JAVA introduction:-

Author : James Gosling

Vendor : Sun Micro System(which has since merged into Oracle Corporation)

Project name : Green Project

Type : open source & free software

Initial Name : OAK language

Present Name : java

Extensions : .java & .class & .jar

Initial version : jdk 1.0 (java development kit)

Present version : java 8 2014

Operating System : multi Operating System

Implementation Lang : c, cpp.....

Symbol : coffee cup with saucer

Objective : To develop web applications

SUN : Stanford Universally Network

Slogan/Motto : WORA(write once run anywhere)

Importance of Core JAVA:- According to the SUN 3 billion devices run on the java language only.

1) Java is used to develop Desktop Applications such as MediaPlayer, Antivirus etc.

2) Java is Used to Develop Web Applications such as sravyajobs.com, irctc.co.in etc.

3) Java is Used to Develop Enterprise Application such as Banking applications.

4) Java is Used to Develop Mobile Applications.

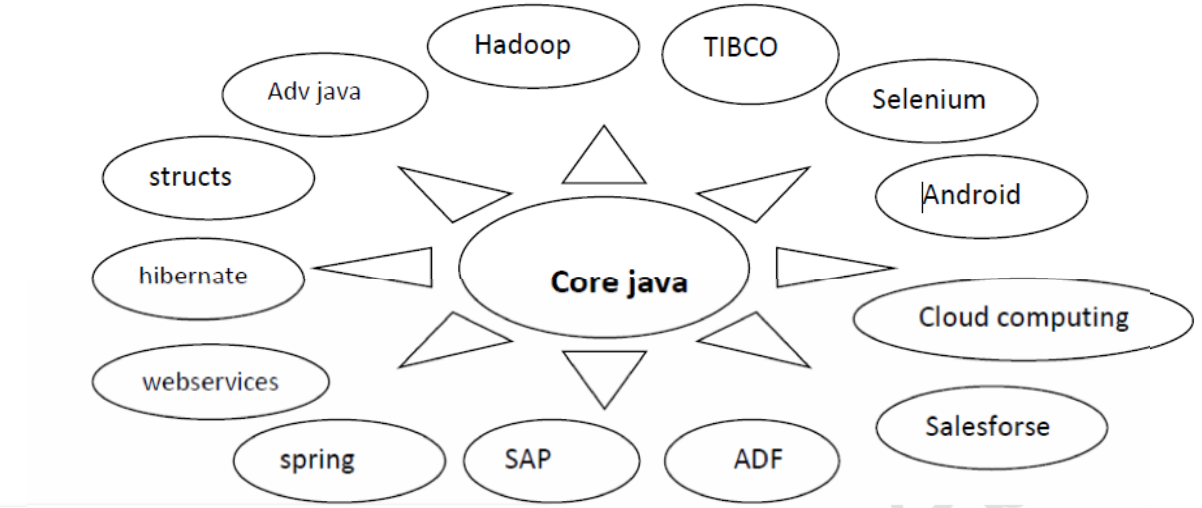
5) Java is Used to Develop Embedded System.

6) Java is Used to Develop SmartCards.

7) Java is Used to Develop Robotics.

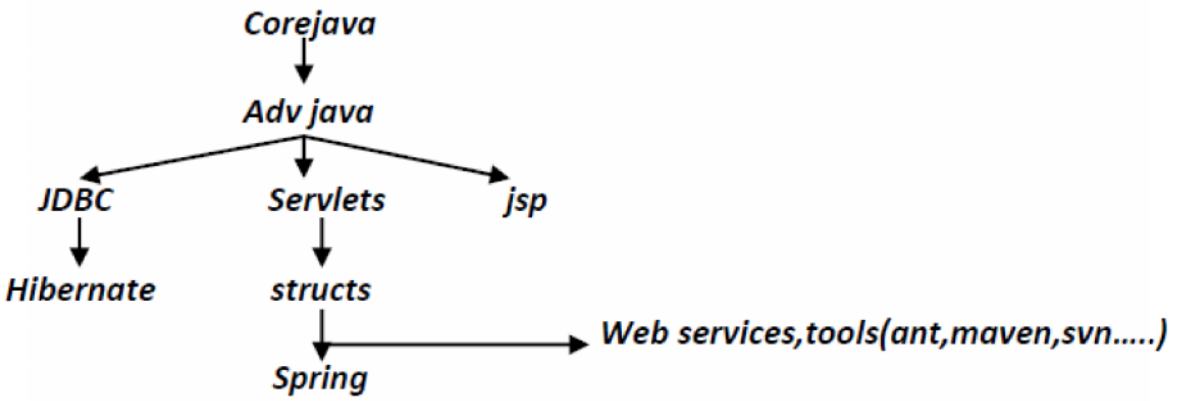
8) Java is used to Develop Games ..... etc.

Technologies Depends on Core java:-



Learning process of java:-

-



Parts of the JAVA:- As per the sun micro system standard the java language is divided into three parts

1) J2SE/JSE(java 2 standard edition)

2) J2EE/JEE(java 2 enterprise edition)

3) J2ME/JME(java 2 micro edition)

Java keywords:-(50)

Data Types

byte
short
int
long
float
double
char
boolean
(8)

Flow-Control:-

if
else
switch
case
default
break
for
while
do
continue
(10)

Predefined constants

method-level:-

void
return
(2)

Object-level:-

new
this
super
instanceof
(4)

source-file:

class
extends
interface
implements
package
import
(6)

True, false, null (3)

Exception *handling:-*

try
catch
finally
throw
throws
(5)

1.5 version:-

enum
assert
(2)

unused:-

goto
const
(2)

Modifiers:-

public
private
protected
abstract
final
static
strictfp
native
transient
volatile
synchronized
(11)

Differences between C & CPP & JAVA:- C-lang

#include<stdio.h>
Void main()
{
Printf("Hello");
}
	#include<iostream.h>
	Void main() { Cout<<"hello world"; }
		Import java.lang.System;
		Import java.lang.String;
		Class Test {
		Public static void main (String [] args)
		{
		System.out.println ("Hello Java");
		}
		}

Author: Dennis Ritchie	**Author : Bjarne Stroustrup**	**Author : James Gosling**
Implementation languages: COBOL,FORTRAN,BCPL, B...	implementation languages: c ,ada,ALGOL68.....	implementation languages C,CPP,ObjectiveC...
In c-lang the predefined support	cpp language the predefined is	In java predefined support is

is available in the form of header files. Ex:- stdio.h , conio.h The header files contain predefined functions. Ex:- printf,scanf.....	maintained in the form of header files. Ex:- iostream.h The header files contains predefined functions. Ex:- cout,cin....	available in the form of packages. Ex: java.lang, java.io,java.awt The packages contains predefined classes&interfaces and these class & interfaces contains predefined methods

JAVA Features :-

1. Simple
2. Object Oriented
3. Platform Independent
4. Architectural Neutral
5. Portable
6. Robust
7. Secure
8. Dynamic
9. Distributed
10. Multithread
11. Interpretive
12. High Performance

1. Simple:-

Java is a simple programming language because,

- Java technology has eliminated all the difficult and confusion oriented concepts like pointers, multiple inheritance in the java language.
- Java uses c,cpp syntaxes mainly hence who knows C,CPP for that java is simple language.

2. Object Oriented:-

- Java is object oriented technology because it is representing total data of the class in the form of object.
- The languages which are support object, class, Inheritance, Polymorphism, Encapsulation, Abstraction those languages are called Object oriented.

3. Platform Independent :-

When we compile the application by using one operating system (windows) that Compiled file can execute only on the same operating system(windows) this behavior is called platform dependency.

Example:- C,CPP ...etc

When we compile the application by using one operating system (windows) that Compiled file can execute in all operating systems(Windows,Linux,Mac...etc) this behavior is called platform independency.

Example:- java, Ruby, Scala, PHP ...etc

4. Architectural Neutral:-

Java tech applications compiled in one Architecture/hardware (RAM, Hard Disk) and that Compiled program runs on any architecture (hardware) is called Architectural Neutral.

5. Portable:-

In Java the applications are compiled and executed in any OS (operating system) and any Architecture (hardware) hence we can say java is a portable language.

6. Robust:- Any technology good at two main areas that technology is robust technology.

a. Exception Handling

b. Memory Allocation Java is providing predefined support to handle the exceptions. Java provides Garbage collector to support memory management.

7. Secure:-

- To provide implicit security Java provides one component inside JVM called Security Manager.
- To provide explicit security for the Java applications we are having very good predefined library in the form of java.security package.

8. Dynamic:-

Java is dynamic technology it follows dynamic memory allocation (at runtime the memory is allocated).

9. Distributed:-

By using java it is possible to develop distributed applications & to develop distributed applications java uses RMI,EJB...etc

10. Multithreaded: -

- Thread is a light weight process and a small task in large program.
- In java it is possible to create user thread & it possible to execute simultaneously is called multithreading.
- The main advantage of multithreading is it shares the same memory & threads are important at multimedia, gaming, web application.

11. High Performance:-

If any technology having features like Robust, Security, Platform Independent, Dynamic and so on then that technology is high performance.

Types of java applications:-

1. Standalone applications:

- It is also known as window based applications or desktop applications.
- This type of applications must install in every machine like media player, antivirus ...etc
- By using AWT & Swings we are developing these type of applications.
- This type of application does not required client-server architecture.

2. Web applications:

- The applications which are executed at server side those applications are called web applications like Gmail, facebook , yahoo...etc
- All applications present in internet those are called web-applications.
- The web applications required client-server architecture.

  i. Client : who sends the request.

  ii. Server : it contains application & it process the app & it will generate response.

  iii. Database : used to store the data.
- To develop the web applications we are using servlets ,structs ,spring...etc

3. Enterprise applications:-

- It is a business application & most of the people use the term it I big business application.
- Enterprise applications are used to satisfy the needs of an organization rather than individual users. Such organizations are business, schools, government ...etc
- An application designed for corporate use is called enterprise application.
- An application in distributed in nature such as banking applications.
- All j2ee & EJB is used to create enterprise application.

4. Mobile applications:-

- The applications which are design for mobile platform are called mobile applications.
- We are developing mobile applications by sing android,IOS,j2me...etc
- There are three types of mobile applications
  - Web-application (gmail ,online shopping,oracle ...etc)
  - Native (run on device without internet or browser)ex: phonecall, calculator, alaram, games These are install from application storec& to run these apps internet not required.

- Hybrid (required internet data to launch) ex:whats up, facebook, LinkedIn...etc

  These are installed form app store but to run this application internet data required.

5. Distributed applications:-

Software that executes on two or more computers in a network. In a client-server environment. Application logic is divided into components according to function.

Ex : aircraft control systems ,industrial control systems, network applications...etc

Types of software :

The set of instructions that makes the computer system do something.

1) Application software

The program that allows the user to perform particular task. Ex: business apps, entertainment apps,ms-office...etc

2) System software

The programs that allow the hardware to run properly. Ex: operating system, device drivers...etc

Java coding conventions:-

Classes:-

- Class name start with upper case letter and every inner word starts with upper case letter.
- This convention is also known as camel case convention.
- The class name should be nouns.

  Ex:- String StringBuffer, InputStreamReader .....etc

Interfaces :-

- Interface name starts with upper case and every inner word starts with upper case letter.
- This convention is also known as camel case convention.
- The class name should be nouns. Ex: Serializable Cloneable RandomAccess

Methods :-

- Method name starts with lower case letter and every inner word starts with upper case letter.
- This convention is also known as mixed case convention
- Method name should be verbs.

  Ex:- post() charAt() toUpperCase() compareToIgnoreCase()

Variables:-

- Variable name starts with lower case letter and every inner word starts with upper case letter.
- This convention is also known as mixed case convention. Ex :- out in pageContext

Package :-

- Package name is always must written in lower case letters.

Ex :- java.lang java.util java.io ...etc

Constants:-

- While declaring constants all the words are uppercase letters .

Ex: MAX_PRIORITY MIN_PRIORITY NORM_PRIORITY

NOTE:- The coding standards are mandatory for predefined library & optional for user defined library but as a java developer it is recommended to follow the coding standards for user defined library also.

Class Elements:-

```
Class Test{
1. variables          int a = 10;
2. methods            void add() {business logic }
3. constructors       Test() {business logic }
4. instance blocks    {business logic }
5. static blocks      static {business logic }
}
```

Java Comments:-

Comments are used to write the detailed description about application logics to understand the logics easily.

Comments are very important in real time because today we are developing the application but that application maintained by some other person so to understand the logics by everyone writes the comments.

Comments are non-executable code these are ignored at compile time.

There are 3 types of comments.

1) Single line Comments:-

By using single line comments it is possible to write the description about our programming logics within a single line & these comments are Starts with // (double slash) symbol.

Syntax:- //description

2) Multi line Comments:-

This comment is used to provide description about our program in more than one line & these commands are start with /* & ends with */

Syntax: - /*----satement-1 ----statement-2 */

3) Documentation Comments:-

By using documentation comments it possible to prepare API(Application programming interface) documents.

Syntax: -

```
/*
*statement-1
*statement-2
*/
```

Example:-

```
/*project name:-green project
team size:- 6 team
lead:- Rajesh*/
class Test //class declaration
{ //class starts
public static void main(String[] args) // execution starting point
{ //main starts
        System.out.println("Hello"); //printing statement
        }//main ends
        } //class ends
```

Separators in java:-

Symbol	name	usage
( )	parentheses	used to contains list of parameters & contains expression.
{ }	braces	block of code for class, method, constructors & local scopes.
[ ]	brackets	used for array declaration.
;	semicolon	terminates statements.
,	comma	separate the variables declaration & chain statements
.	period	used to separate package names from sub packages. And also used for separate a variable,method from a reference type.

Print() vs Println ():-

- **Print():-** used to print the statement in console and the control is present in the same line.
  Example:- System.out.print("HelloWorld");
  System.out.print("core java");
  Output:-HelloWorldcorejava

- **Println():-** used to print the statements in console but the control is there in next line.
  Example:- System.out.println("HelloWorld");
  System.out.println("core java");
  Output: - HelloWorld
  Core java

Java Tokens:-

- Smallest individual part of a java program is called Token.
- It is possible to provide any number of spaces in between two tokens.

Escape Sequences:-

A character preceded by a backslash (\) is an escape sequence and has special meaning to the compiler. The following table shows the Java escape sequences

Escape Sequences

Escape Sequence	Description
\t	Insert a tab in the text at this point.
\b	Insert a backspace in the text at this point.
\n	Insert a newline in the text at this point.
\r	Insert a carriage return in the text at this point.
\f	Insert a formfeed in the text at this point.
\'	Insert a single quote character in the text at this point.
\"	Insert a double quote character in the text at this point.
\\	Insert a backslash character in the text at this point.

Java identifiers:-

Every name in java is called identifier such as,

- Class-name
- Method-name
- Variable-name

Rules to declare identifier:

1. An identifier contains group of Uppercase & lower case characters, numbers ,underscore & dollar sign characters but not start with number.

int abc=10; ---> valid int _abc=30; ---> valid int $abc=40; ---> valid

int a-bc=50; --->not valid int 2abc=20; ---> Invalid int not/ok=100 --->invalid

2. Java identifiers are case sensitive of course java is case sensitive programming language. The below three declarations are different & valid.

```
class Test
{
int NUMBER=10;
int number=10;
int Number=10;
}
```

3. The identifier should not duplicated & below example is invalid because it contains duplicate variable name.

```
class Test
{
int a=10;
int a=20;
}
```

4. In the java applications it is possible to declare all the predefined class names & interfaces names as a identifier but it is not recommended to use.

```
class Test
{
public static void main(String[] args)
{
int String=10;
float Exception=10.2f;
System.out.println(String);
System.out.println(Exception);
}
}
```

5. It is not possible to use keywords as a identifiers.

```
class Test
{
int if=10;
int while=20;
}
```

6. There is no length limit for identifiers but is never recommended to take lengthy names because it reduces readability of the code.

Java primitive Data Types:-

1. Data types are used to represent type of the variable & expressions.
2. Representing how much memory is allocated for variable.
3. Specifies range value of the variable.

There are 8 primitive data types in java

Data Type	size(in bytes)	Range	default values
byte	**1**	**-128 to 127**	**0**
short	**2**	**-32768 to 32767**	**0**
int	**4**	**-2147483648 to 2147483647**	**0**
long	**8**	**–9,223,372,036,854,775,808 to 9 ,223,372,036,854,775,807**	**0**
float	**4**	**-3.4e38 to 3.4e**	**0.0**
double	**8**	**-1.7e308 to 1.7e308**	**0.0**
char	**2**	**0 to 6553**	**single space**
Boolean	**no-size**	**no-range**	**false**

Byte :-
Size : 1-byte
MAX_VALUE : 127
MIN_VALUE : -128
Range : -128 to 127
Formula : -2^n to 2^{n-1} -2^8 to 2^{8-1}

Note :-

- To represent numeric values (10,20,30…etc) use **byte,short,int,long.**
- To represent decimal values(floating point values 10.5,30.6…etc) use **float,double.**
- To represent character use **char** and take the character within single quotes.
- To represent true ,false use **Boolean.**

Except Boolean and char remaining all data types consider as a signed data types because we can represent both +ve & -ve values.

Float vs double:-

- Float will give 5 to 6 decimal places of accuracy but double gives 14 to 15 places of accuracy.
- Float will fallow single precision but double will fallow double precision.

Syntax:- data-type name-of-variable = value/literal;
Ex:- int a=10;
int →Data Type
a →variable name
= →assignment
10 →constant value
; → statement terminator

Java flow control Statements

There are three types of flow control statements in java
a. Selection Statements
b. Iteration statements
c. Transfer statements

1. Selection Statements
a. If
b. If-else
c. Switch
d. else-if

If syntax:-

```
if (condition)
{ true body;
}
else
{
false body;
}
```

If statmenet is taking condition that condition must be Boolean condition. Otherwise compiler will generate error message.

Example-1:- Normal example

```
class Test
{
public static void main(String[] args)
{
 int marks=20;
if (marks>22)
{
System.out.println("if body / true body");
}
else
{
System.out.println("else body/false body ");
}
}
}
```

Example -2:- For the if the condition it is possible to provide Boolean values.

```
class Test
{
 public static void main(String[] args)
{
if (true)
{
System.out.println("true body");
}
else
{
 System.out.println("false body");
}
}

}
```

Example-3:-in c-language 0-false & 1-true but these conventions are not allowed in java.

```
class Test
{
public static void main(String[] args)
{
 if (0)
{
System.out.println("true body");
}
else
{
 System.out.println("false body");
}
}
}
```

Example-4:

```
class Test
{
public static void main(String[] args)
{
 int a=20;
if (a=10)
{
System.out.println("true body");
}
else
{
System.out.println("false body");
}
}
}
```

error: incompatible types: int cannot be converted to Boolean

Example-5:- The curly braces are options but without curly braces it is possible to take only one statement that should not be a initialization.

```
class Test
{
public static void main(String[] args)
{
if (true)
System.out.println("true body");
else
System.out.println("false body");
}
}
```

Switch statement:-

- Switch statement is used to declare multiple selections
- Switch is taking the argument, the allowed arguments are
  - **Byte,short,int,char (primitives)**
  - **Byte, Short,Integer,Character,enum (1.5 version)**
  - **String (1.7 version).**
- Inside the switch it is possible to declare more than one case but it is possible to declare only one default.
- Based on the provided argument the matched case will be executed if the cases are not matched default will be executed.
- While declaring switch statement braces are mandatory otherwise compiler will generate error message.

Note : Float and double and long is not allowed for a switch argument because these are having too large values.

Syntax:-

```
switch(argument)
{
case label1 : statements;
break;
case label2 : statements;
break;
.
.
.
default : statements;
break;
}
```

Example-1: Normal input and normal output.

```
class Test
{ public static void main(String[] args)
{ int a=10;
switch (a)
{ case 10:System.out.println("Ten");
break;
case 20:System.out.println("Twenty");
break;
case 30:System.out.println("Thirty");
break;
default:System.out.println("default");
break;
}
}
}
```

Example-2: Inside the switch the case labels must be unique; if we are declaring duplicate case labels the compiler will raise compilation error "duplicate case label".

```
class Test
{ public static void main(String[] args)
{ int a=10;
switch (a)
{ case 10:System.out.println("Ten");
break;
case 10:System.out.println("10");
break;
default:System.out.println("default");
break;
}
}
}
```

Example-3: Inside the switch for the case labels & switch argument it is possible to provide expressions (10+10+20, 10*4, 10/2).

```
class Test
{ public static void main(String[] args)
{ int a=99;
switch (a+1)
{ case 10+20+70 :System.out.println("100");
break;
case 10+5 :System.out.println("15");
break;
case 30/6 :System.out.println("20");
break;
default :System.out.println("default");
break;
}
}
}
```

Example-4:- Inside the switch the case label must be constant values. If we are declaring variables as a case labels the compiler will show compilation error "constant expression required".

```
class Test
{ public static void main(String[] args)
{ int a=10;
Int b=20;
switch (a)
{ case a:System.out.println("a"); break;
case b:System.out.println("b"); break;
default:System.out.println("default"); break;
}
}
}
```

Example 5: It is possible to declare final variables as a case label. Because the variables are replaced with constants during compilation.

```
class Test
{ public static void main(String[] args)
{ final int a=10;
switch (a)
{ case a:System.out.println("final"); break;
default:System.out.println("default"); break;
}
}
}
```

Example-6:- Inside the switch the default is optional.

```
class Test
{ public static void main(String[] args)
{ int a=10;
switch (a)
{ case 10:System.out.println("10");
break;
}
}
};
```

Example 7:- Inside the switch cases are optional part.

```
class Test
{ public static void main(String[] args)
{ int a=10;
switch (a)
{ default: System.out.println("default");
break;
}
}
};
```

Example 8:- inside the switch both cases and default Is optional.

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

Example -9:- Inside the switch independent statements are not allowed. If we are declaring the statements that statement must be inside the case or default.

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

Example-10:- Internal conversion for unicode. Unicode values a-97 A-65

```
class Test
{ public static void main(String[] args)
{ int a=65;
switch (a)
{ case 66:System.out.println("10"); break;
case 'A':System.out.println("20"); break;
default: System.out.println("default"); break;
}
}
}
```

Example -11: Internal conversion of unicodes.

```
class Test
{ public static void main(String[] args)
{ char ch='d';
switch (ch)
{ case 100:System.out.println("10"); break;
case 'A':System.out.println("20"); break;
default: System.out.println("default"); break;
}
}
}
```

Example-12:- inside the switch the case label must match with provided argument data type otherwise compiler will raise compilation error "incompatible types".

```
class Test
{ public static void main(String[] args)
{ char ch='a';
switch (ch)
{ case "aaa" :System.out.println("aaa"); break;
case 65 :System.out.println("65"); break;
case 'a' :System.out.println("a"); break;
}
}
}
```

Example -13:-

- Inside the switch statement break is optional.
- If we are not providing break statement then from the matched case onwards up to break statement will be executed, if there is no break statement then end of the switch will be executed. This situation is called as fall though inside the switch case.

```
class Test
{ public static void main(String[] args)
{ int a=10;
switch (a)
{ case 10:System.out.println("10");
case 20:System.out.println("20");
case 40:System.out.println("40");
break;
default: System.out.println("default");
break;
}
}
}
```

Example -14: The advantage of fall through is used to executed common actions for multiple cases.

```
class Test
{ public static void main(String[] args)
{ int a=2;
switch (a)
{ case 1:
case 2:
case 3:System.out.println("Question-1");
break;
case 4:
case 5:
case 6:System.out.println("Question-2");
break;
}
}
}
```

Example-15:- Inside the switch it is possible to declare the default statement at starting or middle or end of the switch.

```
class Test
{ public static void main(String[] args)
{ int a=100;
switch (a)
{ default: System.out.println("default");
case 10:System.out.println("10");
}
}
}
```

Example -16:- The below example compiled and executed in 1.7 version & above because switch is taking String argument from 1.7 version.

```
class test
{ public static void main(String[] args)
{ String str = “aaa”;
switch (str)
{ case "aaa" : System.out.println("aaa");
case "bbb" : System.out.println("bbb");
default : System.out.println("default");
}
}
}
```

Example -17:- Inside switch the case labels must be within the range of provided argument data type otherwise compiler will raise compilation error “possible loss of precision”.

```
class Test
{ public static void main(String[] args)
{ byte b=126;
switch (b)
{ case 126:System.out.println("20");
case 127:System.out.println("30");
case 128:System.out.println("40");
default: System.out.println("default");
}
}
}
```

Else-if lader:-

```
class Test
{ public static void main(String[] args)
{ int a=20;
if (a==10)
{ System.out.println("ten");
}
else if (a==20)
{ System.out.println("twenty");
}
else if (a==30)
{ System.out.println("thirty");
}
else
{ System.out.println("default condition");
}
}
}
```

Iteration Statements:-

By using iteration statements we are able to execute group of statements repeatedly or more number of times.

1) For
2) For-each
3) while
4) do-while

for syntax:-

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

With out for loop

```
class Test
{ public static void main(String[] args)
{ System.out.println("Hello Java");
System.out.println("Hello Java");
System.out.println("Hello Java");
System.out.println("Hello Java");
}
}
```

By using for loop

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

Initialization part of for loop:-

Example- 1: Inside the for loop initialization part is optional.

```
class Test
{ public static void main(String[] args)
{ int i=0;
for (;i<10;i++)
{ System.out.println("Hello For Loop");
}
}
}
```

Example-2:- In the initialization part it is possible to take any number of System.out.println("First") statements and each and every statement is separated by comma(,) .

```
class Test
{ public static void main(String[] args)
{ int i=0;
for (System.out.println("Zero"), System.out.println("First");i<10;i++)
{ System.out.println("Value of i"+i);
}
}
}
```

Example 3:- compilation error more than one initialization not possible.

```
class Test
{ public static void main(String[] args)
{ for (int i=0,double j=10.8;i<10;i++)
{ System.out.println("Test loop");
}
}
}
```

Ex :-declaring two variables are possible.

```
class Test
{ public static void main(String[] args)
{ for (int i=0,j=0;i<10;i++)
{ System.out.println("Test Loop");
}
}
}
```

Conditional part of for loop:-

Example 1:- Inside for loop conditional part is optional if we are not providing condition at the time of compilation compiler will provide true value.

```
for (int i=0; ;i++)
{ System.out.println("Test Loop");
}
```

Increment/decrement:-

Example-1:- Inside the for loop increment/decrement part is optional.

```
class Test
{ public static void main(String[] args)
{ for (int i=0; i<10 ; )
{ System.out.println("Test Loop");
}
}
}
```

Example 2:- in the increment/decrement it is possible to take the any number of SOP() statements and each and every statement is separated by comma(,).

```
for(int i=0;i<10;System.out.println("first"),System.out.println("second"))
{ System.out.println("Test Loop");
i++;
}
```

Note : Inside the for loop each and every part is optional.

for(;;)→represent infinite loop because the condition is always true.

unreachable statement:-

Ex:- compiler is unable to identify the unreachable statement.

```
class Test
{ public static void main(String[] args)
{ for (int i=1;i>0;i++)
{ System.out.println("Inside Loop");
}
System.out.println("rest of the code");
}
}
```

Ex:- compiler able to identify the unreachable Statement.

```
class Test
{ public static void main(String[] args)
{ for (int i=1;true;i++)
{ System.out.println("Inside Loop");
}
System.out.println("rest of the code");
}
}
```

Note:- When you provide the condition even though that condition is represent infinite loop compiler is unable to find unreachable statements,(because that compiler is thinking that condition may fail).

When you provide Boolean value as a condition then compiler is identifying unreachable statement because compiler knows that condition never change.

While loop:-

Syntax:- while (condition) //condition must be Boolean & mandatory.

{ body;

}

Example-1 :-

```
class Test
{ public static void main(String[] args)
{ int i=0;
while (i<10)
{ System.out.println("Inside Loop");
i++;
}
}
}
```

Example-2 :- compilation error unreachable statement.

```
class Test
{ public static void main(String[] args)
{ int i=0;
while (false)
{ System.out.println("Inside Loop"); //unreachable statement
i++;
}
} }
```

Do-While:-

If we want to execute the loop body at least one time them we should go for do-while statement.

- In do-while first body will be executed then only condition will be checked.
- In the do-while the while must be ends with semicolon otherwise compilation error.

Syntax:- do
{ //body of loop
} while (condition);

Example-1:-

```
class Test
public static void main(String[] args)
{ int i=0;
do
{ System.out.println("Inside Loop");
i++;
}while (i<10);
}
}
```

Example-2 :- unreachable statement

```
class Test
{ public static void main(String[] args)
{ int i=0;
do
{ System.out.println("Inside Loop");
}while (true);
System.out.println("Unreachable Line"); //unreachable statement
}
}
```

Example-3 :-

```
class Test
{ public static void main(String[] args)
{ int i=0;
do
{ System.out.println("Do Loop");
}while (false);
System.out.println("Out Side Loop");
}
}
```

For-each loop:- (introduced in 1.5 version)

- For loop is used to print the data & it is possible to apply conditions.
- For-each loop is used to print the data but it is not possible to apply the conditions. To print the data starting element to ending element without conditions use for-each loop.

```
class Test
{ public static void main(String[] args)
{ int[] a={10,20,30,40};
//printing data by uisng for-loop
for (int i=0;i<a.length;i++)
{ System.out.println(a[i]);
}
//printing data by using for-each loop
for (int aa: a)
{ System.out.println(aa);
}
}
};
```

Iterator vs Iterable:-

- The target elements in for-each loop should be iterable. And it is interface present in java.lang package.
- If an object is iterable that class should implements Iterable interface.
- Iterator is cursor we will discussed in Collections framework.

Transfer statements:-
By using transfer statements we are able to transfer the flow of execution from one position to another position**.**

- **break**
- **continue**
- **return**
- **try**
- **goto**

break:-
Break is used to stop the execution, and is possible to use the break statement only in two areas.
a. Inside the switch statement.
b. Inside the loops.

Example-1 :- break means stop the execution and come out of loop.

```
class Test
{ public static void main(String[] args)
{ for (int i=0;i<10;i++)
{ if (i==5)
break;
System.out.println(i);
}
}
}
```

Example-2 :- if we are using break outside switch or loops the compiler will raise compilation error **"break outside switch or loop**"

```
class Test
{ public static void main(String[] args)
{ if (true)
{ System.out.println("Inside If");
break;
System.out.println("Hello");
}
}
}
```

Continue:-
Skip the current iteration and it is continue the rest of the iterations normally.

```
class Test
{ public static void main(String[] args)
{ for (int i=0;i<10;i++)
{ if (i==5)
continue;
System.out.println(i);
}
}
}
```

Java Variables:-

- Variables are used to store the constant values by using these values we are achieving project requirements.
- Variables are also known as **fields** of a class or **properties** of a class.
- All variables must have a type. You can use primitive types such as int, float, boolean, etc. or array type or class type or enum type or interface type.
- Variable declaration is composed of three components in order,
    - o Zero or more modifiers.
    - o The variable type.
    - o The variable name.

Example : public final int x=100;
public int a=10;
public ----> modifier (specify permission)
int ----> data type (represent type of the variable)
a ----> variable name
10 ----> constant value or literal;
; ----> statement terminator

There are three types of variables in java
1. Local variables.
2. Instance variables.
3. Static variables.

Local variables:-

- The variables which are declare inside a **method or constructor or blocks** those variables are called local variables.

```
class Test
{ public static void main(String[] args) //execution starts from main method
{ int a=10; //local variables
int b=20;
System.out.println(a);
System.out.println(b);
}
}
```

- It is possible to access local variables only inside the method or constructor or blocks only, it is not possible to access outside of method or constructor or blocks.

```
void add()
{ int a=10; //local variable
System.out.println(a); //possible
}
void mul()
{ System.out.println(a); //not-possible
}
```

For the local variables memory allocated when method starts and memory released when method completed. The local variables are stored in stack memory.

Instance variables (non-static variables):-

- The variables which are declare inside a class but outside of methods those variables are called instance variables.
- The scope (permission) of instance variable is inside the class having global visibility.
- For the instance variables memory allocated during object creation & memory released when object is destroyed.
- Instance variables are stored in heap memory.

Areas of java language:-

There are two types areas in java.
1) Instance Area.
2) Static Area.

Instance Area:-

```
void m1() //instance method
{ Logics here //instance area
}
```

Static Area:-

```
Static void m1() //static method
{ Logics here //static area
}
```

Instance variable accessing:-

(Instance variables & methods)

Example:-

```
class Test
{ //instance variables
int a=10;
int b=20;
//static method
public static void main(String[] args)
{ //Static Area
Test t=new Test();
System.out.println(t.a);
System.out.println(t.b);
t.m1(); //instance method calling
}
// instance method
void m1() //user defined method must called by user inside main method
{ //instance area
System.out.println(a);
System.out.println(b);
}//main ends
};//class ends
```

Static variables (class variables):-

- The variables which are declared inside the class but outside of the methods with static modifier those variables are called static variables.
- Scope of the static variables with in the class global visibility.
- Static variables memory allocated during .class file loading and memory released at .class file unloading time.
- Static variables are stored in non-heap memory.

Static variables & methods accessing:-
(Static variables& static methods)

```
class Test
{ //static variables
static int a=1000;
static int b=2000;
public static void main(String[] args) //static method
{ System.out.println(Test.a);
System.out.println(Test.b);
Test t = new Test();
t.m1(); //instance method calling
}

//instance method
void m1() //user defined method called by user inside main method
{ System.out.println(Test.a);
System.out.println(Test.b);
}
}
```

Static variables calling: - We are able to access the static members inside the static area in three ways.

- Direct accessing.
- By using class name.
- By using reference variable.

In above three approaches second approach is best approach .

```
class Test
{ static int x=100; //static variable
public static void main(String[] args)
{ System.out.println(a); //1-way(directly possible)
System.out.println(Test.a); //2-way(By using class name)
Test t=new Test();
System.out.println(t.a); //3-way(By using reference variable)
}
}
```

Example: - When we create object inside method that object is destroyed when method completed, if any other method required object then create the object inside that method.

```
class Test
{ //instance variable
int a=10;
int b=20;
static void m1()
{ Test t = new Test();
System.out.println(t.a);
System.out.println(t.b);
}
static void m2()
{ Test t = new Test();
System.out.println(t.a);
System.out.println(t.b);
}
public static void main(String[] args)
{ Test.m1(); //static method calling
Test.m2(); //static method calling
}
}
```

Example:-

```
class Test
{ int a=10;
int b=20; // instance variables
static int c=30; static int d=40; //static variables
void m1() //instance method
{ System.out.println(a);
System.out.println(b);
System.out.println(Test.c);
System.out.println(Test.d);
}
static void m2() //static method
{ Test t = new Test();
System.out.println(t.a);
System.out.println(t.b);
System.out.println(Test.c);
System.out.println(Test.d);
}
public static void main(String[] args)
{ Test t = new Test();
t.m1(); //instance method calling
Test.m2(); //static method calling
}
}
```

Variables VS default values:-

Case 1:- for the instance variables JVM will assign default values.

```
class Test
{ int a;
boolean b;
public static void main(String[] args)
{ Test t=new Test();
System.out.println(t.a);
System.out.println(t.b);
}
};
```

Case 2:- for the static variables JVM will assign default values.

```
class Test
{ static int a;
static float b;
public static void main(String[] args)
{ System.out.println(Test.a);
System.out.println(Test.b);
}
};
```

Case 3:-

- For the instance and static variables JVM will assign default values but for the local variables the JVM won't provide default values.
- In java before using local variables must initialize some values to the variables otherwise compiler will raise compilation error "variable a might not have been initialized".

```
class Test
{ public static void main(String[] args)
{ int a;
int b;
System.out.println(a);
System.out.println(b);
}
```

Class Vs Object:-

- **Class is a logical entity it contains logics whereas object is physical entity it is representing memory.**
- Class is blue print it decides object creation without class we are unable to create object.
- **Based on single class (blue print) it is possible to create multiple objects but every object occupies memory.**
- We are declaring the class by using class keyword but we are creating object by using new keyword.
- **We will discuss object creation in detailed in constructor concept.**

Instance vs. Static variables:-

In case of instance variables the JVM will create separate memory for each and every object it means separate instance variable value for each and every object.
In case of static variables irrespective of object creation per class single memory is allocated, here all objects of that class using single copy.

Example :-

```
class Test
{ int a=10; //instance variable
static int b=20; //static variable
public static void main(String[] args)
{ Test t = new Test();
System.out.println(t.a); //10
System.out.println(t.b); //20
t.a=111; t.b=222;
System.out.println(t.a); //111
System.out.println(t.b); //222
Test t1 = new Test(); //10 222
System.out.println(t1.a); //10
System.out.println(t1.b); //222
t1.b=444;
Test t2 = new Test(); //10 444
System.out.println(t2.b); //444
}
}
```

Different ways to initialize the variables :-

```
class Test
{ int s=10;
int a,b,c;
int x=10,y,z;
int i=10,j=20,k;
int p=10,q=20,r=30;
public static void main(String[] args)
{ Test t = new Test();
System.out.println(t.s);
System.out.println(t.a+" "+t.b+" "+t.c);
System.out.println(t.x+" "+t.y+" "+t.z);
System.out.println(t.i+" "+t.j+" "+t.k);
System.out.println(t.p+" "+t.q+" "+t.r);
}
}
```

Summary of variables:-

Characteristic	Local variable	instance variable	static variables
where declared	Inside Method or Constructor or block	Inside the class outside of methods	Inside the class outside of methods.
Usage	Within the method	Inside the class	Inside the class all
When memory allocated	When method starts	When object created	When .class file loading
When memory destroyed	When method ends	When object destroyed default values are Assigned by JVM	When .class unloading default values are Assigned by JVM
Initial values	None, must initialize the before first use	Default values are Assigned by JVM	Default values are Assigned by JVM
Relation with Object	No way related to Object	For every object one copy of instance variable created It means memory	For all objects one copy is created Single memory
Accessing	Directly possible	By using object name Test t=new Test()	By using class name System.out.println(Test.a)
Memory	Stored in stack memory	Stored in heap memory	Method area

Java Methods (behaviors):-

- Inside the classes it is not possible to write the business logics directly hence inside the class declare the method inside that method writes the logics of the application.
- Methods are used to write the business logics of the project.
- **Coding convention**: method name starts with lower case letter and every inner word starts with uppercase letter(**mixed case**).

Example:- post() , char**A**t() , to**U**pper**C**ase() , compare**T**o**I**gnore**C**ase()……etc
There are two types of methods in java,
1. Instance method
2. Static method

- Inside the class it is possible to declare any number of instance & static methods based on the developer requirement.
- It will improve the reusability of the code and we can optimize the code.

Note: - Whether it is an instance method or static method the methods are used to provide business logics of the project.

Instance method :-
void m1() **//instance method**
{ //body //**instance area**
}

Note: - for the instance members memory is allocated during object creation hence access the instance members by using object-name (reference-variable).

Method calling Syntax:-

Void m1() { logics here } //instance method
Objectname.instancemethod( ); //calling instance method
Test t = new Test();
t.m1( );

static method:-
static void m1() **//static method**
{ //body **//static area**
}
Note: - for the static member's memory allocated during **.class** file loading hence access the static members by using class-name.

Method calling syntax:-
Static void m2() { logics here } //static method
Classname.staticmethod( ); // call static method by using class name
Test.m2( );

Every method contains three parts.

1. Method declaration
2. Method implementation (logic)
3. Method calling

Example:- void m1() ------> **method declaration**
{ Body (Business logic); -----> **method implementation**
}
Test t = new Test(); t.m1(); ------> **method calling**

Method Syntax:-
[modifiers-list] return-Type Method-name (parameters list) throws Exception

Modifiers-list →represent access permissions→ **[optional]**
Return-type → functionality return value → **[mandatory]**
Method name → functionality name → **[mandatory]**
Parameter-list → input to functionality → **[optional]**
Throws Exception → representing exception handling → **[optional]**

Example:- **Public void m1(){** logics…**}**
Private int m2(int a,int b) { logics…**}**

Method Signature:-
Method-name & parameters list is called method signature.
Syntax:- Method-name(parameter-list)
Example:- m1(int a)
m1(int a,int b)

Example-1 :- instance & static methods without arguments.

- Instance methods are bounded with objects hence call the instance methods by using object name(reference variable).
- Static methods are bounded with class hence access the static methods by using class-name.

```
class Test
{ void m1()
{ System.out.println("m1 instance method");
}
static void m2()
{ System.out.println("m2 static method");
}
public static void main(String[] args)
{ Test t = new Test();
t.m1(); //calling of instance method by using object-name
Test.m2(); //calling of static method by using class-name
}
}
```

Example-2:-instance & static methods with parameters.

- If the method is expecting parameters (inputs to functionality) then while calling that method must pass the values to that parameters then only that method will be executed.
- While passing parameters, number of arguments & order of arguments important.

void m1(int a) -->t.m1(10); -->valid
void m3(int a,char ch,float f) -->t.m3(10,'a',10.6); -->invalid
void m4(int a,char ch,float f) -->t.m4(10,'a',10.6f); -->valid
void m5(int a,char ch,float f) -->t.m3(10,'c'); -->invalid

```
class Test
{ void m1(int a,char ch) //local variables
{ System.out.println("m1 instance method");
System.out.println(a);
System.out.println(ch);
}

static void m2(boolean b,double d)
{ System.out.println("m2 static method");
System.out.println(b);
System.out.println(d);
}

public static void main(String[] args)
{ Test t = new Test();
t.m1(10,'a'); //calling of instance method by passing inputs
Test.m2(true,10.5); //calling of static method by passing inputs
}
}
```

Example-3 :- while calling methods it is possible to provide variables as a argument values.

```
class Test
{ void m1(int a,char ch,boolean b)
{ System.out.println(a);
System.out.println(ch);
System.out.println(b);
}
public static void main(String[] args)
{ int x=100;
char ch='a';
boolean y=false;
Test t = new Test();
t.m1(x,ch,y);
}
}
```

Example-4 :- For java methods it is possible to provide Objects as a parameters(in real time project).

```
class X{}
class Emp{}
class Y{}
class Test
{ void m1(X x ,Emp e)
{ System.out.println("m1 method");
}
static void m2(int a,Y y)
{ System.out.println("m2 method");
}
public static void main(String[] args)
{ Test t = new Test();
X x = new X();
Emp e = new Emp();
t.m1(x,e); //calling of instance method by passing objects as an input
Y y = new Y();
Test.m2(10,y); //calling of static method by passing objects as an input
}
}
```

Main method project code at realtime project level

```
public static void main(String[] args)
{ new Test().m1(new X(),new Emp());
Test.m2(10,new Y());
}
```

Example -5:-

Inside the class it is not possible to declare two methods with same signature , if we are trying to declare two methods with same signature compiler will raise compilation error message **"m1() is already defined in Test "**(Java class not allowed Duplicate methods)

```
class Test
{ void m1()
{ System.out.println("m1 instance method");
}
void m1()
{ System.out.println("m1 instance method");
}
public static void main(String[] args)
{ Test t = new Test();
t.m1();
}
}
```

Example-6:- For java methods return type is mandatory otherwise the compilation will generate error message **"invalid method declaration; return type required ".**

```
class Test
{ m1()
{ System.out.println("m1 instance method");
}
public static void main(String[] args)
{ Test t = new Test();
t.m1();
}
}
```

Example-7 :-

- Declaring the class inside another class is called inner classes, java supports inner classes.
- Declaring the methods inside other methods is called inner methods but java not supporting inner methods concept if we are trying to declare inner methods compiler generate error message **"illegal start of expression".**

```
class Test
{ void m1()
{ void m2() //inner method
{ System.out.println("m2() inner method");
}
System.out.println("m1() outer method");
}
public static void main(String[] args)
{ Test t1=new Test();
t.m1();
}
}
```

Example 8:-operator overloading

- One operator with more than one behavior is called operator over loading.
- Java is not supporting operator overloading concept but only one implicit overloaded operator in java is + operator.
- If two operands are integers then **plus (+)** perform addition.
- If at least one operand is String then plus (+) perform concatenation.

```
class Test
{ public static void main(String[] args)
{ System.out.println(10+20);
System.out.println("A"+"B"+2+2+"kids");
int a=10,b=20,c=30;
System.out.println(a);
System.out.println(a+"---");
System.out.println(a+"---"+b);
System.out.println(a+"---"+b+"----");
System.out.println(a+"---"+b+"----"+c);
} }
```

Example-9 :- methods vs. All data- types

- By default the numeric values are integer values but to represent other format like byte, short perform typecasting.
- By default the decimal values are double values but to represent float value perform typecasting or use "**f**" constant. (double d=10.5; float f=20.5f;).

```
class Test
{ void m1(byte a) { System.out.println("Byte value-->"+a); }
void m2(short b ) { System.out.println("short value-->"+b); }
void m3(int c) { System.out.println("int value-->"+c); }
void m4(long d) { System.out.println("long value is-->"+d); }
void m5(float e) { System.out.println("float value is-->"+e); }
void m6(double f) { System.out.println("double value is-->"+f); }
void m7(char g) { System.out.println("character value is-->"+g); }
void m8(boolean h) { System.out.println("Boolean value is-->"+h); }
public static void main(String[] args)
{ Test t=new Test();
t.m1((byte)10); t.m2((short)20);
t.m3(30); t.m4(40);
t.m5(10.6f); t.m6(20.5);
t.m7('a'); t.m8(true);
}
}
```

Example-10:-java method calling

- In java one method is calling another method by using method name.
- one java method is able to call more than one method. But once the method is completed the control returns to caller method.

m1() →calling →m2()→ calling → m3()
m1()←after completion m2()←after completion m3()

```
class Test
{ void m1()
{ m2(); //m2() method calling
System.out.println("m1");
m2(); //m2() method calling
}
void m2()
{ m3(100); //m3() method calling
System.out.println("m2 ");
m3(200); //m3() method calling
}
void m3(int a)
{ System.out.println("m3 "); }
public static void main(String[] args)
{ Test t=new Test();
t.m1(); //m1() method calling
} }
```

Example-11 :-

Case 1:- This keyword not required

In below example instance variables and local variables having different names so this keyword not required.

```
class Test
{ //instance variables
int a=100;
int b=200;
void add(int i,int j)
{ System.out.println(a+b); //instance variables addition
System.out.println(i+j); //local variables addition
}
public static void main(String[] args)
{ Test t = new Test();
t.add(10,20);
}
}
```

Case 2:- This keyword required:-

In below example intstance & local variables having same name, then to represent instance variables use **this** keyword.

```
class Test
{ //instance variables
int a=100;
int b=200;
void add(int a,int b)
{ System.out.println(a+b); //local variables addition
System.out.println(this.a+this.b); //instance variables addition
}
public static void main(String[] args)
{ Test t = new Test();
t.add(10,20);
}
}
```

Example-12 :-

- In java **this** keyword is instance variable hence it is not possible to use inside static area. If we are using **this** variable inside static context then compiler will generate error message "**non-static variable** this **cannot be referenced from a static context**".
- In the static context it is not possible to use **this & super** keywords.

```
class Test
{ int a=100;
static void add(int a)
{ System.out.println(this.a);
}
public static void main(String[] args)
{ Test t = new Test();
t.add(10);
}
}
```

Compilation error:- non-static variable this **cannot be referenced from a static context.**

Example 13:- Inside the static area if you want access static variables use object name.

```
class Test
{ int a=100;
static void add(int a)
{ Test t = new Test();
System.out.println(t.a);
}
public static void main(String[] args)
{ Test t = new Test();
t.add(10);
}
}
```

Example-14:- Conversion of local variables to instance variables to improve the scope of the variable.

```
class Test
{ //instance variables
int val1;
int val2;
void values(int val1,int val2)//local variables
{ System.out.println(val1);
System.out.println(val2);
//conversion of local to instance (passing local variables values to instance variables)
this.val1=val1;
this.val2=val2;
}
void add()
{ System.out.println(val1+val2);
}
void mul()
{ System.out.println(val1*val2);
}
public static void main(String[] args)
{ Test t = new Test();
t.values(10,20);
t.add();
t.mul();
}
}
```

Observation :- if the instance variable names and static variable names are different we can use directly without this keyword.

```
//instance variables
int a;
int b;
void values(int val1,int val2)//local variables
{ System.out.println(val1);
System.out.println(val2);
//conversion of local to instance (passing local variables values to instance variables)
a=val1;
b=val2;
}
```

Example-15 :- methods vs return type.

- For java methods return type is mandatory & void represent return nothing.

- Methods can have any return type like
  - primitive type such as byte,short,int,long,float….etc
  - Arrays type
  - Class type
  - Interface type
  - Enum type.

- If the method is having return type other than void then must return the value by using **return** keyword otherwise compiler will generate error message "**missing return statement**"

Below syntax invalid because method must return int value by using return statement.

```
int m1()
{ System.out.println("Hello");
}
```

The below example is valid because it is returning int value by using return statement.

```
int m1()
{ System.out.println("Hello");
return 100;
}
```

- Inside the method we are able to declare only one return statement that statement must be last statement of the method otherwise compiler will generate error message "**unreachable statement**".

The below example is invalid because return statement is must be last statement.

```
int m1()
{ return 100;
System.out.println("Hello");
}
```

The below example valid because return statement is last statement.

```
int m1()
{ System.out.println("Hello");
return 100;
}
```

- Every method is able to returns the value but holding (storing) that return value is optional, but it is recommended to hold the return value check the status o the method.

```
class Test
{ int m1(int a,char ch)
{ System.out.println("***m1 method***");
System.out.println(a+"---"+ch);
return 100;
}
boolean m2(String str1,String str2)
{ System.out.println("****m2 method****");
System.out.println(str1+"---"+str2);
return true;
}
static String m3(double d,boolean b)
{ System.out.println("****m3 method****");
System.out.println(d+"---"+b);
return "ratan";
}
public static void main(String[] args)
{ Test t=new Test();
int x = t.m1(10,'a');
System.out.println("m1() return value-->"+x);
boolean b = t.m2("ratan","anu");
System.out.println("m2() return value-->"+b);
String str = Test.m3(10.5,true);
System.out.println("m3() return value-->"+str);
}
}
```

Example 16 :- It is possible to print return value of the method in two ways,

1. Hold the value print that value.
2. Directly print call method by using System.out.println()

If the method is having return type is void but if we are trying to call method by using System.out.println() then compiler will generate error message.

```
class Test
{ int m1()
{ System.out.println("m1 method");
return 10; }
void m2()
{ System.out.println("m2 method"); }
public static void main(String[] args)
{ Test t =new Test();
int x = t.m1();
System.out.println("return value="+x); //1-way
System.out.println("return value="+t.m1()); //2-way
t.m2(); //valid
//System.out.println(t.m2()); Compilation error:'void' type not allowed here
}
}
```

Example 17 :- Java.util.Scanner

- Scanner class present in **java.util** package and it is introduced in 1.5 versions & it is used to take dynamic input from the keyboard.
- To communicate with system resources use System class & to take input from keyboard use **in** variable(in=input).

Scanner s = new Scanner(System.in); //Scanner object creation
to get int value → s.nextInt()
to get float value →s.nextFloat()
to get byte value → s.nextbyte()
to get String value → s.next()
to get single line → s.nextLine()
to close the input stream → s.close()

Example 18:- The \s represents whitespace.

```
import java.util.*;
public class Test
{ public static void main(String args[])
{ String input = "7 Apple 12 Orange";
Scanner s = new Scanner(input).useDelimiter("\\s");
System.out.println(s.nextInt());
System.out.println(s.next());
System.out.println(s.nextInt());
System.out.println(s.next());
s.close(); }}
```

Example 19:- retrun statement vs if-else

```
import java.util.*;
class Test
{ static String status(int age)
{ if (age>20)
{ return "eligible for marriage"; }
else
{ return "not eligible for marriage"; } }
public static void main(String[] args)
{ Scanner s = new Scanner(System.in);
System.out.println("enter your age:");
int age = s.nextInt();
String str = Test.status(age);
System.out.println("your status is="+str);
}
};
```

Example-20:- methods vs. return variables

Returns local variable as a return value

```
class Test
{ int a=10;
int m1(int a)
{
System.out.println("m1() method");
return a; //return local variable
}
public static void main(String[] args)
{ Test t = new Test();
int x = t.m1(100);
System.out.println(x);
}
}
```

D:\>java Test
m1() method
100

Returns instance variable as a return value(no local variable)

```
class Test
{ int a=10;
int m1()
{
System.out.println("m1() method");
return a; //returns instance value
}
public static void main(String[] args)
{ Test t = new Test();
int x = t.m1();
System.out.println(x);
}
}
```

D:\>java Test
m1() method
10

If the application contains both local & instance variables with same name then first priority goes to local variables but to return instance value use this keyword.

```
class Test
{ int a=10;
int m1(int a)
{ System.out.println("m1() method");
return this.a; //return instance variable as a return value.
}
public static void main(String[] args)
{ Test t = new Test();
int x = t.m1(100);
System.out.println("m1() return value is  "+x); //printing return value
}
}
```

Example 21:- The java class is able to return user defined class as a return value.

```
class X{};
class Emp{};
class Test
{ X m1()
{ System.out.println("m1 method");
X x = new X();
return x;
}
Emp m2()
{ System.out.println("m2 method");
Emp e = new Emp();
return e;
}
static String m3()
{ System.out.println("m3 method");
return "ratan";
}
public static void main(String[] args)
{ Test t = new Test();
X x1 = t.m1();
Emp e = t.m2();
String str = Test.m3();
}
}
```

Note: **when we print object reference variable it always print hash code of the object (we will discuss later).**

Example 22: this keyword representing current class objects.

Java method is able to return current class object in two ways.

1) Creating object & return reference variable.

2) Return **this** keyword.

In above two approaches 2nd approach is recommended to return the current class object.

```
class Test
{ Test m1()
{ System.out.println("m1 method");
Test t = new Test();
return t;
}
Test m2()
{ System.out.println("m2 method");
return this;
}
public static void main(String[] args)
{ Test t = new Test();
Test t1 = t.m1();
Test t2 = t.m2();
}
}
```

Example 23:- Template method

- Let's assume to complete your task you must call four methods then,
  - You must remember number of method names.
  - Order of calling (which one is first & which one is later).
- To overcome above limitation take **x( )** method it is calling four methods internally to complete our task then instead of calling four methods every time call x( ) method that perform our task that x( ) method is called template method.

```
class Test
{ void customer() { System.out.println("custome"); }
void product() { System.out.println("product "); }
void selection() { System.out.println("selection"); }
void billing() { System.out.println("billing"); }
void deliveryManager() //template method
{ System.out.println("****Template method***");
//template method is calling four methods in order to complete our task.
customer(); product(); selection(); billing();
}
public static void main(String[] args)
{ Test t = new Test(); //normal approach
t.customer(); t.product();
t.selection(); t.billing();
//by using template method
t.deliveryManager(); //this method is calling four methods to complete our task.
}
}
```

Example 24:- Method recursion A method is calling itself during execution is called recursion.

case 1:- (normal output)

```
class RecursiveMethod
{ static void recursive(int a)
{ System.out.println("number is :- "+a);
if (a==0)
{return;
}
recursive(--a); //same method is calling [recursion]
}
public static void main(String[] args)
{ RecursiveMethod.recursive(10);
}
}
```

case 2:- (StackOverFlowError)

```
class RecursiveMethod
{ static void recursive(int a)
{ System.out.println("number is :- "+a);
if (a==0)
{return;
}
recursive(++a)
}
public static void main(String[] args)
{ RecursiveMethod.recursive(10);
}
}
```

Example 25 :- Stack Mechanism:-

- In java program execution starts from main method called by JVM & just before calling main method JVM will creates one empty stack memory for that application.
- When JVM calls particular method then that method entry and local variables of that method stored in stack memory & when the method completed, that particular method entry and local variables are destroyed from stack memory & that memory becomes available to other methods.
- The jvm will create stack memory just before calling main method & jvm will destroyed stack memory after completion of main method.

```
class Test
{ void add(int a,int b)
{ System.out.println(a+b); }
void mul(int a,int b)
{ System.out.println(a+b); }
public static void main(String[] args)
{ Test t = new Test();
t.add(5,8); t.mul(10,20);
}
}
```

CONSTRUCTORS

Class Vs Object:-
Class is a logical entity it contains logics whereas object is physical entity it is representing memory.
Class is blue print it decides object creation without class we are unable to create object.
Based on single class (blue print) it is possible to create multiple objects but every object occupies memory.
We are declaring the class by using class keyword but we are creating object by using new keyword.

- We are able to create object in different ways like
  - o By using new operator
  - o By using clone() method
  - o By using new Instance()
  - o By using instance factory method.
  - o By using static factory method
  - o By using pattern factory method
  - o By using deserialization….etc

But we are able to declare the class by using class keyword.

Object creation syntax:-
Class-name reference-variable = new class-name ();
Test t = new Test ();
Test ---> class Name
t ---> Reference variables
= ---> assignment operator
new ---> keyword used to create object
Test () ---> constructor
; ---> statement terminator

When we create new instance (Object) of a class using new keyword, a constructor for that class is called.
New :-

- new keyword is used to create object in java.
- When we create object by using new operator after new keyword that part is constructor then constructor execution will be done.

Rules to declare constructor:-
1) Constructor name class name must be same.
2) It is possible to provide parameters to constructors (just like methods).
3) Constructor not allowed explicit return type. (return type declaration not possible even **void**).

There are two types of constructors,
1) Default Constructor (provided by compiler).
2) User defined Constructor (provided by user) or **parameterized constructor.**

Default Constructor:-

- Inside the class if we are not declaring at least one constructor then compiler generates zero argument constructors with empty implementation at the time of compilation.
- The compiler generated constructor is called **default constructor.**
- Inside the class default constructor is invisible mode.
- To check the default constructor provided by compiler open the .class file code by using java decompiler software.

Application before compilation :-

```
class Test
{ void m1()
{ System.out.println("m1 method");
}
public static void main(String[] args)
{ Test t = new Test();
t.m1();
}
}
```

In above application when we create object by using new keyword "**Test t = new Test ()**" then compiler is searching for **"Test()"** constructor inside the class since not available hence compiler generate default constructor at the time of compilation.
The default constructor is always 0-argument constructor with empty implementations.

Application after compilation :-

```
class Test
{ void m1()
{ System.out.println("m1 method");
}
//default constructor generated by compiler
Test()
{
}
public static void main(String[] args)
{ Test t = new Test();
t.m1();
}
}
```

In above example at run time JVM will execute compiler provide default constructor during object creation.

User defined constructor:-
The constructors which are declared by user are called user defined constructor.
Example :-

```
class Test
{ //constructor declared by user
Test()
{ System.out.println("0-arg constructor");
}
Test(int i)
```

```
{ System.out.println("1-arg constructor"); }
Test(int a,int b)
{ System.out.println("2-arg constructor");
}
public static void main(String[] args)
{ Test t1=new Test();
Test t2=new Test(10);
Test t3=new Test(100,200);
}
}
```

Example:-

- Inside the class if we are declaring at least one constructor (either 0-arg or parameterized) then compiler won't generate default constructor.
- Inside the class if we are not declaring at least one constructor (either 0-arg or parameterized) then compiler will generate default constructor.
- if we are trying to compile below application the compiler will generate error message

"Cannot find symbol " because compiler is unable to generate default constructor.

```
class Test
{ Test(int i)
{ System.out.println("1-arg constructor");
}
Test(int a,int b)
{ System.out.println("2-arg constructor");
}
public static void main(String[] args)
{ Test t1=new Test();
Test t2=new Test(10);
Test t3=new Test(100,200);
}
}
E:\>javac Test.java
Test.java:9: cannot find symbol
Symbol: constructor Test ()
Location: class Test
```

Note :- default constructor is zero argument constructor but all zero argument constructors are not default constructors.

Object creation formats:-

2-formats of object creation.

1) Named object (having reference variable) **Test t = new Test();**

2) Nameless object (without reference variable) **new Test();**

```
class Test
{ void m1()
{ System.out.println("m1 method");
}
public static void main(String[] args)
{ //named object [having reference variable]
Test t = new Test();
t.m1();
//nameless object [without reference variable`]
```

```
new Test().m1();
}
}
```
2- formats of object creation.
1) Eager object creation.
2) Lazy object creation.

```
class Test
void m1(){System.out.println("m1 method");}
public static void main(String[] args)
{ //Eager object creation approach
Test t = new Test();
t.m1();
//lazy object creation approach
Test t1;
;;;;;;;;; //some code here
t1=new Test();
t1.m1();
}
}
```
Advantages of constructors:-
1) Constructors are used to write block of java code that code will be executed during object creation.
2) Constructors are used to initialize instance variables during object creation.

Example :- default constructor execution process.
```
class Employee
{ //instance variables
int eid;
String ename;
double esal;
void display()
{ //printing instance variables values
System.out.println("****Employee details****");
System.out.println("Employee name :-->"+ename);
System.out.println("Employee eid :-->"+eid);
System.out.println("Employee sal :-->"+esal);
}
public static void main(String[] args)
{ Employee e1 = new Employee();
e1.display();
}
}
```

```
D:\>java Employee
****Employee details****
Employee name :-->null
Employee eid :-->0
Employee sal :-->0.0
```

Problems in above example:-

- In above example during object creation time default constructor is executed with empty implementation and initial values of instance variables (default values) printed .
- In above example Emp object is created but default values are printing hence to overcome this limitation use user defined constructor to initialize some values to instance variables.

Example 2:- user defined o-argument constructor execution process.

```
class Employee
{ //instance variables
int eid;
String ename;
double esal;
Employee( ) //user defined 0-argument constructor
{ //assigning values to instance values during object creation
eid=111;
ename="Rajesh";
esal =6000;
}
void display()
{ //printing instance variables values
System.out.println("****Employee details****");
System.out.println("Employee name :-->"+ename);
System.out.println("Employee name :-->"+eid);
System.out.println("Employee name :-->"+esal);
}
public static void main(String[] args)
{ Employee e1 = new Employee();
e1.display();
}
}
```

Compilation & execution process:-

```
D:\>javac Employee.java
D:\>java Employee
****Employee details****
Employee name :-->Rajesh
Employee name :-->111
Employee name :-->6000.0
```

In above example during object creation user provided 0-arg constructor executed used to initialize some values to instance variables.

Problem in above example:-

In above example when we create object it initializing values to instance variables but when we create multiple object for every object same 0-argument constructors is executing it initializing same values for all objects

```
public static void main(String[] args)
{ Emp e1 = new Emp();
e1.display();
```

```
Emp e2 = new Emp();
e2.display();
}
```
D:\>java Employee
******Employee details******
Employee name :-->Rajesh
Employee name :-->111
Employee name :-->6000.0
Employee name :-->Rajesh
Employee name :-->111
Employee name :-->6000.0

To overcome above limitation use parameterized constructor to initialize different values to different objects

Example 3:- User defined parameterized constructors:-

- Inside the class if the default constructor is executed & object is created but initial values of variables only printed.
- To overcome above limitation inside the class we are declaring user defined 0-argument constructor to assign some values to instance variables but when we create multiple objects for every object same constructor is executing and same values are initializing.
- To overcome above limitation declare the parameterized constructor and pass the different values to different objects.

Example :-

```
class Employee
{ //instance variables
int eid;
String ename;
double esal;
Employee(int eid,String ename,double esal) //local variables
{ //conversion (passing local values to instance values)
this.eid = eid;
this.ename = ename;
this.esal = esal;
}
void display()
{ //printing instance variables values
System.out.println("****Employee details****");
System.out.println("Employee name :-->"+ename);
System.out.println("Employee name :-->"+eid);
System.out.println("Employee name :-->"+esal);
}
public static void main(String[] args)
{ // during object creation parameterized constructor executed
Employee e1 = new Employee(111,"Rajesh",6000);
e1.display();
Employee e2 = new Employee(222,"Vikas",7000);
e2.display();
}
}
```

E:\>javac Test.java
E:\>java Employee
******Employee details******
Employee name :-->Rajesh
Employee name :-->111
Employee name :-->6000.0
******Employee details******
Employee name :-->Vikas
Employee name :-->222
Employee name :-->7000.0

Example :- assign values to instance variables [constructor vs. method]

```
class Student
{ //instance variables
int sid;
String sname;
int smarks;
//constructor assigning values to instance variables
Student(int sid,String sname,int smarks)
{ this.sid=sid;
this.sname=sname;
this.smarks=smarks;
}
//method assigning values to instance variables
void assign(int sid,String sname,int smarks)
{ this.sid=sid;
this.sname=sname;
this.smarks=smarks;
}
void disp()
{ System.out.println("****student Details****");
System.out.println("student name = "+sname);
System.out.println("student id = "+sid);
System.out.println("student mrks = "+smarks);
}
public static void main(String[] args)
{ Student s = new Student(111,"Rajesh",100);
s.assign(222,"Ramesh",200);
s.disp();
}
}
```

E:\>java Student
******student Details******
student name = Ramesh
student id = 222
student mrks = 200

Example :- primitive variable vs reference variable

- **a** is variable of primitive type such as int,char,double,boolean…etc
- **t** is reference variable & it is the memory address of object.

```
class Test
{ public static void main(String[] args)
{ //a is primitive variable
int a=10;
System.out.println(a);
//t is reference variable
Test t = new Test ();
System.out.println(t);
}
}
```

Constructor calling:-

To call Current class constructor use this keyword

this(); ----> current class 0-arg constructor calling

this(10); ----> current class 1-arg constructor calling

this(10 , true); ----> current class 2-arg constructor calling

this(10 , "Rajesh" , 'a') ----> current class 3-arg constructor calling

Example-1:-

Call the java methods by uisng method name but to call the current class contructor use this keyword.

```
class Test
{ Test( )
{ this(100); //current class 1-arg constructor calling
System.out.println("0-arg constructor logics");
}
Test(int a)
{ this('g',10); //current class 2-arg constructor calling
System.out.println("1-arg constructor logics");
System.out.println(a);
}
Test(char ch,int a)
{ System.out.println("2-arg constructor logics");
System.out.println(ch+"----"+a);
}
public static void main(String[] args)
{ new Test( );
}
}
```

Example 2:-

Inside the constructor this keyword must be first statement otherwise compiler generate error message "**call to this must be first statement in constructor**".

No compilation error:-(this keyword first statement)

```
Test()
{ this(10); //current class 1-argument constructor calling
System.out.println("0 arg");
}
```

Compilation error:- (this keyword not a first statement)

```
Test()
```

```
{ System.out.println("0 arg");
this(10); //current class 1-argument constructor calling
}
```

Constructor chaining :-

- One constructor is calling same class constructor is called constructor calling.
- We are achieving constructor calling by using this keyword.
- Inside constructor we are able to declare only one this keyword that must be first statement of the constructor.

```
class Test
{ Test()
{ this(10);
System.out.println("0-arg cons");
}
Test(int i)
{ this(10,20);
System.out.println("1-arg cons");
}
Test(int i,int j)
{ System.out.println("2-arg cons");
}
public static void main(String[] args)
{ new Test();
}
}
Example:- Assignment
class Student
{ String sname;
int sid;
static String college="SSCET";
Student(String sname,int sid)
{ this.sname=sname;
this.sid=sid;
}
void display()
{ System.out.println("student id="+sid);
System.out.println("student name="+sname);
System.out.println("student college="+college);
}
public static void main(String[] args)
{
Student s1 = new Student("Rajesh",111);
s1.display();
Student s2 = new Student("Rahul",222);
s2.display();
}
}
```

Instance Blocks:-

 Instance blocks are used to write the logics of projects & these logics are executed during object creation just before constructor execution.

 Instance blocks execution depends on object creation it means if we are creating 10 objects 10 times constructors are executed just before constructors instance blocks are executed.

 Instance block syntax

{ **//logics here** }

Example 1:-

```
class Test
{ //instance block
{ System.out.println("instance block:logics");
}
Test()
{ System.out.println("constructor:logics");
}
public static void main(String[] args)
{ new Test();
}
}
```

Example 2:-

Inside the class it is possible to declare multiple instance blocks but the execution order is top to bottom.

```
class Test
{ { System.out.println("instance block-1:logics");
}
Test()
{ System.out.println("0-arg constructor:logics");
}
{ System.out.println("instance block-2:logics");
}
Test(int a)
{ System.out.println("1-arg constructor:logics");
}
public static void main(String[] args)
{ new Test();
new Test();
new Test(10);
}
}
```

Example 3:-

- Instance block execution depends on object creation but not constructor exaction.
- In below example two constructors are executing but only one object is crating hence only one time instance block is executed.

```
class Test
{ { System.out.println("instance block-1:logics");
}
Test()
{ this(10);
System.out.println("0-arg constructor:logics");
}
Test(int a)
{ System.out.println("1-arg constructor:logics");
}
public static void main(String[] args)
{ new Test();
}
}
```

Example 4:-

Case 1:- When we declare instance block & instance variable the execution order is top to bottom.

In below example instance block is declared first so instance block is executed first.

```
class Test
{ { System.out.println("instance block"); } //instance block
int a=m1(); //instance variables
int m1()
{ System.out.println("m1() method called by variable");
return 100;
}
public static void main(String[] args)
{ new Test();
}
}
```

D:\>java Test
instance block
m1() method called by variable

case 2:- When we declare instance block & instance variable the execution order is top to bottom.

In below example instance variable is declared first so instance block is executed first.

```
class Test
{ int a=m1(); //instance varaibles
int m1()
{ System.out.println("m1() method called by variable")
return 100;
}
{ System.out.println("instance block");
}
public static void main(String[] args)
{ new Test();
} }
```

D:\>java Test
m1() method called by variable
instance block
Example-5 :-

- Instance blocks are used to initialize instance variables during object creation but just before constructor execution.
- In java it is possible to initialize the values in different ways

o By using constructors
o By using instance blocks
o By using methods
o By using setter methods……..…etc

```
class Emp
{ int eid;
//instance block initializing values
{ eid=111;
}
//constructor initializing values
Emp()
{ eid=222;
}
//method initializing values
void assign()
{ eid=333;
}
void disp()
{ System.out.println("****Employee Details****");
System.out.println("emp id="+eid);
}
public static void main(String[] args)
{ Emp e = new Emp();
e.disp();
}
}
```

static block:-

- Static blocks are used to write the logics of project that logics are executed during .class file loading time.
- In java .class file is loaded only one time hence static blocks are executed once per class.
- Static block syntax,

```
static
{ //logics here
}
```

Note : instance blocks execution depends on object creation but static blocks execution depends on .class file loading.
Example-1 :-

```
class Test
{ static
```

```
{ System.out.println("static block");
}
public static void main(String[] args)
{
}
}
```

Example-3 :-static block is initializing variable

Static blocks are used to initialize static variables during .class file loading.

```
class Emp
{ static int eid;
static
{ eid=111;
}
static void assign()
{ eid=333;
}
static void disp()
{ System.out.println(eid);
}
public static void main(String[] args)
{ Emp e = new Emp();
e.assign();
e.disp();
}
}
```

Example-4 :- To execute static blocks inside the class main() method is mandatory or optional.

```
class Test
{ static
{ System.out.println("static block");
}
}
```

- When we execute above code up to 1.5 version the code is compiled & executed it will generate output.

Note : Based on above point up to 1.5 version it is possible to print some statements in output console without using main method inside the class.

- When we execute the above code from 1.6 version onwards the code is not compiled it will generate compilation error.

Note:- based on above point from 1.6 version onwards it is not possible to print some statements in output console without using main method inside the class.

Example-5:-

- When we execute the .class file manually then to execute static blocks inside the class main() method mandatory.
- When we load the .class file into memory programmatically then to execute the static blocks main() method is optional.
- In below example both files must present in same folder (same location).

File 1:- Demo.java

```
class Demo
{ static
{ System.out.println("Demo class static block");
}
void m1()
{ System.out.println("Demo class m1 method");
}
}
```

File -2:- Test.java

```
class Test
{ static
{ System.out.println("Test class static blocks");
}
public static void main(String[] args) throws Exception
{ Class c = Class.forName("Demo");
Demo d = (Demo)c.newInstance();
d.m1();
}
}
```

Difference between methods and constructors:-

Property	*methods*	*constructors*
1)Purpose	*methods are used to write logics but these logics will be executed when we call that method.*	*Constructor is used write logics of the project but the logics will be executed during Object creation.*
2)Variable initialization	*It is initializing variable when We call that method.*	*It is initializing variable during object creation.*
3)Return type	*Return type not allowed Even void.*	*It allows all valid return Types(void,int,Boolean...etc)*
4)Name	*Method name starts with lower Case & every inner word starts With upper case. Ex: charAt(),toUpperCase()....*	*Class name and constructor name must be matched.*
5)types	*a) instance method b)static method*	*a)default constructor b)user defined constructor*
6)inheritance	*methods are inherited*	*constructors are not inherited.*
7)how to call	*To call the methods use method Name.*	*to call the constructor use this keyword.*
8)able to call how many Methods or constructors	*one method is able to call multiple methods at a time.*	*one constructors able to Call only one constructor at a time.*
9)this	*to call instance method use this Keyword but it is not possible to call static method.*	*To call constructor use this keyword but inside constructor use only one this statement.*
10)Super	*used to call super class methods.*	*Used to call super class constructor*
11)Overloading	*it is possible to overload methods*	*it is possible to overload cons.*
12)compiler generate Default cons or not	*yes*	*does not apply*