1. JDK (Java Development Kit) is a Kit that provides the environment to **develop and execute(run)** the Java program. JDK is a kit(or package) that includes two things

- Development Tools(to provide an environment to develop your java programs)
- JRE (to execute your java program).

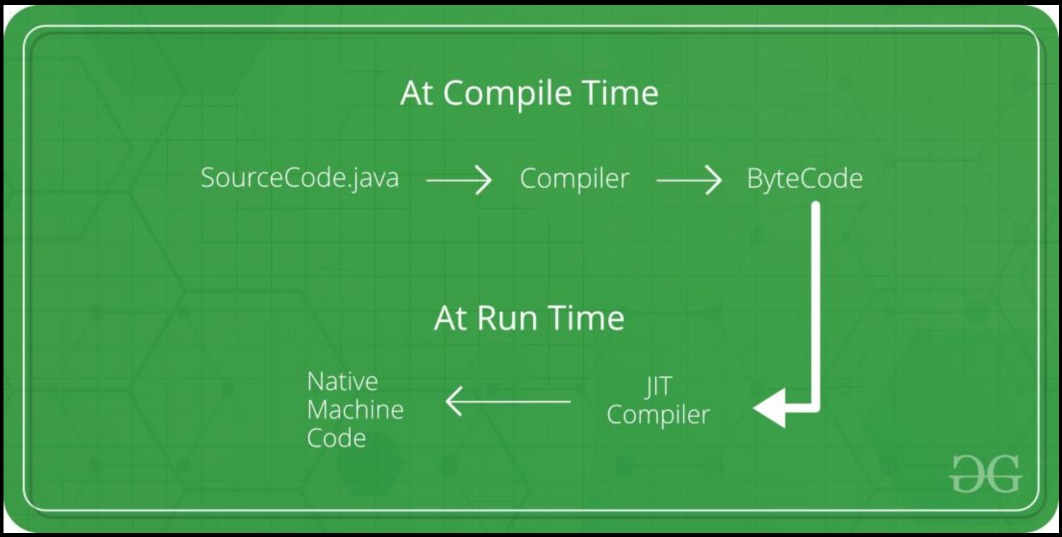
2. JRE (Java Runtime Environment) is an installation package that provides an environment to **only run(not develop)** the java program(or application)onto your machine. JRE is only used by those who only want to run Java programs that are end-users of your system.

3. JVM (Java Virtual Machine) is a very important part of both JDK and JRE because it is contained or inbuilt in both. Whatever Java program you run using JRE or JDK goes into JVM and JVM is responsible for executing the java program line by line, hence it is also known as an ***interpreter*****.**

Now let us discuss the components of JRE in order to understand its importance of it and perceive how it actually works. For this let us discuss components.

The components of JRE are as follows:

1. **Deployment technologies**, including deployment, Java Web Start, and Java Plug-in.
2. **User interface toolkits**, including *Abstract Window Toolkit (AWT), Swing, Java 2D, Accessibility, Image I/O, Print Service, Sound, drag*, *and drop (DnD)*, and *input methods*.
3. **Integration libraries**, including *Interface Definition Language (IDL), Java Database Connectivity (JDBC), Java Naming and Directory Interface (JNDI), Remote Method Invocation (RMI), Remote Method Invocation Over Internet Inter-Orb Protocol (RMI-IIOP)*, and *scripting*.
4. **Other base libraries**, including *international support, input/output (I/O), extension mechanism, Beans, Java Management Extensions (JMX), Java Native Interface (JNI), Math, Networking, Override Mechanism, Security, Serialization*, and *Java for XML Processing (XML JAXP)*.
5. **Lang and util base libraries**, including *lang and util, management, versioning, zip, instrument, reflection, Collections, Concurrency Utilities, Java Archive (JAR), Logging, Preferences API, Ref Objects*, and *Regular Expressions*.
6. **Java Virtual Machine (JVM)**, including *Java HotSpot Client* and *Server Virtual Machines*.



Java bytecode enables cross-platform execution, but its interpretation can impact performance. JIT compilers enhance speed by translating bytecode into native machine code during runtime, reducing interpretation overhead. This optimization boosts execution speed, especially for frequently executed methods.

Working on JIT Compiler

Java follows an object-oriented approach, as a result, it consists of classes. These constitute bytecode that is platform neutral and are executed by the JVM across diversified architectures.

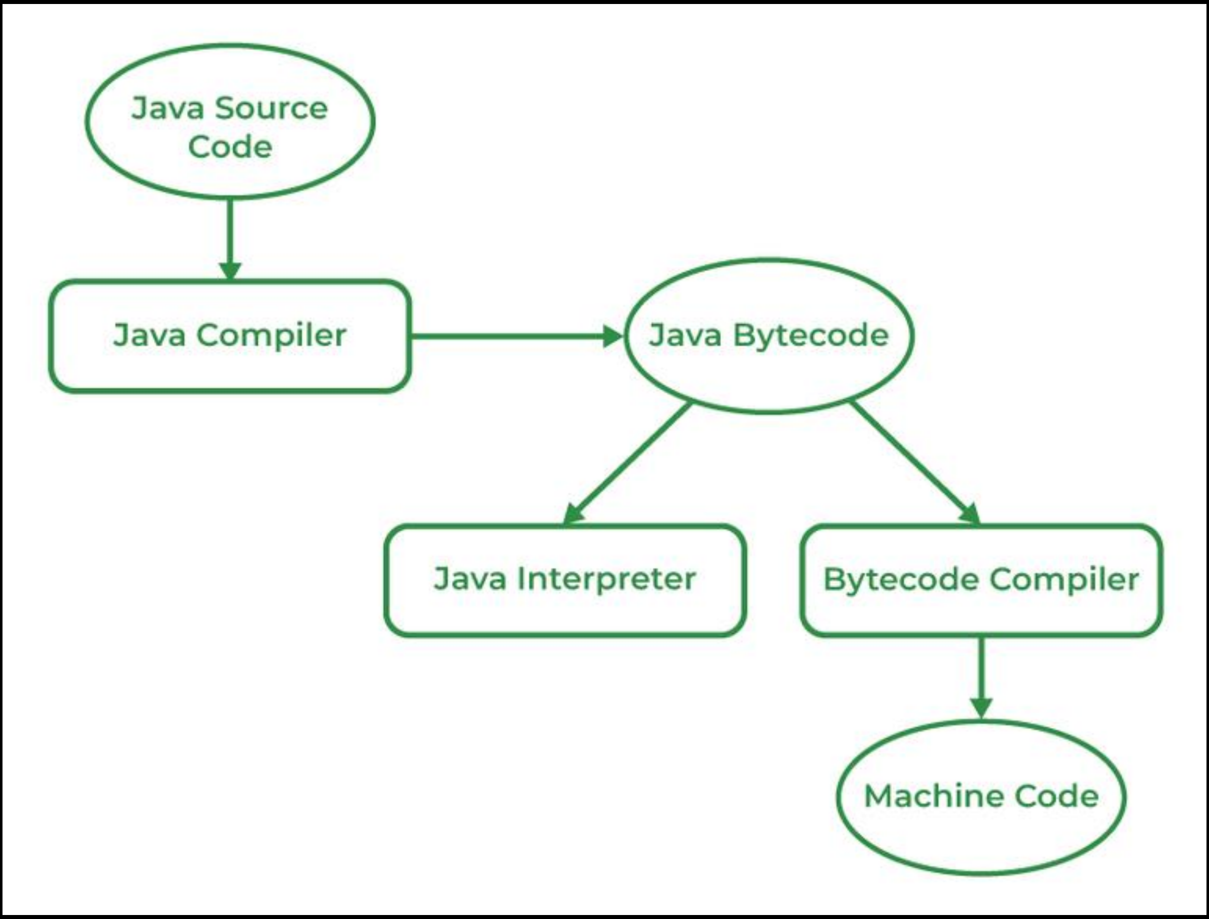
- At run time, the JVM loads the class files, the semantics of each are determined, and appropriate computations are performed. The additional processor and memory usage during interpretation make a Java application perform slowly as compared to a native application.
- The JIT compiler aids in improving the performance of Java programs by compiling bytecode into native machine code at run time.
- The JIT compiler is enabled throughout, while it gets activated when a method is invoked. For a compiled method, the JVM directly calls the compiled code, instead of interpreting it. Theoretically speaking, If compiling did not require any processor time or memory usage, the speed of a native compiler and that of a Java compiler would have been the same.
- JIT compilation requires processor time and memory usage. When the java virtual machine first starts up, thousands of methods are invoked. Compiling all these methods can significantly affect startup time, even if the end result is a very good performance optimization.

JVM	**JIT**
JVM stands for Java Virtual Machine.	JIT stands for Just-in-time compilation.
JVM was introduced for managing system memory and providing a transportable execution environment for Java-based applications	JIT was invented to improve the performance of JVM after many years of its initial release.
JVM consists of many other components like stack area, heap area, etc.	JIT is one of the components of JVM.
JVM compiles complete byte code to machine code.	JIT compiles only the reusable byte code to machine code.
JVM provides platform independence.	JIT improves the performance of JVM.

Difference between Byte Code and Machine Code:

S.NO.	Byte Code	Machine Code
01.	Byte Code consisting of binary, hexadecimal, macro instructions like (new, add, swap, etc) and it is not directly understandable by the CPU. It is designed for efficient execution by software such as a virtual machine.intermediate-level	Machine code consisting of binary instructions that are directly understandable by the CPU.
02.	Byte code is considered as the intermediate-level code.	Machine Code is considered as the low-level code.
03.	Byte code is a non-runnable code generated after compilation of source code and it relies on an interpreter to get executed.	Machine code is a set of instructions in machine language or in binary format and it is directly executed by CPU.
04.	Byte code is executed by the virtual machine then the Central Processing Unit.	Machine code is not executed by a virtual machine it is directly executed by CPU.
05.	Byte code is less specific towards machine than the machine code.	Machine code is more specific towards machine than the byte code.
06.	It is platform-independent as it is dependent on the virtual machine and the system having a virtual machine can be executed irrespective of the platform.	It is not platform independent because the object code of one platform can not be run on the different Operating System. Object varies depending upon system architecture and native instructions associated with the machine.
07.	All the source code need not be converted into byte code for execution by CPU. Some source code written by any specific high-level language is converted into byte code then	All the source code must be converted into machine code before it is executed by the CPU.

S.NO.	Byte Code	Machine Code
	byte code to object code for execution by CPU.	



Java is platform-independent due to its use of bytecode, which is generated by the Java compiler (javac). Unlike languages like C or C++, where the compiler produces OS-dependent executable files, Java compiler generates bytecode (.class files). This bytecode is not executable directly by the machine's CPU but requires an interpreter known as the Java Virtual Machine (JVM).

The JVM, while being platform-dependent, enables Java's platform independence. Different operating systems require different JVM implementations. When a Java program is executed, the JVM interprets the bytecode and translates it into machine code that is understandable by the underlying operating system. This allows Java programs to run on any platform as long as a compatible JVM is available for that platform. In essence, Java's platform independence is achieved through the combination of bytecode and the platform-dependent JVM.

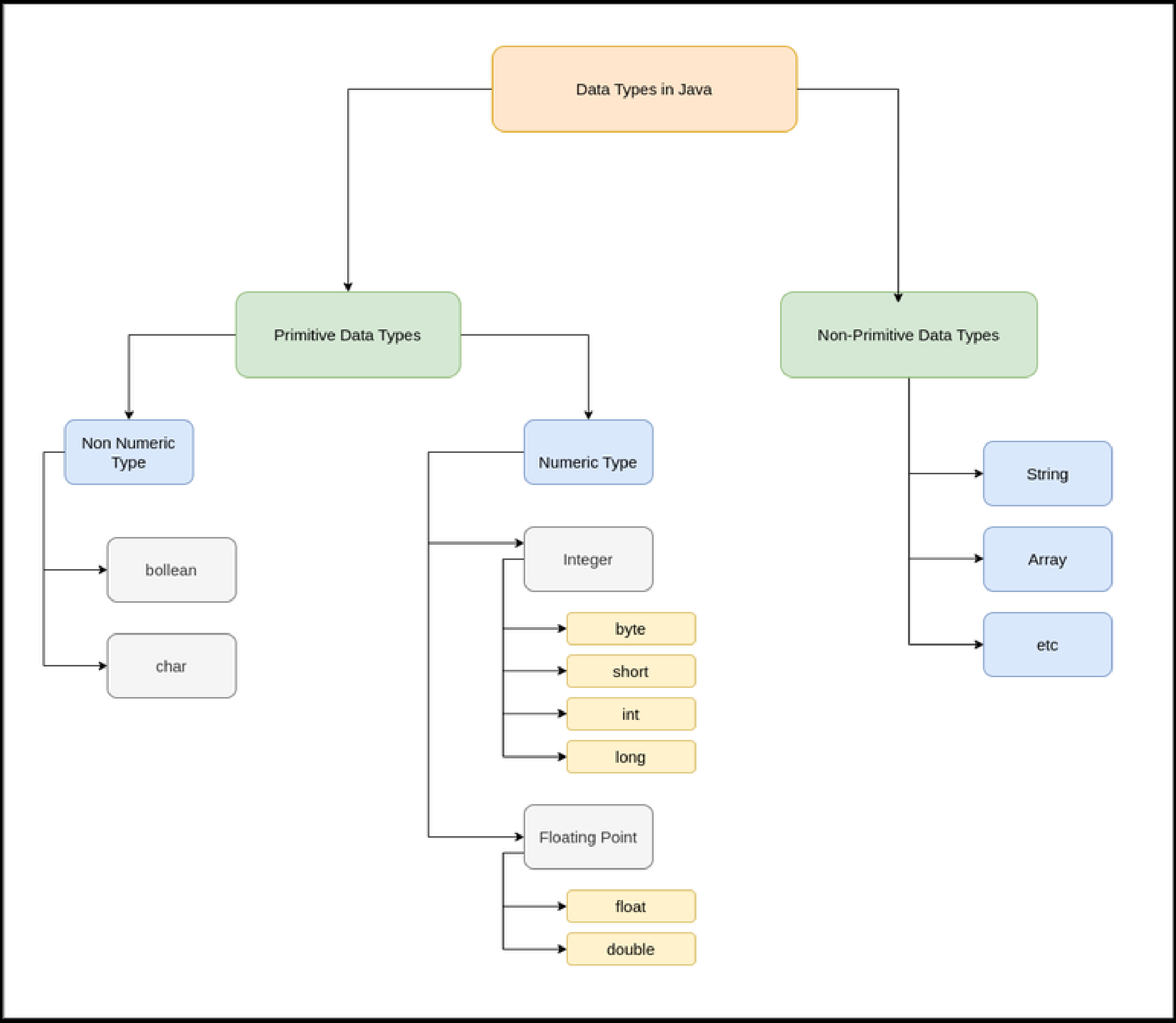
1. **Class:** A blueprint defining properties and methods shared by its instances.
    - Example: "Human" class defines properties and behaviors common to humans.
2. **Object:** An instance of a class with its own state and behavior.
    - Example: Objects like Dog, Cat, Monkey are instances of the "Animal" class.
3. **Method:** Defines the behavior of an object.
    - Example: "run()" method defines the behavior of an animal running.
4. **Instance variables:** Properties unique to each object, defining its state.
    - Example: Instance variables store data specific to each object, like fuel level in a car.

Syntax:

1. **Comments:** Three types: Single-line, multi-line, and documentation comments.
2. **Source File Name:** Must match the public class name with a .java extension.
3. **Case Sensitivity:** Java is case-sensitive.
4. **Class Names:** Start with uppercase, subsequent words start with uppercase, underscores and numbers allowed but discouraged.
5. **Main Method:** Entry point of Java program, starts with "public static void main(String[] args)".
6. **Method Names:** Start with lowercase, subsequent words start with uppercase, underscores and numbers allowed but discouraged.
7. **Identifiers:** Names for variables, methods, classes, etc. Follow specific rules regarding characters and case-sensitivity.
8. **White Spaces:** Blank lines and spaces are ignored by the compiler.
9. **Access Modifiers:** Control the scope of classes and methods.
10. **Java Keywords:** Reserved words used for internal processes or predefined actions.

Access Modifier	Within Class	Within Package	Outside Package by subclass only	Outside Package
Private	Yes	No	No	No
Default	Yes	Yes	No	No

Protected	Yes	Yes	Yes	No
Public	Yes	Yes	Yes	Yes



Type	Description	Default	Size	Example Literals	Range of values
boolean	true or false	false	1 bit	true, false	true, false

Type	**Description**	**Default**	**Size**	**Example Literals**	**Range of values**
byte	twos-complement integer	0	8 bits	(none)	-128 to 127
char	Unicode character	\u0000	16 bits	'a', '\u0041', '\101', '\\', '\', '\n', 'β'	characters representation of ASCII values 0 to 255
short	twos-complement integer	0	16 bits	(none)	-32,768 to 32,767
int	twos-complement intger	0	32 bits	-2,-1,0,1,2	-2,147,483,648 to 2,147,483,647
long	twos-complement integer	0	64 bits	-2L,-1L,0L,1L,2L	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
float	IEEE 754 floating point	0.0	32 bits	1.23e100f , -1.23e-100f , .3f ,3.14F	upto 7 decimal digits
double	IEEE 754 floating point	0.0	64 bits	1.23456e300d , -123456e-300d , 1e1d	upto 16 decimal digits

Non-primitive data types, also known as reference data types, store memory addresses rather than direct variable values. These types include Strings, Classes, Objects, Interfaces, and Arrays.

1. **Strings**: Sequences of characters, stored as arrays but with specialized methods and immutability.
    - Declared using double quotes or the **String** class constructor.
2. **Class**: Blueprint for creating objects, defining properties and methods common to all instances.
    - Includes modifiers, class name, superclass (if any), implemented interfaces, and body.
3. **Object**: Basic unit of Object-Oriented Programming, representing real-life entities.
    - Comprises state, behavior, and identity.
4. **Interface**: Defines methods and variables, with methods being abstract by default.
    - Specifies capabilities a class must implement without defining how.
    - Allows for multiple inheritance of type.
5. **Array**: Group of like-typed variables referred to by a common name.
    - Dynamically allocated objects in Java.
    - Indexed starting from 0, with length accessible through the **length** member.
    - Can be used as static fields, local variables, or method parameters.

Primitive Data Type: In Java, the primitive data types are the predefined data types of Java. They specify the size and type of any standard values. Java has 8 primitive data types namely byte, short, int, long, float, double, char and boolean. When a primitive data type is stored, it is the stack that the values will be assigned. When a variable is copied then another copy of the variable is created and changes made to the copied variable will not reflect changes in the original variable.

Object Data Type: These are also referred to as Non-primitive or Reference Data Type. They are so-called because they refer to any particular object. Unlike the primitive data types, the non-primitive ones are created by the users in Java. Examples include arrays, strings, classes, interfaces etc. When the reference variables will be stored, the variable will be stored in the stack and the original object will be stored in the heap. In Object data type although two copies will be created they both will point to the same variable in the heap, hence changes made to any variable will reflect the change in both the variables. Here is a Java program to demonstrate arrays(an object data type) in Java.

Properties	**Primitive data types**	**Objects**
Origin	Pre-defined data types	User-defined data types
Stored structure	Stored in a stack	Reference variable is stored in stack and the original object is stored in heap

Properties	**Primitive data types**	**Objects**
When copied	Two different variables is created along with different assignment(only values are same)	Two reference variable is created but both are pointing to the same object on the heap
When changes are made in the copied variable	Change does not reflect in the original ones.	Changes reflected in the original ones.
Default value	Primitive datatypes do not have null as default value	The default value for the reference variable is null
Example	byte, short, int, long, float, double, char, boolean	array, string class, interface etc.

abstract	continue	for	protected	transient
Assert	Default	Goto	public	Try
Boolean	Do	If	Static	throws
break	double	implements	strictfp	Package
byte	else	import	super	Private
case	enum	Interface	Short	switch
Catch	Extends	instanceof	return	void
Char	Final	Int	synchronized	volatile
class	finally	long	throw	Date

const	float	Native	This	while

Types of Operators in Java

There are multiple types of operators in Java all are mentioned below:

1. Arithmetic Operators
2. Unary Operators
3. Assignment Operator
4. Relational Operators
5. Logical Operators
6. Ternary Operator
7. Bitwise Operators
8. Shift Operators
9. instance of operator

1. Arithmetic Operators

They are used to perform simple arithmetic operations on primitive data types.

- ***** : Multiplication
- **/** : Division
- **%** : Modulo
- **+** : Addition
- **–** : Subtraction

Example:

Java

```java
// Java Program to implement
// Arithmetic Operators
import java.io.*;

// Drive Class
class GFG {
      // Main Function
    public static void main (String[] args) {

        // Arithmetic operators
        int a = 10;
        int b = 3;

        System.out.println("a + b = " + (a + b));
        System.out.println("a - b = " + (a - b));
        System.out.println("a * b = " + (a * b));
        System.out.println("a / b = " + (a / b));
        System.out.println("a % b = " + (a % b));
    }
}
```

Output

```
a + b = 13

a - b = 7

a * b = 30

a / b = 3

a % b = 1
```

2. Unary Operators

Unary operators need only one operand. They are used to increment, decrement, or negate a value.

- **– : Unary minus**, used for negating the values.
- **+ : Unary plus** indicates the positive value (numbers are positive without this, however). It performs an automatic conversion to int when the type of its operand is the byte, char, or short. This is called unary numeric promotion.
- **++ : Increment operator**, used for incrementing the value by 1. There are two varieties of increment operators.
    - **Post-Increment:** Value is first used for computing the result and then incremented.
    - **Pre-Increment:** Value is incremented first, and then the result is computed.
- **– – : Decrement operator**, used for decrementing the value by 1. There are two varieties of decrement operators.
    - **Post-decrement:** Value is first used for computing the result and then decremented.

- **Pre-Decrement: The value** is decremented first, and then the result is computed.
- **! : Logical not operator**, used for inverting a boolean value.

Example:

Java

```java
// Java Program to implement
// Uniary Operators
import java.io.*;

// Driver Class
class GFG {
      // main function
    public static void main(String[] args)
    {
        // Interger declared
        int a = 10;
        int b = 10;

        // Using unary operators
        System.out.println("Postincrement : " + (a++));
        System.out.println("Preincrement : " + (++a));

        System.out.println("Postdecrement : " + (b--));
        System.out.println("Predecrement : " + (--b));
    }
}
```

Output

```
Postincrement : 10
Preincrement : 12
Postdecrement : 10
Predecrement : 8
```

3. Assignment Operator

'=' Assignment operator is used to assign a value to any variable. It has right-to-left associativity, i.e. value given on the right-hand side of the

operator is assigned to the variable on the left, and therefore right-hand side value must be declared before using it or should be a constant.
The general format of the assignment operator is:

```
variable = value;
```

In many cases, the assignment operator can be combined with other operators to build a shorter version of the statement called a **Compound Statement**. For example, instead of a = a+5, we can write a += 5.

- **+=**, for adding the left operand with the right operand and then assigning it to the variable on the left.
- **-=**, for subtracting the right operand from the left operand and then assigning it to the variable on the left.
- ***=**, for multiplying the left operand with the right operand and then assigning it to the variable on the left.
- **/=**, for dividing the left operand by the right operand and then assigning it to the variable on the left.
- **%=**, for assigning the modulo of the left operand by the right operand and then assigning it to the variable on the left.

Example:

Java

```
// Java Program to implement
// Assignment Operators
import java.io.*;

// Driver Class
class GFG {
    // Main Function
    public static void main(String[] args)
    {

        // Assignment operators
        int f = 7;
        System.out.println("f += 3: " + (f += 3));
        System.out.println("f -= 2: " + (f -= 2));
        System.out.println("f *= 4: " + (f *= 4));
        System.out.println("f /= 3: " + (f /= 3));
        System.out.println("f %= 2: " + (f %= 2));
        System.out.println("f &= 0b1010: " + (f &= 0b1010));
        System.out.println("f |= 0b1100: " + (f |= 0b1100));
        System.out.println("f ^= 0b1010: " + (f ^= 0b1010));
        System.out.println("f <<= 2: " + (f <<= 2));
        System.out.println("f >>= 1: " + (f >>= 1));
        System.out.println("f >>>= 1: " + (f >>>= 1));
    }
}
```

Output

```
f += 3: 10
f -= 2: 8
f *= 4: 32
f /= 3: 10
f %= 2: 0
f &= 0b1010: 0
f |= 0b1100: 12
f ^= 0b1010: 6
f <<= 2: 24
f >>= 1: 12
f >>>= 1: 6
```

4. Relational Operators

These operators are used to check for relations like equality, greater than, and less than. They return boolean results after the comparison and are extensively used in looping statements as well as conditional if-else statements. The general format is,

```
variable relation_operator value
```

Some of the relational operators are-

- **==, Equal to** returns true if the left-hand side is equal to the right-hand side.
- **!=, Not Equal to** returns true if the left-hand side is not equal to the right-hand side.
- **<, less than:** returns true if the left-hand side is less than the right-hand side.
- **<=, less than or equal to** returns true if the left-hand side is less than or equal to the right-hand side.

- **>, Greater than:** returns true if the left-hand side is greater than the right-hand side.
- **>=, Greater than or equal to** returns true if the left-hand side is greater than or equal to the right-hand side.

Example:

Java

```java
// Java Program to implement
// Relational Operators
import java.io.*;

// Driver Class
class GFG {
    // main function
    public static void main(String[] args)
    {
        // Comparison operators
        int a = 10;
        int b = 3;
        int c = 5;

        System.out.println("a > b: " + (a > b));
        System.out.println("a < b: " + (a < b));
        System.out.println("a >= b: " + (a >= b));
        System.out.println("a <= b: " + (a <= b));
        System.out.println("a == c: " + (a == c));
        System.out.println("a != c: " + (a != c));
    }
}
```

Output

```
a > b: true
a < b: false
a >= b: true
a <= b: false
a == c: false
a != c: true
```

5. Logical Operators

These operators are used to perform “logical AND” and “logical OR” operations, i.e., a function similar to AND gate and OR gate in digital electronics. One thing to keep in mind is the second condition is not evaluated if the first one is false, i.e., it has a short-circuiting effect. Used extensively to test for several conditions for making a decision. Java also has “Logical NOT”, which returns true when the condition is false and vice-versa

Conditional operators are:

- **&&, Logical AND:** returns true when both conditions are true.
- **||, Logical OR:** returns true if at least one condition is true.
- **!, Logical NOT:** returns true when a condition is false and vice-versa

Example:

Java

```
// Java Program to implemenet
// Logical operators
import java.io.*;

// Driver Class
class GFG {
      // Main Function
    public static void main (String[] args) {
        // Logical operators
        boolean x = true;
        boolean y = false;

        System.out.println("x && y: " + (x && y));
        System.out.println("x || y: " + (x || y));
        System.out.println("!x: " + (!x));
    }
}
```

Output

```
x && y: false
x || y: true
!x: false
```

6. Ternary operator

The ternary operator is a shorthand version of the if-else statement. It has three operands and hence the name Ternary.

The general format is:

```
condition ? if true : if false
```

The above statement means that if the condition evaluates to true, then execute the statements after the '?' else execute the statements after the ':'.

Example:

Java

```java
// Java program to illustrate
// max of three numbers using
// ternary operator.
public class operators {
    public static void main(String[] args)
    {
        int a = 20, b = 10, c = 30, result;

        // result holds max of three
        // numbers
        result
            = ((a > b) ? (a > c) ? a : c : (b > c) ? b : c);
        System.out.println("Max of three numbers = "
                           + result);
    }
}
```

Output

```
Max of three numbers = 30
```

7. Bitwise Operators

These operators are used to perform the manipulation of individual bits of a number. They can be used with any of the integer types. They are used when performing update and query operations of the Binary indexed trees.

- **&, Bitwise AND operator:** returns bit by bit AND of input values.
- **|, Bitwise OR operator:** returns bit by bit OR of input values.
- **^, Bitwise XOR operator:** returns bit-by-bit XOR of input values.
- **~, Bitwise Complement Operator:** This is a unary operator which returns the one's complement representation of the input value, i.e., with all bits inverted.

Java

```java
// Java Program to implement
// bitwise operators
import java.io.*;

// Driver class
class GFG {
    // main function
    public static void main(String[] args)
    {
        // Bitwise operators
        int d = 0b1010;
        int e = 0b1100;
        System.out.println("d & e: " + (d & e));
        System.out.println("d | e: " + (d | e));
        System.out.println("d ^ e: " + (d ^ e));
        System.out.println("~d: " + (~d));
        System.out.println("d << 2: " + (d << 2));
        System.out.println("e >> 1: " + (e >> 1));
        System.out.println("e >>> 1: " + (e >>> 1));
    }
}
```

Output

```
d & e: 8
d | e: 14
d ^ e: 6
~d: -11
d << 2: 40
e >> 1: 6
e >>> 1: 6
```

8. Shift Operators

These operators are used to shift the bits of a number left or right, thereby multiplying or dividing the number by two, respectively. They can be used when we have to multiply or divide a number by two. General format-

```
 number shift_op number_of_places_to_shift;
```

- **<<, Left shift operator:** shifts the bits of the number to the left and fills 0 on voids left as a result. Similar effect as multiplying the number with some power of two.
- **>>, Signed Right shift operator:** 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 the initial number. Similar effect to dividing the number with some power of two.
- **>>>, Unsigned Right shift operator:** shifts the bits of the number to the right and fills 0 on voids left as a result. The leftmost bit is set to 0.

Java

```java
// Java Program to implement
// shift operators
import java.io.*;

// Driver Class
class GFG {
    // main function
    public static void main(String[] args)
    {
        int a = 10;

          // using left shift
        System.out.println("a<<1 : " + (a << 1));

        // using right shift
        System.out.println("a>>1 : " + (a >> 1));
    }
}
```

Output

```
a<<1 : 20

a>>1 : 5
```

9. instanceof operator

The instance of the operator is used for type checking. It can be used to test if an object is an instance of a class, a subclass, or an interface. General format-

```
object instance of class/subclass/interface
```

Java

```java
// Java program to illustrate
// instance of operator

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

        Person obj1 = new Person();
        Person obj2 = new Boy();

        // As obj is of type person, it is not an
        // instance of Boy or interface
        System.out.println("obj1 instanceof Person: "
                           + (obj1 instanceof Person));
        System.out.println("obj1 instanceof Boy: "
                           + (obj1 instanceof Boy));
        System.out.println("obj1 instanceof MyInterface: "
                           + (obj1 instanceof MyInterface));

        // Since obj2 is of type boy,
        // whose parent class is person
        // and it implements the interface Myinterface
        // it is instance of all of these classes
        System.out.println("obj2 instanceof Person: "
                           + (obj2 instanceof Person));
        System.out.println("obj2 instanceof Boy: "
                           + (obj2 instanceof Boy));
        System.out.println("obj2 instanceof MyInterface: "
                           + (obj2 instanceof MyInterface));
    }
}

class Person {
}
```

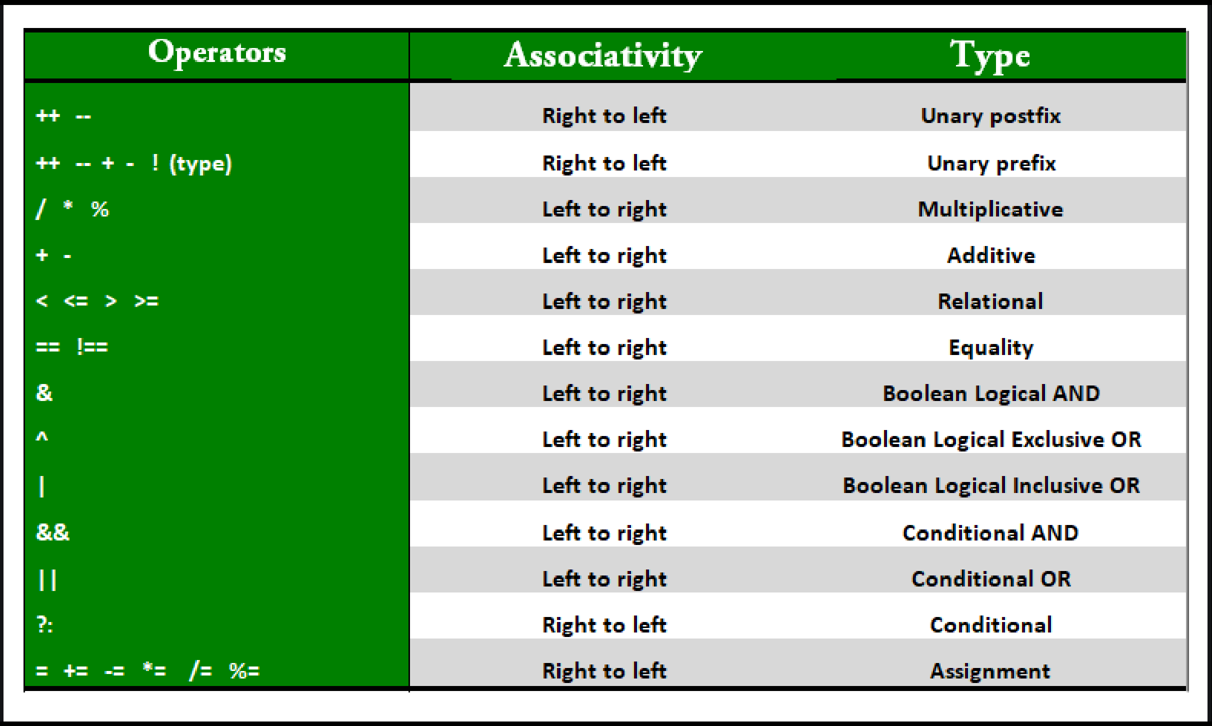
```java
class Boy extends Person implements MyInterface {
}

interface MyInterface {
}
```

Output

```
obj1 instanceof Person: true
obj1 instanceof Boy: false
obj1 instanceof MyInterface: false
obj2 instanceof Person: true
obj2 instanceof Boy: true
obj2 instanceof MyInterface: true
```

Precedence and Associativity of Java Operators

Precedence and associative rules are used when dealing with hybrid equations involving more than one type of operator. In such cases, these rules determine which part of the equation to consider first, as there can be many different valuations for the same equation. The below table depicts the precedence of operators in decreasing order as magnitude, with the top representing the highest precedence and the bottom showing the lowest precedence.



Operators	Associativity	Type
++ --	Right to left	Unary postfix
++ -- + - ! (type)	Right to left	Unary prefix
/ * %	Left to right	Multiplicative
+ -	Left to right	Additive
< <= > >=	Left to right	Relational
== !==	Left to right	Equality
&	Left to right	Boolean Logical AND
^	Left to right	Boolean Logical Exclusive OR
|	Left to right	Boolean Logical Inclusive OR
&&	Left to right	Conditional AND
||	Left to right	Conditional OR
?:	Right to left	Conditional
= += -= *= /= %=	Right to left	Assignment

Types of Variables in Java:

1. **Local Variables**:

   - Defined within a block, method, or constructor.
   - Created and destroyed with the block or method scope.
   - Scope limited to the block where declared.
   - Must be initialized before use.
   - Example: Declared and used within methods or blocks.

```
// Java Program to implement
// Local Variables
import java.io.*;

class GFG {
    public static void main(String[] args)
    {
        // Declared a Local Variable
        int var = 10;

        // This variable is local to this main method only
        System.out.println("Local Variable: " + var);
// output: Local Variable: 10
    }
}
```

```java
package a;
public class LocalVariable {
    public static void main(String[] args)
    {
        // x is a local variable
        int x = 10;

        // message is also a local
        // variable
        String message = "Hello, world!";

        System.out.println("x = " + x);
        System.out.println("message = " + message);

        if (x > 5) {
            // result is a
            // local variable
            String result = "x is greater than 5";
            System.out.println(result);
        }

        // Uncommenting the line below will result in a
        // compile-time error System.out.println(result);

        for (int i = 0; i < 3; i++) {
            String loopMessage
                = "Iteration "
                  + i; // loopMessage is a local variable
            System.out.println(loopMessage);
        }

        // Uncommenting the line below will result in a
        // compile-time error
        // System.out.println(loopMessage);
    }
}
```

Output :

```
message = Hello, world!
x is greater than 5
Iteration 0
Iteration 1
Iteration 2
```

2. **Instance Variables**:

- Non-static variables declared in a class.
- Created when an object is instantiated and destroyed with the object.
- Accessible by creating objects of the class.
- Default values assigned if not explicitly initialized.
- Example: Represent properties of objects.

```java
// Java Program to demonstrate
// Instance Variables
import java.io.*;

class GFG {

    // Declared Instance Variable
    public String geek;
    public int i;
    public Integer I;
    public GFG()
    {
        // Default Constructor
        // initializing Instance Variable
        this.geek = "Shubham Jain";
    }

    // Main Method
    public static void main(String[] args)
    {
        // Object Creation
        GFG name = new GFG();

        // Displaying O/P
        System.out.println("Geek name is: " + name.geek);
        System.out.println("Default value for int is "
                           + name.i);

        // toString() called internally
        System.out.println("Default value for Integer is "
                           + name.I);
    }
}
```

Output

```
Geek name is: Shubham Jain

Default value for int is 0

Default value for Integer is null
```

3. **Static Variables**:

   - Also known as class variables.
   - Declared using the **static** keyword in a class.
   - Only one copy exists per class, regardless of object creation.
   - Created at the start of program execution and destroyed when execution ends.
   - Default values assigned if not explicitly initialized.
   - Accessed using class name or object reference (though discouraged).
   - Example: Shared among all instances of a class.

4.

```java
// Java Program to demonstrate
// Static variables
import java.io.*;

class GFG {
    // Declared static variable
    public static String geek = "Shubham Jain";

    public static void main(String[] args)
    {

        // geek variable can be accessed without object
        // creation Displaying O/P GFG.geek --> using the
        // static variable
        System.out.println("Geek Name is : " + GFG.geek);

        // static int c=0;
        // above line,when uncommented,
        // will throw an error as static variables cannot be
        // declared locally.
    }
}
```

5. **Output**

```
6. Geek Name is : Shubham Jain
```

Scope of Variables In Java

Scope of a variable is the part of the program where the variable is accessible. Like C/C++, in Java, all identifiers are lexically (or statically) scoped, i.e.scope of a variable can determined at compile time and independent of function call stack. Java programs are organized in the form of classes. Every class is part of some package. Java scope rules can be covered under following categories.

Member Variables (Class Level Scope)

These variables must be declared inside class (outside any function). They can be directly accessed anywhere in class. Let’s take a look at an example:

```
public class Test

{

    // All variables defined directly inside a class

    // are member variables

    int a;

    private String b;

    void method1() {....}

    int method2() {....}

    char c;

}
```

- We can declare class variables anywhere in class, but outside methods.
- Access specified of member variables doesn’t affect scope of them within a class.
- Member variables can be accessed outside a class with following rules

```
Modifier      Package  Subclass  World

public          Yes       Yes      Yes

protected       Yes       Yes      No

Default (no
modifier)       Yes        No      No

private         No         No      No
```

Local Variables (Method Level Scope)

Variables declared inside a method have method level scope and can't be accessed outside the method.

```
public class Test
{
    void method1()
    {
       // Local variable (Method level scope)
       int x;
    }
}
```

Note : Local variables don't exist after method's execution is over.
Here's another example of method scope, except this time the variable got passed in as a parameter to the method:

```
class Test
{
    private int x;
    public void setX(int x)
    {
        this.x = x;
    }
}
```

The above code uses this keyword to differentiate between the local and class variables.
As an exercise, predict the output of following Java program.

- Java

```
public class Test
{
    static int x = 11;
    private int y = 33;
    public void method1(int x)
    {
        Test t = new Test();
```

```java
        this.x = 22;
        y = 44;

        System.out.println("Test.x: " + Test.x);
        System.out.println("t.x: " + t.x);
        System.out.println("t.y: " + t.y);
        System.out.println("y: " + y);
    }

    public static void main(String args[])
    {
        Test t = new Test();
        t.method1(5);
    }
}
```

Output:

```
Test.x: 22
t.x: 22
t.y: 33
y: 44
```

Loop Variables (Block Scope)

A variable declared inside pair of brackets "{" and "}" in a method has scope within the brackets only.

- Java

```java
public class Test
{
    public static void main(String args[])
    {
        {
            // The variable x has scope within
            // brackets
            int x = 10;
            System.out.println(x);
        }

        // Uncommenting below line would produce
        // error since variable x is out of scope.

        // System.out.println(x);
    }
}
```

Output:

```
10
```

As another example, consider following program with a for loop.

- Java

```
class Test
{
    public static void main(String args[])
    {
        for (int x = 0; x < 4; x++)
        {
            System.out.println(x);
        }

        // Will produce error
        System.out.println(x);
    }
}
```

Output:

```
11: error: cannot find symbol
```

```
        System.out.println(x);
```

The right way of doing above is,

- Java

```java
// Above program after correcting the error
class Test
{
    public static void main(String args[])
    {
        int x;
        for (x = 0; x < 4; x++)
        {
            System.out.println(x);
        }

       System.out.println(x);
    }
}
```

Output:

```
0
1
2
3
4
```

Let’s look at tricky example of loop scope. Predict the output of following program. You may be surprised if you are regular C/C++ programmer.

- Java

```java
class Test
{
    public static void main(String args[])
    {
        int a = 5;
        for (int a = 0; a < 5; a++)
        {
            System.out.println(a);
        }
    }
}
```

Output :

```
6: error: variable a is already defined in method go(int)
       for (int a = 0; a < 5; a++)
                ^
1 error
```

Note:- In C++, it will run. But in java it is an error because in java, the name of the variable of inner and outer loop must be different.
A similar program in C++ works. See this.
As an exercise, predict the output of the following Java program.

- Java

```
class Test
{
    public static void main(String args[])
    {
        {
            int x = 5;
            {
                int x = 10;
                System.out.println(x);
            }
        }
    }
}
```

Q. From the above knowledge, tell whether the below code will run or not.

- Java

```
class Test {
    public static void main(String args[])
    {
        for (int i = 1; i <= 10; i++) {
            System.out.println(i);
        }
        int i = 20;
        System.out.println(i);
    }
}
```

Output :

```
1
2
3
4
5
6
7
8
9
10
20
```

Yes, it will run!
See the program carefully, inner loop will terminate before the outer loop variable is declared.So the inner loop variable is destroyed first and then the new variable of same name has been created.

Some Important Points about Variable scope in Java:

- In general, a set of curly brackets { } defines a scope.

- In Java we can usually access a variable as long as it was defined within the same set of brackets as the code we are writing or within any curly brackets inside of the curly brackets where the variable was defined.
- Any variable defined in a class outside of any method can be used by all member methods.
- When a method has the same local variable as a member, “this” keyword can be used to reference the current class variable.
- For a variable to be read after the termination of a loop, It must be declared before the body of the loop.

Wrapper Classes in Java

A Wrapper class in Java is a class whose object wraps or contains primitive data types. When we create an object to a wrapper class, it contains a field and in this field, we can store primitive data types. In other words, we can wrap a primitive value into a wrapper class object. Let’s check on the wrapper classes in Java with examples:

Need of Wrapper Classes

There are certain needs for using the Wrapper class in Java as mentioned below:

1. They convert primitive data types into objects. Objects are needed if we wish to modify the arguments passed into a method (because primitive types are passed by value).
2. The classes in java.util package handles only objects and hence wrapper classes help in this case also.
3. Data structures in the Collection framework, such as ArrayList and Vector, store only objects (reference types) and not primitive types.
4. An object is needed to support synchronization in multithreading.

Advantages of Wrapper Classes

1. Collections allowed only object data.
2. On object data we can call multiple methods compareTo(), equals(), toString()
3. Cloning process only objects
4. Object data allowed null values.
5. Serialization can allow only object data.

Below are given examples of wrapper classes in Java with their corresponding Primitive data types in Java.

Primitive Data Types and their Corresponding Wrapper Class

Primitive Data Type	Wrapper Class
char	Character
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double
boolean	Boolean

Autoboxing and Unboxing

1. Autoboxing

The automatic conversion of primitive types to the object of their corresponding wrapper classes is known as autoboxing. For example – conversion of int to Integer, long to Long, double to Double, etc.

Example:

- Java

```
// Java program to demonstrate Autoboxing


import java.util.ArrayList;

class Autoboxing {

    public static void main(String[] args)

    {

        char ch = 'a';


        // Autoboxing- primitive to Character object
        // conversion
        Character a = ch;


        ArrayList<Integer> arrayList
            = new ArrayList<Integer>();


        // Autoboxing because ArrayList stores only objects
        arrayList.add(25);


        // printing the values from object
        System.out.println(arrayList.get(0));
    }
}
```

Output

```
25
```

2. Unboxing

It is just the reverse process of autoboxing. Automatically converting an object of a wrapper class to its corresponding primitive type is known as unboxing. For example – conversion of Integer to int, Long to long, Double to double, etc.

Example:

- Java

```
// Java program to demonstrate Unboxing
import java.util.ArrayList;


class Unboxing {
    public static void main(String[] args)
    {
        Character ch = 'a';
        // unboxing - Character object to primitive
        // conversion
        char a = ch;

        ArrayList<Integer> arrayList
            = new ArrayList<Integer>();
        arrayList.add(24);

        // unboxing because get method returns an Integer
        // object
        int num = arrayList.get(0);

        // printing the values from primitive data types
        System.out.println(num);
    }
}
```

Output

```
24
```

Java Wrapper Classes Example

- Java

```
// Java program to demonstrate Wrapping and UnWrapping
// in Classes
import java.io.*;

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

```java
    {
        // byte data type
        byte a = 1;

        // wrapping around Byte object
        Byte byteobj = new Byte(a);
        // Use with Java 9
        // Byte byteobj = Byte.valueOf(a);

        // int data type
        int b = 10;

        // wrapping around Integer object
        Integer intobj = new Integer(b);
        // Use with Java 9
        // Integer intobj = Integer.valueOf(b);

        // float data type
        float c = 18.6f;

        // wrapping around Float object
        Float floatobj = new Float(c);
        // Use with Java 9
        // Float floatobj = Float.valueOf(c);

        // double data type
        double d = 250.5;

        // Wrapping around Double object
        Double doubleobj = new Double(d);
        // Use with Java 9
        // Double doubleobj = Double.valueOf(d);

        // char data type
        char e = 'a';

        // wrapping around Character object
        Character charobj = e;

        // printing the values from objects
        System.out.println(
            "Values of Wrapper objects (printing as objects)");
        System.out.println("\nByte object byteobj: "
                           + byteobj);
        System.out.println("\nInteger object intobj: "
                           + intobj);
        System.out.println("\nFloat object floatobj: "
                           + floatobj);
        System.out.println("\nDouble object doubleobj: "
                           + doubleobj);
        System.out.println("\nCharacter object charobj: "
                           + charobj);

        // objects to data types (retrieving data types from
        // objects) unwrapping objects to primitive data
        // types
        byte bv = byteobj;
        int iv = intobj;
        float fv = floatobj;
        double dv = doubleobj;
        char cv = charobj;

        // printing the values from data types
        System.out.println(
            "\nUnwrapped values (printing as data types)");
        System.out.println("\nbyte value, bv: " + bv);
        System.out.println("\nint value, iv: " + iv);
        System.out.println("\nfloat value, fv: " + fv);
        System.out.println("\ndouble value, dv: " + dv);
        System.out.println("\nchar value, cv: " + cv);
    }
}
```

Output

```
Values of Wrapper objects (printing as objects)

Byte object byteobj: 1

Integer object intobj: 10

Float object floatobj: 18.6

Double object doubleobj: 250.5

Character object charobj: a

Unwrapped values (printing as data types)

byte value, bv: 1

int value, iv: 10
```

```
float value, fv: 18.6

double value, dv: 250.5

char value, cv: a
```

Custom Wrapper Classes in Java

Java Wrapper classes wrap the primitive data types. We can create a class that wraps data inside it. So let us check how to create our own custom wrapper class in Java. It can be implemented for creating certain structures like queues, stacks, etc.

Example:

- Java

```java
// Java Program to implement

// Custom wrapper class

import java.io.*;


// wrapper class

class Maximum {

    private int maxi = 0;

    private int size = 0;


    public void insert(int x)

    {

        this.size++;

        if (x <= this.maxi)

            return;

        this.maxi = x;

    }
    public int top() { return this.maxi; }

    public int elementNumber() { return this.size; }
};

//
class GFG {
    public static void main(String[] args)
    {
        Maximum x = new Maximum();
        x.insert(12);
        x.insert(3);
        x.insert(23);

        System.out.println("Maximum element: " + x.top());
        System.out.println("Number of elements inserted: "
                           + x.elementNumber());
    }
}
```

Output

```
Maximum element: 23
Number of elements inserted: 3
```