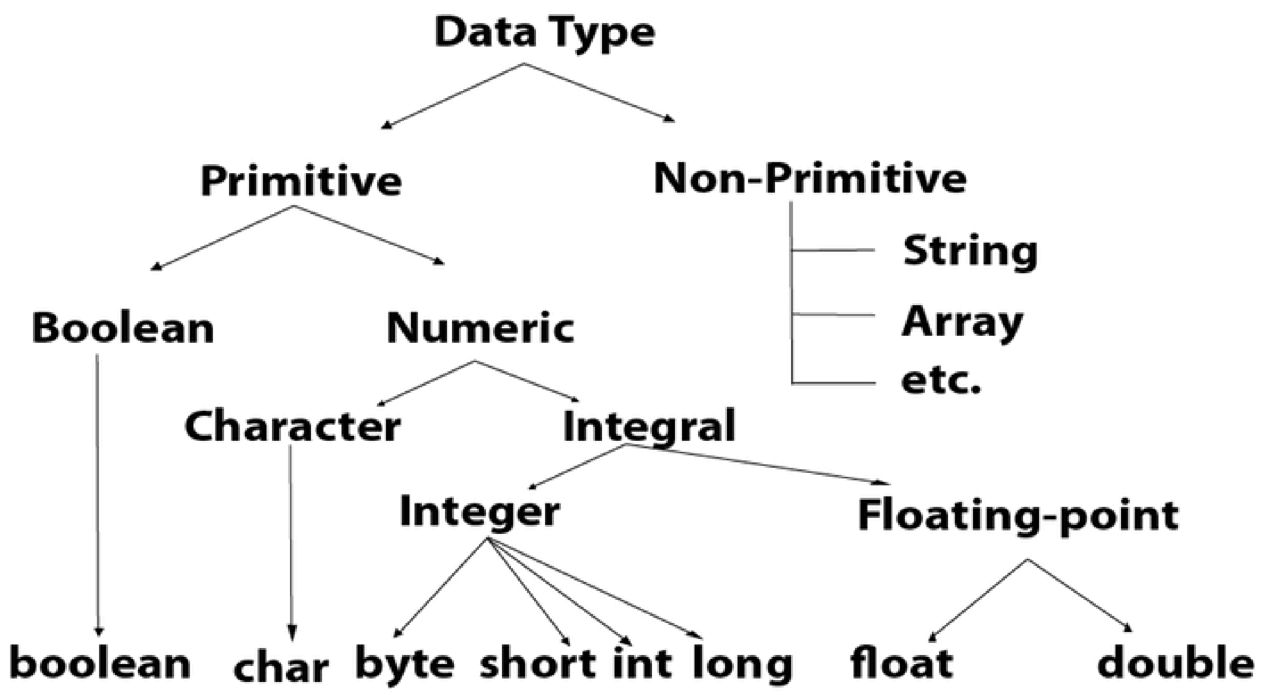
Data Types in Java :



Data type defines the values that a variable can take, for example if a variable has int data type, it can only take integer values.

In java we have two categories of data type: 1) Primitive data types 2) Nonprimitive data types – Arrays and Strings are non-primitive data types. Java is a statically typed language. A language is statically typed, if the data type of a variable is known at compile time. This means that you must specify the type of the variable (Declare the variable) before you can use it.

```
int num;
```

So in order to use the variable num in our program, we must declare it first as shown above. It is a good programming practice to declare all the variables (that you are going to use) in the beginning of the program.

1) Primitive data types

- Primitive data types - includes byte, short, int, long, float, double, boolean and char.

- Java developers included these data types to maintain the portability of java as the size of these primitive data types do not change from one operating system to another.
- Non-primitive data types - such as String, Arrays and Classes (you will learn more about these in a later chapter)

A primitive data type specifies the size and type of variable values, and it has no additional methods.
There are eight primitive data types in Java:

Data Type	**Size**	**Description**
Byte	1 byte	Stores whole numbers from -128 to 127
Short	2 bytes	Stores whole numbers from -32,768 to 32,767
Int	4 bytes	Stores whole numbers from -2,147,483,648 to 2,147,483,647
Long	8 bytes	Stores whole numbers from -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
Float	4 bytes	Stores fractional numbers. Sufficient for storing 6 to 7 decimal digits
Double	8 bytes	Stores fractional numbers. Sufficient for storing 15 decimal digits
Boolean	1 bit	Stores true or false values
char	2 bytes	Stores a single character/letter or ASCII values

Primitive number types are divided into two groups:

Integer types stores whole numbers, positive or negative (such as 123 or 456), without decimals.
Valid types are `byte`, `short`, `int` and `long`. Which type you should use, depends on the numeric value.

Floating point types represents numbers with a fractional part, containing one or more decimals. There are two types: `float` and `double`.
Even though there are many numeric types in Java, the most used for numbers are `int` (for whole numbers) and `double` (for floating point numbers).

Integer Types

Byte

The byte data type can store whole numbers from -128 to 127. This can be used instead of int or other integer types to save memory when you are certain that the value will be within -128 and 127:

. Mostly used to save memory and when you are certain that the numbers would be in the limit specified by byte data type.

Default size of this data type: 1 byte.
Default value: 0

```
public class MyClass {
  public static void main(String[] args) {
    byte myNum = 100;
    System.out.println(myNum);
  }
}
Output: 100
```

Try the same program by assigning value assigning 150 value to variable num, you would get **type mismatch** error because the value 150 is out of the range of byte data type. The range of byte as I mentioned above is -128 to 127.

Short

The short data type can store whole numbers from -32768 to 32767:
This is greater than byte in terms of size and less than integer
Default size of this data type: 2 byte

```
short num = 45678;
```

```
public class MyClass {
  public static void main(String[] args) {
    short myNum = 5000;
    System.out.println(myNum);
  }
}
Output: 5000
```

The byte data type couldn't hold the value 150 but a short data type can because it has a wider range.

int: Used when short is not large enough to hold the number, it has a wider range: -2,147,483,648 to 2,147,483,647 and can store whole numbers.

Default size: 4 byte
Default value: 0

the int data type is the preferred data type when we create variables with a numeric value. Example:

```
public class MyClass {   public static
void main(String[] args) {
    int myNum = 100000;
    System.out.println(myNum);
  }
}
```

Output:

```
100000
```

long:
size: 8 bytes
Default value: 0

The long data type can store whole numbers from -9223372036854775808 to 9223372036854775807. This is used when int is not large enough to store the value. Note that you should end the value with an "L":

```
class JavaExample {     public static void main(String[] args) {
                        long num = -12332252626L;
            System.out.println(num);
    }
}
```

Output:

```
-12332252626
```

Floating Point Types
You should use a floating point type whenever you need a number with a decimal, such as 9.99 or 3.14515.

Float

The float data type can store fractional numbers from 3.4e−038 to 3.4e+038. Note that you should end the value with an "f":
float: Sufficient for holding 6 to 7 decimal digits

size: 4 bytes

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

          float num = 19.98f;
          System.out.println(num);
   }
}
```

Output:

```
19.98
```

Double

The double data type can store fractional numbers from 1.7e−308 to 1.7e+308. Note that you should end the value with a "d":
double: Sufficient for holding 15 decimal digits size:
8 bytes

```
class JavaExample {     public static void main(String[] args) {
                    double num = -42937737.9d;
          System.out.println(num);
   }
}
```

Output:

```
-4.29377379E7
```

Use float or double?

The **precision** of a floating point value indicates how many digits the value can have after the decimal point. The precision of float is only six or seven decimal digits, while double variables have a precision of about 15 digits. Therefore it is safer to use double for most calculations.

Scientific Numbers

A floating point number can also be a scientific number with an "e" to indicate the power of 10:

```
public class MyClass {
  public static void main(String[] args)
{     float f1 = 35e3f;      double d1 =
12E4d;      System.out.println(f1);
    System.out.println(d1);
  }
}
```

Output:

```
35000.0
120000.0
```

Booleans

A boolean data type is declared with the boolean keyword and can only take the values true or false:

```
public class MyClass {   public static
void main(String[] args) {      boolean
isJavaFun = true;      boolean
isFishTasty = false;
System.out.println(isJavaFun);
    System.out.println(isFishTasty);
  }
}
```

Output:

```
true
fals
e
```

Characters

The char data type is used to store a **single** character. The character must be surrounded by single quotes, like 'A' or 'c':

```
public class MyClass {
  public static void main(String[] args) {
    char myGrade = 'B';
    System.out.println(myGrade);
  }
}
```

Output:
```
B
```

Alternatively, you can use ASCII values to display certain characters:

```
public class MyClass {   public static
void main(String[] args) {
    char a = 65, b = 66, c = 67;
    System.out.println(a);
    System.out.println(b);
    System.out.println(c);
  }
}
```

Output:

Strings

The String data type is used to store a sequence of characters (text). String values must be surrounded by double quotes:

Example

```
public class MyClass {
  public static void main(String[] args) {
    String greeting = "Hello World";
    System.out.println(greeting);
  }
}
```

Output:
```
Hello World
```

Non-Primitive Data Types

Non-primitive data types are called **reference types** because they refer to objects.

The main difference between **primitive** and **non-primitive** data types are:

- Primitive types are predefined (already defined) in Java. Non-primitive types are created by the programmer and is not defined by Java (except for String).
- Non-primitive types can be used to call methods to perform certain operations, while primitive types cannot.
- A primitive type has always a value, while non-primitive types can be null.
- A primitive type starts with a lowercase letter, while non-primitive types starts with an uppercase letter.
- The size of a primitive type depends on the data type, while nonprimitive types have all the same size.

Examples of non-primitive types are Strings, Arrays, Classes, Interface, etc. You will learn more about these in a later chapter.

1.8 Identifiers-Naming Conventions

In programming languages, identifiers are used for identification purpose. In Java, an identifier can be a class name, method name, variable name or a label.

Java naming convention is a rule to follow as you decide what to name your identifiers such as class, package, variable, constant, method, etc.

But, it is not forced to follow. So, it is known as convention not rule. These conventions are suggested by several Java communities such as Sun Microsystems and Netscape.

All the classes, interfaces, packages, methods and fields of Java programming language are given according to the Java naming convention. If you fail to follow these conventions, it may generate confusion or erroneous code. **For example :**

```
public class Test
{
   public static void main(String[] args)
   {
        int a = 20;
    }
}
```

In the above java code, we have 5 identifiers namely :

- **Test :** class name.
- **main :** method name.
- **String :** predefined class name.
- **args :** variable name.
- **a :** variable name.

Rules for defining Java Identifiers:

There are certain rules for defining a valid java identifier. These rules must be followed, otherwise we get compile-time error. These rules are also valid for other languages like C,C++.

- ✓ The only allowed characters for identifiers are all alphanumeric characters([A-Z],[a-z],[0-9]), ‗$‗(dollar sign) and ‗_‗ (underscore).For example ―geek@‖ is not a valid java identifier as it contain ‗@' special character.
- ✓ Identifiers should not start with digits([0-9]). For example ―123geeks‖ is a not a valid java identifier.
- ✓ Java identifiers are case-sensitive.
- ✓ There is no limit on the length of the identifier but it is advisable to use an optimum length of 4 – 15 letters only.
- ✓ Reserved Words can't be used as an identifier. For example ―int while = 20;‖ is an invalid statement as while is a reserved word. There are 53 reserved words in Java.

Examples of valid identifiers:

MyVariable MYVARIABLE
myvariable
x
i

x1
i1
_myvariable $myvariable sum_of_array geeks123

Examples of invalid identifiers :
My Variable // contains a space 123geeks   // Begins with a digit
a+c // plus sign is not an alphanumeric character variable-2 //
hyphen is not an alphanumeric character sum_&_difference //
ampersand is not an alphanumeric character

Advantage of naming conventions in java

By using standard Java naming conventions, you make your code easier to read for yourself and other programmers. Readability of Java program is very important. It indicates that less time is spent to figure out what the code does.

The following are the key rules that must be followed by every identifier:

- The name must not contain any white spaces.
- The name should not start with special characters like & (ampersand), $ (dollar), _ (underscore).

Let's see some other rules that should be followed by identifiers.

Class ○ It should start with the uppercase letter. ○ It should be a noun such as Color, Button, System, Thread, etc. ○ Use appropriate words, instead of acronyms. ○ **Example: -**

```
public class Employee
{
//code snippet
}
```

Interface ○ It should start with the uppercase letter. ○ It should be an adjective such as Runnable, Remote, ActionListener. ○ Use appropriate words, instead of acronyms. ○ **Example: -**

```
interface Printable
{
//code snippet
}
```

Method

- It should start with lowercase letter.
- It should be a verb such as main(), print(), println().

- If the name contains multiple words, start it with a lowercase letter followed by an uppercase letter such as actionPerformed(). ○ **Example:-**

```
 class Employee
{
//method
void draw()
{
//code snippet
}
}
```

Variable

- It should start with a lowercase letter such as id, name.
- It should not start with the special characters like & (ampersand), $ (dollar), _ (underscore). ○ If the name contains multiple words, start it with the lowercase letter followed by an uppercase letter such as firstName, lastName.
- Avoid using one-character variables such as x, y, z. ○ **Example :-**

```
  class Employee
{
//variable   int
id;
//code snippet
}
```

Package

- It should be a lowercase letter such as java, lang.
- If the name contains multiple words, it should be separated by dots (.) such as java.util, java.lang.

Example :-

```
package com.javatpoint; //package   class
Employee
{
//code snippet   }
```

Constant

- It should be in uppercase letters such as RED, YELLOW.
- If the name contains multiple words, it should be separated by an underscore(_) such as MAX_PRIORITY.
- It may contain digits but not as the first letter.
- **Example :-**

```
class Employee
```

```
{
//constant
 static final int MIN_AGE = 18;
//code snippet
}
```

1.9 Reserved Words:

Any programming language reserves some words to represent functionalities defined by that language. These words are called **Reserved Words**. They can be briefly categorized into two parts : keywords(50) and literals(3).

Identifiers are used by symbol tables in various analyzing phases(like lexical, syntax, semantic) of a compiler architecture.

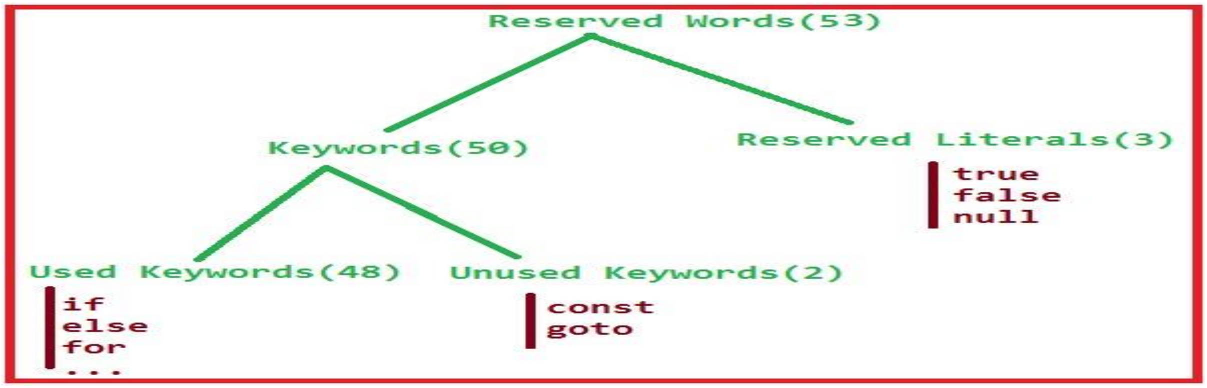
Note: The keywords const and goto are reserved, even though they are not currently used. In place of const, final keyword is used. Some keywords like strictfp are included in later versions of Java

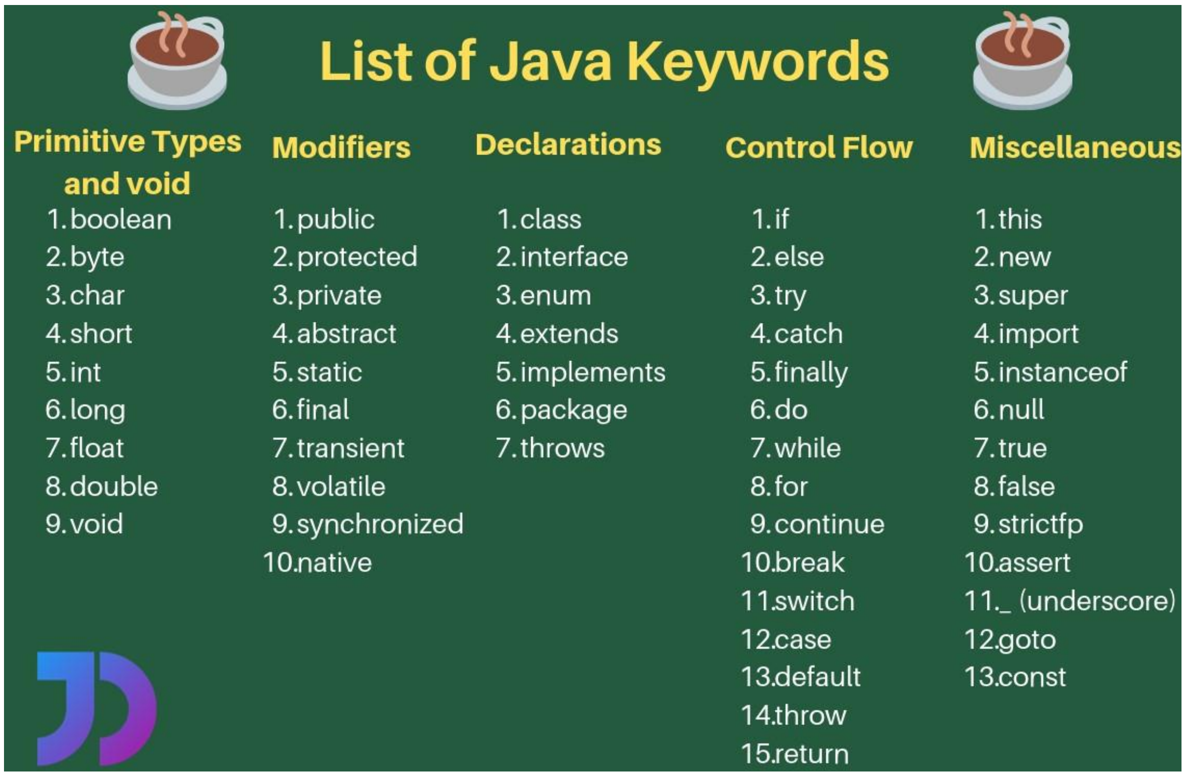
Keywords have a special meaning in a language, and are part of the syntax. **Reserved words** are words that cannot be used as identifiers (variables, functions, etc.), because they are reserved by the language.

In practice most keywords are reserved words and vice versa. But because they're two different things it may happen that a keyword is not a reserved word (e.g. a keyword only has meaning in a special context, and can therefore be used as an identifier), or a reserved word is not a keyword (e.g. because it is reserved for future use).

A good example of this distinction is "goto" in Java. It's not a language keyword (i.e. it's not valid Java), but it *is* a reserved word **keywords**

define functionalities and literals defines a value.





List of Java Keywords

Primitive Types and void	Modifiers	Declarations	Control Flow	Miscellaneous
1. boolean	1. public	1. class	1. if	1. this
2. byte	2. protected	2. interface	2. else	2. new
3. char	3. private	3. enum	3. try	3. super
4. short	4. abstract	4. extends	4. catch	4. import
5. int	5. static	5. implements	5. finally	5. instanceof
6. long	6. final	6. package	6. do	6. null
7. float	7. transient	7. throws	7. while	7. true
8. double	8. volatile		8. for	8. false
9. void	9. synchronized		9. continue	9. strictfp
	10. native		10. break	10. assert
			11. switch	11. _ (underscore)
			12. case	12. goto
			13. default	13. const
			14. throw	
			15. return	

Keywords:

Keywords or Reserved words are the words in a language that are used for some internal process or represent some predefined actions. These words are therefore not allowed to use as a variable names or objects. Doing this will result into a compile time error. Java also contains a list of reserved words or keywords. These are:

1. **abstract** -Specifies that a class or method will be implemented later, in a subclass

2. **assert** -Assert describes a predicate (a true–false statement) placed in a Java program to indicate that the developer thinks that the predicate is always true at that place. If an assertion evaluates to false at run-time, an assertion failure results, which typically causes execution to abort.
3. **boolean** – A data type that can hold True and False values only
4. **break** – A control statement for breaking out of loops
5. **byte** – A data type that can hold 8-bit data value
6. **case** – Used in switch statements to mark blocks of text
7. **catch** – Catches exceptions generated by try statements
8. **char** – A data type that can hold unsigned 16-bit Unicode characters
9. **class** -Declares a new class
10. **continue** -Sends control back outside a loop
11. **default** -Specifies the default block of code in a switch statement
12. **do** -Starts a do-while loop
13. **double** – A data type that can hold 64-bit floating-point numbers
14. **else** – Indicates alternative branches in an if statement
15. **enum** – A Java keyword used to declare an enumerated type. Enumerations extend the base class.
16. **extends** -Indicates that a class is derived from another class or interface
17. **final** -Indicates that a variable holds a constant value or that a method will not be overridden
18. **finally** -Indicates a block of code in a try-catch structure that will always be executed
19. **float** -A data type that holds a 32-bit floating-point number
20. **for** -Used to start a for loop
21. **if** -Tests a true/false expression and branches accordingly
22. **implements** -Specifies that a class implements an interface
23. **import** -References other classes
24. **instanceof** -Indicates whether an object is an instance of a specific class or implements an interface
25. **int** – A data type that can hold a 32-bit signed integer
26. **interface** – Declares an interface
27. **long** – A data type that holds a 64-bit integer
28. **native** -Specifies that a method is implemented with native (platform-specific) code
29. **new** – Creates new objects
30. **null** -Indicates that a reference does not refer to anything
31. **package** – Declares a Java package

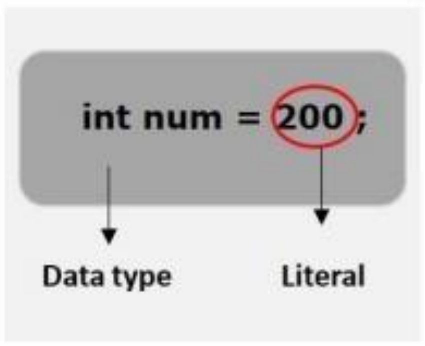
32. **private** -An access specifier indicating that a method or variable may be accessed only in the class it's declared in
33. **protected** – An access specifier indicating that a method or variable may only be accessed in the class it's declared in (or a subclass of the class it's declared in or other classes in the same package)
34. **public** – An access specifier used for classes, interfaces, methods, and variables indicating that an item is accessible throughout the application (or where the class that defines it is accessible)
35. **return** -Sends control and possibly a return value back from a called method
36. **short** – A data type that can hold a 16-bit integer
37. **static** -Indicates that a variable or method is a class method (rather than being limited to one particular object)
38. **strictfp** – A Java keyword used to restrict the precision and rounding of floating point calculations to ensure portability.
39. **super** – Refers to a class's base class (used in a method or class constructor)
40. **switch** -A statement that executes code based on a test value
41. **synchronized** -Specifies critical sections or methods in multithreaded code
42. **this** -Refers to the current object in a method or constructor
43. **throw** – Creates an exception
44. **throws** -Indicates what exceptions may be thrown by a method
45. **transient** -Specifies that a variable is not part of an object's persistent state
46. **try** -Starts a block of code that will be tested for exceptions
47. **void** -Specifies that a method does not have a return value
48. **volatile** -Indicates that a variable may change asynchronously
49. **while** -Starts a while loop

Literals in Java

Literal: Any constant value which can be assigned to the variable is called as literal/constant.

A literal is a source code representation of a fixed value. They are represented directly in the code without any computation.

Example byte a = 68; char a = 'A'



byte, int, long, and short can be expressed in decimal(base 10), hexadecimal(base 16) or octall(base 8) number systems as well .Prefix 0 is used to indicate octal, and prefix 0x indicates hexadecimal when using these number systems for literals.

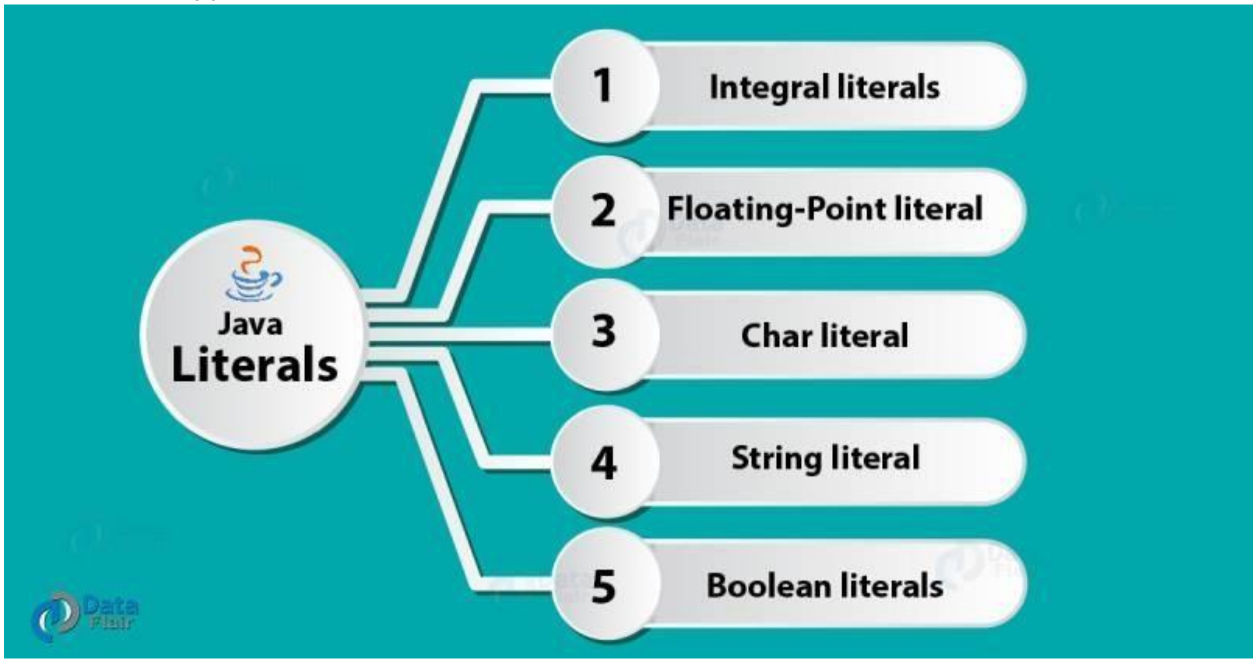
Literals are number, text, or anything that represent a value. In other words, Literals in Java are the constant values assigned to the variable. It is also called a constant.

For example

1. int x = 100;

So, 100 is literal.

There are 5 types of Literals can be seen in Java.



1. Integral Literals in Java

We can specify the integer literals in 4 different ways –

- **Decimal (Base 10)**

Digits from 0-9 are allowed in this form.

```
1. Int x = 101;
```

- ✓ First, a decimal or hex bit in this tutorial represents a single number, digit, or letter. A decimal is also called base 10 and denary because it consists of ten numbers. These are 0, 1, 2, 3, 4, 5, 6, 7, 8, 9.

- ✓ A decimal is a number system and can be represented using a subscript of 10 (i.e. 23510 reads as two hundred and thirty-five base 10).
- ✓ Decimals are the numbers we use in everyday counting. We mostly use the decimal number system because we have ten fingers. The number 10 is made by using a combination of two of these decimal numbers: 1 and 0 while a number like 209 is a combination of three decimal numbers: 2, 0, and 9.

- ✓ There is no limit as to how many times the numbers can be reused, that's why it is often said that numbers are never ending.

- **Octal (Base 8)**

Digits from 0 – 7 are allowed. It should always have a prefix 0.

```
1. int x = 0146;
```

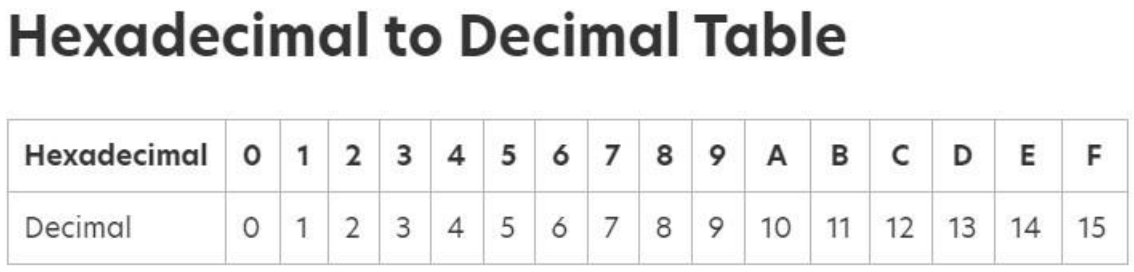
- **Hexa-Decimal (Base 16)**

Digits 0-9 are allowed and also characters from a-f are allowed in this form. Furthermore, both uppercase and lowercase characters can be used, ***Java provides an exception*** here.

1. int x = 0X123Face;

- ✓ A hexadecimal, which is also called base 16 or "hex" for short, is a representation of four binary bits and consists of sixteen numbers and letters. The numbers in a hex are the same as decimal numbers: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9. The big difference between a hex and a decimal is that a hex also contains letters. These letters are: A, B, C, D, E, F.

- ✓ A hex number can be represented using a subscript of 16 (i.e. 23516). These letters come after the decimals in ascending order. Therefore, the hexadecimal series looks like this: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F. A hex can be considered a shorter version of a decimal. For example a large number in decimal form has a much smaller hex equivalent (using less hex bits to represent the decimal number). I will demonstrate this later.



Hexadecimal to Decimal Table

Hexadecimal	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
Decimal	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

- **Binary**

A literal in this type should have a prefix 0b and 0B, from 1-7 one can also specify in binary literals, i.e. 0 and 1.

1. int x = 0b1111;

```
public class IntegralLiteral {
```

```
public static void main(String[] args)
{ int decimalValue = 123; // decimal-form literal int
octalValue = 01200; // octal-form literal int
hexaDecimalValue = 0xAce; // Hexa-decimal form literal
int binaryValue = 0b00101; // Binary literal

System.out.println("Decimal form literal is "+decimalValue);
System.out.println("Octal form literal is "+b);
System.out.println("Hexa-decimal form literal is "+hexaDecimalValue);
System.out.println("Binary literal is "+binaryValue);
}
}
```

Output-

```
Decimal form literal is 123

Octal form literal is 2766

Hexa-decimal form literal is 2766 Binary
literal is 5
```

We can specify explicitly as long type by suffixed with l or L but there is no way to specify byte and short, but if it's in the range the compiler automatically treats it as a byte.

2. **Floating-Point Literals in Java**

Here, datatypes can only be specified in decimal forms and not in octal or hexadecimal form.

Decimal (Base 10)

```
public class FloatingPointLiteral
{
public static void main(String args[])
{
double decimalValue = 101.230; // decimal-form literal double
decimalValue1 = 0123.222; // It also acts as decimal literal
double hexaDecimalValue = 1.234e2; // Hexa-decimal form
System.out.println("Decimal form literal is "+decimalValue);
System.out.println("Second Decimal form literal is "+decimalValue1);
System.out.println("Hexa decimal form literal is "+hexaDecimalValue);
}
}
```

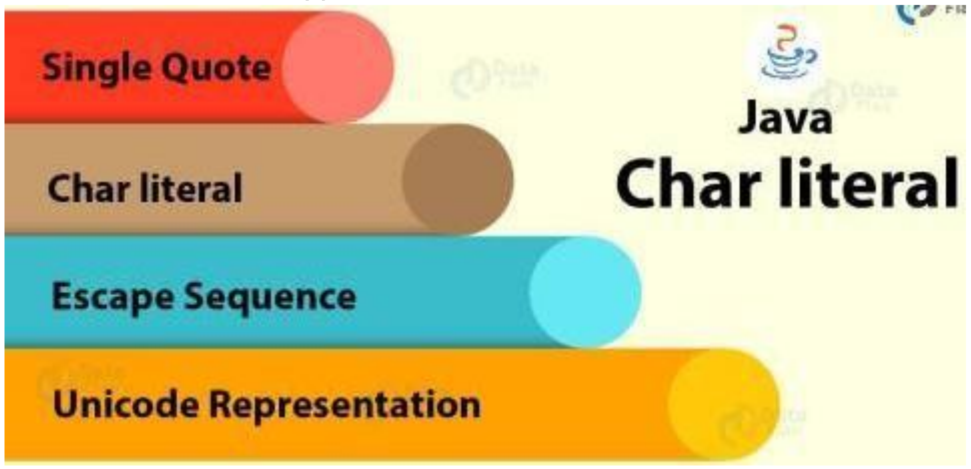
Output – Decimal form literal is 101.23

```
Second Decimal form literal is 123.222
Hexa decimal form literal is 123.4
```

Every floating type is a double type and this the reason why we cannot assign it directly to float variable, to escape this situation we use f or F as suffix, and for double we use d or D.

3. Char Literals in Java

These are the four types of char-



- **Single Quote**

Java Literal can be specified to a char data type as a single character within a single quote.

```
1. char ch = 'a';
```

- **Char as Integral**

A char literal in Java can specify as integral literal which also represents the Unicode value of a character.

Furthermore, an integer can specify in decimal, octal and even hexadecimal type, but the range is 0-65535.

```
1. char ch = 062;
```

- **Unicode Representation**

Char literals can specify in Unicode representation ‗\uxxxx‘. Here XXXX represents 4 hexadecimal numbers.

```
1. char ch = '\u0061';// Here /u0061 represent a.
```

- **Escape Sequence**

Escape sequences can also specify as char literal.

```
1. char ch = '\n';
```

Example

```
public class CharacterLiteral {
public static void main(String[] args)
{
char character = 'd';
//char number = 0789; error: Integer number too large
char unicodeCharacter = '\u0064';
System.out.println(character);
System.out.println(unicodeCharacter);
System.out.println("\" is a symbol");
}
}
```

Output:

```
d
d
" is a symbol
```

4. String Literals

Java String literals are any sequence of characters with a double quote.

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

They may not contain unescaped newline or linefeed characters.
However, the Java compiler will evaluate compile-time expressions.

```
public class StringLiteral {
public static void main(String[] args)
{
String myString = "Hello! Welcome to DataFlair";

// If we assign without "" then it treats as a variable
// and causes compiler error
// String myString1 = Hello;

System.out.println(myString );
}
}
```
Output –

```
Hello! Welcome to DataFlair
```

5. Boolean Literals

They allow only two values i.e. true and false.

1. boolean b = true;

Example:

```
public class BooleanLiteral {
public static void main(String[] args)
{
boolean boolVar1 = true;
boolean boolVar2 = false;
// boolean boolVar3 = 0; error: incompatible types: int cannot be converted to
boolean
// boolean boolVar1 = 1; error: incompatible types: int cannot be converted to
boolean
System.out.println(boolVar1);
System.out.println(boolVar2);
}
}
```

Output:

```
true
false
```

Operators in Java

An operator is a character that **represents an action**,or an operator is a symbol and it will tell to the computer to perform certain mathamtical and logical caluculations.

For example + is an arithmetic operator that represents addition.

Types of Operator in Java

1) Basic Arithmetic Operators
2) Assignment Operators

3) Auto-increment and Auto-decrement Operators
4) Logical Operators
5) Comparison (relational) operators
6) Bitwise Operators
7) Ternary Operator

1) Basic Arithmetic Operators

Basic arithmetic operators are: +, -, *, /, %

+ is for addition.
– is for subtraction.
***** is for multiplication.
/ is for division.
% is for modulo.
Note: Modulo operator returns remainder, for example 10 % 5 would return 0

Example of Arithmetic Operators

```
public class ArithmeticOperatorDemo {
  public static void main(String args[]) {
    int num1 = 100;
    int num2 = 20;

    System.out.println("num1 + num2: " + (num1 + num2) );
    System.out.println("num1 - num2: " + (num1 - num2) );
    System.out.println("num1 * num2: " + (num1 * num2) );
    System.out.println("num1 / num2: " + (num1 / num2) );
    System.out.println("num1 % num2: " + (num1 % num2) );
  }
}
```

Output:

```
num1 + num2: 120
num1 - num2: 80
num1 * num2: 2000
num1 / num2: 5
num1 % num2: 0
```

2) Assignment Operators

Assignments operators in java are: =, +=, -=, *=, /=, %=

num2 = num1 would assign value of variable num1 to the variable.
num2+=num1 is equal to num2 = num2+num1 **num2-=num1** is equal to num2 = num2-num1 **num2*=num1** is equal to num2 = num2*num1 **num2/=num1** is equal to num2 = num2/num1 **num2%=num1** is equal to num2 = num2%num1

Example of Assignment Operators

```
public class AssignmentOperatorDemo {    public static void
main(String args[]) {       int num1 = 10;       int num2 = 20;

      num2 = num1;
      System.out.println("= Output: "+num2);

      num2 += num1;
      System.out.println("+= Output: "+num2);

      num2 -= num1;
      System.out.println("-= Output: "+num2);

      num2 *= num1;
      System.out.println("*= Output: "+num2);

      num2 /= num1;
      System.out.println("/= Output: "+num2);

      num2 %= num1;
      System.out.println("%= Output: "+num2);
   }
}
```

Output:

```
= Output: 10
+= Output: 20
-= Output: 10
*= Output: 100
/= Output: 10
%= Output: 0
```

3) Auto-increment and Auto-decrement Operators
++ and --

num++ is equivalent to num=num+1;
num–– is equivalent to num=num-1;

Example of Auto-increment and Auto-decrement Operators

```
public class AutoOperatorDemo {
public static void main(String args[]){
int num1=100;       int num2=200;
num1++;       num2--;
     System.out.println("num1++ is: "+num1);
     System.out.println("num2-- is: "+num2);
  }
}
```

Output:

```
num1++ is: 101 num2--
is: 199
```

Pre & post increment operator behavior in Java

Increment operator

It is used to increment a value by 1. There are two varieties of increment operator:

- **Post-Increment :** Value is first used for computing the result and then incremented.
- **Pre-Increment :** Value is incremented first and then result is computed.

Decrement operator

It is used for decrementing the value by 1. There are two varieties of decrement operator.

- **Post-decrement :** Value is first used for computing the result and then decremented.
- **Pre-decrement :** Value is decremented first and then result is computed.

Pre increment example

```
// Java program to illustrate
// Increment and Decrement operators
// Can be applied to variables only

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

                System.out.println(―Value of b is‖ +b);
                System.out.println(―Value of a is‖+a);
        }
}
```

```
Value of b is11

Value of a is11
```

Post increment example

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

               System.out.println(—Value of b is‖ +b);
                 System.out.println(—Value of a is‖ +a);
        }
}
```

```
Value of b is10
Value of a is11
```

Pre decrement example

```
public class Test {

        public static void main(String[] args)
        {
                int a = 10;
                int b = --a;

                System.out.println(—Value of b is‖ +b);
                System.out.println(—Value of a is‖ +a);
        }
}
```

```
Value of b is9
Value of a is9
```

Post decrement example

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

                System.out.println(—Value of b is‖ +b);
                System.out.println(—Value of a is‖ +a);

        }
}
```

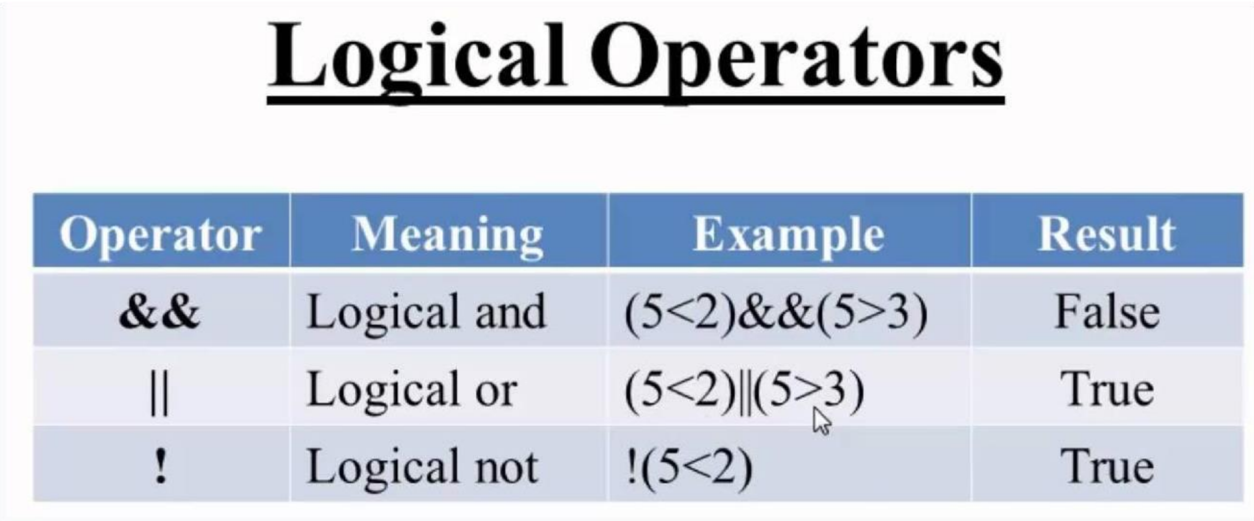
```
Value of b is 10
Value of a is 9
```

4) Logical Operators

Logical Operators are used with binary variables. They are mainly used in conditional statements and loops for evaluating a condition.

Logical operators in java are: &&, ||, !

The logical operators || (conditional-OR) and && (conditional-AND) operate on boolean expressions. Here's how they work.



Logical Operators

Operator	Meaning	Example	Result
&&	Logical and	(5<2)&&(5>3)	False
||	Logical or	(5<2)||(5>3)	True
!	Logical not	!(5<2)	True

Operator	Description	Example
||	**conditional-OR:** true if either of the boolean expression is true.	false || true is evaluated to true
&&	**conditional-AND**: true if all boolean expressions are true	false && true is evaluated to false

Let's say we have two boolean variables b1 and b2.

b1&&b2 will return true if both b1 and b2 are true else it would return false.

b1||b2 will return false if both b1 and b2 are false else it would return true.

!b1 would return the opposite of b1, that means it would be true if b1 is false and it would return false if b1 is true.

Example:

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

       int number1 = 1, number2 = 2, number3 = 9;
boolean result;

       // At least one expression needs to be true for the result to be true
result = (number1 > number2) || (number3 > number1);

       // result will be true because (number1 > number2) is true
       System.out.println(result);

       // All expression must be true from result to be true
       result = (number1 > number2) && (number3 > number1);

       // result will be false because (number3 > number1) is false
       System.out.println(result);
   }
}
```

Output:

```
true
false
```

Note: Logical operators are used in decision making and looping.

Example of Logical Operators

```
public class LogicalOperatorDemo {
   public static void main(String args[]) {
      boolean b1 = true;
      boolean b2 = false;

      System.out.println("b1 && b2: " + (b1&&b2));
      System.out.println("b1 || b2: " + (b1||b2));
      System.out.println("!(b1 && b2): " + !(b1&&b2));
   }
}
```

Output:

```
b1 && b2: false
b1 || b2: true
!(b1 && b2): true
```

5) Comparison(Relational) operators

We have six relational operators in Java: ==, !=, >, <, >=, <=
== returns true if both the left side and right side are equal **!=**
returns true if left side is not equal to the right side of operator.
> returns true if left side is greater than right.
< returns true if left side is less than right side.
>= returns true if left side is greater than or equal to right side.
<= returns true if left side is less than or equal to right side.

Example of Relational operators
Note: This example is using if-else statement which is our next tutorial, if you are finding it difficult to understand then refer if-else in Java.

```
public class RelationalOperatorDemo {
public static void main(String args[]) {
int num1 = 10;
```

```
    int num2 = 50;      if
(num1==num2) {
          System.out.println("num1 and num2 are equal");
    }
else{
          System.out.println("num1 and num2 are not equal");
    }

    if( num1 != num2 ){
          System.out.println("num1 and num2 are not equal");
    }
else{
          System.out.println("num1 and num2 are equal");
    }

    if( num1 > num2 ){
          System.out.println("num1 is greater than num2");
    }
else{
          System.out.println("num1 is not greater than num2");
    }

    if( num1 >= num2 ){
           System.out.println("num1 is greater than or equal to num2");
    }
else{
          System.out.println("num1 is less than num2");
    }

    if( num1 < num2 ){
          System.out.println("num1 is less than num2");
    }
else{
          System.out.println("num1 is not less than num2");
    }

    if( num1 <= num2){
          System.out.println("num1 is less than or equal to num2");
    }
else{
          System.out.println("num1 is greater than num2");
    }
  }
}
```

Output:

```
num1 and num2 are not equal
num1 and num2 are not equal
num1 is not greater than num2
num1 is less than num2 num1
is less than num2
num1 is less than or equal to num2
```

Example

```
public class Test {

  public static void main(String args[]) {
     int a = 10;
     int b = 20;

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

Output

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

6) Bitwise Operators

There are six bitwise Operators: &, |, ^, ~, <<, >>
num1 = 11; /* equal to 00001011*/ num2
= 22; /* equal to 00010110 */
Bitwise operator performs bit by bit processing.

num1 & num2 compares corresponding bits of num1 and num2 and generates 1 if both bits are equal, else it returns 0. In our case it would return: 2 which is 00000010 because in the binary form of num1 and num2 only second last bits are matching.

num1 | num2 compares corresponding bits of num1 and num2 and generates 1 if either bit is 1, else it returns 0. In our case it would return 31 which is 00011111

num1 ^ num2 compares corresponding bits of num1 and num2 and generates 1 if they are not equal, else it returns 0. In our example it would return 29 which is equivalent to 00011101

~num1 is a complement operator that just changes the bit from 0 to 1 and 1 to 0. In our example it would return -12 which is signed 8 bit equivalent to 11110100

num1 << 2 is left shift operator that moves the bits to the left, discards the far left bit, and assigns the rightmost bit a value of 0. In our case output is 44 which is equivalent to 00101100

Note: In the example below we are providing 2 at the right side of this shift operator that is the reason bits are moving two places to the left side. We can change this number and bits would be moved by the number of bits specified on the right side of the operator. Same applies to the right side operator.

num1 >> 2 is right shift operator that moves the bits to the right, discards the far right bit, and assigns the leftmost bit a value of 0. In our case output is 2 which is equivalent to 00000010

Example of Bitwise Operators

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

    int num1 = 11;  /* 11 = 00001011 */
int num2 = 22;  /* 22 = 00010110 */
int result = 0;

    result = num1 & num2;
    System.out.println("num1 & num2: "+result);

    result = num1 | num2;
    System.out.println("num1 | num2: "+result);

    result = num1 ^ num2;
    System.out.println("num1 ^ num2: "+result);

    result = ~num1;
    System.out.println("~num1: "+result);

    result = num1 << 2;
    System.out.println("num1 << 2: "+result); result = num1 >> 2;
    System.out.println("num1 >> 2: "+result);
  }
}
```

Output:

```
num1 & num2: 2 num1
| num2: 31 num1 ^
num2: 29 ~num1: -12
num1 << 2: 44 num1 >> 2: 2
```

Bitwise operators Examples

Bitwise operators are used to perform manipulation of individual bits of a number. They can be used with any of the integral types (char, short, int, etc). They are used when performing update and query operations of Binary indexed tree.

Bitwise OR (|)

This operator is binary operator, denoted by ‗|‘. It returns bit by bit OR of input values, i.e, if either of the bits is 1, it gives 1, else it gives 0.
For example,

```
a = 5 = 0101 (In Binary)
b = 7 = 0111 (In Binary)

Bitwise OR Operation of 5 and 7
  0101
| 0111
 ________
  0111  = 7 (In decimal)
```

Bitwise AND (&)
This operator is binary operator, denoted by '&'. It returns bit by bit AND of input values, i.e, if both bits are 1, it gives 1, else it gives 0.

For example,

```
a = 5 = 0101 (In Binary)
b = 7 = 0111 (In Binary)
Bitwise AND Operation of 5 and 7
  0101
& 0111
 ________
  0101  = 5 (In decimal)
```

Bitwise XOR (^)

This operator is binary operator, denoted by '^'. It returns bit by bit XOR of input values, i.e, if corresponding bits are different, it gives 1, else it gives 0.

For example

```
a = 5 = 0101 (In Binary)
b = 7 = 0111 (In Binary)

Bitwise XOR Operation of 5 and 7
  0101
^ 0111
 ________
  0010  = 2 (In decimal)
```

Bitwise Complement (~)

This operator is unary operator, denoted by '~'. It returns the one's compliment representation of the input value, i.e, with all bits inversed, means it makes every 0 to 1, and every 1 to 0.

For example

```
a = 5 = 0101 (In Binary)

Bitwise Compliment Operation of 5

~ 0101

  1010  = 10 (In decimal)
```

Note – **Compiler will give 2's complement of that number, i.e., 2's compliment of 10 will be -6.**

```
// Java program to illustrate
// bitwise operators
public class operators {
        public static void main(String[] args)
        {
                //Initial values
                int a = 5;
                int b = 7;

                // bitwise and
                // 0101 & 0111=0101 = 5
                System.out.println("a&b = " + (a & b));

                // bitwise or
                // 0101 | 0111=0111 = 7
                System.out.println("a|b = " + (a | b));

                // bitwise xor
                // 0101 ^ 0111=0010 = 2
                System.out.println("a^b = " + (a ^ b));

                // bitwise and
                // ~0101=1010
                // will give 2's complement of 1010 = -6
                System.out.println("~a = " + ~a);

                // can also be combined with
```

Output :

```
a&b = 5

a|b = 7

a^b = 2

~a = -6

a= 5
```

```
                // assignment operator to provide shorthand
                // assignment
                // a=a&b
                a &= b;
                System.out.println("a= " + a);
        }
}
```

Assume integer variable A holds 60 and variable B holds 13 then

Operator	Description	Example
& (bitwise and)	Binary AND Operator copies a bit to the result if it exists in both operands.	(A & B) will give 12 which is 0000 1100
| (bitwise or)	Binary OR Operator copies a bit if it exists in either operand.	(A | B) will give 61 which is 0011 1101
^ (bitwise XOR)	Binary XOR Operator copies the bit if it is set in one operand but not both.	(A ^ B) will give 49 which is 0011 0001
~ (bitwise compliment)	Binary Ones Complement Operator is unary and has the effect of 'flipping' bits.	(~A) will give -61 which is 1100 0011 in 2's complement form due to a signed binary number.
<< (left shift)	Binary Left Shift Operator. The left operands value is moved left by the number of bits specified by the right operand.	A << 2 will give 240 which is 1111 0000
>> (right shift)	Binary Right Shift Operator. The left operands value is moved right by the number of bits specified by the right operand.	A >> 2 will give 15 which is 1111
>>> (zero fill right shift)	Shift right zero fill operator. The left operands value is moved right by the number of bits specified by the right operand and shifted values are filled up with zeros.	A >>>2 will give 15 which is 0000 1111

7) Ternary Operator

This operator evaluates a boolean expression and assign the value based on the result.

Syntax:

EXP1?EXP2:EXP3;

OR

```
variable num1 = (expression) ? value if true : value if false
```

If the expression results true then the first value before the colon (:) is assigned to the variable num1 else the second value is assigned to the num1.

Example of Ternary Operator

```java
public class TernaryOperatorDemo {

   public static void main(String args[]) {
       int num1, num2;
       num1 = 25;
       /* num1 is not equal to 10 that's why
           * the second value after colon is assigned
           * to the variable num2
           */
          num2 = (num1 == 10) ? 100: 200;
          System.out.println( "num2: "+num2);

          /* num1 is equal to 25 that's why
           * the first value is assigned
           * to the variable num2
           */
          num2 = (num1 == 25) ? 100: 200;
          System.out.println( "num2: "+num2);
   }
}
```

Output:

```
num2: 200
num2: 100
```

Operator Precedence in Java

This determines which operator needs to be evaluated first if an expression has more than one operator. Operator with higher precedence at the top and lower precedence at the bottom.

Unary Operators
++ − − ! ~
Multiplicative
* / %
Additive
+ −
Shift
<< >> >>>
Relational
> >= < <=
Equality
== !=
Bitwise AND
&
Bitwise XOR
^
Bitwise OR
|
Logical AND
&&
Logical OR
||
Ternary
?:
Assignment
= += -= *= /= %= > >= < <= &= ^= |=

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

- **-** :Unary minus, used for negating the values.
- + :Unary plus, used for giving positive values. Only used when deliberately converting a negative value to positive.
- **++** :Increment operator, used for incrementing the value by 1. There are two varieties of increment operator.
  - ✓ Post-Increment : Value is first used for computing the result and then incremented.
  - ✓ Pre-Increment : Value is incremented first and then result is computed.
- **--**: Decrement operator, used for decrementing the value by 1. There are two varieties of decrement operator.
  - ✓ Post-decrement : Value is first used for computing the result and then decremented.

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

Examples for unary operators:

- ✓ a+ , +a, -a, a-, ++a ,--a, a++, a—
- ✓ (in previous topic we covered incriment and decrement operators with example. Please refer that section)

```
// Java program to illustrate unary
operators  public class operators {      public
static void main(String[] args)
   {
       int a = 20, b = 10, c = 0, d = 20, e = 40, f = 30;
boolean condition = true;

       // pre-increment operator
// a = a+1 and then c = a;
       c = ++a;
       System.out.println("Value of c (++a) = " + c);

       // post increment operator
       // c=b then b=b+1
       c = b++;
       System.out.println("Value of c (b++) = " + c);

       // pre-decrement operator
       // d=d-1 then c=d
       c = --d;
       System.out.println("Value of c (--d) = " + c);

       // post-decrement operator
       // c=e then e=e-1
       c = e--;
       System.out.println("Value of c (e--) = " + c);

       // Logical not operator
       System.out.println("Value of !condition ="
+ !condition);
   }
}
```

Output:

```
Value of c (++a) = 21
Value of c (b++) = 10
Value of c (--d) = 19
Value of c (e--) = 40
Value of !condition =false
```

Binary operators:

Binary operators are those **operators** that work with two operands. For example, a common **binary** expression would be a + b

The binary operators are categorized as follows:

- Multiplicative operators: multiplication (*), remainder (%), and division (/)
- Additive operators: addition (+) and subtraction (-)
- Shift operators: left shift (<<) and right shift (>>)
- Relational operators: less than (<), less than or equal to (<=), greater than (>), and greater than or equal to (>=)
- Equality operators: equality (==) and inequality (!=)
- Bitwise operators: AND (&), OR (|), and XOR (^)
- Logical operators: AND (&&) and OR (||)

We have already covered above arthamatic operators in previpus section.

Java Bitwise Operators are also binary operators

Java defines several bitwise operators, which can be applied to the integer types, long, int, short, char, and byte.

Bitwise operator works on bits and performs the bit-by-bit operation. Assume if a = 60 and b = 13; now in binary format they will be as follows −

```
a = 0011 1100 b =
0000 1101 -------
---------- a&b =
0000 1100

a|b = 0011 1101
 a^b = 0011
0001

~a = 1100 0011
```

Java Bitwise Operators are clearly explained in the previous section

Java Expressions

A Java expression consists of variables, operators, literals, and method calls.

An expression is a construct made up of variables, operators, and method invocations, which are constructed according to the syntax of the language that evaluates to a singlevalue.

For example

```
int score;  score = 90;
```

Here, `score = 90` is an expression that returns an `int`. Consider another example,

```
Double a = 2.2, b = 3.4, result; result = a + b -
3.4;
```

Here, `a + b - 3.4` is an expression.

```
if (number1 == number2)
    System.out.println("Number 1 is larger than number 2");
```

Here, `number1 == number2` is an expression that returns a boolean value.
Similarly, `"Number 1 is larger than number 2"` is a string expression.

Primitive Type conversion and casting

Programming is playing around with data. In Java, there are many data types. Most of the times while coding, it is necessary to change the type of data to understand the processing of a variable and this is called Type Casting.

Type casting is nothing but assigning a value of one primitive data type to another. When you assign the value of one data type to another, you should be aware of the compatibility of the data type. If they are compatible, then Java will perform the conversion automatically known as *Automatic Type Conversion* and if not, then they need to be casted or converted explicitly.

There are two types of casting in Java as follows:

- **Widening Casting (automatically)** – This involves the conversion of a smaller data type to the larger type size.

  byte -> short -> char -> int -> long -> float -> double

- **Narrowing Casting (manually)** – This involves converting a larger data type to a smaller size type.

  double -> float -> long -> int -> char -> short -> byte

Widening Casting
This type of casting takes place when two data types are automatically converted. It is also known as Implicit Conversion. This happens when the two data types are compatible and also when we assign the value of a smaller data type to a larger data type.

For Example, 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. Now let's write a logic for Implicit type casting to understand how it works.

```
public class Conversion{
public static void main(String[] args)
{
int i = 200;

//automatic type conversion
long l = i;

//automatic type conversion
float f = l;
System.out.println("Int value "+i);
System.out.println("Long value "+l);
System.out.println("Float value "+f);
}
}
Output:
Int value 200
Long value 200
Float value 200.0
```

Example: Converting int to double

```
class Main {   public static void
main(String[] args) {
   //  create  int  type  variable
int num = 10;
   System.out.println("The integer value: " + num);

   // convert into double type
double data = num;
   System.out.println("The double value: " + data);
}
}
```

Output

```
The integer value: 10

The double value: 10.0
```

In the above example, we are assigning the int type variable named num to a double type variable named data.

Here, the Java first converts the int type data into the double type. And then assign it to the double variable.

In the case of **Widening Type Casting**, the lower data type (having smaller size) is converted into the higher data type (having larger size). Hence there is no loss in data. This is why this type of conversion happens automatically.

Note: This is also known as **Implicit Type Casting**.

Narrowing Casting

In this case, if you want to assign a value of larger data type to a smaller data type, you can perform *Explicit type casting* or narrowing. This is useful for incompatible data types where automatic conversion cannot be done.

```
/Java program to illustrate explicit type conversion public
class Narrowing
{
public static void main(String[] args)
{
double d = 200.06;
//explicit type casting
long l = (long)d;

//explicit type casting
int i = (int)l;
System.out.println("Double Data type value "+d);

//fractional part lost
System.out.println("Long Data type value "+l);

//fractional part lost
System.out.println("Int Data type value "+i);
}
}
```

Output:

```
Double Data type value 200.06
Long Data type value 200
Int Data type value 200
```

Now that you know how to perform Explicit type casting, let�really move further and understand how explicit casting can be performed on Java expressions.

Narrowing Type Casting Example:

In **Narrowing Type Casting**, we manually convert one data type into another using the parenthesis.

Example: Converting double into an int

```
class Main {   public static void
main(String[] args) {
    // create double type variable
    double num = 10.99;
    System.out.println("The double value: " + num);

    // convert into int type
int data = (int)num;
    System.out.println("The integer value: " + data);
}
}
```

Output

```
The double value: 10.99
The integer value: 10
```

Control statements.

Control Statements in Java is one of the fundamentals required for Java Programming. It allows the smooth flow of a program.
:

Decision Making Statements

- Simple if statement
- if-else statement
- Nested if statement
- Switch statement

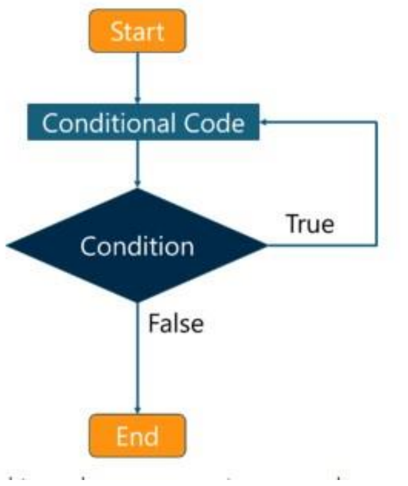
Looping statements

- While
- Do-while For
- For-Each

Branching statements

- Break
- Continue

Every programmer is familiar with the term statement, which can simply be defined as an instruction given to the computer to perform specific operations. A control statement in java is a statement that determines whether the other statements will be executed or not. It controls the flow of a program. An ‗if' statement in java determines the sequence of execution between a set of two statements.



Control Statements can be divided into three categories, namely

- Selection statements
- Iteration statements
- Jump statements

Decision-Making Statements

Statements that determine which statement to execute and when are known as decision-making statements. The flow of the execution of the program is controlled by the control flow statement.

There are four decision-making statements available in java.

Simple if statement

The if statement determines whether a code should be executed based on the specified condition.

Syntax:

```
if (condition) {
Statement 1; //executed if condition is true
}
Statement 2; //executed irrespective of the condition
```

```
public class Main
{
public static void main(String args[])
{ int a =
15; if (a >
20)
{
System.out.println("a is greater than 10");
}
System.out.println(—This statement is executed  afternif");
}
}
}
```

Output:

```
This statement is executed  after
```

If..else statement

In this statement, if the condition specified is true, the if block is executed. Otherwise, the else block is executed.

Syntax:

```
if (condition) {
Statement 1; //executed if condition is true
}
Else
{
Statement 2; //executed if condition is false

}
Statement 2; //executed irrespective
```

Example:

```
public class Main
{ public static void main(String
args[])
{ int a =
15; if (a >
20)
System.out.println("a is greater than 10");
else
System.out.println("a is less than 10");
System.out.println("Hello World!");
}
}
}
```

Output:

```
a is less than 10 Hello World!
```

Nested if statement

An if present inside an if block is known as a nested if block. It is similar to an if..else statement, except they are defined inside another if..else statement.

Syntax:

```
if (condition1) {
Statement 1; //executed if first condition is true if
(condition2) {
Statement 2; //executed if second condition is true
}
else
{
Statement 3; //executed if second condition is false
}
}
```

Example:

```
public class Main
{ public static void main(String
args[])
{ int s =
18; if (s >
10)
{
        if (s%2==0)
        System.out.println("s is an even number and greater than 10!"); else
        System.out.println("s is a odd number and greater than 10!");
} else
{
System.out.println("s is less than 10");
}


System.out.println("Hello World!");
}
}
```

Output:

```
s is an even number and greater than 10!
Hello World!
```

Switch statement

A switch statement in java is used to execute a single statement from multiple conditions. The switch statement can be used with short, byte, int, long, enum types, etc.

Certain points must be noted while using the switch statement:

ɑ One or N number of case values can be specified for a switch expression. ɑ Case values that are duplicate are not permissible. A compile-time error is generated by the compiler if unique values are not used. ɑ The case value must be literal or constant. Variables are not permissible. ɑ Usage of break statement is made to terminate the statement sequence. It is optional to use this statement. If this statement is not specified, the next case is executed.

Example:

```
public class Music { public static
void main(String[] args)
{ int instrument =
4;
String musicInstrument;
// switch statement with int data type
switch (instrument) { case 1:
musicInstrument = "Guitar"; break;
case 2: musicInstrument = "Piano";
break; case 3: musicInstrument =
"Drums";
break; case 4:
musicInstrument =
"Flute"; break; case 5:
musicInstrument = "Ukelele";
```

break;

```
case 6: musicInstrument =
"Violin"; break; case 7:
musicInstrument = "Trumpet";
break; default:
musicInstrument = "Invalid";
break;
}
System.out.println(musicInstrument); }
```

```
} Output:
Flute
```

Looping Statements

Statements that execute a block of code repeatedly until a specified condition is met are known as looping statements. Java provides the user with three types of loops:

While

It is Known as the most common loop, the while loop evaluates a certain condition. If the condition is true, the code is executed. This process is continued until the specified condition turns out to be false.

The condition to be specified in the while loop must be a Boolean expression. An error will be generated if the type used is int or a string. **Syntax:**

```
while (condition)
{
statementOne;
}
```

Example:

```
public class whileTest
{
public static void main(String args[])
{ int i = 5;
while (i <=
15)
{
System.out.println(i); i
= i+2;
}
}
} Output:
5
7
```

```
9
11
13
15
```

Do..while

The do-while loop is similar to the while loop, the only difference being that the condition in the do-while loop is evaluated after the execution of the loop body. This guarantees that the loop is executed at least once.

Syntax:

```
do{
//code to be executed
}while(condition);
```

Example:

```
public class Main
{
public static void main(String args[])
{  int i =
20;
do
{
System.out.println(i);
i = i+1;
} while (i <= 20);
}
}
&nbsp;
Output:
20
```

```
public class whileTest
{
public static void main(String args[])
{ int i =
5; do
{
System.out.println(i); i
= i+2;
} while (i <= 15);
}
}
Output:
5  7 9 11 13 15
```

For

The for loop in java is used to iterate and evaluate a code multiple times. When the number of iterations is known by the user, it is recommended to use the for loop.

Syntax:

```
for (initialization; condition; increment/decrement)
{
statement; }
```

Example:

```
public class forLoop
{
public static void main(String args[])
{
for (int i = 1; i <= 10; i++)
System.out.println(i);
}
}
Output:
5
6
7
8
9
10
```

```
public class whileTest
{
public static void main(String args[])
{
for (i=5;i <= 15;i=i+2)
{
System.out.println(i);
} }}
Output:5
7
9
11
13
15
```

For-Each

The traversal of elements in an array can be done by the for-each loop. The elements present in the array are returned one by one. It must be noted that the user does not have to increment the value in the for-each loop.

Example:

```
public class foreachLoop{ public static
void main(String args[]){ int s[] =
{18,25,28,29,30}; for (int i : s) {
System.out.println(i);
}
}
} Output:
18
25
28
29
30
```

Branching Statements

Branching statements in java are used to jump from a statement to another statement, thereby the transferring the flow of execution. **Continue**

To jump to the next iteration of the loop, we make use of the continue statement. This statement continues the current flow of the program and skips a part of the code at the specified condition.

Example:

```
public class Main
{
public static void main(String args[])
{
for (int i = 1; i < 15; i++)
{ if (i ==
8)
continue;
System.out.println(i);
}}}
```

Output:

```
1
2
3
4
5
6
7
9
10
11
12
13
14
```

```
public class Test
{
public static void main(String args[])
{
for (int i = 5; i < 10; i++)
{ if (i ==
8) break;
System.out.println(i);
}
}
}
Output:
5
6
7
```

Break

The break statement in java is used to terminate a loop and break the current flow of the program.

```
public class Main
{
public static void main(String args[])
{
for (int i = 1; i < 15; i++)
{ if (i ==
8) break;
System.out.println(i);
}
}
}
```

Output:

```
1
2
3
4
5
6
7
```

Example:

```
public class Main
{ public static void main(String
args[])
{ for (int k = 5; k < 15;
k++)
{
// Odd numbers are skipped
if (k%2 != 0) continue;
// Even numbers are printed
System.out.print(k + " ");
}
}
}
```

Output:

```
6 8 10 12 14
```

With this, we come to an end of this Control Statements in Java . The control statements in java must be used efficiently to make the program effective and user-friendly.