Introduction to Java

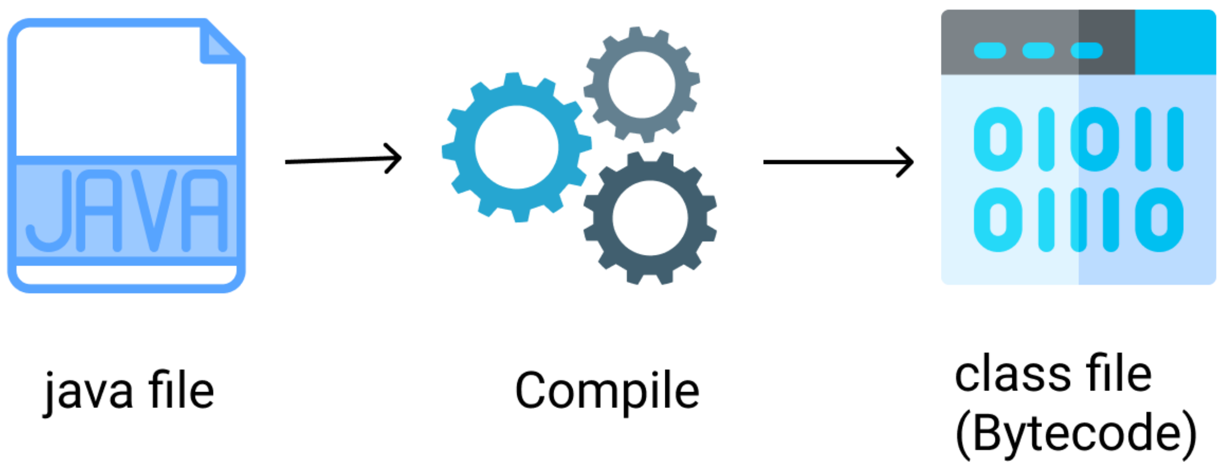
Getting started with any programming language should be purposeful, we must understand the **features and power** of the programming language to develop a deep interest in it.
Java was developed after C and C++ around 1991. We can consider Java as a **complete package** for software developers to play with it for a variety of applications they want to build.

Java is currently the most used programming language worldwide. The biggest tech giants such as **Amazon, Google, TCS, Wipro, etc.,** have a huge percentage of code in Java.

Java is power-packed with ample features. Here are some of the most important features of Java Programming Language:

- **Simple -** Java was designed for professional developers to learn and use easily. If you are aware of basic programming principles, then Java is not a hard nut to crack.
- **Secure -** Java achieves security by confining Java Programs to Java Execution Environment and not allowing Java programs to access other parts of the computer system.
- **Portable -** Portable code must run on a variety of operating systems, CPUs, and browsers. When a Java program is compiled, a Bytecode is generated, this bytecode can be executed on any platform using JVM (Java Virtual Machine). We will discuss JVM details in the latter part of the tutorial.
- **Object-Oriented -** Java provides a clean, usable, and pragmatic approach to objects. It was designed on the principle that "Everything under the sun is an Object."
- **Robust -** The robustness of the program is the reliable execution on a variety of systems. To better understand this, consider one of the main reasons for program failure i.e memory management. In C/C++, a programmer must manually allocate and free dynamic memory. There are cases where programmers forgot to free up the dynamic memory which may cause program failure. Java virtually solves this problem as unused objects are garbage collected automatically.
- **Architecture Neutral -** The central issue for Java Designers was that of code longevity and portability. One of the main problems for programmers was that there was no guarantee that the program running perfectly fine today will run tomorrow because of the system upgrades, processor upgrades, or changes in core system resources. Java Designers made several hard decisions and designed the Java Virtual Machine to solve this problem. Their goal was "write once, run anywhere, any time, forever".
- **Distributed -** Java is designed for the distributed environment because it handles TCP/IP protocol. It is very much useful in building large-scale distributed computing-based applications. Java also supports Remote Method Invocation. This feature enables a program to invoke methods across a network.
- **Dynamic -** Java programs have run time metainformation to verify and resolve accesses of the objects at runtime. This makes it possible to dynamically link code in a safe and secure environment. It also gives us the feature to update small fragments of bytecode, dynamically updated on a running system.

Java Program Compilation - Java program compilation converts a java program file to a highly optimized class file which is called bytecode.



Java Program Execution - Java program execution is completed with the help of the following tools.

- **Classloader -** It loads the class file for JVM execution.
- **Bytecode Verifier -** It verifies the bytecode and restricts the objects from illegal access to other parts of the system.
- **Interpreter -** It read the bytecode instructions and executes them line by line.

Java Virtual Machine (JVM)

JVM is called an abstract virtual machine because it does not exist physically, but it is a kind of specification that provides a secure runtime environment to execute the bytecode generated through the compiler, JVM actually invokes the main( ) method present in the Java program.
JVM is a part of the JRE(Java Runtime Environment).
Java applications are called WORA (Write once, run anywhere). This means a programmer can develop Java code on one system and can expect it to run on any other Java-enabled system without any adjustment. This is all possible because of JVM.

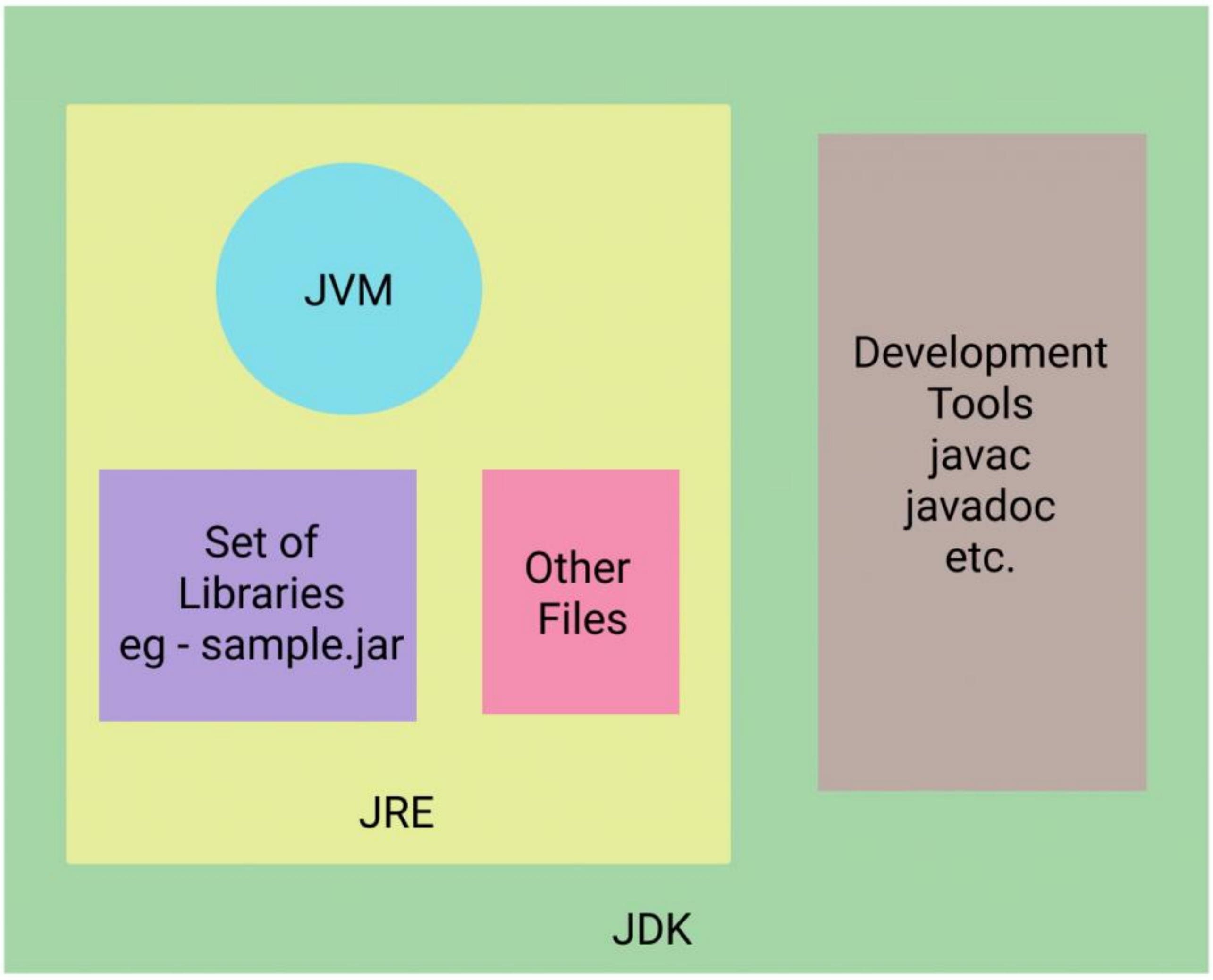
Java Runtime Environment (JRE)

JRE stands for "Java Runtime Environment" and may also be written as "Java RTE." The Java Runtime Environment provides the minimum requirements for executing a Java application, it consists of the Java Virtual Machine (JVM), core classes, and supporting files.

- A **specification** where the working of Java Virtual Machine is specified. But implementation provider is independent to choose the algorithm. Its implementation has been provided by Sun and other companies.
- An **implementation** is a computer program that meets the requirements of the JVM specification.
- **Runtime Instance** Whenever you write a java command on the command prompt to run the java class, an instance of JVM is created.

Java Development Kit (JDK)

Java Development Kit (in short JDK) is a Kit that provides the environment to develop and execute. The JDK contains a private Java Virtual Machine (JVM), interpreter/loader (java), a compiler (javac), an archiver (jar), a documentation generator (Javadoc), etc. to complete the development of a Java Application.


JVM
Set of Libraries eg - sample.jar
Other Files
JRE
Development Tools javac javadoc etc.
JDK

Difference between Java and C++

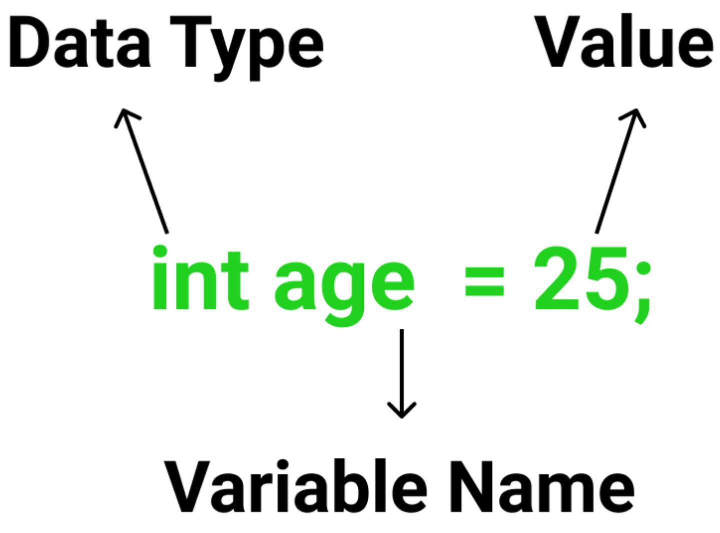
Comparison Metric	C++	Java
Domain and Usage	C++ is mainly used for system programming.	Java is mainly used for building desktop, web, and Mobile Applications.
Compiler and Interpreter	In C++, the program is only compiled and converted into machine code. That is why it is platform dependent.	In Java , program once compiled converted into bytecode, then interpreter reads the bytecode stream to execute line by line through JVM. That is why it is platform-independent
Pointers	C++ supports pointers for memory allocation, You can write pointer programs.	Java supports pointers internally, however you can not write pointer programs.
Structure and Union	C++ have Structures and Unions.	Java doesn't have these, however, you can implement this using classes.
Thread Support	C++ doesn't have inbuilt thread support, you have to use a third-party library.	Java has built-in thread support.
Documentation Comment	C++ doesn't have documentation comments.	Java has documentation comments, which are useful for maintaining the code.
Call by value and reference	C++ supports both call-by-value and call-by-reference.	Java supports only call-by-value.
Operator Overloading	C++ supports operator overloading.	Java doesn't have support operator overloading.

Variables in Java

A **variable** is a name given to a memory location. It is the basic unit of storage in a program.

- The value stored in a variable can be changed during program execution.
- A variable is only a name given to a memory location, all the operations done on the variable effects that memory location.
- In Java, all the variables must be declared before use.

How to declare variables?
We can declare variables in java as follows:



20
RAM
Reserved memory for variable

- **Data Type**: Type of data that can be stored in this variable.
- **Variable Name**: Name given to the variable.
- **Value**: It is the initial value stored in the variable.

Examples:

```
//Declaring float variable
float simpleInterest;
//Declaring and Initializing integer variable
int time = 10, speed = 20;
// Declaring and Initializing character variable
char var = 'h';
```

Types of variables

There are three types of variables in Java:

- Local Variables
- Instance Variables
- Static Variables

Let us now learn about each one of these variables in detail.

1. **Local Variables**: A variable defined within a block or method or constructor is called a local variable.
    - 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.
    - The scope of these variables exists only within the block in which the variable is declared. i.e. we can access these variables only within that block.
    - The initialization of local variables is mandatory.
2. **Instance Variables**: Instance variables are non-static variables declared in a class outside any method, constructor, or block.
    - As instance variables are declared in a class, these variables are created when an object of the class is created and destroyed when the object is destroyed.
    - Unlike local variables, we may use access specifiers for instance variables. If we do not specify any access specifier then the *default* access specifier will be used.
    - Initialisation of the Instance Variable is not mandatory. They are initialized to default values by constructors.
    - Instance Variable can be accessed only by creating objects, outside the class in which they are declared.
3. **Static Variables**: Static variables are also known as Class variables.
    - These variables are declared similarly to instance variables, the difference is that static variables are declared using the static keyword within a class outside any method, constructor, or block.
    - Unlike instance variables, we can only have one copy of a static variable per class irrespective of how many objects we create.
    - Static variables are created at the start of the program execution and are destroyed automatically when the execution ends.
    - Initialisation of Static Variable is not mandatory. They are initialized to default values.
    - If we access the static variable like the Instance variable (through an object), the compiler will show the warning message and it won't halt the program. The compiler will replace the object name with the class name automatically.
    - If we access the static variable without class name, the compiler will automatically append the class name.

Instance variable Vs Static variable

- Each object will have its **own copy** of the instance variable whereas We can only have **one copy** of a static variable per class irrespective of how many objects we create.
- Changes made in an instance variable using one object will **not be reflected** in other objects as each object has its own copy of the instance variable. In the case of static variables, changes **will be reflected** in other objects as static variables are common to all objects of a class.
- We can access instance variables **through object references** and Static Variables can be accessed **directly using the class name.**
- The syntax for static and instance variables:

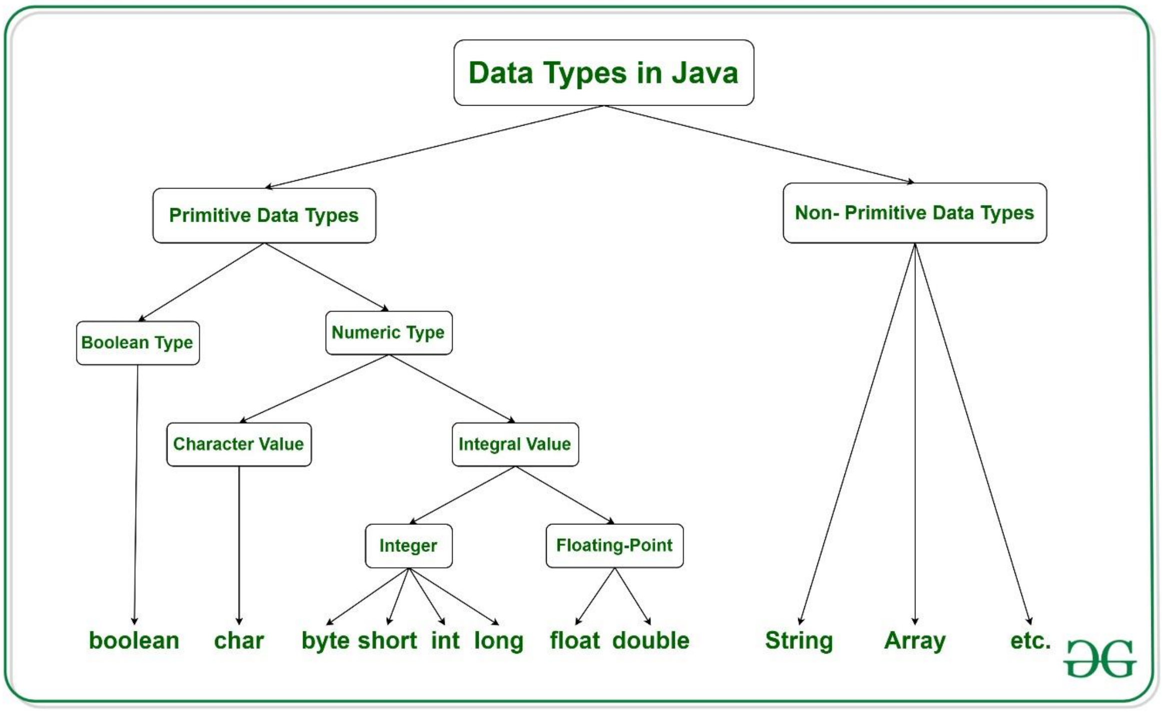
```
class Example
    {
        static int a; //static variable
        int b;        //instance variable
    }
```

Data Types in Java

Data types are different sizes and values that can be stored in the variable that is made as per convenience and circumstances to cover up all test cases. Also, let us cover up other important ailments that there are majorly two types of languages that are as follows:

1. First, one is a **Statically typed language** where each variable and expression type is already known at compile time. Once a variable is declared to be of a certain data type, it cannot hold values of other data types. For example C, C++, and Java.
2. The other is **Dynamically typed languages.** These languages can receive different data types over time. For example Ruby, Python

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

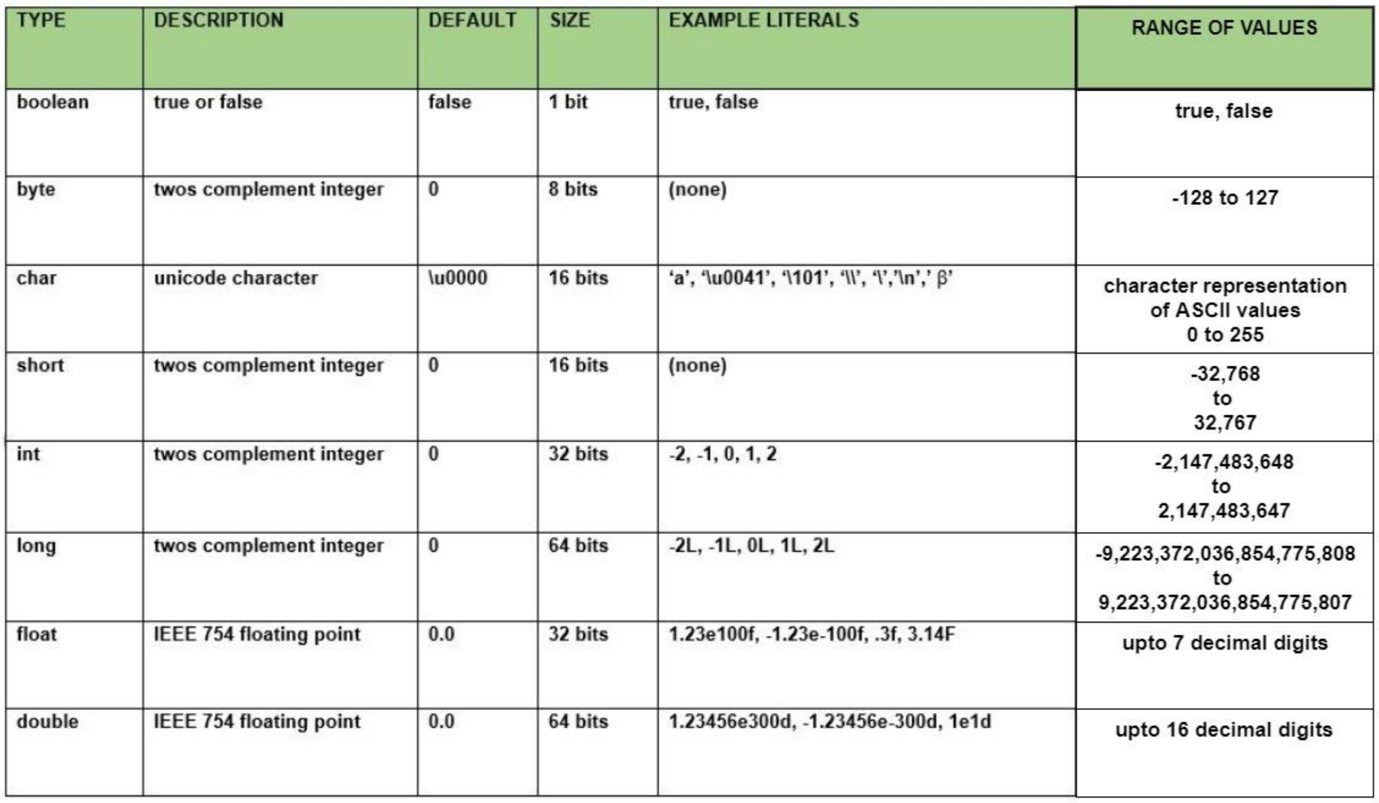


Java has two categories in which data types are segregated

1. **Primitive Data Type:** such as boolean, char, int, short, byte, long, float, and double
2. **Non-Primitive Data Type or Object Data type:** such as String, Array, etc.

Types Of Primitive Data Types

Primitive data are only single values and have no special capabilities. There are **8 primitive data types.** They are depicted below in tabular format below as follows:



TYPE	DESCRIPTION	DEFAULT	SIZE	EXAMPLE LITERALS	RANGE OF VALUES
boolean	true or false	false	1 bit	true, false	true, false
byte	twos complement integer	0	8 bits	(none)	-128 to 127
char	unicode character	\u0000	16 bits	'a', '\u0041', '\101', '\\', '\','\n',' β'	character representation of ASCII values 0 to 255
short	twos complement integer	0	16 bits	(none)	-32,768 to 32,767
int	twos complement integer	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, -1.23456e-300d, 1e1d	upto 16 decimal digits

Type conversion in Java

Java provides various data types just likely any other dynamic languages such as boolean, char, int, unsigned int, signed int, float, double, long, etc in total providing 7 types where every datatype acquires different space while storing in memory. When you assign a value of one data type to another, the two types might not be compatible with each other. If the data types are compatible, then Java will perform the conversion automatically known as Automatic Type Conversion, and if not then they need to be cast or converted explicitly. For example, assigning an int value to a long variable.

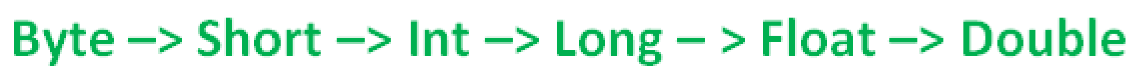
Datatype	**Bits Acquired In Memory**
boolean	1
byte	8 (1 byte)
char	16 (2 bytes)
short	16(2 bytes)
int	32 (4 bytes)
long	64 (8 bytes)
float	32 (4 bytes)
double	64 (8 bytes)

Widening or Automatic Type Conversion

Widening conversion takes place when two data types are automatically converted. This happens when:

- The two data types are compatible.
- When we assign a value of a smaller data type to a bigger data type.

For Example, in java, the numeric data types are compatible with each other but no automatic conversion is supported from numeric type to char or boolean. Also, char and boolean are not compatible with each other.



Widening or Automatic Conversion

Example:

Java
```java
// Java Program to Illustrate Automatic Type Conversion

// Main class
class GFG {

	// Main driver method
	public static void main(String[] args)
	{
		int i = 100;

		// Automatic type conversion
		// Integer to long type
		long l = i;

		// Automatic type conversion
		// long to float type
		float f = l;

		// Print and display commands
		System.out.println("Int value " + i);
		System.out.println("Long value " + l);
		System.out.println("Float value " + f);
	}
}
```

Output
```
Int value 100
Long value 100
Float value 100.0
```

Narrowing or Explicit Conversion

If we want to assign a value of a larger data type to a smaller data type we perform explicit type casting or narrowing.

- This is useful for incompatible data types where automatic conversion cannot be done.

- Here, the target type specifies the desired type to convert the specified value to.

Double –> Float –> Long –> Int –> Short –> Byte

Narrowing or Explicit Conversion

char and number are not compatible with each other. Let's see when we try to convert one into another.

Java

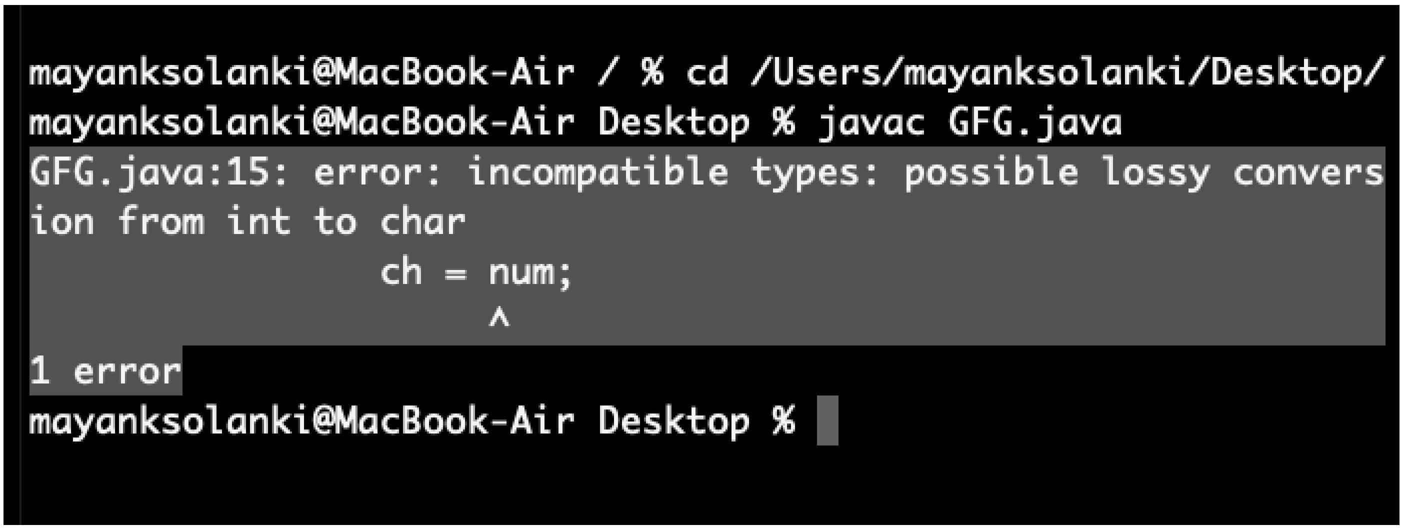
```java
// Java program to illustrate Incompatible data Type
// for Explicit Type Conversion

// Main class
public class GFG {

	// Main driver method
	public static void main(String[] argv)
	{

		// Declaring character variable
		char ch = 'c';
		// Declaringinteger variable
		int num = 88;
		// Trying to insert integer to character
		ch = num;
	}
}
```

Output: An error will be generated



```
mayanksolanki@MacBook-Air / % cd /Users/mayanksolanki/Desktop/
mayanksolanki@MacBook-Air Desktop % javac GFG.java
GFG.java:15: error: incompatible types: possible lossy convers
ion from int to char
                ch = num;
                     ^
1 error
mayanksolanki@MacBook-Air Desktop %
```

This error is generated as an integer variable takes 4 bytes while character datatype requires 2 bytes.We are trying to plot data from 4 bytes into 2 bytes which is not possible.

How to do Explicit Conversion?

Java

```java
// Java program to Illustrate Explicit Type Conversion

// Main class
public class GFG {
        // Main driver method
        public static void main(String[] args)
        {

                // Double datatype
                double d = 100.04;

                // Explicit type casting by forcefully getting
                // data from long datatype to integer type
                long l = (long)d;

                // Explicit type casting
                int i = (int)l;

                // Print statements
                System.out.println("Double value " + d);

                // While printing we will see that
                // fractional part lost
                System.out.println("Long value " + l);

                // While printing we will see that
                // fractional part lost
                System.out.println("Int value " + i);
        }
}
```

Output

```
Double value 100.04
Long value 100
Int value 100
```

Note: *While assigning value to byte type the fractional part is lost and is reduced to modulo 256(range of byte).*

Example:

Java

```
// Java Program to Illustrate Conversion of
// Integer and Double to Byte

// Main class
class GFG {

        // Main driver method
        public static void main(String args[])
        {
                // Declaring byte variable
                byte b;

                // Declaring and initializing integer and double
                int i = 257;
                double d = 323.142;

                // Display message
                System.out.println("Conversion of int to byte.");
```

```
                    // i % 256
                    b = (byte)i;

                    // Print commands
                    System.out.println("i = " + i + " b = " + b);
                    System.out.println(
                              "\nConversion of double to byte.");

                    // d % 256
                    b = (byte)d;

                    // Print commands
                    System.out.println("d = " + d + " b= " + b);
          }
}
```

Output

```
Conversion of int to byte.
i = 257 b = 1

Conversion of double to byte.
d = 323.142 b= 67
```

Operators in Java

Java provides many types of operators which can be used according to the need. They are classified based on the functionality they provide. Some of the types are:

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

Let's take a look at them in detail.

1. Arithmetic Operators: They are used to perform simple arithmetic operations on primitive data types.

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

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

3. Assignment Operator: '=' Assignment operator is used to assigning a value to any variable. It has a 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 left operand with right operand and then assigning it to the variable on the left.
- **-=**, for subtracting right operand from left operand and then assigning it to the variable on the left.
- ***=**, for multiplying left operand with right operand and then assigning it to the variable on the left.
- **/=**, for dividing left operand by right operand and then assigning it to the variable on the left.
- **%=**, for assigning modulo of left operand by right operand and then assigning it to the variable on the left.

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.

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

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.

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 as 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.
- **Precedence and Associativity of 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