Primitive Data Types

- Primitive data types can be classified in four groups:

1) Integers :

- This group includes **byte**, **short**, **int**, and **long**.
- All of these are signed, **positive** and **negative** values.

DataType	Size	Example
byte	8-bit	byte b, c;
short	16-bit	short b,c;
int	32-bit	int b,c;
long	64-bit	long b,c;

2) Floating-point :

- This group includes **float** and **double**, which represent numbers with fractional precision.

DataType	Size	Example
float	32 bits	float a,b;
double	64 bits	double pi;

3) Characters :

- This group includes **char**, which represents symbols in a character set, like letters and numbers.
- Java **char** is a 16-bit type. The range of a **char** is 0 to 65,536. There are no negative **char**s.

  For example : char name = 'x';

4) Boolean :

- This group includes **boolean**, which is a special type for representing true/false values.
- It can have only one of two possible values, **true** or **false**.

  For example : boolean b = true;

Example:

```
class datatypeDemo
{
        public static void main(String args[])
        {
                int r=10;
                float pi=3.14f,a;              //You can also use here double data type
                char ch1=97,ch2='A' ;
                boolean x=true;
                a = pi*r*r;
                System.out.println("Areat of Circle is :: "+a);
                System.out.println("Ch1 and Ch2 are :: "+ch1+" "+ch2);
                System.out.println("Value of X is :: "+x);
}
```

```
}
```

Output:

```
Area of Circle is :: 314.0
Ch1 and Ch2 are :: a A
Value of X is :: true
```

User Defined Data Type

Class

- A **class** is a template that specifies the attributes and behavior of things or objects.
- A **class** is a blueprint or prototype from which creates as many desired objects as required.

Example:

```
class Box
{
        double width=1;
        double height=2;
        double depth=3;
        void volume()
        {
                System.out.print("Volume is ");
                System.out.println(width * height * depth);
        }
}
class demo
{
        public static void main(String args[])
        {
                Box  b1 = new Box();
                b1.volume();
        }
}
```

Interface

- An **interface** is a collection of abstract methods.
- A **class** implements an interface, thereby inheriting the abstract methods of the interface.
- An **interface** is not a class. A class describes the attributes and behaviors of an object. An interface contains behaviors that a class implements.
- Unless the class that implements the interface is abstract, all the methods of the interface need to be defined in the class.

Example:

```
interface  Animal
{
     public void eat();
     public void travel();
}
```

Identifiers and Literals

Identifiers

- Identifiers are used for class names, method names, and variable names.
- An identifier may be any descriptive sequence of **uppercase and lowercase letters, numbers, or the underscore characters and dollar-sign characters.**
- There are some rules to define identifiers as given below:
  1) Identifiers must start with a **letter or underscore** (_).
  2) Identifiers cannot start with a **number**.
  3) **White space(blank, tab, newline)** are not allowed.
  4) You can't use a **Java keyword** as an identifier.
  5) Identifiers in Java are **case-sensitive**; **foo** and **Foo** are two different identifiers.

Examples :

Valid Identifiers	Not Valid Identifiers
AvgNumber	2number
A1	int
$hello	-hello
First_Name	First-Name

Literals (Constants)

- A constant value in a program is denoted by a **literal**. Literals represent numerical (integer or floating-point), character, boolean or string values.

Integer	Floating-point	Character	Boolean	String
33, 0, -19	0.3, 3.14	(' 'R' 'r' '{'	(predefined values) true, false	“language”,”0.2” ,“r”

Variables

- A variable is defined by the combination of an **identifier**, a **type**, and an optional **initializer**.
- All variables have a scope, which is defines their visibility and lifetime.

Declaring of variable

- All variables must be declared before they can be used.

  Syntax:

  type identifier [= value][, identifier [= value] ...];

  - The **type** is one of Java’s **atomic types, or the name of a class or interface**.
  - The **identifier** is the name of the **variable**.
  - You can **initialize** the variable by specifying an **equal sign and a value**.
  - To declare **more than one variable** of the specified type, use a comma separated list.

Example: int a, b, c = 10;

Dynamic Initialization

- Java allows variables to be initialized dynamically by using any valid expression at the time the **variable is declared**.

Example:

```
int  a=2, b=3;        // Constants as initializer
int  c = a+b;         // Dynamic initialization
```

Scope of Variables

- Java allows variables to be declared within any **block**.
- A **block** is begun with an opening curly brace and ended by a closing curly brace.
- A block defines a scope. Thus, each time you start a new block, you are creating a new scope.
- A **scope** determines which **objects** are visible to other parts of your program.
- Java defines two general categories of scopes: **global** and **local**.
- Variables declared inside a scope is not **visible** to code that is defined outside that scope.
- Thus, when you declare a variable within a scope, then you can access it within that scope only and protecting it from **unauthorized access**.
- Scopes can be **nested**. So, outer scope encloses the inner scope. This means that objects declared in the **outer** scope will be **visible** to code within the **inner** scope.
- However, the reverse is not true. **Objects** declared within **inner** scope will not be **visible outside** it.

Example:

```
class  scopeDemo
{
        public static void main(String args[])
        {
                int  x;            // visible to all code within main
                x = 10;
                if(x == 10) // start new scope
                {
                        int  y = 20;      // Visible only to this block
                                // x and y both are visible here.
                        System.out.println("x and y: " + x + " " + y);
                        x = y * 2;
                }
                // y = 100;                    // Error! y not visible here
                // x is still visible here.
                System.out.println("x is " + x);
        }

}
```

Default values of variables declared

- If you are not assigning value, then Java runtime assigns default value to variable and when you try to access the variable you get the default value of that variable.
- Following table shows variables types and their default values:

Data type	Default value
boolean	FALSE
char	\u0000
int,short,byte / long	0 / 0L
float /double	0.0f / 0.0d
any reference type	null

- Here, char primitive default value is \u0000, which means blank/space character.
- When you declare any local/block variable, they didn't get the default values.

Type Conversion and Casting

- It assigns a value of one type variable to a variable of another type. If the two types are **compatible**, then Java will perform the conversion automatically (Implicit conversion).
- **For example**, it is always possible to assign an **int** value to a **long** variable. However, not all types are compatible, and thus, not all type conversions are implicitly allowed.
- There is no automatic conversion defined from **double** to **byte**. Fortunately, it is possible conversion between **incompatible** types. For that you must perform type casting operation, which performs an **explicit conversion** between incompatible types.

Implicit Type Conversion (Widening Conversion)

- When one type of data is assigned to another type of variable, an automatic type conversion will take place if the following two conditions are met:
  1. **Two types are compatible.**
  2. **Destination type is larger than the source type.**
- When these two conditions are met, a **widening conversion** takes place.
- **For example**, the **int** type is always large enough to hold all valid **byte** values, so no explicit cast statement is required.
- **Following table shows compatibility of numeric data type:**

Status	Integer	Floating-Point	Char	Boolean
Compatible	✓	✓		
Incompatible			✓	✓

- Also, **Boolean** and **char** are not compatible with each other.

Explicit Type Conversion (Narrowing Conversion)

- Although the automatic type conversions are helpful, they will not fulfill all needs. **For example**, if we want to assign an **int** value to a **byte** variable then conversion will not be performed automatically, because a **byte** is smaller than an **int**.
- This kind of conversion is sometimes called a **narrowing** conversion.
- For, this type of conversion we need to make the value narrower explicitly. so that, it will fit into the target data type.
- To create a conversion between **two incompatible** types, you must use a **cast**.
- A cast is simply an explicit type conversion.

  Synatax: **(target-type) value**
- Here, **target-type** specifies the desired type to convert the specified value to.
- **For example**, the following casts an **int** to a **byte**.

```
int  a;
byte  b;
b = (byte) a;
```

- A different type of conversion will occur when a floating-point value is assigned to an integer type: **truncation**.
- As we know, integers do not have fractional components. Thus, when a floating-point value is assigned to an integer type, the fractional component is lost.
- **For example**, if the value 1.23 is assigned to an integer, the resulting value will simply be 1. The 0.23 will have been truncated.

Example:

```
class  conversionDemo
{
        public static void main(String args[])
        {
                byte b;
                int i = 257;
                double d = 323.142;

                System.out.println("\nConversion of int to byte.");
                b = (byte) i;
                System.out.println("i and b " + i + " " + b);

                System.out.println("\nConversion of double to int.");
                i = (int) d;
                System.out.println("d and i " + d + " " + i);

                System.out.println("\nConversion of double to byte.");
                b = (byte) d;
```

```
                System.out.println("d and b " + d + " " + b);
            }
        }
```

Output:

```
Conversion of int to byte.
i and b 257   1
Conversion of double to int.
d and i 323.142   323
Conversion of double to byte.
d and b 323.142   67
```

- When the value 257 is cast into a **byte** variable, the result is the remainder of the division of 257 by 256 (the range of a **byte**), which is 1 in this case.
- When the **d** is converted to an **int**, its fractional component is lost.
- When **d** is converted to a **byte**, its fractional component is lost, and the value is reduced modulo 256, which in this case is 67.

Wrapper Class

- **Wrapper** class wraps (encloses) around a data type and gives it an **object** appearance.
- Wrapper classes are used to convert any data type into an **object**.
- The primitive data types are not objects and they do not belong to any class.
- So, sometimes it is required to convert data types into objects in java.
- Wrapper classes include methods to unwrap the object and give back the data type.

  Example:

```
int  k = 100;
Integer  it1 = new Integer(k);
```

  - The **int** data type **k** is converted into an object, **it1** using **Integer** class.
  - The **it1** object can be used wherever **k** is required an object.
  - To unwrap (getting back int from Integer object) the object **it1**.

  Example:

```
int  m = it1.intValue();
System.out.println(m*m);       // prints 10000
```

  - **intValue()** is a method of **Integer** class that returns an **int** data type.
- Eight wrapper classes exist in **java.lang** package that represent 8 data types:

Primitive Data Type	Wrapper Class	Unwrap Methods
byte	Byte	byteValue()
short	Short	shortValue()
int	Integer	intValue()
long	Long	longValue()
float	Float	floatValue()
double	Double	doubleValue()

char	Character	charValue()
boolean	Boolean	booleanValue()

- There are mainly two uses with wrapper classes.
  1) To convert **simple data types** into **objects**.
  2) To convert **strings** into **data types** (known as parsing operations), here methods of type **parseX()** are used. (Ex. parseInt())
- The **wrapper classes** also provide methods which can be used to convert a **String** to any of the **primitive data types**, except **character**.
- These methods have the format **parsex()** where **x** refers to any of **the primitive data types** except **char**.
- To convert any of the primitive data type value to **a String**, we use the **valueOf()** methods of the String class.

  Example:

```
int x =  Integer.parseInt("34");          // x=34
double y = Double.parseDouble("34.7");      // y =34.7
String s1= String.valueOf('a');             // s1="a"
String s2=String.valueOf(true);          // s2="true"
```

Comment Syntax

- Comments are the statements which are never execute. (i.e. non-executable statements).
- Comments are often used to add notes between source code. So that it becomes easy to understand & explain the function or operation of the corresponding part of source code.
- Java Compiler doesn't read comments. **Comments are simply ignored during compilation**.
- There are three types of comments available in Java as follows;
  1. Single Line Comment
  2. Multi Line Comment
  3. Documentation Comment

Single Line Comment

- This comment is used whenever we need to write anything in single line.

  Syntax : //<write comment here>

  Example: //This is Single Line Comment.

Multi Line Comment

- These types of comments are used whenever we want to write detailed notes (i.e. more than one line or in multiple lines) related to source code.

  Syntax :

```
/*
   <Write comment here>
*/
```

Example :

```
/*
        This Is
        Multi line comment.
*/
```

Documentation Comment

- The documentation comment is used commonly to produce an HTML file that documents our program.
- This comment begins with /** and end with a */.
- Documentation comments allow us to embed information about our program into the program itself.
- Then, by using the **javadoc** utility program to extract the information and put it into an HTML file.
- In the documentation comment, we can add different notations such as author of the project or program, version, parameters required, information on results in return if any, etc.
- To add these notations, we have '@' operator. We just need to write the required notation along with the '@' operator.
- **Some javadoc tags are given below:**
    - @author – To describe the author of the project.
    - @version – To describe the version of the project.
    - @param – To explain the parameters required to perform respective operation.
    - @return – To explain the return type of the project.

Syntax :

```
/**
*<write comment/description here>
*@author <write author name>
*@version <write version here>
*/
```

Example:

```
/**
* This is Documentation Comment.
* @author Vishal Makwana
* @version 1.0.0
*/
```

Garbage Collection

- Since objects are dynamically allocated by using the **new** operator, you might be wondering how such objects are destroyed and their memory released for later reallocation.
- In some languages, such as C++, dynamically allocated objects must be manually released by use of a **delete** operator.

- Java takes a different approach; it handles **deallocation** for you **automatically**. The technique that accomplishes this is called **garbage collection**.
- When no references to an object exist, that object is assumed to be no longer needed, and the memory occupied by the object can be recovered.
- **Garbage collection** only occurs occasionally during the execution of your program.
- Different Java run-time implementations will take varying approaches to garbage collection.

Array

- An array is a group of like-typed variables that are referred to by a common name.
- A specific element in an array is accessed by its index.
- Array index start at zero.
- Arrays of any type can be created and may have one or more dimensions.

One Dimensional Array

- **Steps To create a one dimensional array:**
  - You must declare a variable of the desired array type.
  - You must allocate the memory that will hold the array, using **new** operator and assign it to the array variable.
- In java, all arrays are dynamically allocated.

  Syntax:

```
data_type  array_var[];
array_var  =  new  data_type[size];
        OR
data_type  array_var []  =  new  data_type[size];
```

  Example:

```
int  a[]  = new  int[10];
```

- Here, **datatype** specifies the type of data being allocated, **size** specifies the number of elements in the array, and **array_var** is the array variable that is linked to the array.
- **new** is a special operator that allocates memory.
- The elements in the array allocated by **new** will automatically be initialized to **zero.**

  Example:

```
class  demo_array
{
        public static void main (String args[])
        {
            int a[] = new int[3];              // int a[] = {1,2,3};
            a[0] = 1;
            a[1] = 2;
            a[2] = 3;
            System.out.println("Your array elements are :: "+ a[0]+" "+a[1]+" "+a[2]);
        }
}
```

Output:

Your array elements are :: 1 2 3

Multidimensional Array

- Multidimensional arrays are actually **arrays of arrays**.
- To declare a multidimensional array variable, specify each additional index using another set of square brackets.

Syntax:

```
data_type  array_var [][] = new  data_type[size][size];
```

Example:

```
int  a[][] = new int[3][5];
```

- This allocates a 3 by 5 array and assigns it to **a**. Internally, this matrix is implemented as an array of arrays of **int**.

Example:

```
class demo_Tarray
{
        public static void main (String args[])
        {
                int a[][] = new int[2][2];              // int a[][] = {{1,2},{3,4}};
                a[0][0] = 1;
                a[0][1] = 2;
                a[1][0] = 3;
                a[1][1] = 4;
                System.out.println("Your array elements are :: "+ a[0][0]+" "+a[0][1]+"
                                "+a[1][0]+" "+a[1][1]);
        }
}
```

Output:

Your array elements are :: 1 2 3 4

String

- Strings are widely used in JAVA Programming, are not only a sequence of characters but it defines object.
- **String** Class is defined in **java.lang** package.
- The **String** type is used to declare string variables. Also we can declare array of strings.
- A variable of type **String** can be assign to another variable of type **String**.

Example:

```
String s1 = “Welcome To Java String”;
System.out.println(s1);
```

  - Here , **s1** is an object of type **String**.
- **String** objects have many special features and attributes that makes them powerful.

String class has following features:

- It is **Final** class
- Due to Final, String class cannot be inherited.
- It is immutable.

Method	Description
charAt(int index)	Returns the char value at the specified index.
compareTo(String anotherString)	Compares two strings lexicographically.
compareToIgnoreCase(String str)	Compares two strings, ignoring case differences.
concat(String str)	Concatenates the specified string to the end of this string.
contentEquals(StringBuffer sb)	Compares this string to the specified StringBuffer.
equals(Object anObject)	Compares this string to the specified object.
isEmpty()	Returns true if, and only if, length() is 0.
length()	Returns the length of this string.
split(String regex)	Splits this string around matches of the given regular expression.
toLowerCase()	Converts all of the characters in this String to lower case.
toString()	This object (which is already a string!) is itself returned.
toUpperCase()	Converts all of the characters in this String to upper case.
trim()	Returns a copy of the string, with leading and trailing whitespace omitted.

Example:

```
import java.io.*;
class  stringDemo
{
        public static void main(String args[])
        {
                String str = "Darshan Institute of Engineering & Technology";
                System.out.println(str.length());
                if(str.equals("DIET"))
                {
                        System.out.println("Same");
                }
else
{
        System.out.println("Not Same");
}
if ( str.compareTo("Darshan") > 0)
{
        System.out.println("Darshan is greater than Darshan Institute of
        Engineering & Technology ");
}
```

```
                else
                {
                        System.out.println("Darshan Institute of Engineering &
                        Technology is greater than Darshan");
                }
                System.out.println( str.substring(1,3));
        }
}
```

StringBuffer Class

- Java **StringBuffer** class is a thread-safe, mutable sequence of characters.
- Every string buffer has a capacity.
- It contains some particular sequence of characters, but the length and content of the sequence can be changed through certain method calls.

StringBuffer class Constructor

Constructor	Description
StringBuffer()	This constructs a string buffer with no characters in it and an initial capacity of **16** characters.
StringBuffer(CharSequence seq)	This constructs a string buffer that contains the same characters as the specified **CharSequence**.
StringBuffer(int capacity)	This constructs a string buffer with no characters in it and the specified initial **capacity**.
StringBuffer(String str)	This constructs a string buffer initialized to the contents of the specified **string**.

StringBuffer Methods

Method	Description
capacity()	Returns the current capacity of the String buffer.
charAt(int index)	This method returns the char value in this sequence at the specified index.
toString()	This method returns a string representing the data in this sequence.
insert(int offset, char c)	Inserts the string representation of the char argument into this character sequence.
append(String str)	Appends the string to this character sequence.
reverse()	The character sequence contained in this string buffer is replaced by the reverse of the sequence.

Example:

```
import java.io.*;
class  stringBufferDemo
{
    public static void main(String args[])
    {
            StringBuffer strBuf1 = new StringBuffer("DIET");
            StringBuffer strBuf2 = new StringBuffer(100);
```

```
        System.out.println("strBuf1 : " + strBuf1);
        System.out.println("strBuf1 capacity : " + strBuf1.capacity());
        System.out.println("strBuf2 capacity : " + strBuf2.capacity());
        System.out.println("strBuf1 reverse : " + strBuf1.reverse());
        System.out.println("strBuf1 charAt 2 : " +strBuf1.charAt(2));
        System.out.println("strBuf1 toString() is : " + strBuf1.toString());
        strBuf1.append("Darshan Institute of Engineering & Tech.");
        System.out.println("strBuf3 when appended with a String : "+ strBuf1);
    }
```

Output:

```
strBuf1 : DIET
strBuf1 capacity : 20
strBuf2 capacity : 100
strBuf1 reverse : TEID
strBuf1 charAt 2 : I
strBuf1 toString() is : TEID
strBuf3 when appended with a String : TEIDDarshan Institute of Engineering & Tech.
```

Difference Between String and StringBuffer

String	StringBuffer
It is **immutable** means you cannot modify.	It is **mutable** means you can modify.
String class is **slower** than the StringBuffer.	StringBuffer class is **faster** than the String.
String is **not safe** for use by multiple **threads**.	String buffers are **safe** for use by multiple **threads**.
String Class not provides **insert()** Operation.	StringBuffer Class provides **insert()** Operation.
String Class provides **split()** Operation.	StringBuffer Class not provides **split()** Operation.
String class overrides the **equals()** method of Object class. So you can compare the contents of two strings by equals() method.	StringBuffer class doesn't override the **equals()** method of Object class.

Operator

Arithmetic Operator

- Arithmetic operators are used in **mathematical** expressions.
- The following table lists the arithmetic operators: Assume integer variable **A** holds **10** and variable **B** holds **20**, then

Operator	Description	Example
+	Addition - Adds values on either side of the operator	A + B will give 30
-	Subtraction - Subtracts right hand operand from left hand operand	A - B will give -10
*	Multiplication - Multiplies values on either side of the operator	A * B will give 200

/	Division - Divides left hand operand by right hand operand	B / A will give 2
%	Modulus - Divides left hand operand by right hand operand and returns remainder	B % A will give 0
++	Increment - Increases the value of operand by 1	B++ gives 21
--	Decrement - Decreases the value of operand by 1	B-- gives 19

Relational Operator

- There are following relational operators supported by Java language. Assume variable **A** holds **10** and variable **B** holds **20**, then:

Operator	Description	Example
==	Checks if the values of two operands are equal or not, if yes then condition becomes true.	(A == B) is not true.
!=	Checks if the values of two operands are equal or not, if values are not equal then condition becomes true.	(A != B) is true.
>	Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true.	(A > B) is not true.
<	Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true.	(A < B) is true.
>=	Checks if the value of left operand is greater than or equal to the value of right operand, if yes then condition becomes true.	(A >= B) is not true.
<=	Checks if the value of left operand is less than or equal to the value of right operand, if yes then condition becomes true.	(A <= B) is true.

Bitwise Operator

- Bitwise operator works on bits and performs 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
```

Operator	Description	Example
&	Binary **AND** Operator copies a bit to the result if it exists in both operands.	(A & B) will give 12 which is 0000 1100
|	Binary **OR** Operator copies a bit if it exists in either operand.	(A | B) will give 61 which is 0011 1101

^	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
~	Binary **ones Complement** Operator is **unary** and has the effect of 'flipping' bits.	(~A) will give -61 which is 1100 0011
<<	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
>>	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 0000 1111

Logical Operator

- The following table lists the logical operators: Assume Boolean variables **A** holds **true** and variable **B** holds **false**, then:

Operator	Description	Example
&&	Called Logical **AND** operator. If both the operands are non-zero, then the condition becomes true.	(A && B) is false.
||	Called Logical **OR** Operator. If any of the two operands are non-zero, then the condition becomes true.	(A || B) is true.
!	Called Logical **NOT** Operator. Use to reverses the logical state of its operand. If a condition is true then Logical NOT operator will make false.	!(A && B) is true.

Assignment Operator

- There are following assignment operators supported by Java:

Operator	Description	Example
=	Simple assignment operator, Assigns values from right side operands to left side operand.	C = A + B will assign value of A + B into C
+=	Add and assignment operator, It adds right operand to the left operand and assign the result to left operand.	C += A is equivalent to C = C + A
-=	Subtract and assignment operator, It subtracts right operand from the left operand and assign the result to left operand.	C -= A is equivalent to C = C - A
*=	Multiply and assignment operator, It multiplies right operand with the left operand and assign the result to left operand.	C *= A is equivalent to C = C * A
/=	Divide and assignment operator, It divides left operand with the right operand and assign the result to left operand.	C /= A is equivalent to C = C / A

Conditional (Ternary) Operator

- **Conditional** operator is also known as the **ternary** operator.
- This operator consists of three operands and is used to evaluate **boolean** expressions.
- The goal of the operator is to decide which value should be assigned to the variable.

Syntax:

variable x = (expression) ? value if true : value if false

Example:

```
class conditionalDemo
{
        public static void main(String args[])
        {
            int a , b;
            a = 10;
            b = (a == 1) ? 20: 30;
            System.out.println( "Value of b is : " +  b );
            b = (a == 10) ? 20: 30;
            System.out.println( "Value of b is : " + b );
        }
}
```

Output:

```
Value of b is : 30
Value of b is : 20
```

Increment and Decrement Operator

- There are 2 Increment or decrement operators :: **++ (Increment value by 1) and -- (Decrement value by 1)**
- Both operators can be written before the operand called prefix increment/decrement, or after, called postfix increment/decrement.

Example: Assume, **x = 1**

```
y = ++x;
System.out.println(y);
z = x++;
System.out.println(z);
```

Output:

```
2
1
```

Mathematical Function

- Built-in mathematical function are available in **java.lang.math** package.
- There are so many mathematical function supported by Java. But few of them are as given below:

Function	Description
abs()	Returns the absolute value of the input **integer** argument.
round()	It returns the closest **integer** to the **float** argument.
ceil()	It returns the smallest integer greater than or equal to the number.
floor()	It returns the largest integer less than or equal to the number
min()	It returns the smaller of the two arguments.

max()	It returns the larger of two arguments.
sqrt()	It returns the square root of argument.
cos()	It returns the trigonometric cosine of argument(angle). Same for sine.
pow()	It returns the first argument raised to the power of second argument.

- **All argument to the function should be double.**

Example:

```
import java.lang.*;
class mathLibraryExample
{
        public static void main(String[] args)
        {
                int j = -9;
                double x = 72.3;

                System.out.println("|" + j + "| is " + Math.abs(j));
                System.out.println("|" + x + "| is " + Math.abs(x));
               System.out.println(x + " is approximately " + Math.round(x));
                System.out.println("The ceiling of " + x + " is " + Math.ceil(x));
               System.out.println("The floor of " + x + " is " + Math.floor(x));
                        System.out.println("min(" + j + "," + x + ") is " + Math.min(j,x));
                        System.out.println("max(" + j + "," + x + ") is " + Math.max(j,x));
                        System.out.println("pow(2.0, 2.0) is "  + Math.pow(2.0,2.0));
                        System.out.println("The square root of 9 is " + Math.sqrt(9));
                }
        }
```

Output:

```
|-9| is  9
|72.3| is  72.3
72.3 is approximately   72
The ceiling of 72.3 is   73.0
The floor of 72.3 is   72.0
min(-9,72.3) is   -9.0
max(-9,72.3) is   72.3
pow(2.0, 2.0) is  4.0
The square root of 9 is 3.0
```

Selection Statement

- Java supports two **selection** statements: **if** and **switch**.
- These statements allow you to control the flow of your program's execution based upon conditions known only during run time.

If statement

- if statement consists of a condition followed by one or more statements.

Syntax:

```
if (condition)
{
  //Statements will execute if the Boolean expression is true
}
```

- If the condition is true then the block of code inside **if** statement will be executed.

Example:

```
class ifDemo
{
        public static void main(String[] args)
        {
            int marks = 76;
            String grade = null;
            if (marks >= 40)
            {
                    grade = "Pass";
            }
            if (marks < 40)
            {
                    grade = "Fail";
            }
            System.out.println("Grade = " + grade);
        }
}
```

Output:

```
Grade = Pass
```

If … else Statement

- The **if …else** statement is Java's conditional branch statement.
- Here is the general form of the **if..else** statement:

Syntax:

```
if (condition)
        statement1;
else
        statement2;
```

- Here, each statement may be a **single statement** or a **compound statement** enclosed in curly braces (that is, a block).
- The condition is any expression that returns a **boolean** value. The **else** clause is optional.

- The **if..else** works as follow: If the condition is **true**, then **statement1** is executed. Otherwise, **statement2** (if it exists) is executed.

Example:

```
class ifelseDemo
{
        public static void main(String[] args)
        {
            int marks = 76;
            String grade;
            if (marks >= 40)
            {
                    grade = "Pass";
            }
            else
            {
                    grade = "Fail";
            }
            System.out.println("Grade = " + grade);
        }
}
```

Output:

```
Grade = Pass
```

Switch Statement

- The **switch** statement is Java's multiway branch statement. It provides an easy way to execute different parts of your code based on the value of an **expression**.
- Here is the general form of a **switch** statement:

Syntax:

```
switch (expression)
{
        case value1:
                // statement sequence
                break;
        case value2:
                // statement sequence
                break;
                ...
        case valueN:
                // statement sequence
                break;
        default:
                // default statement sequence
}
```

- The **expression** must be of type **byte**, **short**, **int** or **char**.
- Each **case** value must be a unique literal (that is, it must be a constant, not a variable). Duplicate **case** values are not allowed.

- The **switch** statement works as follow: The value of the **expression** is compared with each of the **literal** values in the **case** statements. If a match is found, the code sequence following that **case** statement is executed.
- If none of the literal matches the value of the **expression**, then the **default** statement is executed. However, the **default** statement is optional.
- The **break** statement is used inside the **switch** to terminate a statement sequence.

Example:

```
class switchDemo
{
public static void main(String[] args)
{
    int day = 8;
    switch (day)
    {
       case 1:  System.out.println("Monday"); break;
       case 2:  System.out.println("Tuesday"); break;
       case 3:  System.out.println("Wednesday"); break;
       case 4:  System.out.println("Thursday"); break;
       case 5:  System.out.println("Friday"); break;
       case 6:  System.out.println("Saturday"); break;
       case 7:  System.out.println("Sunday"); break;
       default: System.out.println("Invalid Day");break;
    }
}
```

Output:

```
Thursday
```

Iteration Statement

- Java's iteration statements are **for**, **while**, and **do-while**.
- These statements commonly call loops.
- Loop repeatedly executes the same **set of instructions** until a **termination** condition is met.

while

- It repeats a statement or block until its **controlling expression** is true.

Syntax:

```
while(condition)
{
    // body of loop
}
```

- The condition can be any **boolean** expression. The body of the loop will be executed as long as the conditional expression is **true**.
- When condition becomes **false**, control passes to the next line of code immediately following the loop.

Example:

```
class whileDemo
{
```

```
public static void main(String args[])
{
    int x = 1;
    while( x < 5 )
    {
        System.out.print(x+" " );
        x++;
    }
}
```

Output:

```
1 2 3 4
```

do-while

- Sometimes it is desirable to execute the body of a loop at least **once**, even if the conditional expression is **false**.
- The **do-while** loop always executes its body at least once, because its conditional expression is at the bottom of the loop.

Syntax:

```
do
{
    // body of loop
} while (condition);
```

- Each iteration of the **do-while** loop first executes the body of the loop and then evaluates the conditional expression. If this expression is **true**, the loop will **repeat**. Otherwise, the loop terminates. Condition must be a **boolean** expression.

Example:

```
class dowhileDemo
{
    public static void main(String args[])
    {
        int x = 1;
        do
        {
            System.out.print(x+" ");
            x++;
        }while( x < 5);
    }
}
```

Output:

```
1 2 3 4
```

for

Syntax:

```
for(initialization; condition; iteration)
{
```

```
        // body
    }
```

- When the loop first starts, the **initialization** portion of the loop is executed. Generally, this is an expression that sets the value of the loop control variable, which acts as a counter that controls the loop. The **initialization** expression is only executed **once**.
- Next, **condition** is evaluated. This must be a **boolean** expression. It usually tests the loop control variable against a target value. If this expression is **true**, then the body of the loop is executed. If it is **false**, the loop **terminates**.
- Next, the **iteration portion** of the loop is executed. This is usually an expression that **increments** or **decrements** the loop control variable.
- The loop then iterates, **first** evaluating the **conditional expression**, then executing the body of the loop, and then executing the iteration expression with each pass. This process repeats until the **controlling expression is false**.

Example:

```
class forDemo
{
        public static void main(String args[])
        {
                for(int x = 1; x < 5; x++ )
                {
                    System.out.print(x+" ");
                }
        }
}
```

Output:

```
1 2 3 4
```

Jump Statement

- These statements transfer control to another part of your program.

break

- The **break** keyword is used to **stop** the entire loop. The **break** keyword must be used inside any **loop or a switch** statement.
- The **break** keyword will stop the execution of the **innermost** loop and start executing the next line of code after the block.

Example:

```
class breakDemo
{
        public static void main(String args[])
        {
                int [] numbers = {10, 20, 30, 40, 50};
                for(int x=0;x<5;x++ )
                {
                        if( numbers[x] == 30 )
                        {
```

```
                        break;
                    }
                    System.out.print( numbers[x] +"  ");
                }
            }
        }
```

Output:

```
10  20
```

Continue

- **Continue** statement is used when we want to **skip** the rest of the statement in the body of the loop and **continue** with the **next** iteration of the loop.

Example:

```
class  continueDemo
{
        public static void main(String args[])
        {
            int [] numbers = {10, 20, 30, 40, 50};
            for(int x=0;x<5;x++ )
            {
                if( numbers[x] == 30 )
                {
                    continue;
                }
                System.out.print( numbers[x] +"  ");
            }
        }
    }
```

Output:

```
10  20  40  50
```

return

- The **return** statement is used to **explicitly return** from a **method**. It transfers control of program **back to the caller** of the method.
- The **return** statement immediately **terminates the method** in which it is executed.

Example:

```
class Test
{
public static void main(String args[])
{
    int a=10, b=20, c;
    c=add(a,b);
    System.out.println("Addition="+c);
}
public static int add(int x, int y)
{
        return x+y;
```

```
                }
        }
```

Output:

```
Addition=30
```