Java Basic

**What is java?**

Java is a **programming language** and a **platform**. Java is a high level, robust, object-oriented and secure programming language.

**History**

develop by James Gosling in 1995

sun microsystem/oracle

previous name oracle

**It is used for:**

* Mobile applications (specially Android apps)
* Desktop applications
* Web applications
* Web servers and application servers
* Games
* Database connection
* And much, much more!

**Features of java**



**Types of java applications**

Mainly 4 types of applications are created using java

1 standalone application

-desktop or windows based applications created using, AWT, swing

-media player, antivirus

2 web application

-runs o server side to create dynamic web pages

-servlet, jsp, strusts, spring, hibernate, jfs etc

3 enterprise application

-applications that are distributed in nature(banking applications or EA).

-high level security

-load balancing

-clustering

-EJB used for creating enterprise applications

4 mobile applications

-android and java ME use to create mobile applications.

**Java Platform/Editions**

There are 4 platforms or editions of Java:

**1) Java SE (Java Standard Edition)**

It is a Java programming platform. It includes Java programming APIs such as java.lang, java.io, java.net, java.util, java.sql, java.math etc. It includes core topics like OOPs, [String](https://www.javatpoint.com/java-string), Regex, Exception, Inner classes, Multithreading, I/O Stream, Networking, AWT, Swing, Reflection, Collection, etc.

**2) Java EE (Java Enterprise Edition)**

It is an enterprise platform that is mainly used to develop web and enterprise applications. It is built on top of the Java SE platform. It includes topics like Servlet, JSP, Web Services, EJB, [JPA](https://www.javatpoint.com/jpa-tutorial), etc.

**3) Java ME (Java Micro Edition)**

It is a micro platform that is dedicated to mobile applications.

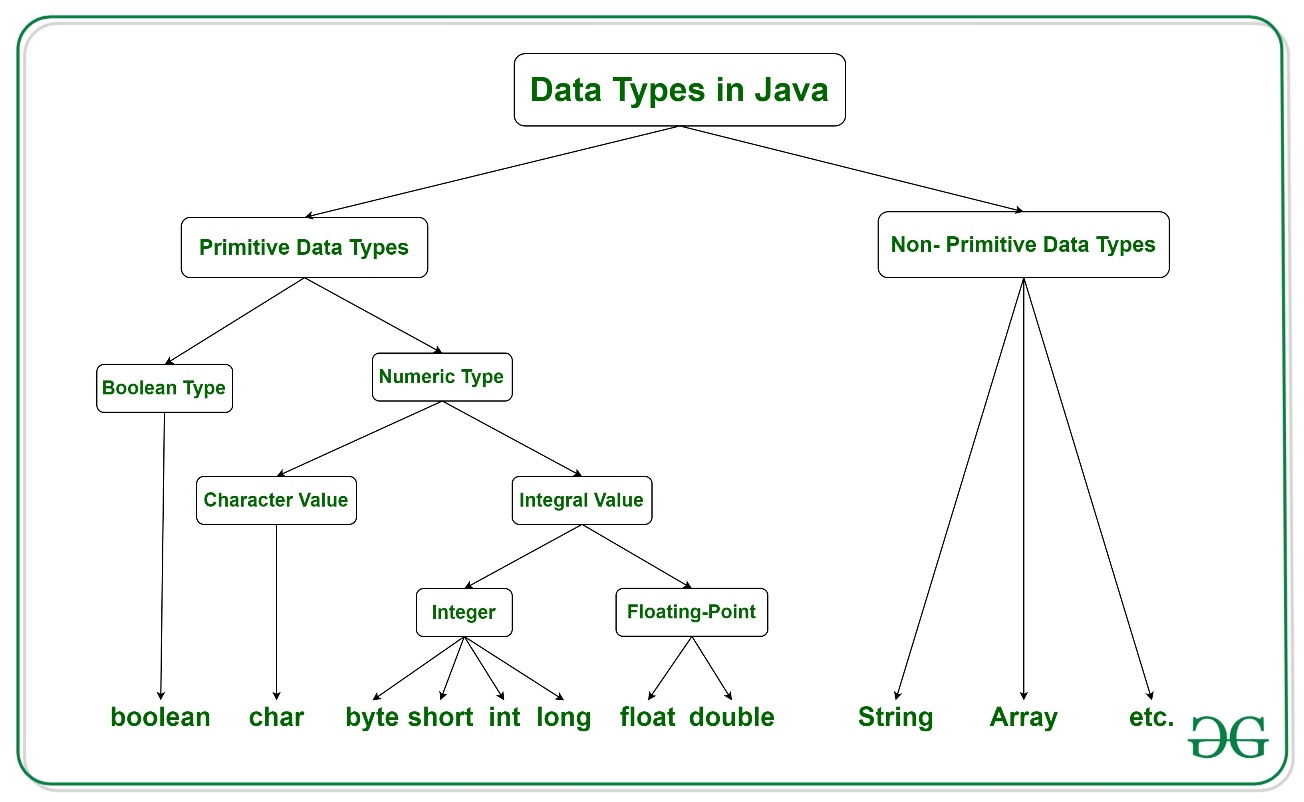
**4) JavaFX**

It is used to develop rich internet applications. It uses a lightweight user interface API.

Data Types

Data types specify the different sizes and values that can be stored in the variable. There are two types of data types in Java:

1. **Primitive data types:** The primitive data types include boolean, char, byte, short, int, long, float and double.
2. **Non-primitive data types:** The non-primitive data types include [Classes](https://www.javatpoint.com/object-and-class-in-java), [Interfaces](https://www.javatpoint.com/interface-in-java), and [Arrays](https://www.javatpoint.com/array-in-java).



statically type: here we declare data type of defined variable

dynamically type: we don’t need to declare data type of variable.

**Primitive data type**

primitive data types are the building blocks of data manipulation. These are the most basic data types.



**Non-primitive data type or reference data type**

The Reference Data Types will contain a memory address of variable values because the reference types won’t store the variable value directly in memory. They are strings, objects, arrays, etc.

Variables

-it is a container which holds value while program is executed.

-it is name of memory location inside JVM

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

**Declaring & initializing variables**

int a =20;

data\_type= int

variable\_name=a

value\_initialise=20

**Rules for declaring variable name**

1 not begin with a digit

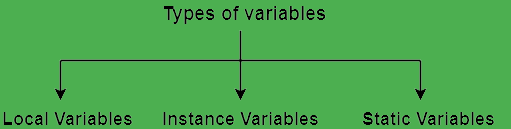
2 case sensitive

3 should not be keyword

4 whitespaces are not allowed

5 no other special character is allow other than ‘$’, \_.

**Types of variables**



declare inside declare inside class declare with static keyword

method outside method inside class outside method

1 local variables

A variable defined within a block or method or constructor is called a local variable.

* These variables are created when the block is entered, or the function is called and destroyed after exiting from the block or when the call returns from the function.
* The scope of these variables exists only within the block in which the variables are declared, i.e., we can access these variables only within that block.
* Initialization of the local variable is mandatory before using it in the defined scope.

2 instance variables

Instance variables are non-static variables and are declared in a class outside of any method, constructor, or block.

* As instance variables are declared in a class, these variables are created when an object of the class is created and destroyed when the object is destroyed.
* Unlike local variables, we may use access specifiers for instance variables. If we do not specify any access specifier, then the default access specifier will be used.
* Initialization of an instance variable is not mandatory. Its default value is 0.
* Instance variables can be accessed only by creating objects.

3 static variables

Static variables are also known as class variables.

* These variables are declared similarly as instance variables. The difference is that static variables are declared using the static keyword within a class outside of any method, constructor or block.
* Unlike instance variables, we can only have one copy of a static variable per class, irrespective of how many objects we create.
* Static variables are created at the start of program execution and destroyed automatically when execution ends.
* Initialization of a static variable is not mandatory. Its default value is 0.
* If we access a static variable like an instance variable (through an object), the compiler will show a warning message, which won’t halt the program. The compiler will replace the object name with the class name automatically.
* If we access a static variable without the class name, the compiler will automatically append the class name.

class A

{

static int a=20;

int b=30;

void func(int c)

{

c=10;

}

}

p s v m(String[] args)

{

int d=40;

sopln(d);

sopln(c);

A ref=new A();

sopln(ref.b);

sopln(A.a);//or sopln(a);

}

}

Note:

-local variable can be easily accessible

-instance variable is accessible by creating object of that class

-static variable can be access with or without class name.

**Scope of variables**

Scope of a variable is the part of the program where the variable is accessible.

-class level scope.

-method level scope.

Wrapper classes

-java being object oriented programming it is always expecting object not primitive.

-so wrapper class is mechanism to convert primitive to object and object to primitive.

**Objective of wrapper class**

-to wrap primitive to object form so that we can handle primitive just like object

int Interger

-to wrap int into object some concept is required that concept is known as wrapper class.

-here to wrap int into its object Integer wrapper class in needed.

-to define several utility methods for primitive

eg

convert int to String

String s=Integer.toString(10);

**Need of wrapper class**

Java is an object-oriented programming language, so we need to deal with objects many times like in Collections, Serialization, Synchronization, etc. Let us see the different scenarios, where we need to use the wrapper classes.

* **Change the value in Method:** Java supports only call by value. So, if we pass a primitive value, it will not change the original value. But, if we convert the primitive value in an object, it will change the original value.
* **Serialization:** We need to convert the objects into streams to perform the serialization. If we have a primitive value, we can convert it in objects through the wrapper classes.
* **Synchronization:** Java synchronization works with objects in Multithreading.
* **java.util package:** The java.util package provides the utility classes to deal with objects.
* **Collection Framework:** Java collection framework works with objects only. All classes of the collection framework (ArrayList, LinkedList, Vector, HashSet, LinkedHashSet, TreeSet, PriorityQueue, ArrayDeque, etc.) deal with objects only.

-also wrapper class contains several utility methods.

1 valueOf()

2 parseInt()

3 toString()

1 valueOf()

The **java string valueOf()** method converts different types of values into string. By the help of string valueOf() method, you can convert int to string, long to string, boolean to string, character to string, float to string, double to string, object to string and char array to string.

**int** value=30;

String s1=String.valueOf(value);

System.out.println(s1+10);//concatenating string with 10

2 parseInt()

The method **parseInt()** belongs to the Integer class which is under **java.lang package**. It is used to parse the string value as a signed decimal value. It is used in Java for converting a string value to an integer by using the method parseInt().

public class ParseIntEx {

public static void main(String[] args) {

int decimalExample = Integer.parseInt("3");

int signedPositiveExample = Integer.parseInt("+3");

int signedNegativeExample = Integer.parseInt("-3");

System.out.println("Integer.parseInt(\"3\") = " + decimalExample);

System.out.println("Integer.parseInt(\"+3\") = " + signedPositiveExample);

System.out.println("Integer.parseInt(\"-3\") = " + signedNegativeExample);

}

}

3 toString()

it is in build method which is use to convert object into string.

The **java.lang.Integer.toString()** is an inbuilt method in Java which is used to return the String object representing this Integer’s value.

1 public static String toString()

2 public static String toString(int *a*)

3 public static String toString(int a, int base)

import java.lang.\*;

public class Geeks{

public static void main(String[] args) {

    Integer obj = new Integer(8);

    //It will return a string representation

    String stringvalue1 = obj.toString();

    System.out.println("String Value= " +

                            stringvalue1);

    String stringvalue2 = obj2.toString(6787);

    System.out.println("String Value = " +

                            stringvalue2);

String returnvalue = a.toString(5254, 2);

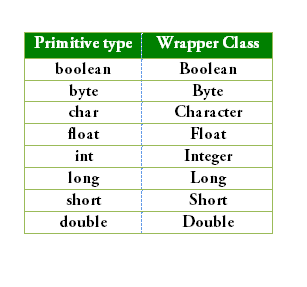
    System.out.println("String Value = " +

                            returnvalue);

}

}

Primitive data types & their corresponding wrapper classes.



**Autoboxing & Autounboxing**

Autoboxing

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

import java.util.ArrayList;

class Autoboxing

{

    public static void main(String[] args)

    {

        char ch = 'a';

        // Autoboxing- primitive to Character object conversion

        Character a = ch;

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

        // Autoboxing because ArrayList stores only objects

        arrayList.add(25);

        // printing the values from object

        System.out.println(arrayList.get(0));

    }

}

Note:

before java 1.5 v, autoboxing concept was not there we use valueOf() method to convert primitive to wrapper class object.

public class WrapperExample1{

public static void main(String args[]){

//Converting int into Integer

int a=20;

Integer i=Integer.valueOf(a);//converting int into Integer explicitly

Integer j=a;//autoboxing, now compiler will write Integer.valueOf(a) internally

System.out.println(a+" "+i+" "+j);

}

}

Autounboxing

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

import java.util.ArrayList;

class Unboxing

{

    public static void main(String[] args)

    {

        Character ch = 'a';

        // unboxing - Character object to primitive conversion

        char a = ch;

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

        arrayList.add(24);

        // unboxing because get method returns an Integer object

        int num = arrayList.get(0);

        // printing the values from primitive data types

        System.out.println(num);

    }

}

Note:

before 1.5 v autounboxing concept was not there then we use xxxValue() method to convert wrapper class into its corresponding primitive datatype.

xxx=int, float etc

// Java program to demonstrate Wrapping and UnWrapping

// in Java Classes

class WrappingUnwrapping

{

public static void main(String args[])

{

// byte data type

byte a = 1;

// wrapping around Byte object

Byte byteobj = new Byte(a);

// int data type

int b = 10;

//wrapping around Integer object

Integer intobj = new Integer(b);

// float data type

float c = 18.6f;

// wrapping around Float object

Float floatobj = new Float(c);

// double data type

double d = 250.5;

// Wrapping around Double object

Double doubleobj = new Double(d);

// char data type

char e='a';

// wrapping around Character object

Character charobj=e;

// printing the values from objects

System.out.println("Values of Wrapper objects (printing as objects)");

System.out.println("Byte object byteobj: " + byteobj);

System.out.println("Integer object intobj: " + intobj);

System.out.println("Float object floatobj: " + floatobj);

System.out.println("Double object doubleobj: " + doubleobj);

System.out.println("Character object charobj: " + charobj);

// objects to data types (retrieving data types from objects)

// unwrapping objects to primitive data types

byte bv = byteobj;

int iv = intobj;

float fv = floatobj;

double dv = doubleobj;

char cv = charobj;

// printing the values from data types

System.out.println("Unwrapped values (printing as data types)");

System.out.println("byte value, bv: " + bv);

System.out.println("int value, iv: " + iv);

System.out.println("float value, fv: " + fv);

System.out.println("double value, dv: " + dv);

System.out.println("char value, cv: " + cv);

} }

Type casting

**-type casting** is a method or process that converts a data type into another data type in both ways manually and automatically. The automatic conversion is done by the compiler and manual conversion performed by the programmer.

In Java, there are two types of casting:

* **Widening Casting** (automatically) - converting a smaller type to a larger type size  
  byte -> short -> char -> int -> long -> float -> double
* **Narrowing Casting** (manually) - converting a larger type to a smaller size type  
  double -> float -> long -> int -> char -> short -> byte

**Widening**

public class Main {

public static void main(String[] args) {

int myInt = 9;

double myDouble = myInt; // Automatic casting: int to double

System.out.println(myInt); // Outputs 9

System.out.println(myDouble); // Outputs 9.0

}

}

**Narrowing**

public class Main {

public static void main(String[] args) {

double myDouble = 9.78d;

int myInt = (int) myDouble; // Manual casting: double to int

System.out.println(myDouble); // Outputs 9.78

System.out.println(myInt); // Outputs 9

}

}

Operators

**Operator** in [Java](https://www.javatpoint.com/java-tutorial) is a symbol that is used to perform operations. For example: +, -, \*, / etc.

* Unary Operator,
* Arithmetic Operator,
* Shift Operator,
* Relational Operator,
* Bitwise Operator,
* Logical Operator,
* Ternary Operator and
* Assignment Operator.
* instance of operator

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

**-left shift operator**

The Java left shift operator << is used to shift all of the bits in a value to the left side of a specified number of times.

Left shift operator shifts the bits of the number towards **left** a specified number of positions. The symbol for this operator is **<<**. When you write **x<<n**, the meaning is to shift the bits of **x** towards left **n specified** positions.

Example

If **x=10**, then calculate **x<<2** value.

Shifting the value of x towards the left two positions will make the leftmost 2 bits to be lost. The value of x is 10. The binary representation of 10 is **00001010**. The procedure to do left shift explained in the following example:

Observe the above example, after shifting the bits to the left the binary number **00001010** (in decimal 10) becomes **00101000** (in decimal 40).

public class OperatorExample

{

public static void main(String args[])

{

System.out.println(10<<2);//10\*2^2=10\*4=40

System.out.println(10<<3);//10\*2^3=10\*8=80

}

}

**-right shift operator**

The Java right shift operator >> is used to move the value of the left operand to right by the number of bits specified by the right operand.

The Right Shift Operator shifts the bits of the number towards **right** a specified n number of positions. Right shift operator represented by the symbol **>>**, read as double greater than. When you write **x>>n**, the meaning is to shift the bits **x** towards the right n specified positions.

**>>** shifts the bits towards the right and also **preserve the sign bit**, which is the leftmost bit. The leftmost bit represents the **sign** of the number. The sign bit **0** represents a **positive number**, and **1** represents a **negative number**. So after performing **>>** on a positive number, we get a **positive value** in the result also. When we perform **>>** on a **negative number**, again we get a **negative value**.

Example

If **x=10**, then calculate **x>>2** value.

Shifting the value of x towards the right two positions will make the rightmost 2 bits to be lost. The value of x is 10. The binary representation of **10** is **00001010**. The procedure to do right shift explained in the following example:

Observe the above example, after shifting the bits to the right the binary number **00001010** (in decimal 10) becomes **00000010** (in decimal 2).

public OperatorExample

{

public static void main(String args[])

{

System.out.println(20>>2);//20/2^2=20/4=5

System.out.println(20>>3);//20/2^3=20/8=2

}

}

**Bitwise Zero Fill Right Shift Operator (>>>)**

public class OperatorExample

{

public static void main(String args[])

{

//For positive number, >> and >>> works same

System.out.println(20>>2);

System.out.println(20>>>2);

//For negative number, >>> changes parity bit (MSB) to 0

System.out.println(-20>>2);

System.out.println(-20>>>2);

}

}

**-logical (&&) and bitwise (&) AND operator**

The logical && operator doesn't check the second condition if the first condition is false. It checks the second condition only if the first one is true.

The bitwise & operator always checks both conditions whether first condition is true or false.

public class OperatorExample

{

public static void main(String args[])

{

int a=10;

int b=5;

int c=20;

System.out.println(a<b&&a++<c);//false && true = false

System.out.println(a);//10 because second condition is not checked

System.out.println(a<b&a++<c);//false && true = false

System.out.println(a);//11 because second condition is checked

}

}

**-logical (||) and bitwise (|) OR operator**

The logical || operator doesn't check the second condition if the first condition is true. It checks the second condition only if the first one is false.

The bitwise | operator always checks both conditions whether first condition is true or false.

public class OperatorExample

{

public static void main(String args[])

{

int a=10;

int b=5;

int c=20;

System.out.println(a>b||a<c);//true || true = true

System.out.println(a>b|a<c);//true | true = true

//|| vs |

System.out.println(a>b||a++<c);//true || true = true

System.out.println(a);//10 because second condition is not checked

System.out.println(a>b|a++<c);//true | true = true

System.out.println(a);//11 because second condition is checked

}

}

**-ternary operator**

Ternary operator is used as one line replacement for **if-then-else** statement.

It is the only conditional operator which takes three operands.

public class OperatorExample

{

public static void main(String args[])

{

int a=10;

int b=5;

int min=(a<b)?a:b;

System.out.println(min);

}

}

-if a>b then print a else b

Keywords

Java keywords are also known as reserved words. Keywords are particular words that act as a key to a code. These are predefined words by Java so they cannot be used as a variable or object name or class name.

-**instanceof**

The **java instanceof operator** is used to test whether the object is an instance of the specified type (class or subclass or interface).

The instanceof in java is also known as type *comparison operator* because it compares the instance with type. It returns either true or false. If we apply the instanceof operator with any variable that has null value, it returns false.

class Animal{}

class Dog1 extends Animal

{//Dog inherits Animal

public static void main(String args[])

{

Dog1 d=new Dog1();

System.out.println(d instanceof Animal);//true

}

}

-**strictfp**

Java strictfp keyword ensures that you will get the same result on every platform if you perform operations in the floating-point variable. The precision may differ from platform to platform that is why java programming language have provided the strictfp keyword, so that you get same result on every platform. So, now you have better control over the floating-point arithmetic.

The strictfp keyword can be applied on methods, classes and interfaces.

strictfp class A{}//strictfp applied on class

strictfp interface M{}//strictfp applied on interface

class A{

strictfp void m(){}//strictfp applied on method

}

Control statement

Java compiler executes the code from top to bottom. The statements in the code are executed according to the order in which they appear. However, [Java](https://www.javatpoint.com/java-tutorial) provides statements that can be used to control the flow of Java code. Such statements are called control flow statements. It is one of the fundamental features of Java, which provides a smooth flow of program.

Java provides three types of control flow statements.

1. Decision Making statements
   * if statements
   * switch statement
2. Loop statements
   * do while loop
   * while loop
   * for loop
   * for-each loop
3. Jump statements
   * break statement
   * continue statement

**1 Decision Making Statements**

decision-making statements decide which statement to execute and when.

The [Java](https://www.javatpoint.com/java-tutorial) *if statement* is used to test the condition. It checks [boolean](https://www.javatpoint.com/boolean-keyword-in-java) condition: *true* or *false*. There are various types of if statement in Java.

* if statement
* if-else statement
* if-else-if ladder
* nested if statement

**if and if-else statement**

public class Example

{

public static void main(String[] args)

{

int age=20;

int number=13;

if(age>18){

System.out.print("Age is greater than 18");

}

if(number%2==0){

System.out.println("even number");

}else{

System.out.println("odd number");

}

}

}

**if-else using ternary operator**

ternary operator (? :) can be used to perform if-else operation

public class IfElseTernaryExample

{

public static void main(String[] args)

{

int number=13;

String output=(number%2==0)?"even number":"odd number";

System.out.println(output);

}

}

**if-else-if ladder**

if-else-if ladder statement executes one condition from multiple statements.

public class IfElseIfExample {

public static void main(String[] args) {

int marks=65;

if(marks<50){

System.out.println("fail");

}

else if(marks>=50 && marks<60){

System.out.println("D grade");

}

else if(marks>=60 && marks<70){

System.out.println("C grade");

}

else if(marks>=70 && marks<80){

System.out.println("B grade");

}

else if(marks>=80 && marks<90){

System.out.println("A grade");

}else if(marks>=90 && marks<100){

System.out.println("A+ grade");

}else{

System.out.println("Invalid!");

}

}

**nested if**

The nested if statement represents the if block within another if block. Here, the inner if block condition executes only when outer if block condition is true.

if(age>=18){

if(weight>50){

System.out.println("You are eligible to donate blood");

}

}

public class JavaNestedIfExample2 {

public static void main(String[] args) {

//Creating two variables for age and weight

int age=25;

int weight=48;

//applying condition on age and weight

if(age>=18){

if(weight>50){

System.out.println("You are eligible to donate blood");

} else{

System.out.println("You are not eligible to donate blood");

}

} else{

System.out.println("Age must be greater than 18");

}

}

}

**switch statement**

The Java *switch statement* executes one statement from multiple conditions. It is like [if-else-if](https://www.javatpoint.com/java-if-else) ladder statement. The switch statement works with byte, short, int, long, enum types, String and some wrapper types like Byte, Short, Int, and Long. Since Java 7, you can use [strings](https://www.javatpoint.com/java-string) in the switch statement.

In other words, the switch statement tests the equality of a variable against multiple values.

* here can be one or N number of case values for a switch expression.
* The case value must be of switch expression type only. The case value must be literal or constant. It doesn't allow [variables](https://www.javatpoint.com/java-variables).
* The case values must be unique. In case of duplicate value, it renders compile-time error.
* The Java switch expression must be of byte, short, int, long (with its Wrapper type), [*enums*](https://www.javatpoint.com/java-switch) and string.
* Each case statement can have a break statement which is optional. When control reaches to the [break statement](https://www.javatpoint.com/java-break), it jumps the control after the switch expression. If a break statement is not found, it executes the next case.
* The case value can have a default label which is optional.

public class SwitchExample {

public static void main(String[] args) {

//Declaring a variable for switch expression

int number=20;

//Switch expression

switch(number){

//Case statements

case 10: System.out.println("10");

break;

case 20: System.out.println("20");

break;

case 30: System.out.println("30");

break;

//Default case statement

default:System.out.println("Not in 10, 20 or 30");

}

}

}

Note:

Java switch statement is fall-through. It means it executes all statements after the first match if a break statement is not present.

public class SwitchExample2 {

public static void main(String[] args) {

int number=20;

//switch expression with int value

switch(number){

//switch cases without break statements

case 10: System.out.println("10");

case 20: System.out.println("20");

case 30: System.out.println("30");

default:System.out.println("Not in 10, 20 or 30");

}

}

}

O/P

20

30

Not in 10, 20 or 30

**Note**:

Java allows us to use enum in switch statement. Java enum is a class that represent the group of constants. (immutable such as final variables). We use the keyword enum and put the constants in curly braces separated by comma.

Java allows us to use four [wrapper classes](https://www.javatpoint.com/wrapper-class-in-java): Byte, Short, Integer and Long in switch statement.

**2 Loop statement**

sometimes we need to execute the block of code repeatedly while some condition evaluates to true

in this condition we use loop

1. for loop
2. while loop
3. do-while loop

**-for loop**

The Java for loop is used to iterate a part of the program several times.

If the number of iteration is **fixed**, it is recommended to use for loop.

-simple for loop

for(initialization; condition; increment/decrement)

{

//statement or code to be executed

}

**-nested for loop**

If we have a for loop inside the another loop, it is known as nested for loop. The inner loop executes completely whenever outer loop executes.

public class NestedForExample

{

public static void main(String[] args)

{

//loop of i

for(int i=1;i<=3;i++)

{

//loop of j

for(int j=1;j<=3;j++)

{

System.out.println(i+" "+j);

}//end of i

}//end of j

}

}

**-for-each loop**

The for-each loop is used to traverse array or collection in Java. It is easier to use than simple for loop because we don't need to increment value and use subscript notation.

It works on the basis of elements and not the index. It returns element one by one in the defined variable.

for(data\_type variable : array\_name)

{

//code to be executed

}

int arr[]={12,23,44,56,78};

//Printing array using for-each loop

for(int i:arr)

{

System.out.println(i);

}

**-labeled for loop**

We can have a name of each Java for loop. To do so, we use label before the for loop. It is useful while using the nested for loop as we can break/continue specific for loop.

labelname:

for(initialization; condition; increment/decrement)

{

//code to be executed

}

public class LabeledForExample {

public static void main(String[] args) {

//Using Label for outer and for loop

aa:

for(int i=1;i<=3;i++){

bb:

for(int j=1;j<=3;j++){

if(i==2&&j==2){

break aa;

}

System.out.println(i+" "+j);

}

}

}

}

-while loop

The [Java](https://www.javatpoint.com/java-tutorial) *while loop* is used to iterate a part of the [program](https://www.javatpoint.com/programs-list) repeatedly until the specified Boolean condition is true. As soon as the Boolean condition becomes false, the loop automatically stops.

The while loop is considered as a repeating if statement. If the number of iteration is not fixed, it is recommended to use the while [loop](https://www.javatpoint.com/java-for-loop).

while (condition)

{

//code to be executed

I ncrement / decrement statement

}

public class WhileExample {

public static void main(String[] args) {

int i=1;

while(i<=10){

System.out.println(i);

i++;

}

}

}

**-do-while loop**

The Java *do-while loop* is used to iterate a part of the program repeatedly, until the specified condition is true. If the number of iteration is not fixed and you must have to execute the loop at least once, it is recommended to use a do-while loop.

Java do-while loop is called an **exit control loop**. Therefore, unlike while loop and for loop, the do-while check the condition at the end of loop body. The Java *do-while loop* is executed at least once because condition is checked after loop body.

do

{

//code to be executed / loop body

//update statement

}

while (condition);

public class DoWhileExample {

public static void main(String[] args) {

int i=1;

do{

System.out.println(i);

i++;

}while(i<=10);

}

}

-Difference between 3 loops

|  |  |  |  |
| --- | --- | --- | --- |
| **Comparison** | **for loop** | **while loop** | **do-while loop** |
| Introduction | The Java for loop is a control flow statement that iterates a part of the [programs](https://www.javatpoint.com/java-programs) multiple times. | The Java while loop is a control flow statement that executes a part of the programs repeatedly on the basis of given boolean condition. | The Java do while loop is a control flow statement that executes a part of the programs at least once and the further execution depends upon the given boolean condition. |
| When to use | If the number of iteration is fixed, it is recommended to use for loop. | If the number of iteration is not fixed, it is recommended to use while loop. | If the number of iteration is not fixed and you must have to execute the loop at least once, it is recommended to use the do-while loop. |
| Syntax | for(init;condition;incr/decr){ // code to be executed } | while(condition){ //code to be executed } | do{ //code to be executed }while(condition); |
| Example | //for loop for(int i=1;i<=10;i++){ System.out.println(i); } | //while loop int i=1; while(i<=10){ System.out.println(i); i++; } | //do-while loop int i=1; do{ System.out.println(i); i++; }while(i<=10); |
| Syntax for infinitive loop | for(;;){ //code to be executed } | while(true){ //code to be executed } | do{ //code to be executed }while(true); |

-break

Break Statement is a loop control statement that is used to terminate the loop. As soon as the break statement is encountered from within a loop, the loop iterations stop there, and control returns from the loop immediately to the first statement after the loop.

We can use Java break statement in all types of loops such as [for loop](https://www.javatpoint.com/java-for-loop), [while loop](https://www.javatpoint.com/java-while-loop) and [do-while loop](https://www.javatpoint.com/java-do-while-loop).

break;

for(int i=1;i<=10;i++){

if(i==5){

//breaking the loop

break;

}

System.out.println(i);

}

-continue

The continue statement is used in loop control structure when you need to jump to the next iteration of the loop immediately. It can be used with for loop or while loop.

for(int i=1;i<=10;i++){

if(i==5){

//using continue statement

continue;//it will skip the rest statement

}

System.out.println(i);

}

Keywords

| .No | Keyword | Usage |
| --- | --- | --- |
| 1. | **abstract** | Specifies that a class or method will be implemented later, in a subclass |
| 2. | **assert** | Assert describes a predicate placed in a java program to indicate that the developer thinks that the predicate is always true at that place. |
| 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 values |
| 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 is 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** | This 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 is 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** | This indicates that a variable may change asynchronously |
| 49. | **while** | Starts a while loop |

**-super keyword**

The**super** keyword in java is a reference variable that is used to refer to parent class objects.

* Use of super with variables
* Use of super with methods
* Use of super with constructors

1

This scenario occurs when a derived class and base class has the same data members. In that case, there is a possibility of ambiguity.

class Vehicle {

int maxSpeed = 120;

}

// sub class Car extending vehicle

class Car extends Vehicle {

int maxSpeed = 180;

void display()

{

// print maxSpeed of base class (vehicle)

System.out.println("Maximum Speed: "

+ super.maxSpeed);

}

}

// Driver Program

class Test {

public static void main(String[] args)

{

Car small = new Car();

small.display();

}

}

O/P 120

2

This is used when we want to call the parent class [method](https://www.geeksforgeeks.org/methods-in-java/). So whenever a parent and child class have the same-named methods then to resolve ambiguity we use the super keyword.

class Person {

void message()

{

System.out.println("This is person class\n");

}

}

// Subclass Student

class Student extends Person {

void message()

{

System.out.println("This is student class");

}

// Note that display() is

// only in Student class

void display()

{

// will invoke or call current

// class message() method

message();

// will invoke or call parent

// class message() method

super.message();

}

}

// Driver Program

class Test {

public static void main(String args[])

{

Student s = new Student();

// calling display() of Student

s.display();

}

}

3

The super [keyword](https://www.geeksforgeeks.org/list-of-all-java-keywords/)can also be used to access the parent class constructor. One more important thing is that ‘super’ can call both parametric as well as non-parametric constructors depending upon the situation.

class Person {

Person()

{

System.out.println("Person class Constructor");

}

}

// subclass Student extending the Person class

class Student extends Person {

Student()

{

// invoke or call parent class constructor

super();

System.out.println("Student class Constructor");

}

}

// Driver Program

class Test {

public static void main(String[] args)

{

Student s = new Student();

}

}

-for constructor super keyword in child class must be first statement

**Note: super keyword is use to call parent from child.**

**-this keyword**

‘this’ is a reference variable that refers to the current object.

-enum

java Enums are classes that have a fixed set of constants or variables that do not tend to change.

is achieved using keyword enum.

java enum constants are static and final implicitly.

**difference between class and enum.**

|  |  |
| --- | --- |
| class | enum |
| class constants can be overridden | enum constants cannot be overridden |
| classes support creation of objects | enum cannot support creation of objects |
| classes can extends others classes | enum cannot extends other classes |
| classes can implements interface | enum can implements interface |

**syntax**

enum variable\_name{constants}

eg

enum days{Monday, Sunday}

**define enum**

-we can define enum within or outside class.

enum bike

{

BAJAJ, KTM, YAMAHA

}

public class Edureka

{

p s v m(s[] args)

{

bike c1=bike.YAMAHA;

sopl(c1);

}

}

enum with switch

enum Day

{

SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THRISDAY, FRIDAY, SATURDAY;

}

public class Edureka

{

Day day;

public class Edureka

{

Day day;

public Edureka(Day day)

{

this.day=day;

}

public void dayIsLike()

{

switsh(day)

{

case MONDAY;

sopln(“Hi, Today is Monday”);

break;

case TUESDAY;

sopln(“Hi, Today is Tuesday”);

break;

case WEDNESDAY;

sopln(“Hi Today is Wednesday”);

break;

case TUESDAY;

sopln(“Hi Today is Tuesday)’;

break;

case FRIDAY’

sopln(“Hi Today is Friday);

break;

case SATURDAY;

sopln(“Hi Today is Saturday);

break;

case Default;

sopln(“Enter valid Day”);

break;

}

}

**Inheritance using ENUM.**

public class Edureka{

p s v m(s[] a){

List<HTTPMethodConvertible> inherit=new ArrayList<>();

inherit.add(LibraryEnum.FIRST);

inherit.add(ClientEnum.google);

for(HTTPMethodConvertible element: inherit){

sopln(element.getHTTPMethodType());

}

}

static interface HTTPMethodConvertible {

public String getHTTPMethodType();

}

static enum LibraryEnum implements HTTPMethodConvertible {

FIRST(“Goggle Pixel”), SECOND(“Huawei”), THIRD(“Apple 11 Pro”);

String httpMethodType;

LibraryEnum(String phone) {

httpMethodType=phone;

}

public String getHTTPMethodType(){

return httpMethodType;

}

}

static enum ClientEnum implements HTTPMethodConvertible {

huawei(“Hoonging OS”), apple(“IOS”), google(“Android One”);

String httpMethodType;

ClientEnum(String os){

httpMethodType=os;

}

public String getHTTPMethodType(){

return httpMethodType;

}

}

}

enum TrafficSingle{

RED(“Stop”), GREEN(“Go”), ORANGE(“Wait”);

private String action;

public String getAction() {

return this.action=action;

}

private TrafficSignal(String action) {

this.action=action;

}

}

public class Edureka{

p s v m(s[] a){

TrafficSignal[] signal=TrafficSignal.values();

for(TrafficSignal signal:signal){

sopln(“name:” +signal.name()+ “action:”+ signal.getAction());

}

}

}

**if else with ENUM**

enum Directions{

EAST, WEST, NORTH, SOUTH

}

public class Edureka{

p s v m(String[] args) {

Directions dir=Directions.NORTH;

if(dir==Directions.EAST){

sopln(“Direction EAST”);

}

else if(dir==Direction.WEST){

sopln(“direction west”)

}

--//--

else{

sopln(“direction south”);

}

}

**#Methods of ENUM.**

Java compiler internally adds values(), valueOf() and ordinal() methods within the enum at compile time. It internally creates a static and final class for the enum.

**What is the purpose of the values() method in the enum?**

The Java compiler internally adds the values() method when it creates an enum. The values() method returns an array containing all the values of the enum.

**What is the purpose of the valueOf() method in the enum?**

The Java compiler internally adds the valueOf() method when it creates an enum. The valueOf() method returns the value of given constant enum.

**What is the purpose of the ordinal() method in the enum?**

The Java compiler internally adds the ordinal() method when it creates an enum. The ordinal() method returns the index of the enum value.

enum Color {

    RED,

    GREEN,

    BLUE;

}

public class Test {

    public static void main(String[] args)

    {

        // Calling values()

        Color arr[] = Color.values();

        // enum with loop

        for (Color col : arr) {

            // Calling ordinal() to find index

            // of color.

            System.out.println(col + " at index "

                               + col.ordinal());

        }

        // Using valueOf(). Returns an object of

        // Color with given constant.

        // Uncommenting second line causes exception

        // IllegalArgumentException

        System.out.println(Color.valueOf("RED"));

        // System.out.println(Color.valueOf("WHITE"));

    }

}

**enum and constructor:**

* enum can contain a constructor and it is executed separately for each enum constant at the time of enum class loading.
* We can’t create enum objects explicitly and hence we can’t invoke enum constructor directly.

**enum and methods:**

* enum can contain both **concrete** methods and **abstract** methods. If an enum class has an abstract method, then each instance of the enum class must implement it

enum Color {

    RED,

    GREEN,

    BLUE;

    // enum constructor called separately for each

    // constant

    private Color()

    {

        System.out.println("Constructor called for : "

                           + this.toString());

    }

    public void colorInfo()

    {

        System.out.println("Universal Color");

    }

}

public class Test {

    // Driver method

    public static void main(String[] args)

    {

        Color c1 = Color.RED;

        System.out.println(c1);

        c1.colorInfo();

    }

}

**#Advantages of enum.**

-improve type safety.

-easily usable in switch cases.

-can be traversed.

-has fields, methods, constructors.

-can implements interfaces.