Language Fundamentals

Agenda :

1. **Introduction**
2. **Identifiers**
   - **Rules to define java identifiers:**
3. **Reserved words**
   - **Reserved words for data types: (8)**
   - **Reserved words for flow control:(11)**
   - **Keywords for modifiers:(11)**
   - **Keywords for exception handling:(6)**
   - **Class related keywords:(6)**
   - **Object related keywords:(4)**
   - **Void return type keyword**
   - **Unused keywords**
   - **Reserved literals**
   - **Enum**
   - **Conclusions**
4. **Data types**
   - **Integral data types**
     - **Byte**
     - **Short**
     - **Int**
     - **long**
   - **Floating Point Data types**
   - **boolean data type**
   - **Char data type**
   - **Java is pure object oriented programming or not ?**
   - **Summary of java primitive data type**
5. **Literals**
   - **Integral Literals**
   - **Floating Point Literals**
   - **Boolean literals**
   - **Char literals**
   - **String literals**
   - **1.7 Version enhansements with respect to Literals**
     - **Binary Literals**
     - **Usage of _ (underscore)symbol in numeric literals**
6. **Arrays**
   1. **Introduction**
   2. **Array declaration**
      - **Single dimensional array declaration**
      - **Two dimensional array declaration**
      - **Three dimensional array declaration**
   3. **Array construction**

- **Multi dimensional array creation**

4. **Array initialization**
5. **Array declaration, construction, initialization in a single line.**
6. **length Vs length() method**
7. **Anonymous arrays**
8. **Array element assignments**
9. **Array variable assignments**

Types of variables

- **Primitive variables**
- **Reference variables**
- **Instance variables**
- **Static variables**
- **Local variables**
- **Conclusions**

Un initialized arrays

- **Instance level**
- **Static level**
- **Local level**

Var arg method

- **Single Dimensional Array Vs Var-Arg Method**

Main method

- **1.7 Version Enhansements with respect to main()**

Command line arguments

Java coding standards

- **Coding standards for classes**
- **Coding standards for interfaces**
- **Coding standards for methods**
- **Coding standards for variables**
- **Coding standards for constants**
- **Java bean coding standards**
  - **Syntax for setter method**
  - **Syntax for getter method**
- **Coding standards for listeners**
  - **To register a listener**
  - **To unregister a listener**

Various Memory areas present inside JVM

Identifier :

A name in java program is called identifier. It may be class name, method name, variable name and label name.

Example:

1.Class name(Test)
2.method Name(main)
3.Pre defined class name(String)
4.name of array(args)
5.name of variable(x)

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

1 2 3 4 5

Rules to define java identifiers:

Rule 1: The only allowed characters in java identifiers are:

1) a to z
2) A to Z
3) 0 to 9
4) _ (underscore)
5) $ Dollar

Rule 2: If we are using any other character we will get compile time error.

Example:

```
1)      total_number-------valid
2)      Total#------------------invalid
```

Rule 3: identifiers are not allowed to starts with digit. other than we can use remaining all small,big letters and Special characters(Dollar&underscore)

Example:

```
1)      ABC123---------valid
2)      123ABC---------invalid
```

Rule 4: java identifiers are case sensitive up course java language itself treated as case sensitive language.

Example:

```
class Test{
int number=10;
int Number=20;
int NUMBER=20; we can differentiate with case.
int NuMbEr=30;
}
```

Rule 5: There is no length limit for java identifiers but it is not recommended to take more than 15 lengths.

Rule 6: We can't use reserved words as identifiers.

Example:

```
 int if=10; --------------invalid
```

Rule 7: All predefined java class names and interface names we use as identifiers.

Example 1:

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

int Integer=13;//It is also valid

//Class is a reserved word. But class String is not a reserved word, It is a class name.

Example 2:

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

```
int Runnable=10;
System.out.println(Runnable);
}}
Output:
10
```

Interface is a reserved word, but interfcae name Runnable is not a reserved word. So we can able to use it happily

Even though it is legal to use class names and interface names as identifiers but it is not a good programming practice.

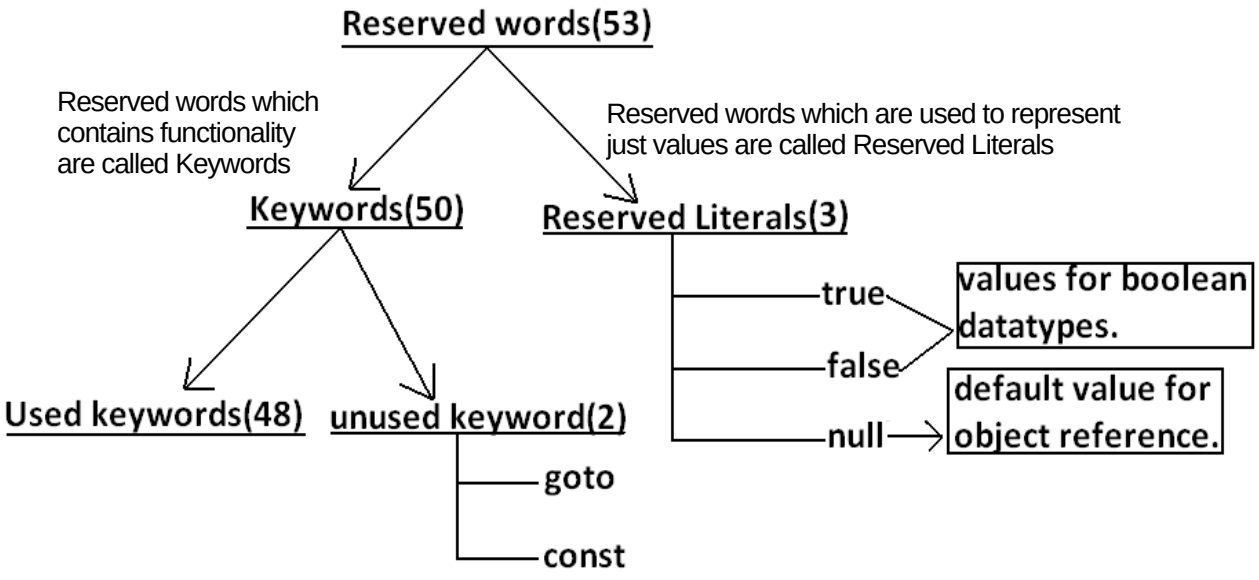
Which of the following are valid java identifiers?

1)_$_(valid)
2)Ca$h(valid)
3)Java2share(valid)
4)all@hands(invalid)
5)123abc (invalid)
6)Total# (invalid)
7)Int (valid)
8)Integer(valid)
9)int (invalid)
10) tot123

Reserved words:

In java some identifiers are reserved to associate some functionality or meaning such type of reserved identifiers are called reserved words.

Diagram: In java some words are reserved to represent meaning and functionality are called Reserved words



Reserved words for data types: (8)

```
1)      byte
2)      short
3)      int
4)      long
5)      float
6)      double
7)      char
8)      boolean
```

Reserved words for flow control:(11)

```
1)      if
2)      else
3)      switch
4)      case
5)      default
6)      for
7)      do
8)      while
9)      break
10)     continue
11)     return
```

Keywords for modifiers:(11)

```
1)      public
2)      private
3)      protected
4)      static
5)      final
6)      abstract
7)      synchronized
8)      native
9)      strictfp(1.2 version)
10)     transient
11)     volatile
```

Keywords for exception handling:(6)

```
1)      try
2)      catch
3)      finally
4)      throw
5)      throws
6)      assert(1.4 version)
```

Class related keywords:(6)

```
1)      class
2)      package
3)      import
4)      extends
5)      implements
6)      interface
```

Object related keywords:(4)

```
1)      new
2)      instanceof
3)      super
4)      this
```

Void return type keyword:

If a method won't return anything compulsory that method should be declared with the void return type in java but it is optional in C++.

```
1)      void
```


Unused keywords:

goto: Create several problems in old languages and hence it is banned in java.
Const: Use final instead of this.
By mistake if we are using these keywords in our program we will get compile time error.

Reserved literals:

```
1)      true    values for boolean data type.
2)      false
3)      null----------------- default value for object reference.
```

Enum:

This keyword introduced in 1.5v to define a group of named constants

```
Example:
enum Beer
{
KF, RC, KO, FO;
}
```


Conclusions :

1. **All reserved words in java contain only lowercase alphabet symbols.**
2. **New keywords in java are:**
3. `strictfp----------1.2v`
4. `assert------------1.4v`
5. `enum-------------1.5v`
6. **In java we have only new keyword but not delete because destruction of useless objects is the responsibility of Garbage Collection.**
7. `instanceof  but not  instanceOf`
8. `strictfp    but not  strictFp`
9. `const     but not  Constant`
10. `syncronized but not  syncronize`
11. `extends     but not  extend`
12. `implements  but not  implement`
13. `import      but not  imports`
14. `int          but not  Int`
15.

Which of the following list contains only java reserved words ?

1. **final, finally, finalize (invalid) //here finalize is a method in Object class.**
2. **throw, throws, thrown(invalid) //thrown is not available in java**
3. **break, continue, return, exit(invalid) //exit is not reserved keyword**
4. **goto, constant(invalid) //here constant is not reserved keyword** const is a reserved word
5. **byte, short, Integer, long(invalid) //here Integer is a wrapper class**
6. **extends, implements, imports(invalid) //imports keyword is not available in java**

import is available not imports

7. finalize, synchronized(invalid) //finalize is a method in Object class
8. instanceof, sizeOf(invalid) //sizeOf is not reserved keyword
9. new, delete(invalid) //delete is not a keyword
10. None of the above(valid)

Which of the following are valid java keywords?

1. public(valid)
2. static(valid)
3. void(valid)
4. main(invalid)
5. String(invalid)
6. args(invalid)
7. 

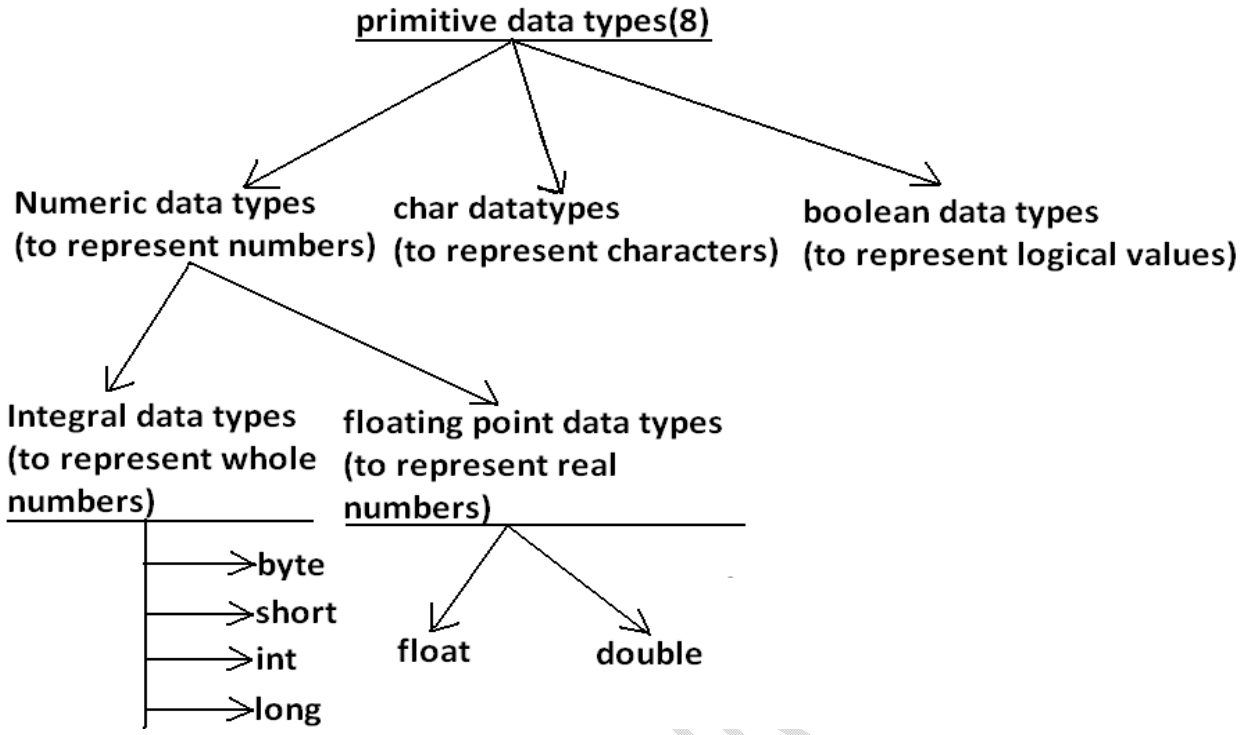
Data types:

Every variable has a type, every expression has a type and all types are strictly define more over every assignment should be checked by the compiler by the type compatibility hence java language is considered as strongly typed programming language.

Java is pure object oriented programming or not?

Java is not considered as pure object oriented programming language because several oops features (like multiple inheritance, operator overloading) are not supported by java moreover we are depending on primitive data types which are non objects.

Diagram:

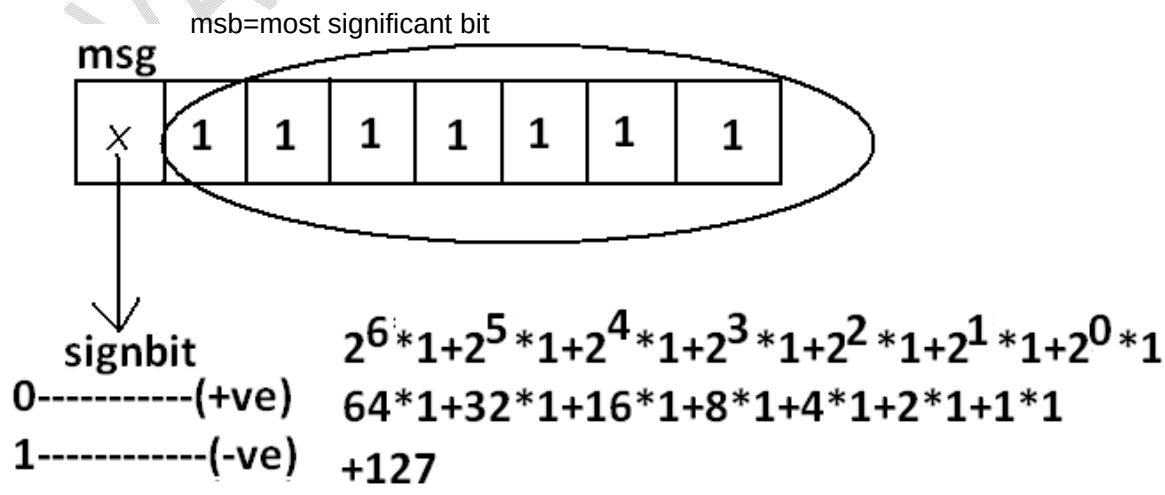


Except Boolean and char all remaining data types are considered as signed data types because we can represent both "+ve" and"-ve" numbers.

Integral data types :

Byte:

```
Size: 1byte (8bits)
Maxvalue: +127
Minvalue:-128
Range:-128to 127[-2^7 to 2^7-1]
```



- **The most significant bit acts as sign bit. "0" means "+ve" number and "1" means "–ve" number.**
- **"+ve" numbers will be represented directly in the memory whereas "–ve" numbers will be represented in 2's complement form.**

```
Example:
byte b=10;
byte b2=130;//C.E:possible loss of precision
                    found : int
                    required : byte
byte b=10.5;//C.E:possible loss of precision found double required byte
byte b=true;//C.E:incompatible types found boolean required byte
byte b="ashok";//C.E:incompatible types
                        found : java.lang.String
                        required : byte
```

byte data type is best suitable if we are handling data in terms of streams either from the file or from the network.

Short:

The most rarely used data type in java is short.

```
Size: 2 bytes
Range: -32768 to 32767(-2^15 to 2^15-1)
```

Example:

```
short s=130;
short s=32768;//C.E:possible loss of precision found int required short
short s=true;//C.E:incompatible types found boolean required short
```

Short data type is best suitable for 16 bit processors like 8086 but these processors are completely outdated and hence the corresponding short data type is also out data type.

Int:

This is most commonly used data type in java.

```
Size: 4 bytes
```

```
Range:-2147483648 to 2147483647 (-2^31 to 2^31-1)
```

Example:

```
int i=130;
int i=10.5;//C.E:possible loss of precision
int i=true;//C.E:incompatible types
```

int x=2147483648 //C.E Integer number too large

Every integral number(without decimal values) are treated as int so error be like this, if we specify long then only it will take it as long.
Ex:Int x=2147483648L; at end(l=long)
//It gives compile time error as possible loss of precision found:long required:int

long:

Whenever int is not enough to hold big values then we should go for long data type.

Example:

To hold the no. Of characters present in a big file int may not enough hence the return type of length() method is long.

```
long l=f.length();//f is a file
Size: 8 bytes
Range:-2^63  to 2^63-1
```

Note: All the above data types (byte, short, int and long) can be used to represent whole numbers. If we want to represent real numbers then we should go for floating point data types.

Floating Point Data types:

Float	double
If we want to 5 to 6 decimal places of accuracy then we should go for float.	If we want to 14 to 15 decimal places of accuracy then we should go for double.
Size:4 bytes.	Size:8 bytes.
Range:-3.4e38 to 3.4e38.	-1.7e308 to1.7e308.
float follows single precision.	double follows double precision.

boolean data type:

```
Size: Not applicable (virtual machine dependent)
Range: Not applicable but allowed values are true or false.
```

Which of the following boolean declarations are valid?

Example 1:

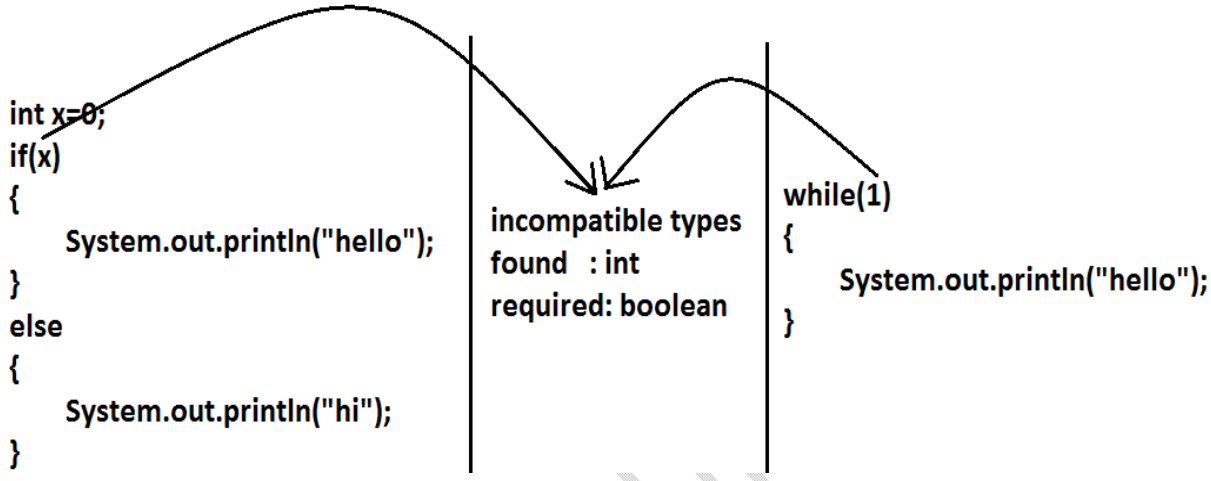
```
boolean b=true;
boolean b=True;//C.E:cannot find symbol
boolean b="True";//C.E:incompatible types
boolean b=0;//C.E:incompatible types
```

Cannot find symbol symbol:variable True

Incompatable types found:String required:boolean

found int required boolean

Example 2:



Char data type:

In old languages like C & C++ are ASCII code based the no.Of ASCII code characters are < 256 to represent these 256 characters 8 - bits enough hence char size in old languages 1 byte.

telugu,hindi and tamil etc alphabets

In java we are allowed to use any worldwide alphabets character and java is Unicode based and no.Of unicode characters are > 256 and <= 65536 to represent all these characters one byte is not enough compulsory we should go for 2 bytes.

```
Size: 2 bytes
Range: 0 to 65535
```

Example:

```
char ch1=97;
char ch2=65536;//C.E:possible loss of precision
```

Summary of java primitive data type:

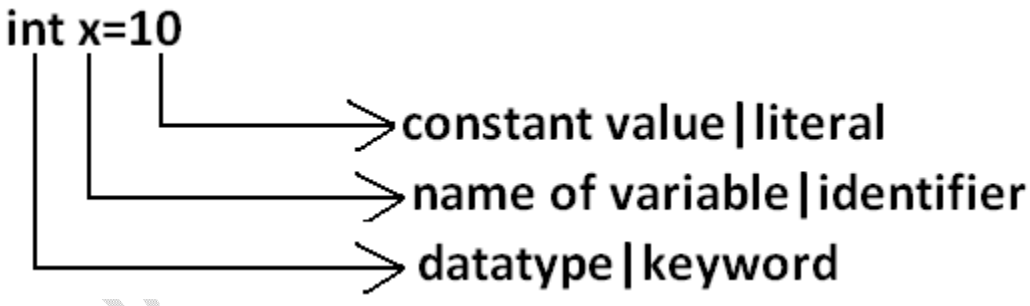
data type	Size	Range	Corresponding Wrapper class	Default value
byte	1 byte	-2^7 to 2^7-1(-128 to 127)	Byte	0
short	2 bytes	-2^{15} to $2^{15}-1$ (-32768 to 32767)	Short	0
int	4 bytes	-2^{31} to $2^{31}-1$ (-2147483648 to 2147483647)	Integer	0
long	8 bytes	-2^{63} to $2^{63}-1$	Long	0
float	4 bytes	-3.4e38 to 3.4e38	Float	0.0
double	8 bytes	-1.7e308 to 1.7e308	Double	0.0
boolean	Not applicable	Not applicable(but allowed values true|false)	Boolean	false
char	2 bytes	0 to 65535	Character	0(represents blank space)

The default value for the object references is "null".

int x=null; C.E Incompatible types found: nulltype required: int

Literals:

Any constant value which can be assigned to the variable is called literal.
Example:



Integral Literals:

For the integral data types (byte, short, int and long) we can specify literal value in the following ways.

1) Decimal literals: Allowed digits are 0 to 9.
Example: int x=10;

2) Octal literals: Allowed digits are 0 to 7. Literal value should be prefixed with zero.
Example: int x=010;

3) Hexa Decimal literals:

- **The allowed digits are 0 to 9, A to Z.** A-F
- **For the extra digits we can use both upper case and lower case characters.**
- **This is one of very few areas where java is not case sensitive.**
- **Literal value should be prefixed with ox(or)oX.**

Example: int x=0x10;

These are the only possible ways to specify integral literal.

Which of the following are valid declarations?

Peefixed 0 so octal(0-7) only allowed

1. **int x=0777; //(valid)**
2. **int x=0786; //C.E:integer number too large: 0786(invalid)**
3. **int x=0xFACE; (valid)**
4. **int x=0xbeef; (valid)**
5. **int x=0xBeer; //C.E:';' expected(invalid) //:int x=0xBeer;** ^// ^
6. **int x=0xabb2cd;(valid)**

Example:

```
int x=10;
int y=010;
int z=0x10;
System.out.println(x+"----"+y+"----"+z); //10----8----16
```

Programmer can able to declare values in differenet forms like octa and hexa decimal but compiler converts them to decimal and then it will gonna to give result.

By default every integral literal is int type but we can specify explicitly as long type by suffixing with small "l" (or) capital "L".

Example:

```
int x=10;(valid)
long l=10L;(valid)
long l=10;(valid)
int x=10l;//C.E:possible loss of precision(invalid)
          found : long
          required : int
```

There is no direct way to specify byte and short literals explicitly. But whenever we are assigning integral literal to the byte variables and its value within the range of byte compiler automatically treats as byte literal. Similarly short literal also.

Example:

```
byte b=127;(valid)
byte b=130;//C.E:possible loss of precision(invalid)
short s=32767;(valid)
short s=32768;//C.E:possible loss of precision(invalid)
```

Floating Point Literals:

Floating point literal is by default double type but we can specify explicitly as float type by suffixing with f or F.

SInce by default float type takes as double, double takes 8 bytes and float take 4 bytes how we can insert 8 bytes value in 4bytes.

Example:

```
float f=123.456;//C.E:possible loss of precision(invalid)
float f=123.456f;(valid)
double d=123.456;(valid)
```

We can specify explicitly floating point literal as double type by suffixing with d or D.

Example:

```
double d=123.456D;
```

We can specify floating point literal only in decimal form and we can't specify in octal and hexadecimal forms.

Example:

```
double d=123.456;(valid)
double d=0123.456;(valid)  //it is treated as decimal value  but not octal
double d=0x123.456;//C.E:malformed floating point literal(invalid)
```

Which of the following floating point declarations are valid?

double d=0786;//C.E:-Intger number too large //Since it is a integral literal not floating by placing 0 it takes it as octal, in octal 8 is not allowed

1. **float f=123.456; //C.E:possible loss of precision(invalid)**
2. **float f=123.456D; //C.E:possible loss of precision(invalid)**
3. **double d=0x123.456; //C.E:malformed floating point literal(invalid)**
4. **double d=0xFace; (valid)** double d=0xFace.0//Invalid since floating point values doesnt start with 0x.
5. **double d=0xBeef; (valid)**

We can assign integral literal directly to the floating point data types and that integral literal can be specified in decimal , octal and Hexa decimal form also.

Example:

```
double d=0xBeef;
System.out.println(d);//48879.0
```
But we can't assign floating point literal directly to the integral types.

Example:
```
int x=10.0;//C.E:possible loss of precision
```
We can specify floating point literal even in exponential form also(significant notation).

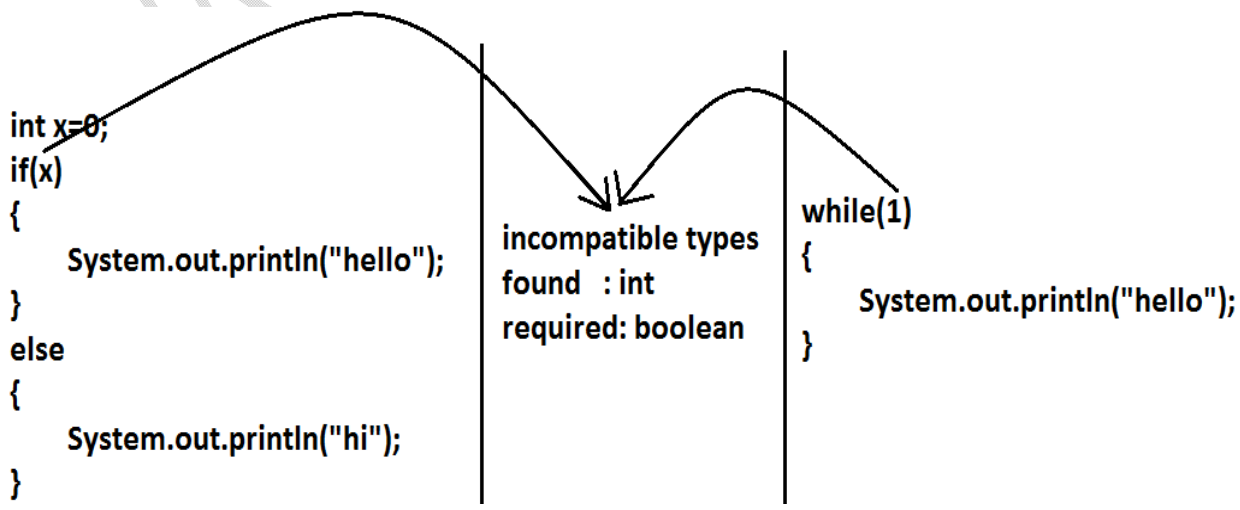
Example:
```
double d=10e2;//==>10*10^2(valid)
System.out.println(d);//1000.0
float f=10e2;//C.E:possible loss of precision(invalid)   found double required float
float f=10e2F;(valid)
```

Boolean literals:

The only allowed values for the boolean type are true (or) false where case is important. i.e., lower case

Example:

1. **boolean b=true;(valid)**
2. **boolean b=0;//C.E:incompatible types(invalid)** found:int required:boolean
3. **boolean b=True;//C.E:cannot find symbol(invalid)** Symbol: variable True
4. **boolean b="true";//C.E:incompatible types(invalid)** found: String required:boolean



Lec 05:-Literals part 2

Char literals:

1) A char literal can be represented as single character within single quotes.

Example:

1. **char ch='a';(valid)**
2. **char ch=a;//C.E:cannot find symbol(invalid)**
3. **char ch="a";//C.E:incompatible types(invalid)**
4. **char ch='ab';//C.E:unclosed character literal(invalid)**

2) We can specify a char literal as integral literal which represents Unicode of that character.
We can specify that integral literal either in decimal or octal or hexadecimal form but allowed values range is 0 to 65535.

Example:

1. **char ch=97; (valid)**
2. **char ch=0xFace; (valid)**
   System.out.println(ch); //?
3. **char ch=65536; //C.E: possible loss of precision(invalid)**

3) We can represent a char literal by Unicode representation which is nothing but '\uxxxx' (4 digit hexa-decimal number) .

Example:

1. **char ch='\ubeef';**
2. **char ch1='\u0061';**
   System.out.println(ch1); //a

3. **char ch2=\u0062; //[illegible] //b**
4. **char ch3='\iface'; //C.E:illegal escape character**
5. **Every escape character in java acts as a char literal.**

Example: '\t' /valid

```
1) char ch='\n';  //(valid)
2) char ch='\l';  //C.E:illegal escape character(invalid)
```

Escape Character	Description
\n	New line
\t	Horizontal tab
\r	Carriage return
\f	Form feed

\b	Back space character
\'	Single quote
\"	Double quote
\\	Back space

Which of the following char declarations are valid?

1. **char ch=a; //C.E:cannot find symbol(invalid)**
2. **char ch='ab'; //C.E:unclosed character literal(invalid)**
3. **char ch=65536; //C.E:possible loss of precision(invalid)**
4. **char ch=\uface; //C.E:illegal character: \64206(invalid)** single quotes missing
5. **char ch='/n'; //C.E:unclosed character literal(invalid)** forward slash there
6. **none of the above. (valid)**

String literals:

Any sequence of characters with in double quotes is treated as String literal.

Example:
String s="Ashok"; (valid)

1.7 Version enhansements with respect to Literals :

The following 2 are enhansements

1. **Binary Literals**
2. **Usage of '_' in Numeric Literals**

Binary Literals :

For the integral data types untill 1.6v we can specified literal value in the following ways

1. **Decimal**
2. **Octal**
3. **Hexa decimal**

But from 1.7v onwards we can specified literal value in binary form also.
The allowed digits are 0 to 1.
Literal value should be prefixed with Ob or OB .

```
int x = 0b111;
System.out.println(x);  // 7
```

Usage of _ symbol in numeric literals :

From 1.7v onwards we can use underscore(_) symbol in numeric literals.

```
double d = 123456.789;  //valid

double d = 1_23_456.7_8_9;  //valid
double d = 123_456.7_8_9;   //valid
```

The main advantage of this approach is readability of the code will be improved At the time of compilation ' _ ' symbols will be removed automatically , hence after compilation the above lines will become double d = 123456.789

We can use more than one underscore symbol also between the digits.
Ex : double d = 1_23_ _456.789;

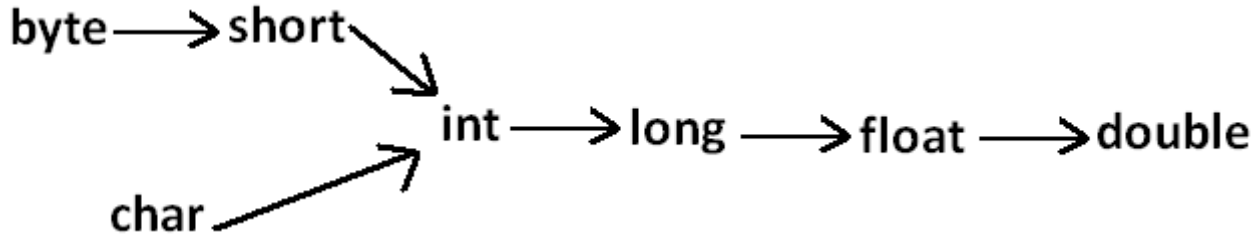
We should use underscore symbol only between the digits

```
double d=_1_23_456.7_8_9;     //invalid  (since underscore at first)
double d=1_23_456.7_8_9_;     //invalid  (since underscore at last)
double d=1_23_456_.7_8_9;     //invalid  (underscore at point)
double d='a';
System.out.println(d); //97
  integral  data types
float f=10L;
System.out.println(f); //10.0
  floating-point  data types
```

8 byte long value, we can be able to assign 4 byte float variable b/c both are following diffrent memory representation internally.Since long is integral type and float is floating point type.

Diagram:

short and char take 2 bytes, but they are not compatable since short is signed data type (in 16 bits first bit for sign) and char is unsigned



Arrays

1) Introduction
2) Array declaration
3) Array construction
4) Array initialization
5) Array declaration, construction, initialization in a single line.
6) length Vs length() method
7) Anonymous arrays
8) Array element assignments
9) Array variable assignments.

Introduction

An array is an indexed collection of fixed number of homogeneous data elements.

The main advantage of arrays is we can represent multiple values with the same name so that readability of the code will be improved.

But the main disadvantage of arrays is:
Fixed in size that is once we created an array there is no chance of increasing or decreasing the size based on our requirement that is to use arrays concept compulsory we should know the size in advance which may not possible always.

We can resolve this problem by using collections.

Array declarations:

Single dimensional array declaration:

Example:
```
int[] a;//recommended to use because name is clearly separated from the
type
int []a;
int a[];
```
At the time of declaration we can't specify the size otherwise we will get compile time error.

Example:
```
int[] a;//valid
int[5] a;//invalid
```

Two dimensional array declaration:

Example:
```
int[][] a;
int [][]a;
int a[][];      All are valid.(6 ways)
int[] []a;
int[] a[];
```

```
int []a[];
```

Three dimensional array declaration:

Example:

```
int[][][] a;
int [][][]a;
int a[][][];
int[] [][]a;
int[] a[][];      All are valid.(10 ways)
int[] []a[];
int[][] []a;
int[][] a[];
int []a[][];
int [][]a[];
```

Which of the following declarations are valid?

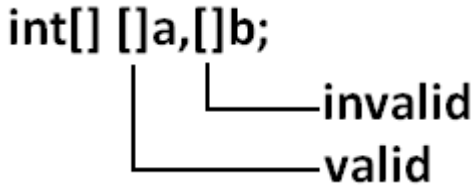
```
1)      int[] a1,b1; //a-1,b-1 (valid)
2)      int[] a2[],b2; //a-2,b-1 (valid)
3)      int[] []a3,b3; //a-2,b-2 (valid)
4)      int[] a,[]b; //C.E: expected (invalid)
```

Note :

If we want to specify the dimension before the variable that rule is applicable only for the 1st variable.

Second variable onwards we can't apply in the same declaration.

Example:



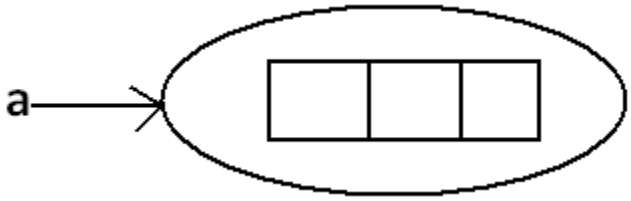
Array construction:

Every array in java is an object hence we can create by using new operator.

Example:

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

Diagram:



For every array type corresponding classes are available but these classes are part of java language and not available to the programmer level.

Array Type	corresponding class name
int[]	[I
int[][]	[[I
double[]	[D

byte[ ] --->[B
short[ ]--->[S
boolean[ ]-->[Z

Rule 1:
At the time of array creation compulsory we should specify the size otherwise we will get compile time error.

Example:

```
int[] a=new int[3];
int[] a=new int[];//C.E:array dimension missing
```

Rule 2:
It is legal to have an array with size zero in java.

Example: Ex: main method(String[] args)
sysout(args.length);

```
int[] a=new int[0];
System.out.println(a.length);//0
```

Rule 3:

If we are taking array size with -ve int value then we will get runtime exception saying NegativeArraySizeException.

Example:

```
int[] a=new int[-3];//R.E:NegativeArraySizeException
```

Rule 4:

The allowed data types to specify array size are byte, short, char, int.
By mistake if we are using any other type we will get compile time error.

Example:

```
int[] a=new int['a'];//(valid)
byte b=10;
int[] a=new int[b];//(valid)
short s=20;
int[] a=new int[s];//(valid)
int[] a=new int[10l];//C.E:possible loss of precision//(invalid)
int[] a=new int[10.5];//C.E:possible loss of precision//(invalid)
```

now size is 97i.e., a=97

byte, short and char acceptable

l=long

found long required int

Rule 5:

The maximum allowed array size in java is maximum value of int size [2147483647].

Example:

```
int[] a1=new int[2147483647];(valid)
int[] a2=new int[2147483648];
  //C.E:integer number too large: 2147483648(invalid)
```

In the first case we may get RE : OutOfMemoryError. (if memory is not available in system)

LEC 07:-Arrays Part 02

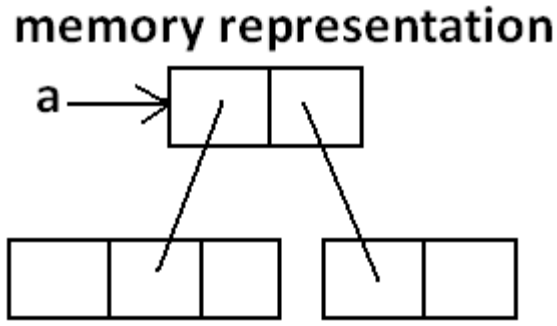
Multi dimensional array creation:

In java multidimensional arrays are implemented as array of arrays approach but not matrix form. B/c to avoid memory wastage, If all cells filled ok, but some are filled.Ex:Students and Supplies
The main advantage of this approach is to improve memory utilization.

Example 1:

```
int[][] a=new int[2][];
a[0]=new int[3];
a[1]=new int[2];
```

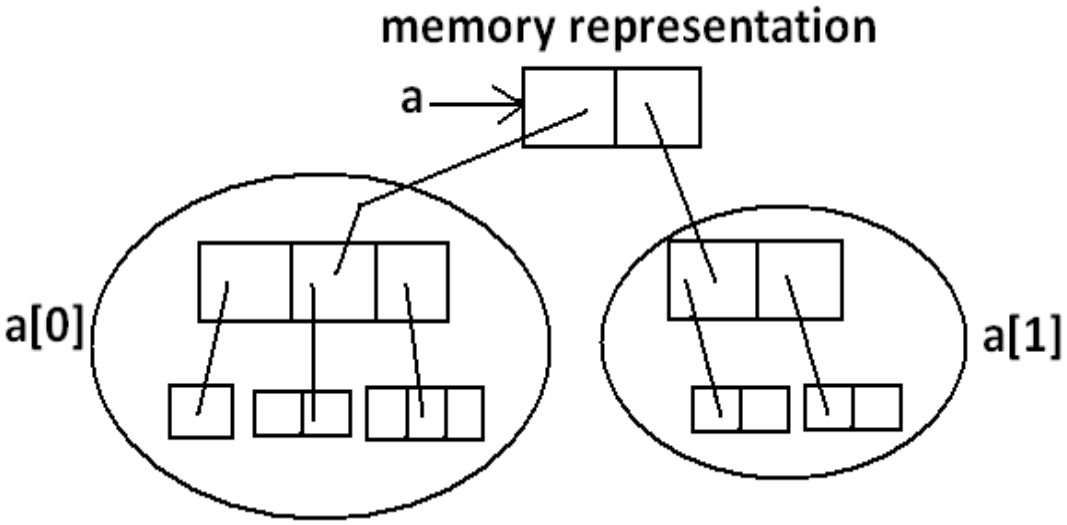
Diagram:



Example 2:

```
int[][][] a=new int[2][][];
a[0]=new int[3][];
a[0][0]=new int[1];
a[0][1]=new int[2];
a[0][2]=new int[3];
a[1]=new int[2][2];
```

Diagram:



Which of the following declarations are valid?

```
1)    int[] a=new int[]//C.E: array dimension missing(invalid)
2)    int[][] a=new int[3][4];(valid)
3)    int[][] a=new int[3][];(valid)
4)    int[][] a=new int[][4];//C.E:']' expected(invalid)
5)    int[][][] a=new int[3][4][5];(valid)
6)    int[][][] a=new int[3][4][];(valid)
7)    int[][][] a=new int[3][][5];//C.E:']' expected(invalid)
```

Array Initialization:

Whenever we are creating an array every element is initialized with default value automatically.

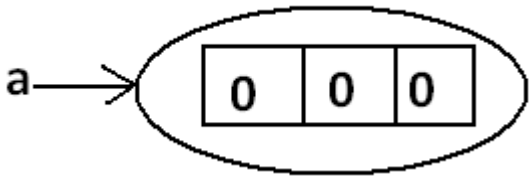
Class Name For two dimensional array class name is [[I

Example 1:

```
int[] a=new int[3];
System.out.println(a);//[I@3e25a5
System.out.println(a[0]);//0
```

hashCode

Diagram:



Note: Whenever we are trying to print any object reference internally toString() method will be executed which is implemented by default to return the following. classname@hexadecimalstringrepresentationofhashcode.

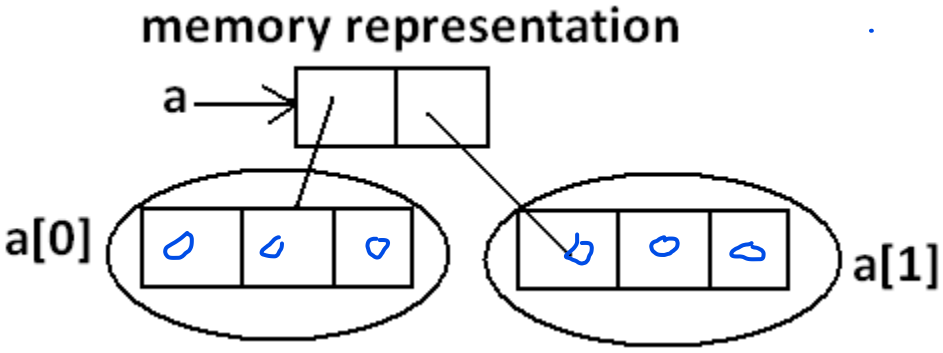
Example 2:

int[][] a=new int[2][3]; base size

```
System.out.println(a);//[[I@3e25a5
System.out.println(a[0]);//[I@19821f
System.out.println(a[0][0]);//0
```

At last stage only values are assigned, remaining all are references only.

Diagram:



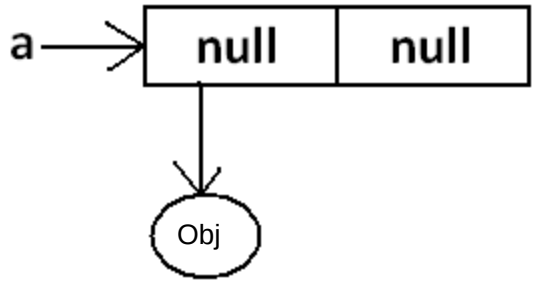
Example 3:

```
int[][] a=new int[2][];
System.out.println(a);//[[I@3e25a5
System.out.println(a[0]);//null
System.out.println(a[0][0]);//R.E:NullPointerException
```

Diagram:

In array if we create array with base size , it will refer to object if we wont specify the second level size, it defaultlt take null. SInce object default value is null.

If u perform any operation on null, we will get null pointer exception



Once we created an array all its elements by default initialized with default values.
If we are not satisfied with those default values then we can replays with our customized values.

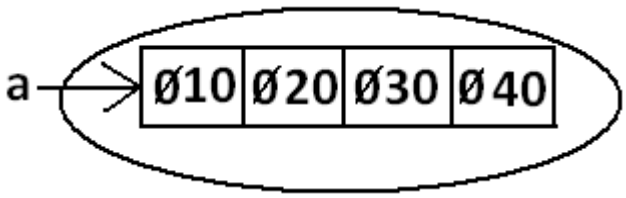
Example:

```
int[] a=new int[4];
a[0]=10;
a[1]=20;
a[2]=30;
a[3]=40;
a[4]=50;//R.E:ArrayIndexOutOfBoundsException: 4
a[-4]=60;//R.E:ArrayIndexOutOfBoundsException: -4
```

R.E-->RunTime Error
C.E-->Compile time error

Diagram:

a[2.5]=90;//C.E possible loss of precision found double required int



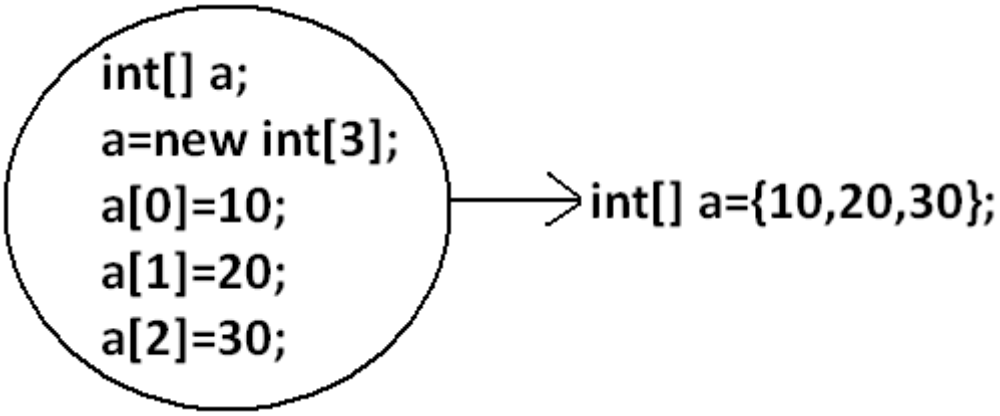
Note: if we are trying to access array element with out of range index we will get Runtime Exception saying ArrayIndexOutOfBoundsException.

LEC 08: Arrays Part-03

Declaration, construction and initialization of an array in a single line:

We can perform declaration, construction and initialization of an array in a single line.

Example:



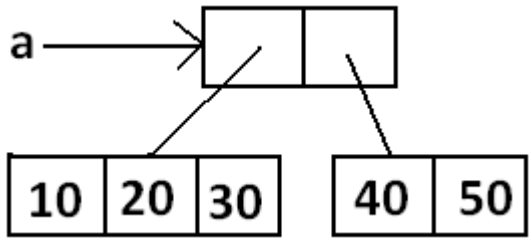

```
char[] ch={'a','e','i','o','u'};(valid)
String[] s={"balayya","venki","nag","chiru"};(valid)
```

We can extend this short cut even for multi dimensional arrays also.

Example:

```
int[][] a={{10,20,30},{40,50}};
```

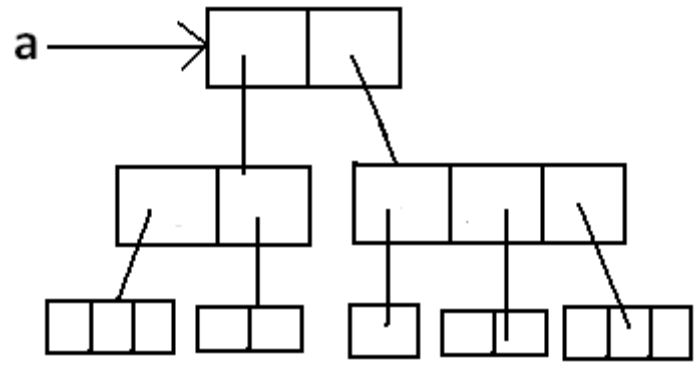
Diagram:



Example:

```
int[][][] a={{{10,20,30},{40,50}},{{60},{70,80},{90,100,110}}};
```

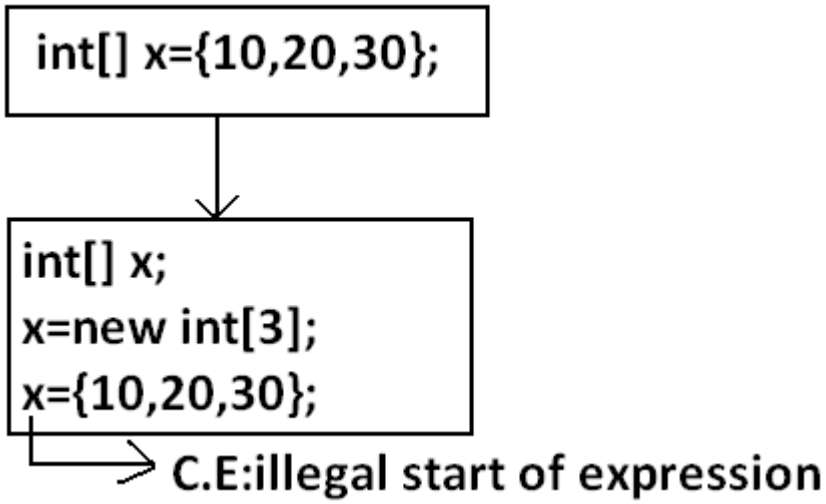
Diagram:




```
int[][][] a={{{10,20,30},{40,50}},{{60},{70,80},{90,100,110}}};
System.out.println(a[0][1][1]);//50(valid)
System.out.println(a[1][0][2]);//R.E:ArrayIndexOutOfBoundsException:
2(invalid)
System.out.println(a[1][2][1]);//100(valid)
System.out.println(a[1][2][2]);//110(valid)
System.out.println(a[2][1][0]);//R.E:ArrayIndexOutOfBoundsException:
2(invalid)
System.out.println(a[1][1][1]);//80(valid)
```

- **If we want to use this short cut compulsory we should perform declaration, construction and initialization in a single line.**
- **If we are trying to divide into multiple lines then we will get compile time error.**

Example:



length Vs length():

length:

Since we can't change the length, once it is fixed

1. **It is the final variable applicable only for arrays.**
2. **It represents the size of the array.**

Example:

```
int[] x=new int[3];
System.out.println(x.length());//C.E: cannot find symbol
System.out.println(x.length);//3
```

length() method:

1. **It is a final method applicable for String objects.**
2. **It returns the no of characters present in the String.**

Example:

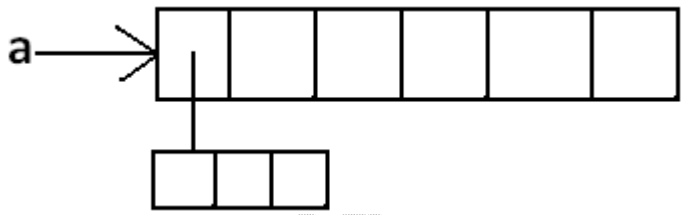
```
String s="bhaskar";
System.out.println(s.length);//C.E:cannot find symbol
System.out.println(s.length());//7
```

In multidimensional arrays length variable represents only base size but not total size.

Example:

```
int[][] a=new int[6][3];
System.out.println(a.length);//6
System.out.println(a[0].length);//3
```

Diagram:



length variable applicable only for arrays where as length()method is applicable for String objects.

There is no direct way to find total size of multi dimentional array but indirectly we can find as follows
x[o].length +x[1].length + x[2].length + ...

Anonymous Arrays:

- **Sometimes we can create an array without name such type of nameless arrays are called anonymous arrays.**
- **The main objective of anonymous arrays is "just for instant use".**
- **We can create anonymous array as follows.**
- `new int[]{10,20,30,40};(valid)`
- `new int[][]{{10,20},{30,40}};(valid)`

- **At the time of anonymous array creation we can't specify the size otherwise we will get compile time error.**

Example: no need to specify the length

```
new int[3]{10,20,30,40};//C.E:';' expected(invalid)
new int[]{10,20,30,40};(valid)
```

Based on our programming requirement we can give the name for anonymous array then it is no longer anonymous.

Example:

```
int[] a=new int[]{10,20,30,40};(valid)
```

Example:

```
class Test
{
        public static void main(String[] args)
        {
         System.out.println(sum(new int[]{10,20,30,40}));//100
        }
        public static int sum(int[] x)
        {
                int total=0;
                for(int x1:x)
                {
                        total=total+x1;
                }
                return total;
        }
}
```

In the above program just to call sum() , we required an array but after completing sum() call we are not using that array any more, ananimous array is best suitable.

Array element assignments:

Case 1:

In the case of primitive array as array element any type is allowed which can be promoted to declared type.

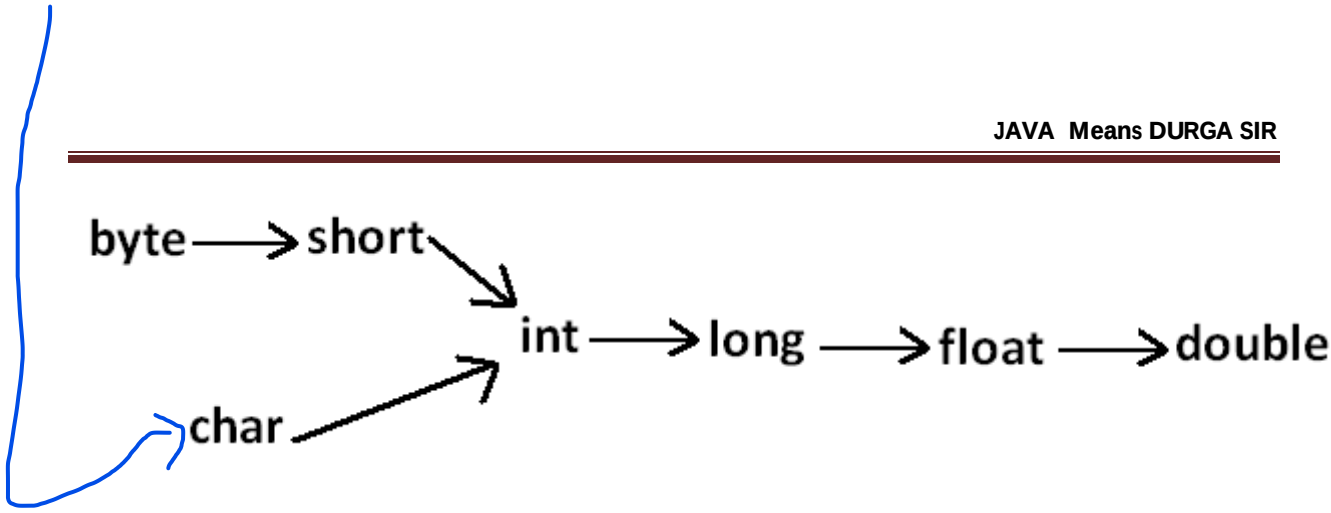
Example 1:

For the int type arrays the allowed array element types are byte, short, char, int.

```
int[] a=new int[10];
a[0]=97;//(valid)
a[1]='a';//(valid)
byte b=10;
a[2]=b;//(valid)
short s=20;
a[3]=s;//(valid)
a[4]=10l;//C.E:possible loss of precision
```

Example 2:

For float type arrays the allowed element types are byte, short, char, int, long, float.



Case 2:

In the case of Object type arrays as array elements we can provide either declared type objects or its child class objects.

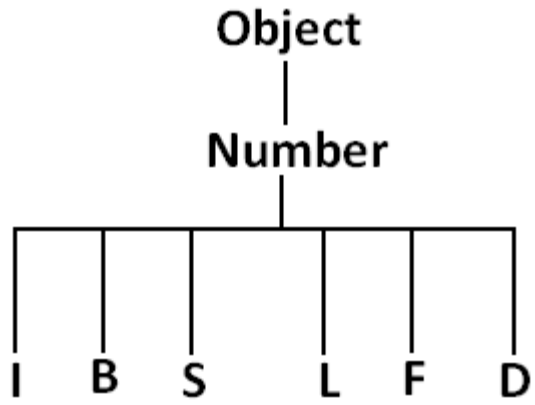
Example 1:

```
Object[] a=new Object[10];
a[0]=new Integer(10);//(valid)
a[1]=new Object();//(valid)
a[2]=new String("bhaskar");//(valid)
```

Example 2:

```
Number[] n=new Number[10];
n[0]=new Integer(10);//(valid)
n[1]=new Double(10.5);//(valid)
n[2]=new String("bhaskar");//C.E:incompatible types//(invalid)
```


Diagram:



Case 3:

In the case of interface type arrays as array elements we can provide its implemented class objects.

Example:

Here we are creating array of type runnable(we are not creating objects for runnable directly), but the elements must be of type runnable for that we insert objects which implements the interface runnable ex: thread

```
Runnable[] r=new Runnable[10];
r[0]=new Thread();
r[1]=new String("bhaskar");//C.E: incompatible types
```

Array Type	Allowed Element Type
1) Primitive arrays.	1) Any type which can be promoted to declared type.
2) Object type arrays.	2) Either declared type or its child class objects allowed.
3) Interface type arrays.	3) Its implemented class objects allowed.
4) Abstract class type arrays.	4) Its child class objects are allowed.

Number is an abstract class

Array variable assignments:

LEC 09: Arrays Part-04

Case 1:

- **Element level promotions are not applicable at array object level.**
- **Ex : A char value can be promoted to int type but char array cannot be promoted to int array.**

Example:

int []a={'a',2,3}; sysout(a[0]);//97

```
int[] a={10,20,30};
char[] ch={'a','b','c'};
int[] b=a;//(valid)
int[] c=ch;//C.E:incompatible types(invalid)
```

Which of the following promotions are valid?

1)char ——— int (valid)
2)char[] ——— int[] (invalid)
3)int ——— long (valid)
4)int[] ——— long[] (invalid)
5)double ——— float (invalid)
6)double[] ——— float[] (invalid)
7)String ——— Object (valid)
8)String[] ——— Object[] (valid)

Note: In the case of object type arrays child type array can be assign to parent type array variable.

Example:

```
String[] s={"A","B"};
Object[] o=s;
```

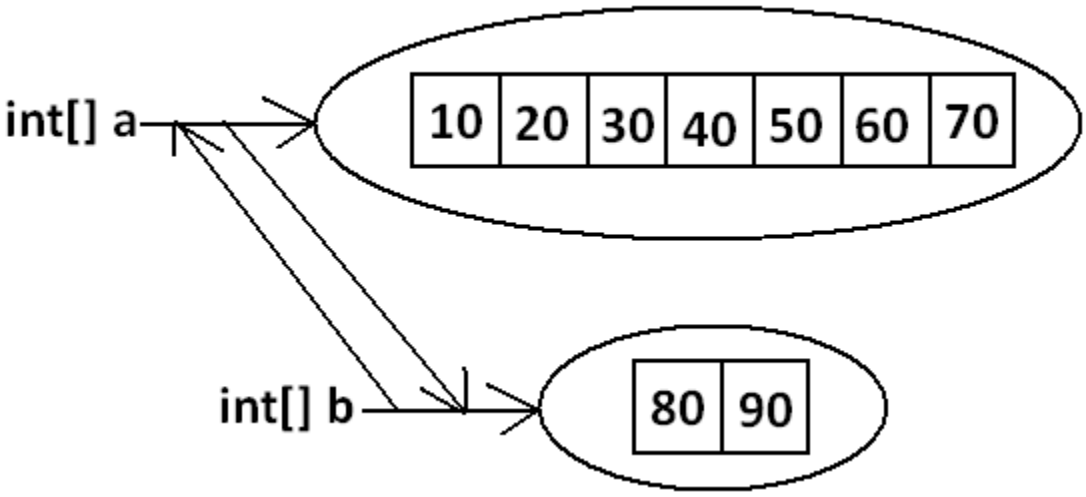
Case 2:

Whenever we are assigning one array to another array internal elements won't be copy just reference variables will be reassigned hence sizes are not important but types must be matched.

Example:

```
int[] a={10,20,30,40,50,60,70};
int[] b={80,90};
a=b;//(valid)
b=a;//(valid)
```

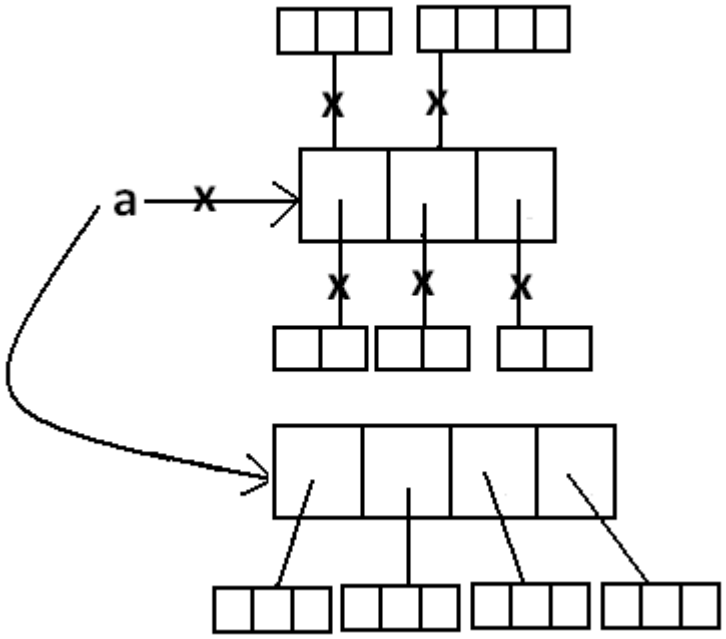
Diagram:



Case 3:
Whenever we are assigning one array to another array dimensions must be matched that is in the place of one dimensional array we should provide the same type only otherwise we will get compile time error.

Example:
```
int[][] a=new int[3][];
a[0]=new int[4][5];//C.E:incompatible types(invalid)
a[0]=10;//C.E:incompatible types(invalid)
a[0]=new int[4];//(valid)
```
Note: Whenever we are performing array assignments the types and dimensions must be matched but sizes are not important.

Example 1:
```
int[][] a=new int[3][2];
a[0]=new int[3];
a[1]=new int[4];
a=new int[4][3];
```
Diagram:



Total how many objects created?
Ans: 11
How many objects eligible for GC: 6

Example 2:

```
class Test
{
        public static void main(String[] args)
        {
                String[] argh={"A","B"};
                args=argh;
                System.out.println(args.length);//2
                for(int i=0;i<=args.length;i++)
                {
                        System.out.println(args[i]);
                }
        }
```

```
}
Output:
java Test x y
R.E: ArrayIndexOutOfBoundsException: 2
java Test x
R.E: ArrayIndexOutOfBoundsException: 2
java Test
R.E: ArrayIndexOutOfBoundsException: 2
```

Note: Replace with i<args.length

Example 3:

```
class Test
{
        public static void main(String[] args)
        {
                String[] argh={"A","B"};
                args=argh;
                System.out.println(args.length);//2
                for(int i=0;i<args.length;i++)
                {
                        System.out.println(args[i]);
                }
        }
}
Output:
2
A
B
```

Example 4:

```
class Test
{
        public static void main(String[] args)
        {
                String[] argh={"A","B"};
                 args=argh;

                 for(String s : args) {
                   System.out.println(s);
        }
}
Output:
A
B
```

Types of Variables

Division 1 : Based on the type of value represented by a variable all variables are divided into 2 types. They are:

1. **Primitive variables**
2. **Reference variables**

Primitive variables:

Primitive variables can be used to represent primitive values.

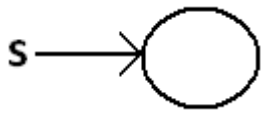
Example: int x=10;

Reference variables:

Reference variables can be used to refer objects.

Example: Student s=new Student();

Diagram:



Division 2 : Based on the behaviour and position of declaration all variables are divided into the following 3 types.

1. **Instance variables**
2. **Static variables**
3. **Local variables**

Instance variables:

- **If the value of a variable is varied from object to object such type of variables are called instance variables.**
- **For every object a separate copy of instance variables will be created.**
- **Instance variables will be created at the time of object creation and destroyed at the time of object destruction hence the scope of instance variables is exactly same as scope of objects.**
- **Instance variables will be stored on the heap as the part of object.**

- **Instance variables should be declared with in the class directly but outside of any method or block or constructor.**
- **Instance variables can be accessed directly from Instance area. But cannot be accessed directly from static area.**
- **But by using object reference we can access instance variables from static area.**

Example:

```
class Test
{
        int i=10;
        public static void main(String[] args)
        {
                //System.out.println(i);
//C.E:non-static variable i cannot be referenced from a static
context(invalid)
                Test t=new Test();
                System.out.println(t.i);//10(valid)
                t.methodOne();
        }
        public void methodOne()
        {
                System.out.println(i);//10(valid)
        }
}
```

since main method is a static method. static is no way realted to object, but instance variables are related to object, so we can't access directly.

It is a instance method,since to call this method it force us to create object, so it is a object related so no problem to access instance variables

For the instance variables it is not required to perform initialization JVM will always provide default values.

Example:

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

Instance variables also known as object level variables or attributes.

Static variables:

- **If the value of a variable is not varied from object to object such type of variables is not recommended to declare as instance variables. We have to declare such type of variables at class level by using static modifier.**
- **In the case of instance variables for every object a separate copy will be created but in the case of static variables for entire class only one copy will be created and shared by every object of that class.**
- **Static variables will be crated at the time of class loading and destroyed at the time of class unloading hence the scope of the static variable is exactly same as the scope of the .class file.**
- **Static variables will be stored in method area. Static variables should be declared with in the class directly but outside of any method or block or constructor.**

- **Static variables can be accessed from both instance and static areas directly.**
- **We can access static variables either by class name or by object reference but usage of class name is recommended.**
- **But within the same class it is not required to use class name we can access directly.**

java TEST

1. **Start JVM.**
2. **Create and start Main Thread by JVM.**
3. **Locate(find) Test.class by main Thread.**
4. **Load Test.class by main Thread. // static variable creation**
5. **Execution of main() method.**
6. **Unload Test.class // static variable destruction**
7. **Terminate main Thread.**
8. **Shutdown JVM.**

Example:

```
class Test
{
```

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

For the static variables it is not required to perform initialization explicitly, JVM will always provide default values.

Example:

```
class Test
{
        static String s;
        public static void main(String[] args)
        {
                System.out.println(s);//null
        }
}
```

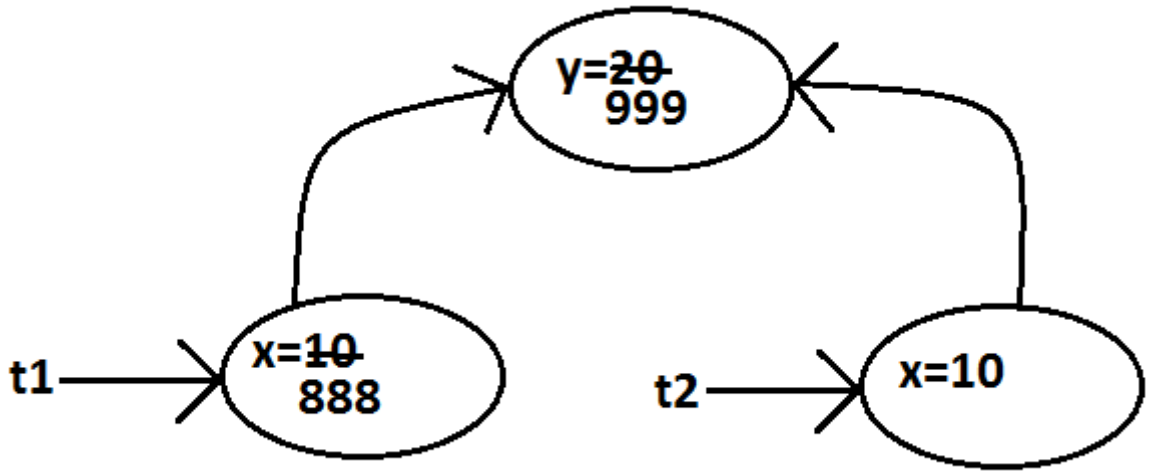
Example:

```
class Test
{
        int x=10;
        static int y=20;
        public static void main(String[] args)
        {
                Test t1=new Test();
                t1.x=888;
                t1.y=999;
                Test t2=new Test();
                System.out.println(t2.x+"----"+t2.y);//10----999
        }
}
```

For static variables only one copy will be created if we modifed by using the one object reference it will modified to all.

For instance variable, for every object one sepatrate copy will be created,if we changed for one object it wont reflect for another objec instance variables

Diagram:



Static variables also known as class level variables or fields.

Local variables:

Some times to meet temporary requirements of the programmer we can declare variables inside a method or block or constructors such type of variables are called local variables or automatic variables or temporary variables or stack variables.

Local variables will be stored inside stack.

The local variables will be created as part of the block execution in which it is declared and destroyed once that block execution completes. Hence the scope of the local variables is exactly same as scope of the block in which we declared.

Example 1:

```
class Test
{
        public static void main(String[] args)
        {
                int i=0;
                for(int j=0;j<3;j++)
                {
                        i=i+j;
                }
```

Here "i" is the local variable for main method, so we can access inside the main method anywhere but "j" is the local variable for for loop we can't able to access outside the the loop.

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

C.E

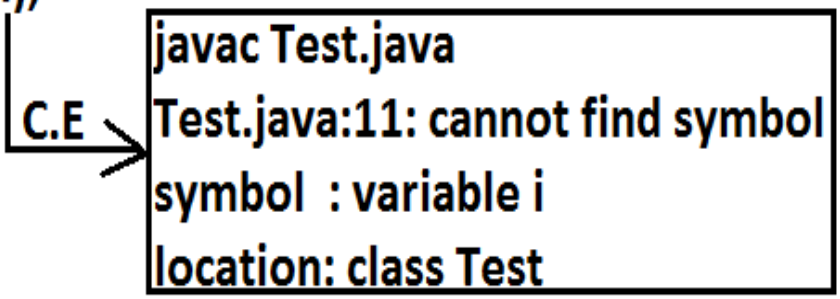
```
javac Test.java
Test.java:10: cannot find symbol
symbol  : variable j
location: class Test
```


```
}
}
```

Example 2:

```
class Test
{
        public static void main(String[] args)
        {
                try
                {
                        int i=Integer.parseInt("ten");
                }
                catch(NullPointerException e)
                {
```

System.out.println(i);




```
                }
        }
}
```

- **The local variables will be stored on the stack.**
- **For the local variables JVM won't provide any default values compulsory we should perform initialization explicitly before using that variable.**

If you are not using the local variables in the program, then it won't give any C.E even though u didn't initialized it.

Example:

```
class Test
{
    public static void main(String[] args)
    {
        int x;
        System.out.println("hello");//hello
    }
}
```

```
class Test
{
    public static void main(String[] args)
    {
        int x;
        System.out.println(x);//C.E:variable x migh
    }
}
```

//C.E variable x might not have been initialized

Example:

```
class Test
{
```

```
        public static void main(String[] args)
        {
                int x;
                if(args.length>0)
                {
                        x=10;
                }
                System.out.println(x);
                   //C.E:variable x might not have been initialized
        }
}
```

The value of x is initailized inside the if loop not in main method, what if , if condition fails it wont enter inside as it is local variable jvm wont provide default value. So it gives compilation error

Example:

```
class Test
{
        public static void main(String[] args)
        {
                int x;
                if(args.length>0)
                {
                        x=10;
                }
                else
                {
                        x=20;
                }
                System.out.println(x);
        }
}
Output:
java Test x
10
java Test x y
10
java Test
20
```

- **It is never recommended to perform initialization for the local variables inside logical blocks because there is no guarantee of executing that block always at runtime.**
- **It is highly recommended to perform initialization for the local variables at the time of declaration at least with default values.**

Note: The only applicable modifier for local variables is final. If we are using any other modifier we will get compile time error.

since scope of the local variable is inisde the method itself. when we go for atleast private, the scope of that is with in the class but this also not satisified by local variables. So access modifiers we can't apply

Example:

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

                public int x=10; //(invalid)
```

```
                private int x=10; //(invalid)
                protected int x=10; //(invalid)    C.E: illegal start of
expression
                static int x=10; //(invalid)
                volatile int x=10; //(invalid)
                transient int x=10; //(invalid)

                final int x=10;//(valid)
        }
}
```

Conclusions:

1. **For the static and instance variables it is not required to perform initialization explicitly JVM will provide default values. But for the local variables JVM won't provide any default values compulsory we should perform initialization explicitly before using that variable.**
2. **For every object a separate copy of instance variable will be created whereas for entire class a single copy of static variable will be created. For every Thread a separate copy of local variable will be created.**
3. **Instance and static variables can be accessed by multiple Threads simultaneously and hence these are not Thread safe but local variables can be accessed by only one Thread at a time and hence local variables are Thread safe.**
4. **If we are not declaring any modifier explicitly then it means default modifier but this rule is applicable only for static and instance variables but not local variable.**

Un Initialized arrays

Example:

```
class Test
{
        int[] a;
        public static void main(String[] args)
        {
                Test t1=new Test();
                System.out.println(t1.a);//null
                System.out.println(t1.a[0]);//R.E:NullPointerException
        }
}
```

Instance level:

Example 1:

```
int[] a;
System.out.println(obj.a);//null
System.out.println(obj.a[0]);//R.E:NullPointerException
```

Example 2:

```
int[] a=new int[3];
System.out.println(obj.a);//[I@3e25a5
System.out.println(obj.a[0]);//0
```

Static level:

Example 1:

```
static int[] a;
System.out.println(a);//null
System.out.println(a[0]);//R.E:NullPointerException
```

Example 2:

```
static int[] a=new int[3];
System.out.println(a);//[I@3e25a5
System.out.println(a[0]);//0
```

Local level:

Example 1:

```
int[] a;
System.out.println(a); //C.E: variable a might not have been initialized
System.out.println(a[0]);
```

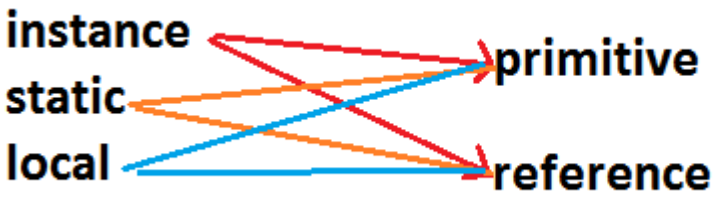
Example 2:

```
int[] a=new int[3];
System.out.println(a);//[I@3e25a5
System.out.println(a[0]);//0
```

Once we created an array every element is always initialized with default values irrespective of whether it is static or instance or local array.

Every variable in java should be either instance or static or local.
Every variable in java should be either primitive or reference

Hence the following are the various possible combinations for variables

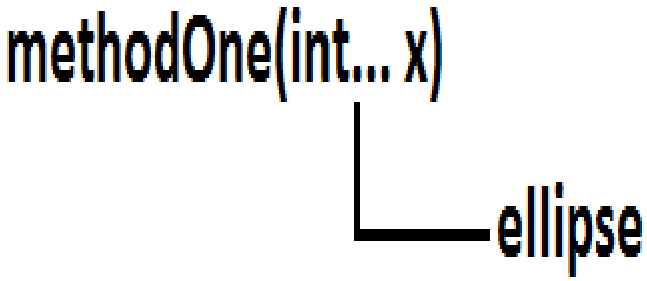



```
class Test {
 int[]  a=new int[3];    // instance-reference
 static  int  x=20;        //static-primitive
 public static void main(String[] args) {
 String s="xyz";        //local-reference
  }
}
```

Var- arg methods (variable no of argument methods) (1.5)

- **Until 1.4v we can't declared a method with variable no. Of arguments.**
- **If there is a change in no of arguments compulsory we have to define a new method.**
- **This approach increases length of the code and reduces readability.**
- **But from 1.5 version onwards we can declare a method with variable no. Of arguments such type of methods are called var-arg methods.**

We can declare a var-arg method as follows.



We can call or invoke this method by passing any no. Of int values including zero number also.

Example:

```
class Test
{
        public static void methodOne(int... x)
        {
                System.out.println("var-arg method");
        }
        public static void main(String[] args)
        {
                methodOne();
                methodOne(10);
                methodOne(10,20,30);
        }
}
Output:
var-arg method
var-arg method
var-arg method
```

Internally var-arg parameter implemented by using single dimensional array hence within the var-arg method we can differentiate arguments by using index.

Example:

```
class Test
{
 public static void sum(int... x)
 {
    int total=0;
    for(int i=0;i<x.length;i++)
    {
```

```
        total=total+x[i];
        }
         System.out.println("The sum :"+total);
}
public static void main(String[] args)
{
 sum();
 sum(10);
 sum(10,20);
 sum(10,20,30,40);
 }
}
Output:
The sum: 0
The sum: 10
The sum: 30
The sum: 100
```

Example:

```
class Test
{
 public static void sum(int... x)
 {
    int total=0;
    for(int x1 : x)
    {
        total=total+x1;
     }
      System.out.println("The sum :"+total);
}
public static void main(String[] args)
{
 sum();
 sum(10);
 sum(10,20);
 sum(10,20,30,40);
 }
}
Output:
The sum: 0
The sum: 10
```

```
The sum: 30
The sum: 100
```

Case 1:
Which of the following var-arg method declarations are valid?

1. **methodOne(int... x) (valid)**
2. **methodOne(int ...x) (valid)**
3. **methodOne(int...x) (valid)**
4. **methodOne(int x...) (invalid)**
5. **methodOne(int. ..x) (invalid)**
6. **methodOne(int .x..) (invalid)**

Case 2:
We can mix var-arg parameter with general parameters also.

Example:

```
methodOne(int a,int... b)        //valid
methodOne(String s,int... x)   //valid
```

Case 3:
if we mix var-arg parameter with general parameter then var-arg parameter should be the last parameter.

Example:

```
methodOne(int a,int... b)        //valid
methodOne(int... a,int b)   //(invalid)
```

Case 4:
With in the var-arg method we can take only one var-arg parameter. i.e., if we are trying to more than one var-arg parameter we will get CE.

Example:

```
methodOne(int... a,int... b)  //(invalid)
```

Case 5:

```
class Test
{
```

```
        public static void methodOne(int i)
        {
                System.out.println("general method");
        }
        public static void methodOne(int... i)
        {
                System.out.println("var-arg method");
        }
        public static void main(String[] args)
        {
                methodOne();//var-arg method
                methodOne(10,20);//var-arg method
                methodOne(10);//general method
        }
}
```

In general var-arg method will get least priority that is if no other method matched then only var-arg method will get the chance this is exactly same as default case inside a switch.

Case 6:
For the var-arg methods we can provide the corresponding type array as argument.

Example:

```
class Test
{
```

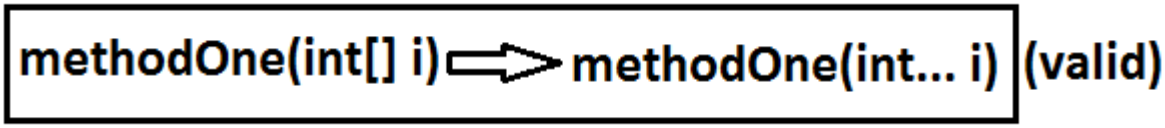

```
{
                System.out.println("var-arg method");
        }
        public static void main(String[] args)
        {
                methodOne(new int[]{10,20,30});//var-arg method
        }
}
```

Case 7:

```
class Test
{
        public void methodOne(int[] i){}
        public void methodOne(int... i){}
}
Output:
Compile time error.
Cannot declare both methodOne(int...) and methodOne(int[]) in Test
```

Single Dimensional Array Vs Var-Arg Method:

Case 1:
Wherever single dimensional array present we can replace with var-arg parameter.



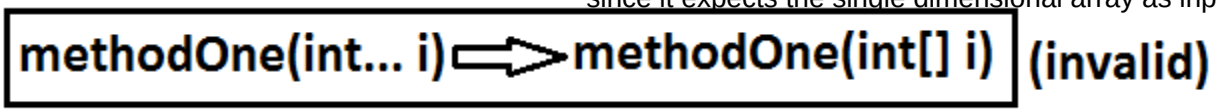
Example:

```
class Test
{
        public static void main(String... args)
        {
                System.out.println("var-arg main method");//var-arg main
method
        }
}
```

Case 2:

Wherever var-arg parameter present we can't replace with single dimensional array.

methodOne(1); In this case var-args works fine, but single D array wont work since it expects the single dimensional array as input



Note :

1. **methodOne(int... x)**
   we can call this method by passing a group of int values and x will become 1D array. (i.e., int[] x)
2. **methodOne(int[]... x)**
   we can call this method by passing a group of 1D int[] and x will become 2D array. (i.e., int[][] x)

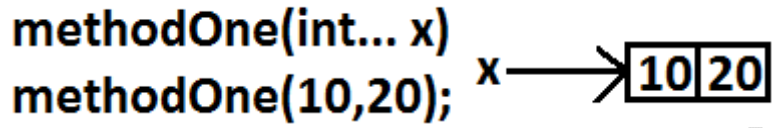
Above reasons this case 2 is invalid.

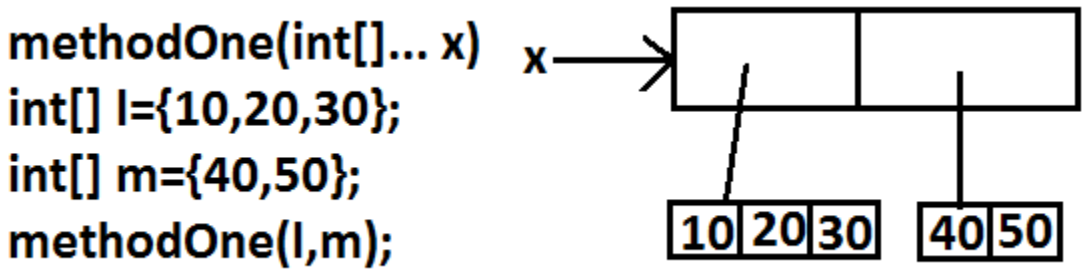
Example:

```
class Test
{
        public static void methodOne(int[]... x)
        {
                for(int[] a:x)
                {
```

```
                        System.out.println(a[0]);
                }
        }
        public static void main(String[] args)
        {
                int[] l={10,20,30};
                int[] m={40,50};
                methodOne(l,m);

        }
}
Output:
10
40
```

Analysis:





Lec 13: - main method part-01

Main Method

Whether the class contains main() method or not,
and whether it is properly declared or not,
these checking's are not responsibilities of the compiler, at runtime JVM is responsible

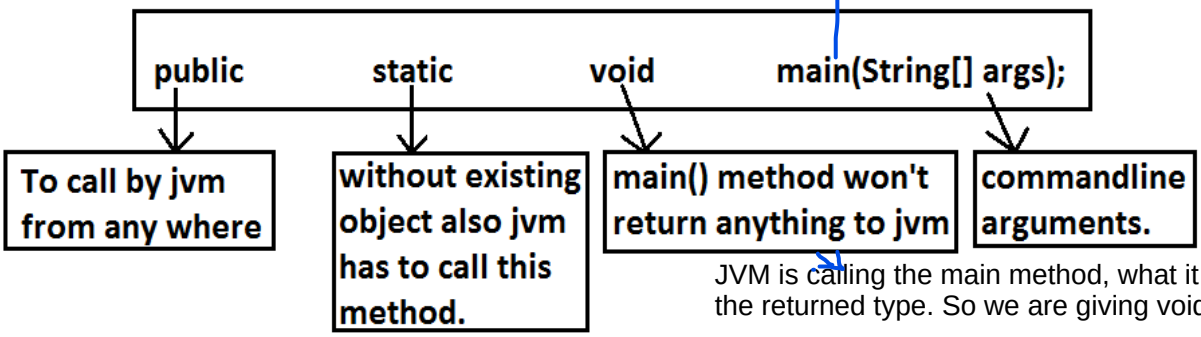
for this.
If JVM unable to find the required main() method then we will get runtime exception saying NoSuchMethodError: main.

Example:

```
class Test
{}
Output:
javac Test.java
java Test R.E: NoSuchMethodError: main
```

This is the name which is configured inside the JVM

At runtime JVM always searches for the main() method with the following prototype.



JVM is calling the main method, what it will do with the returned type. So we are giving void

If we are performing any changes to the above syntax then the code won't run and will get Runtime exception saying NoSuchMethodError.

Even though above syntax is very strict but the following changes are acceptable to main() method.

1. **The order of modifiers is not important that is instead of public static we can take static public.**
2. **We can declare string[] in any acceptable form**
   - **String[] args**
   - **String []args**
   - **String args[]**
3. **Instead of args we can use any valid java identifier.**
4. **We can replace string[] with var-arg parameter.**
   Example: main(String... args)
5. **main() method can be declared with the following modifiers.**
   final, synchronized, strictfp.

Instead of args we can write whatever like purush

```
6. class Test {
7.  static final syncronized strictfp public void  main(String... ask){
8.   System.out.println("valid  main method");
9.  }
10. }
11. output :
12. valid main method
```

Which of the following main() method declarations are valid ?

1. **public static void main(String args){} (invalid)**

2. **public synchronized final strictfp void main(String[] args){} (invalid)**
3. **public static void Main(String... args){} (invalid)**
4. **public static int main(String[] args){} //int return type we can't take //(invalid)**
5. **public static synchronized final strictfp void main(String... args){}(valid)**
6. **public static void main(String... args){}(valid)**
7. **public void main(String[] args){}(invalid)**

In which of the above cases we will get compile time error ?

No case, in all the cases we will get runtime exception.

Case 1 :
Overloading of the main() method is possible but JVM always calls string[] argument main() method only.

Example:


```
class Test
{
        public static void main(String[] args)
        {
        System.out.println("String[] array main method");      //overloaded
methods
        }
        public static void main(int[] args)
        {
        System.out.println("int[] array main method");
        }
}
Output:
String[] array main method
```

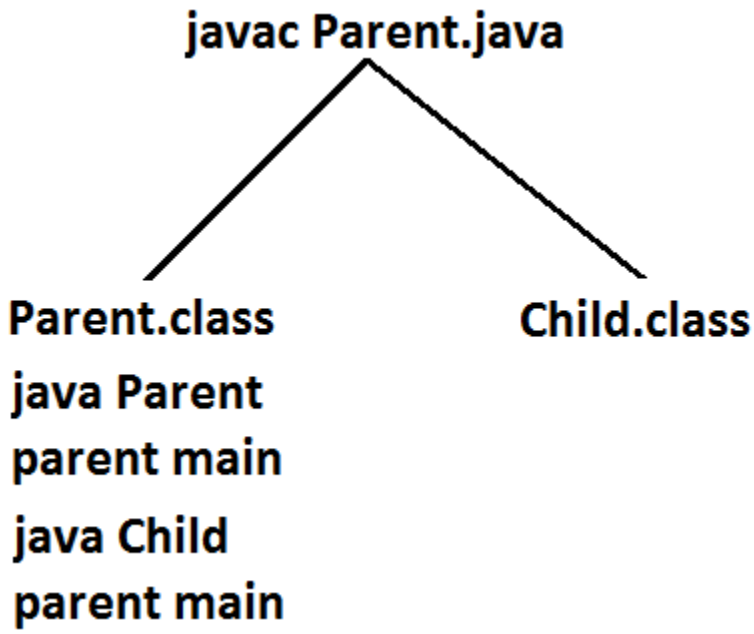
The other overloaded method we have to call explicitly then only it will be executed.

Case 2:
Inheritance concept is applicable for static methods including main() method hence while executing child class if the child class doesn't contain main() method then the parent class main() method will be executed.

Example 1:

```
class Parent
{
        public static void main(String[] args)
        {
                System.out.println("parent main");          //Parent.java
        }
}
class Child extends Parent
{}
```

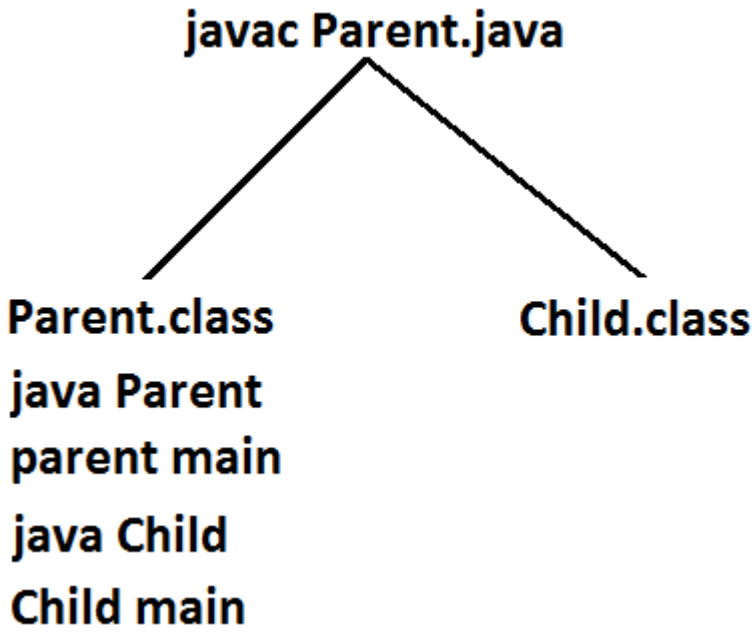

Analysis:



Example 2:

```
class Parent
{
        public static void main(String[] args)
        {
                System.out.println("parent main");          // Parent.java
        }
}
class Child extends Parent
{
        public static void main(String[] args)
        {
                System.out.println("Child main");
        }
}
```

Analysis:



It seems to be overriding concept is applicable for static methods but it is not overriding it is method hiding.

Lec 14:-main method part -2

1.7 Version Enhansements with respect to main() :

Case 1 :

- **Untill 1.6v if our class doesn't contain main() method then at runtime we will get Runtime Exception saying NosuchMethodError:main**
- **But from 1.7 version onwards instead of NoSuchMethodError we will get more meaning full description**

```
class Test {
}
```

```
 1.6 version :
javac Test.java
java Test
RE: NoSuchMethodError:main

1.7 version :
javac Test.java
java Test
Error: main method not found in class Test, please  define the main  method
as
public static void main(String[]  args)
```

Case 2 :

From 1.7 version onwards to start program execution compulsory main method should be required, hence even though the class contains static block if main method not available then won't be executed

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

```
 1.6 version :
javac Test.java
java Test
output :
static block
RE: NoSuchMethodError:main

1.7 version :
javac Test.java
java Test
Error: main method not found in class Test, please  define the main  method
as
public static void main(String[]  args)
```

after executing the static block, it will gives error

Immediatley it will give error

Case 3 :

```
class Test {
 static {
  System.out.println("static block");
  System.exit(0);
 }
```

```
}
```

1.6 version :

```
javac Test.java
java Test
output :
static block
```

1.7 version :

```
javac Test.java
java Test
Error: main method not found in class Test, please  define the main  method
as
public static void main(String[]  args)
```

Case 4 :

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

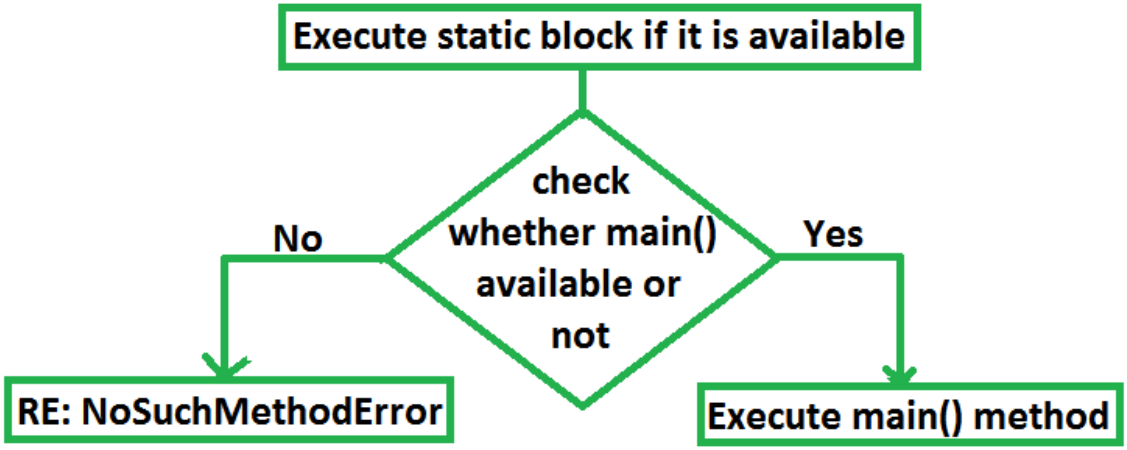
1.6 version :

```
javac Test.java
java Test
output :
static block
main method
```

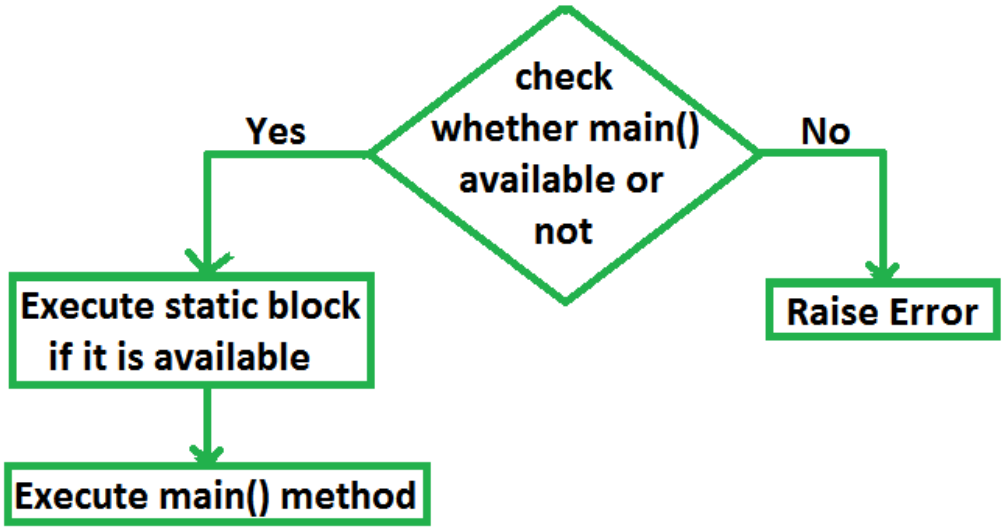
1.7 version :

```
javac Test.java
java Test
output :
static block
main method
```

1.6 version :

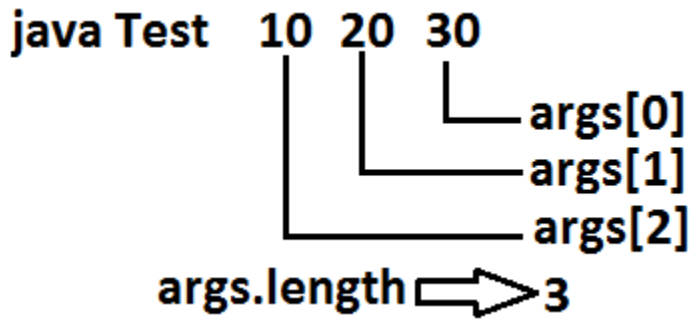


1.7 version :



Command line arguments:

The arguments which are passing from command prompt are called command line arguments.
The main objective of command line arguments are we can customize the behavior of the main() method.



Example 1:

```
class Test
{
        public static void main(String[] args)
        {
                for(int i=0;i<=args.length;i++)
                {
                        System.out.println(args[i]);
                }
        }
}
Output:
java Test x y z
ArrayIndexOutOfBoundsException: 3
```

Replace i<=args.length with i<args.length then it will run successfully.

Example 2 :

```
class Test
{
        public static void main(String[] args)
        {
         String[] argh={"X","Y","Z"};
          args=argh;
                for(String s : args)
                {
                        System.out.println(s);
                }
        }
}
Output:
java Test A B C
X
Y
Z

java Test A B
X
Y
Z

java Test
X


Y
```

```
Z
```

Within the main() method command line arguments are available in the form of String

hence "+" operator acts as string concatenation but not arithmetic addition.

Example 3 :

```
class Test
{
        public static void main(String[] args)
        {
                System.out.println(args[0]+args[1]);
        }
}
Output:
E:\SCJP>javac Test.java
E:\SCJP>java Test 10 20
1020
```

Space is the separator between 2 command line arguments and if our command line argument itself contains space then we should enclose with in double quotes.

Example 4 :

```
class Test
{
        public static void main(String[] args)
        {
                System.out.println(args[0]);
        }
}
Output:
E:\SCJP>javac Test.java
E:\SCJP>java Test "Sai Charan"
Sai Charan
```

Java coding standards

- **Whenever we are writing java code , It is highly recommended to follow coding standards , which improves the readability and understandability of the code.**
- **Whenever we are writing any component(i.e., class or method or variable) the name of the component should reflect the purpose or functionality.**

Example:

```
class A
{
     public int methodOne(int x,int y)
     {
          return x+y;
     }
}
     Ameerpet standards
```

```
package com.durgasoft.scjpdemo;
class Calc
{
     public static int add(int number1,int number2)
     {
          return number1+number2;
     }
}
          Hitech-city standards
```

Coding standards for classes:

- **Usually class names are nouns.**
- **Should starts with uppercase letter and if it contains multiple words every inner word should starts with upper case letter.**

Example:

Coding standards for interfaces:

- **Usually interface names are adjectives.**
- **Should starts with upper case letter and if it contains multiple words every inner word should starts with upper case letter.**

Example:

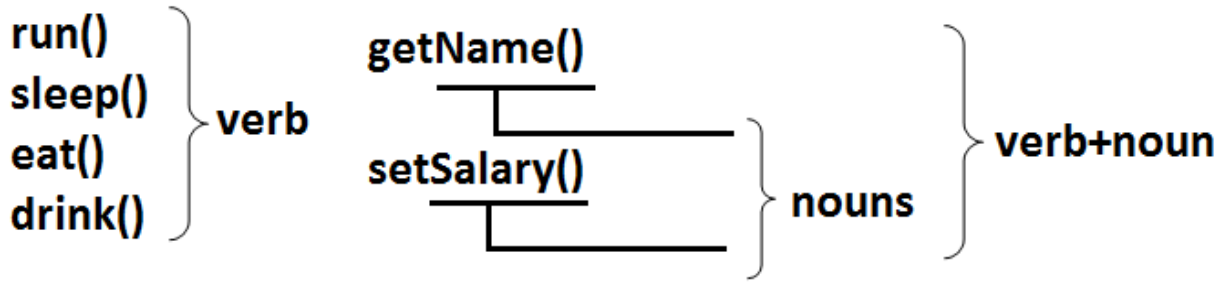
1. **Serializable**
2. **Runnable**
3. **Cloneable**

these are adjectives

Coding standards for methods:

- **Usually method names are either verbs or verb-noun combination.**
- **Should starts with lowercase character and if it contains multiple words every inner word should starts with upper case letter.(camel case convention)**

Example:



Coding standards for variables:

- **Usually variable names are nouns.**
- **Should starts with lowercase alphabet symbol and if it contains multiple words every inner word should starts with upper case character.(camel case convention)**

Example:

```
length
name
salary                          nouns
age
mobileNumber
```

Coding standards for constants:

- **Usually constants are nouns.**
- **Should contain only uppercase characters and if it contains multiple words then these words are separated with underscore symbol.**
- **Usually we can declare constants by using public static final modifiers.**

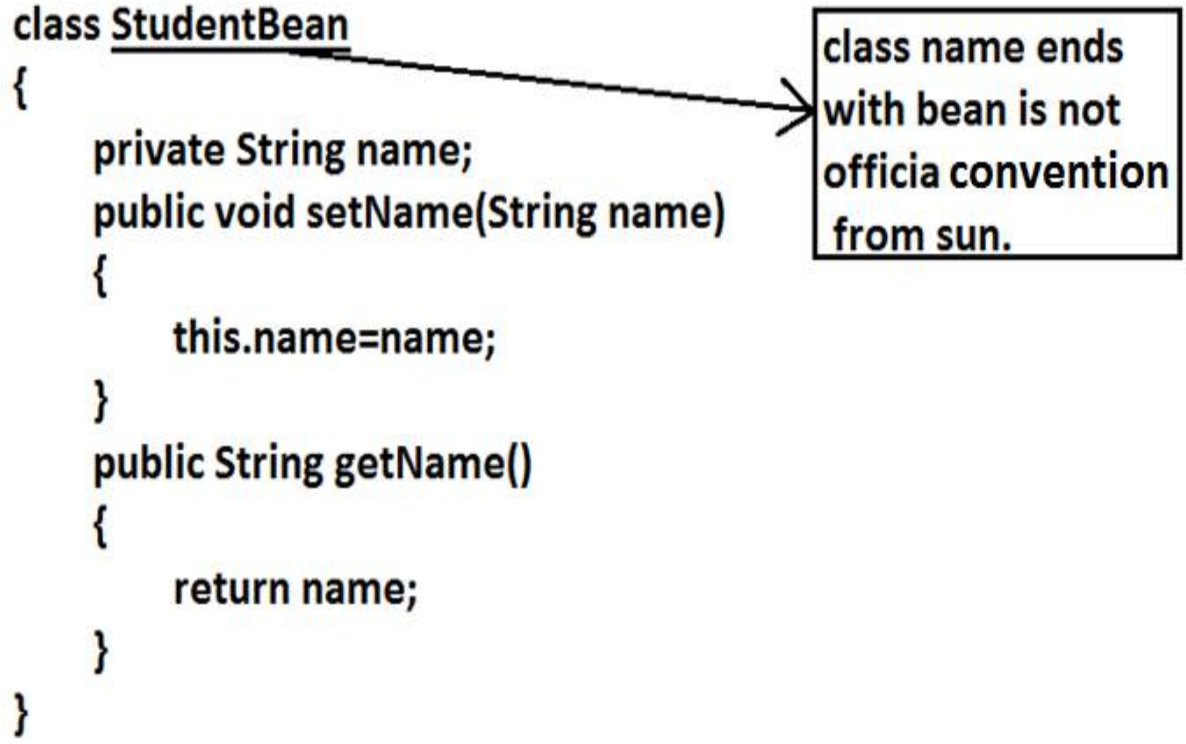
Example:

```
MAX_VALUE
MIN_VALUE                                   nouns
NORM_PRIORITY
```

Java bean coding standards:

A java bean is a simple java class with private properties and public getter and setter methods.

Example:



Syntax for setter method:

1. **Method name should be prefixed with set.**
2. **It should be public.**
3. **Return type should be void.**
4. **Compulsory it should take some argument.**

Syntax for getter method:

1. **The method name should be prefixed with get.**
2. **It should be public.**
3. **Return type should not be void.**
4. **It is always no argument method.**

Note: For the boolean properties the getter method can be prefixed with either get or is. But recommended to use is.

Example:

```
private boolean empty;
public boolean getEmpty()
    {
        return empty;
    }
```

(valid)

```
private boolean empty;
public boolean isEmpty()
    {
        return empty;
    }
```

(valid)

both are valid.

Coding standards for listeners:

To register a listener:

Method name should be prefixed with add.

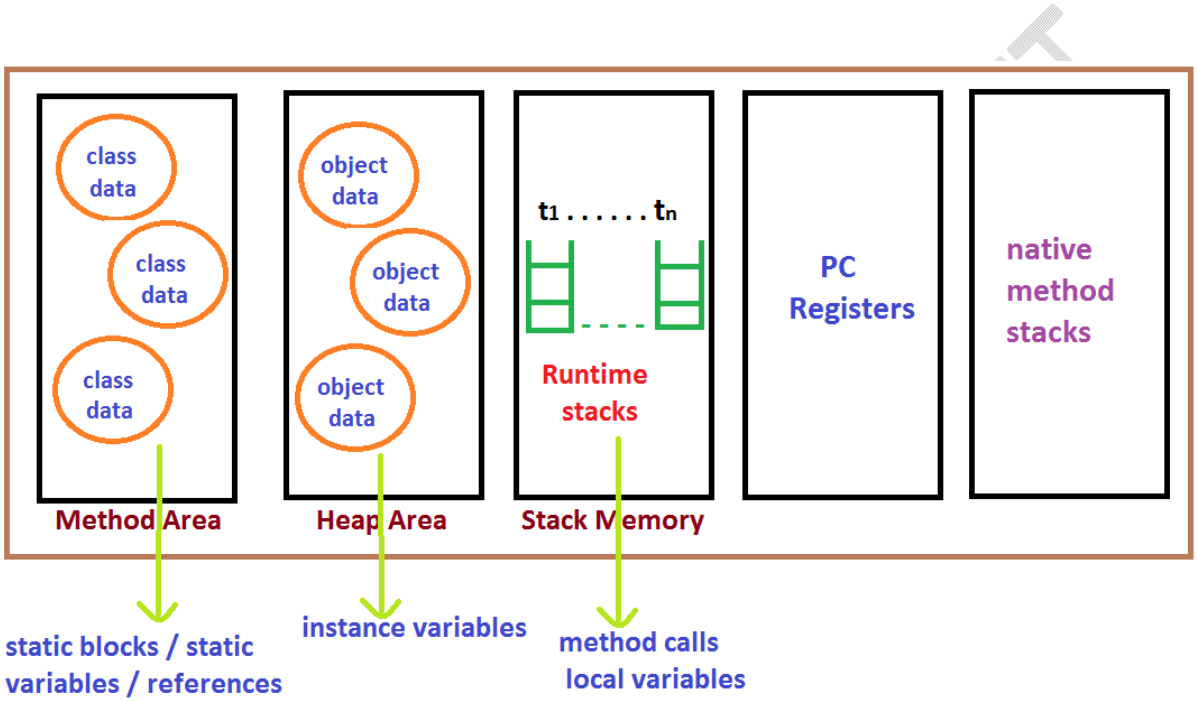
1. **public void addMyActionListener(MyActionListener l) (valid)**
2. **public void registerMyActionListener(MyActionListener l) (invalid)**
3. **public void addMyActionListener(ActionListener l) (invalid)**

To unregister a listener:

The method name should be prefixed with remove.

1. public void removeMyActionListener(MyActionListener l) (valid)
2. publc void unregisterMyActionListener(MyActionListener l) (invalid)
3. public void removeMyActionListener(ActionListener l) (invalid)
4. public void deleteMyActionListener(MyActionListener l) (invalid)

Various Memory areas present inside JVM :



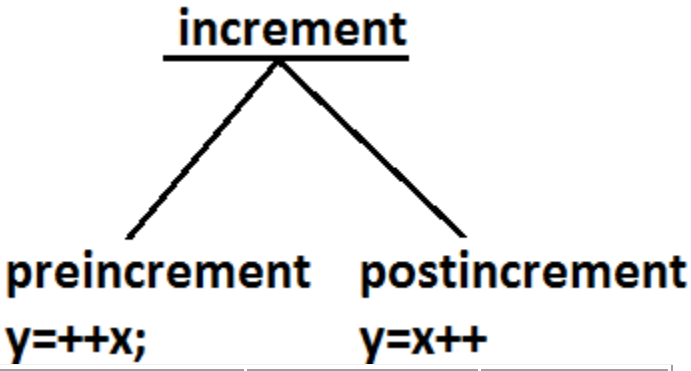
1. Class level binary data includung static variables will be stored in method area.
2. Objects and corresponding instance variables will be stored in Heap area.
3. For every method the JVM will create a Runtime stack all method calls performed by that Thread and corresponding local variables will be stored in that stack.
   Every entry in stack is called Stack Frame or Action Record.
4. The instruction which has to execute next will be stored in the corresponding PC Registers.

OPERATORS &0 ASSIGNMENTS

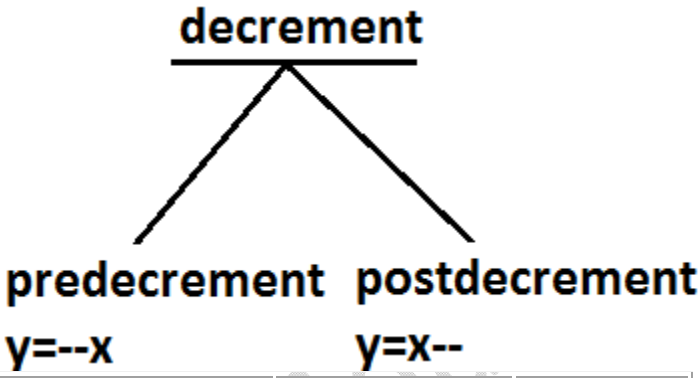
Agenda:

1. **increment & decrement operators**
2. **arithmetic operators**
3. **string concatenation operators**
4. **Relational operators**
5. **Equality operators**
6. **instanceof operators**
7. **Bitwise operators**
8. **Short circuit operators**
9. **type cast operators**
10. **assignment operator**
11. **conditional operator**
12. **new operator**
13. **[ ] operator**
14. **Precedence of java operators**
15. **Evaluation order of java operands**
16. **new Vs newInstance( )**
17. **instanceof Vs isInstance( )**
18. **ClassNotFoundException Vs NoClassDefFoundError**

Increment & Decrement operators :



Increment Operator	pre-increment	ex : y=++x ;
	post-increment	ex: y=x++;



Decrement Operator	pre-decrement	ex : y=--x ;
	post-decrement	ex : y=x-- ;

The following table will demonstrate the use of increment and decrement operators.

Expression	initial value of x	value of y	final value of x
y=++x	10	11	11
y=x++	10	10	11
y=--x	10	9	9
y=x--	10	10	9

Ex :

```
class Test{
public static void main(String[] args){
int x=4;
int y=++x;
System.out.println("value of y :"+y);
}
}
```

output:

```
5
```

```
class Test{
public static void main(String[] args){
int x=4;
int y=++4;
System.out.println("value of y :"+y);
}
}
```

output:

```
compile time error
Test.java:4: unexpected type
required: variable
found   : value
int y=++4;
```

1. **Increment & decrement operators we can apply only for variables but not for constant values.other wise we will get compile time error .**

Ex :

```
int x = 4;
int y = ++x;
System.out.pritnln(y);  //output : 5
```

Ex 2 :

```
int x = 4;
int y = ++4;
System.out.pritnln(y);

C.E: unexpected type
required: varialbe
found : value
```

2. **We can't perform nesting of increment or decrement operator, other wise we will get compile time error**

```
class Test{
public static void main(String[] args){
int x=4;
int y=++(++x); it will become constant
System.out.println("value of y :"+y);
}
}
```

output:

compile time error

```
Test.java:4: unexpected type
required: variable
found   : value
int y=++(++x);
```

```
 int x= 4;
int y = ++(++x);
System.out.println(y);

C.E: unexpected type
required: varialbe
 found : value
```

3. **For the final variables we can't apply increment or decrement operators ,other wise we will get compile time error**

```
class Test{
public static void main(String[] args){
final int x=4;
x++;
System.out.println("value of x:"+x);
}
}
```

output:

compile time error

```
Test.java:4: cannot assign a value to final variable x
x++;
```

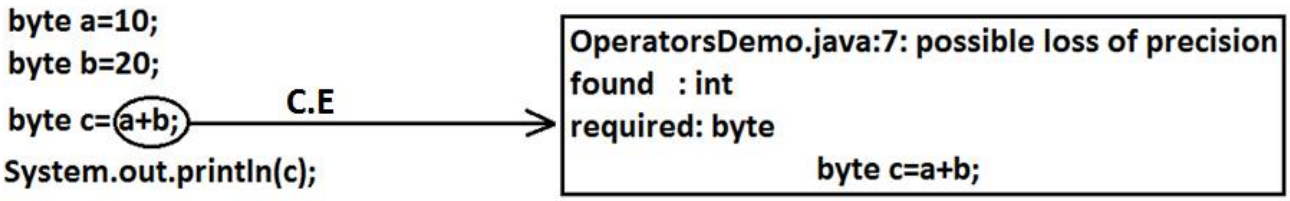
```
Ex:
final int x = 4;
x++;                          //  x = x + 1
System.out.println(x);

C.E :  can't assign a value to final variable 'x' .
```

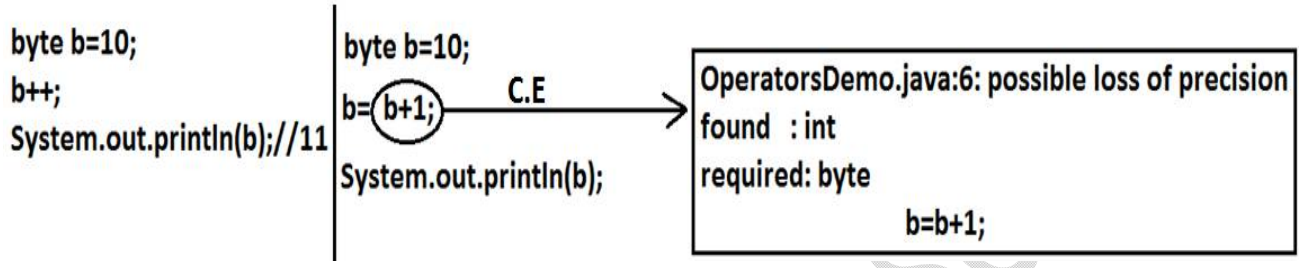
4. **We can apply increment or decrement operators even for primitive data types except boolean .**

```
Ex:
int x=10;
x++;
System.out.println(x);    //output :11

 char ch='a';
  ch++;
  System.out.println(ch); //b

 double d=10.5;
  d++;
  System.out.println(d); //11.5

 boolean  b=true;
  b++;
  System.out.println(b);
  CE : operator ++ can't be applied to boolean
```


Difference between b++ and b = b+1?

If we are applying any arithmetic operators b/w 2 operands 'a' & 'b' the result type is max(int , type of a , type of b)






```
Ex 1:
byte a=10;
byte b=20;
byte c=a+b;                        //byte c=byte(a+b);  valid
System.out.println(c);

CE : possible loss of precession
               found : int
               required : byte
Ex 2:
byte b=20;
byte  b=b+1;                   //byte b=(byte)b+1 ; valid
System.out.println(c);

CE : possible loss of precession
               found : int
               required : byte
```

In the case of Increment & Decrement operators internal type casting will be performed automatically by the compiler

b++; means

b=(type of b)(b+1);

b=(byte)(b+1);

```
b++;  => b=(type of b)b+1;

Ex:
byte b=10;
b++;
System.out.println(b);  //output : 11
```

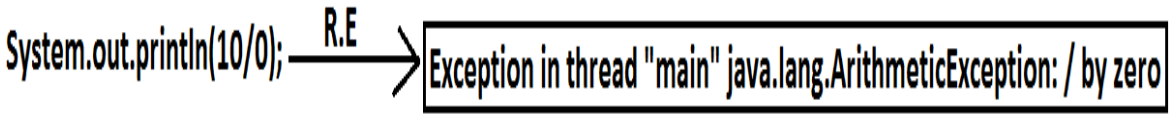
Arithmetic Operator :

1. **If we apply any Arithmetic operation b/w 2 variables a & b , the result type is always max(int , type of a , type of b)**
2. **Example :**

```
3.
4. byte + byte=int
5. byte+short=int
6. short+short=int
7. short+long=long
8. double+float=double
9. int+double=double
10.  char+char=int
11.  char+int=int
12.  char+double=double
13.
14.  System.out.println('a' + 'b');  // output : 195
15.  System.out.println('a' + 1);  // output : 98
16.  System.out.println('a' + 1.2);  // output : 98.2
```

byte+byte=int	int+long=long
byte+short=int	float+double=double
byte+int=int	long+long=long
char+char=int	long+float=float
char+int=int	
byte+char=int	

17. **In integral arithmetic (byte , int , short , long) there is no way to represents infinity , if infinity is the result we will get the ArithmeticException / by zero**
System.out.println(10/0); // output RE : ArithmeticException / by zero
But in floating point arithmetic(float , double) there is a way represents infinity.
System.out.println(10/0.0); // output : infinity



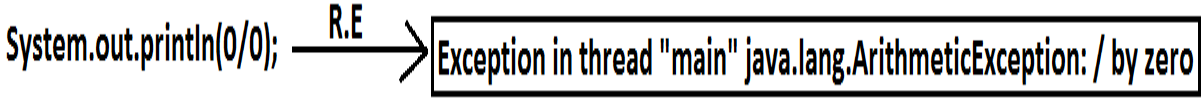
For the Float & Double classes contains the following constants :
1. **POSITIVE_INFINITY**
2. **NEGATIVE_INFINITY**

Hence , if infinity is the result we won't get any ArithmeticException in floating point arithmetics
Ex :
System.out.println(10/0.0); // output : infinity
System.out.println(-10/0.0); // output : - infinity

18. **NaN(Not a Number) in integral arithmetic (byte , short , int , long) there is no way to represent undefine the results. Hence the result is undefined we will get ArithmericException in integral arithmetic**
System.out.println(0/0); // output RE : ArithmeticException / by zero

But floating point arithmetic (float , double) there is a way to represents undefined the results .
For the Float , Double classes contains a constant NaN , Hence the result is undefined we won't get ArithmeticException in floating point arithmetics .
System.out.println(0.0/0.0); // output : NaN
System.out.println(-0.0/0.0); // output : NaN

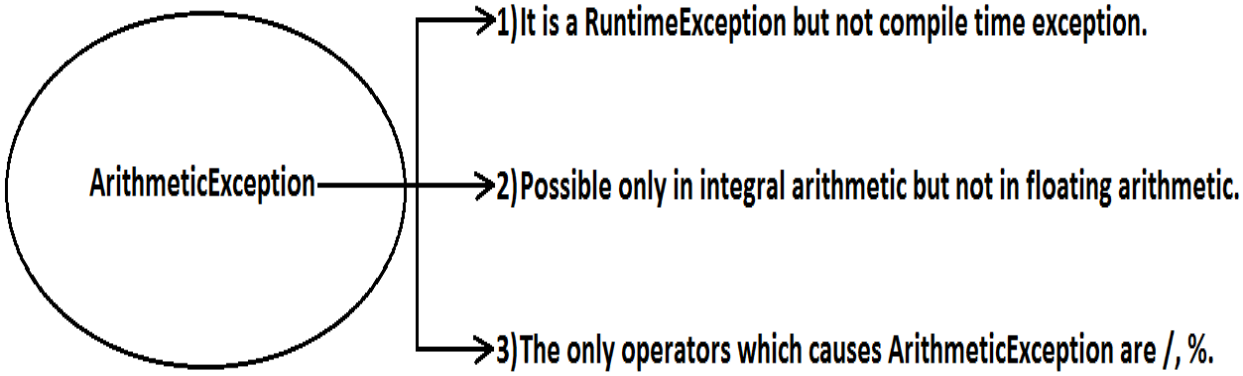
19. For any 'x' value including NaN , the following expressions returns false




```
20.    // Ex :       x=10;
21.  System.out.println(10 < Float.NaN );                  // false
22.  System.out.println(10 <= Float.NaN );          // false
23.  System.out.println(10 > Float.NaN );                  // false
24.  System.out.println(10 >= Float.NaN );          // false
25.  System.out.println(10 == Float.NaN );                 // false
26.  System.out.println(Float.NaN == Float.NaN );          // false
27.
28.  System.out.println(10 != Float.NaN );                          //true
29.  System.out.println(Float.NaN != Float.NaN );          //true
```

30. ArithmeticException :

1. **It is a RuntimeException but not compile time error**
2. **It occurs only in integral arithmetic but not in floating point arithmetic.**
3. **The only operations which cause ArithmeticException are : ' / ' and ' % '**



String Concatenation operator :

1. **The only overloaded operator in java is ' + ' operator some times it access arithmetic addition operator & some times it access String concatenation operator.**
2. **If acts as one argument is String type , then '+' operator acts as concatenation and If both arguments are number type , then operator acts as arithmetic operator**
3. **Ex :**
4.
```
String a="ashok";
int  b=10 , c=20 , d=30 ;
System.out.println(a+b+c+d);  //output : ashok102030
System.out.println(b+c+d+a);    //output : 60ashok
System.out.println(b+c+a+d);    //output : 30ashok30
System.out.println(b+a+c+d);    //output : 10ashok 2030
```

Example :

```
String a="bhaskar";
int b=10,c=20,d=30;
a= b+c+d;          C.E
System.out.println(c);
```

```
E:\scjp>javac OperatorsDemo.java
OperatorsDemo.java:7: incompatible types
found   : int
required: java.lang.String
          a=b+c+d;
```

Example :

```
String a="bhaskar";
int b=10,c=20,d=30;
a=a+b+c;
c=b+d;
c= a+b+d;
System.out.println(a);//bhaskar1020
System.out.println(c);//40
System.out.println(c);
```

```
E:\scjp>javac OperatorsDemo.java
OperatorsDemo.java:9: incompatible types
found   : java.lang.String
required: int
          c=a+b+d;
```

5. **consider the following declaration**
   String a="ashok";
   int b=10 , c=20 , d=30 ;

6.
```
Example :
a=b+c+d ;

CE : incompatible type
          found : int
          required : java.lang.String
```

```
7. Example :
8.
   a=a+b+c ; // valid
9. Example :
10.
   b=a+c+d ;
11.

12.
   CE : incompatible type
13.
            found : java.lang.String
14.
            required : int
15.  Example :
16.
   b=b+c+d ;        // valid
```

Relational Operators(< , <= , > , >=)

We can apply relational operators for every *primitive type* except *boolean* .

```
System.out.println(10>10.5);//false
System.out.println('a'>95.5);//true
System.out.println('z'>'a');//true
System.out.println(true>false);
```

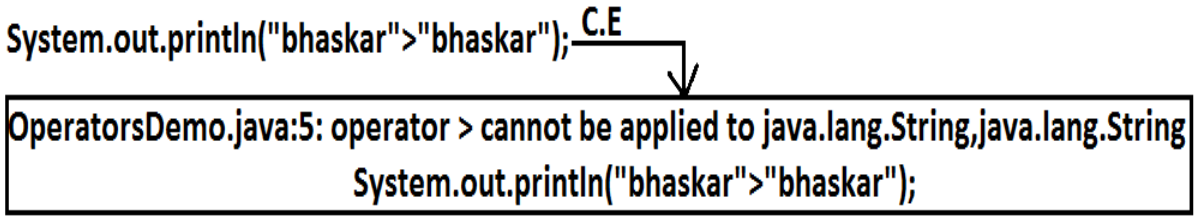
C.E →

```
E:\scjp>javac OperatorsDemo.java
OperatorsDemo.java:8: operator > cannot be applied to boolean,boolean
        System.out.println(true>false);
```



```
1.  System.out.println(10 < 10.5);    //true
2.  System.out.println('a' > 100.5);    //false
3.  System.out.println('b'  >  'a');    //true
4.  System.out.println(true > false);
5.   //CE : operator  >  can't  be  applied  to  boolean , boolean
```

6. **We can't apply relational operators for object types**




```
7. System.out.println("ashok123" > "ashok");
8.  //  CE: operator > can't be applied to java.lang.String ,
   java.lang.String
```

9. **Nesting of relational operator is not allowed**


```
10.  System.out.println(10 >  20 > 30); //  System.out.println(true  >
   30);
11.    //CE : operator  > can't  be  applied  to  boolean , int
```

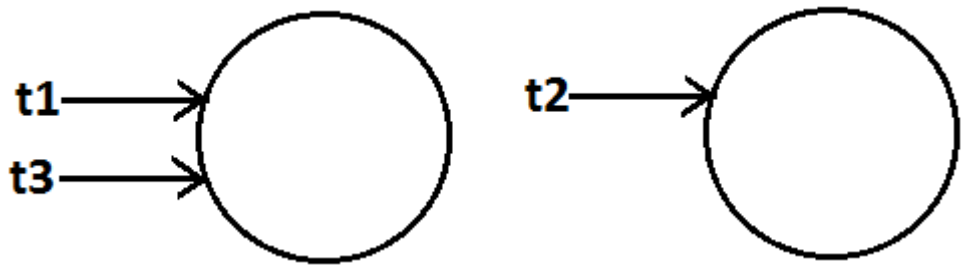
Equality Operators : (== , !=)

1. **We can apply equality operators for every primitive type including boolean type also**

```
2.          System.out.println(10 == 20) ;  //false
3.          System.out.println('a' == 'b' );   //false
4.          System.out.println('a' == 97.0 )  //true
5.          System.out.println(false == false)   //true
```

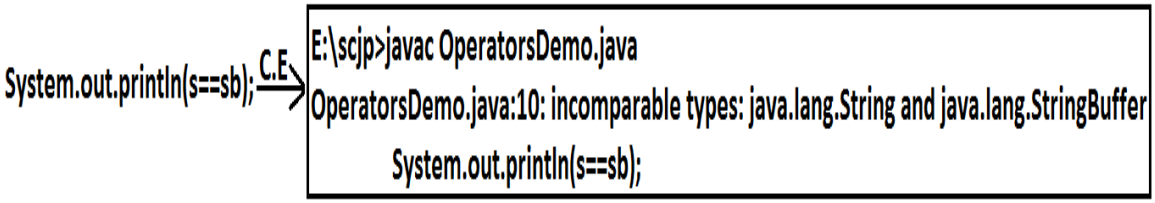
6. **We can apply equality operators for object types also .**
   For object references r1 and r2 , r1 == r2 returns true if and only if both r1 and r2 pointing to the same object. i.e., == operator meant for reference-comparision Or address-comparision.

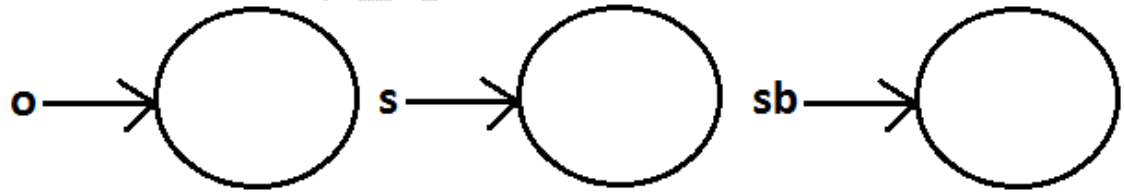
```
7.          Thread t1=new Thread( ) ;
8.          Thread t2=new Thread( );
9.          Thread t3=t1 ;
10.                System.out.println(t1==t2);  //false
11.                System.out.println(t1==t3);  //true
```



12. **To use the equality operators between object type compulsory these should be some relation between argument types(child to parent , parent to child) , Otherwise we will get Compiletime error incompatible types**

```
13.  Thread t=new Thread( ) ;
14.  Object o=new Object( );
15.  String s=new String("durga");
16.  System.out.println(t ==o);    //false
17.  System.out.println(o==s);     //false
18.  System.out.println(s==t);
19.  CE : incompatible types :  java.lang.String and java.lang.Thread
```



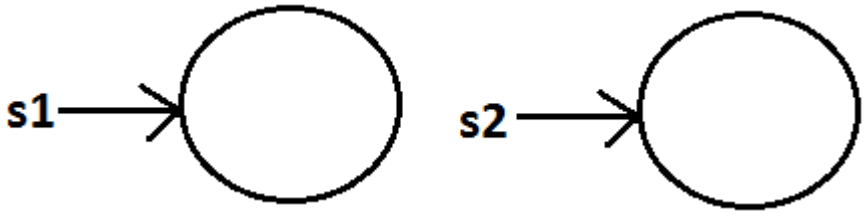


20. **For any object reference of on r==null is always false , but null==null is always true .**

```
21.          String s= new String("ashok");
22.          System.out.println(s==null);  //output : false
23.          String   s=null ;
24.          System.out.println(r==null);  //true
25.          System.out.println(null==null); //true
```

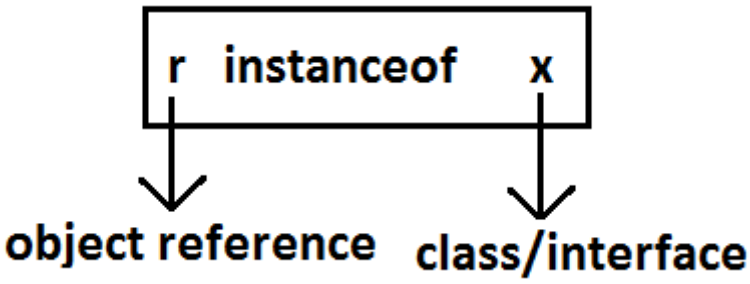
26. **<u>What is the difference between == operator and .equals( ) method ?</u>**
In general we can use .equals( ) for content comparision where as == operator for reference comparision

```
27.
28.                          String  s1=new String("ashok");
29.                          String  s2=new  String("ashok");
30.                          System.out.println(s1==s2);  //false
31.                          System.out.println(s1.equals(s2));  //true
```



instanceof operator :

1. **We can use the instanceof operator to check whether the given an object is perticular type or not**



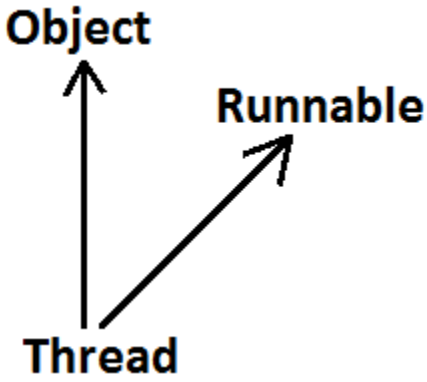

```
2.          Object o=l.get(0);          // l is an array name
3.          if(o  instanceof Student) {
4.            Student s=(Student)o ;
5.                  //perform  student  specific  operation
6.            }
7.            elseif(o instanceof Customer) {
8.              Customer c=(Customer)o;
9.                //perform  Customer specific  operations
10.                  }
```

11. O instanceof X here O is object reference , X is ClassName/Interface name

```
12.                 Thread  t = new  Thread( );
13.                 System.out.println(t  instanceof Thread);   //true
14.                 System.out.println(t instanceof Object);     //true
```

```
15.                System.out.println(t instanceof Runnable);  //true
```

```
Ex :
          public class Thread extends Object  implements Runnable {
          }
```



16. To use instance of operator compulsory there should be some relation between argument types (either child to parent Or parent to child Or same type) Otherwise we will get compile time error saying inconvertible types

```
String s=new String("bhaskar");
System.out.println(s instanceof Thread);
```

C.E →

```
E:\scjp>javac OperatorsDemo.java
OperatorsDemo.java:6: inconvertible types
found   : java.lang.String
required: java.lang.Thread
        System.out.println(s instanceof Thread);
```



```
17.
18.                Thread t=new Thread( );
19.                System.out.println(t  instanceof String);
20.                       CE : inconvertable  errors
21.                                    found : java.lang.Thread
22.                                    required : java.lang.String
```

23. Whenever we are checking the parent object is child type or not by using instanceof operator that we get false.

```
24.                 Object o=new Object( );
25.                 System.out.println(o  instanceof  String );
    //false
26.
27.                 Object o=new String("ashok");
28.                 System.out.println(o  instanceof String);   //true
```

29. For any class or interface X null instanceof X is always returns false

```
30.                 System.out.println(null  instanceof  X);  //false
```

Bitwise Operators : (& , | , ^)

1. & (AND) : If both arguments are true then only result is true.
2. | (OR) : if at least one argument is true. Then the result is true.
3. ^ (X-OR) : if both are different arguments. Then the result is true.

```
Example:
System.out.println(true&false);//false
System.out.println(true|false);//true
System.out.println(true^false);//true
```

We can apply bitwise operators even for integral types also.

```
Example:
System.out.println(4&5);//4                    using binary digits
System.out.println(4|5);//5                                4-->100
System.out.println(4^5);//1                                5-->101
```

Example :

System.out.println(4&5);//4	100	100	100
System.out.println(4|5);//5	101	101	101
System.out.println(4^5);//1	100	101	001

Bitwise complement (~) (tilde symbol) operator:

1. We can apply this operator only for *integral types* but not for boolean types.

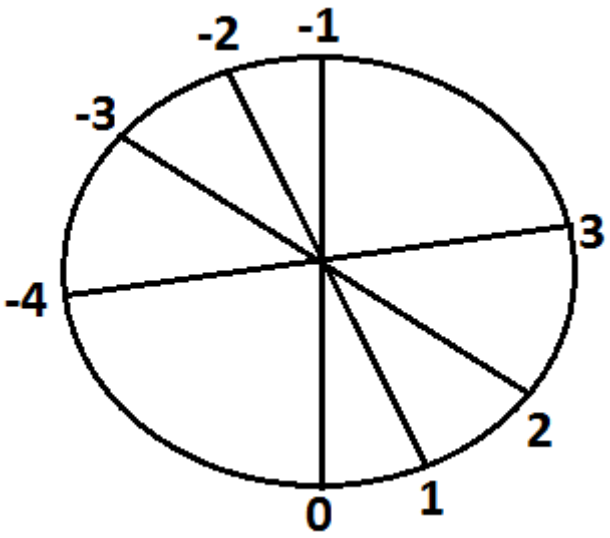
System.out.println(~true); C.E →

```
E:\scjp>javac OperatorsDemo.java
OperatorsDemo.java:5: operator ~ cannot be applied to boolean
                System.out.println(~true);
```

```
2. Example :
3. System.out.println(~true); // CE :opetator ~ can not be applied to
   boolean
4. System.out.println(~4);  //-5
5.
6. description about above program :
7. 4-->  0 000.......0100                  0-----+ve
8. ~4--> 1 111.......1011                   1--- -ve
```

```
9.
10.  2's compliment  of ~4 -->  000....0100 add  1
11.   result is : 000...0101 =5
```

12. **Note : The most significant bit access as sign bit 0 means +ve number , 1 means -ve number.**
    +ve number will be represented directly in memory where as -ve number will be represented in 2's comlement form.



Boolean complement (!) operator:

This operator is applicable only for *boolean types* but not for integral types.

System.out.println(!4); C.E →

```
E:\scjp>javac OperatorsDemo.java
OperatorsDemo.java:5: operator ! cannot be applied to int
        System.out.println(!4);
```

Example :

```
Example:
System.out.println(!true);//false
System.out.println(!false);//true
System.out.println(!4);//CE : operator ! can not be applied to int
```

Summary:

```
&
|      Applicable for both boolean and integral types.
^
~ --------Applicable for integral types only but not for boolean types.
! --------Applicable for boolean types only but not for integral types.
```

Short circuit (&&, ||) operators:

These operators are exactly same as normal bitwise operators &(AND), |(OR) except the following differences.

& , |	&& , ||
Both arguments should be evaluated always.	Second argument evaluation is optional.
Relatively performance is low.	Relatively performance is high.
Applicable for both integral and boolean types.	Applicable only for boolean types but not for integral types.

x&&y : y will be evaluated if and only if x is true.(If x is false then y won't be evaluated i.e., If x is ture then only y will be evaluated)

x||y : y will be evaluated if and only if x is false.(If x is true then y won't be evaluated i.e., If x is false then only y will be evaluated)

Example :

```
        int x=10 , y=15 ;
        if(++x < 10  ||  ++y > 15) {    //instead of || using  &,&&, |
operators
         x++;
        }
        else {
                y++;
        }

        System.out.println(x+"----"+y);
```

Output:

operator	x	y
&	11	17
|	12	16
&&	11	16
||	12	16

Example :

```
        int x=10  ;
        if(++x < 10  && ((x/0)>10) ) {
         System.out.println("Hello");
        }
        else {
                System.out.println("Hi");
        }

        output : Hi
```

Type Cast Operator :

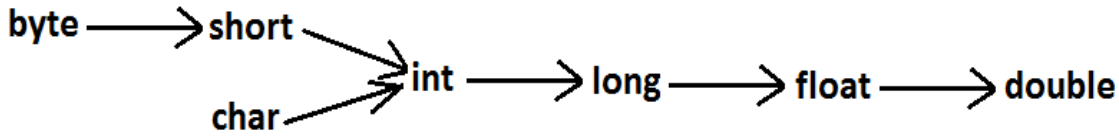
There are 2 types of type-casting

1. **implicit**
2. **explicit**

implicit type casting :

```
int x='a';
System.out.println(x);  //97
```

1. **The compiler is responsible to perform this type casting.**
2. **When ever we are assigning lower datatype value to higher datatype variable then implicit type cast will be performed .**
3. **It is also known as Widening or Upcasting.**
4. **There is no lose of information in this type casting.**
5. **The following are various possible implicit type casting.**
   Diagram:



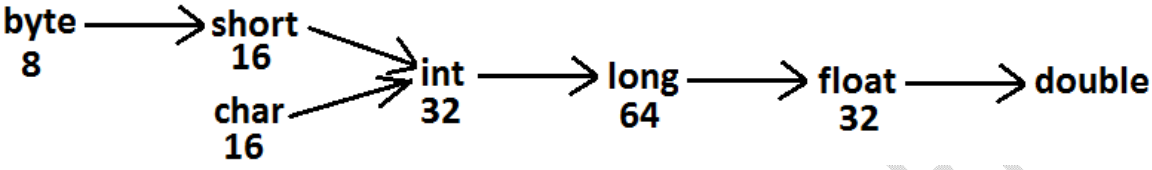
6.
7. **Example 1:**
8. `int x='a';`
9. `System.out.println(x);//97`
10. **Note: Compiler converts char to int type automatically by implicit type casting.**
11. **Example 2:**
12. `double d=10;`
13. `System.out.println(d);//10.0`

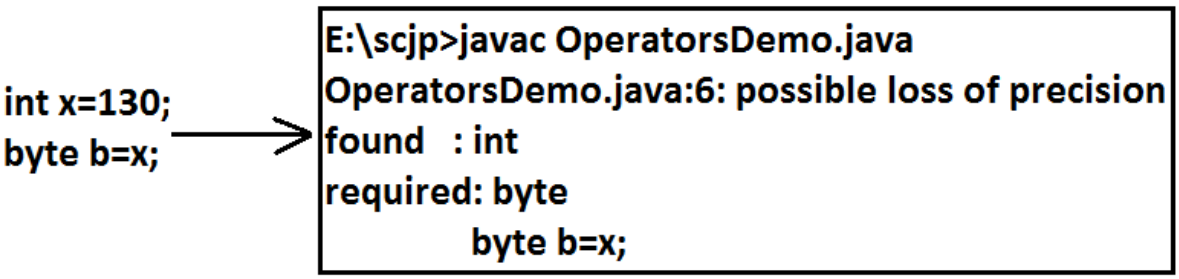
Note: Compiler converts int to double type automatically by implicit type casting.

Explicit type casting:

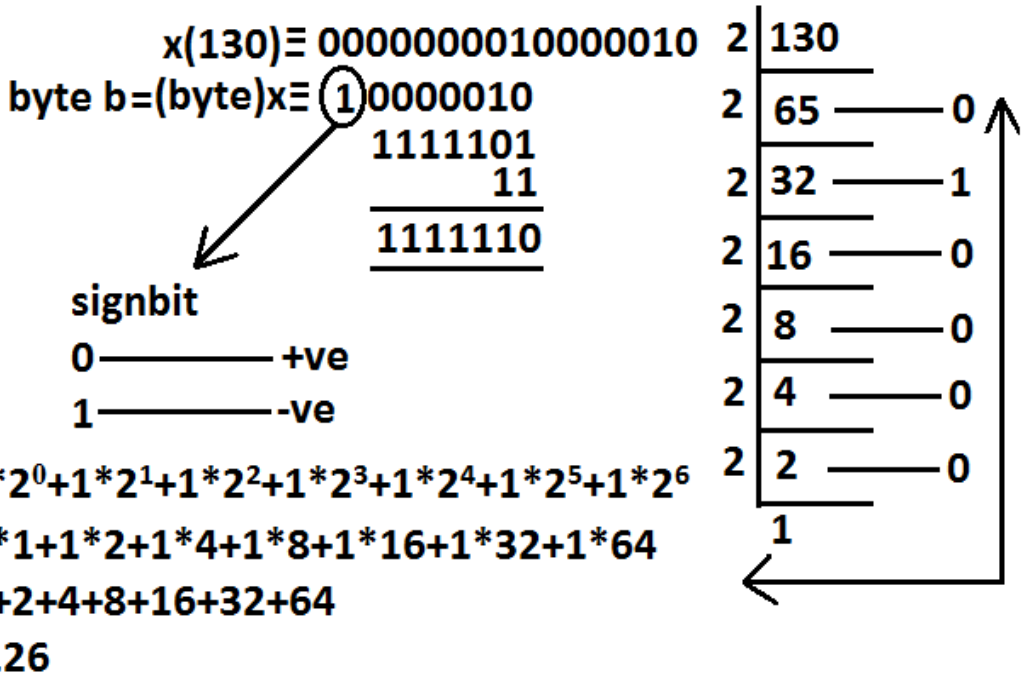
1. **Programmer is responsible for this type casting.**
2. **Whenever we are assigning bigger data type value to the smaller data type variable then explicit type casting is required.**
3. **Also known as Narrowing or down casting.**
4. **There may be a chance of lose of information in this type casting.**
5. **The following are various possible conversions where explicit type casting is required.**

Diagram:





6.
7. Example :
8.
9. int x=130;
10. byte b=(byte)x;
11. System.out.println(b); //-126
12.



13. Example 2 :
14.
15. int x=130;

```
byte b=x;
System.out.println(b);  //CE : possible loss of precision
```

18. When ever we are assigning higher datatype value to lower datatype value variable by explicit type-casting ,the most significant bits will be lost i.e., we have considered least significant bits.

```
Example 3 :

int x=150;
short s=(short)x;
byte b=(byte)x;
System.out.println(s); //150
System.out.println(b);  //-106
```

26. When ever we are assigning floating point value to the integral types by explicit type casting , the digits of after decimal point will be lost .

```
Example 4:

double d=130.456 ;

int x=(int)d ;
System.out.println(x);  //130

byte b=(byte)d ;
System.out.println(b);  //-206
```

```
float x=150.1234f;
int i=(int)x;
System.out.println(i);//150
```

```
double d=130.456;
int i=(int)d;
System.out.println(i);//130
```

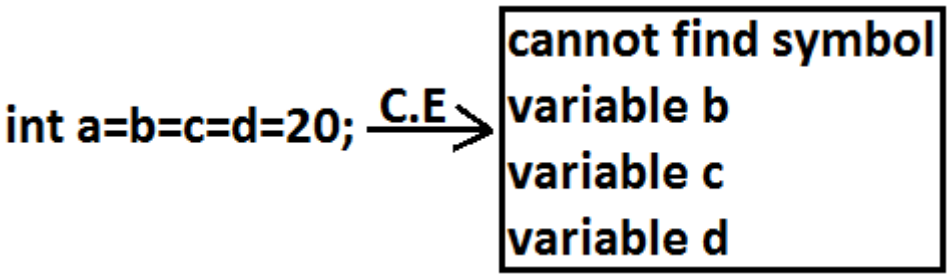
Assignment Operator :

There are 3 types of assignment operators

1. **Simple assignment:**
   Example: **int x=10;**
2. **Chained assignment:**

```
Example:
int a,b,c,d;
a=b=c=d=20;
System.out.println(a+"---"+b+"---"+c+"---"+d);//20---20---20---20
int b , c , d ;
int a=b=c=d=20 ;  //valid
```

We can't perform chained assignment directly at the time of declaration.

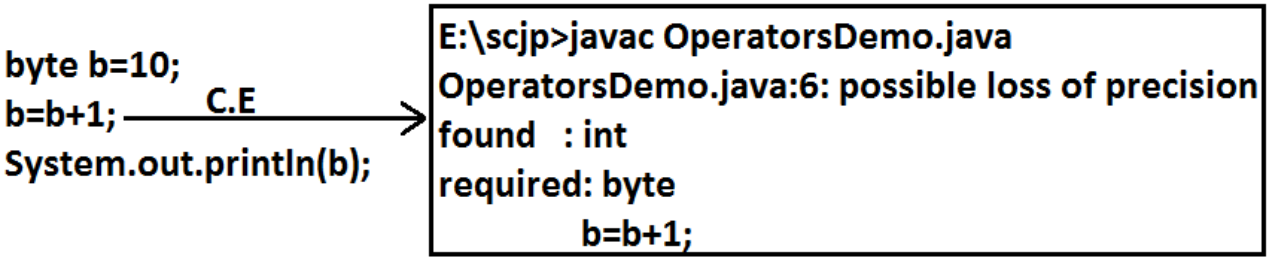



```
Example 2:

int a=b=c=d=30;
  CE : can not find symbol
                symbol : variable b
                location : class Test
```

9. **Compound assignment:**
   1. **Sometimes we can mixed assignment operator with some other operator to form compound assignment operator.**
   2. **Ex:**
   3. **int a=10 ;**
   4. **a +=20 ;**
   5. **System.out.println(a); //30**
   6. **The following is the list of all possible compound assignment operators in java.**

+=	&=	>>=
-=	|=	>>>=
*=	^=	<<=
/=		
%=		

   7. **In the case of compound assignment operator internal type casting will be performed automatically by the compiler (similar to increment and decrement operators.)**



```
byte b=10;
b++;
System.out.println(b);//11
```

```
byte b=10;
//b+=1;
b=(byte)(b+1);
System.out.println(b);//11
```

```
int a,b,c,d;
a=b=c=d=20;
a+=b-=c*=d/=2;
System.out.println(a+"--"+b+"---"+c+"---"+d);
                    //-160---180---200---10
```

```
byte b=10;
b=b+1;
System.out.println(b);

CE :
 possible loss of precission
        found : int
        required : byte
```

```
byte b=10;
b++;
System.out.println(b); //11
```

```
byte b=10;
b+=1;
System.out.println(b); //11
```

```
byte b=127;
b+=3;
System.out.println(b);
        //-126
```

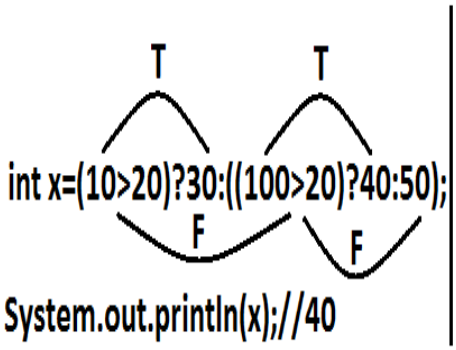
```
Ex :
int a , b , c , d ;
a=b=c=d=20 ;
a += b-= c *= d /= 2 ;
System.out.println(a+"---"+b+"---"+c+"---"+d);// -160...-180---200---10
```

Conditional Operator (? :)

The only possible ternary operator in java is conditional operator

```
Ex 1 :
int x=(10>20)?30:40;
System.out.println(x); //40

Ex 2 :
int x=(10>20)?30:((40>50)?60:70);
System.out.println(x); //70
```

Nesting of conditional operator is possible




```
int x=(10>20)?30:((100<20)?40:50);
System.out.println(x);//50
```

```
int a=10,b=20;
byte c1=(10>20)?30:40;
byte c2=(10<20)?30:40;
System.out.println(c1);//40
System.out.println(c2);//30
```

```
int a=10,b=20;
byte c1=(a>b)?30:40;
byte c2=(a<b)?30:40;
System.out.println(c1);
System.out.println(c2);
```

C.E

```
E:\scjp>javac OperatorsDemo.java
OperatorsDemo.java:6: possible loss of precision
found   : int
required: byte
        byte c1=(a>b)?30:40;
```

new operator :

1. **We can use "new" operator to create an object.**
2. **There is no "delete" operator in java because destruction of useless objects is the responsibility of garbage collector.**

[ ] operator:

We can use this operator to declare under construct/create arrays.

Java operator precedence:

1. **Unary operators: [] , x++ , x-- , ++x , --x , ~ , ! , new , <type>**
2. **Arithmetic operators : * , / , % , + , - .**
3. **Shift operators : >> , >>> , << .**
4. **Comparision operators : <, <=,>,>=, instanceof.**
5. **Equality operators: == , !=**
6. **Bitwise operators: & , ^ , | .**
7. **Short circuit operators: && , || .**
8. **Conditional operator: (?:)**
9. **Assignment operators: += , -= , *= , /= , %= . . .**

Evaluation order of java operands:

There is no precedence for operands before applying any operator all operands will be evaluated from left to right.

```
Example:
class OperatorsDemo {
        public static void main(String[] args){
                System.out.println(m1(1)+m1(2)*m1(3)/m1(4)*m1(5)+m1(6));
        }
        public static int m1(int i)    {
                System.out.println(i);
                return i;
        }
}
```

output:	Analysis:
1	1+2*3/4*5+6
2	1+6/4*5+6
3	1+1*5+6
4	1+5+6
5	12
6	
12	

```
int x=10;
x=++x;
System.out.println(x);//11
```

```
int x=10;
x=x+1;
System.out.println(x);//11
```

```
int x=10;
int y=x++;
System.out.println(y);//10
System.out.println(x);//11
```

Ex 2:

```
int i=1;
i+=++i + i++ + ++i + i++;
System.out.println(i); //13

description :
i=i +  ++i  +  i++   +  ++i  +   i++ ;
i=1+2+2+4+4;
i=13;
```

new Vs newInstance( ) :

1. new is an operator to create an objects , if we know class name at the beginning then we can create an object by using new operator .
2. newInstance( ) is a method presenting class " Class " , which can be used to create object.
3. If we don't know the class name at the beginning and its available dynamically Runtime then we should go for newInstance() method

```
4.  public  class Test {
5.    public static void main(String[] args) Throws Exception {
6.          Object o=Class.forName(arg[0]).newInstance( ) ;
7.          System.out.println(o.getClass().getName( ) );
8.    }
9.  }
```

10. If dynamically provide class name is not available then we will get the RuntimeException saying ClassNotFoundException
11. To use newInstance( ) method compulsory corresponding class should contains no argument constructor , otherwise we will get the RuntimeException saying InstantiationException.

Difference between *new* and *newInstance( )* :

new	newInstance( )
new is an operator , which can be used to create an object	newInstance( ) is a method , present in class Class , which can be used to create an object .
We can use new operator if we know the class name at the beginning. Test t= new Test( );	We can use the newInstance( ) method , If we don't class name at the beginning and available dynamically Runtime. Object o=Class.forName(arg[0]).newInstance( );
If the corresponding .class file not available at Runtime then we will get RuntimeException saying NoClassDefFoundError , It is unchecked	If the corresponding .class file not available at Runtime then we will get RuntimeException saying ClassNotFoundException , It is checked

To used new operator the corresponding class not required to contain no argument constructor	To used newInstance( ) method the corresponding class should compulsory contain no argument constructor , Other wise we will get RuntimeException saying InstantiationException.

Difference between ClassNotFoundException & NoClassDefFoundError :

1. For hard coded class names at Runtime in the corresponding .class files not available we will get NoClassDefFoundError , which is unchecked
   Test t = new Test( );
   In Runtime Test.class file is not available then we will get NoClassDefFoundError
2. For Dynamically provided class names at Runtime , If the corresponding .class files is not available then we will get the RuntimeException saying ClassNotFoundException
   Ex : Object o=Class.forname("Test").newInstance( );
   At Runtime if Test.class file not available then we will get the ClassNotFoundException , which is checked exception

Difference between instanceof and isInstance( ) :

instanceof	isInstance( )
instanceof an operator which can be used to check whether the given object is perticular type or not We know at the type at beginning it is available	isInstance( ) is a method , present in class Class , we can use isInstance( ) method to checked whether the given object is perticular type or not We don't know at the type at beginning it is available Dynamically at Runtime.

`String s = new String("ashok");`
`System.out.println(s instanceof Object );`
`//true`
If we know the type at the beginning only.
	`class Test {`
	`public static void main(String[] args) {`
	`Test t = new Test( ) ;`
	`System.out.println(`
	`Class.forName(args[0]).isInstance( ));`
	
	`//arg[0] --- We don't know the type at beginning`
	`}`
	`}`
	`java Test Test //true`
	`java Test String //false`
	`java Test Object //true`

`int x= 10 ;`
`x=x++;`
`System.out.println(x); //10`
	1. consider old value of x for assignment x=10
	2. Increment x value x=11
	3. Perform assignment with old considered x value x=10

Flow Control

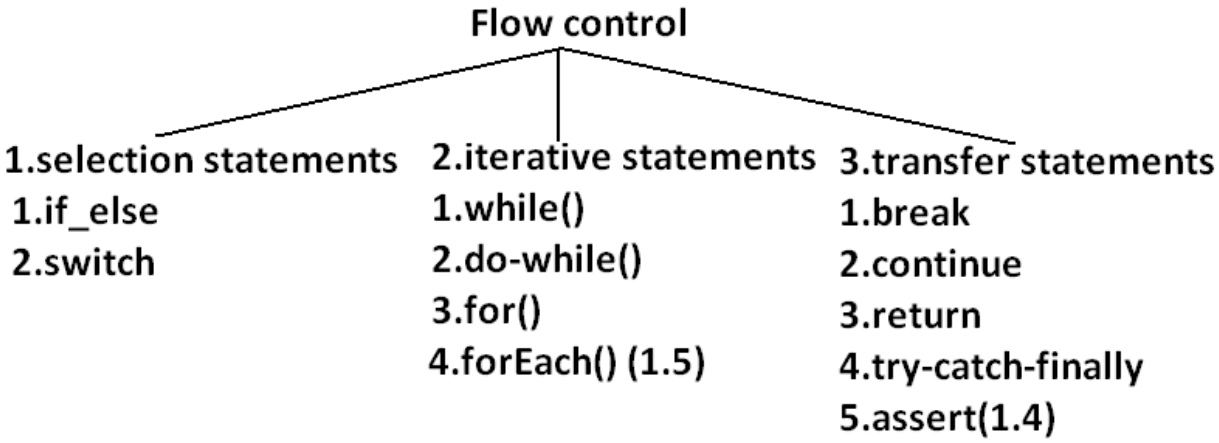
Agenda :

1. **Introduction**
2. **Selection statements**
   - i. **if-else**
   - ii. **Switch**
     - **Case Summary**
     - **fall-through inside a switch**
     - **default case**
3. **Iterative Statements**
   - i. **While loop**
     - **Unreachable statement in while**
   - ii. **Do-while**
     - **Unreachable statement in do while**
   - iii. **For Loop**
     - **Initilizationsection**
     - **Conditional check**
     - **Increment and decrement section**
     - **Unreachable statement in for loop**
   - iv. **For each**
     - **Iterator Vs Iterable(1.5v)**
     - **Difference between Iterable and Iterator**
4. **Transfer statements**
   - **Break statement**
   - **Continue statement**
   - **Labeled break and continue statements**
   - **Do-while vs continue (The most dangerous combination)**

Introduction :

Flow control describes the order in which all the statements will be executed at run time.

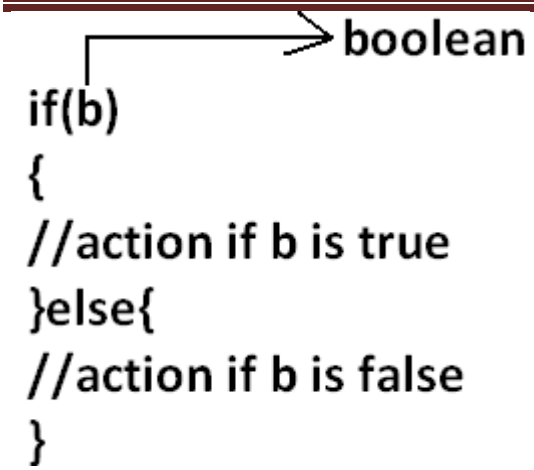
Diagram:



Selection statements:

if-else:

syntax:



The argument to the if statement should be Boolean by mistake if we are providing any other type we will get "compile time error".

Example 1:

```
 public class ExampleIf{
public static void main(String args[]){
int x=0;
if(x)
{
System.out.println("hello");
}else{
System.out.println("hi");
}}}
OUTPUT:
Compile time error:
D:\Java>javac ExampleIf.java
ExampleIf.java:4: incompatible types
found   : int
required: boolean
if(x)
```

Example 2:

```
public class ExampleIf{
public static void main(String args[]){
int x=10;
if(x=20)
{
System.out.println("hello");
}
else{
System.out.println("hi");
```

```
}}}
OUTPUT:
Compile time error
D:\Java>javac ExampleIf.java
ExampleIf.java:4: incompatible types
found   : int
required: boolean
if(x=20)
```

Example 3:

```
public class ExampleIf{
public static void main(String args[]){
int x=10;
if(x==20)
{
System.out.println("hello");
}else{
System.out.println("hi");
}}}
OUTPUT:
Hi
```

Example 4:

```
public class ExampleIf{
public static void main(String args[]){
boolean b=false;
if(b=true)
{
System.out.println("hello");
}else{
System.out.println("hi");
}}}
OUTPUT:
Hello
```

Example 5:

```
public class ExampleIf{
public static void main(String args[]){
boolean b=false;
if(b==true)
{
System.out.println("hello");
}else{
System.out.println("hi");
}}}
OUTPUT:
Hi
```

Both else part and curly braces are optional.
Without curly braces we can take only one statement under if, but it should not be declarative statement.

Example 6:

```
 public class ExampleIf{
public static void main(String args[]){
if(true)
System.out.println("hello");
}}
OUTPUT:
Hello
```

Example 7:

```
public class ExampleIf{
public static void main(String args[]){
if(true);
}}
OUTPUT:
No output
```

Example 8:

```
public class ExampleIf{
public static void main(String args[]){
if(true)
int x=10;
}}
OUTPUT:
Compile time error
D:\Java>javac ExampleIf.java
ExampleIf.java:4: '.class' expected
int x=10;
ExampleIf.java:4: not a statement
int x=10;
```

Example 9:

```
public class ExampleIf{
public static void main(String args[]){
if(true){
int x=10;
}}}
OUTPUT:
D:\Java>javac ExampleIf.java
D:\Java>java ExampleIf
```


Example 10:

```
public class ExampleIf{
public static void main(String args[]){
if(true)
System.out.println("hello"); ——————> dependent statement on if
System.out.println("hi"); ——————> this is independent statement on if
}
}
```

```
OUTPUT:
Hello
Hi
```

Semicolon(;) is a valid java statement which is call empty statement and it won't produce any output.

Switch:

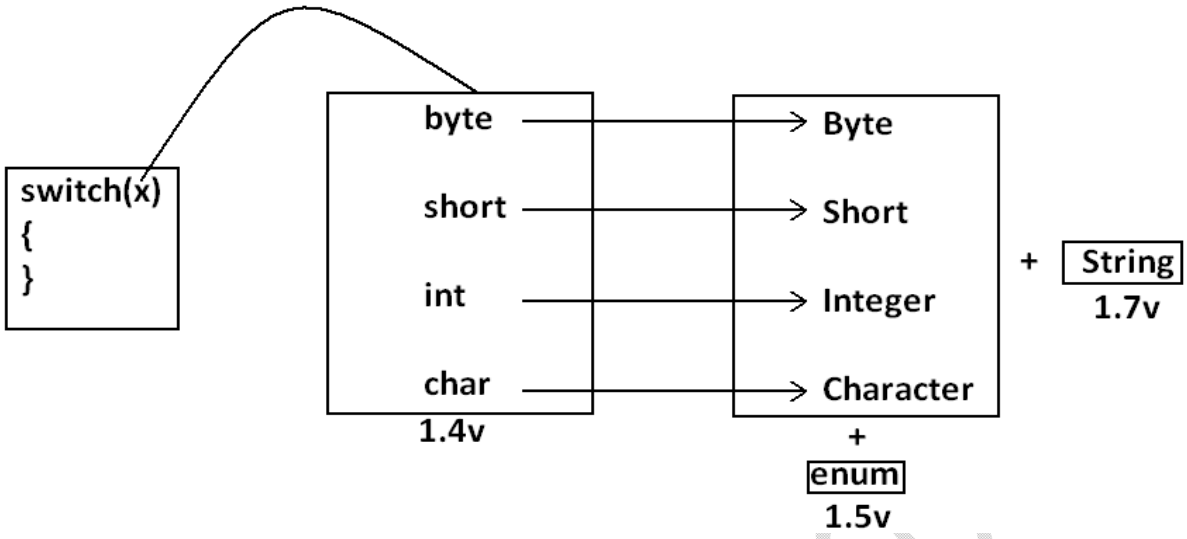
If several options are available then it is not recommended to use if-else we should go for switch statement.Because it improves readability of the code.

Syntax:

```
switch(x)
{
case 1:
action1
case 2:
action2
.
.
.
default:
default action
}
```

Until 1.4 version the allow types for the switch argument are byte, short, char, int but from 1.5 version on wards the corresponding wrapper classes (Byte, Short, Character, Integer) and "enum" types also allowed.

Diagram:



- **Curly braces are mandatory.(except switch case in all remaining cases curly braces are optional)**
- **Both case and default are optional.**
- **Every statement inside switch must be under some case (or) default. Independent statements are not allowed.**

Example 1:

```
public class ExampleSwitch{
public static void main(String args[]){
int x=10;
switch(x)
{
System.out.println("hello");
}}}
OUTPUT:
Compile time error.
D:\Java>javac ExampleSwitch.java
ExampleSwitch.java:5: case, default, or '}' expected
System.out.println("hello");
```

Every case label should be "compile time constant" otherwise we will get compile time error.

Example 2:

```
public class ExampleSwitch{
public static void main(String args[]){
int x=10;
int y=20;
switch(x)
{
case 10:
System.out.println("10");
case y:
System.out.println("20");
```

```
}}}
OUTPUT:
Compile time error
D:\Java>javac ExampleSwitch.java
ExampleSwitch.java:9: constant expression required
case y:
```

If we declare y as final we won't get any compile time error.

Example 3:

```
public class ExampleSwitch{
public static void main(String args[]){
int x=10;
final int y=20;
switch(x)
{
case 10:
System.out.println("10");
case y:
System.out.println("20");
}}}
OUTPUT:
10
20
```


But switch argument and case label can be expressions , but case label should be constant expression.

Example 4:

```
public class ExampleSwitch{
public static void main(String args[]){
int x=10;
switch(x+1)
{
case 10:
case 10+20:
case 10+20+30:
}}}
OUTPUT:
No output.
```

Every case label should be within the range of switch argument type.

Example 5:

```
public class ExampleSwitch{
public static void main(String args[]){
byte b=10;
switch(b)
{
case 10:
System.out.println("10");
case 100:
System.out.println("100");




case 1000:
System.out.println("1000");
}}}
```

```
OUTPUT:
Compile time error
D:\Java>javac ExampleSwitch.java
ExampleSwitch.java:10: possible loss of precision
found   : int
required: byte
case 1000:
```

Example :

```
public class ExampleSwitch{
public static void main(String args[]){
byte b=10;
switch(b+1)
{
case 10:
System.out.println("10");
case 100:
System.out.println("100");
case 1000:
System.out.println("1000");
}}}
```

```
OUTPUT:
```

Duplicate case labels are not allowed.

Example 6:

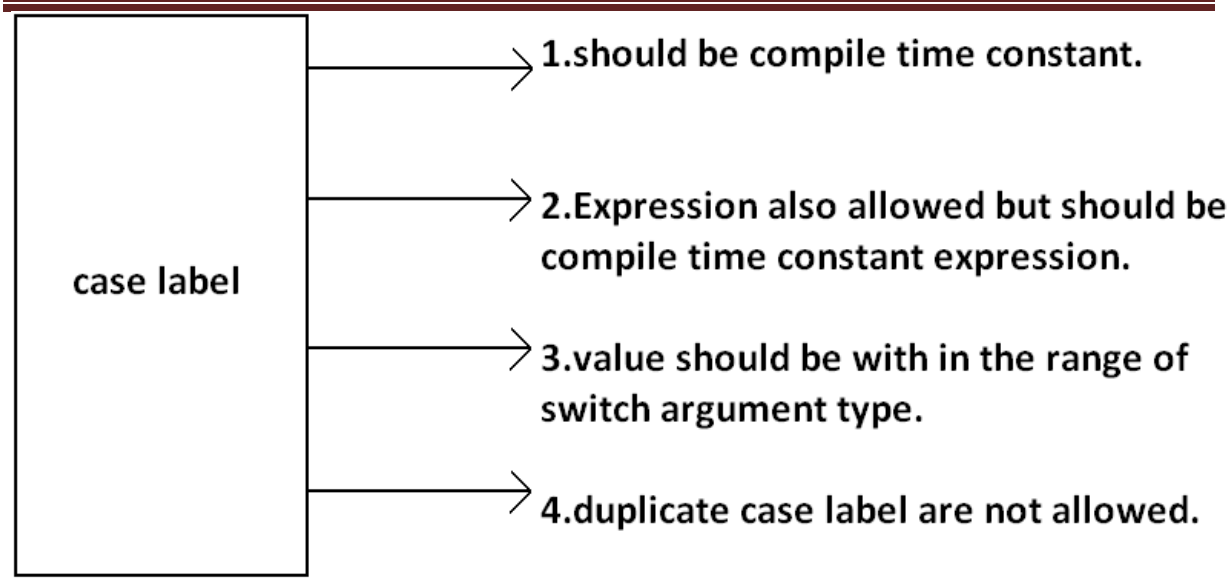
```
public class ExampleSwitch{
public static void main(String args[]){
int x=10;
switch(x)
{
case 97:
System.out.println("97");
case 99:
System.out.println("99");
case 'a':
System.out.println("100");
}}}
```

```
OUTPUT:
Compile time error.
D:\Java>javac ExampleSwitch.java
ExampleSwitch.java:10: duplicate case label
```

```
case 'a':
```


CASE SUMMARY:

Diagram:



FALL-THROUGH INSIDE THE SWITCH:

With in the switch statement if any case is matched from that case onwards all statements will be executed until end of the switch (or) break. This is call "fall-through" inside the switch .

The main advantage of fall-through inside a switch is we can define common action for multiple cases

Example 7:

```
public class ExampleSwitch{
public static void main(String args[]){
int x=0;
switch(x)
{
case 0:
System.out.println("0");
case 1:
System.out.println("1");
break;
case 2:
System.out.println("2");
default:
System.out.println("default");
}}}
OUTPUT:
x=0             x=1             x=2             x=3
0               1               2               default
1                               default
```

DEFAULT CASE:

- **With in the switch we can take the default only once**
- **If no other case matched then only default case will be executed**
- **With in the switch we can take the default any where, but it is convension to take default as last case.**

Example 8:

```
public class ExampleSwitch{
public static void main(String args[]){
int x=0;
switch(x)
{
default:
System.out.println("default");
case 0:
System.out.println("0");
break;
case 1:
System.out.println("1");
case 2:
System.out.println("2");
}}}
OUTPUT:
X=0             x=1             x=2             x=3
0               1               2               default
                2                               0
```

ITERATIVE STATEMENTS:

While loop:

if we don't know the no of iterations in advance then best loop is while loop:

Example 1:

```
while(rs.next())
{
}
```

Example 2:

```
while(e.hasMoreElements())
{
----------
----------
----------
}
```

Example 3:

```
while(itr.hasNext())
{
----------
----------
----------
}
```

The argument to the while statement should be Boolean type. If we are using any other type we will get compile time error.

Example 1:

```
public class ExampleWhile{
public static void main(String args[]){
while(1)
{
System.out.println("hello");
}}}
OUTPUT:
Compile time error.
D:\Java>javac ExampleWhile.java
ExampleWhile.java:3: incompatible types
found   : int
required: boolean
while(1)
```

Curly braces are optional and without curly braces we can take only one statement which should not be declarative statement.

Example 2:

```
public class ExampleWhile{
public static void main(String args[]){
while(true)
System.out.println("hello");
}}
OUTPUT:
Hello (infinite times).
```

Example 3:

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

```
while(true);
}}
OUTPUT:
No output.
```

Example 4:

```
public class ExampleWhile{
public static void main(String args[]){
while(true)
int x=10;
}}
OUTPUT:
Compile time error.
D:\Java>javac ExampleWhile.java
ExampleWhile.java:4: '.class' expected
int x=10;
ExampleWhile.java:4: not a statement
int x=10;
```

Example 5:

```
public class ExampleWhile{
public static void main(String args[]){
while(true)
{
int x=10;
}}}
OUTPUT:
No output.
```

Unreachable statement in while:

Example 6:

```
public class ExampleWhile{
public static void main(String args[]){
while(true)
{
System.out.println("hello");
}
System.out.println("hi");
}}
OUTPUT:
Compile time error.
D:\Java>javac ExampleWhile.java
```

```
ExampleWhile.java:7: unreachable statement
System.out.println("hi");
```

Example 7:

```
public class ExampleWhile{
public static void main(String args[]){
while(false)
{
System.out.println("hello");
}
System.out.println("hi");
}}
OUTPUT:
D:\Java>javac ExampleWhile.java
ExampleWhile.java:4: unreachable statement
{
```

Example 8:

```
public class ExampleWhile{
public static void main(String args[]){
int a=10,b=20;
while(a<b)
{
System.out.println("hello");
}
System.out.println("hi");
}}
OUTPUT:
Hello (infinite times).
```

Example 9:

```
public class ExampleWhile{
public static void main(String args[]){
final int a=10,b=20;
while(a<b)
{
System.out.println("hello");
}
System.out.println("hi");
}}
OUTPUT:
Compile time error.
D:\Java>javac ExampleWhile.java
ExampleWhile.java:8: unreachable statement
System.out.println("hi");
```

Example 10:

```
public class ExampleWhile{
public static void main(String args[]){
final int a=10;
while(a<20)
{
System.out.println("hello");
}
System.out.println("hi");
}}
OUTPUT:
D:\Java>javac ExampleWhile.java
ExampleWhile.java:8: unreachable statement
System.out.println("hi");
```

Note:

- **Every final variable will be replaced with the corresponding value by compiler.**
- **If any operation involves only constants then compiler is responsible to perform that operation.**
- **If any operation involves at least one variable compiler won't perform that operation. At runtime jvm is responsible to perform that operation.**

Example 11:

```
public class ExampleWhile{
public static void main(String args[]){
int a=10;
while(a<20)
{
System.out.println("hello");
}
System.out.println("hi");
}}
OUTPUT:
Hello (infinite times).
```

Do-while:

If we want to execute loop body at least once then we should go for do-while.

Syntax:

```
do
{
----------
----------
----------
}while(b); ——→semicolon is the mandatory.
```


Curly braces are optional.
Without curly braces we can take only one statement between do and while and it should not be declarative statement.

Example 1:

```
public class ExampleDoWhile{
public static void main(String args[]){
do
System.out.println("hello");
while(true);
}}
Output:
Hello (infinite times).
```

Example 2:

```
public class ExampleDoWhile{
public static void main(String args[]){
do;
while(true);
}}
Output:
Compile successful.
```

Example 3:

```
public class ExampleDoWhile{
public static void main(String args[]){
do
int x=10;
while(true);
}}
Output:
D:\Java>javac ExampleDoWhile.java
ExampleDoWhile.java:4: '.class' expected
int x=10;
ExampleDoWhile.java:4: not a statement
int x=10;
ExampleDoWhile.java:4: ')' expected
int x=10;
```

Example 4:

```
public class ExampleDoWhile{
public static void main(String args[]){
do
{
int x=10;
}while(true);
}}
Output:
Compile successful.
```

Example 5:

```
public class ExampleDoWhile{
public static void main(String args[]){
do while(true)
System.out.println("hello");
while(true);
}}
Output:
Hello (infinite times).
```

Rearrange the above Example:

```
public class ExampleDoWhile{
```

```
public static void main(String args[]){
do
  while(true)
       System.out.println("hello");
while(true);
}}
Output:
Hello (infinite times).
```

Example 6:

```
public class ExampleDoWhile{
public static void main(String args[]){
do
while(true);
}}
Output:
Compile time error.
D:\Java>javac ExampleDoWhile.java
ExampleDoWhile.java:4: while expected
while(true);
ExampleDoWhile.java:5: illegal start of expression
}
```

Unreachable statement in do while:

Example 7:

```
public class ExampleDoWhile{
public static void main(String args[]){
do
{
System.out.println("hello");
}
while(true);
System.out.println("hi");
}}
Output:
Compile time error.
D:\Java>javac ExampleDoWhile.java
ExampleDoWhile.java:8: unreachable statement
System.out.println("hi");
```

Example 8:

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

```
do
{
System.out.println("hello");
}
while(false);
System.out.println("hi");
}}
Output:
Hello
Hi
```

Example 9:

```
public class ExampleDoWhile{
public static void main(String args[]){
int a=10,b=20;
do
{
System.out.println("hello");
}
while(a<b);
System.out.println("hi");
}}
Output:
Hello (infinite times).
```

Example 10:

```
public class ExampleDoWhile{
public static void main(String args[]){
int a=10,b=20;
do
{
System.out.println("hello");
}
while(a>b);
System.out.println("hi");
}}
Output:
Hello
Hi
```

Example 11:

```
public class ExampleDoWhile{
public static void main(String args[]){
final int a=10,b=20;
do
{
System.out.println("hello");
}
while(a<b);
System.out.println("hi");
}}
Output:
Compile time error.
D:\Java>javac ExampleDoWhile.java
ExampleDoWhile.java:9: unreachable statement
System.out.println("hi");
```

Example 12:

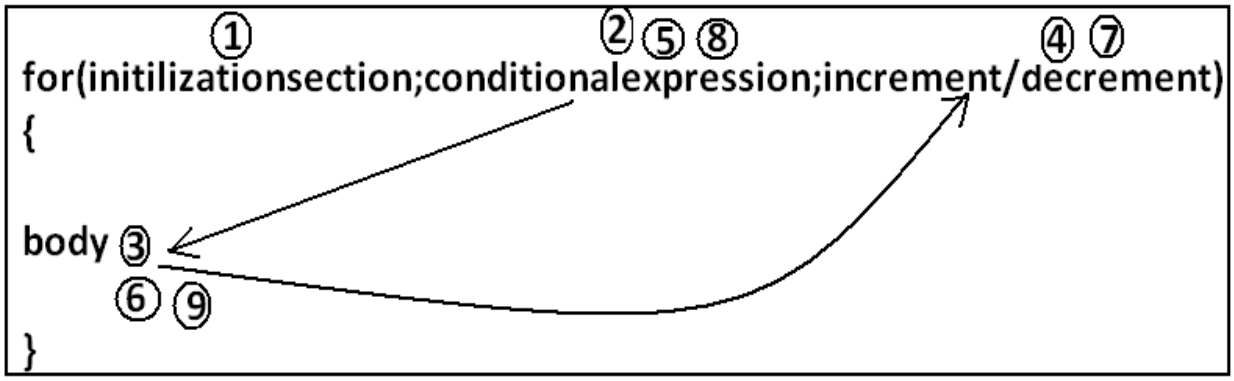
```
public class ExampleDoWhile{
public static void main(String args[]){
final int a=10,b=20;
do
{
```

```
System.out.println("hello");
}
while(a>b);
System.out.println("hi");
}}
Output:
D:\Java>javac ExampleDoWhile.java
D:\Java>java ExampleDoWhile
Hello
Hi
```


For Loop:

This is the most commonly used loop and best suitable if we know the no of iterations in advance.

Syntax:



Curly braces are optional and without curly braces we can take only one statement which should not be declarative statement.

Initilizationsection:

This section will be executed only once.
Here usually we can declare loop variables and we will perform initialization.
We can declare multiple variables but should be of the same type and we can't declare different type of variables.

Example:

```
Int i=0,j=0; valid
Int i=0,Boolean b=true; invalid
Int i=0,int j=0; invalid
```

In initialization section we can take any valid java statement including "s.o.p" also.

Example 1:

```
public class ExampleFor{
public static void main(String args[]){
int i=0;
for(System.out.println("hello u r sleeping");i<3;i++){
System.out.println("no boss, u only sleeping");
}}}
Output:
D:\Java>javac ExampleFor.java
D:\Java>java ExampleFor
Hello u r sleeping
No boss, u only sleeping
No boss, u only sleeping
No boss, u only sleeping
```

Conditional check:

We can take any java expression but should be of the type Boolean.
Conditional expression is optional and if we are not taking any expression compiler will place true.

Increment and decrement section:

Here we can take any java statement including s.o.p also.

Example:

```
public class ExampleFor{
public static void main(String args[]){
int i=0;
for(System.out.println("hello");i<3;System.out.println("hi")){
i++;
}}}
Output:
D:\Java>javac ExampleFor.java
D:\Java>java ExampleFor
Hello
Hi
Hi
Hi
```

All 3 parts of for loop are independent of each other and all optional.

Example:

```
public class ExampleFor{
public static void main(String args[]){
for(;;){
System.out.println("hello");
}}}
```

```
Output:
Hello (infinite times).
```

Curly braces are optional and without curly braces we can take exactly one statement and it should not be declarative statement.

Unreachable statement in for loop:

Example 1:

```
public class ExampleFor{
public static void main(String args[]){
for(int i=0;true;i++){
System.out.println("hello");
}
System.out.println("hi");
}}
Output:
Compile time error.
D:\Java>javac ExampleFor.java
ExampleFor.java:6: unreachable statement
System.out.println("hi");
```

Example 2:

```
public class ExampleFor{
public static void main(String args[]){
for(int i=0;false;i++){
System.out.println("hello");
}
System.out.println("hi");
}}
Output:
Compile time error.
D:\Java>javac ExampleFor.java
ExampleFor.java:3: unreachable statement
for(int i=0;false;i++){
```

Example 3:

```
public class ExampleFor{
public static void main(String args[]){
for(int i=0;;i++){
System.out.println("hello");
}
System.out.println("hi");
}}
Output:
Compile time error.
D:\Java>javac ExampleFor.java
ExampleFor.java:6: unreachable statement
System.out.println("hi");
```

Example 4:

```
public class ExampleFor{
public static void main(String args[]){
int a=10,b=20;
for(int i=0;a<b;i++){
System.out.println("hello");
}
System.out.println("hi");
}}
Output:
```

```
Hello (infinite times).
```

Example 5:

```
public class ExampleFor{
public static void main(String args[]){
final int a=10,b=20;
for(int i=0;a<b;i++){
System.out.println("hello");
}
System.out.println("hi");
}}
Output:
D:\Java>javac ExampleFor.java
ExampleFor.java:7: unreachable statement
System.out.println("hi");
```

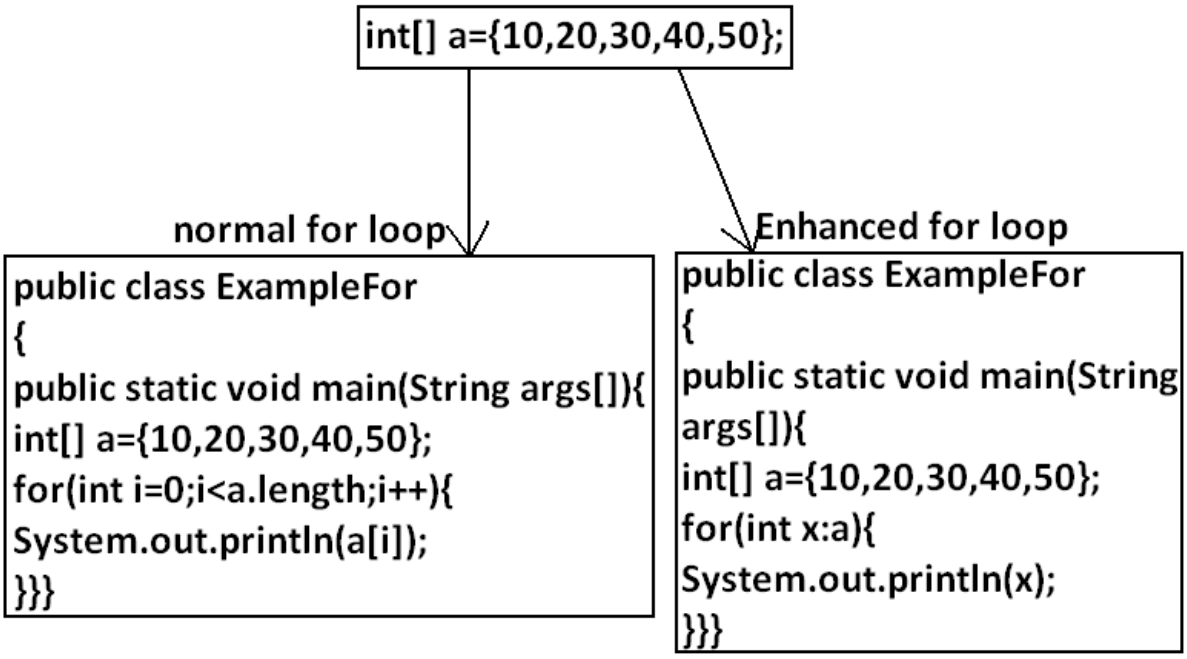
For each:(Enhanced for loop)

- **For each Introduced in 1.5version.**
- **Best suitable to retrieve the elements of arrays and collections.**

Example 1:

Write code to print the elements of single dimensional array by normal for loop and enhanced for loop.

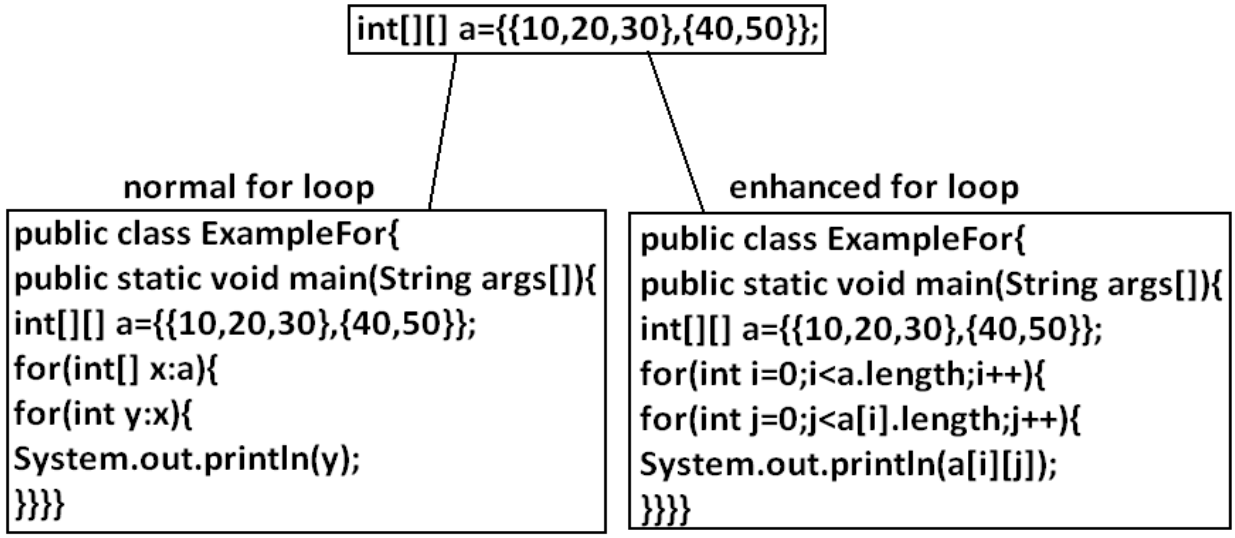
Example:




```
Output:
D:\Java>javac ExampleFor.java
D:\Java>java ExampleFor
10
20
30
40
50
```

Example 2:

Write code to print the elements of 2 dimensional arrays by using normal for loop and enhanced for loop.



Example 3:

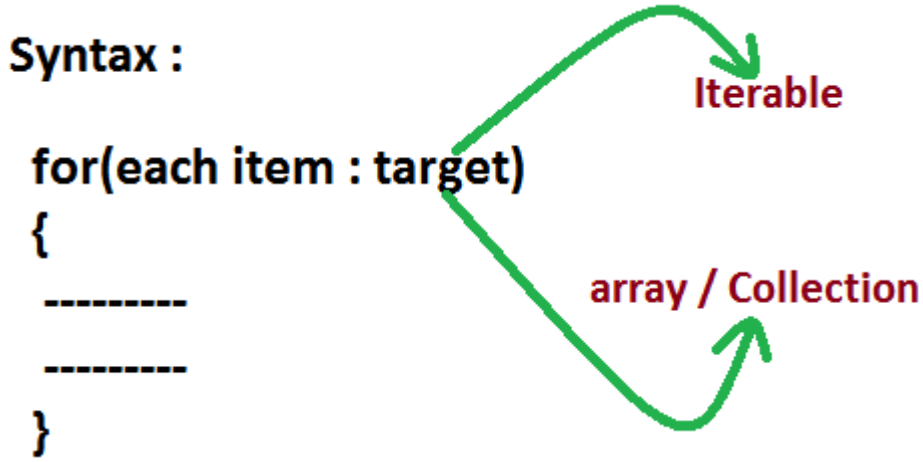
Write equivalent code by For Each loop for the following for loop.

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

```
Output:
D:\Java>javac ExampleFor1.java
D:\Java>java ExampleFor1
Hello
Hello
Hello
Hello
Hello
Hello
Hello
Hello
Hello
Hello
```

- **We can't write equivalent for each loop.**
- **For each loop is the more convenient loop to retrieve the elements of arrays and collections, but its main limitation is it is not a general purpose loop.**
- **By using normal for loop we can print elements either from left to right or from right to left. But using for-each loop we can always print array elements only from left to right.**

Iterator Vs Iterable(1.5v)



- **The target element in for-each loop should be Iterable object.**
- **An object is set to be iterable iff corresponding class implements java.lang.Iterable interface.**
- **Iterable interface introduced in 1.5 version and it's contains only one method iterator().**

Syntax : public Iterator iterator();

Every array class and Collection interface already implements Iterable interface.

Difference between Iterable and Iterator:

Iterable	Iterator
It is related to forEach loop	It is related to Collection
The target element in forEach loop should be Iterable	We can use Iterator to get objects one by one from the collection
Iterator present in java.lang package	Iterator present in java.util package
contains only one method iterator()	contains 3 methods hasNext(), next(), remove()
Introduced in 1.5 version	Introduced in 1.2 version

Transfer statements:

Break statement:

We can use break statement in the following cases.

- **Inside switch to stop fall-through.**
- **Inside loops to break the loop based on some condition.**
- **Inside label blocks to break block execution based on some condition.**

Inside switch :

We can use break statement inside switch to stop fall-through

Example 1:

```
class Test{
public static void main(String args[]){
int x=0;
switch(x)
{
case 0:
    System.out.println("hello");
     break ;
case 1:
    System.out.println("hi");
}

Output:
D:\Java>javac Test.java
D:\Java>java Test
Hello
```

Inside loops :

We can use break statement inside loops to break the loop based on some condition.

Example 2:

```
class Test{
public static void main(String args[]){
for(int i=0; i<10; i++) {
 if(i==5)
  break;
  System.out.println(i);
}
}}
Output:
D:\Java>javac Test.java
D:\Java>java Test
0
1
2
3
4
```

Inside Labeled block :

We can use break statement inside label blocks to break block execution based on some condition.

Example:

```
class Test{
public static void main(String args[]){
int x=10;
l1 : {
  System.out.println("begin");
  if(x==10)
  break l1;
  System.out.println("end");
```

```
 }
 System.out.println("hello");
}
}
Output:
D:\Java>javac Test.java
D:\Java>java Test
begin
hello
```

These are the only places where we can use break statement. If we are using anywhere else we will get compile time error.

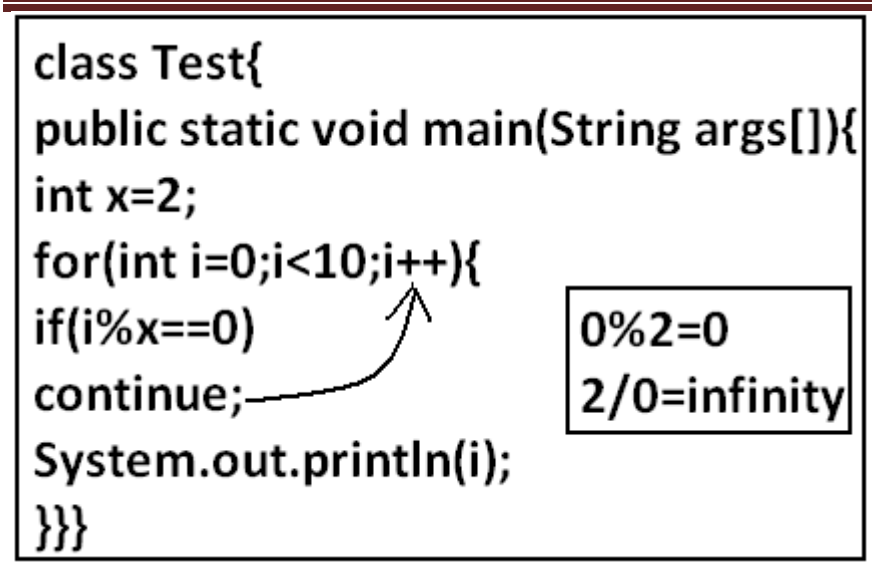
Example:

```
class Test{
public static void main(String args[]){
int x=10;
 if(x==10)
 break;
 System.out.println("hello");
}
}
Output:
Compile time error.
D:\Java>javac Test.java
Test.java:5: break outside switch or loop
break;
```


Continue statement:

We can use continue statement to skip current iteration and continue for the next iteration.

Example:




```
Output:
D:\Java>javac Test.java
D:\Java>java Test
1
3
5
7
9
```

We can use continue only inside loops if we are using anywhere else we will get compile time error saying "continue outside of loop".

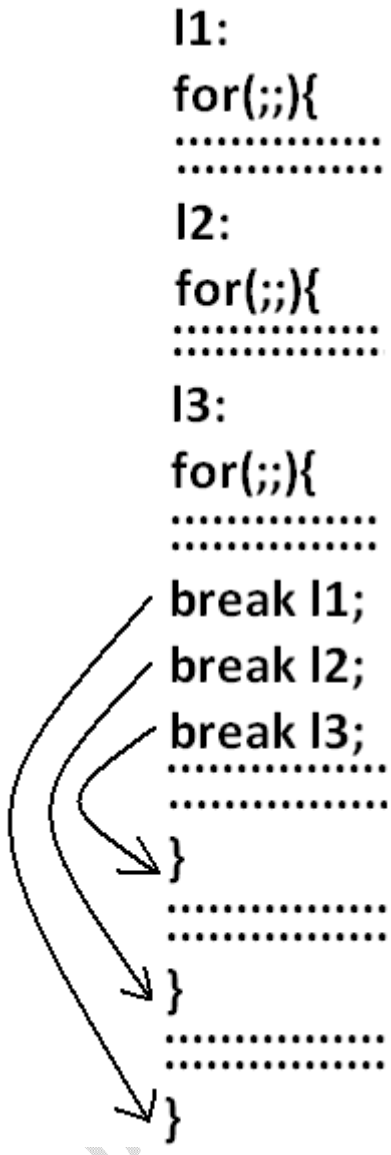
Example:

```
class Test
{
public static void main(String args[]){
int x=10;
if(x==10);
continue;
System.out.println("hello");
}
}
Output:
Compile time error.
D:\Enum>javac Test.java
Test.java:6: continue outside of loop
continue;
```

Labeled break and continue statements:

In the nested loops to break (or) continue a particular loop we should go for labeled break and continue statements.

Syntax:



Example:

```
class Test
{
public static void main(String args[]){
l1:
 for(int i=0;i<3;i++)
 {
   for(int j=0;j<3;j++)
   {
     if(i==j)
       break;
         System.out.println(i+".........."+j);
    }


 }
 }
}
```

```
Break:
1.........0
2.........0
2.........1

Break l1:
No output.

Continue:
0.........1
0.........2
1.........0
1.........2
2.........0
2.........1

Continue l1:
1.........0
2.........0
2.........1
```

Do-while vs continue (The most dangerous combination):

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



```
Output:
1
4
6
8
10
```

Compiler won't check unreachability in the case of if-else it will check only in loops.

Example 1:

```
class Test
{
public static void main(String args[]){
while(true)
{
System.out.println("hello");
}
System.out.println("hi");
}
}
Output:
Compile time error.
D:\Enum>javac Test.java
Test.java:8: unreachable statement
System.out.println("hi");
```

Example 2:

```
class Test
{
public static void main(String args[]){
if(true)
{
System.out.println("hello");
}
else
{
System.out.println("hi");
}}}
Output:
Hello
```


LEARN FROM EXPERTS ...
COMPLETE JAVA
CORE JAVA, ADV. JAVA, ORACLE, STRUTS, HIBERNATE, SPRING, WEB SERVICES,...
COMPLETE .NET
C#.NET, ASP.NET, SQL SERVER, MVC 5 & WCF
TESTING TOOLS
MANUAL + SELENIUM
ORACLE
D2K
MSBI
SHARE POINT
HADOOP
ANDROID
C, C++, DS, UNIX
CRT & APTITUDE TRAINING
AN ISO 9001:2008 CERTIFIED
DURGA
Software Solutions®
202, 2nd Floor, HUDA Maitrivanam,
Ameerpet, Hyd. Ph: 040-64512786,
9246212143, 8096969696
www.durgasoft.com

CORE JAVA

With

SCJP / OCJP

Study Material

Chapter 4: Declaration & Access Modifiers

DURGA M.Tech

(Sun certified & Realtime Expert)

Ex. IBM Employee

Trained Lakhs of Students
for last 14 years across INDIA

Declaration and Access Modifiers

Agenda

1. **Java source file structure**
    - **Import statement**
    - **Types of Import Statements**
        - **Explicit class import**
        - **Implicit class import**
    - **Difference between C language #include and java language import ?**
    - **1.5 versions new features**
    - **Static import**
        - **Without static import**
        - **With static import**
    - **Explain about System.out.println statement ?**
    - **What is the difference between general import and static import ?**
    - **Package statement**
        - **How to compile package Program**
        - **How to execute package Program**
    - **Java source file structure**
2. **Class Modifiers**
    - **Only applicable modifiers for Top Level classes**
    - **What is the difference between access specifier and access modifier ?**
    - **Public Classes**
    - **Default Classes**
    - **Final Modifier**
        - **Final Methods**
        - **Final Class**
    - **Abstract Modifier**
        - **Abstract Methods**
        - **Abstract class**
    - **The following are the various illegal combinations for methods**
    - **What is the difference between abstract class and abstract method ?**
    - **What is the difference between final and abstract ?**
    - **Strictfp**
    - **What is the difference between abstract and strictfp ?**
3. **Member modifiers**
    - **Public members**
    - **Default member**
    - **Private members**
    - **Protected members**
    - **Compression of private, default, protected and public**
    - **Final variables**
        - **Final instance variables**
            - **At the time of declaration**
            - **Inside instance block**

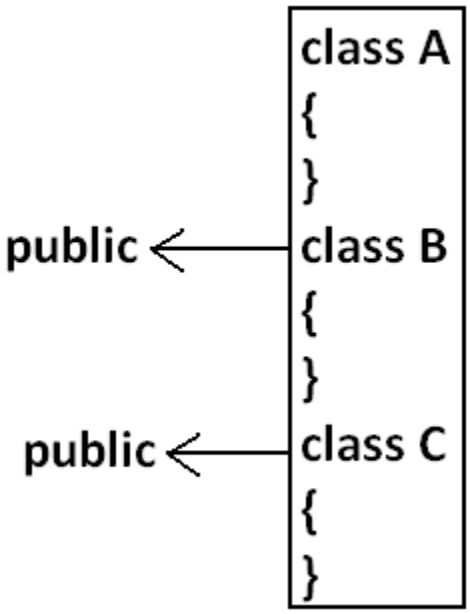
        - Inside constructor
    - Final static variables
        - At the time of declaration
        - Inside static block
    - Final local variables
- Formal parameters
- Static modifier
- Native modifier
    - Pseudo code
- Synchronized
- Transient modifier
- Volatile modifier
- Summary of modifier

4. Interfaces
    - Interface declarations and implementations
    - Extends vs implements
    - Interface methods
    - Interface variables
    - Interface naming conflicts
        - Method naming conflicts
        - Variable naming conflicts
    - Marker interface
    - Adapter class
    - Interface vs abstract class vs concrete class
    - Difference between interface and abstract class?
    - Conclusions

Java source file structure:

- **A java Program can contain any no. Of classes but at most one class can be declared as public. "If there is a public class the name of the Program and name of the public class must be matched otherwise we will get compile time error".**
- **If there is no public class then any name we gives for java source file.**

Example:



Case 1:

If there is no public class then we can use any name for java source file there are no restrictions.


```
Example:
A.java
B.java
C.java
Ashok.java
```

case 2:

If class B declared as public then the name of the Program should be B.java otherwise we will get compile time error saying "class B is public, should be declared in a file named B.java".

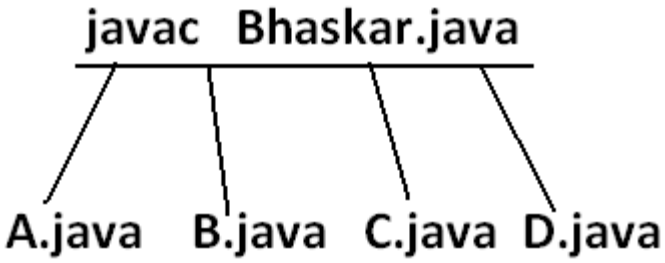
Case 3:

- **If both B and C classes are declared as public and name of the file is B.java then we will get compile time error saying "class C is public, should be declared in a file named C.java".**
- **It is highly recommended to take only one class for source file and name of the Program (file) must be same as class name. This approach improves readability and understandability of the code.**

Example:

```
class A
{
public static void main(String args[]){
System.out.println("A class main method is executed");
}
}
class B
{
public static void main(String args[]){
System.out.println("B class main method is executed");
}
}
class C
{
public static void main(String args[]){
System.out.println("C class main method is executed");
}
}
class D
{
}
```

```
Output:
```




```
D:\Java>java A
A class main method is executed
D:\Java>java B
B class main method is executed
D:\Java>java C
C class main method is executed
D:\Java>java D
Exception in thread "main" java.lang.NoSuchMethodError: main
D:\Java>java Ashok
Exception in thread "main" java.lang.NoClassDefFoundError: Ashok
```

- **We can compile a java Program but not java class in that Program for every class one dot class file will be created.**
- **We can run a java class but not java source file whenever we are trying to run a class the corresponding class main method will be executed.**
- **If the class won't contain main method then we will get runtime exception saying "NoSuchMethodError: main".**
- **If we are trying to execute a java class and if the corresponding .class file is not available then we will get runtime execution saying "NoClassDefFoundError: Ashok".**

Import statement:

```
class Test{
public static void main(String args[]){
ArrayList l=new ArrayList();
}
}
Output:
Compile time error.
D:\Java>javac Test.java
Test.java:3: cannot find symbol
symbol  : class ArrayList
location: class Test

ArrayList l=new ArrayList();
```

- **We can resolve this problem by using fully qualified name "java.util.ArrayList l=new java.util.ArrayList();". But problem with using fully qualified name every time is it increases length of the code and reduces readability.**
- **We can resolve this problem by using import statements.**

Example:

```
import java.util.ArrayList;
class Test{
public static void main(String args[]){
ArrayList l=new ArrayList();
}
}
Output:
D:\Java>javac Test.java
```

Hence whenever we are using import statement it is not require to use fully qualified names we can use short names directly. This approach decreases length of the code and improves readability.

Case 1: Types of Import Statements:

There are 2 types of import statements.

1) Explicit class import
2) Implicit class import.

Explicit class import:

Example: **Import java.util.ArrayList**

- **This type of import is highly recommended to use because it improves readability of the code.**
- **Best suitable for Hi-Tech city where readability is important.**

Implicit class import:

Example: **import java.util.*;**

- **It is never recommended to use because it reduces readability of the code.**
- **Best suitable for Ameerpet where typing is important.**

Case 2:

Which of the following import statements are meaningful ?

import java.util; ✗
import java.util.ArrayList.*; ✗
import java.util.*; ✓
import java.util.ArrayList; ✓

Case 3:

consider the following code.

```
class MyArrayList extends java.util.ArrayList
{
}
```

- **The code compiles fine even though we are not using import statements because we used fully qualified name.**
- **Whenever we are using fully qualified name it is not required to use import statement. Similarly whenever we are using import statements it is not require to use fully qualified name.**

Case 4:

Example:

```
import java.util.*;
import java.sql.*;
class Test
```

```
{
public static void main(String args[])
{
Date d=new Date();
}}
Output:
Compile time error.
D:\Java>javac Test.java
Test.java:7: reference to Date is ambiguous,
 both class java.sql.Date in java.sql and class java.util.Date in java.util
match
```

Date d=new Date();

Note: Even in the List case also we may get the same ambiguity problem because it is available in both util and awt packages.

Case 5:

While resolving class names compiler will always gives the importance in the following order.

1. **Explicit class import**
2. **Classes present in current working directory.**
3. **Implicit class import.**

Example:

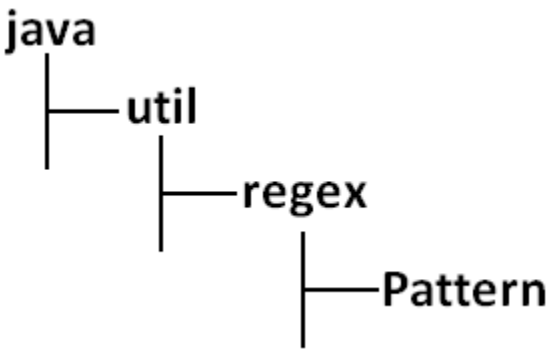
```
import java.util.Date;
import java.sql.*;
class Test
{
public static void main(String args[]){
Date d=new Date();
}}
```

The code compiles fine and in this case util package Date will be considered.

Case 6:

Whenever we are importing a package all classes and interfaces present in that package are by default available but not sub package classes.

Example:



To use pattern class in our Program directly which import statement is required ?

1.import java.*; ✗
2.import java.util.*; ✗
3.import java.util.regex.*; ✓
4.import java.util.regex.Pattern; ✓

Case7:

In any java Program the following 2 packages are not require to import because these are available by default to every java Program.

1. java.lang package
2. default package(current working directory)

Case 8:

"Import statement is totally compile time concept" if more no of imports are there then more will be the compile time but there is "no change in execution time".

Difference between C language #include and java language import ?

#include	import
It can be used in C & C++	It can be used in Java
At compile time only compiler copy the code from standard library and placed in current program.	At runtime JVM will execute the corresponding standard library and use it's result in current program.
It is static inclusion	It is dynamic inclusion
wastage of memory	No wastage of memory
Ex : <jsp:@ file="">	Ex : <jsp:include >

- **In the case of C language #include all the header files will be loaded at the time of include statement hence it follows static loading.**
- **But in java import statement no ".class" will be loaded at the time of import statements in the next lines of the code whenever we are using a particular class then only corresponding ".class" file will be loaded. Hence it follows "dynamic loading" or "load-on -demand" or "load-on-fly".**

1.5 versions new features :

1. **For-Each**
2. **Var-arg**
3. **Queue**
4. **Generics**
5. **Auto boxing and Auto unboxing**
6. **Co-varient return types**
7. **Annotations**
8. **Enum**
9. **Static import**
10. **String builder**

Static import:

This concept introduced in 1.5 versions. According to sun static import improves readability of the code but according to worldwide Programming exports (like us) static imports creates confusion and reduces readability of the code. Hence if there is no specific requirement never recommended to use a static import.

Usually we can access static members by using class name but whenever we are using static import it is not require to use class name we can access directly.

Without static import:

```
class Test
{
public static void main(String args[]){
System.out.println(Math.sqrt(4));
System.out.println(Math.max(10,20));
System.out.println(Math.random());
}}
Output:
D:\Java>javac Test.java
D:\Java>java Test
2.0
20
0.841306154315576
```

With static import:

```
import static java.lang.Math.sqrt;
import static java.lang.Math.*;
```

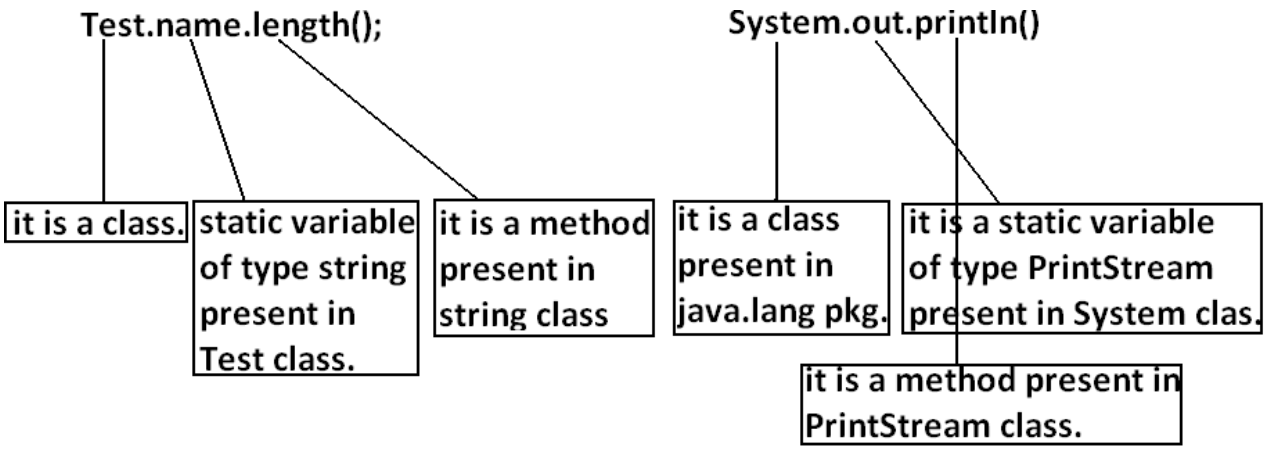
```
class Test
{
public static void main(String args[]){
System.out.println(sqrt(4));
System.out.println(max(10,20));
System.out.println(random());
}}
Output:
D:\Java>javac Test.java
D:\Java>java Test
2.0
20
0.4302853847363891
```

Explain about System.out.println statement ?

Example 1 and Example 2:

```
1)
class Test
{
static String name="bhaskar";
}
```

```
2)
import java.io.*;
class System
{
static PrintStream out;
}
```



Example 3:

```
import static java.lang.System.out;
class Test
{
public static void main(String args[]){
out.println("hello");
out.println("hi");
}}
Output:
D:\Java>javac Test.java
D:\Java>java Test
hello
hi
```

Example 4:

```
import static java.lang.Integer.*;
import static java.lang.Byte.*;
class Test
{
public static void main(String args[]){
System.out.println(MAX_VALUE);
}}
Output:
Compile time error.
D:\Java>javac Test.java
Test.java:6: reference to MAX_VALUE is ambiguous,
 both variable MAX_VALUE in java.lang.Integer and variable MAX_VALUE in
java.lang.Byte match
System.out.println(MAX_VALUE);
```

Note: Two packages contain a class or interface with the same is very rare hence ambiguity problem is very rare in normal import.

But 2 classes or interfaces can contain a method or variable with the same name is very common hence ambiguity problem is also very common in static import.

While resolving static members compiler will give the precedence in the following order.

1. **Current class static members**

2. **Explicit static import**
3. **implict static import.**

Example:

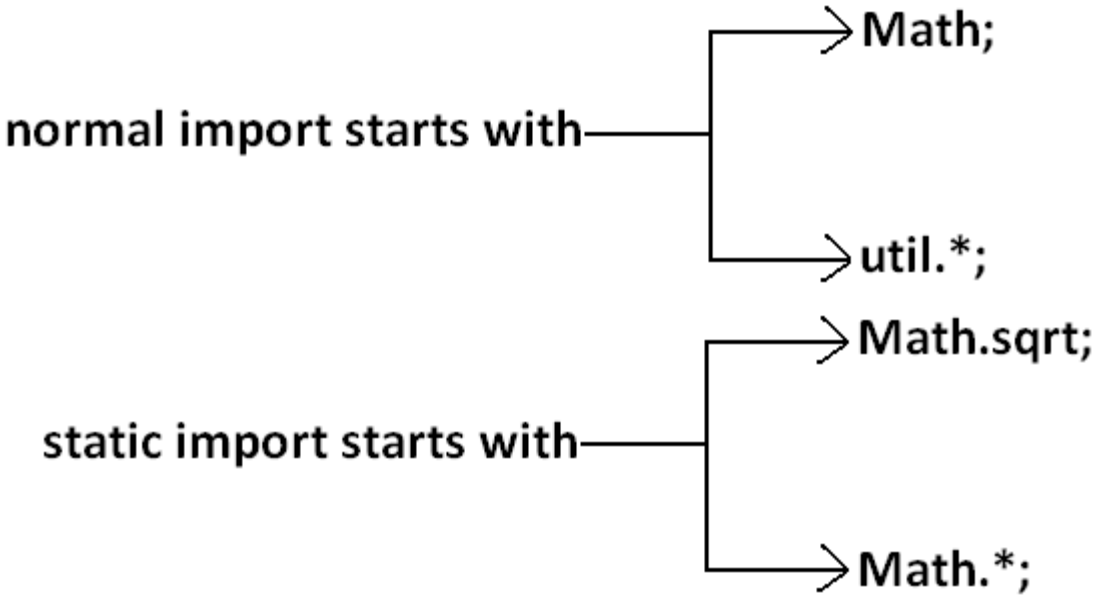
```
//import static java.lang.Integer.MAX_VALUE;------>line2
import static java.lang.Byte.*;
class Test
{
//static int MAX_VALUE=999;------>line1
public static void main(String args[])throws Exception{
System.out.println(MAX_VALUE);
}
}
```


- **If we comet line one then we will get Integer class MAX_VALUE 2147483647.**
- **If we comet lines one and two then Byte class MAX_VALUE will be considered 127.**

Which of the following import statements are valid ?

1.import java.lang.Math.*; ×
2.import static java.lang.Math.*; ✓
3.import java.lang.Math; ✓
4.import static java.lang.Math; ×
5.import static java.lang.Math.sqrt.*; ×
6.import java.lang.Math.sqrt; ×
7.import static java.lang.Math.sqrt(); ×
8.import static java.lang.Math.sqrt; ✓

Diagram:



DURGASOFT, # 202,2nd Floor,HUDA Maitrivanam,Ameerpet, Hyderabad - 500038,
☎ 040 – 64 51 27 86, 80 96 96 96 96, 9246212143 | www.durgasoft.com

Usage of static import reduces readability and creates confusion hence if there is no specific requirement never recommended to use static import.

What is the difference between general import and static import ?

- **We can use normal imports to import classes and interfaces of a package. whenever we are using normal import we can access class and interfaces directly by their short name it is not require to use fully qualified names.**
- **We can use static import to import static members of a particular class. whenever we are using static import it is not require to use class name we can access static members directly.**

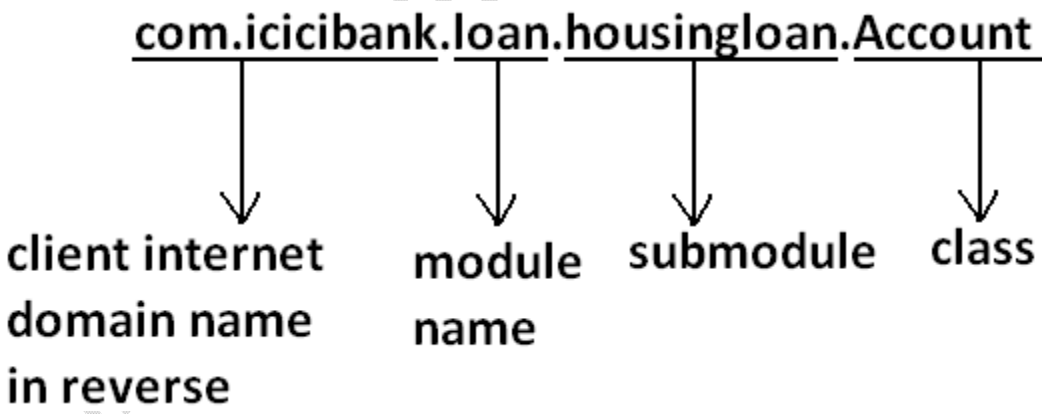
Package statement:

It is an encapsulation mechanism to group related classes and interfaces into a single module.

The main objectives of packages are:

- **To resolve name confects.**
- **To improve modularity of the application.**
- **To provide security.**
- **There is one universally accepted naming conversion for packages that is to use internet domain name in reverse.**

Example:



How to compile package Program:

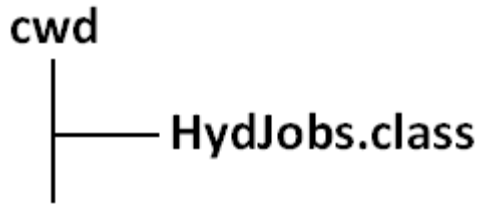
Example:

```
package com.durgajobs.itjobs;
class HydJobs
{
public static void main(String args[]){
System.out.println("package demo");
```

```
}
}
```

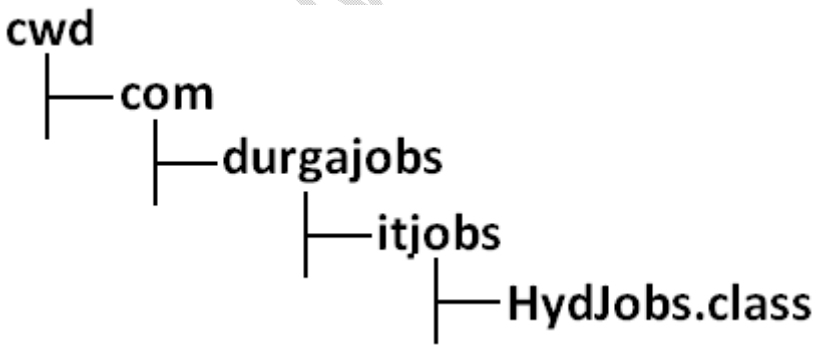
Javac HydJobs.java generated class file will be placed in current working directory.

Diagram:



- **Javac -d . HydJobs.java**
- **-d means destination to place generated class files "." means current working directory.**
- **Generated class file will be placed into corresponding package structure.**

Diagram:

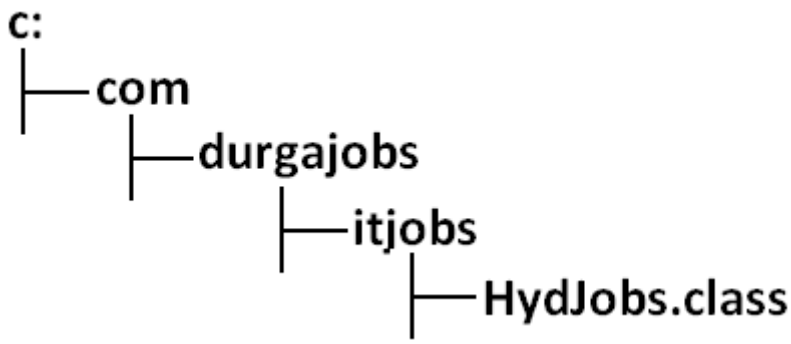


- **If the specified package structure is not already available then this command itself will create the required package structure.**
- **As the destination we can use any valid directory.**
- **If the specified destination is not available then we will get compile time error.**

Example:

D:\Java>javac -d c: HydJobs.java

Diagram:



If the specified destination is not available then we will get compile time error.

Example:

D:\Java>javac -d z: HydJobs.java
If Z: is not available then we will get compile time error.

How to execute package Program:

D:\Java>java com.durgajobs.itjobs.HydJobs

At the time of execution compulsory we should provide fully qualified name.

Conclusion 1:

In any java Program there should be at most one package statement that is if we are taking more than one package statement we will get compile time error.

Example:

```
package pack1;
package pack2;
class A
{
}
Output:
Compile time error.
D:\Java>javac A.java
A.java:2: class, interface, or enum expected
package pack2;
```

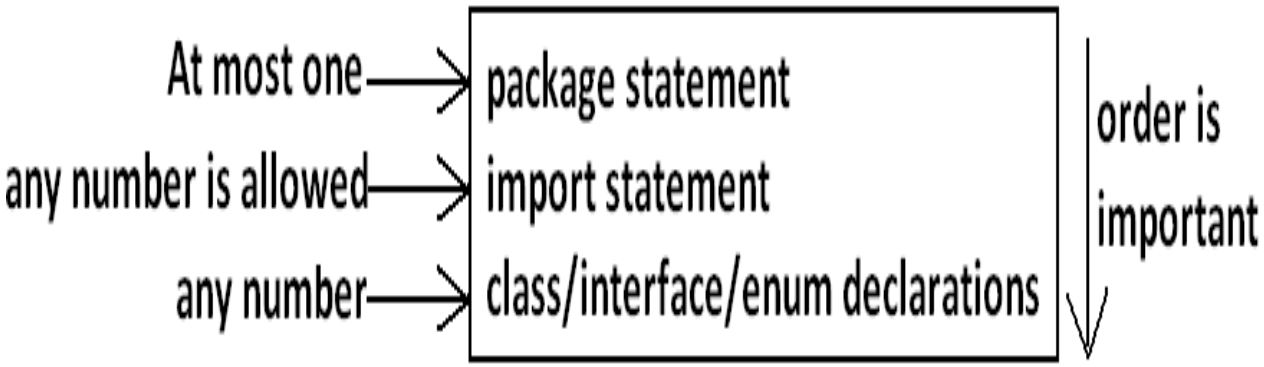
Conclusion 2:

In any java Program the 1st non comment statement should be package statement [if it is available] otherwise we will get compile time error.

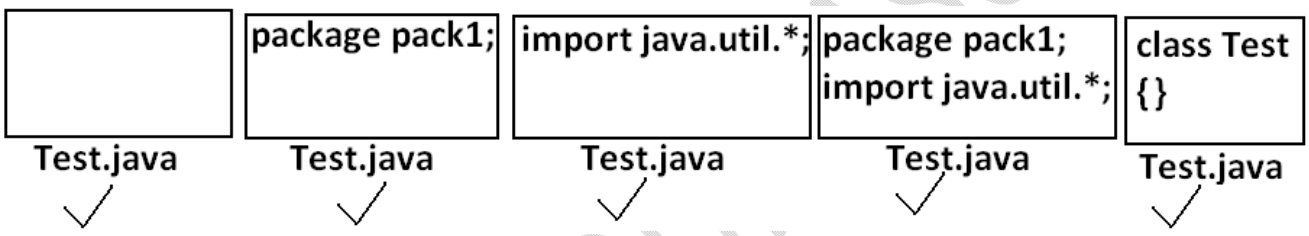
Example:

```
import java.util.*;
package pack1;
class A
{
}
Output:
Compile time error.
D:\Java>javac A.java
A.java:2: class, interface, or enum expected
package pack1;
```


Java source file structure:



All the following are valid java Programs.



Note: **An empty source file is a valid java Program.**

Class Modifiers

Whenever we are writing our own classes compulsory we have to provide some information about our class to the jvm.

Like

1. **Whether this class can be accessible from anywhere or not.**
2. **Whether child class creation is possible or not.**
3. **Whether object creation is possible or not etc.**

We can specify this information by using the corresponding modifiers.

The only applicable modifiers for Top Level classes are:

1. **Public**
2. **Default**
3. **Final**
4. **Abstract**
5. **Strictfp**

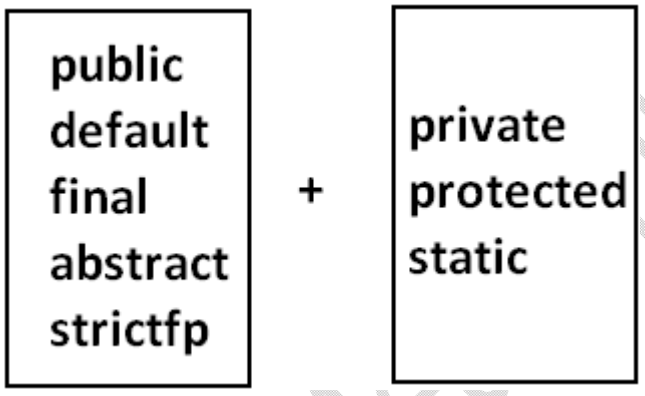
If we are using any other modifier we will get compile time error.

Example:

```
private class Test
{
public static void main(String args[]){
int i=0;
for(int j=0;j<3;j++)
{
i=i+j;
}
System.out.println(i);
}}
OUTPUT:
Compile time error.
D:\Java>javac Test.java
Test.java:1: modifier private not allowed here
private class Test
```

But For the inner classes the following modifiers are allowed.

Diagram:



What is the difference between access specifier and access modifier ?

- **In old languages 'C' (or) 'C++' public, private, protected, default are considered as access specifiers and all the remaining are considered as access modifiers.**
- **But in java there is no such type of division all are considered as access modifiers.**

Public Classes:

If a class declared as public then we can access that class from anywhere. With in the package or outside the package.

Example:

Program1:

```
package pack1;
public class Test
{
public void methodOne(){
System.out.println("test class methodone is executed");
}}
Compile the above Program:
D:\Java>javac -d . Test.java
```

Program2:

```
package pack2;
import pack1.Test;
class Test1
{
public static void main(String args[]){
Test t=new Test();
t.methodOne();
}}
OUTPUT:
D:\Java>javac -d . Test1.java
D:\Java>java pack2.Test1
Test class methodone is executed.
```

If class Test is not public then while compiling Test1 class we will get compile time error saying pack1.Test is not public in pack1; cannot be accessed from outside package.

Default Classes:

If a class declared as the default then we can access that class only within the current package hence default access is also known as "package level access".

Example:

Program 1:

```
package pack1;
class Test
{
public void methodOne(){
System.out.println("test class methodone is executed");
}}
```

Program 2:

```
package pack1;
import pack1.Test;
class Test1
{
public static void main(String args[]){
Test t=new Test();
t.methodOne();
}}
OUTPUT:
D:\Java>javac -d . Test.java
D:\Java>javac -d . Test1.java
D:\Java>java pack1.Test1
Test class methodone is executed
```

Final Modifier:

Final is the modifier applicable for classes, methods and variables.

Final Methods:

- **Whatever the methods parent has by default available to the child.**
- **If the child is not allowed to override any method, that method we have to declare with final in parent class. That is final methods cannot overridden.**

Example:

Program 1:

```
class Parent
{
public void property(){
System.out.println("cash+gold+land");
}
public final void marriage(){
System.out.println("subbalakshmi");
}}
```

Program 2:

```
class child extends Parent
{
public void marriage(){
System.out.println("Thamanna");
}}
OUTPUT:
Compile time error.
D:\Java>javac Parent.java
D:\Java>javac child.java
child.java:3: marriage() in child cannot override marriage() in Parent;
overridden method is final
public void marriage(){
```

Final Class:

If a class declared as the final then we cann't creates the child class that is inheritance concept is not applicable for final classes.

Example:

Program 1:

```
final class Parent
{
}
```

Program 2:

```
class child extends Parent
{
}
OUTPUT:
Compile time error.
D:\Java>javac Parent.java
D:\Java>javac child.java
child.java:1: cannot inherit from final Parent
class child extends Parent
```

Note: Every method present inside a final class is always final by default whether we are declaring or not. But every variable present inside a final class need not be final.

Example:

```
final class parent
{
static int x=10;
static
{
x=999;
}}
```

The main advantage of final keyword is we can achieve security.
Whereas the main disadvantage is we are missing the key benefits of oops: polymorsim (because of final methods), inheritance (because of final classes) hence if there is no specific requirement never recommended to use final keyboard.

Abstract Modifier:

Abstract is the modifier applicable only for methods and classes but not for variables.

Abstract Methods:

Even though we don't have implementation still we can declare a method with abstract modifier.
That is abstract methods have only declaration but not implementation.
Hence abstract method declaration should compulsory ends with semicolon.

Example:

public abstract void methodOne(); ⟶ valid

public abstract void methodOne(){} ⟶ invalid

Child classes are responsible to provide implementation for parent class abstract methods.

Example:

Program:

```
abstract class vehicle
{
public abstract int getNoOfWheels();
}
```

vehicle.java

```
class Bus extends vehicle
{
public int getNoOfWheels(){
return 7;
}
}
```

Bus.java

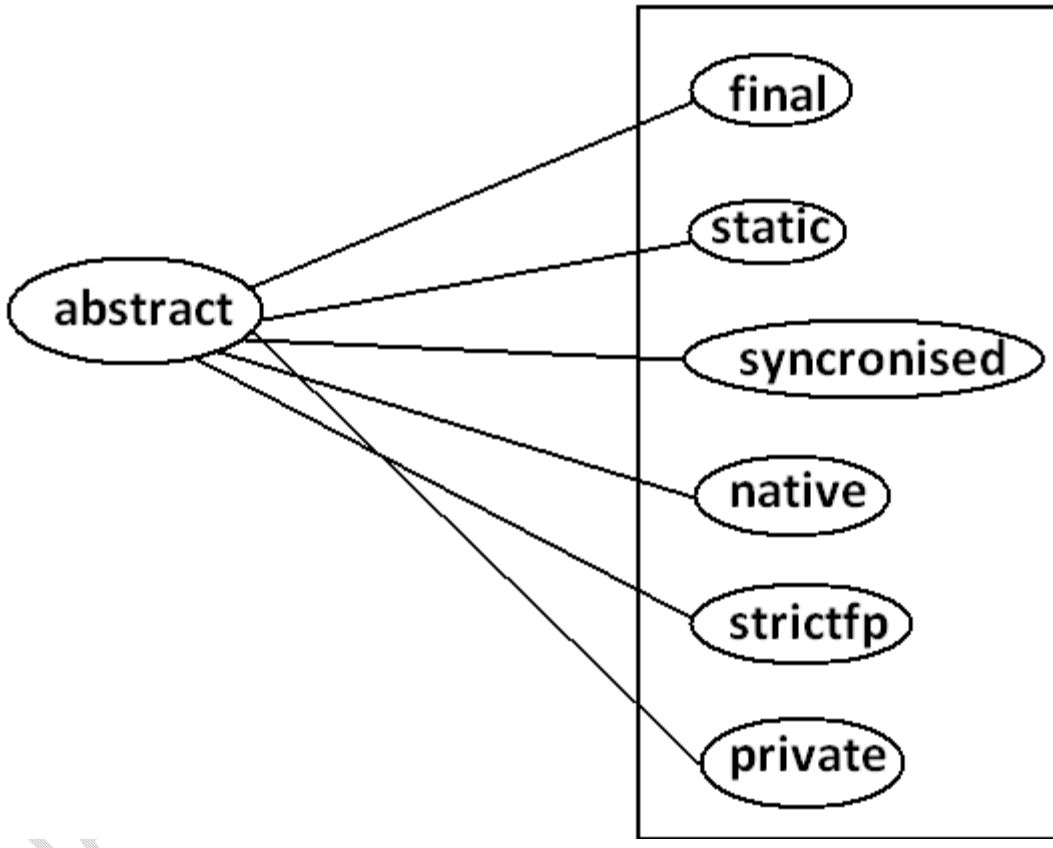
```
class Auto extends vehicle
{
public int getNoOfWheels(){
return 3;
}
}
```

Auto.java

- **The main advantage of abstract methods is , by declaring abstract method in parent class we can provide guide lines to the child class such that which methods they should compulsory implement.**
- **Abstract method never talks about implementation whereas if any modifier talks about implementation then the modifier will be enemy to abstract and that is always illegal combination for methods.**

The following are the various illegal combinations for methods.

Diagram:



All the 6 combinations are illegal.

Abstract class:

For any java class if we are not allow to create an object such type of class we have to declare with abstract modifier that is for abstract class instantiation is not possible.

Example:

```
abstract class Test
{
public static void main(String args[]){
Test t=new Test();
}}
Output:
Compile time error.
D:\Java>javac Test.java
Test.java:4: Test is abstract; cannot be instantiated
Test t=new Test();
```

What is the difference between abstract class and abstract method ?

- **If a class contain at least on abstract method then compulsory the corresponding class should be declare with abstract modifier. Because implementation is not complete and hence we can't create object of that class.**
- **Even though class doesn't contain any abstract methods still we can declare the class as abstract that is an abstract class can contain zero no of abstract methods also.**

Example1: HttpServlet class is abstract but it doesn't contain any abstract method.
Example2: Every adapter class is abstract but it doesn't contain any abstract method.

Example1:

```
class Parent
{
public void methodOne();
}
Output:
Compile time error.
D:\Java>javac Parent.java
Parent.java:3: missing method body, or declare abstract
public void methodOne();
```

Example2:

```
class Parent
{
public abstract void methodOne(){}
}
Output:
Compile time error.
Parent.java:3: abstract methods cannot have a body
public abstract void methodOne(){}
```

Example3:

```
class Parent
{
public abstract void methodOne();
}
Output:
Compile time error.
D:\Java>javac Parent.java
```

```
Parent.java:1: Parent is not abstract and does not
 override abstract method methodOne() in Parent
class Parent
```

If a class extends any abstract class then compulsory we should provide implementation for every abstract method of the parent class otherwise we have to declare child class as abstract.

Example:

```
abstract class Parent
{
public abstract void methodOne();
public abstract void methodTwo();
}
class child extends Parent
{
public void methodOne(){}
}
Output:
Compile time error.
D:\Java>javac Parent.java
Parent.java:6: child is not abstract and does not
 override abstract method methodTwo() in Parent
class child extends Parent
```

If we declare class child as abstract then the code compiles fine but child of child is responsible to provide implementation for methodTwo().

What is the difference between final and abstract ?

- **For abstract methods compulsory we should override in the child class to provide implementation. Whereas for final methods we can't override hence abstract final combination is illegal for methods.**
- **For abstract classes we should compulsory create child class to provide implementation whereas for final class we can't create child class. Hence final abstract combination is illegal for classes.**

- **Final class cannot contain abstract methods whereas abstract class can contain final method.**

Example:

```
final class A
{
public abstract void
methodOne();
}
```

invalid

```
abstract class A
{
public final void methodOne(){
}
}
```

valid

Note:

Usage of abstract methods, abstract classes and interfaces is always good Programming practice.

Strictfp:

- **strictfp is the modifier applicable for methods and classes but not for variables.**
- **Strictfp modifier introduced in 1.2 versions.**
- **Usually the result of floating point of arithmetic is varing from platform to platform , to overcome this problem we should use strictfp modifier.**
- **If a method declare as the Strictfp then all the floating point calculations in that method has to follow IEEE754 standard, So that we will get platform independent results.**

Example:

```
System.out.println(10.0/3);
```

P4	P3	IEEE754
3.33333333333333	3.333333	3.333

If a class declares as the Strictfp then every concrete method(which has body) of that class has to follow IEEE754 standard for floating point arithmetic, so we will get platform independent results.

What is the difference between abstract and strictfp ?

- **Strictfp method talks about implementation where as abstract method never talks about implementation hence abstract, strictfp combination is illegal for methods.**
- **But we can declare a class with abstract and strictfp modifier simultaneously. That is abstract strictfp combination is legal for classes but illegal for methods.**

Example:

public abstract strictfp void methodOne(); (invalid)

abstract strictfp class Test (valid)

{

}

Member modifiers:

Public members:

If a member declared as the public then we can access that member from anywhere "but the corresponding class must be visible" hence before checking member visibility we have to check class visibility.

Example:

Program 1:

```
package pack1;
class A
{
public void methodOne(){
System.out.println("a class method");
}}
D:\Java>javac -d . A.java
```

Program 2:

```
package pack2;
import pack1.A;
class B
{
public static void main(String args[]){
A a=new A();
a.methodOne();
```

```
}}
Output:
Compile time error.
D:\Java>javac -d . B.java
B.java:2: pack1.A is not public in pack1;
   cannot be accessed from outside package
import pack1.A;
```

In the above Program even though methodOne() method is public we can't access from class B because the corresponding class A is not public that is both classes and methods are public then only we can access.

Default member:

If a member declared as the default then we can access that member only within the current package hence default member is also known as package level access.

Example 1:

Program 1:

```
package pack1;
class A
{
void methodOne(){
System.out.println("methodOne is executed");
}}
```

Program 2:

```
package pack1;
import pack1.A;
class B
{
public static void main(String args[]){
A a=new A();
a.methodOne();
}}
Output:
D:\Java>javac -d . A.java
D:\Java>javac -d . B.java
D:\Java>java pack1.B
methodOne is executed
```

Example 2:

Program 1:

```
package pack1;
class A
{
void methodOne(){
System.out.println("methodOne is executed");
}}
```

Program 2:

```
package pack2;
import pack1.A;
class B
{
```

```
public static void main(String args[]){
A a=new A();
a.methodOne();
}}
Output:
Compile time error.
D:\Java>javac -d . A.java
D:\Java>javac -d . B.java
B.java:2: pack1.A is not public in pack1; cannot be accessed from outside
package
import pack1.A;
```

Private members:

- **If a member declared as the private then we can access that member only with in the current class.**
- **Private methods are not visible in child classes where as abstract methods should be visible in child classes to provide implementation hence private, abstract combination is illegal for methods.**

Protected members:

- **If a member declared as the protected then we can access that member within the current package anywhere but outside package only in child classes. Protected=default+kids.**
- **We can access protected members within the current package anywhere either by child reference or by parent reference**
- **But from outside package we can access protected members only in child classes and should be by child reference only that is we can't use parent reference to call protected members from outside package.**

Example:

Program 1:

```
package pack1;
public class A
{
protected void methodOne(){
System.out.println("methodOne is executed");
}}
```

Program 2:

```
package pack1;
class B extends A
{
public static void main(String args[]){
A a=new A();
a.methodOne();
B b=new B();
b.methodOne();
A a1=new B();
a1.methodOne();
}}
Output:
D:\Java>javac -d . A.java
D:\Java>javac -d . B.java
D:\Java>java pack1.B
methodOne is executed
methodOne is executed
methodOne is executed
```

Example 2:

```
package pack2;
import pack1.A;
public class C extends A
{
public static void main(String args[]){
A a=new A();
a.methodOne();
C c=new C();
c.methodOne();
A a1=new B();
a1.methodOne();
}
}
```

output:

```
compile time error.
D:\Java>javac -d . C.java
C.java:7: methodOne() has protected access in
pack1.A
a.methodOne();
```

Compression of private, default, protected and public:

visibility	private	default	protected	public
1)With in the same class	✓	✓	✓	✓
2)From child class of same package	✗	✓	✓	✓
3)From non-child class of same package	✗	✓	✓	✓
4)From child class of outside package	✗	✗	✓ but we should use child reference only	✓
5)From non-child class of outside package	✗	✗	✗	✓

- **The least accessible modifier is private.**
- **The most accessible modifier is public.**

Private<default<protected<public

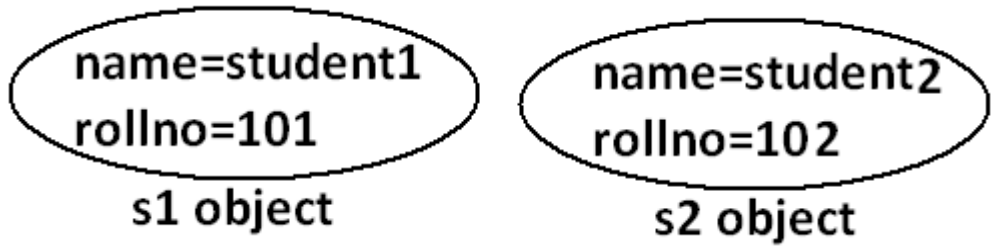
Recommended modifier for variables is private where as recommended modifier for methods is public.

Final variables:

Final instance variables:

- **If the value of a variable is varied from object to object such type of variables are called instance variables.**
- **For every object a separate copy of instance variables will be created.**

DIAGRAM:



For the instance variables it is not required to perform initialization explicitly jvm will always provide default values.

Example:

```
class Test
{
int i;
public static void main(String args[]){
Test t=new Test();
System.out.println(t.i);
}}
Output:
D:\Java>javac Test.java
D:\Java>java Test
0
```

If the instance variable declared as the final compulsory we should perform initialization explicitly and JVM won't provide any default values. whether we are using or not otherwise we will get compile time error.

Example:

Program 1:

```
class Test
{
int i;
}
Output:
D:\Java>javac Test.java
D:\Java>
```

Program 2:

```
class Test
{
final int i;
}
Output:
Compile time error.
D:\Java>javac Test.java
Test.java:1: variable i might not have been initialized
class Test
```

Rule:

For the final instance variables we should perform initialization before constructor completion. That is the following are various possible places for this.

1) At the time of declaration:

Example:

```
class Test
{
 final int i=10;
}
Output:
D:\Java>javac Test.java
```

```
D:\Java>
```

2) Inside instance block:

Example:

```
class Test
{
final int i;
{
i=10;
}}
Output:
D:\Java>javac Test.java
D:\Java>
```

3) Inside constructor:

Example:

```
class Test
{
final int i;
Test()
{
i=10;
}}
Output:
D:\Java>javac Test.java
D:\Java>
```

If we are performing initialization anywhere else we will get compile time error.

Example:

```
class Test
{
final int i;
public void methodOne(){
i=10;
}}
Output:
Compile time error.
D:\Java>javac Test.java
Test.java:5: cannot assign a value to final variable i
i=10;
```

Final static variables:

- **If the value of a variable is not varied from object to object such type of variables is not recommended to declare as the instance variables. We have to declare those variables at class level by using static modifier.**
- **In the case of instance variables for every object a seperate copy will be created but in the case of static variables a single copy will be created at class level and shared by every object of that class.**

- **For the static variables it is not required to perform initialization explicitly jvm will always provide default values.**

Example:

```
class Test
{
static int i;
public static void main(String args[]){
System.out.println("value of i is :"+i);
}}
Output:
D:\Java>javac Test.java
D:\Java>java Test
Value of i is: 0
```

If the static variable declare as final then compulsory we should perform initialization explicitly whether we are using or not otherwise we will get compile time error.(The JVM won't provide any default values)

Example:

```
class Test
{
static int i;
}
```

valid

```
class Test
{
final static int i;
}
```

invalid

```
output:
D:\Java>javac Test.java
Test.java:1: variable i might not have been initialized
class Test
```

Rule:

For the final static variables we should perform initialization before class loading completion otherwise we will get compile time error. That is the following are possible places.

1) At the time of declaration:

Example:

```
class Test
{
final static int i=10;
}
Output:
D:\Java>javac Test.java
D:\Java>
```

2) Inside static block:

Example:

```
class Test
{
final static int i;
static
{
i=10;
}}
Output:
Compile successfully.
```

If we are performing initialization anywhere else we will get compile time error.

Example:

```
class Test
{
```

```
final static int i;
public static void main(String args[]){
i=10;
}}
Output:
Compile time error.
D:\Java>javac Test.java
Test.java:5: cannot assign a value to final variable i
i=10;
```

Final local variables:

- **To meet temporary requirement of the Programmer sometime we can declare the variable inside a method or block or constructor such type of variables are called local variables.**
- **For the local variables jvm won't provide any default value compulsory we should perform initialization explicitly before using that variable.**

Example:

```
class Test
{
public static void main(String args[]){
int i;
System.out.println("hello");
}}
Output:
D:\Java>javac Test.java
D:\Java>java Test
Hello
```

Example:

```
class Test
{
public static void main(String args[]){
int i;
System.out.println(i);
}}
Output:
Compile time error.
D:\Java>javac Test.java
Test.java:5: variable i might not have been initialized
System.out.println(i);
```

Even though local variable declared as the final before using only we should perform initialization.

Example:

```
class Test
{
public static void main(String args[]){
final int i;
System.out.println("hello");
}}
Output:
D:\Java>javac Test.java
```

```
D:\Java>java Test
hello
```

Note: The only applicable modifier for local variables is final if we are using any other modifier we will get compile time error.

Example:

```
class Test
{
public static void main(String args[])
{
private int x=10;————(invalid)
public int x=10; ————(invalid)
volatile int x=10;————(invalid)
transient int x=10; ——— (invalid)
final int x=10;————(valid)
}
}
```

```
Output:
Compile time error.
D:\Java>javac Test.java
Test.java:5: illegal start of expression
private int x=10;
```

Formal parameters:

- **The formal parameters of a method are simply acts as local variables of that method hence it is possible to declare formal parameters as final.**
- **If we declare formal parameters as final then we can't change its value within the method.**

Example:

```
class Test{
public static void main(String args[]){
methodOne(10,20);
}
public static void methodOne(final int x,int y){
//x=100;
y=200;
System.out.println(x+"...."+y);
}
}
```

Formal parameters

output:

```
compile time error.
D:\Java>javac Test.java
Test.java:6: final parameter x may not be assigned
x=100;
```

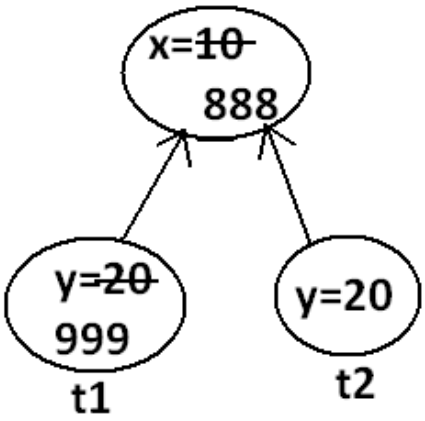
- **For instance and static variables JVM will provide default values but if instance and static declared as final JVM won't provide default value compulsory we should perform initialization whether we are using or not .**
- **For the local variables JVM won't provide any default values we have to perform explicitly before using that variables , this rule is same whether local variable final or not.**

Static modifier:

- **Static is the modifier applicable for methods, variables and blocks.**
- **We can't declare a class with static but inner classes can be declaring as the static.**
- **In the case of instance variables for every object a separate copy will be created but in the case of static variables a single copy will be created at class level and shared by all objects of that class..**

Example:

```
class Test{
static int x=10;
int y=20;
public static void main(String args[]){
Test t1=new Test();
t1.x=888;
t1.y=999;
Test t2=new Test();
System.out.println(t2.x+"....."+t2.y);
}
}
```




```
Output:
D:\Java>javac Test.java
D:\Java>java Test
888.....20
```

- **Instance variables can be accessed only from instance area directly and we can't access from static area directly.**
- **But static variables can be accessed from both instance and static areas directly.**

```
1)      Int x=10;

2)      Static int x=10;
```

```
3)      Public void methodOne(){
                    System.out.println(x);
                }

4)      Public static void methodOne(){
                     System.out.println(x);
                }
```

Which are the following declarations are allow within the same class simultaneously ?

a) 1 and 3

Example:

```
class Test
{
int x=10;
public void methodOne(){
System.out.println(x);
}}
Output:
Compile successfully.
```

b) 1 and 4

Example:

```
class Test
{
int x=10;
public static void methodOne(){
System.out.println(x);
}}
Output:
Compile time error.
D:\Java>javac Test.java
Test.java:5: non-static variable x cannot be referenced from a static
context
System.out.println(x);
```

c) 2 and 3

Example:

```
class Test
{
static int x=10;
public void methodOne(){
System.out.println(x);
}}
Output:
Compile successfully.
```

d) 2 and 4

Example:

```
class Test
{
static int x=10;
public static void methodOne(){
System.out.println(x);
}}
```

```
Output:
Compile successfully.
```

e) 1 and 2

Example:

```
class Test
{
int x=10;
static int x=10;
}
Output:
Compile time error.
D:\Java>javac Test.java
Test.java:4: x is already defined in Test
static int x=10;
```

f) 3 and 4

Example:

```
class Test{
public void methodOne(){
System.out.println(x);
}
public static void methodOne(){
System.out.println(x);
}}
Output:
Compile time error.
D:\Java>javac Test.java
Test.java:5: methodOne() is already defined in Test
public static void methodOne(){
```

For static methods implementation should be available but for abstract methods implementation is not available hence static abstract combination is illegal for methods.

case 1:

Overloading concept is applicable for static method including main method also.But JVM will always call String[] args main method .
The other overloaded method we have to call explicitly then it will be executed just like a normal method call .

Example:

```
class Test{
public static void main(String args[]){
System.out.println("String() method is called");
}
public static void main(int args[]){
System.out.println("int() method is called");
}
}
```

This method we have to call explicitly.

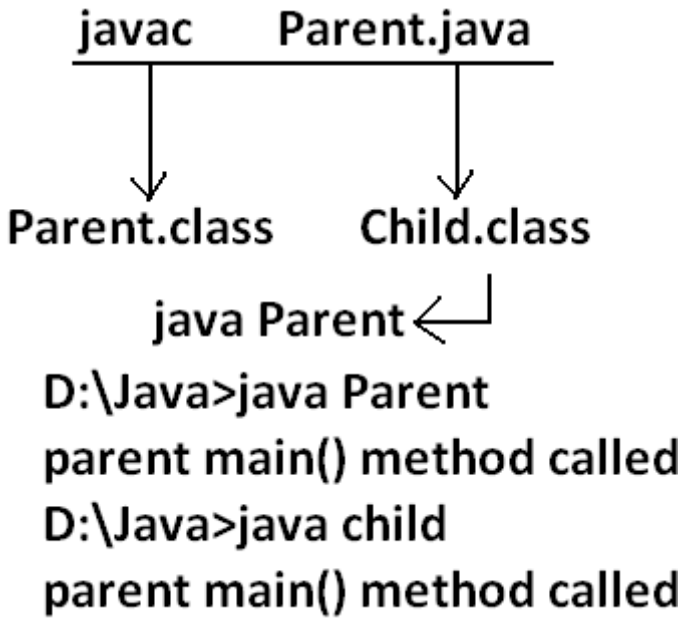
Output :
String() method is called

case 2:

Inheritance concept is applicable for static methods including main() method hence while executing child class, if the child doesn't contain main() method then the parent class main method will be executed.

Example:

```
class Parent{
public static void main(String args[]){
System.out.println("parent main() method called");
}
}
class child extends Parent{
}
Output:
```

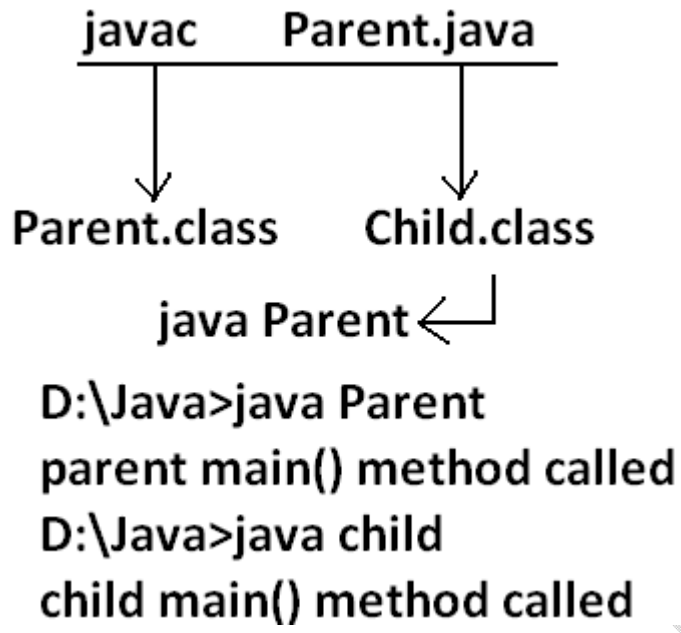

javac Parent.java
Parent.class
Child.class
java Parent
D:\Java>java Parent
parent main() method called
D:\Java>java child
parent main() method called

Example:

```
class Parent{
public static void main(String args[]){
System.out.println("parent main() method called");
}
}
class child extends Parent{
public static void main(String args[]){
System.out.println("child main() method called");
}
}
```

it is not overriding but method hiding.

Output:



- **It seems to be overriding concept is applicable for static methods but it is not overriding it is method hiding.**

Native modifier:

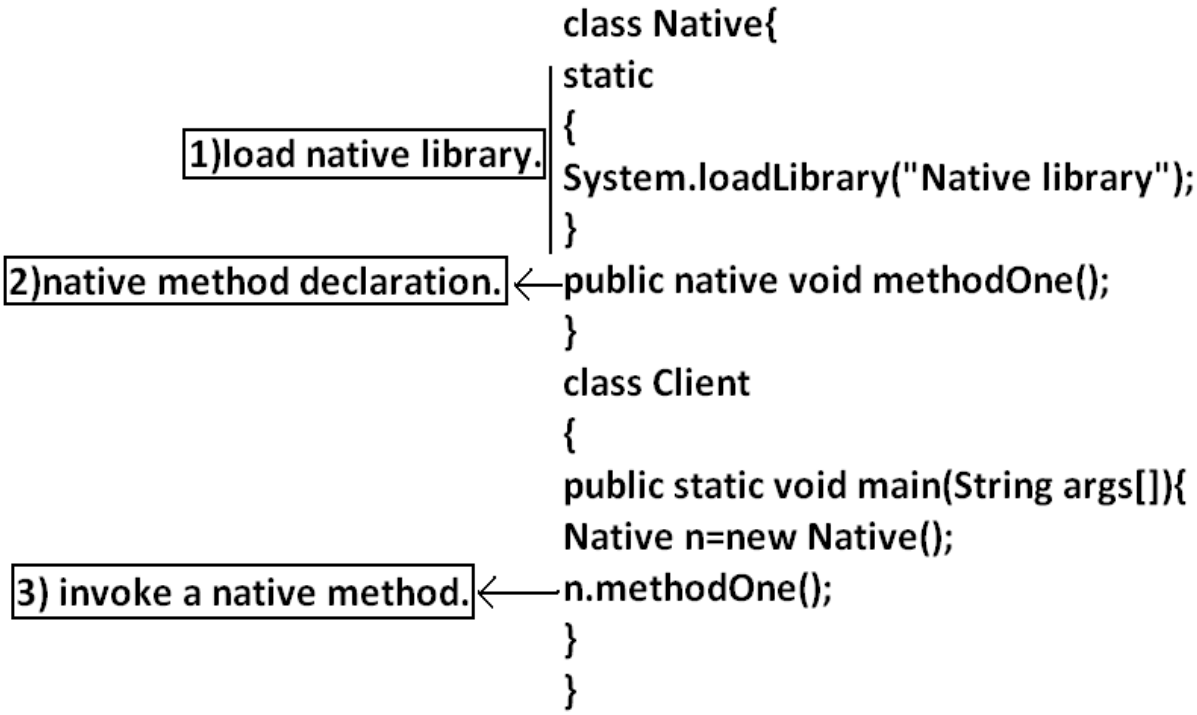
- **Native is a modifier applicable only for methods but not for variables and classes.**
- **The methods which are implemented in non java are called native methods or foreign methods.**

The main objectives of native keyword are:

- **To improve performance of the system.**
- **To use already existing legacy non-java code.**
- **To achieve machine level communication(memory level - address)**
- **Pseudo code to use native keyword in java.**

To use native keyword:

Pseudo code:



For native methods implementation is already available and we are not responsible to provide implementation hence native method declaration should compulsory ends with semicolon.

- Public native void methodOne()----invalid
- Public native void methodOne();---valid

- **For native methods implementation is already available where as for abstract methods implementation should not be available child class is responsible to provide that, hence abstract native combination is illegal for methods.**
- **We can't declare a native method as strictfp because there is no guaranty whether the old language supports IEEE754 standard or not. That is native strictfp combination is illegal for methods.**
- **For native methods inheritance, overriding and overloading concepts are applicable.**
- **The main advantage of native keyword is performence will be improves.**
- **The main disadvantage of native keyword is usage of native keyword in java breaks platform independent nature of java language.**

Synchronized:

1. **Synchronized is the modifier applicable for methods and blocks but not for variables and classes.**
2. **If a method or block declared with synchronized keyword then at a time only one thread is allow to execute that method or block on the given object.**
3. **The main advantage of synchronized keyword is we can resolve data inconsistency problems.**
4. **But the main disadvantage is it increases waiting time of the threads and effects performance of the system. Hence if there is no specific requirement never recommended to use synchronized keyword.**

For syncronized methods compulsory implementation should be available , but for abstract methods implementation won't be available , Hence abstract - syncronized combination is illegal for methods.

Transient modifier:

1. **Transient is the modifier applicable only for variables but not for methods and classes.**
2. **At the time of serialization if we don't want to serialize the value of a particular variable to meet the security constraints then we should declare that variable with transient modifier.**
3. **At the time of serialization jvm ignores the original value of the transient variable and save default value that is transient means "not to serialize".**
4. **Static variables are not part of object state hence serialization concept is not applicable for static variables duo to this declaring a static variable as transient there is no use.**
5. **Final variables will be participated into serialization directly by their values due to this declaring a final variable as transient there is no impact.**

Volatile modifier:

1. **Volatile is the modifier applicable only for variables but not for classes and methods.**
2. **If the value of variable keeps on changing such type of variables we have to declare with volatile modifier.**
3. **If a variable declared as volatile then for every thread a separate local copy will be created by the jvm, all intermediate modifications performed by the thread will takes place in the local copy instead of master copy.**
4. **Once the value got finalized before terminating the thread that final value will be updated in master copy.**
5. **The main advantage of volatile modifier is we can resolve data inconsistency problems, but creating and maintaining a separate copy for every thread increases complexity of the Programming and effects performance of the system. Hence if there is no specific requirement never recommended to use volatile modifier and it's almost outdated.**
6. **Volatile means the value keep on changing where as final means the value never changes hence final volatile combination is illegal for variables.**

Summary of modifier:

Modifiers	Classes	Method	Variable	Block	Interfaces	enum	Constructor

	Outer	Inner				Outer	inner	Outer	Inner	
public	√	√	√	√	×	√	√	√	√	√
private	×	√	√	√	×	×	√	×	√	√
protected	×	√	√	√	×	×	√	×	√	√
<default>	√	√	√	√	×	√	√	√	√	√
final	√	√	√	√	×	×	×	×	×	×
static	×	√	√	√	√	×	√	×	√	×
synchronized	×	×	√	×	√	×	×	×	×	×
abstract	√	√	√	×	×	√	√	×	×	×
native	×	×	√	×	×	×	×	×	×	×
strictfp	√	√	√	×	×	√	√	√	√	×
transient	×	×	×	√	×	×	×	×	×	×
volatile	×	×	×	√	×	×	×	×	×	×

Conclusions:

- **The Only Applicable Modifiers for Constructors are *public, private, protected,* and *<default>.***
- **The Only Applicable Modifiers for Local Variable is *final.***
- **The Only Modifier which is applicable for Classes but Not for Interfaces is *final.***
- **The Modifiers which are Applicable for Classes but Not for enum are *final* and *abstract.***
- **The Modifiers which are Applicable for Inner Classes but Not for Outer Classes are *public, protected,* and *static.***
- **The Only Modifier which is Applicable for Methods is *native.***
- **The Modifiers which are Applicable for Variables are *transient* and *volatile.***

Java Modifiers Important Questions

1) For the Top Level Classes which Modifiers are Allowed?

2) Is it Possible to Declare a Class as static, private, and protected?

3) What are Extra Modifiers Applicable for Inner Classes when compared with Outer Classes?

4) What is a final Class?

5) Explain the Differences between final, finally and finalize?

6) Is Every Method Present in final Class is final?

7) Is Every Variable Present Inside a final Class is final?

8) What is abstract Class?

9) What is abstract Method?

10) If a Class contain at least One abstract Method is it required to declared that Class Compulsory abstract?

11) If a Class doesn't contain any abstract Methods is it Possible to Declare that Class as abstract?

12) Whenever we are extending abstract Class is it Compulsory required to Provide Implementation for Every abstract Method of that Class?

13) Is final Class can contain abstract Method?

14) Is abstract Class can contain final Methods?

15) Can You give Example for abstract Class which doesn't contain any abstract Method?

16) Which of the following Modifiers Combinations are Legal for Methods?
- public – static
- static – abstract
- abstract - final
- final - synchronized
- synchronized - native
- native – abstract

17) Which of the following Modifiers Combinations are Legal for Classes?
- public - final
- final - abstract
- abstract - strictfp
- strictfp – public

18) What is the Difference between abstract Class and Interface?

19) What is strictfp Modifier?

20) Is it Possible to Declare a Variable with strictfp?

21) abstract - strictfp Combination, is Legal for Classes OR Methods?

22) Is it Possible to Override a native Method?

23) What is the Difference between Instance and Static Variable?

24) What is the Difference between General Static Variable and final Static Variable?

25) Which Modifiers are Applicable for Local Variable?

26) When the Static Variables will be Created?

27) What are Various Memory Locations of Instance Variables, Local Variables and Static Variables?

28) Is it Possible to Overload a main()?

29) Is it Possible to Override Static Methods?

30) What is native Key Word and where it is Applicable?

31) What is the Main Advantage of the native Key Word?

32) If we are using native Modifier how we can Maintain Platform Independent Nature?

33) How we can Declare a native Method?

34) Is abstract Method can contain Body?

35) What is synchronized Key Word where we can Apply?

36) What are Advantages and Disadvantages of synchronized Key Word?

37) Which Modifiers are the Most Dangerous in Java?

38) What is Serialization and Explain how its Process?

39) What is Deserialization?

40) By using which Classes we can Achieve Serialization and Deserialization?

41) What is Serializable interface and Explain its Methods?

42) What is a Marker Interface and give an Example?

43) Without having any Method in Serializable Interface, how we can get Serializable Ability for Our Object?

44) What is the Purpose of transient Key Word and Explain its Advantages?

45) Is it Possible to Serialize Every Java Object?

46) Is it Possible to Declare a Method, a Class with transient?

47) If we Declare Static Variable with transient is there any Impact?

48) What is the Impact of declaring final Variable a transient?

49) What is volatile Variable?

50) Is it Possible to Declare a Class OR a Method with volatile?

51) What is the Advantage and Disadvantage of volatile Modifier?

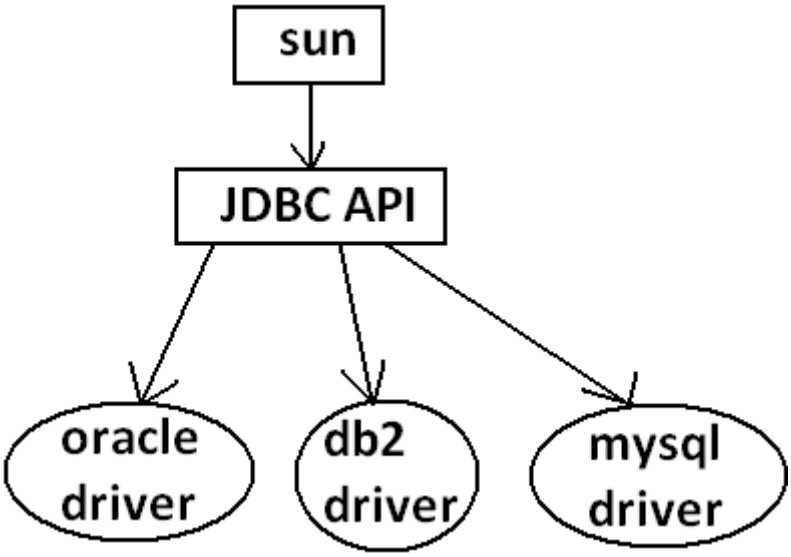
Note :

1. **The modifiers which are applicable for inner classes but not for outer classes are private, protected, static.**
2. **The modifiers which are applicable only for methods native.**
3. **The modifiers which are applicable only for variables transient and volatile.**
4. **The modifiers which are applicable for constructor public, private, protected, default.**
5. **The only applicable modifier for local variables is final.**
6. **The modifiers which are applicable for classes but not for enums are final , abstract.**
7. **The modifiers which are applicable for classes but not for interface are final.**

Interfaces:

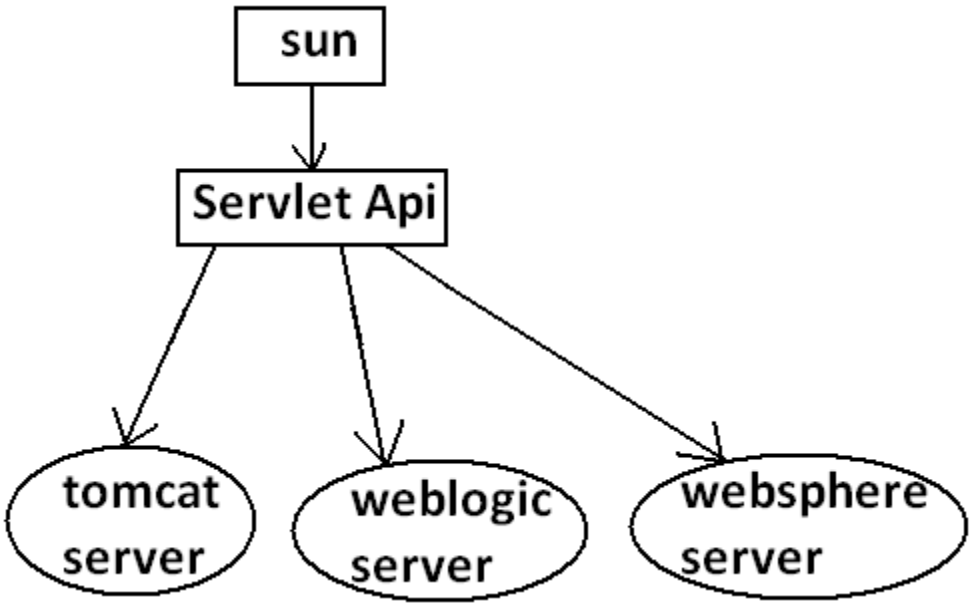
Def1: Any service requirement specification (srs) is called an interface.
Example1: Sun people responsible to define JDBC API and database vendor will provide implementation for that.

Diagram:



Example2: **Sun people define Servlet API to develop web applications web server vendor is responsible to provide implementation.**

Diagram:



Def2: **From the client point of view an interface define the set of services what is expecting. From the service provider point of view an interface defines the set of services what is offering. Hence an interface is considered as a contract between client and service provider.**

Example: **ATM GUI screen describes the set of services what bank people offering, at**

the same time the same GUI screen the set of services what customer is expecting hence this GUI screen acts as a contract between bank and customer.

Def3: Inside interface every method is always abstract whether we are declaring or not hence interface is considered as 100% pure abstract class.

Summery def: Any service requirement specification (SRS) or any contract between client and service provider or 100% pure abstract classes is considered as an interface.

Declaration and implementation of an interface:

Note1:

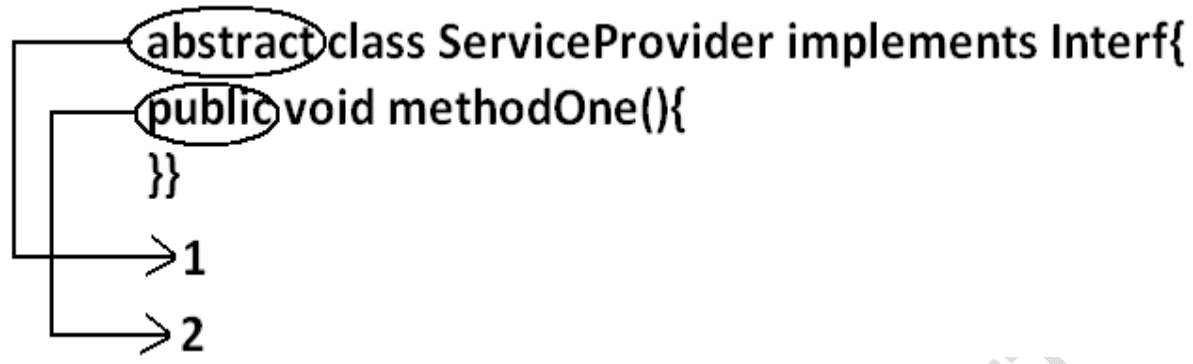
Whenever we are implementing an interface compulsory for every method of that interface we should provide implementation otherwise we have to declare class as abstract in that case child class is responsible to provide implementation for remaining methods.

Note2:

Whenever we are implementing an interface method compulsory it should be declared as public otherwise we will get compile time error.

Example:

```
interface Interf
{
void methodOne();
void methodTwo();
}
```




```
class SubServiceProvider extends ServiceProvider
{
}
Output:
Compile time error.
D:\Java>javac SubServiceProvider.java
SubServiceProvider.java:1:
SubServiceProvider is not abstract and does not override
       abstract method methodTwo() in Interf
class SubServiceProvider extends ServiceProvider
```

Extends vs implements:

A class can extends only one class at a time.

Example:

```
class One{
public void methodOne(){
}
}
class Two extends One{
}
```

A class can implements any no. Of interfaces at a time.

Example:

```
interface One{
public void methodOne();
}
interface Two{
public void methodTwo();
}
class Three implements One,Two{
public void methodOne(){
}
public void methodTwo(){
}
}
```

A class can extend a class and can implement any no. Of interfaces simultaneously.

```
interface One{
void methodOne();
}
class Two
{
public void methodTwo(){
}
}
class Three extends Two implements One{
public void methodOne(){
}
}
```

An interface can extend any no. Of interfaces at a time.

Example:

```
interface One{
void methodOne();
}
interface Two{
void methodTwo();
}
interface Three extends One,Two
{
```

}
Which of the following is true?

1. **A class can extend any no.Of classes at a time.**
2. **An interface can extend only one interface at a time.**
3. **A class can implement only one interface at a time.**
4. **A class can extend a class and can implement an interface but not both simultaneously.**
5. **An interface can implement any no.Of interfaces at a time.**
6. **None of the above.**

Ans: 6

Consider the expression X extends Y for which of the possibility of X and Y this expression is true?

1. **Both x and y should be classes.**
2. **Both x and y should be interfaces.**
3. **Both x and y can be classes or can be interfaces.**
4. **No restriction.**

Ans: 3

X extends Y, Z ?
X, Y, Z should be interfaces.

X extends Y implements Z ?
X, Y should be classes.
Z should be interface.

X implements Y, Z ?
X should be class.
Y, Z should be interfaces.

X implements Y extend Z ?

Example:

```
interface One{
}
class Two {
}
class Three implements One extends Two{
}
Output:
Compile time error.
D:\Java>javac Three.java
Three.java:5: '{' expected
class Three implements One extends Two{
```

Interface methods:

Every method present inside interface is always public and abstract whether we are declaring or not. Hence inside interface the following method declarations are equal.

```
void methodOne();
public Void methodOne();
abstract Void methodOne();                Equal
public abstract Void methodOne();
```

public: To make this method available for every implementation class.
abstract: Implementation class is responsible to provide implementation .

As every interface method is always public and abstract we can't use the following modifiers for interface methods.
Private, protected, final, static, synchronized, native, strictfp.

Inside interface which method declarations are valid?

1. **public void methodOne(){}**
2. **private void methodOne();**
3. **public final void methodOne();**
4. **public static void methodOne();**
5. **public abstract void methodOne();**

Ans: 5

Interface variables:

- **An interface can contain variables**
- **The main purpose of interface variables is to define requirement level constants.**
- **Every interface variable is always public static and final whether we are declaring or not.**

Example:

```
interface interf
{
int x=10;
}
```

public: To make it available for every implementation class.
static: Without existing object also we have to access this variable.
final: Implementation class can access this value but cannot modify.

Hence inside interface the following declarations are equal.

```
int x=10;
public int x=10;
static int x=10;
final int x=10;                          Equal
public static int x=10;
public final int x=10;
static final int x=10;
public static final int x=10;
```

- **As every interface variable by default public static final we can't declare with the following modifiers.**
  - **Private**
  - **Protected**
  - **Transient**
  - **Volatile**
- **For the interface variables compulsory we should perform initialization at the time of declaration only otherwise we will get compile time error.**

Example:

```
interface Interf
{
int x;
}
Output:
Compile time error.
D:\Java>javac Interf.java
Interf.java:3: = expected
int x;
```

Which of the following declarations are valid inside interface ?

1. **int x;**
2. **private int x=10;**
3. **public volatile int x=10;**
4. **public transient int x=10;**
5. **public static final int x=10;**

Ans: 5

Interface variables can be access from implementation class but cannot be modified.

Example:

```
interface Interf
{
int x=10;
}
```

Example 1:

```
class Test implements Interf
{
public static void main(String args[]){
x=20;
System.out.println("value of x"+x);
}
}
```

output:

```
compile time error.
D:\Java>javac Test.java
Test.java:4: cannot assign a value to final variable x
x=20;
```

Example 2:

```
class Test implements Interf
{
public static void main(String args[]){
int x=20;
//here we declaring the variable x.
System.out.println(x);
}
}
Output:
D:\Java>javac Test.java
D:\Java>java Test
20\
```

Interface naming conflicts:

Method naming conflicts:

Case 1:

If two interfaces contain a method with same signature and same return type in the implementation class only one method implementation is enough.

Example 1:

```
interface Left
{
public void methodOne();
}
```

Example 2:

```
interface Right
{
public void methodOne();
}
```

Example 3:

```
class Test implements Left,Right
{
public void methodOne()
{
}}
Output:
D:\Java>javac Left.java
D:\Java>javac Right.java
D:\Java>javac Test.java
```

Case 2:

if two interfaces contain a method with same name but different arguments in the implementation class we have to provide implementation for both methods and these methods acts as a overloaded methods

Example 1:

```
interface Left
{
public void methodOne();
}
```

Example 2:

```
interface Right
{
public void methodOne(int i);
}
```

Example 3:

```
class Test implements Left,Right
{
public void methodOne()
{
}
public void methodOne(int i)
{
}}
Output:
D:\Java>javac Left.java
D:\Java>javac Right.java
D:\Java>javac Test.java
```

Case 3:

If two interfaces contain a method with same signature but different return types then it is not possible to implement both interfaces simultaneously.

Example 1:

```
interface Left
{
public void methodOne();
```

```
}
```

Example 2:

```
interface Right
{
public int methodOne(int i);
}
```

We can't write any java class that implements both interfaces simultaneously.

Is a java class can implement any no. Of interfaces simultaneously ?
Yes, except if two interfaces contains a method with same signature but different return types.

Variable naming conflicts:

Two interfaces can contain a variable with the same name and there may be a chance variable naming conflicts but we can resolve variable naming conflicts by using interface names.

Example 1:

```
interface Left
{
int x=888;
}
```

Example 2:

```
interface Right
{
int x=999;
}
```

Example 3:

```
class Test implements Left,Right
{
public static void main(String args[]){
//System.out.println(x);
System.out.println(Left.x);
System.out.println(Right.x);
}
}
Output:
D:\Java>javac Left.java
D:\Java>javac Right.java
D:\Java>javac Test.java
D:\Java>java Test
888
999
```

Marker interface:

If an interface doesn't contain any methods and by implementing that interface if our objects will get some ability such type of interfaces are called Marker interface (or) Tag interface (or) Ability interface.

Example:

```
Serializable
Cloneable
RandomAccess          These are marked for some ability
SingleThreadModel
        .
        .
        .
        .
```


Example 1:

By implementing Serilaizable interface we can send that object across the network and we can save state of an object into a file.

Example 2:

By implementing SingleThreadModel interface Servlet can process only one client request at a time so that we can get "Thread Safety".

Example 3:

By implementing Cloneable interface our object is in a position to provide exactly duplicate cloned object.

Without having any methods in marker interface how objects will get ability ?
Internally JVM is responsible to provide required ability.

Why JVM is providing the required ability in marker interfaces ?
To reduce complexity of the programming.

Is it possible to create our own marker interface ?
Yes, but customization of JVM must be required.
Ex : Sleepable , Jumpable , ...

Adapter class:

- **Adapter class is a simple java class that implements an interface only with empty implementation for every method.**
- **If we implement an interface directly for each and every method compulsory we should provide implementation whether it is required or not. This approach increases length of the code and reduces readability.**

Example 1:

```
interface X{
void m1();
void m2();
void m3();
void m4();
  //.
  //.
  //.
  //.
void m5();
}
```

Example 2:

```
class Test implements X{
public void m3(){
System.out.println("m3() method is called");
}
public void m1(){}
public void m2(){}
public void m4(){}
public void m5(){}
}
```

- **We can resolve this problem by using adapter class.**
- **Instead of implementing an interface if we can extend adapter class we have to provide implementation only for required methods but not for all methods of that interface.**
- **This approach decreases length of the code and improves readability.**

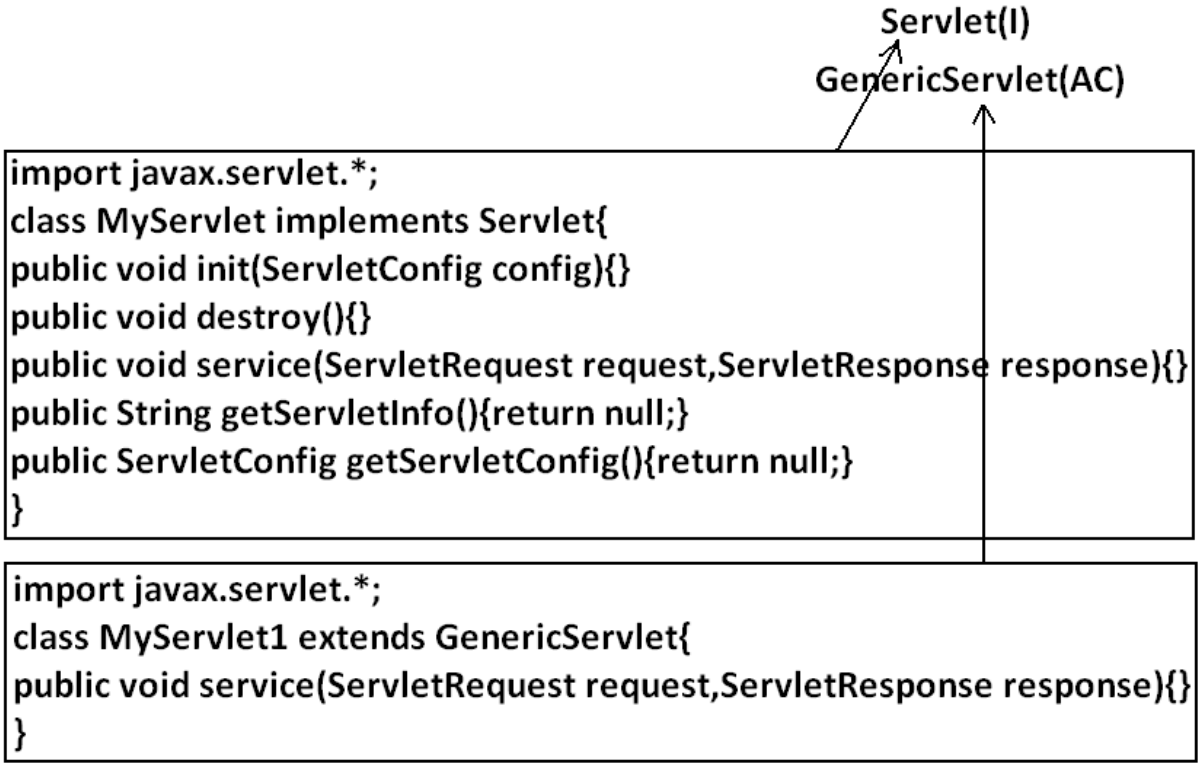
Example 1:

```
abstract class AdapterX implements X{
public void m1(){}
public void m2(){}
public void m3(){}
public void m4(){}
 //.
 //.
 //.
public void m1000(){}
}
```

Example 2:

```
public class Test extend AdapterX{{
public void m3(){
}}
```

Example:

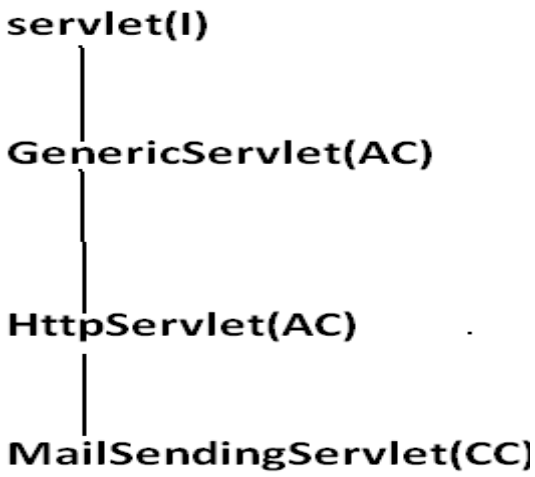


Generic Servlet simply acts as an adapter class for Servlet interface.
Note : marker interface and Adapter class are big utilities to the programmer to simplify programming.

What is the difference between interface, abstract class and concrete class? When we should go for interface, abstract class and concrete class?

- **If we don't know anything about implementation just we have requirement specification then we should go for interface.**
- **If we are talking about implementation but not completely (partial implementation) then we should go for abstract class.**
- **If we are talking about implementation completely and ready to provide service then we should go for concrete class.**

Example:



What is the Difference between interface and abstract class ?

interface	Abstract class
If we don't' know anything about implementation just we have requirement specification then we should go for interface.	If we are talking about implementation but not completely (partial implementation) then we should go for abstract class.
Every method present inside interface is always public and abstract whether we are declaring or not.	Every method present inside abstract class need not be public and abstract.
We can't declare interface methods with the modifiers private, protected, final, static, synchronized, native, strictfp.	There are no restrictions on abstract class method modifiers.
Every interface variable is always public static final whether we are declaring or not.	Every abstract class variable need not be public static final.
Every interface variable is always public static final we can't declare with the	There are no restrictions on abstract class variable modifiers.

following modifiers. Private, protected, transient, volatile.	
For the interface variables compulsory we should perform initialization at the time of declaration otherwise we will get compile time error.	**It is not require to perform initialization for abstract class variables at the time of declaration.**
Inside interface we can't take static and instance blocks.	**Inside abstract class we can take both static and instance blocks.**
Inside interface we can't take constructor.	**Inside abstract class we can take constructor.**

We can't create object for abstract class but abstract class can contain constructor what is the need ?

abstract class constructor will be executed when ever we are creating child class object to perform initialization of child object.

Example:

```
class Parent{
Parent()
{
System.out.println(this.hashCode());
}
}
class child extends Parent{
child(){
System.out.println(this.hashCode());
}
}
class Test{
public static void main(String args[]){
child c=new child();
System.out.println(c.hashCode());
}
}
```

Note : We can't create object for abstract class either directly or indirectly.

Every method present inside interface is abstract but in abstract class also we can take only abstract methods then what is the need of interface concept ?

We can replace interface concept with abstract class. But it is not a good programming practice. We are misusing the roll of abstract class. It may create performence problems also.
(this is just like recruiting IAS officer for sweeping purpose)

Example :

```
interface X {
 -------
 -------
}
class Test  implements  X {
 ----------
 ----------
}

Test t=new Test();
        // takes 2 sec
```

1) performance is high
2) while implementing X we can extend some other classes

```
abstract class X  {
  -----------------
  ----------------
}
class Test extends  X {
  ------------------
  ------------------
}

Test t=new Test();
        //takes  20sec
```

1) performance is low
2) while extending X we can't extend any other classes

If every thing is abstract then it is recommended to go for interafce.

Why abstract class can contain constructor where as interface doesn't contain constructor ?

The main purpose of constructor is to perform initialization of an object i.e., provide values for the instance variables, Inside interface every variable is always static and there is no chance of existing instance variables. Hence constructor is not required for interface.

But abstract class can contains instance variable which are required for the child object to perform initialization for those instance variables constructor is required in the case of abstract class.

Interfaces Important Questions

1) What is Interface?

2) What is Difference between Interface and Abstract Class?

3) When we should Go for Interface and Abstract Class and Concrete Class?

4) What Modifiers Applicable for Interfaces?

5) Explain about Interface Variables and what Modifiers are Applicable for them?

6) Explain about Interface Methods and what Modifiers are Applicable for them?

7) Can Java Class implement any Number of Interfaces?

8) If 2 Interfaces contains a Method with Same Signature but different Return Types, then how we can implement Both Interfaces Simultaneously?

9) Difference between extends and implements Key Word?

10) We cannot Create an Object of Abstract Class then what is Necessity of having Constructor Inside Abstract Class?

11) What is a Marker Interface? Give an Example?

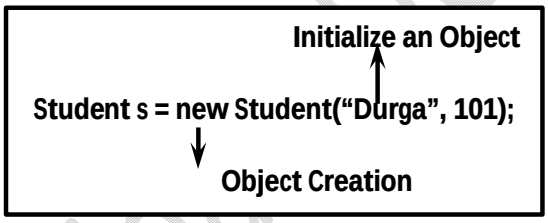
12) What is Adapter Class and Explain its Usage?

13) An Interface contains only Abstract Methods and an Abstract Class also can contain only Abstract Methods then what is the Necessity of Interface?

14) In Your Previous Project where You used the following Marker Interface, Abstract Class, Interface and Adapter Class?

The Purpose of Constructor is

- **To Initialize an Object but Not to Create an Object.**
- **Whenever we are creating an Object after Object Creation automatically Constructor will be executed to Initialize that Object.**
- **Object Creation by New Operator and then Initialization by Constructor.**



- **Before Constructor Only Object is Ready and Hence within the Constructor we can Access Object Properties Like Hash Code.**

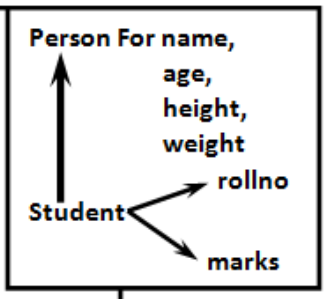
```
class Test {
        Test() {
                System.out.println(this); // Test@6e3d60
                System.out.println(this.hashCode()); //7224672
        }
        public static void main(String[] args) {
                Test t = new Test();
        }
                                }
```

- **Whenever we are creating Child Class Object automatically Parent Constructor will be executed but Parent Object won't be created.**
- **The Purpose of Parent Constructor Execution is to Initialize Child Object Only. Of Course for the Instance Variables which are inheriting from parent Class.**

```
class P {
        P() {
                System.out.println(this.hashCode()); //7224672
        }
}
class C extends P {
                C() {
                        System.out.println(this.hashCode()); //7224672
                }
}
class Test {
        public static void main(String[] args) {
                C c = new C();
                System.out.println(c.hashCode()); //7224672
        }
}
```

- **In the Above Example whenever we are creating Child Object Both Parent and Child Constructors will be executed for Child Object Purpose Only.**
- **In the Above Example we are Just creating Student Object but Both Person and Student Constructor to Initialize Student Object Only.**

```
Person (String name, int age, int height, int weight)
{
        this.name = name;
        this.age = age;
        this.height = height;
        this.weight = weight;
}

Student (String name, int age, int height, int weight, int rollno, int marks)
{
        this.name = name;
        this.age = age;
        this.height = height;
        this.weight = weight;
        this.rollno = rollno;
        this.marks = marks;
}

Student s = new Student ("Ravi", 30, 6, 60, 101, 70);
```



name: Ravi
age: 30
height: 6
weight: 60
} These 4 Properties will be initialized by Parent Constructor

rollno: 101
marks: 70
} These 2 Properties will be initialized by Child Constructor

- **Whenever we are creating an Object automatically Constructor will be executed but it May be One Constructor OR Multiple Constructors.**
- **Either Directly OR Indirectly we can't Create Object for Abstract Class but Abstract Class can contain Constructor what is the Need.**
- **Whenever we are creating Child Object Automatically Abstract Class Constructor will be executed to Perform Initialization of Child Object for the Properties whether inheriting from Abstract Class (Code Reusability).**

Without abstract Class Constructor

```
class Student extends Person {
        int rollno;
        int marks;
        Student(String name, int age, int height, int weight) {
                this.name = name;
                this.age = age;
```

With abstract Class Constructor

```
abstract class Person {
	String name;
	int age;
	int height;
	int weight;
	public Person(String name, int age, int height, int weight) {
			super();
			this.name = name;
			this.age = age;
```

Any Way we can't Create Object for Abstract Class and Interface. But Abstract Class can contain Constructor but Interface doesn't Why?

- **The Main Purpose of Constructor is to Perform Initialization of an Object i.e. to Provide Values for Instance Variables.**
- **Abstract Class can contain Instance Variables which are required for Child Class Object to Perform Initialization of these Instance Variables Constructor is required Inside Abstract Class.**
- **But Every Variable Present Inside Interface is Always public, static and final whether we are declaring OR Not and Every Interface Variable we should Perform Initialization at the Time of Declaration and Hence Inside Interface there is No Chance of existing Instance Variable.**
- **Due to this Initialization of Instance Variables Concept Not Applicable for Interfaces.**
- **Hence Constructor Concept Not required for Interface.**

```
abstract class Person {
        String name;
        int age;
        Person(String name, int age) {
           this.name = name;
                this.age = age; → Current Child Class Object
        }
}
```

Inside Interface we can take Only Abstract Methods. But in Abstract Class Also we can take Only Abstract Methods Based on Our Requirement. Then what is the Need of Interface? i.e. Is it Possible to Replace Interface Concept with Abstract Class?

We can Replace Interface with Abstract Class but it is Not Good Programming Practice (This is Like requesting IAS Officer for sweeping Activity)

❶

```
abstract class X
{
    ---------------
    ---------------
}
class Test extends X {}
```

❷

```
interface X
{
    ---------------
    ---------------
}
class Test implements X {}
```

- **While extending X we can't extend any Other Class.**
- **While implementing Interface we can extend any Other Class and Hence we won't Miss any Inheritance Benefit.**

- **In the Case ❶ Object Creation is Costly.**
  Test t = new Test(); → 2 Mins

- **In the Case ❷ Object Creation is Not Costly.**
  Test t = new Test(); → 2 Sec

- **Hence if everything is Abstract Highly Recommended to Use Interface but Not Abstract Class.**
- **If we are talking About Implementation of Course Partial Implementation then Only we should go for Abstract Class.**

Which of the following are Valid?

1) The Purpose of Constructor is to Create an Object. ✕
2) The Purpose of Constructor is to Initialize an Object but Not to Create Object. √
3) Once Constructor completes then Only Object Creation completes. ✕
4) First Object will be Created and then Constructor will be executed. √
5) The Purpose of new Key Word is to Create Object and the Purpose of Constructor is to Initialize that Object. √
6) We can't Create Object for Abstract Class Directly but Indirectly we can Create. ✕
7) Whenever we are creating Child Class Object Automatically Parent Class Object will be created Internally. ✕
8) Whenever we are creating Child Class Object Automatically Abstract Class Constructor will be executed. √
9) Whenever we are creating Child Class Object Automatically Parent Object will be Created. ✕
10) Whenever we are creating Child Class Object Automatically Parent Constructor will be executed but Parent Object won't be Created. √
11) Either Directly OR Indirectly we can't Create Object for Abstract Class and Hence Constructor Concept is Not Applicable for Abstract Class. ✕
12) Interface can contain Constructor. ✕

CORE JAVA

With

SCJP / OCJP

Study Material

Chapter 5: OOPS

DURGA M.Tech

(Sun certified & Realtime Expert)

Ex. IBM Employee

Trained Lakhs of Students
for last 14 years across INDIA

India's No.1 Software Training Institute

DURGASOFT

www.durgasoft.com Ph: 9246212143 ,8096969696

Object Oriented Programming (OOPS)

Agenda:

1. **Data Hiding**
2. **Abstraction**
3. **Encapsulation**
4. **Tightly Encapsulated Class**
5. **IS-A Relationship(Inheritance)**
   - **Multiple inheritance**
   - **Cyclic inheritance**
6. **HAS-A Relationship**
   - **Composition**
   - **Aggregation**
7. **Method Signature**
8. **Polymorphism**
   - **Overloading**
     - **Automatic promotion in overloading**
   - **Overriding**
     - **Rules for overriding**
     - **Checked Vs Un-checked Exceptions**
     - **Overriding with respect to static methods**
     - **Overriding with respect to Var-arg methods**
     - **Overriding with respect to variables**
     - **Differences between overloading and overriding ?**
   - **Method Hiding**
9. **Static Control Flow**
   - **Static control flow parent to child relationship**
   - **Static block**
10. **Instance Control Flow**
    - **Instance control flow in Parent to Child relationship**
11. **Constructors**
    - **Constructor Vs instance block**
    - **Rules to write constructors**
    - **Default constructor**
    - **Prototype of default constructor**
    - **super() vs this():**
    - **Overloaded constructors**
    - **Recursive functions**
12. **Coupling**
13. **Cohesion**
14. **Object Type Casting**
    - **Compile time checking**
    - **Runtime checking**

- **Difference between ArrayList l=new ArrayList() & List l=new ArrayList() ?**
- **In how many ways get an object in java ?**

- **Singleton classes**
- **Factory method**

Data Hiding :

- **Our internal data should not go out directly that is outside person can't access our internal data directly.**
- **By using private modifier we can implement data hiding.**

```
Example:
class Account  {
        private double balance;
        ......................;
        ......................;
}
```

After providing proper username and password only , we can access our Account information.

The main advantage of data hiding is security.
Note: recommended modifier for data members is private.

Abstraction :

- **Hide internal implementation and just highlight the set of services, is called abstraction.**
- **By using abstract classes and interfaces we can implement abstraction.**

Example :

By using ATM GUI screen bank people are highlighting the set of services what they are offering without highlighting internal implementation.

The main advantages of Abstraction are:

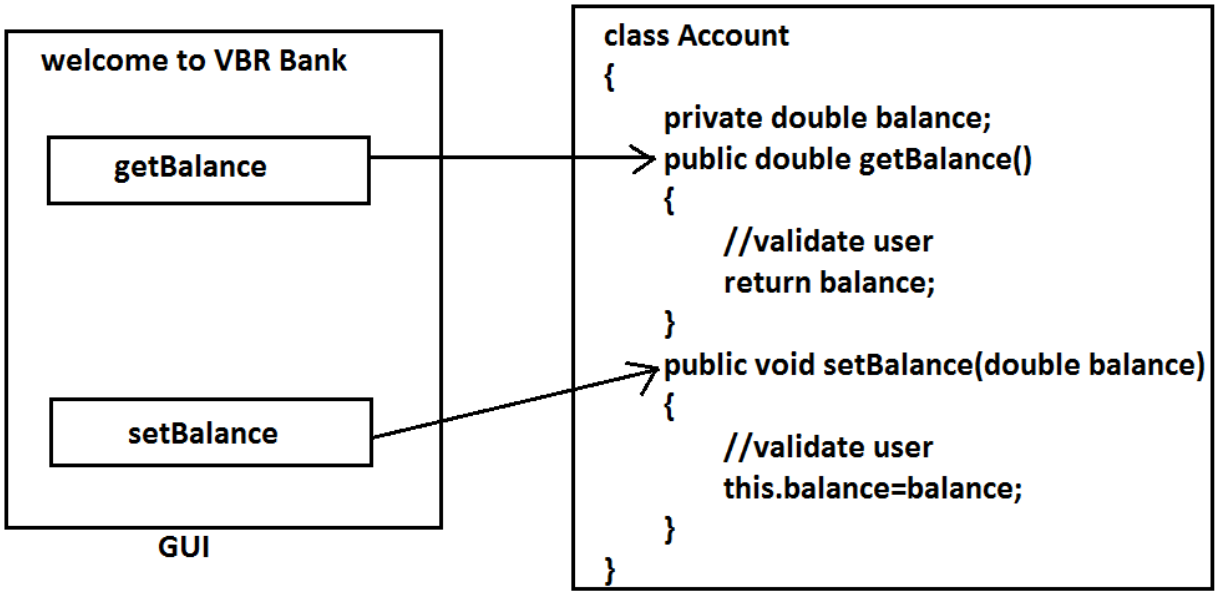
1. **We can achieve security as we are not highlighting our internal implementation.(i.e., outside person doesn't aware our internal implementation.)**
2. **Enhancement will become very easy because without effecting end user we can able to perform any type of changes in our internal system.**
3. **It provides more flexibility to the end user to use system very easily.**
4. **It improves maintainability of the application.**
5. **It improves modularity of the application.**
6. **It improves easyness to use our system.**

By using interfaces (GUI screens) we can implement abstraction

Encapsulation :

- **Binding of data and corresponding methods into a single unit is called Encapsulation .**
- **If any java class follows data hiding and abstraction such type of class is said to be encapsulated class.**

Encapsulation=Datahiding+Abstraction
Example:



Every data member should be declared as private and for every member we have to maintain getter & Setter methods.

<u>The main advantages of encapsulation are :</u>

1. **We can achieve security.**
2. **Enhancement will become very easy.**
3. **It improves maintainability and modularity of the application.**
4. **It provides flexibility to the user to use system very easily.**

The main disadvantage of encapsulation is it increases length of the code and slows down execution.

Tightly encapsulated class :

A class is said to be tightly encapsulated if and only if every variable of that class declared as private whether the variable has getter and setter methods are not , and whether these methods declared as public or not, these checkings are not required to perform.

```
Example:
class Account   {
        private double balance;
        public double getBalance()  {
                return balance;
        }
}
```

Which of the following classes are tightly encapsulated?

```
class A
{
      private int x=10; (valid)
}
class B extends A
{
      int y=20;(invalid)
}
class C extends A
{
      private int z=30; (valid)
}
```

Which of the following classes are tightly encapsulated?

```
class A {
        int x=10;    //not
}
class B extends A {
        private int y=20;  //not
}
class C extends B {
        private int z=30;  //not
}
```

Note: if the parent class is not tightly encapsulated then no child class is tightly encapsulated.

IS-A Relationship(inheritance) :

1. Also known as inheritance.
2. By using "extends" keywords we can implement IS-A relationship.
3. The main advantage of IS-A relationship is reusability.

```
Example:
class Parent {
        public void methodOne(){ }
}
class Child extends Parent {
        public void methodTwo() { }
}
```

```
class Test
{
    public static void main(String[] args)
    {
        Parent p=new Parent();
        p.methodOne();
        p.methodTwo();
        Child c=new Child();
        c.methodOne();
        c.methodTwo();
        Parent p1=new Child();
        p1.methodOne();
        p1.methodTwo();
        Child c1=new Parent();
    }
}
```

C.E: cannot find symbol
symbol : method methodTwo()
location: class Parent

C.E: incompatible types
found : Parent
required: Child

Conclusion :

1. **Whatever the parent has by default available to the child but whatever the child has by default not available to the parent. Hence on the child reference we can call both parent and child class methods. But on the parent reference we can call only methods available in the parent class and we can't call child specific methods.**
2. **Parent class reference can be used to hold child class object but by using that reference we can call only methods available in parent class and child specific methods we can't call.**
3. **Child class reference cannot be used to hold parent class object.**

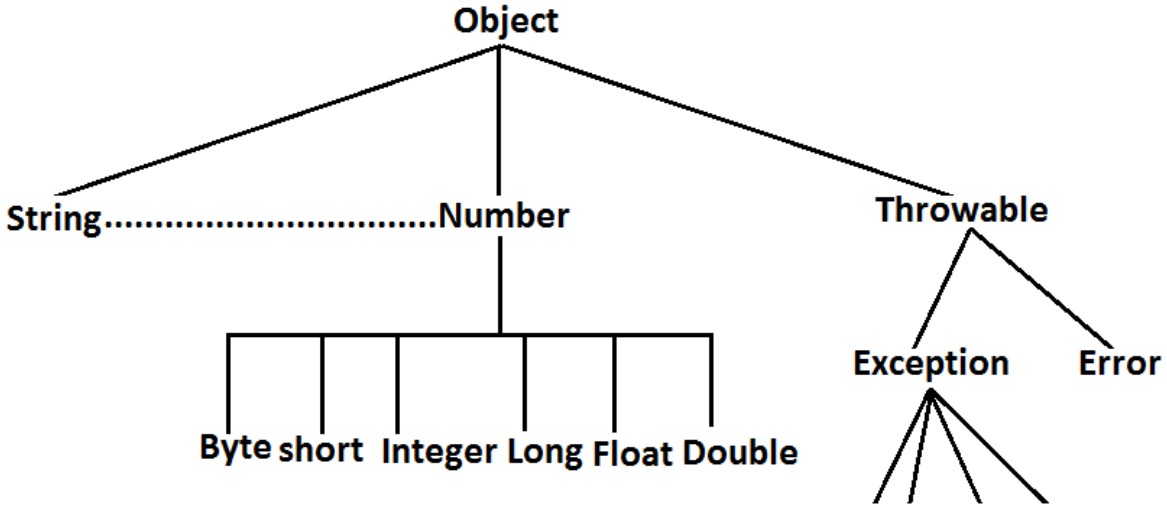
Example:

The common methods which are required for housing loan, vehicle loan, personal loan and education loan we can define into a separate class in parent class loan. So that automatically these methods are available to every child loan class.

```
Example:
class Loan {
        //common methods which are required for any type of loan.
}
class HousingLoan extends Loan {
        //Housing loan specific methods.
}
class EducationLoan extends Loan {
        //Education Loan specific methods.
}
```

- **For all java classes the most commonly required functionality is define inside object class hence object class acts as a root for all java classes.**
- **For all java exceptions and errors the most common required functionality defines inside Throwable class hence Throwable class acts as a root for exception hierarchy.**

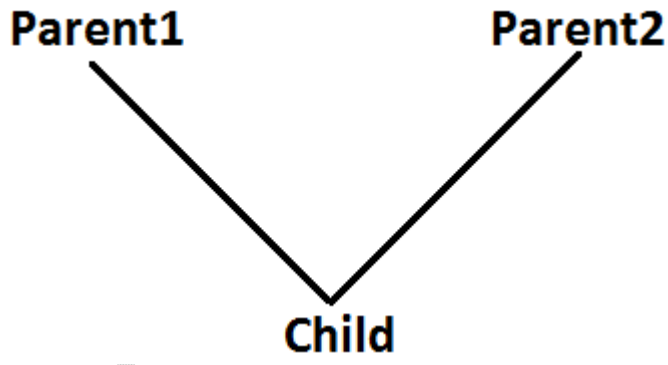
Diagram:



Multiple inheritance :

Having more than one Parent class at the same level is called multiple inheritance.

Example:



Any class can extends only one class at a time and can't extends more than one class simultaneously hence java won't provide support for multiple inheritance.

Example:

```
class A{}
class B{}
class C extends A,B
{}
```
(invalid)

But an interface can extends any no. Of interfaces at a time hence java provides support for multiple inheritance through interfaces.

Example:

```
interface A{}
interface B{}
interface C extends A,B{}
```

If our class doesn't extends any other class then only our class is the direct child class of object.

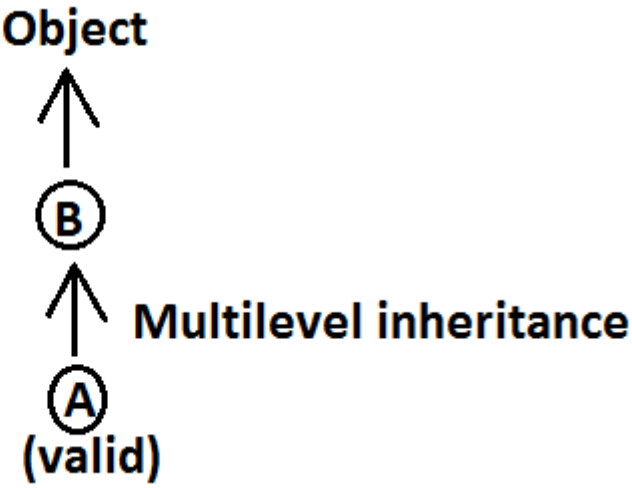
Example:

```
class A{}
```

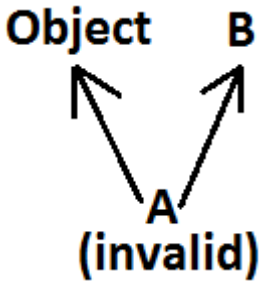
If our class extends any other class then our class is not direct child class of object, It is indirect child class of object , which forms multilevel inheritance.

Example 1:

```
class B{}
class  A extends B{}
```



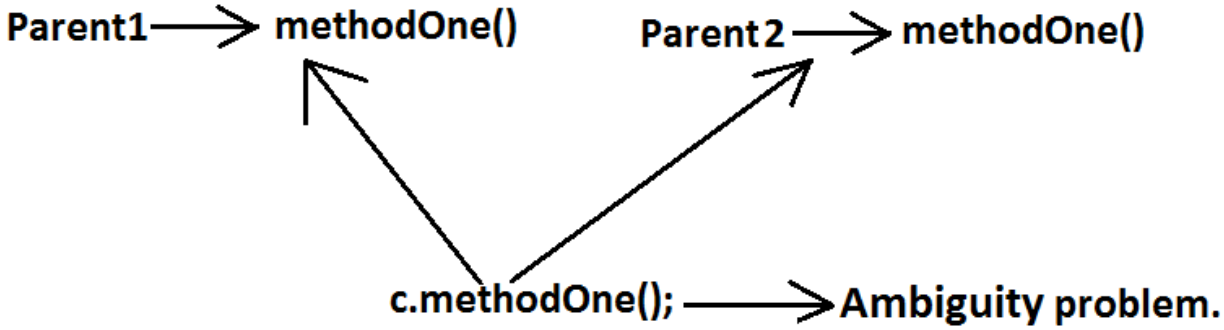
Example 2:



Why java won't provide support for multiple inheritance?

There may be a chance of raising ambiguity problems.

Example:



Why ambiguity problem won't be there in interfaces?

Interfaces having dummy declarations and they won't have implementations hence no ambiguity problem.

Example:

```
interface inter1
{
      public void methodOne();
}
```

```
interface inter2
{
      public void methodOne();
}
```

```
interface inter3 extends inter1,inter2{}
```

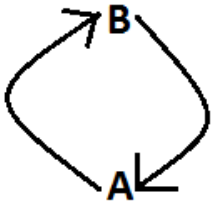
```
class Test implements inter3
{
      public void methodOne()
      {
            System.out.println("This is methodOne()");
      }
}
```

Cyclic inheritance :

Cyclic inheritance is not allowed in java.

Example 1:

class A extends B{} } (invalid)
class B extends A{} } C.E:cyclic inheritance involving A



Example 2:

class A extends A{} —C.E→ cyclic inheritance involving A

HAS-A relationship:

1. **HAS-A relationship is also known as composition (or) aggregation.**
2. **There is no specific keyword to implement HAS-A relationship but mostly we can use new operator.**
3. **The main advantage of HAS-A relationship is reusability.**

```
Example:
class Engine
{
        //engine specific functionality
}
class Car
{
        Engine e=new Engine();
        //.......................;
        //.......................;
        //.......................;
}
```

- **class Car HAS-A engine reference.**
- **The main dis-advantage of HAS-A relationship increases dependency between the components and creates maintains problems.**

Composition vs Aggregation:

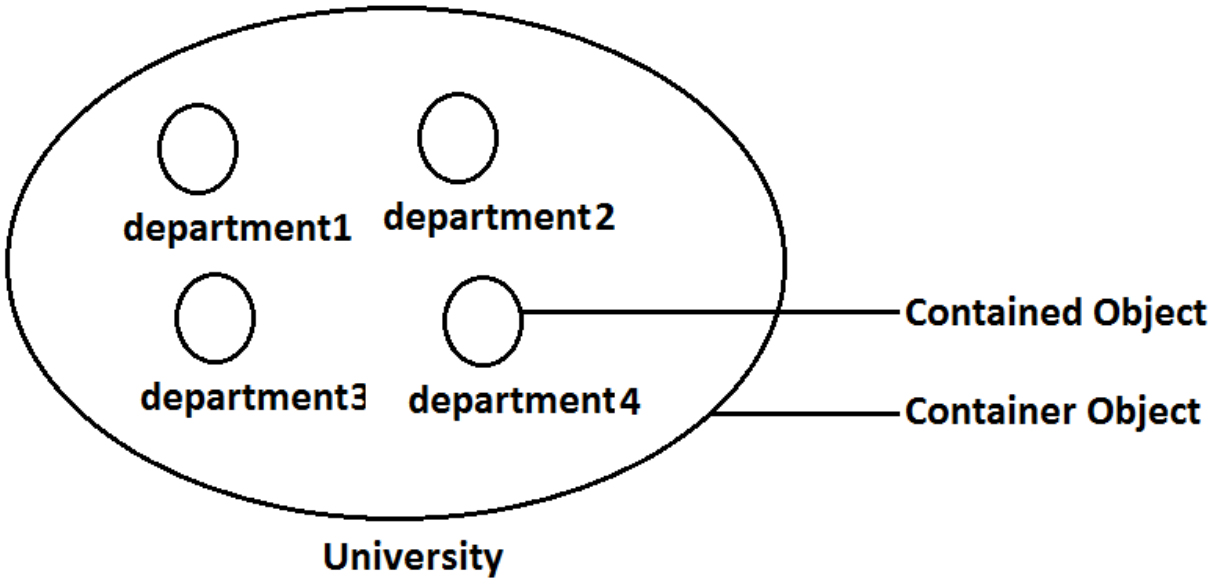
Composition:

Without existing container object if there is no chance of existing contained objects then the relationship between container object and contained object is called composition which is a strong association.

Example:

University consists of several departments whenever university object destroys automatically all the department objects will be destroyed that is without existing university object there is no chance of existing dependent object hence these are strongly associated and this relationship is called composition.

Example:



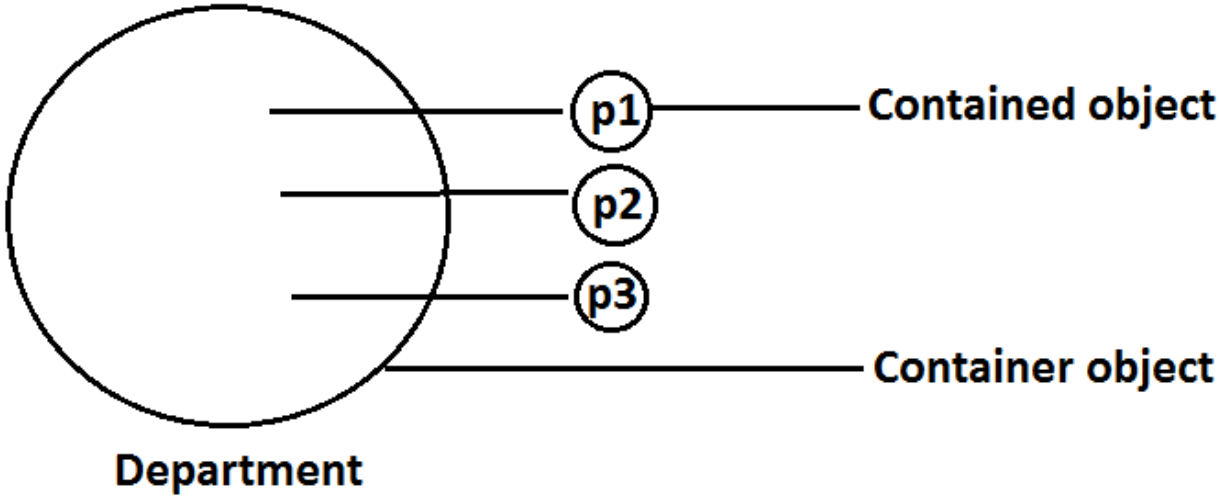
Aggregation :

Without existing container object if there is a chance of existing contained objects such type of relationship is called aggregation. In aggregation objects have weak association.

Example:

Within a department there may be a chance of several professors will work whenever we are closing department still there may be a chance of existing professor object without existing department object the relationship between department and professor is called aggregation where the objects having weak association.

Example:



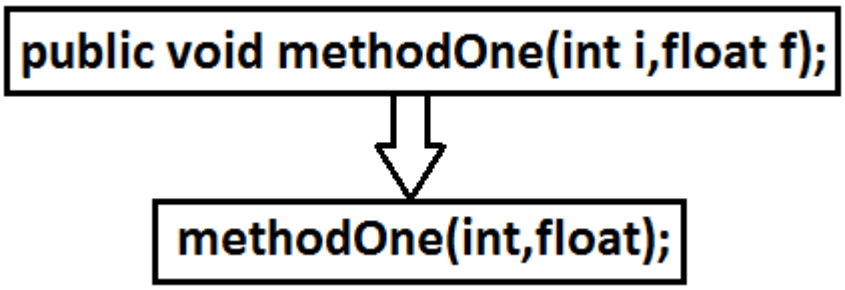
Note :

In composition container , contained objects are strongly associated, and but container object holds contained objects directly
But in Aggregation container and contained objects are weakly associated and container object just now holds the reference of contained objects.

Method signature:

In java, method signature consists of name of the method followed by argument types.

Example:



- **In java return type is not part of the method signature.**
- **Compiler will use method signature while resolving method calls.**

```
class Test {
public void m1(double d) {  }
public void m2(int i) { }
public static void main(String ar[]) {
 Test t=new Test();
 t.m1(10.5);
 t.m2(10);
 t.m3(10.5);  //CE
}
}
CE : cannot find symbol
symbol : method m3(double)
location : class Test
```

Within the same class we can't take 2 methods with the same signature otherwise we will get compile time error.

```
Example:
public void methodOne() {  }
public int methodOne()  {
        return 10;
}
Output:
Compile time error
methodOne() is already defined in Test
```

Polymorphism:

Same name with different forms is the concept of polymorphism.

Example 1: We can use same abs() method for int type, long type, float type etc.

Example:

1. **abs(int)**
2. **abs(long)**
3. **abs(float)**

Example 2:
We can use the parent reference to hold any child objects.
We can use the same List reference to hold ArrayList object, LinkedList object, Vector object, or Stack object.

Example:

1. **List l=new ArrayList();**
2. **List l=new LinkedList();**
3. **List l=new Vector();**
4. **List l=new Stack();**

Diagram:

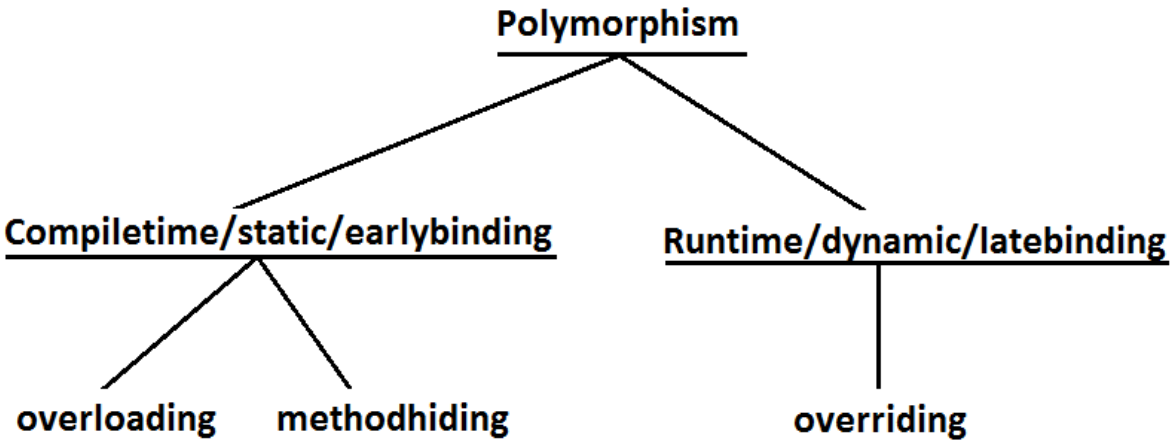
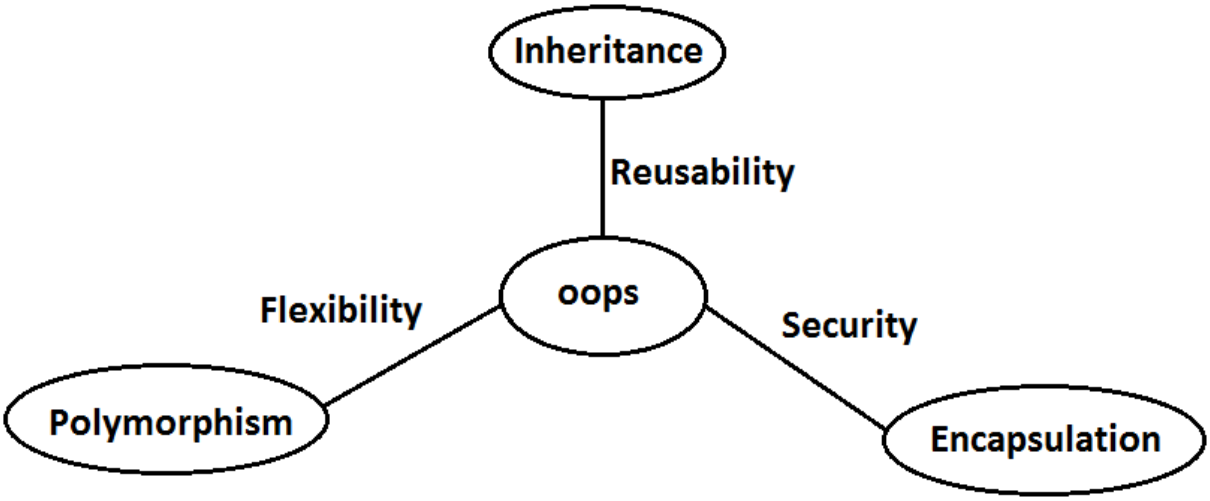


Diagram: 3 Pillars of OOPS



1) Inheritance talks about reusability.
2) Polymorphism talks about flexibility.
3) Encapsulation talks about security.

Beautiful definition of polymorphism:

A boy starts love with the word friendship, but girl ends love with the same word friendship, word is the same but with different attitudes. This concept is nothing but polymorphism.

Overloading :

1. **Two methods are said to be overload if and only if both having the same name but different argument types.**
2. **In 'C' language we can't take 2 methods with the same name and different types. If there is a change in argument type compulsory we should go for new method name.**
   Example :

```
abs() ———— for int type
labs() ———— for long type
fabs() ———— for float type
.
.
etc
```

3. **Lack of overloading in "C" increases complexity of the programming.**
4. **But in java we can take multiple methods with the same name and different argument types.**

Example:

abs(int)
abs(long)
abs(float)
.
.
.

5. **Having the same name and different argument types is called method overloading.**
6. **All these methods are considered as overloaded methods.**
7. **Having overloading concept in java reduces complexity of the programming.**

8. **Example:**

```
9.  class Test {
10.         public void methodOne() {
11.                 System.out.println("no-arg method");
12.         }
13.         public void methodOne(int i) {
14.          System.out.println("int-arg method"); //overloaded methods
15.         }
16.         public void methodOne(double d) {
17.          System.out.println("double-arg method");
18.         }
19.         public static void main(String[] args)       {
20.                 Test t=new Test();
21.                 t.methodOne();//no-arg method
22.                 t.methodOne(10);//int-arg method
23.                 t.methodOne(10.5);//double-arg method
24.         }
25.  }
```

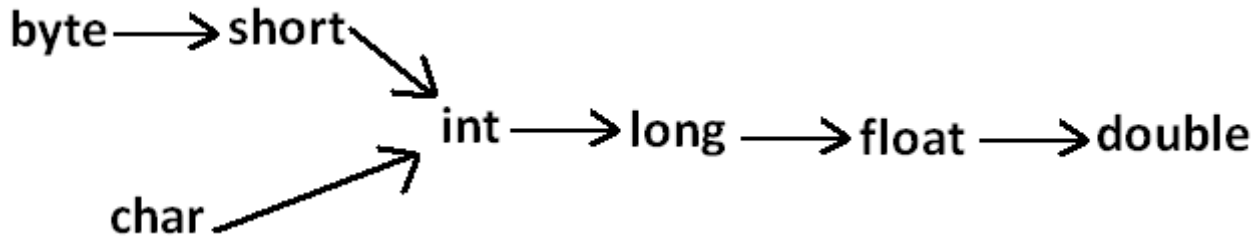
26. **<u>Conclusion :</u> In overloading compiler is responsible to perform method resolution(decision) based on the reference type(but not based on run time object). Hence overloading is also considered as compile time polymorphism(or) static polymorphism (or)early biding.**

Case 1: Automatic promotion in overloading.

- **In overloading if compiler is unable to find the method with exact match we won't get any compile time error immediately.**
- **1st compiler promotes the argument to the next level and checks whether the matched method is available or not if it is available then that method will be considered if it is not available then compiler promotes the argument once again to the next level. This process will be continued until all possible promotions still if the matched method is not available then we will get compile time error. This process is called automatic promotion in overloading.**

The following are various possible automatic promotions in overloading.

Diagram :




```
Example:
class Test
{
        public void methodOne(int i)
        {
                System.out.println("int-arg method");
        }
        public void methodOne(float f)        //overloaded methods
        {
                System.out.println("float-arg method");
        }
        public static void main(String[] args)
        {
                Test t=new Test();
                //t.methodOne('a');//int-arg method
                //t.methodOne(10l);//float-arg method
                t.methodOne(10.5);//C.E:cannot find symbol
        }
}
```

Case 2:

```
class Test
{
        public void methodOne(String s)
        {
                System.out.println("String version");
        }
        public void methodOne(Object o)         //Both methods are said to
                                                        //be
overloaded methods.
        {
```

```
                System.out.println("Object version");
        }
        public static void main(String[] args)
        {
                Test t=new Test();
                t.methodOne("arun");//String version
                t.methodOne(new Object());//Object version
                t.methodOne(null);//String version
        }
}
```

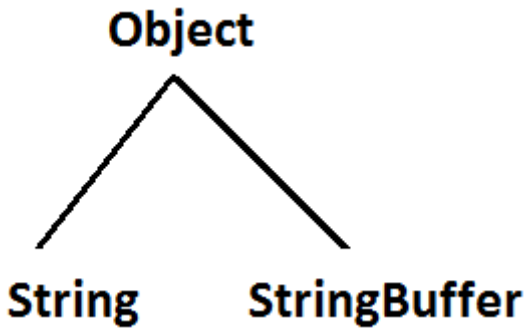
Note :

While resolving overloaded methods exact match will always get high priority,
While resolving overloaded methods child class will get the more priority than parent class

Case 3:

```
class Test{
        public void methodOne(String s)       {
                System.out.println("String version");
        }
        public void methodOne(StringBuffer s) {
                System.out.println("StringBuffer version");
        }
        public static void main(String[] args){
        Test t=new Test();
        t.methodOne("arun");//String version
        t.methodOne(new StringBuffer("sai"));//StringBuffer version
        t.methodOne(null);//CE : reference to m1() is ambiquous
        }
}
```

Output:



Case 4:

```
class Test {
public void methodOne(int i,float f) {
                System.out.println("int-float method");
        }
        public void methodOne(float f,int i)  {
                System.out.println("float-int method");
        }
        public static void main(String[] args){
                Test t=new Test();
                t.methodOne(10,10.5f);//int-float method
                t.methodOne(10.5f,10);//float-int method
                t.methodOne(10,10); //C.E:
                   //CE:reference to methodOne is ambiguous,
            //both method methodOne(int,float) in Test
```

```
                //and method methodOne(float,int) in Test match
              t.methodOne(10.5f,10.5f);//C.E:
                            cannot find symbol
                            symbol : methodOne(float, float)
                            location : class Test

        }
}
```

Case 5 :

```
class Test{
        public void methodOne(int i)   {
                System.out.println("general method");
        }
        public void methodOne(int...i) {
                System.out.println("var-arg method");
        }
        public static void main(String[] args){
                Test t=new Test();
                t.methodOne();//var-arg method
                t.methodOne(10,20);//var-arg method
                t.methodOne(10);//general method
        }
}
```

In general var-arg method will get less priority that is if no other method matched then only var-arg method will get chance for execution it is almost same as default case inside switch.

Case 6:

```
class Animal{ }
class Monkey extends Animal{}
class Test{
        public void methodOne(Animal a)       {
                System.out.println("Animal version");
        }
        public void methodOne(Monkey m)       {
                System.out.println("Monkey version");
        }
        public static void main(String[] args){
                Test t=new Test();
                Animal a=new Animal();
                t.methodOne(a);//Animal version
                Monkey m=new Monkey();
                t.methodOne(m);//Monkey version
                Animal a1=new Monkey();
                t.methodOne(a1);//Animal version
        }
}
```

In overloading method resolution is always based on reference type and runtime object won't play any role in overloading.

Overriding :

1. **Whatever the Parent has by default available to the Child through inheritance, if the Child is not satisfied with Parent class method implementation then Child is**

allow to redefine that Parent class method in Child class in its own way this process is called overriding.

2. The Parent class method which is overridden is called overridden method.
3. The Child class method which is overriding is called overriding method.

4. Example 1:
5.
6. class Parent {
7. public void property(){
8. System.out.println("cash+land+gold");
9. }
10. public void marry() {
11. System.out.println("subbalakshmi"); //overridden method
12. }
13. }
14. class Child extends Parent{ //overriding
15. public void marry() {
16. System.out.println("3sha/4me/9tara/anushka"); //overriding method
17. }
18. }
19. class Test {
20. public static void main(String[] args){
21. Parent p=new Parent();
22. p.marry();//subbalakshmi(parent method)
23. Child c=new Child();
24. c.marry();//Trisha/nayanatara/anushka(child method)
25. Parent p1=new Child();
26. p1.marry();//Trisha/nayanatara/anushka(child method)
27. }
28. }

29. In overriding method resolution is always takes care by JVM based on runtime object hence overriding is also considered as runtime polymorphism or dynamic polymorphism or late binding.
30. The process of overriding method resolution is also known as dynamic method dispatch.

Note: In overriding runtime object will play the role and reference type is dummy.

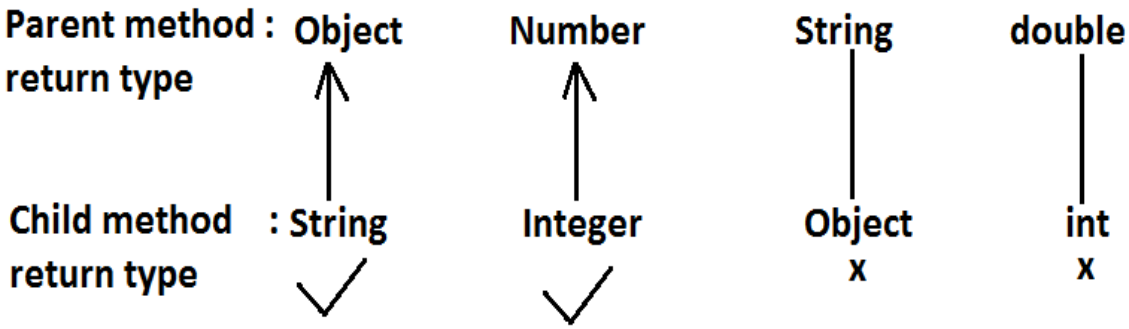
Rules for overriding :

1. **In overriding method names and arguments must be same. That is method signature must be same.**
2. **Until 1.4 version the return types must be same but from 1.5 version onwards co-variant return types are allowed.**
3. **According to this Child class method return type need not be same as Parent class method return type its Child types also allowed.**

```
4. Example:
5. class Parent {
6.   public Object methodOne()      {
7.           return null;
8.   }
9. }
10.  class Child extends Parent {
11.          public String methodOne() {
12.                  return null;
13.          }
14.  }
15.
16.  C:> javac -source 1.4  Parent.java //error
```

It is valid in "1.5" but invalid in "1.4".

Diagram:



Co-variant return type concept is applicable only for object types but not for primitives.

Private methods are not visible in the Child classes hence overriding concept is not applicable for private methods. Based on own requirement we can declare the same Parent class private method in child class also. It is valid but not overriding.

Example:

```
class Parent
{
    private void methodOne()
    {}
}
class Child extends Parent
{
    private void methodOne()
    {}
}
```

it is valid but not overriding.

Parent class final methods we can't override in the Child class.

```
17. Example:
18. class Parent {
19.         public final void methodOne()  {}
20. }
21. class Child extends Parent{
22.         public void methodOne(){}
23. }
24. Output:
```

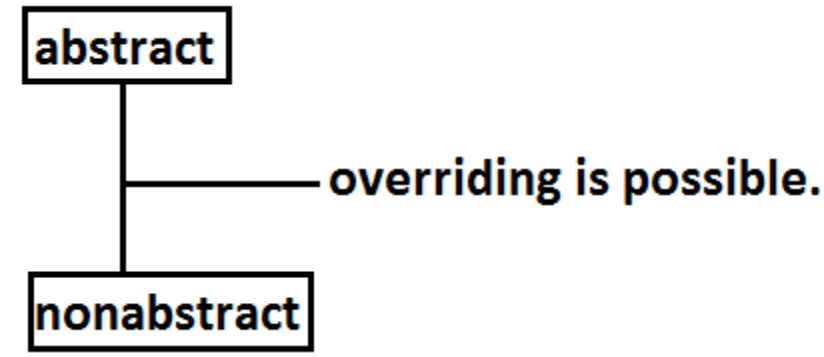
```
Compile time error:
 methodOne() in Child cannot override methodOne()
      in Parent; overridden method is final
```

Parent class non final methods we can override as final in child class. We can override native methods in the child classes.

28. We should override Parent class abstract methods in Child classes to provide implementation.

```
Example:
abstract class Parent {
        public abstract void methodOne();
}
class Child extends Parent {
        public void methodOne() { }
}
```

Diagram:



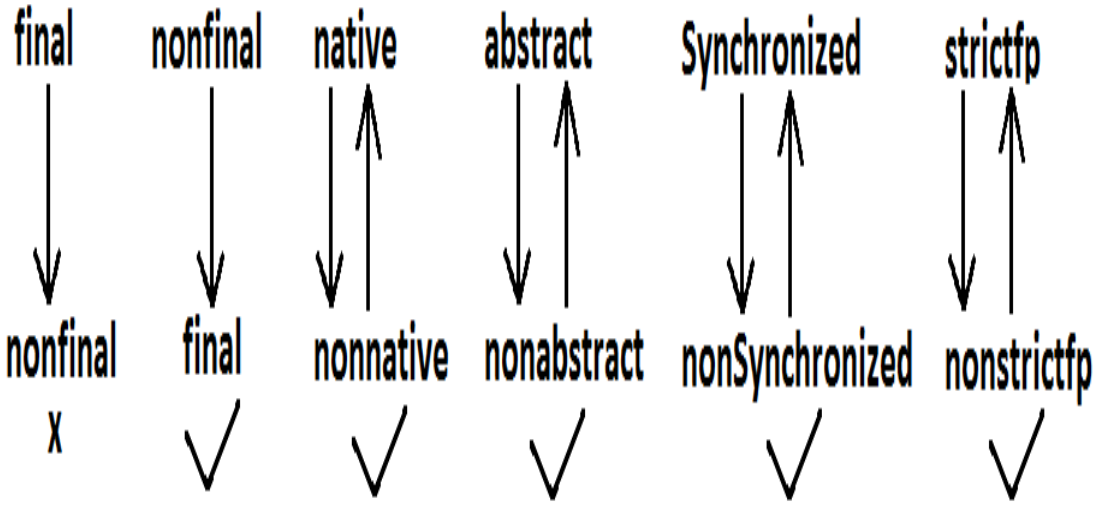
36. We can override a non-abstract method as abstract
this approach is helpful to stop availability of Parent method implementation to the next level child classes.

```
Example:
class Parent {
```

```
39.       public void methodOne()  { }
40.  }
41.  abstract class Child extends Parent {
42.       public abstract void methodOne();
43.  }
```

Synchronized, strictfp, modifiers won't keep any restrictions on overriding.

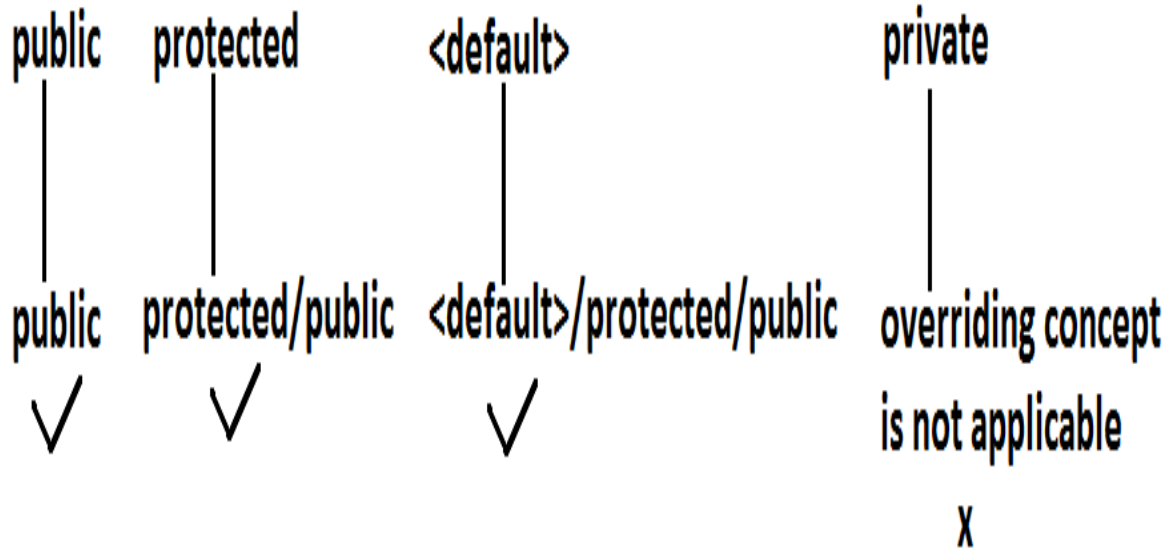
Diagram:



44. While overriding we can't reduce the scope of access modifier.

```
45.  Example:
46.  class Parent {
47.       public void methodOne() { }
48.  }
49.  class Child extends Parent {
50.       protected void methodOne( )  { }
51.  }
52.  Output:
53.  Compile time error :
54.  methodOne() in Child cannot override methodOne() in Parent;
55.   attempting to assign weaker access privileges; was public
```

Diagram:

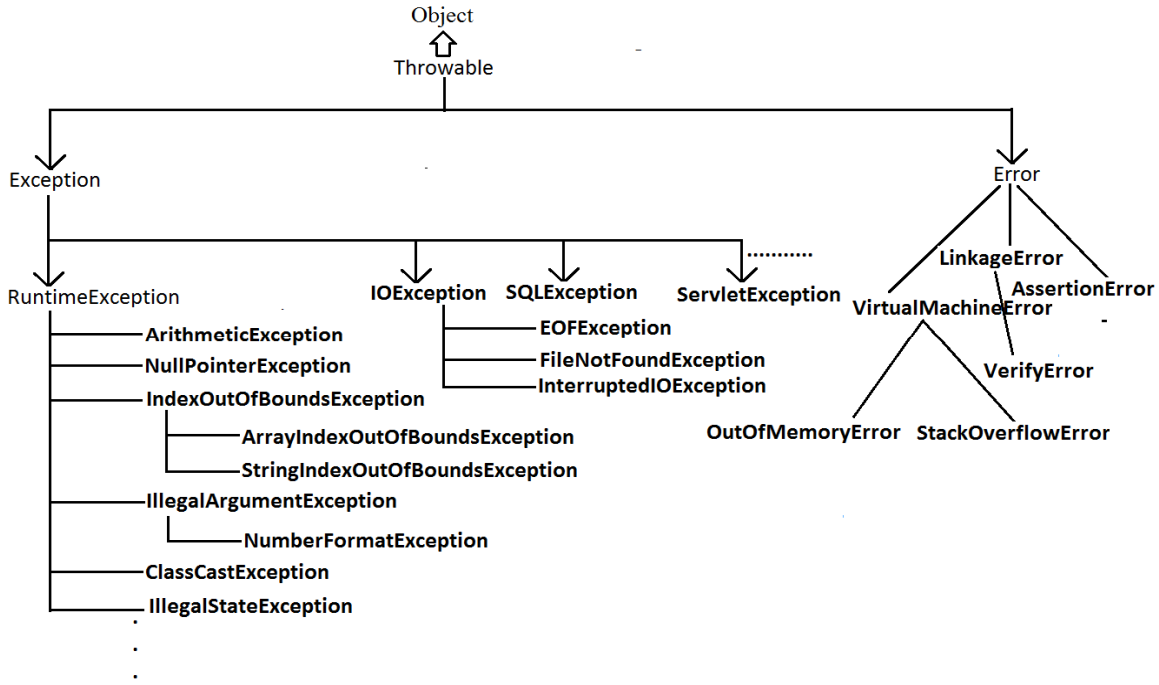


private < default < protected < public

Checked Vs Un-checked Exceptions :

- **The exceptions which are checked by the compiler for smooth execution of the program at runtime are called checked exceptions.**
- **The exceptions which are not checked by the compiler are called un-checked exceptions.**
- **RuntimeException and its child classes, Error and its child classes are unchecked except these the remaining are checked exceptions.**

Diagram:



Rule: While overriding if the child class method throws any checked exception compulsory the parent class method should throw the same checked exception or its parent otherwise we will get compile time error.

But there are no restrictions for un-checked exceptions.

```
Example:
class Parent {
     public void methodOne() {}
}
class Child extends Parent{
     public void methodOne()throws Exception {}
}
```

```
Output:
Compile time error :
methodOne() in Child cannot override methodOne() in Parent;
overridden method does not throw java.lang.Exception
```

Examples :

1. Parent: public void methodOne()throws Exception
   Child: public void methodOne()
   } valid

2. Parent: public void methodOne()
   Child : public void methodOne()throws Exception
   } invalid

3. Parent: public void methodOne()throws Exception
   Child: public void methodOne()throws Exception
   } valid

4. Parent: public void methodOne()throws IOException
   Child: public void methodOne()throws Exception
   } invalid

5. Parent: public void methodOne()throws IOException
   Child: public void methodOne()throws EOFException,FileNotFoundException
   } valid

6. Parent: public void methodOne()throws IOException
   Child : public void methodOne()throws EOFException,InterruptedException
   } invalid

7. Parent: public void methodOne()throws IOException
   Child: public void methodOne()throws EOFException,ArithmeticException
   } valid

8. Parent: public void methodOne()
   Child: public void methodOne()throws
   ArithmeticException,NullPointerException,ClassCastException,RuntimeException
   } valid

Overriding with respect to static methods:

Case 1:

We can't override a static method as non static.

```
Example:
class Parent
{
public static void methodOne(){}
   //here static methodOne() method is a class level
}
class Child extends Parent
{
public void methodOne(){}
//here methodOne() method is a object level hence
   // we can't override methodOne() method
}

output :
CE: methodOne in Child  can't  override  methodOne() in Parent ;
  overriden method is  static
```

Case 2:

Similarly we can't override a non static method as static.

Case 3:

```
class Parent
{
     public static void methodOne() {}
}
class Child extends Parent {
     public static void methodOne() {}
}
```

It is valid. It seems to be overriding concept is applicable for static methods but it is not overriding it is method hiding.

METHOD HIDING :

All rules of method hiding are exactly same as overriding except the following differences.

Overriding	Method hiding
1. Both Parent and Child class methods should be non static.	1. Both Parent and Child class methods should be static.
2. Method resolution is always takes care by JVM based on runtime object.	2. Method resolution is always takes care by compiler based on reference type.

3. Overriding is also considered as runtime polymorphism (or) dynamic polymorphism (or) late binding.	3. Method hiding is also considered as compile time polymorphism (or) static polymorphism (or) early biding.

```
Example:
class Parent {
  public static void methodOne() {
   System.out.println("parent class");
 }
}
class Child extends Parent{
 public static void methodOne(){
  System.out.println("child class");
 }
}
class Test{
 public static void main(String[] args)       {
               Parent p=new Parent();
               p.methodOne();//parent class
               Child c=new Child();
               c.methodOne();//child class
               Parent p1=new Child();
               p1.methodOne();//parent class
       }
}
```

Note: If both Parent and Child class methods are non static then it will become overriding and method resolution is based on runtime object. In this case the output is

```
Parent class
Child class
Child class
```

Overriding with respect to Var-arg methods:

A var-arg method should be overridden with var-arg method only. If we are trying to override with normal method then it will become overloading but not overriding.

```
Example:
class Parent {
public void methodOne(int... i){
System.out.println("parent class");
}
}
class Child extends Parent {   //overloading but not overriding.
public void methodOne(int i) {
System.out.println("child class");
}
}
class Test {
public static void main(String[] args) {
Parent p=new Parent();
p.methodOne(10);//parent class
Child c=new Child();
c.methodOne(10);//child class
Parent p1=new Child();
p1.methodOne(10);//parent class
```

```
}
}
```

In the above program if we replace child class method with var arg then it will become overriding. In this case the output is

```
Parent class
Child class
Child class
```

Overriding with respect to variables:

- **Overriding concept is not applicable for variables.**
- **Variable resolution is always takes care by compiler based on reference type.**

```
Example:
class Parent
{
     int x=888;
}
class Child extends Parent
{
     int x=999;
}
class Test
{
        public static void main(String[] args)
        {
                Parent p=new Parent();
                System.out.println(p.x);//888
                Child c=new Child();
                System.out.println(c.x);//999
                Parent p1=new Child();
                System.out.println(p1.x);//888
        }
}
```


Note: In the above program Parent and Child class variables, whether both are static or non static whether one is static and the other one is non static there is no change in the answer.

Differences between overloading and overriding ?

Property	Overloading	Overriding
1) Method names	Must be same.	Must be same.
2) Argument type	Must be different(at least order)	Must be same including order.
3) Method signature	Must be different.	Must be same.
4) Return types	No restrictions.	Must be same until 1.4v but from 1.5v onwards we can take co-variant return types also.
5) private, static, final methods	Can be overloaded.	Can not be overridden.
6) Access modifiers	No restrictions.	Weakering/reducing is not allowed.
7) Throws clause	No restrictions.	If child class method throws any checked exception compulsory parent class method should throw the same checked exceptions or its parent but no restrictions for un-checked exceptions.
8) Method resolution	Is always takes care by compiler based on referenced type.	Is always takes care by JVM based on runtime object.
9) Also known as	Compile time polymorphism (or) static(or)early binding.	Runtime polymorphism (or) dynamic (or) late binding.

Note:

1. **In overloading we have to check only method names (must be same) and arguments (must be different) the remaining things like return type extra not required to check.**
2. **But In overriding we should compulsory check everything like method names, arguments, return types, throws keyword, modifiers etc.**

Consider the method in parent class
Parent: public void methodOne(int i)throws IOException

In the child class which of the following methods we can take..

1. **public void methodOne(int i)//valid(overriding)**
2. **private void methodOne()throws Exception//valid(overloading)**
3. **public native void methodOne(int i);//valid(overriding)**
4. **public static void methodOne(double d)//valid(overloading)**

5. **public static void methodOne(int i)**
   Compile time error :

   methodOne(int) in Child cannot override methodOne(int) in Parent; overriding method is static

6. **public static abstract void methodOne(float f)**
   Compile time error :
   1. **illegal combination of modifiers: abstract and static**
   2. **Child is not abstract and does not override abstract method methodOne(float) in Child**

What is the difference between ArrayList l=new ArrayList() & List l=new ArrayList() ?

ArrayList al=new ArrayList(); [Child c=new Child();]	List l=new ArrayList(); [Parent p=new Child();]
If we know runtime object type exactly then we have to used this approach	If we don't know exact Runtime object type then we have to used this approach
By using child reference we can call both parent & child calss methods.	By using parent reference we can call only method available in parent class and child specific method we can't call.

- **We can use ArrayList reference to hold ArrayList object where as we can use List reference to hold any list implemented class object (ArrayList, LinkedList, Vector, Stack)**
- **By using ArrayList reference we can call both List and ArrayList methods but by using List reference we can call only List interface specific methods and we can't call ArrayList specific methods.**

IIQ : In how many ways we can create an object ? (or) In how many ways get an object in java ?

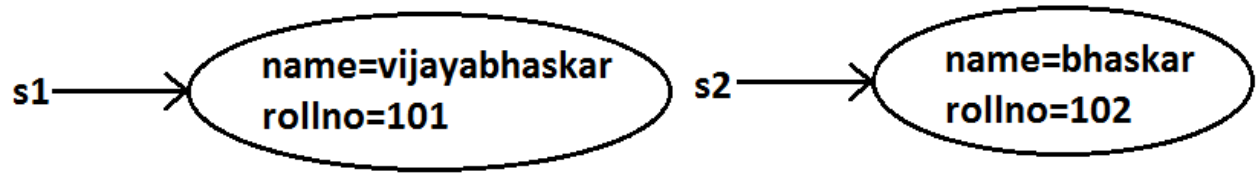
1. By using new Operator :
2. `Test t = new Test();`
3. By using newInstance() :(Reflection Mechanism)
4. `Test t=(Test)Class.forName("Test").newInstance();`
5. By using Clone() :
6. `Test t1 = new Test();`
7. `Test t2 = (Test)t1.Clone();`
8. By using Factory methods :
9. `Runtime r = Runtime.getRuntime();`
10. `DateFormat df = DateFormat.getInstance();`
11. By using Deserialization :
12. `FileInputStream fis = new FileInputStream("abc.ser");`
13. `ObjectInputStream ois = new ObjectInputStream(fis);`
14. `Test t = (Test)ois.readObject();`

Constructors :

1. Object creation is not enough compulsory we should perform initialization then only the object is in a position to provide the response properly.
2. Whenever we are creating an object some piece of the code will be executed automatically to perform initialization of an object this piece of the code is nothing but constructor.
3. Hence the main objective of constructor is to perform initialization of an object.

```
Example:
class Student
{
        String name;
        int rollno;
        Student(String name,int rollno)  //Constructor
        {
                this.name=name;
                this.rollno=rollno;
        }
        public static void main(String[] args)
        {
                Student s1=new Student("vijayabhaskar",101);
                Student s2=new Student("bhaskar",102);
        }
}
```

Diagram:



Constructor Vs instance block:

1. **Both instance block and constructor will be executed automatically for every object creation but instance block 1st followed by constructor.**
2. **The main objective of constructor is to perform initialization of an object.**
3. **Other than initialization if we want to perform any activity for every object creation we have to define that activity inside instance block.**
4. **Both concepts having different purposes hence replacing one concept with another concept is not possible.**
5. **Constructor can take arguments but instance block can't take any arguments hence we can't replace constructor concept with instance block.**
6. **Similarly we can't replace instance block purpose with constructor.**

Demo program to track no of objects created for a class:

```
class Test
{
        static int count=0;
        {
                count++;      //instance block
        }
        Test()
        {}
        Test(int i)
        {}
        public static void main(String[] args)
        {
                Test t1=new Test();
                Test t2=new Test(10);
                Test t3=new Test();
                System.out.println(count);//3
        }
}
```

Rules to write constructors:

1. **Name of the constructor and name of the class must be same.**
2. **Return type concept is not applicable for constructor even void also by mistake if we are declaring the return type for the constructor we won't get any compile time error and runtime error compiler simply treats it as a method.**

```
3. Example:
4. class Test
5. {
6.   void Test()    //it is not a constructor and it is a method
7.   {}
8. }
```

9. **It is legal (but stupid) to have a method whose name is exactly same as class name.**
10. **The only applicable modifiers for the constructors are public, default, private, protected.**
11. **If we are using any other modifier we will get compile time error.**

```
Example:
class Test
{
        static Test()
        {}
}
Output:
Modifier static not allowed here
```

Default constructor:

1. **For every class in java including abstract classes also constructor concept is applicable.**
2. **If we are not writing at least one constructor then compiler will generate default constructor.**
3. **If we are writing at least one constructor then compiler won't generate any default constructor. Hence every class contains either compiler generated constructor (or) programmer written constructor but not both simultaneously.**

Prototype of default constructor:

1. **It is always no argument constructor.**
2. **The access modifier of the default constructor is same as class modifier. (This rule is applicable only for public and default).**
3. **Default constructor contains only one line. super(); it is a no argument call to super class constructor.**

Programmers code	Compiler generated code

class Test { }	class Test {
	Test()
	{
	super();
	}
	}

public class Test { }	public class Test {
	public Test()
	{
	super();
	}
	}

class Test
{
void Test(){}
}
	class Test
	{
	Test()
	{
	super();
	}
	void Test()
	{}
	}

class Test
{
Test(int i)
{}
}
	class Test
	{
	Test(int i)
	{
	super();
	}
	}

class Test
{
Test()
{
super();
}
}
	class Test
	{
	Test()
	{
	super();
	}
	}

class Test
{
Test(int i)
{
this();
}
Test()
{}
}
	class Test
	{
	Test(int i)
	{
	this();
	}
	Test()
	{
	super();
	}
	}

super() vs this():

The 1st line inside every constructor should be either super() or this() if we are not writing anything compiler will always generate super().

Case 1: We have to take super() (or) this() only in the 1st line of constructor. If we are taking anywhere else we will get compile time error.

```
Example:
class Test
{
        Test()
        {
                System.out.println("constructor");
                super();
        }
}
Output:
Compile time error.
Call to super must be first statement in constructor
```

Case 2: We can use either super() (or) this() but not both simultaneously.

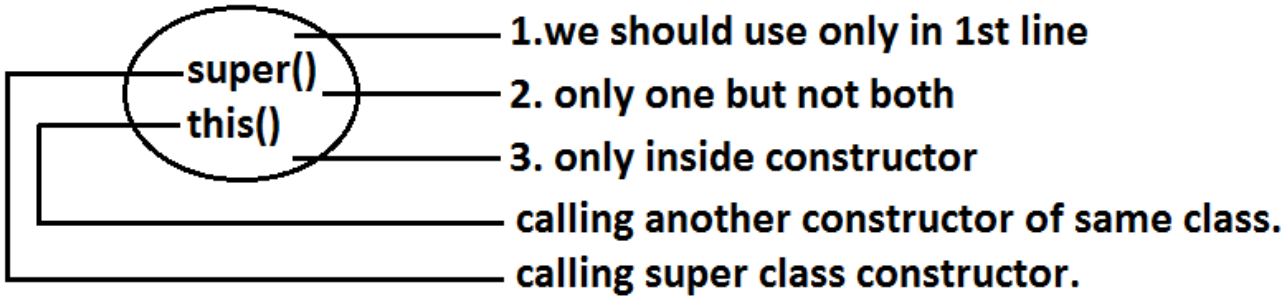
```
Example:
class Test
{
        Test()
        {
                super();
                this();
        }
}
Output:
Compile time error.
Call to this must be first statement in constructor
```

Case 3: We can use super() (or) this() only inside constructor. If we are using anywhere else we will get compile time error.

```
Example:
class Test
{
        public void methodOne()
        {
                super();
        }
}
Output:
Compile time error.
Call to super must be first statement in constructor
```

That is we can call a constructor directly from another constructor only.

Diagram:



Example:

super(), this()	super, this
These are constructors calls.	These are keywords
We can use these to invoke super class & current constructors directly	We can use refers parent class and current class instance members.
We should use only inside constructors as first line, if we are using outside of constructor we will get compile time error.	We can use anywhere (i.e., instance area) except static area , other wise we will get compile time error .

```
Example:
class Test
{
        public static void main(String[] args)
        {
                System.out.println(super.hashCode());
        }
}
Output:
Compile time error.
Non-static variable super cannot be referenced from a static context.
```

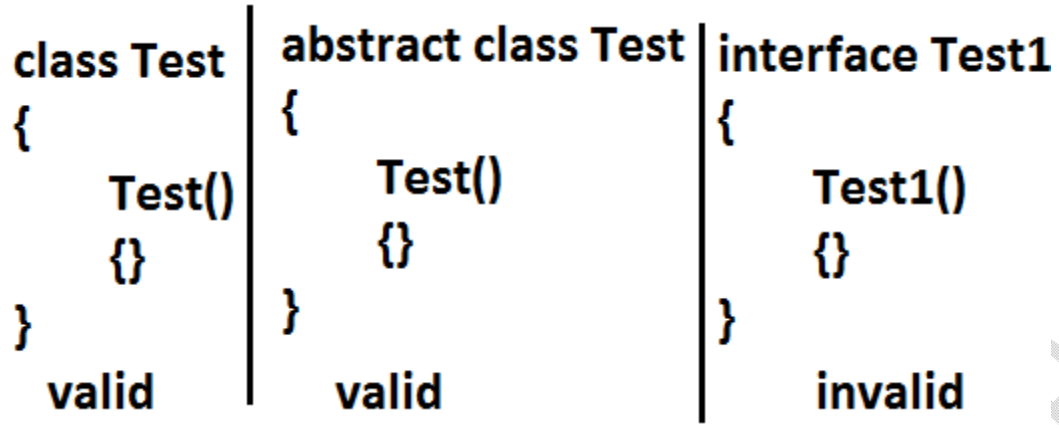
Overloaded constructors :

A class can contain more than one constructor and all these constructors having the same name but different arguments and hence these constructors are considered as overloaded constructors.

```
Example:
class Test {
 Test(double d){
   System.out.println("double-argument constructor");
        }
 Test(int i) {
        this(10.5);
        System.out.println("int-argument constructor");
        }
 Test() {
        this(10);
   System.out.println("no-argument constructor");
        }
 public static void main(String[] args) {
        Test t1=new Test();  //no-argument constructor/int-argument
                             //constructor/double-argument constructor
        Test t2=new Test(10);
                    //int-argument constructor/double-argument constructor
        Test t3=new Test(10.5);//double-argument constructor
  }
}
```


- **Parent class constructor by default won't available to the Child. Hence Inheritance concept is not applicable for constructors and hence overriding concept also not applicable to the constructors. But constructors can be overloaded.**
- **We can take constructor in any java class including abstract class also but we can't take constructor inside interface.**

Example:



We can't create object for abstract class but abstract class can contain constructor what is the need ?

Abstract class constructor will be executed for every child class object creation to perform initialization of child class object only.

Which of the following statement is true ?

1. **Whenever we are creating child class object then automatically parent class object will be created.(false)**
2. **Whenever we are creating child class object then parent class constructor will be executed.(true)**

```
Example:
abstract class Parent
{
	Parent()
	{
		System.out.println(this.hashCode());
				//11394033//here this means child class object
	}
}
class Child extends Parent
{
	Child()
	{
		System.out.println(this.hashCode());//11394033
	}
}
class Test
{
	public static void main(String[] args)
	{
		Child c=new Child();
		System.out.println(c.hashCode());//11394033
	}
}
```

<u>Case 1:</u> recursive method call is always runtime exception where as recursive constructor invocation is a compile time error.

Note:

Recursive functions:

A function is called using two methods (types).

1. **Nested call**
2. **Recursive call**

Nested call:

- **Calling a function inside another function is called nested call.**
- **In nested call there is a calling function which calls another function(called function).**

```
Example:
public static void methodOne()
        {
                methodTwo();
        }
        public static void methodTwo()
        {
                methodOne();
        }
```

Recursive call:

- **Calling a function within same function is called recursive call.**
- **In recursive call called and calling function is same.**

```
Example:
public void methodOne()
        {
                methodOne();
        }
```

Example:

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

R.E:StackOverflowError

```
class Test
{
        Test(int i)
        {
                this();
        }
        Test()
        {
                this(10);
        }
        public static void main(String[] args)
        {
                System.out.println("hello");
        }
}
```

C.E:recursive constructor invocation

Note: Compiler is responsible for the following checkings.

1. **Compiler will check whether the programmer wrote any constructor or not. If he didn't write at least one constructor then compiler will generate default constructor.**
2. **If the programmer wrote any constructor then compiler will check whether he wrote super() or this() in the 1st line or not. If his not writing any of these compiler will always write (generate) super().**
3. **Compiler will check is there any chance of recursive constructor invocation. If there is a possibility then compiler will raise compile time error.**

Case 2:

```
class Parent
{}
class Child extends Parent
{}
```
valid

```
class Parent
{
    Parent()
    {}
}
class Child extends Parent
{}
```

```
class Parent
{
    Parent(int i)
    {}
}
class Child extends Parent
{
    Child()
    {
        super();
    }
}
```
output:
compile time error

```
E:\scjp>javac Child.java
Child.java:10: cannot find symbol
symbol  : constructor Parent()
location: class Parent
        super();
```

- **If the Parent class contains any argument constructors while writing Child classes we should takes special care with respect to constructors.**
- **Whenever we are writing any argument constructor it is highly recommended to write no argument constructor also.**

Case 3:

```
class Parent
{
        Parent()throws java.io.IOException
        {}
}
class Child extends Parent
{}
Output:
Compile time error
Unreported exception java.io.IOException in default constructor.
Example:
class Parent
{
        Parent()throws java.io.IOException
        {}
}
class Child extends Parent
{
        Child()throws Exception
        {
                super();
        }
}
```

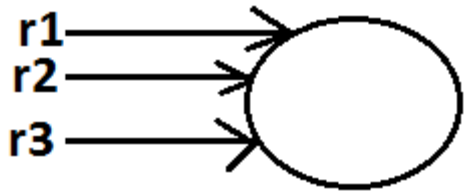
If Parent class constructor throws some checked exception compulsory Child class constructor should throw the same checked exception (or) its Parent.

Singleton classes :

For any java class if we are allow to create only one object such type of class is said to be singleton class.

```
Example:
1)      Runtime class
2)      ActionServlet
3)      ServiceLocator
4)      BusinessDelegate

Runtime r1=Runtime.getRuntime();
          //getRuntime() method is a factory method
Runtime r2=Runtime.getRuntime();
Runtime r3=Runtime.getRuntime();
.................................................
.................................................

System.out.println(r1==r2);//true
System.out.println(r1==r3);//true
```

Diagram:



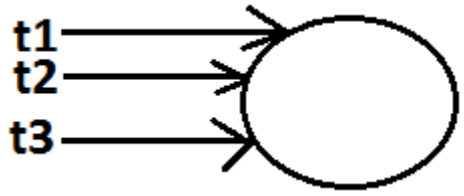
Advantage of Singleton class :
If the requirement is same then instead of creating a separate object for every person we will create only one object and we can share that object for every required person we can achieve this by using singleton classes. That is the main advantages of singleton classes are Performance will be improved and memory utilization will be improved.

Creation of our own singleton classes:

We can create our own singleton classes for this we have to use private constructor, static variable and factory method.

```
Example:
class Test
{
        private static Test t=null;
        private Test()
        {}
        public static Test getTest()
                        //getTest() method is a factory method
        {
                if(t==null)
                {
                        t=new Test();
                }
                return t;
        }
}
class Client
{
        public static void main(String[] args)
        {
                System.out.println(Test.getTest().hashCode());//1671711
                System.out.println(Test.getTest().hashCode());//1671711
                System.out.println(Test.getTest().hashCode());//1671711
                System.out.println(Test.getTest().hashCode());//1671711
        }
}
```

Diagram:



Note:

<u>We can create any xxxton classes like(double ton,trible ton...etc)</u>

```
Example:
class Test
{
        private static Test t1=null;
        private static Test t2=null;
```

```
        private Test()
        {}
        public static Test getTest()
                        //getTest() method is a factory method
        {
                if(t1==null)
                {
                        t1=new Test();
                        return t1;
                }
                else if(t2==null)
                {
                        t2=new Test();
                        return t2;
                }
                else
                {
                        if(Math.random()<0.5) //Math.random() limit : 0<=x<1
                                return t1;
                        else
                                return t2;
                }
        }
}
class Client
{
        public static void main(String[] args)
        {
                System.out.println(Test.getTest().hashCode());//1671711
                System.out.println(Test.getTest().hashCode());//11394033
                System.out.println(Test.getTest().hashCode());//11394033
                System.out.println(Test.getTest().hashCode());//1671711
        }
}
```

IIQ : We are not allowed to create child class but class is not final , How it is Possible ?

By declaring every constructor has private.

```
class Parent {
  private Parent() {
  }
```

We can't create child class for this class

Note : When ever we are creating child class object automatically parent class constructor will be executed but parent object won't be created.

```
class Parent {
 Parent() {
  System.out.println(this.hashCode()); //123
 }
}
class  Child extends Parent {
Child() {
 System.out.println(this.hashCode());//123
 }
}
class Test {
 public static void main(String ar[]) {
```

```
Child c=new Child();
System.out.println(c.hashCode());//123
```

Which of the following is true ?

1. **The name of the constructor and name of the class need not be same.(false)**
2. **We can declare return type for the constructor but it should be void. (false)**
3. **We can use any modifier for the constructor. (false)**
4. **Compiler will always generate default constructor. (false)**
5. **The modifier of the default constructor is always default. (false)**
6. **The 1st line inside every constructor should be super always. (false)**
7. **The 1st line inside every constructor should be either super or this and if we are not writing anything compiler will always place this().(false)**
8. **Overloading concept is not applicable for constructor. (false)**
9. **Inheritance and overriding concepts are applicable for constructors. (false)**
10. **Concrete class can contain constructor but abstract class cannot. (false)**
11. **Interface can contain constructor. (false)**
12. **Recursive constructor call is always runtime exception. (false)**
13. **If Parent class constructor throws some un-checked exception compulsory Child class constructor should throw the same un-checked exception or it's Parent. (false)**
14. **Without using private constructor we can create singleton class. (false)**
15. **None of the above.(true)**

Factory method:

By using class name if we are calling a method and that method returns the same class object such type of method is called factory method.

Example:

```
Runtime r=Runtime.getRuntime();//getRuntime is a factory method.
DateFormat df=DateFormat.getInstance();
```

If object creation required under some constraints then we can implement by using factory method.

Static control flow :

Example:

```
class Base
{
  ①static int i=10;                                   ——⑦
  ②static
    {
        methodOne();                                  ——⑧
        System.out.println("first static block");      ——⑩
    }
  ③public static void main(String[] args)
    {
        methodOne();                                  ——⑬
        System.out.println("main method");            ——⑮
    }
  ④public static void methodOne()
        {
            System.out.println(j);                    ——⑨,⑭
        }
     ⑤static
        {
            System.out.println("second static block"); ——⑪
        }
     ⑥static int j=20;                                ——⑫
  }
```

Analysis:

i=0[RIWO]
j=0[RIWO]
i=10[R&W]
j=20[R&W]

1.identification of static members from top to bottom[1 to 6]
2.execution of static variable assignments and static blocks from top to bottom[7 to 12]
3.execution of main method[13 to 15]

```
Output:
E:\scjp>javac Base.java
E:\scjp>java Base
0
First static block
Second static block
```

```
20
Main method
```

Read indirectly write only state (or) RIWO :

With in the static block if we are trying to read any variable then that read is considered as "direct read"
If we are calling a method , and with in the method if we are trying to read a method , that read is called Indirect read

If a variable is in RIWO state then we can't perform read operation directly otherwise we will get compile time error saying " illegal forward reference ".

Example:

```
class Test
{
    static int i=10;
    static
    {
        System.out.println(i);//10
        System.exit(0);
    }
}
```

```
class Test
{
    static
    {
        System.out.println(i);
    }
    static int i=10;
}
output:
compile time error:
illegal forward reference
```

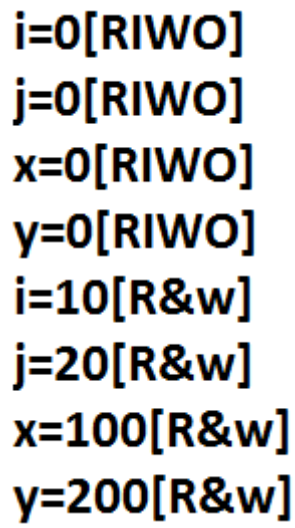
```
class Test
{
    static
    {
        methodOne();
    }
    public static void methodOne()
    {
        System.out.println(i);
    }
    static int i=10;
}
output:
Runtime exception:
0
NoSuchMethodError: main
```

Static control flow parent to child relationship :

```
class Base
{
 1 static int i =10;                                   12
 2 static
   {
       methodOne();                                    13
       System.out.println("base static block");        15
   }
 3 public static void main(String[] args)
   {
       methodOne();
       System.out.println("base main");
   }
 4 public static void methodOne()
       {
           System.out.println(j);                      14
       }
   5 static int j=20;                                  16
}
class Derived extends Base
{
 6 static int x =100;                                  17
 7 static
   {
       methodTwo();                                    18
       System.out.println("derived first static block");   20
   }
 8 public static void main(String[] args)
   {
       methodTwo();                                    23
       System.out.println("derived main");             25
   }
 9 public static void methodTwo()
       {
           System.out.println(y);                      19,24
       }
   10 static
       {
           System.out.println("derived second  static block");   21
       }
   11 static int y=200;                                22
}
```


DURGASOFT, # 202,2nd Floor,HUDA Maitrivanam,Ameerpet, Hyderabad - 500038,
☎ 040 – 64 51 27 86, 80 96 96 96 96, 9246212143 | www.durgasoft.com

Analysis:




```
Output:
E:\scjp>java Derived
0
Base static block
0
Derived first static block
Derived second static block
200
Derived main
Output:
E:\scjp>java Base
0
Base static block
20
Basic main
```

Whenever we are executing Child class the following sequence of events will be performed automatically.

1. **Identification of static members from Parent to Child. [1 to 11]**
2. **Execution of static variable assignments and static blocks from Parent to Child.[12 to 22]**
3. **Execution of Child class main() method.[23 to 25].**

Note : When ever we are loading child class autimatically the parent class will be loaded but when ever we are loading parent class the child class don't be loaded automatically.

Static block:

- **Static blocks will be executed at the time of class loading hence if we want to perform any activity at the time of class loading we have to define that activity inside static block.**
- **With in a class we can take any no. Of static blocks and all these static blocks will be executed from top to bottom.**

Example:

The native libraries should be loaded at the time of class loading hence we have to define that activity inside static block.

```
Example:
class Test
{
        static
        {
                System.loadLibrary("native library path");
        }
}
```

Ex 2 : Every JDBC driver class internally contains a static block to register the driver with DriverManager hence programmer is not responsible to define this explicitly.

```
Example:
class Driver
{
        static
        {
                //Register this driver with DriverManager
        }
}
```

IIQ : Without using main() method is it possible to print some statements to the console?
Ans : Yes, by using static block.

```
Example:
class Google
{
        static
        {
                System.out.println("hello i can print");
                System.exit(0);
        }
}
Output:
Hello i can print
```

IIQ : Without using main() method and static block is it possible to print some statements to the console ?

```
Example 1:
class Test
{
        static int i=methodOne();
        public static int methodOne()
        {
```

```
System.out.println("hello i can print");
                System.exit(0);
                return 10;
        }
}
Output:
Hello i can print
```

Example 2:

```
class Test
{
        static Test t=new Test();
        Test()
        {
                System.out.println("hello i can print");
                System.exit(0);
        }
}
Output:
Hello i can print
```

Example 3:

```
class Test
{
        static Test t=new Test();
        {
        System.out.println("hello i can print");
        System.exit(0);
        }
}
Output:
Hello i can print
```

IIQ : Without using System.out.println() statement is it possible to print some statement to the console ?

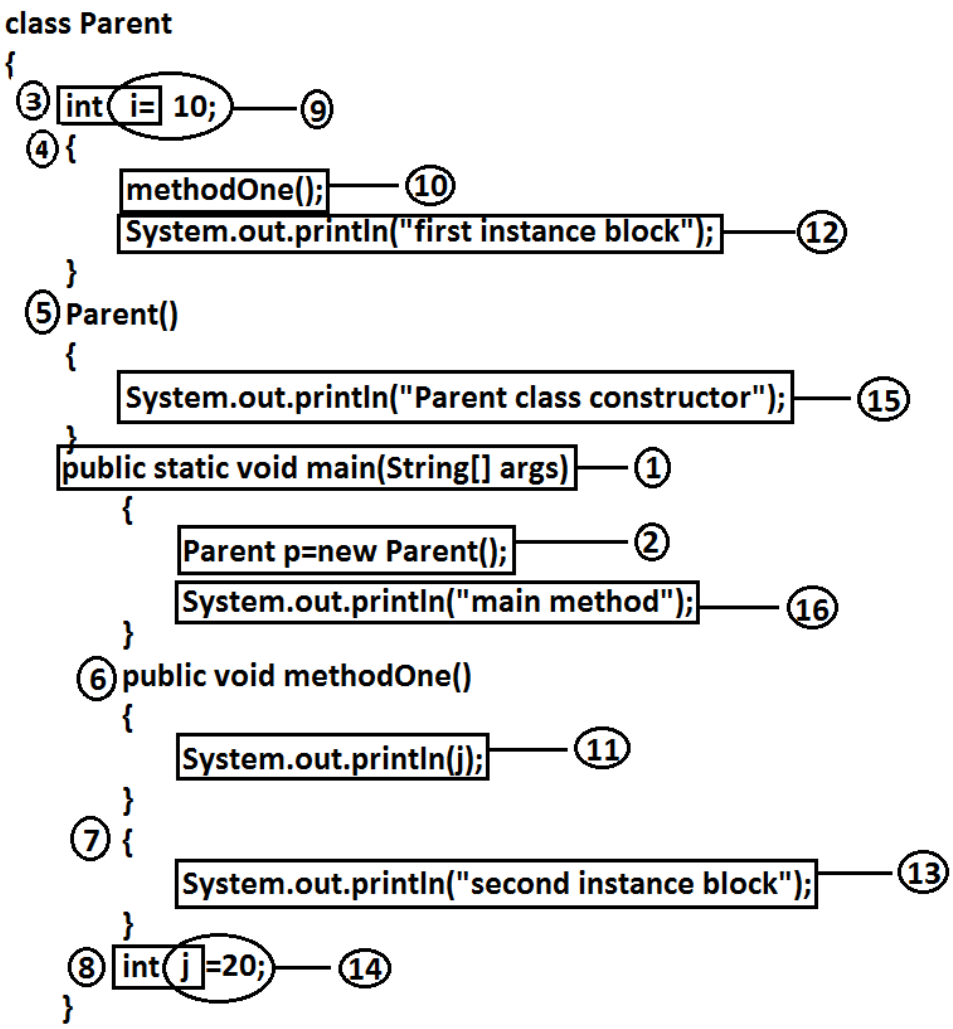
```
Example:
class Test
{
        public static void main(String[] args)
        {
                System.err.println("hello");
        }
}
```

Note : Without using main() method we can able to print some statement to the sonsole , but this rule is applicable untill 1.6 version from 1.7 version onwards to run java program main() method is mandatory.

```
class Test {
 static {
  System.out.println("ststic block");
  System.exit(0);
  }
 }
```

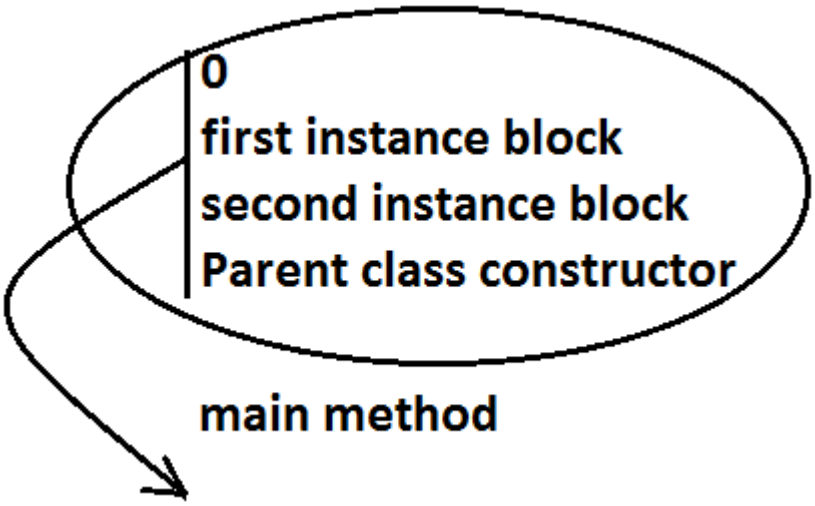
It is valid in 1.6 version but invalid or won't run in 1.7 version

Instance control flow:




```
Analysis:
i=0[RIWO]
j=0[RIWO]
i=10[R&W]
j=20[R&W]
```

Output:



Whenever we are executing a java class static control flow will be executed. In the Static control flow Whenever we are creating an object the following sequence of events will be performed automatically.

1. **Identification of instance members from top to bottom(3 to 8).**
2. **Execution of instance variable assignments and instance blocks from top to bottom(9 to 14).**
3. **Execution of constructor.**

Note: static control flow is one time activity and it will be executed at the time of class loading.
But instance control flow is not one time activity for every object creation it will be executed.

Instance control flow in Parent to Child relationship :

```
Example:
class Parent
{
        int x=10;
        {
                methodOne();
                System.out.println("Parent first instance block");
        }
        Parent()
        {
                System.out.println("parent class constructor");
        }
        public static void main(String[] args)
        {
                Parent p=new Parent();
                System.out.println("parent class main method");
        }
        public void methodOne()
        {
                System.out.println(y);
        }
        int y=20;
```

```
}
class Child extends Parent
{
        int i=100;
        {
                methodTwo();
                System.out.println("Child first instance block");
        }
        Child()
        {
                System.out.println("Child class constructor");
        }
        public static void main(String[] args)
        {
                Child c=new Child();
                System.out.println("Child class main method");
        }
        public void methodTwo()
        {
                System.out.println(j);
        }
        {
                System.out.println("Child second instance block");
        }
        int j=200;
}

Output:
E:\scjp>javac Child.java
E:\scjp>java Child
0
Parent first instance block
Parent class constructor
0
Child first instance block
Child second instance block
Child class constructor
Child class main method
```

Whenever we are creating child class object the following sequence of events will be executed automatically.

1. **Identification of instance members from Parent to Child.**
2. **Execution of instance variable assignments and instance block only in Parent class.**
3. **Execution of Parent class constructor.**
4. **Execution of instance variable assignments and instance blocks in Child class.**
5. **Execution of Child class constructor.**

Note: Object creation is the most costly operation in java and hence if there is no specific requirement never recommended to crate objects.

```
Example 1:
public class Initilization
{
        private static String methodOne(String msg)  //-->1
        {
                System.out.println(msg);
```

```
                return msg;
        }
        public Initilization()  //-->4
        {
                m=methodOne("1");  //-->9
        }
        {                               //-->5
                m=methodOne("2");               //-->7
        }
        String m=methodOne("3");        //-->6 , //-->8
        public static void main(String[] args) //-->2
        {
                Object obj=new Initilization();   //-->3
        }
}
```

Analysis:

1
3
2
m=methodOne()[RIWO]


```
Output:
2
3
1
Example 2:
public class Initilization
{
        private static String methodOne(String msg)  //-->1
        {
                System.out.println(msg);
```

```
                return msg;
        }
        static String m=methodOne("1");  //-->2,      //-->5
        {
                m=methodOne("2");
        }
        static                          //-->3
        {
                m=methodOne("3");                       //-->6
        }
        public static void main(String[] args)  //-->4
        {
                Object obj=new Initilization();
        }
}
Output:
1
3
2
```

We can't access instance variables directly from static area because at the time of execution of static area JVM may not identify those members.

Example:

```
class Test
{
      int i=10;
      public static void main(String[] args)
      {
           System.out.printl(i);
      }
} output:
  compile time error
  non-static variable i cannot be referenced from a static context
```

- **But from the instance area we can access instance members directly.**
- **Static members we can access from anywhere directly because these are identified already at the time of class loading only.**

Type casting:

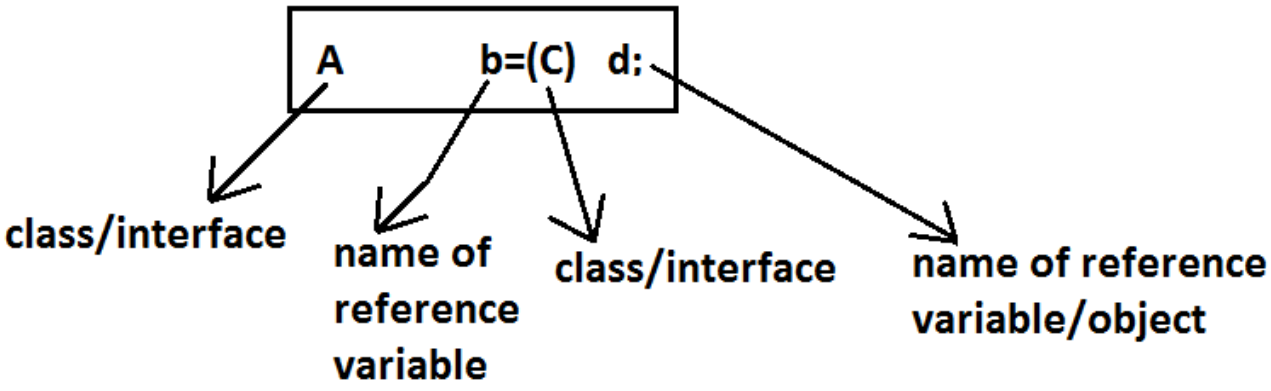
Parent class reference can be used to hold Child class object but by using that reference we can't call Child specific methods.

```
Example:
Object o=new String("ashok");//valid
System.out.println(o.hashCode());//valid
```

```
System.out.println(o.length());//
        C.E:cannot find symbol,
          symbol  : method length(),
        location: class java.lang.Object
```

Similarly we can use interface reference to hold implemented class object.

Example:
Runnable r=new Thread();
Type casting syntax:



Compile time checking :

Rule 1: The type of "d" and "c" must have some relationship [either Child to Parent (or) Parent to Child (or) same type] otherwise we will get compile time error saying inconvertible types.

Example 1:

```
Object o=new String("bhaskar");
StringBuffer sb=(StringBuffer)o;
```
(valid)

Example 2:

```
String s=new String("bhaskar");
StringBuffer sb=(StringBuffer)s;
```
(inval id)

output:

```
compile time error
E:\scjp>javac Test.java
Test.java:6: inconvertible types
found   : java.lang.String
required: java.lang.StringBuffer
        StringBuffer sb=(StringBuffer)s;
```

Rule 2: "C" must be either same (or) derived type of "A" otherwise we will get compile time error saying incompatible types.
Found: C
Required: A
Example 1:

```
Object o=new String("bhaskar");
StringBuffer sb=(StringBuffer)o;
```
(valid)

Example 2:

```
Object o=new String("bhaskar");
StringBuffer sb=(String)o;
```
(invalid)

output:

```
compile time error
E:\scjp>javac Test.java
Test.java:6: incompatible types
found   : java.lang.String
required: java.lang.StringBuffer
        StringBuffer sb=(String)o;
```

Runtime checking :

The underlying object type of "d" must be either same (or) derived type of "C" otherwise we will get runtime exception saying ClassCastException.

Example:

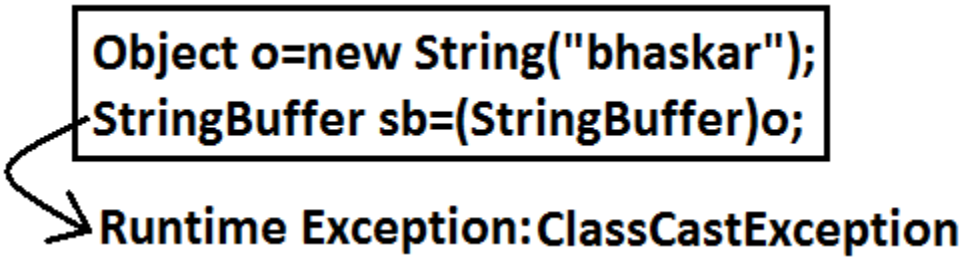
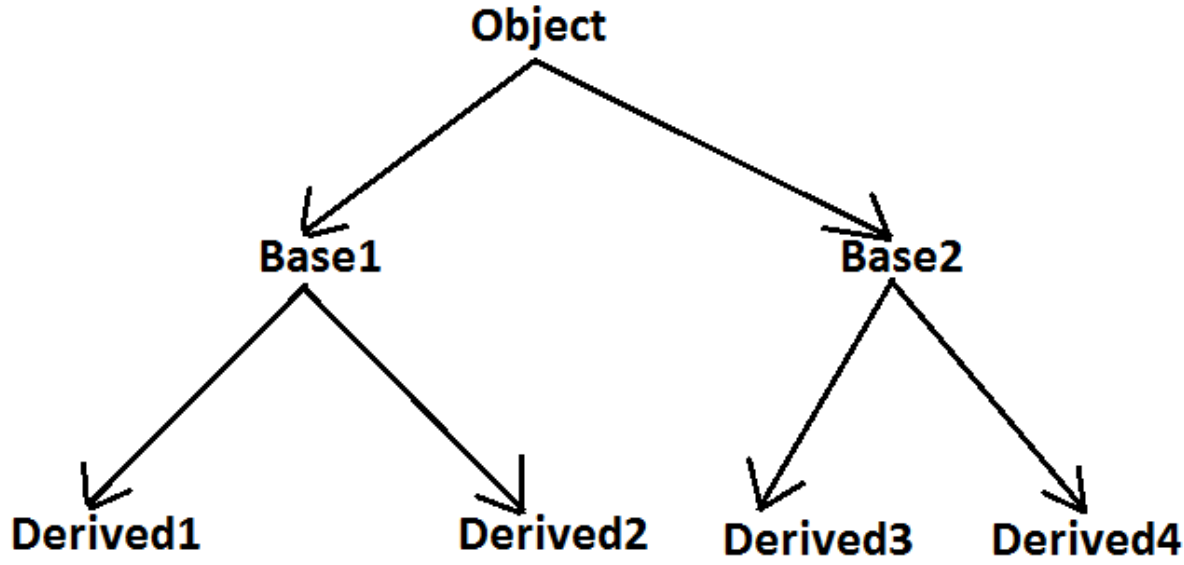


Diagram:

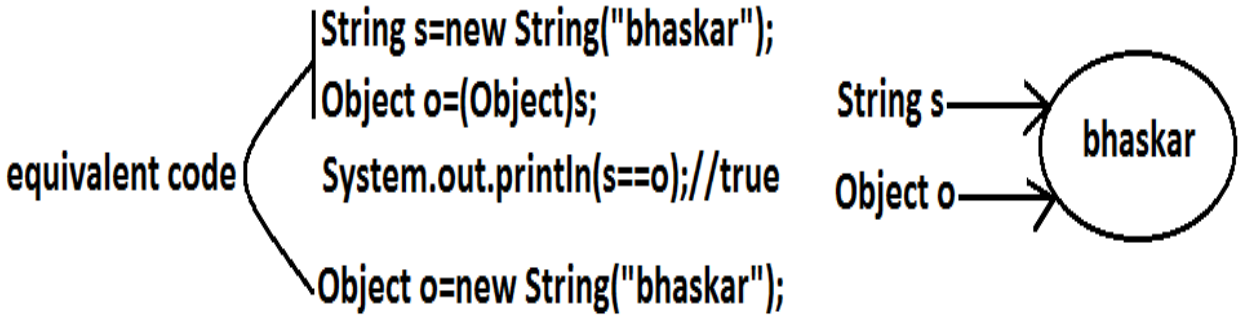



```
Base1 b=new Derived2();//valid
Object o=(Base1)b;//valid
Object o1=(Base2)o;//invalid
Object o2=(Base2)b;//invalid
Base2 b1=(Base1)(new Derived1());//invalid
Base2 b2=(Base2)(new Derived3());//valid
Base2 b2=(Base2)(new Derived1());//invalid
```

Through Type Casting just we are converting the type of object but not object itself that is we are performing type casting but not object casting.

Through Type Casting we are not create any new objects for the existing objects we are providing another type of reference variable(mostly Parent type).

Example:



Example 1:

Parent →

```
class Parent
{
    public void methodOne()
    {
        System.out.println("A");
    }
}
```

Child →

```
class Child extends Parent
{
    public void methodOne()
    {
        System.out.println("B");
    }
    public void methodTwo()
    {
        System.out.println("C");
    }
    public static void main(String[] args)
    {
        Child c=new Child();
        c.methodOne();//B
        c.methodTwo();//C
        ((Parent)c).methodOne();//B
    }
}
```

```
Parent p=new Parent();
((Parent)c).methodTwo();
```

output:

compile time error.

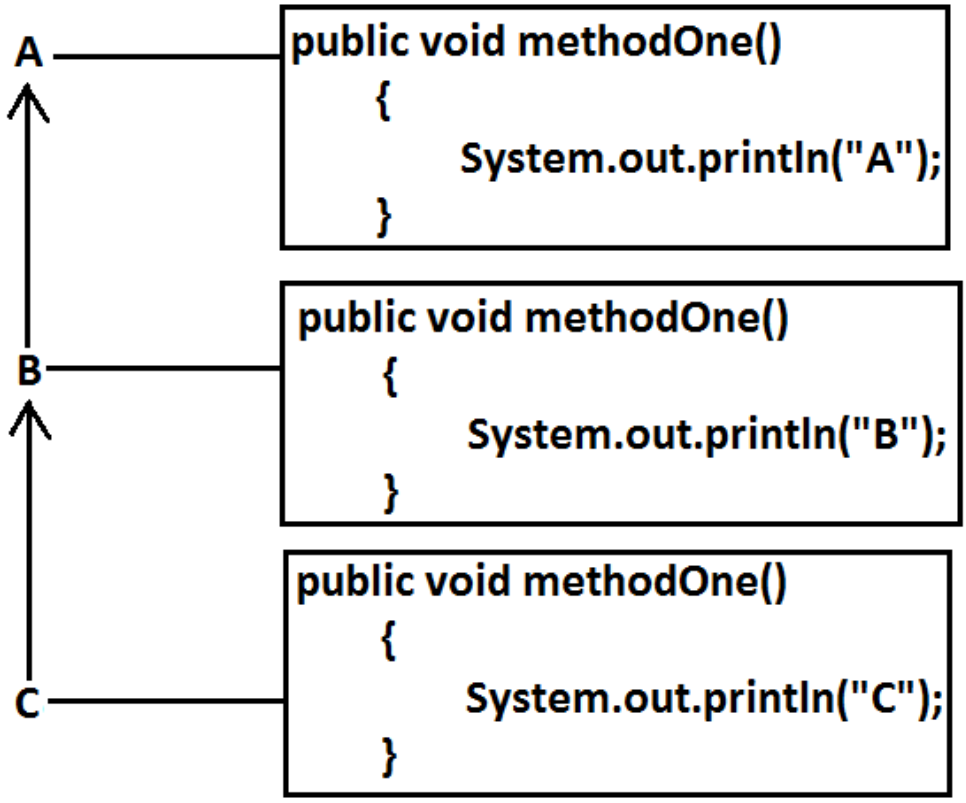
Child.java:25: cannot find symbol

symbol : method methodTwo()

location: class Parent

((Parent)c).methodTwo();

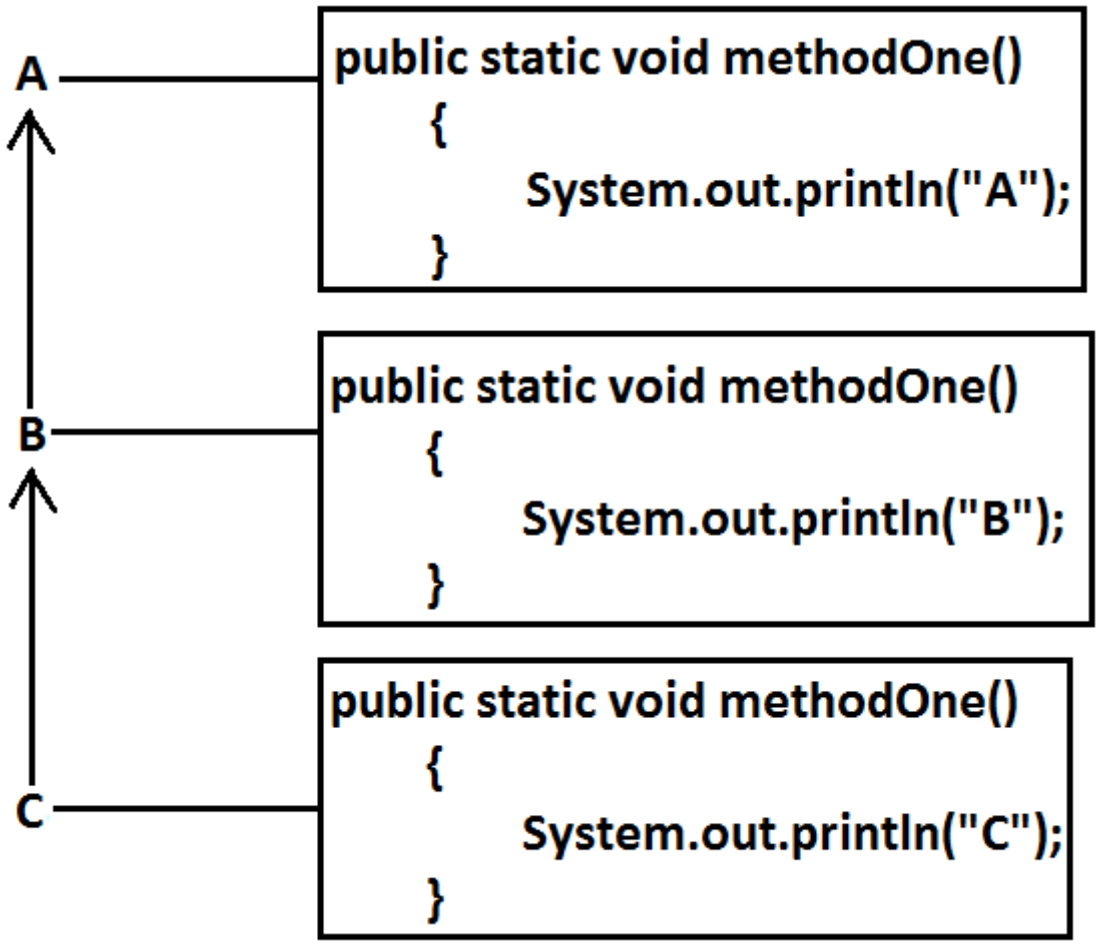
Example 2:



It is overriding and method resolution is based on runtime object.

```
C c=new C();
c.methodOne();//c
((B)c).methodOne();//c
((A)((B)c)).methodOne();//c
```

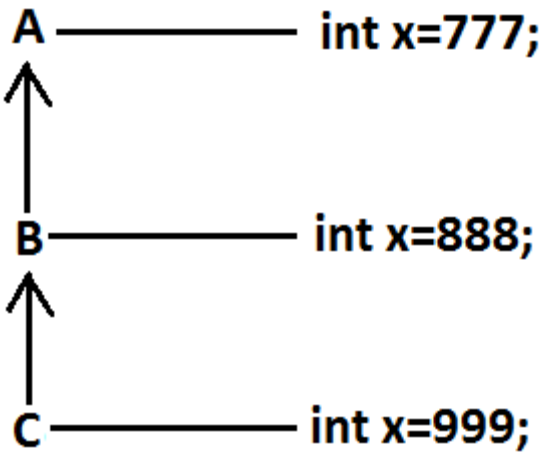
Example 3:



It is method hiding and method resolution is based on reference type.

```
C c=new C();
c.methodOne();//C
((B)c).methodOne();//B
((A)((B)c)).methodOne();//A
```

Example 4:



```
C c=new C();
System.out.println(c.x);//999
System.out.println(((B)c).x);//888
System.out.println(((A)((B)c)).x);//777
```

- **Variable resolution is always based on reference type only.**
- **If we are changing variable as static then also we will get the same output.**

Coupling :

The degree of dependency between the components is called coupling.

```
Example:
class A
{
        static int i=B.j;
}
class B extends A
{
        static int j=C.methodOne();
}
class C extends B
{
        public static int methodOne()
        {
                return D.k;
        }
}
class D extends C
{
        static int k=10;
        public static void main(String[] args)
        {
                D d=new D();
        }
}
```

The above components are said to be tightly coupled to each other because the dependency between the components is more.

Tightly coupling is not a good programming practice because it has several serious disadvantages.

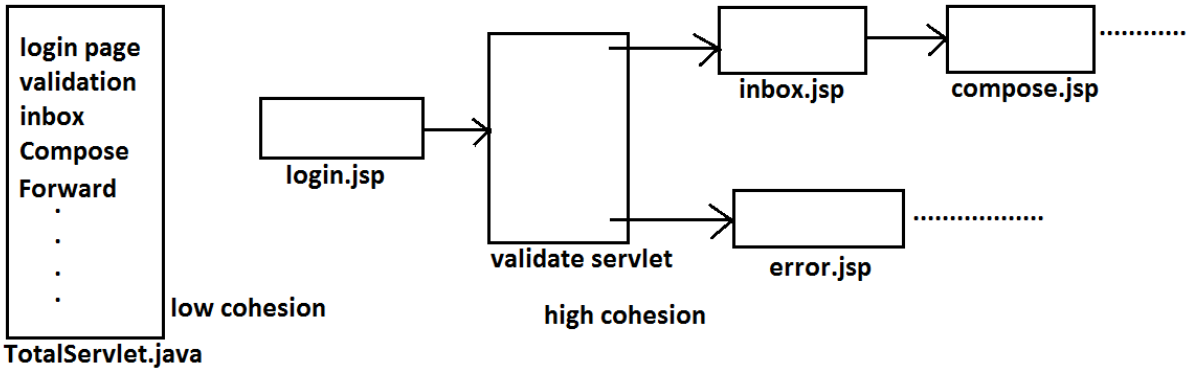
1. **Without effecting remaining components we can't modify any component hence enhancement(development) will become difficult.**
2. **It reduces maintainability of the application.**
3. **It doesn't promote reusability of the code.**

It is always recommended to maintain loosely coupling between the components.

Cohesion:

For every component we have to maintain a clear well defined functionality such type of component is said to be follow high cohesion.

Diagram:



High cohesion is always good programming practice because it has several advantages.

1. **Without effecting remaining components we can modify any component hence enhancement will become very easy.**
2. **It improves maintainability of the application.**
3. **It promotes reusability of the application.(where ever validation is required we can reuse the same validate servlet without rewriting)**

Note: It is highly recommended to follow loosely coupling and high cohesion.