is assigned to the variable age.



Operator	Example	Equivalent to
=	a = b;	a = b;
+=	a += b;	a = a + b;
-=	a -= b;	a = a - b;
*=	a *= b;	a = a * b;
/=	a /= b;	a = a / b;
%=	a %= b;	a = a % b;

# Ternary Operator

- Java ternary operator is the only conditional operator that takes three operands.
- It's a one-liner replacement for the if-then-else statement and is used a lot in Java programming.
- We can use the ternary operator in place of if-else conditions or even switch conditions using nested ternary operators.

## Syntax:

variable = Condition ? Expression2: Expression3



**Ex:**

```
import java.util.Scanner;

public class Max
{
    public static void main(String[] args) {
        System.out.println("Enter num1,num2");
        Scanner scr=new Scanner(System.in);
        int num1=scr.nextInt();
        int num2=scr.nextInt();
        int max=num1>num2?num1:num2;
        System.out.println("Max number:"+max);
    }
}
```

**Output:**

```
Enter num1,num2
200
300
Max number:300
```

# Control statements

