1.Language Fundamentals

1. Identifiers
2. Reserved words
3. Data types
4. Literals
5. Arrays
6. Types of variables
7. Var arg method
8. Main method
9. Command line arguments
10. Java coding standards

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

Example:

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

1 — Test
2 — main
3 — String[]
4 — args
5 — x

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) _
5) $

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.

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;
int NuMbEr=30;
}
```

we can differentiate with case.

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

Example 2:

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

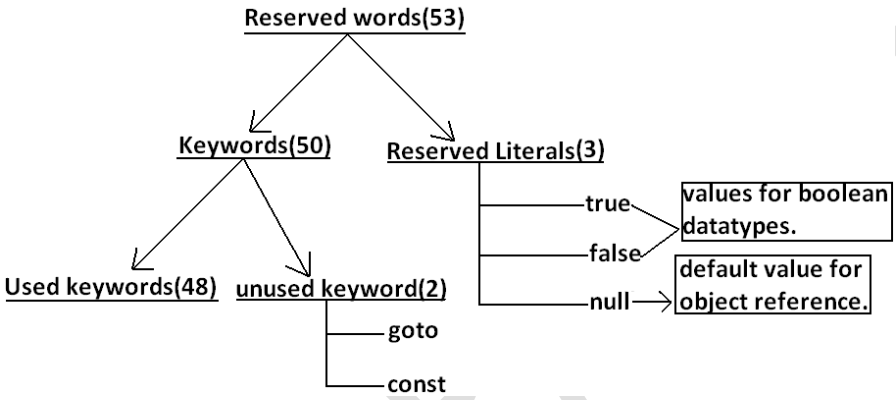
Output:
10
Even though it is legal to use class names and interface names as identifiers but it is not a good programming practice.
Which of the following are valid java identifiers?

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

Reserved words:

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

Diagram:



Reserved words for data types:

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

Reserved words for flow control:

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:

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:

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

Class related keywords:

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

Object related keywords:

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;
}
```

Note: All reserved words in java contain only lowercase alphabet symbols.

New keywords are:

Strictfp----------1.2

Assert----------1.4

Enum----------1.5

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
5) byte, short, Integer, long(invalid)//here Integer is a wrapper class
6) extends, implements, imports(invalid)//imports keyword is not available in java
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)

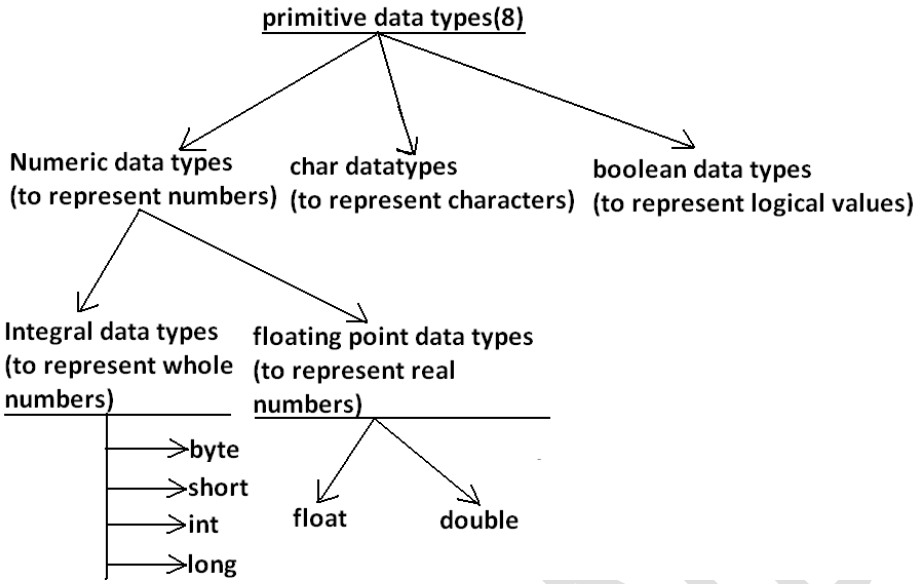
Which of the following are valid java keywords?

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

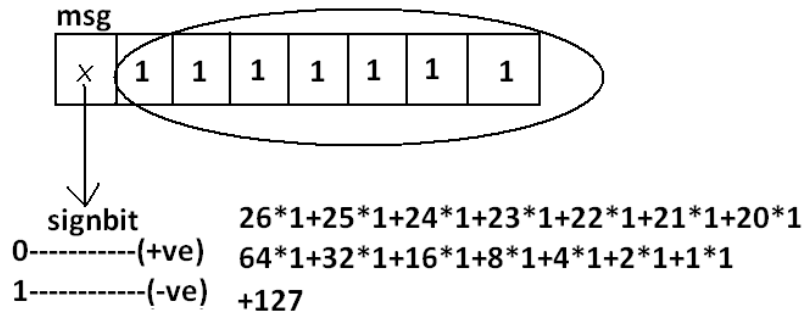
Data types: Every variable has a type, every expression has a type and all types are strictly define more over every assignment should be checked by the compiler by the type compatibility hence java language is considered as strongly typed language.

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.
  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
byte b=10.5;//C.E:possible loss of precision
byte b=true;//C.E:incompatible types
byte b="durga";//C.E:incompatible types

- 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
  short s=true;//C.E:incompatible types
- 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

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

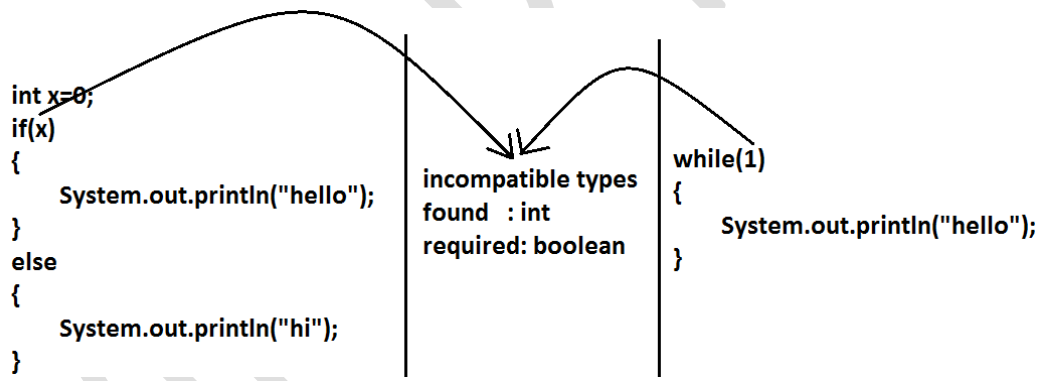
  Size: 8 bytes
  Range:-2^{63} to $2^{63}-1$

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
1) If we want to 5 to 6 decimal places of accuracy then we should go for float.	1) If we want to 14 to 15 decimal places of accuracy then we should go for double.
2) Size:4 bytes.	2) Size:8 bytes.
3) Range:-3.4e38 to 3.4e38.	3) -1.7e308 to1.7e308.
4) float follows single precision.	4) 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
Example 2:



Char data type:

- In java we are allowed to use any worldwide alphabets character and java is Unicode based 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

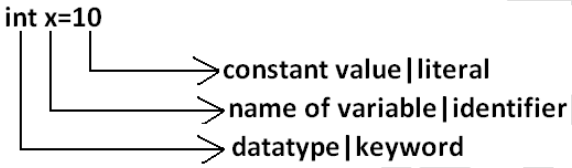
Int	4 bytes	-2^{31} to $2^{31}-1$ (-2147483648 to 2147483647)	Integer	0
Long	8 bytes	-2^{63} to $2^{63}-1$	Long	0
Float	4 bytes	-3.4e38 to 3.4e38	Float	0.0
Double	8 bytes	-1.7e308 to 1.7e308	Double	0.0
Boolean	Not applicable	Not applicable(but allowed values true|false)	Boolean	False
Char	2 bytes	0 to 65535	Character	0(represents blank space)

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

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 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?

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

Example:

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

- 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)
```

- 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=10;(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.

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)
  double d=0x123.456;//C.E:malformed floating point literal(invalid)
  Which of the following floating point declarations are valid?

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)
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 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)
  float f=10e2F;(valid)

Boolean literals: The only allowed values for the boolean type are true (or) false where case is important.
Example:

1) boolean b=true;(valid)
2) boolean b=0;//C.E:incompatible types(invalid)
3) boolean b=True;//C.E:cannot find symbol(invalid)
4) boolean b="true";//C.E:incompatible types(invalid)

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'.
   Example:
1) char ch1='\u0061';
   System.out.println(ch1);//a
2) char ch2=\u0062;//C.E:cannot find symbol
3) char ch3='\iface';//C.E:illegal escape character
4) Every escape character in java acts as a char literal.

Example:

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)
5) char ch='/n';//C.E:unclosed character literal(invalid)
6) none of the above.(valid)

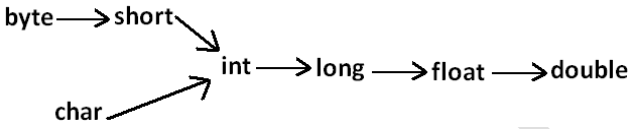
String literals:

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

Example:

String s="bhaskar";(valid)

Diagram:



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.

- 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[][];
int[] []a;
int[] a[];
int []a[];

All are valid.

Three dimensional array declaration:

Example:

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

int[] []a;
int[] a[]; **All are valid.**
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:<identifier> expected(invalid)

- 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:

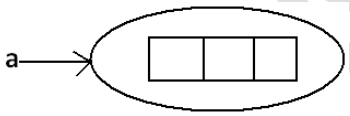
```
int[] []a,[]b;
      |  └──invalid
      └─────valid
```

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
.	.
.	.

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:

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)

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)

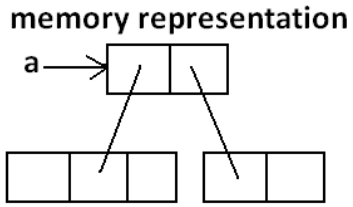
Two dimensional array creation:

- In java multidimensional arrays are implemented as array of arrays approach but not matrix form.
- 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];
```

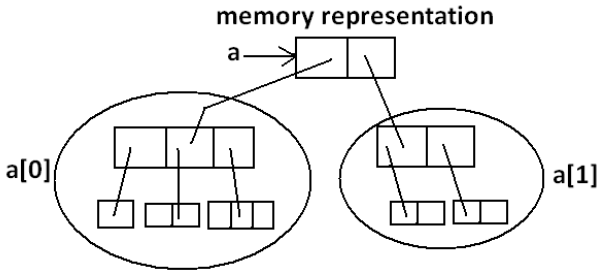
Diagram:



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?

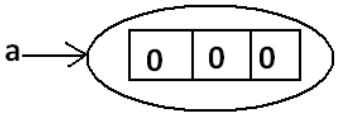
1) int[] a=new int[]//C.E: array dimension missing(invalid)
2) int[][] a=new int[3][4];(valid)
3) int[][] a=new int[3][];(valid)
4) int[][] a=new int[][4];//C.E:']' expected(invalid)
5) int[][][] a=new int[3][4][5];(valid)
6) int[][][] a=new int[3][4][];(valid)
7) int[][][] a=new int[3][][5];//C.E:']'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

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

Example 1:

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

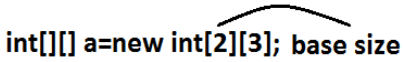
Diagram:



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.

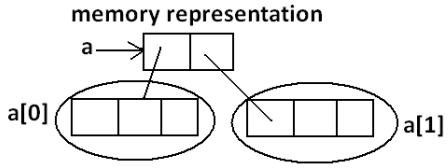
classname@hexadecimalstringrepresentationofhashcode.

Example 2:



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

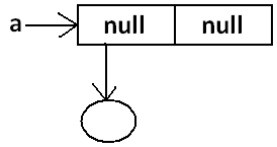
Diagram:



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:

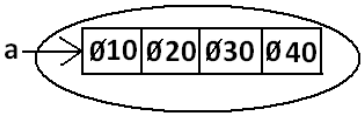


- Once we created an array all its elements by default initialized with default values. If we are not satisfied with those default values then we can replays with our customized values.

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

Diagram:

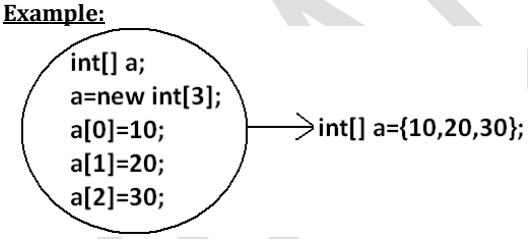


Note: if we are trying to access array element with out of range index we will get Runtime Exception saying ArrayIndexOutOfBoundsException.

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:



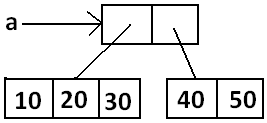
char[] ch={'a','e','i','o','u'};(valid)
String[] s={"balayya","venki","nag","chiru"};(valid)

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

Example:

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

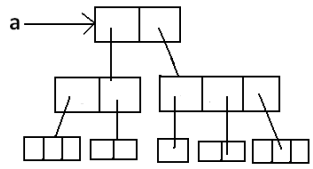
Diagram:



Example:

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

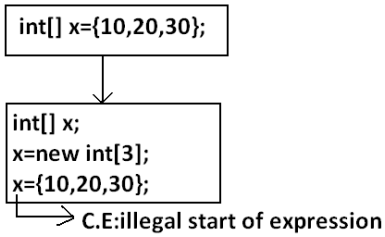
Diagram:




```
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:

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:

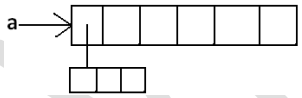
```
String s="bhaskar";
System.out.println(s.length);//C.E:cannot find symbol
System.out.println(s.length());//7
```

- 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.

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:

```
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;
	}
}
```

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.

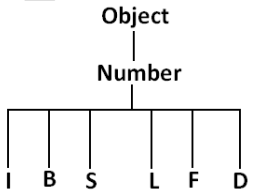
Example 1:

```
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:

```
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.
3) Interface type arrays.	3) Its implemented class objects.
4) Abstract class type arrays.	4) Its child class objects are allowed.

Array variable assignments:

Case 1:

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

Example:

```
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:

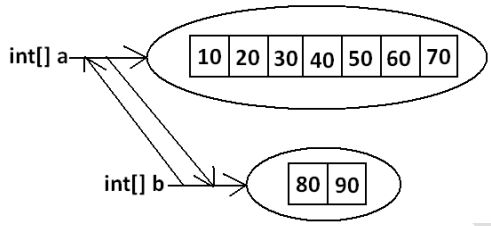
String[] s={"A","B"};
Object[] o=s;

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:

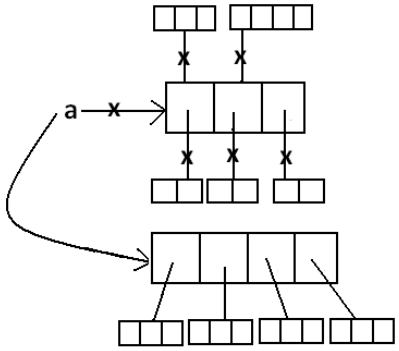
int[][] a=new int[3][];
a[0]=new int[4][5];//C.E:incompatible types(invalid)
a[0]=10;//C.E:incompatible types(invalid)
a[0]=new int[4];//(valid)

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:

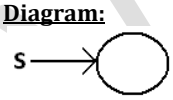
2
A
B

Types of Variables

- Based the type of value represented by the 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:



- Based on the purpose 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)
	}
}
```

- 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.

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.
5) Execution of main() method.
6) Unload Test.class
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:

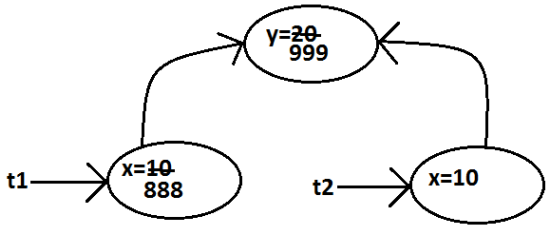
```
class Test
{
```

```
        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
        }
}
```

Diagram:



- Static variables also known as class level variables or fields.

Local variables:

- Some time 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.
- 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;
                }
                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);
```

C.E

```
javac Test.java
Test.java:11: cannot find symbol
symbol  : variable i
location: class Test
```

```
        }
    }
}
```

- 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.

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 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
    }
}
```

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.

Example:

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

    public int x=10;
            private int x=10;
            protected int x=10;
            static int x=10;
            volatile int x=10;
            transient int x=10;
            final int x=10;//(valid)
    }
}
```

C.E: illegal start of expression

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.

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);
System.out.println(a[0]);
```

C.E: variable a might not have been initialized

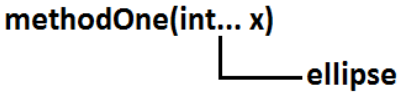
Example 2:

```
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.

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.

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 different 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
```

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...)(invalid)
4) methodOne(int. ..x)(invalid)
5) methodOne(int .x..)(invalid)

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

Example:

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

Case 4: We can take only one var-arg parameter inside var-arg method

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
{
	public static void methodOne(int... i)    int[] i
	{
		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.

methodOne(int[] i) ⇨ methodOne(int... i) (valid)

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(int... i) ⇨ methodOne(int[] i) (invalid)

Example:

```
class Test
```

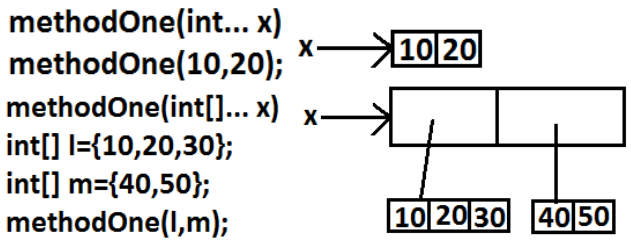
```
{
    public static void methodOne(int[]... x)
    {
        for(int[] a:x)
        {
            System.out.println(a[0]);
        }
    }
    public static void main(String[] args)
    {
        int[] l={10,20,30};
        int[] m={40,50};
        methodOne(l,m);

    }
}
```

Output:

10
40

Analysis:



Main Method

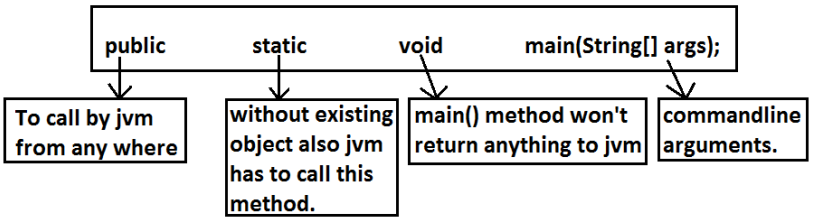
- Whether the class contains main() method or not and whether it is properly declared or not these checking's are not responsibilities of the compiler, at runtime JVM is responsible for this. If jvm unable to find the required main() method then we will get runtime exception saying NoSuchMethodError: main.

  Example:

  class Test
  {}

  Output:

  javac Test.java
  java Test **R.E: NoSuchMethodError: main**
- JVM always searches for the main() method with the following signature.



- If we are performing any changes to the above signature then the code won't run and will get Runtime exception saying NoSuchMethodError. Anyway 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
1) String[] args
2) String []args
3) 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.

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)

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

- No case, in all the cases we will get runtime exception.
- 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");
    }
    public static void main(int[] args)
    {
        System.out.println("int[] array main method");
    }
}
```

overloaded methods

Output:

String[] array main method

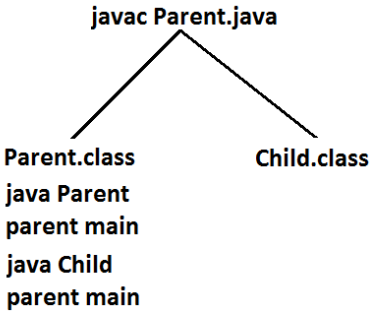
- The other overloaded method we have to call explicitly then only it will be executed.
- Inheritance concept is applicable for static methods including main() method hence while executing child class if the child class doesn't contain main() method then the parent class main() method will be executed.

Example 1:

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

Parent.java

Analysis:

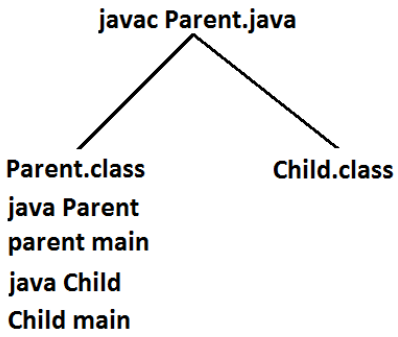


Example 2:

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

Parent.java

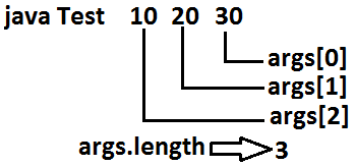
Analysis:



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

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
```

Example 2:

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

```
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:

```
class Test
{
	public static void main(String[] args)
	{
		System.out.println(args[0]);
	}
}
```

Output:

```
E:\SCJP>javac Test.java
E:\SCJP>java Test "vijaya bhaskar"
Vijaya bhaskar
```

Java coding standards

- It is highly recommended to follow coding standards.
- Whenever we are writing any component 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:

String,Customer,Object,Student,StringBuffer → nouns

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

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.

Example:

run()
sleep()
eat()
drink()
} verb

getName()
setSalary()
} nouns

verb+noun

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.

Example:

length
name
salary
age
mobileNumber

nouns

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:

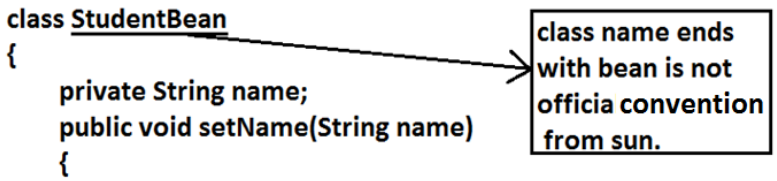
MAX_VALUE
MIN_VALUE

nouns

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.

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 delete MyActionListener(MyActionListener l)(invalid)

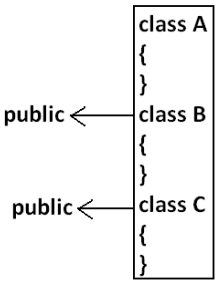
Declaration and Access Modifiers

1) Java source file structure
2) Class modifiers
3) Member modifiers
4) Interfaces

Java source file structure:

- A java program can contain any no. Of classes but at mot 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:



Case1:

- 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
Bhaskar.java

case2:

- 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".

Cae3:

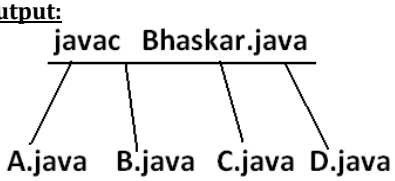
- If both B and C classes are declared as public and name of the file is B.java then we will get compile time error saying "class C is public, should be declared in a file named C.java".
- It is highly recommended to take only one class for source file and name of the program (file) must be same as class 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 Bhaskar
Exception in thread "main" java.lang.NoClassDefFoundError: Bhaskar

- 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: Bhaskar".

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();
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.
- Bet suitable for Ameerpet where typing is important.

Case2:

Which of the following import statements are valid?

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

Case3:

- 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.

Case4:

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.

Case5:

- 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:

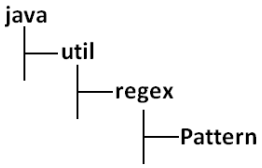
```
import java.util.Date;
import java.sql.*;
class Test
{
public static void main(String args[]){
Date d=new Date();
}}
```

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

Case6:

- 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)

Case8:

- "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.

- 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".

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.

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

- 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:

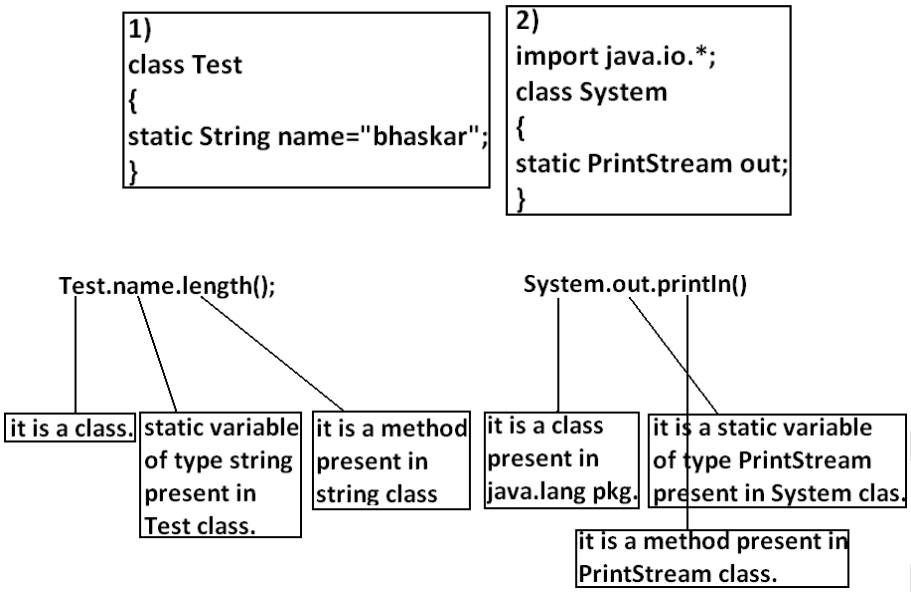
```
import static java.lang.Math.sqrt;
import static java.lang.Math.*;
class Test
{
public static void main(String args[]){
System.out.println(sqrt(4));
System.out.println(max(10,20));
System.out.println(random());
}}
```

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:



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:

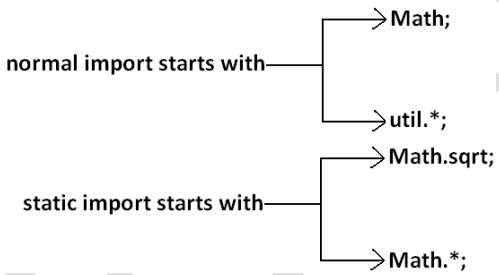
```
//import static java.lang.Integer.MAX_VALUE;——> line2
import static java.lang.Byte.*;
class Test
{
//static int MAX_VALUE=999;——>line1
public static void main(String args[])throws Exception{
System.out.println(MAX_VALUE);
}
}
```

- 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:



- 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.

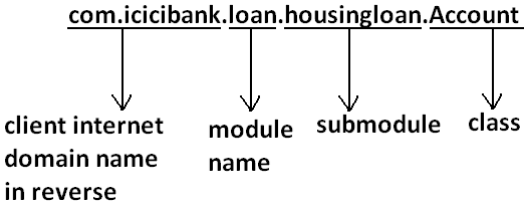
Package statement:

- 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:



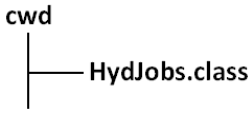
How to compile package program:

Example:

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

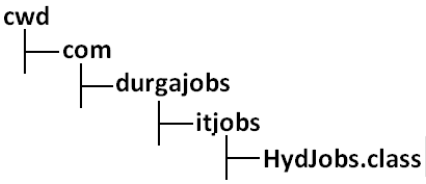
}

- 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.

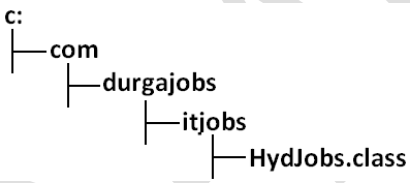
Diagram:



- 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 cement statement should be package statement [if it is available] otherwise we will get compile time error.
  Example:

```
import java.util.*;
package pack1;
class A
{
}
```

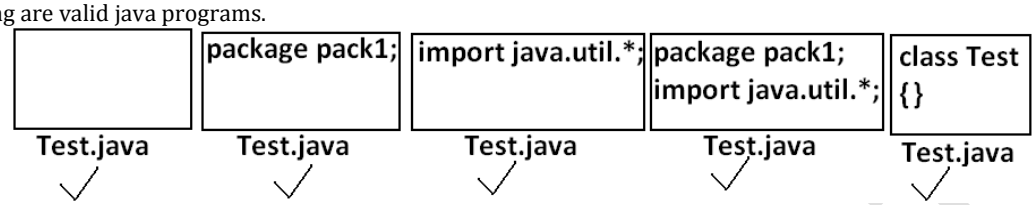
Output:

```
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) Better this class can be accessible from anywhere or not.
2) Better 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:

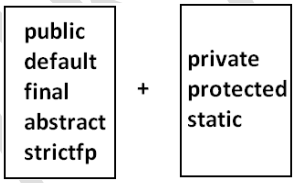
```
private class Test
{
public static void main(String args[]){
int i=0;
for(int j=0;j<3;j++)
{
i=i+j;
}
System.out.println(i);
}}
```

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.

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.

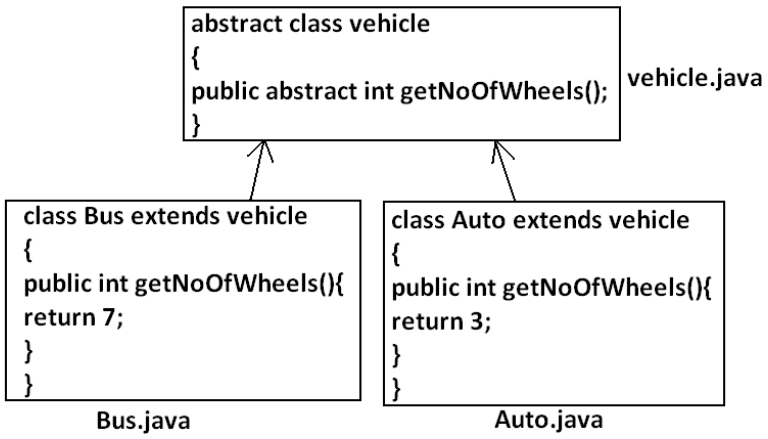
EXAMPLE:

```
public abstract void methodOne();  ------> valid
public abstract void methodOne(){} ------> invalid
```

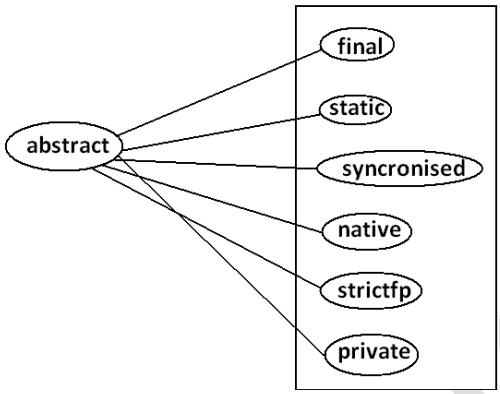
- Child classes are responsible to provide implementation for parent class abstract methods.

EXAMPLE:

PROGRAM:



- 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 it is always illegal combination.
- 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.
- 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 flat from 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.

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 language.

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	✗	✗	✗	✓

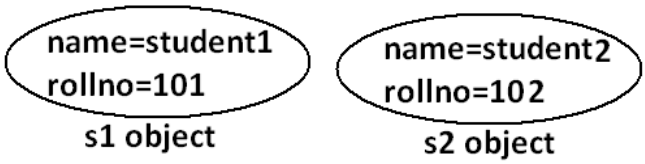
- The least accessible modifier is private.
- The most accessible modifier is public.
- Private<default<protected<public.
- Recommended modifier for variables is private where as recommended modifier for methods is public.

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 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.**
- 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.

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 access 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

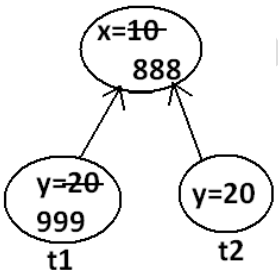
```
output:
compile time error.
D:\Java>javac Test.java
Test.java:6: final parameter x may not be assigned
x=100;
```

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(){
```

- Overloading concept is applicable for static method including main method also.

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.

- 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:

```
javac Parent.java
Parent.class  Child.class
java Parent
D:\Java>java Parent
parent main() method called
D:\Java>java child
child main() method called
```

- It seems to be overriding concept is applicable for static methods but it is not overriding it is method hiding.
- For static methods compulsory implementation should be available where as for abstract methods implementation should be available **hence abstract static combination is illegal for methods.**

Native modifier:

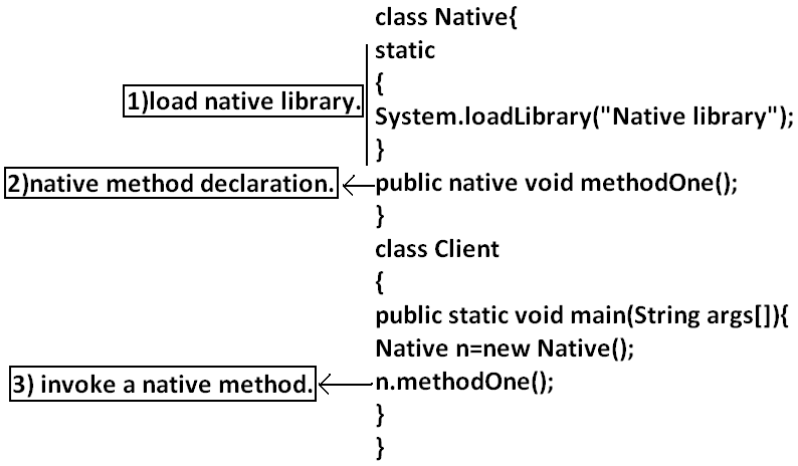
- Native is a modifier applicable only for methods but not for variables and classes.
- The methods which are implemented in non java are called native methods or foreign methods.

The main objectives of native keyword are:

- To improve performance of the system.
- To use already existing legacy non java code.

To 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 disadvantage of native keyword is usage of native keyword in java breaks platform independent nature of java language.

Synchronized:

- Synchronized is the modifier applicable for methods and blocks but not for variables and classes.
- 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.
- The main advantage of synchronized keyword is we can resolve data inconsistency problems, 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.

Transient modifier:

- Transient is the modifier applicable only for variables but not for methods and classes.
- 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.
- 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".
- 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.
- 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:

- Volatile is the modifier applicable only for variables but not for classes and methods.
- If the value of variable keeps on changing such type of variables we have to declare with volatile modifier.
- 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.
- Once the value got finalized before terminating the thread that final value will be updated in master copy.
- 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.
- Volatile means the value keep on changing where as final means the value never changes hence final volatile combination is illegal for variables.

Modifier	Classes		Methods	Variables	Blocks	Interfaces	Enum	Constructors
	Outer	Inner						
Public	✓	✓	✓	✓	✗	✓	✓	✓
Private	✗	✓	✓	✓	✗	✗	✗	✓

Protected	✗	✓	✓	✓	✗	✗	✗	✓
Default	✓	✓	✓	✓	✗	✓	✓	✓
Final	✓	✓	✓	✓	✗	✗	✗	✗
Abstract	✓	✓	✓	✗	✗	✓	✗	✗
Strictfp	✓	✓	✓	✗	✗	✓	✓	✗
Static	✗	✓	✓	✓	✓	✗	✗	✗
Synchronized	✗	✗	✓	✗	✓	✗	✗	✗
Native	✗	✗	✓	✗	✗	✗	✗	✗
Transient	✗	✗	✗	✓	✗	✗	✗	✗
Volatile	✗	✗	✗	✓	✗	✗	✗	✗

Summary of modifier:

- The modifiers which are applicable for inner classes but not for outer classes are **private, protected, static**.
- The modifiers which are applicable only for methods **native**.
- The modifiers which are applicable only for variables **transient and volatile**.
- The modifiers which are applicable for constructor public, private, protected, default.
- The only applicable modifier for local variables is **final**.

Interfaces:

1) Introduction
2) Interface declarations and implementations.
3) Extends vs implements
4) Interface methods
5) Interface variables
6) Interface naming conflicts

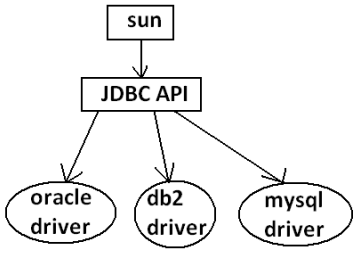
a) Method naming conflicts
b) Variable naming conflicts

7) Marker interface
8) Adapter class
9) Interface vs abstract class vs concrete class.
10) Difference between interface and abstract class?
11) Conclusions

Def1: Any service requirement specification (srs) is called an interface.

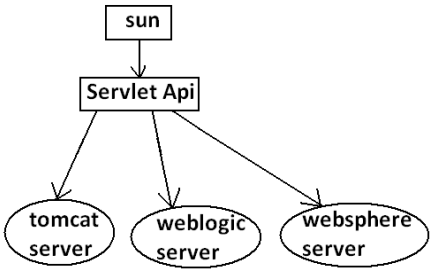
Example1: Sun people responsible to define JDBC API and database vendor will provide implementation for that.

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 his excepting. 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 his excepting 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.

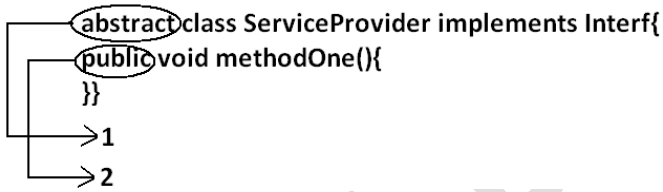
Declaration and implementation of an interface:

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 extend 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 an interface 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
{
}
```

1) 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. None of the above.

Ans: 5

2) 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

3) X extends Y, Z?

- X, Y, Z should be interfaces.

4) X extends Y implements Z?

- X, Y should be classes.
- Z should be interface.

5) X implements Y, Z?

- X should be class.
- Y, Z should be interfaces.

6) 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{
```

- 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();

- 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 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;
public static int x=10;
public final int x=10;
static final int x=10;
public static final int x=10;
```

Equal

- 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 object gets some ability such type of interfaces are called Marker interface (or) Tag interface (or) Ability interface.

Example:

Serilizable
cloneable
RandomAccess — These are marked for some ability
SingleThreadModel
.
.

.
.

Example 1: By implementing Serilizable 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 will provide required ability.

Why JVM is providing the required ability?

- To reduce complexity of the programming.

Is it possible to create our own marker interface?

- Yes, but customization of JVM is required.

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.

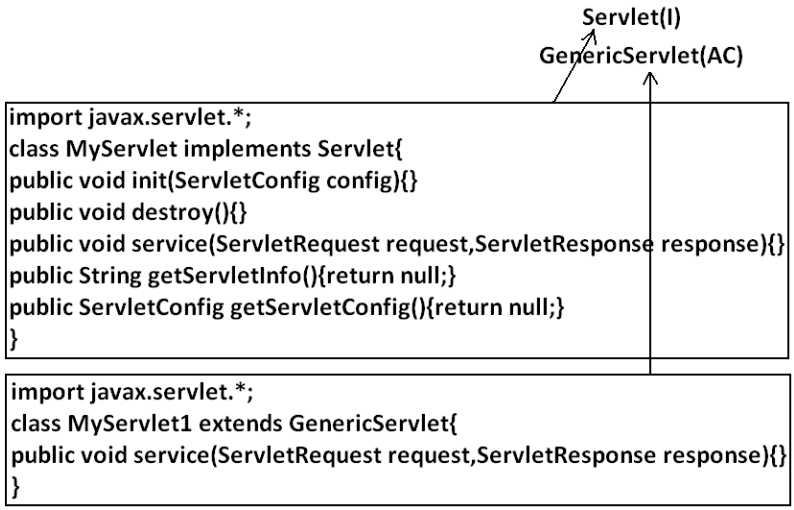
Example 1:

```
abstract class AdapterX implements X{
public void m1(){}
public void m2(){}
public void m3(){}
public void m4(){}
 //.
 //.
 //.
public void m1000(){}
}
```

Example 2:

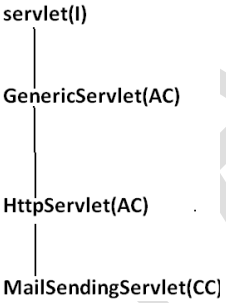
```
public class Test extend AdapterX{{
public void m3(){
}}
```

Example:



- Generic Servlet simply acts as an adapter class for Servlet interface.
  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
1)	If we don't' know anything about implementation just we have requirement specification then we should go for interface.	1)	If we are talking about implementation but not completely (partial implementation) then we should go for abstract class.
2)	Every method present inside interface is always **public and abstract** whether we are declaring or not.	2)	Every method present inside abstract class **need not be public and abstract**.
3)	We can't declare interface methods with the modifiers **private**, **protected**, **final**, **static**, **synchronized**, **native**, **strictfp**.	3)	There are no restrictions on abstract class method modifiers.
4)	Every interface variable is always **public static final** whether we are declaring or not.	4)	Every abstract class variable need not be public static final.
5)	Every interface variable is always **public static final** we can't declare with the following modifiers. **Private, protected, transient, volatile.**	5)	There are no restrictions on abstract class variable modifiers.
6)	For the interface variables compulsory we should perform initialization at the time of declaration otherwise we will get compile time error.	6)	It is not require to perform initialization for abstract class variables at the time of declaration.
7)	Inside interface we can't take static and instance blocks.	7)	Inside abstract class we can take both static and instance blocks.
8)	Inside interface we can't take constructor.	8)	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?

- This constructor will be executed for the 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());
}
}
```

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.

Operator and assignments

1) Increment and decrement operator
2) Arithmetic operators
3) String concatenation operator
4) Relational operators
5) Equality operator
6) Instanceof operator
7) Bitwise operators
8) Short circuit operators
9) Type cast operator
10) Assignment operator
11) Conditional operator
12) new operator
13) [] operator
14) Java operator precedence
15) Evaluation order of java operands

Increment and decrement operator:

Diagram 1:

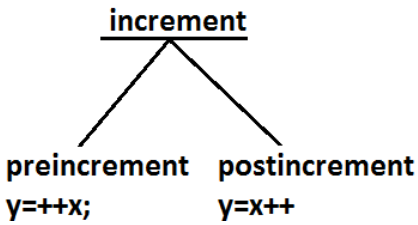
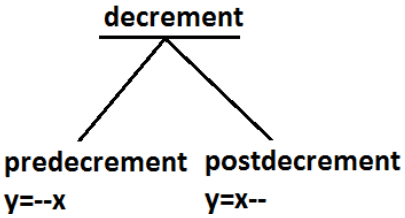


Diagram 2:



Example:

Expression	Initial value of x	Final value of x	Final value of y
Y=++x;	10	11	11
Y=x++;	10	11	10
Y=--x;	10	9	9
Y=x--;	10	9	10

- We can apply increment or decrement operator only for variables but not for constant values.

Example:

```
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;
```

- Nesting of increment or decrement operators is not allowed.

Example:

```
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);
```

- We can't apply increment or decrement operator for final variables.

Example:

```
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++;
```

- We can apply increment or decrement operator for any primitive type except Boolean.

Example:

```
class Test{
public static void main(String[] args){
int x=4;
x++;
System.out.println("value of x:"+x);
} outpt:
} value of x:5
```

```
class Test{
public static void main(String[] args){
char ch='a';
ch++;
System.out.println("value of ch:"+ch);
} output:
} value of ch:b
```

```
class Test{
public static void main(String[] args){
double d=10.5;
d++;
System.out.println("value of d:"+d);
} output:
} value of d:11.5
```

```
class Test{
public static void main(String[] args){
boolean b=true;
b++;
System.out.println("value of b:"+b);
}output:
}compile time error
Test.java:4: operator ++ cannot be applied to boolean
b++;
```

- If we apply any arithmetic operator between two variables "a" and "b" the result type is always.

max(int,typeof a,typeof b)

Example 1:

```
byte a=10;
byte b=20;
byte c=(a+b);        C.E
System.out.println(c);
```

```
OperatorsDemo.java:7: possible loss of precision
found   : int
required: byte
                byte c=a+b;
```

Example 2:

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

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

```
OperatorsDemo.java:6: possible loss of precision
found   : int
required: byte
                b=b+1;
```

- In the case of increment or decrement operator the required type-casting will be performed automatically by the compiler.

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

Example:

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

Arithmetic operators: (+,-,*, /, %)

- If we apply any arithmetic operation between two variables "a" and "b". The result type is always.

max(int, type of a, type b)

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	

Example:

```
System.out.println('a'+1);//98
System.out.println('a'+'b');//195
System.out.println(10+0.5);//10.5
System.out.println('a'+3.5);//100.5
```

Infinity:

- In the case of integral arithmetic (byte, short, int, long) there is no way to represent infinity.
- Hence if infinity is the result then we will get ArithmeticException.

Example:

```
System.out.println(10/0);  R.E  Exception in thread "main" java.lang.ArithmeticException: / by zero
```

- But in floating point arithmetic(float, double), there is a way to represent **infinity**. For this Float and Double classes contains the following two constants.

POSITIVE-INFINITIVE;
NEGITIVE-INFINITIVE;

- Hence if infinity is the result we won't get any runtime exception in floating point arithmetic.

Example:

```
System.out.println(10/0.0);//+Infinity
System.out.println(-10/0.0);//-Infinity
```

NaN(Not a Number):

- In the case of integral arithmetic there is no way to represent "undefined results". Hence if the result is undefined we will get runtime exception saying ArithmeticException.

Example:

```
System.out.println(0/0);  R.E  Exception in thread "main" java.lang.ArithmeticException: / by zero
```

- But in floating point arithmetic (float, double), there is a way to represent undefined result for this Float and Double classes contain "**NaN**".
- Hence if the result is undefined, we won't get any runtime exception in floating point arithmetic.

Example:

```
System.out.println(0.0/0);//NaN
System.out.println(-0.0/0);//NaN
System.out.println(0/0.0);//NaN
```

- For any **x** value including **NaN** the following expressions return false.

Example:

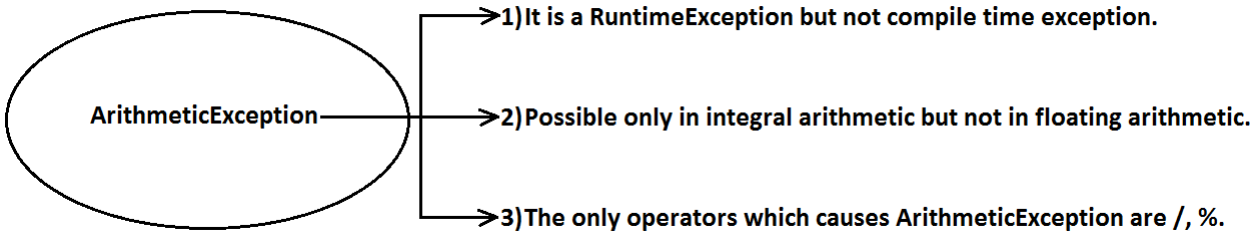
```
class Test
{
	public static void main(String[] args)
	{
		int x=10;
		System.out.println(x>Float.NaN);//false
		System.out.println(x<Float.NaN);//false
		System.out.println(x>=Float.NaN);//false
		System.out.println(x<=Float.NaN);//false
		System.out.println(x==Float.NaN);//false
	}
}
```

- For any **x** value including **NaN** the following expression return true.

Example:

```
class Test
{
	public static void main(String[] args)
	{
		int x=10;
		System.out.println(x!=Float.NaN);//true
		System.out.println(Float.NaN!=Float.NaN);//true
	}
}
```

Summary:



String concatenation operator:

- The only operator which is overloaded in java is "+" operator. Sometime it acts as arithmetic addition operator and some time concatenation operator.
- If at least one argument is string type then "+" operator acts as concatenation and if both arguments are number type then it acts as arithmetic addition operator.

Example 1:

```
String a="bhaskar";
    int b=10,c=20,d=30;
    System.out.println(a+b+c+d);//bhaskar102030
    System.out.println(b+c+d+a);//60bhaskar
    System.out.println(b+c+a+d);//30bhaskar30
    System.out.println(b+a+c+d);//10bhaskar2030
```

Example 2:

```
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 3:

```
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;
```

Relational operator: (<, <=, >, >=)

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

Example:

```
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);
```

- We can't apply relational operators for the object types.

Example:

```
System.out.println("bhaskar">"bhaskar");   C.E
```

```
OperatorsDemo.java:5: operator > cannot be applied to java.lang.String,java.lang.String
                    System.out.println("bhaskar">"bhaskar");
```

- We can't perform nesting of relational operators.

Example:

```
System.out.println(10<20<30);   C.E
```

```
E:\scjp>javac OperatorsDemo.java
OperatorsDemo.java:5: operator < cannot be applied to boolean,int
            System.out.println(10<20<30);
```

Equality operator :(==,!=)

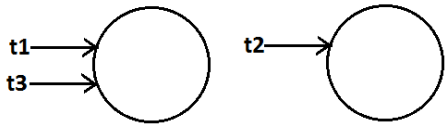
- We can apply equality operators for every primitive type including boolean type also.

Example:
```
System.out.println(10==10.0);//true
System.out.println('a'==97.0);//true
System.out.println(true==true);//true
System.out.println('a'!='b');//true
```

- We can apply equality operator even for object reference also.
- In the case of object references == (double equal operator) is always meant for reference comparison only (address comparison).
- i.e. r1==r2 return true if and only if both r1 and r2 point to the same object.

Example:
```
Thread t1=new Thread();
Thread t2=new Thread();
Thread t3=t1;
System.out.println(t1==t2);//false
System.out.println(t1==t3);//true
```

Diagram:

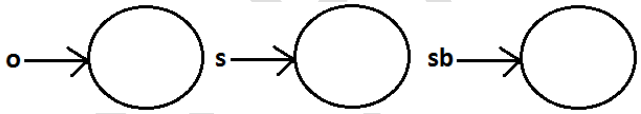


- To use equality operator compulsory there should be some relationship between argument type(either parent-child (or)child-parent (or) same type)otherwise we will get compile time error saying " incomparable types".

Example:
```
Object o=new Object();
String s=new String("bhaskar");
StringBuffer sb=new StringBuffer();
System.out.println(o==s);//false
System.out.println(o==sb);//false
```

```
System.out.println(s==sb); C.E
E:\scjp>javac OperatorsDemo.java
OperatorsDemo.java:10: incomparable types: java.lang.String and java.lang.StringBuffer
        System.out.println(s==sb);
```

Diagram:



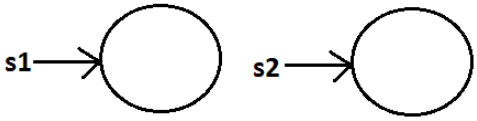
- For any object reference of, r==null is always false. But null==null is true.

== Vs .equals():

- ==operator is always meant for reference comparison whereas .equals() method mostly meant for content comparison.

Example:
```
String s1=new String("bhaskar");
String s2=new String("bhaskar");
System.out.println(s1==s2);//false
System.out.println(s1.equals(s2));//true
```

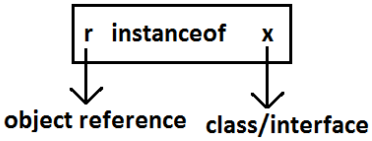
Diagram:



Instanceof operator:

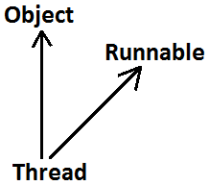
- We can use this operator to check whether the given object is of particular type (or) not.

Syntax:



Example:
```
Thread t=new Thread();
System.out.println(t instanceof Thread);//true
System.out.println(t instanceof Object);//true
System.out.println(t instanceof Runnable);//true
```

Diagram:



Note:

- To use "instanceof" operator compulsory there should be some relationship between argument types (either parent-child (or) child-parent (or) same type) otherwise we will get compile time error saying "inconvertible types".

Example:

```
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);
```

- Whenever we are comparing parent object is child type or not by using instanceof operator then we will get "false" as output.

Example:

```
Object o=new Object();
System.out.println(o instanceof String);//false
```

- For any class or interface x

  null instanceof x the result is always "false".

Example:

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

Bitwise operators:

& (AND): If both arguments are true then result is true.
| (OR): if at least one argument is true. Then the result is true.
^ (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     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:

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

Example 1:

```
System.out.println(~true);
```

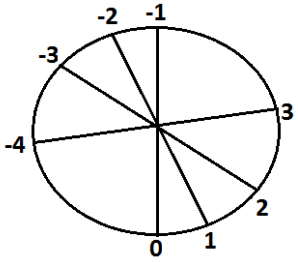
C.E →

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

Example 2:

```
System.out.println(~4);//5
```

Diagram:



Boolean complement (!) operator:

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

Example:

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

System.out.println(!true);//false
System.out.println(!false);//true

Summary:

&
| Applicable for both boolean and integral types.
^
~-------- Applicable for integral types only.
! --------Applicable for boolean types only.

Short circuit (&&, ||) operators:

- These operators are exactly same as normal bitwise operators &, | except the following differences.

&,|	&&,||
1) Both arguments should be evaluated always.	1) Second argument evaluation is optional.
2) Relatively performance is low.	2) Relatively performance is high.
3) Applicable for both integral and boolean types.	3) Applicable only for boolean types but not for integral types.

1) r1&&r2

- r2 will be evaluated if and only if r1 is true.

2) r1||r2

- r2 will be evaluated if and only if r1 is false.

Example 1:

```
class OperatorsDemo
{
    public static void main(String[] args)
    {
        int x=10, y=15;
        if(++x>10(operator)++y<15)
        {
            ++x;
        }
        else
        {
            ++y;
        }
        System.out.println(x+"------------"+y);
    }
}
```

Output:

operator	x	y
&	11	17
|	12	16
&&	11	17
||	12	15

Example 2:

```
class OperatorsDemo
{
    public static void main(String[] args)
    {
        int x=10;
        if(++x<10(operator)x/0>10)
        {
            System.out.println("hello");
        }
        else
        {
            System.out.println("hi");
        }
    }
}
```

&&	**Output:** Hi
&	**Output:** R.E: Exception in thread "main" java.lang.ArithmeticException: / by zero

Type-cast operator (primitive type casting):

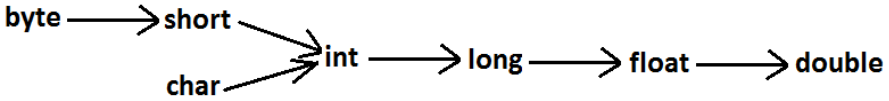
- There are two types of primitive type casting.

1) **Implicit type casting.**
2) **Explicit type casting.**

Implicit type casting:

- Compiler is the responsible for this typecasting.
- Whenever we are assigning smaller data type value to the bigger data type variable this type casting will be performed.
- Also known as widening or up casting.
- There is no lose of information in this type casting.
- The following are various possible implicit type casting.

Diagram:



Example 1:

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

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

Example 2:

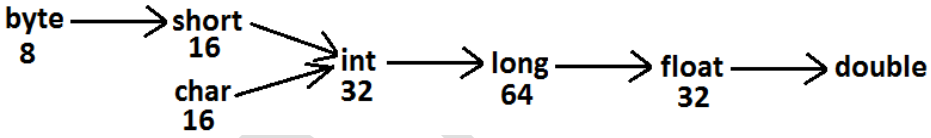
```
double d=10;
System.out.println(d);//10.0
```

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

Explicit type casting:

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

Diagram:



Example 1:

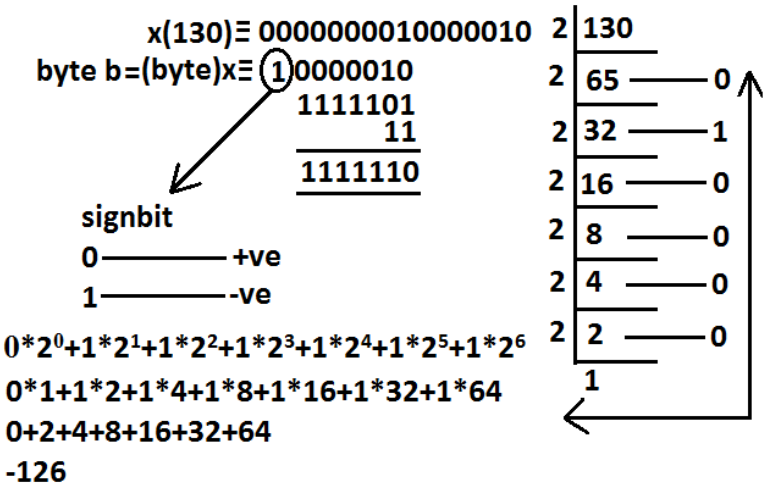
```
int x=130;
byte b=x;
```

```
E:\scjp>javac OperatorsDemo.java
OperatorsDemo.java:6: possible loss of precision
found   : int
required: byte
            byte b=x;
```

Example 2:

```
int x=130;
byte b=(byte)x;
System.out.println(b);//-126
```

Analysis:



- Whenever we are assigning bigger data type value to the smaller data type variable by explicit type casting the most significant bit(MSB)will be lost.

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

- Whenever we are assigning floating point data types to the integer data type by explicit type casting the digits after the decimal point will be loosed.

Example:

```
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 operators:

- They are three types of assignment operators.

Simple assignment:

Example: int x=10;

Chained assignment:

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

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

Example 1:

```
int a=b=c=d=20; C.E
```

```
cannot find symbol
variable b
variable c
variable d
```

Example 2:
```
int a,b,c,d;
a=b=c=d=30;
```

Compound assignment:

- Sometimes we can mix assignment operator with some other operator to form compound assignment operator.
- The following is the list of all possible compound assignment operators in java.

Example 1:

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

- In the case of compound assignment operator the required type casting will be performed automatically by the compiler similar to increment and decrement operators.

Example 2:

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

```
E:\scjp>javac OperatorsDemo.java
OperatorsDemo.java:6: possible loss of precision
found   : int
required: byte
          b=b+1;
```

Example 3:

```
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
```

Conditional operator:

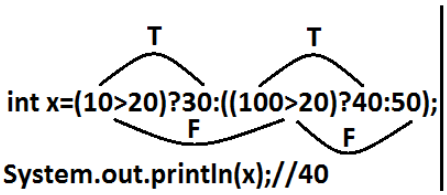
- The only ternary operator which is available in java is conditional operator.

Example 1:

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

- We can perform nesting of conditional operator also.

Example 2:




```
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
```

Example 5:

```
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;
```

Example 6:

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

new operator:

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

[] operator:

- We can use this operator to declare and construct arrays.

Java operator precedence:

Unary operators:

[], x++, X--,
++x,--x, ~,!,
new, <type>.

Arithmetic operators:

*,/,%,
+,-.

Shift operators:

>>,>>>,<<.

Compression operators:

<, <=,>,>=, instanceof.

Equality operators:

==,!=

Bitwise operators:

&,^,|.

Short circuit operators:

&&,||.

Conditional operator:

(?:)

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	

Example 1:

```
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
```

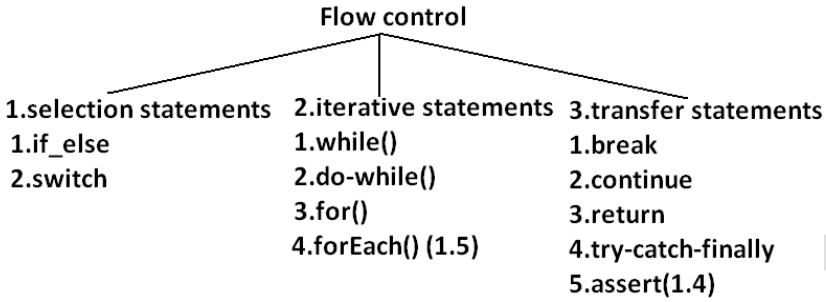
Example 2:

```
int i=1;
i+=++i + i++ + ++i + i++;
System.out.println(i);//13Analysis:
i=i+ ++i + i++ + ++i + i++;
i=1+2+2+4+4;
i=13;
```

Flow Control

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

Diagram:



Selection statements:

1. **if-else:**

Syntax:

```
        boolean
if(b)
{
//action if b is true
}else{
//action if b is false
}
```

- The argument to the if statement should be Boolean 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** 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.
- If several options are available then it is not recommended to use **if-else** we should go for **switch statement**.

Switch:
Syntax:

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

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

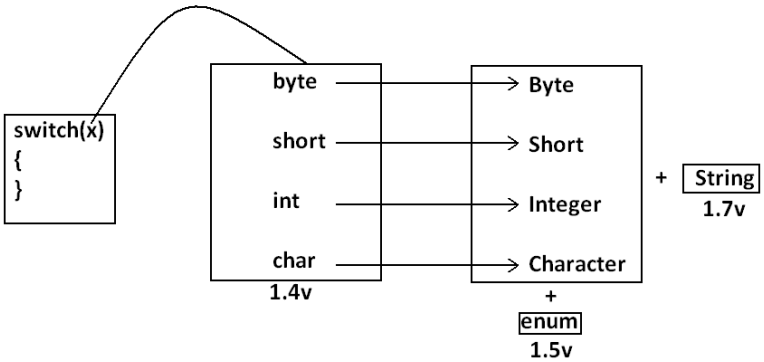
EXAMPLE 1:

```
public class ExampleSwitch{
public static void main(String args[]){
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");

- 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 are allowed.

DIAGRAM:



- 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
```

- Switch argument and case label can be expressions also, **but case 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:
```

- Duplicate case labels are not allowed.

EXAMPLE 6:

```
public class ExampleSwitch{
```

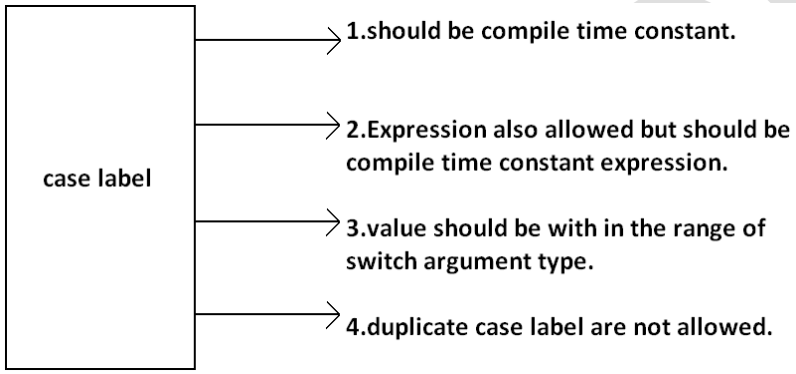
```
public static void main(String args[]){
int x=10;
switch(x)
{
case 97:
System.out.println("97");
case 99:
System.out.println("99");
case 'a':
System.out.println("100");
}}}
```

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:

- Within 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 .

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:

- Within the switch we can take the **default** anywhere, but at most once it is convention 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");
}}
```

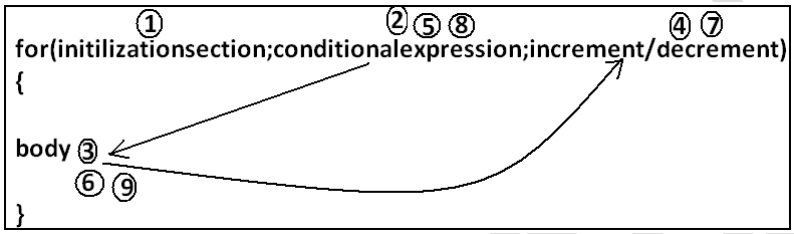
Output:

```
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:



1) 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:

1) Int i=0,j=0; **valid**
2) Int i=0,Boolean b=true; **invalid**
3) 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
```

2) 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.

3) 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:

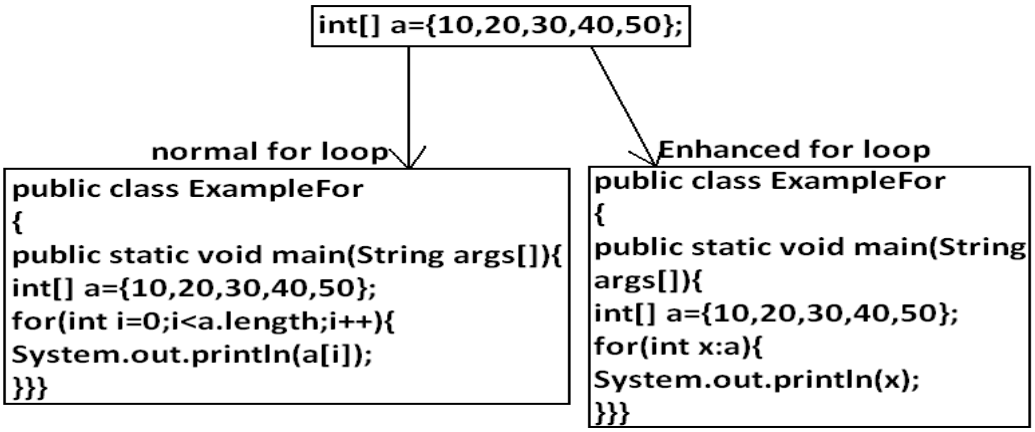
```
D:\Java>javac ExampleFor.java
ExampleFor.java:7: unreachable statement
System.out.println("hi");
```

For each:

- 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.

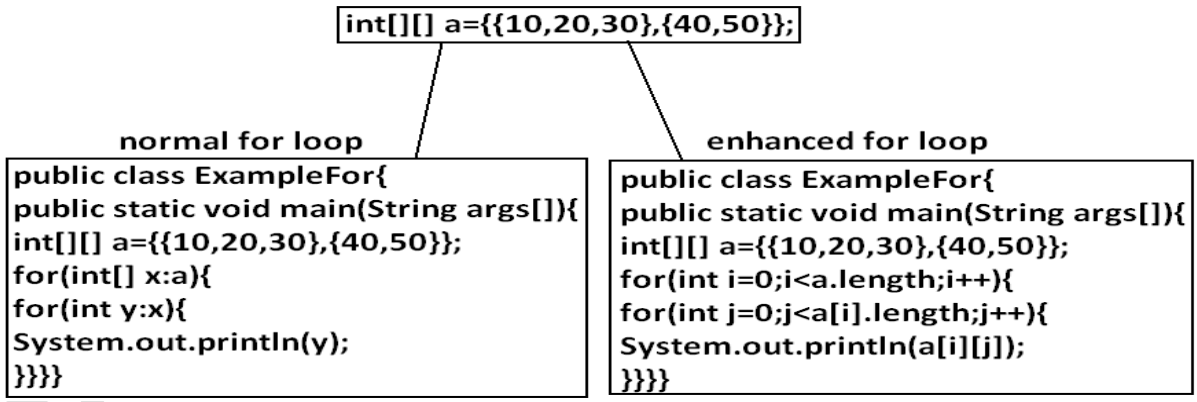
Example:



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.

Transfer statements:

Break statement:

- We can use break statement in the following cases.

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

Example 1:

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

Output:

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

- 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:

```
class Test{
public static void main(String args[]){
int x=2;
for(int i=0;i<10;i++){
if(i%x==0)
continue;
System.out.println(i);
}}}
```

0%2=0
2/0=infinity

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");
}
}
```

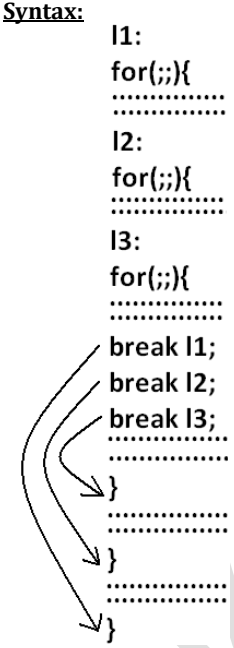
Output:

```
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

1. Introduction
2. Runtime stack mechanism
3. Default exception handling in java
4. Exception hierarchy
5. Customized exception handling by try catch
6. Control flow in try catch
7. Methods to print exception information
8. Try with multiple catch blocks
9. Finally
10. Difference between final, finally, finalize
11. Control flow in try catch finally
12. Control flow in nested try catch finally
13. Various possible combinations of try catch finally
14. throw keyword
15. throws keyword
16. Exception handling keywords summary
17. Various possible compile time errors in exception handling
18. Customized exceptions
19. Top-10 exceptions

Exception: An unwanted unexpected event that disturbs normal flow of the program is called exception.

Example:

SleepingException
TyrePunchuredException
FileNotFoundException…..etc

- It is highly recommended to handle exceptions. The main objective of exception handling is graceful (normal) termination of the program.

What is the meaning of exception handling?

- Exception handling doesn't mean repairing an exception. We have to define alternative way to continue rest of the program normally this way of "defining alternative is nothing but exception handling".

Example: Suppose our programming requirement is to read data from London file at runtime if London file is not available our program should not be terminated abnormally. We have to provide a local file to continue rest of the program normally. This way of defining alternative is nothing but exception handling.

Example:

```
try
{
read data from london file
}
catch(FileNotFoundException e)
{
use local file and continue rest of the program normally
}
}
.
.
.
```

Runtime stack mechanism: For every thread JVM will create a separate stack all method calls performed by the thread will be stored in that stack. Each entry in the stack is called "one activation record" (or) "stack frame". After completing every method call JVM removes the corresponding entry from the stack. After completing all method calls JVM destroys the empty stack and terminates the program normally.

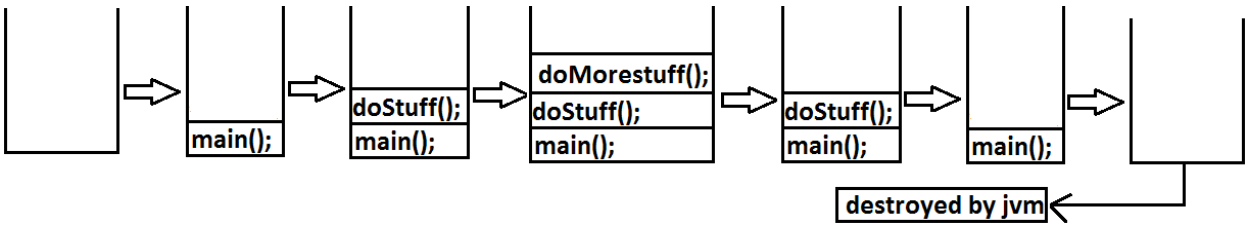
Example:

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

Output:

Hello

Diagram:



Default exception handling in java:

1) If an exception raised inside any method then the method is responsible to create Exception object with the following information.
1) Name of the exception.
2) Description of the exception.
3) Location of the exception.
2) After creating that Exception object the method handovers that object to the JVM.
3) JVM checks whether the method contains any exception handling code or not. If method won't contain any handling code then JVM terminates that method abnormally and removes corresponding entry form the stack.
4) JVM identifies the caller method and checks whether the caller method contain any handling code or not. If the caller method also does not contain handling code then JVM terminates that caller also abnormally and the removes corresponding entry from the stack.
5) This process will be continued until main() method and if the main() method also doesn't contain any exception handling code then JVM terminates main() method and removes corresponding entry from the stack.
6) Then JVM handovers the responsibility of exception handling to the default exception handler.
7) Default exception handler just print exception information to the console in the following formats and terminates the program abnormally.
   Name of exception: description
   Location of exception (stack trace)

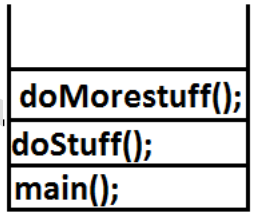
Example:

```
class Test
{
public static void main(String[] args){
doStuff();
}
public static void doStuff(){
doMoreStuff();
}
public static void doMoreStuff(){
System.out.println(10/0);
}}
```

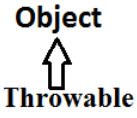
Output:

```
Runtime error
Exception in thread "main" java.lang.ArithmeticException: / by zero
    at Test.doMoreStuff(Test.java:10)
    at Test.doStuff(Test.java:7)
    at Test.main(Test.java:4)
```

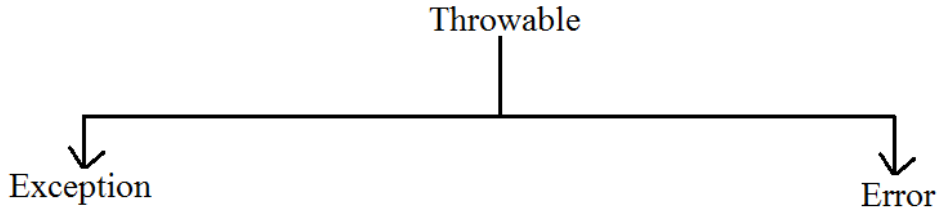
Diagram:



Exception hierarchy:



Throwable acts as a root for exception hierarchy.
Throwable class contains the following two child classes.



Exception: Most of the cases exceptions are caused by our program and these are recoverable.
Error: Most of the cases errors are not caused by our program these are due to lack of system resources and these are non recoverable.

Checked Vs Unchecked Exceptions:

- The exceptions which are checked by the compiler for smooth execution of the program at runtime are called checked exceptions.

1) HallTicketMissingException
2) PenNotWorkingException
3) FileNotFoundException

- The exceptions which are not checked by the compiler are called unchecked exceptions.

1) BombBlaustException
2) ArithmeticException
3) NullPointerException

Note: RuntimeException and its child classes, Error and its child classes are unchecked and all the remaining are considered as checked exceptions.
Note: Whether exception is checked or unchecked compulsory it should occur at runtime only there is no chance of occurring any exception at compile time.

Partially checked Vs fully checked:

- A checked exception is said to be fully checked if and only if all its child classes are also checked.

Example:

1) IOException
2) InterruptedException

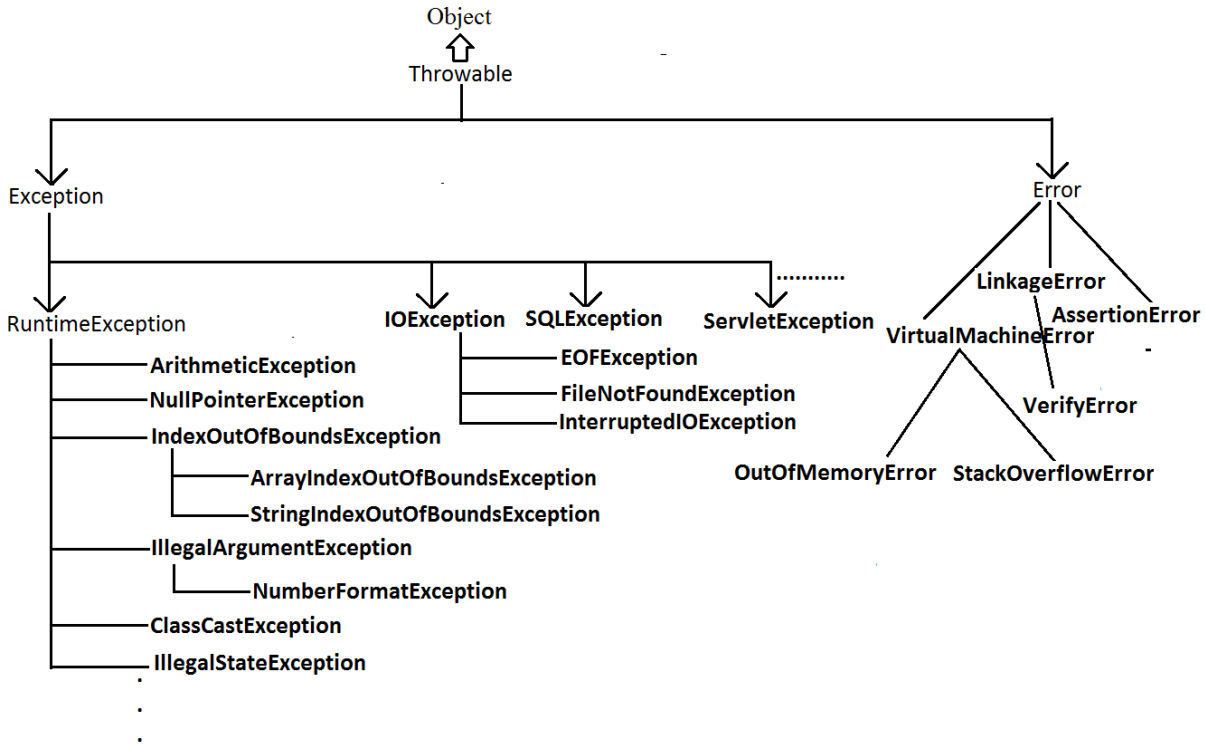
- A checked exception is said to be partially checked if and only if some of its child classes are unchecked.

Example: Exception

- The only partially checked exceptions available in java are:
  1. Throwable.
  2. Exception.

Which of the following are checked?
1. RuntimeException-----unchecked
2. Error-----unchecked
3. IOException-----fully checked
4. Exception-----partially checked
5. InterruptedException-----fully checked
6. Throwable------partially checked

Diagram:



Customized exception handling by try catch:

- It is highly recommended to handle exceptions.
- In our program the code which may cause an exception is called risky code we have to place risky code inside try block and the corresponding handling code inside catch block.

Example:

```
try
{
risky code
}
catch(Exception e)
{
handling code
}
```

Without try catch	With try catch

class Test
{
public static void main(String[] args){
System.out.println("statement1");
System.out.println(10/0);
System.out.println("statement3");
}
}
Abnormal termination.
	class Test{
	public static void main(String[] args){
	System.out.println("statement1");
	try{
	System.out.println(10/0);
	}
	catch(ArithmeticException e){
	System.out.println(10/2);
	}
	System.out.println("statement3");
	}}
	Output:
	statement1
	5
	statement3
	Normal termination.

Control flow in try catch:

Case 1: There is no exception.
1, 2, 3, 5 normal termination.
Case 2: if an exception raised at statement 2 and corresponding catch block matched 1, 4, 5 normal termination.
Case 3: if an exception raised at statement 2 but the corresponding catch block not matched 1 followed by abnormal termination.
Case 4: if an exception raised at statement 4 or statement 5 then it's always abnormal termination of the program.

Note:

1. Within the try block if anywhere an exception raised then rest of the try block won't be executed even though we handled that exception. Hence we have to place/take only risk code inside try and **length** of the try block should be as **less** as possible.
2. If any statement which raises an exception and it is not part of any try block then it is always abnormal termination of the program.
3. There may be a chance of raising an exception inside catch and finally blocks also in addition to try block.

Various methods to print exception information:

- Throwable class defines the following methods to print exception information to the console.

printStackTrace(): This method prints exception information in the following format.

Name of the exception: description of exception Stack trace

toString(): This method prints exception information in the following format.

Name of the exception: description of exception

getMessage(): This method returns only description of the exception.

Description.

Example:

```
class Test
{
public static void main(String[] args){
try
{
System.out.println(10/0);
}
catch(ArithmeticException e)
{
e.printStackTrace();
System.out.println(e);
System.out.println(e.getMessage());
}}}
```

```
java.lang.ArithmeticException: / by zero
    at Test.main(Test.java:6)
```

```
java.lang.ArithmeticException: / by zero
```

```
/ by zero
```

Note: Default exception handler internally uses printStackTrace() method to print exception information to the console.

Try with multiple catch blocks: The way of handling an exception is varied from exception to exception hence for every exception raise a separate catch block is required that is try with multiple catch blocks is possible and recommended to use.

Example:

```
try
{
.
.
.
.
}
catch(Exception e)
{
default handler
}
```

- This approach is not recommended because for any type of Exception we are using the same catch block.

```
try
{
.
.
.
.
catch(FileNotFoundException e)
{
use local file
}
catch(ArithmeticException e)
{
perform these Arithmetic operations
}
catch(SQLException e)
{
don't use oracle db, use mysql db
}
catch(Exception e)
{
default handler
}
```

- This approach is highly recommended because for any exception raise we are defining a separate catch block.

- If try with multiple catch blocks presents then order of catch blocks is very important it should be from child to parent by mistake if we are taking from parent to child then we will get Compile time error saying "exception xxx has already been caught".

Example:

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

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

```
{
try
{
System.out.println(10/0);
}
catch(Exception e)
{
e.printStackTrace();
}
catch(ArithmeticException e)
{
e.printStackTrace();
}}}
```

Output:

```
Compile time error.
Test.java:13: exception java.lang.ArithmeticException has
already been caught
catch(ArithmeticException e)
```

```
{
try
{
System.out.println(10/0);
}
catch(ArithmeticException e)
{
e.printStackTrace();
}
catch(Exception e)
{
e.printStackTrace();
}}}
```

Output:

```
Compile successfully.
```

Finally block:

- It is never recommended to take clean up code inside try block because there is no guarantee for the execution of every statement inside a try.
- It is never recommended to place clean up code inside catch block because if there is no exception then catch block won't be executed.
- We require some place to maintain clean up code which should be executed always irrespective of whether exception raised or not raised and whether handled or not handled such type of place is nothing but finally block.
- Hence the main objective of finally block is to maintain cleanup code.

Example:

```
try
{
risky code
}
catch(x e)
{
handling code
}
finally
{
cleanup code
}
```

- The specialty of finally block is it will be executed always irrespective of whether the exception raised or not raised and whether handled or not handled.

Example 1:

```
class Test
{
public static void main(String[] args)
{
try
{
System.out.println("try block executed");
}
catch(ArithmeticException e)
{
System.out.println("catch block executed");
}
finally
{
System.out.println("finally block executed");
}}}
```

Output:

```
Try block executed
Finally block executed
```

Example 2:

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

```
try
{
System.out.println("try block executed");
System.out.println(10/0);
}
catch(ArithmeticException e)
{
System.out.println("catch block executed");
}
finally
{
System.out.println("finally block executed");
}}}
```

Output:

```
Try block executed
Catch block executed
Finally block executed
```

Example 3:

```
class Test
{
public static void main(String[] args)
{
try
{
System.out.println("try block executed");
System.out.println(10/0);
}
catch(NullPointerException e)
{
System.out.println("catch block executed");
}
finally
{
System.out.println("finally block executed");
}}}
```

Output:

```
Try block executed
Finally block executed
Exception in thread "main" java.lang.ArithmeticException: / by zero
    at Test.main(Test.java:8)
```

Return Vs Finally:

- Even though return present in try or catch blocks first finally will be executed and after that only return statement will be considered that is finally block dominates return statement.

Example:

```
class Test
{
public static void main(String[] args)
{
try
{
System.out.println("try block executed");
return;
}
catch(ArithmeticException e)
{
System.out.println("catch block executed");
}
finally
{
System.out.println("finally block executed");
}}}
```

Output:

```
Try block executed
Finally block executed
```

- If return statement present try catch and finally blocks then finally block return statement will be considered.

Example:

```
class Test
```

```
{
public static void main(String[] args)
{
System.out.println(methodOne());
}
public static int methodOne(){
try
{
System.out.println(10/0);
return 777;
}
catch(ArithmeticException e)
{
return 888;
}
finally{
return 999;
}}}
```

Output:

999

- There is only one situation where the finally block won't be executed is whenever we are using System.exit(0) method.

Example:

```
class Test
{
public static void main(String[] args)
{
try
{
System.out.println("try");
System.exit(0);
}
catch(ArithmeticException e)
{
System.out.println("catch block executed");
}
finally
{
System.out.println("finally block executed");
}}}
```

Output:

Try

Difference between final, finally, and finalize:

Final:

- Final is the modifier applicable for class, methods and variables.
- If a class declared as the final then child class creation is not possible.
- If a method declared as the final then overriding of that method is not possible.
- If a variable declared as the final then reassignment is not possible.

Finally:

- It is the block always associated with try catch to maintain clean up code which should be executed always irrespective of whether exception raised or not raised and whether handled or not handled.

Finalize:

- It is a method which should be called by garbage collector always just before destroying an object to perform cleanup activities.

Note:

- To maintain clean up code faunally block is recommended over finalize() method because we can't expert exact behavior of GC.

Control flow in try catch finally:

Example:

```
class Test
{
public static void main(String[] args){
try{
System.out.println("statement1");
System.out.println("statement2");
System.out.println("statement3");
}
catch(Exception e){
System.out.println("statement4");
```

```
}
finally
{
System.out.println("statement5");
}
System.out.println("statement6");
}
}
```

Case 1: If there is no exception. 1, 2, 3, 5, 6 normal termination.
Case 2: if an exception raised at statement 2 and corresponding catch block matched. 1,4,5,6 normal terminations.
Case 3: if an exception raised at statement 2 and corresponding catch block is not matched. 1,5 abnormal termination.
Case 4: if an exception raised at statement 4 then it's always abnormal termination but before the finally block will be executed.
Case 5: if an exception raised at statement 5 or statement 6 its always abnormal termination.

Control flow in nested try catch finally:

Example:

```
class Test
{
public static void main(String[] args){
try{
System.out.println("statement1");
System.out.println("statement2");
System.out.println("statement3");
      try{
           System.out.println("statement4");
           System.out.println("statement5");
           System.out.println("statement6");
          }
          catch(ArithmeticException  e){
          System.out.println("statement7");
          }
          finally
          {
          System.out.println("statement8");
          }
          System.out.println("statement9");
  }
catch(Exception e)
{
System.out.println("statement10");
}
finally
{
System.out.println("statement11");
}
System.out.println("statement12");
}
}
```

Case 1: if there is no exception. 1, 2, 3, 4, 5, 6, 8, 9, 11, 12 normal termination.
Case 2: if an exception raised at statement 2 and corresponding catch block matched 1,10,11,12 normal terminations.
Case 3: if an exception raised at statement 2 and corresponding catch block is not matched 1, 11 abnormal termination.
Case 4: if an exception raised at statement 5 and corresponding inner catch has matched 1, 2, 3, 4, 7, 8, 9, 11, 12 normal termination.
Case 5: if an exception raised at statement 5 and inner catch has not matched but outer catch block has matched. 1, 2, 3, 4, 8, 10, 11, 12 normal termination.
Case 6: if an exception raised at statement 5 and both inner and outer catch blocks are not matched. 1, 2, 3, 4, 8, 11 abnormal termination.
Case 7: if an exception raised at statement 7 and the corresponding catch block matched 1, 2, 3,.,.,., 8, 10, 11, 12 normal termination.
Case 8: if an exception raised at statement 7 and the corresponding catch block not matched 1, 2, 3,.,.,.,8,11 abnormal terminations.
Case 9: if an exception raised at statement 8 and the corresponding catch block has matched 1, 2, 3,.,.,.,., 10, 11,12 normal termination.
Case 10: if an exception raised at statement 8 and the corresponding catch block not matched 1, 2, 3,.,.,.,., 11 abnormal terminations.
Case 11: if an exception raised at statement 9 and corresponding catch block matched 1, 2, 3,.,.,.,., 8,10,11,12 normal termination.
Case 12: if an exception raised at statement 9 and corresponding catch block not matched 1, 2, 3,.,.,.,., 8, 11 abnormal termination.
Case 13: if an exception raised at statement 10 is always abnormal termination but before that finally block 11 will be executed.
Case 14: if an exception raised at statement 11 or 12 is always abnormal termination.
Note: if we are not entering into the try block then the finally block won't be executed. Once we entered into the try block without executing finally block we can't come out.

Example:

```
class Test
{
public static void main(String[] args){
try{
System.out.println(10/0);
}
catch(ArithmeticException e)
{
System.out.println(10/0);
}
finally{
String s=null;
System.out.println(s.length());
}}}
```

Note: Default exception handler can handle only one exception at a time and that is the most recently raised exception.

Various possible combinations of try catch finally:

Example 1:

```
class Test1{
public static void main(String[] args){
try
{}
catch(ArithmeticException e)
{}
}}
```

Output:

Compile and running successfully.

Example 2:

```
class Test1{
public static void main(String[] args){
try
{}
catch(ArithmeticException e)
{}
catch(NullPointerException e)
{}
}
}
```

Output:

Compile and running successfully.

Example 3:

```
class Test1{
public static void main(String[] args){
try
{}
catch(ArithmeticException e)
{}
catch(ArithmeticException e)
{}
}
}
```

Output:

Compile time error.

Test1.java:7: exception java.lang.ArithmeticException has already been caught

catch(ArithmeticException e)

Example 4:

```
class Test1{
public static void main(String[] args){
try
{}
}
}
```

Output:

Compile time error

Test1.java:3: 'try' without 'catch' or 'finally'

try

Example 5:

```
class Test1{
public static void main(String[] args){
catch(Exception e)
{}
}
}
```

Output:

```
Compile time error.
Test1.java:3: 'catch' without 'try'
catch(Exception e)
```

Example 6:

```
class Test1{
public static void main(String[] args){
try
{}
System.out.println("hello");
catch(Exception e)
{}
}
}
```

Output:

```
Compile time error.
Test1.java:3: 'try' without 'catch' or 'finally'
Try
```

Example 7:

```
class Test1{
public static void main(String[] args){
try
{}
catch(Exception e)
{}
finally
{}
}
}
```

Output:

```
Compile and running successfully.
```

Example 8:

```
class Test1{
public static void main(String[] args){
try
{}
finally
{}
}
}
```

Output:

```
Compile and running successfully.
```

Example 9:

```
class Test1{
public static void main(String[] args){
try
{}
finally
{}
finally
{}
}
}
```

Output:

```
Compile time error.
Test1.java:7: 'finally' without 'try'
Finally
```

Example 10:

```
class Test1{
public static void main(String[] args){
try
```

```
{}
catch(Exception e)
{}
System.out.println("hello");
finally
{}
}
}
```

Output:

```
Compile time error.
Test1.java:8: 'finally' without 'try'
Finally
```

Example 11:

```
class Test1{
public static void main(String[] args){
try
{}
finally
{}
catch(Exception e)
{}
}
}
```

Output:

```
Compile time error.
Test1.java:7: 'catch' without 'try'
catch(Exception e)
```

Example 12:

```
class Test1{
public static void main(String[] args){
finally
{}
}
}
```

Output:

```
Test1.java:3: 'finally' without 'try'
Finally
```

Example 13:

```
class Test1{
public static void main(String[] args){
try
{  try{}
   catch(Exception e){}
}
catch(Exception e)
{}
}
}
```

Output:

```
Compile and running successfully.
```

Example 14:

```
class Test1{
public static void main(String[] args){
try
{}
catch(Exception e)
{
 try{}
  finally{}
}
}
}
```

Output:

```
Compile and running successfully.
```

Example 15:

```
class Test1{
public static void main(String[] args){
```

```
try
{ }
catch(Exception e)
{
 try{}
 catch(Exception e){}
}
finally{
  finally{}
   }
}
}
```

Output:

```
Compile time error.
Test1.java:11: 'finally' without 'try'
  finally{}
```

Example 16:

```
class Test1{
public static void main(String[] args){
finally{}
try{ }
catch(Exception e){}
}
}
```

Output:

```
Compile time error.
Test1.java:3: 'finally' without 'try'
finally{}
```

Example 17:

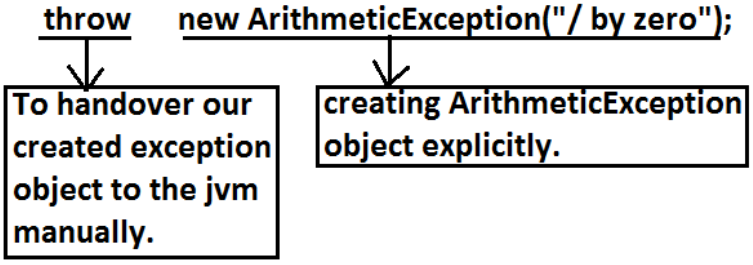
```
class Test1{
public static void main(String[] args){
try{ }
catch(Exception e){}
finally
{
try{}
catch(Exception e){}
finally{}
}
}
}
```

Output:

Compile and running successfully.

Throw statement: Sometime we can create exception object explicitly and we can hand over to the JVM manually by using throw keyword.

Example:



- The result of following 2 programs is exactly same.

class Test
{
public static void main(String[] args){
System.out.println(10/0);
}}
• In this case creation of ArithmeticException object and handover to the jvm will be performed automatically by the main() method.
	class Test
	{
	public static void main(String[] args){
	throw new ArithmeticException("/ by zero");
	}}
	• In this case we are creating exception object explicitly and handover to the JVM manually.

Note: In general we can use throw keyword for customized exceptions but not for predefined exceptions.

Case 1: throw e;

- If e refers null then we will get NullPointerException.

Example:

```
class Test3
{
static ArithmeticException e=new ArithmeticException();
public static void main(String[] args){
throw e;
}
}
```

Output:

```
Runtime exception: Exception in thread "main" java.lang.ArithmeticException
```

```
class Test
{
static ArithmeticException e;
public static void main(String[] args){
throw e;
}
}
```

Output:

```
Exception in thread "main" java.lang.NullPointerException
    at Test3.main(Test3.java:5)
```

Case 2: After throw statement we can't take any statement directly otherwise we will get compile time error saying unreachable statement.

Example:

```
class Test3
{
public static void main(String[] args){
System.out.println(10/0);
System.out.println("hello");
}
}
```

Output:

Runtime error: Exception in thread "main" java.lang.ArithmeticException: / by zero
    at Test3.main(Test3.java:4)

```
class Test3
{
public static void main(String[] args){
throw new ArithmeticException("/ by zero");
System.out.println("hello");
}
}
```

Output:

```
Compile time error.
Test3.java:5: unreachable statement
System.out.println("hello");
```

Case 3: We can use throw keyword only for Throwable types otherwise we will get compile time error saying incomputable types.

Example:

```
class Test3
{
public static void main(String[] args){
throw new Test3();
}
}
```

Output:

```
Compile time error.
Test3.java:4: incompatible types
found   : Test3
required: java.lang.Throwable
throw new Test3();
```

```
class Test3 extends RuntimeException
{
public static void main(String[] args){
throw new Test3();
}
}
```

Output:

```
Runtime error: Exception in thread "main" Test3
    at Test3.main(Test3.java:4)
```

Throws statement: in our program if there is any chance of raising checked exception compulsory we should handle either by try catch or by throws keyword otherwise the code won't compile.

Example:

```
class Test3
{
public static void main(String[] args){
Thread.sleep(5000);
}
}
```

- Unreported exception java.lang.InterruptedException; must be caught or declared to be thrown. We can handle this compile time error by using the following 2 ways.

Example:

By using try catch

```
class Test3
{
public static void main(String[] args){
try{
Thread.sleep(5000);
}
catch(InterruptedException e){}
}
}
```

Output:

```
Compile and running successfully
```

By using throws keyword

- We can use throws keyword to delicate the responsibility of exception handling to the caller method. Then caller method is responsible to handle that exception.

```
class Test3
{
public static void main(String[] args)throws InterruptedException{
Thread.sleep(5000);
}
}
```

Output:

```
Compile and running successfully
```

- Hence the main objective of "throws" keyword is to delicate the responsibility of exception handling to the caller method.
- "throws" keyword required only checked exceptions. Usage of throws for unchecked exception there is no use.
- "throws" keyword required only to convenes complier. Usage of throws keyword doesn't prevent abnormal termination of the program.

Example:

```
class Test
{
public static void main(String[] args)throws InterruptedException{
doStuff();
}
public static void doStuff()throws InterruptedException{
doMoreStuff();
}
public static void doMoreStuff()throws InterruptedException{
Thread.sleep(5000);
}
}
```

Output:
Compile and running successfully.

- In the above program if we are removing at least one throws keyword then the program won't compile.

Case 1: we can use throws keyword only for Throwable types otherwise we will get compile time error saying incompatible types.

Example:

```
class Test3{
public static void main(String[] args)throws Test3
{}
}
```

Output:

```
Compile time error
Test3.java:2: incompatible types
found   : Test3
required: java.lang.Throwable
public static void main(String[] args)throws Test3
```

```
class Test3 extends RuntimeException{
public static void main(String[] args)throws Test3
{}
}
```

Output:
Compile and running successfully.

Case 2:

Example:

```
class Test3{
public static void main(String[] args){
throw new Exception();
}
}
```

Output:

```
Compile time error.
Test3.java:3: unreported exception
java.lang.Exception; must be caught or declared to be
thrown
```

```
class Test3{
public static void main(String[] args){
throw new Error();
}
}
```

Output:

```
Runtime error
Exception in thread "main" java.lang.Error
      at Test3.main(Test3.java:3)
```

Case 3:

- In our program if there is no chance of rising an exception then we can't right catch block for that exception otherwise we will get compile time error saying exception XXX is never thrown in body of corresponding try statement. But this rule is applicable only for fully checked exception.

Example:

```
class Test
{
public static void main(String[] args){
try{
System.out.println("hello");
}
catch(Exception e)
{}
}
```
output:
hello
partial checked

```
class Test
{
public static void main(String[] args){
try{
System.out.println("hello");
}
catch(ArithmeticException e)
{}
}
```
output:
hello
unchecked

```
class Test
{
public static void main(String[] args){
try{
System.out.println("hello");
}
catch(java.io.IOException e)
{}
}
```
output:
compile time error
fully checked

```
class Test
{
public static void main(String[] args){
try{
System.out.println("hello");
}
catch(InterruptedException e)
{}
}
}
```

output:
compile time error
Fully checked

```
class Test
{
public static void main(String[] args){
try{
System.out.println("hello");
}
catch(Error e)
{}
}
}
```

output:
compile successfully
unchecked

Exception handling keywords summary:

1) **try:** To maintain risky code.
2) **catch:** To maintain handling code.
3) **finally:** To maintain cleanup code.
4) **throw:** To handover our created exception object to the JVM manually.
5) **throws:** To delegate responsibility of exception handling to the caller method.

Various possible compile time errors in exception handling:

1) Exception XXX has already been caught.
2) Unreported exception XXX must be caught or declared to be thrown.
3) Exception XXX is never thrown in body of corresponding try statement.
4) Try without catch or finally.
5) Catch without try.
6) Finally without try.
7) Incompatible types.
- Found:test
- Requried:java.lang.Throwable;

8) Unreachable statement.

Customized Exceptions (User defined Exceptions):

- Sometimes we can create our own exception to meet our programming requirements. Such type of exceptions are called customized exceptions (user defined exceptions).

Example:

1) InSufficientFundsException
2) TooYoungException
3) TooOldException

Program:

```
class TooYoungException extends RuntimeException
{
TooYoungException(String s)
{
super(s);
}
}
class TooOldException extends RuntimeException
{
TooOldException(String s)
{
super(s);
}
}
class CustomizedExceptionDemo
{
public static void main(String[] args){
int age=Integer.parseInt(args[0]);
if(age>60)
{
throw new TooYoungException("please wait some more time.... u will get best match");
}
else if(age<18)
{
throw new TooOldException("u r age already crossed....no chance of getting married");
}
else
{
```

```
System.out.println("you will get match details soon by e-mail");
}}}
```

Output:

1) E:\scjp>java CustomizedExceptionDemo 61
   Exception in thread "main" TooYoungException: please wait some more time.... u will get best match
   at CustomizedExceptionDemo.main(CustomizedExceptionDemo.java:21)
2) E:\scjp>java CustomizedExceptionDemo 27
   You will get match details soon by e-mail
3) E:\scjp>java CustomizedExceptionDemo 9
   Exception in thread "main" TooOldException: u r age already crossed....no chance of getting married
   at CustomizedExceptionDemo.main(CustomizedExceptionDemo.java:25)

Note: It is highly recommended to maintain our customized exceptions as unchecked by extending RuntimeException.

- We can catch any Throwable type including Errors also.

Example:

```
try
{}
catch(Error e)
{}
```
valid

Top-10 Exceptions:

- Exceptions are divided into two types. They are:

1) JVM Exceptions:
2) Programatic exceptions:

JVM Exceptions:

- The exceptions which are raised automatically by the jvm whenever a particular event occurs.

Example:

1) ArrayIndexOutOfBoundsException(AIOOBE)
2) NullPointerException (NPE).

Programatic Exceptions:

- The exceptions which are raised explicitly by the programmer (or) by the API developer are called programatic exceptions.

Example:

1) IllegalArgumentException(IAE).

1) ArrayIndexOutOfBoundsException:

- It is the child class of RuntimeException and hence it is unchecked. Raised automatically by the JVM whenever we are trying to access array element with out of range index.

Example:

```
class Test{
public static void main(String[] args){
int[] x=new int[10];
System.out.println(x[0]);//valid
System.out.println(x[100]);//AIOOBE
System.out.println(x[-100]);//AIOOBE
}
}
```

2) NullPointerException:

- It is the child class of RuntimeException and hence it is unchecked. Raised automatically by the JVM, whenever we are trying to call any method on null.

Example:

```
class Test{
public static void main(String[] args){
String s=null;
System.out.println(s.length());→R.E: NullPointerException
}
}
```

3) StackOverFlowError:

- It is the child class of Error and hence it is unchecked. Whenever we are trying to invoke recursive method call JVM will raise StackOverFloeError automatically.

Example:

```
class Test
{
public static void methodOne()
{
methodTwo();
}
public static void methodTwo()
```

```
{
methodOne();
}
public static void main(String[] args)
{
methodOne();
}
}
```

Output:

Run time error: StackOverFloeError

4) NoClassDefFound:

- It is the child class of Error and hence it is unchecked. JVM will raise this error automatically whenever it is unable to find required .class file.

  Example: java Test
- If Test.class is not available. Then we will get NoClassDefFound error.

5) **ClassCastException:**

- It is the child class of RuntimeException and hence it is unchecked. Raised automatically by the JVM whenever we are trying to typecast parent object to child type.

  Example:

```
class Test
{
public static void main(String[] args)
{
String s=new String("bhaskar");
Object o=(Object)s;
}
}
```
output:
valid

```
class Test
{
public static void main(String[] args)
{
Object o=new Object();
String s=(String)o;
}
}
```
output:
Runtime exception:ClassCastException

```
class Test
{
public static void main(String[] args)
{
Object o=new String("bhaskar");
String s=(String)o;
}
}
```
output:
valid

6) ExceptionInInitializerError:

- It is the child class of Error and it is unchecked. Raised automatically by the JVM, if any exception occurs while performing static variable initialization and static block execution.

  Example 1:

```
class Test{
static int i=10/0;
}
```

  Output:

  Runtime exception:
- Exception in thread "main" java.lang.ExceptionInInitializerError

  Example 2:

```
class Test{
static {
String s=null;
System.out.println(s.length());
}}
```

  Output:

  Runtime exception: Exception in thread "main" java.lang.ExceptionInInitializerError

7) IllegalArgumentException:

- It is the child class of RuntimeException and hence it is unchecked. Raised explicitly by the programmer (or) by the API developer to indicate that a method has been invoked with inappropriate argument.

  Example:

```
class Test{
public static void main(String[] args){
Thread t=new Thread();
t.setPriority(10);→valid
t.setPriority(100);→invalid
}}
```

  Output:

  Runtime exception
- Exception in thread "main" java.lang.IllegalArgumentException.

8) NumberFormatException:

- It is the child class of IllegalArgumentException and hence is unchecked. Raised explicitly by the programmer or by the API developer to indicate that we are attempting to convert string to the number. But the string is not properly formatted.

  Example:

```
class Test{
```

```
public static void main(String[] args){
int i=Integer.parseInt("10");
int j=Integer.parseInt("ten");
}}
```

Output:

Runtime Exception

- Exception in thread "main" java.lang.NumberFormatException: For input string: "ten"

9) **IllegalStateException:**

- It is the child class of RuntimeException and hence it is unchecked. Raised explicitly by the programmer or by the API developer to indicate that a method has been invoked at inappropriate time.

Example:

- Once session expires we can't call any method on the session object otherwise we will get IllegalStateException

```
HttpSession session=req.getSession();
System.out.println(session.getId());
session.invalidate();
System.out.println(session.getId());→illgalstateException
```

10) **AssertionError:**

- It is the child class of Error and hence it is unchecked. Raised explicitly by the programmer or by API developer to indicate that Assert statement fails.

Example:

```
assert(false);
```

Exception/Error	Raised by
1) AIOOBE 2) NPE(NullPointerException) 3) StackOverFlowError 4) NoClassDefFoundError 5) CCE(ClassCastException) 6) ExceptionInInitializerError 7) IAE(IllegalArgumentException) 8) NFE(NumberFormatException) 9) ISE(IllegalStateException) 10) AE(AssertionError)	Raised automatically by JVM(JVM Exceptions) Raised explicitly either by programmer or by API developer (Programatic Exceptions).

OOPS

1) Data Hiding
2) Abstraction
3) Encapsulation
4) Tightly Encapsulated Class
5) IS-A Relationship
6) HAS-A Relationship
7) Method Signature
8) Overloading
9) Overriding
10) Method Hiding
11) Static Control Flow
12) Instance Control Flow
13) Constructors
14) Coupling
15) Cohesion
16) Object Type Casting

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;
```

.....;
.....;
}

- 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:

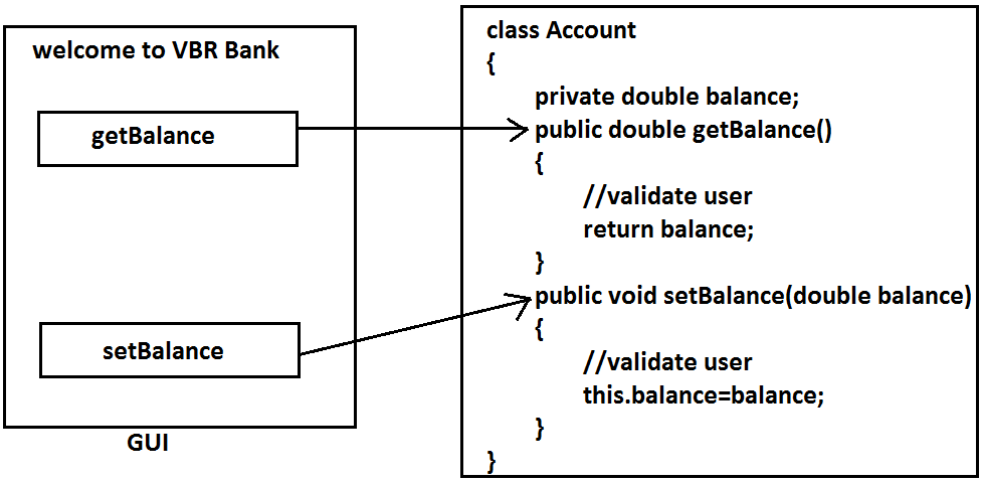
1) We can achieve security as we are not highlighting our internal implementation.
2) Enhancement will become very easy because without effecting end user we can able to perform any type of changes in our internal system.
3) It provides more flexibility to the end user to use system very easily.
4) It improves maintainability of the application.

Encapsulation:

- It is the process of Encapsulating data and corresponding methods into a single module.
- If any java class follows data hiding and abstraction such type of class is said to be encapsulated class.

Encapsulation=Datahiding+Abstraction

Example:



- In encapsulated class we have to maintain getter and setter methods for every data member.
- The main advantages of encapsulation are:

1) We can achieve security.
2) Enhancement will become very easy.
3) It improves maintainability 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, not required to check.

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;
}
class B extends A
{
      private int y=20;
}
class C extends B
{
      private int z=30;
}
```

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:

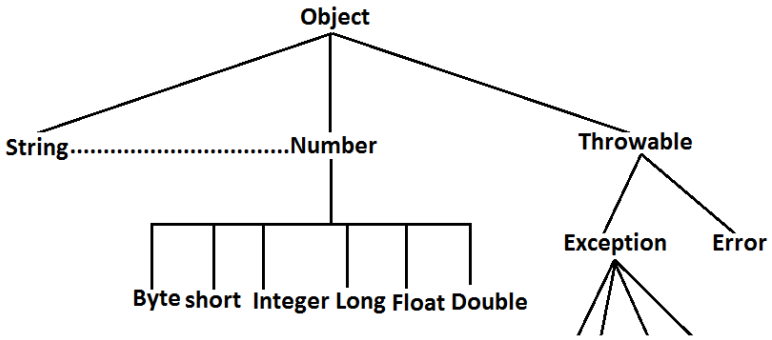
- The common methods which are required for housing loan, vehicle loan, personal loan and education loan we can define into a separate class in parent class loan. So that automatically these methods are available to every child loan class.

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.

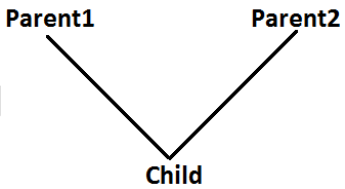
Diagram:



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)

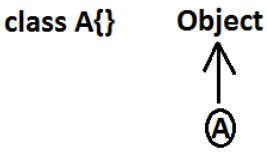
- But an interface can extends any no. Of interfaces at a time hence java provides support for multiple inheritance through interfaces.

Example:

```
interface A{}
interface B{}
interface C extends A,B{}
```

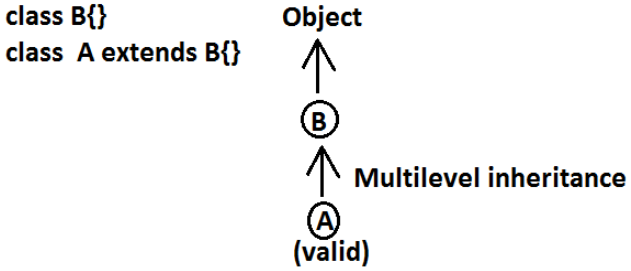
- If our class doesn't extends any other class then only our class is the direct child class of object.

Example:

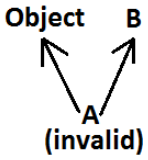


- If our class extends any other class then our class is not direct child class of object.

Example 1:



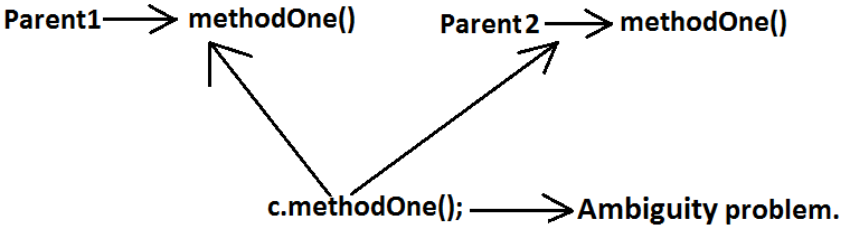
Example 2:



Why java won't provide support for multiple inheritance?

- There may be a chance of raising ambiguity problems.

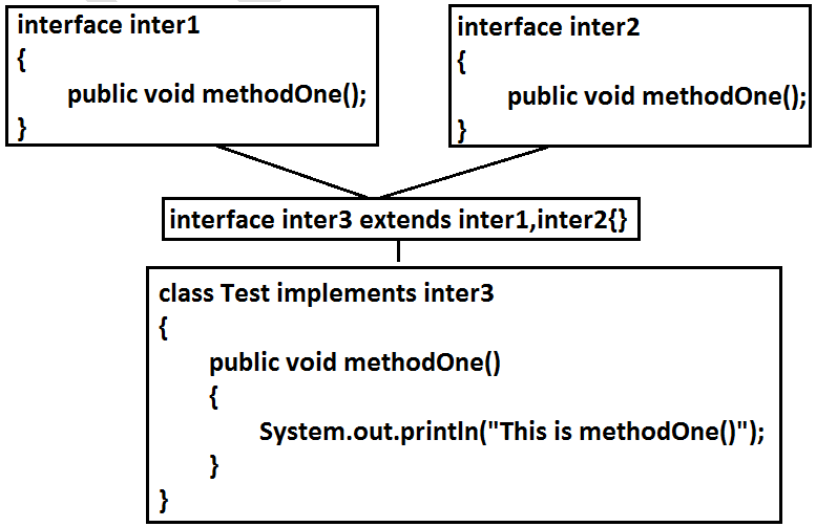
Example:



Why ambiguity problem won't be there in interfaces?

- Interfaces having dummy declarations and they won't have implementations hence no ambiguity problem.

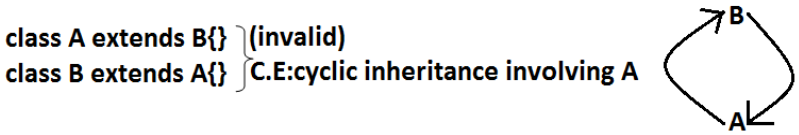
Example:



Cyclic inheritance:

- Cyclic inheritance is not allowed in java.

Example 1:



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.
- HAS-A relationship increases dependency between the components and creates maintains problems.

Composition vs Aggregation:

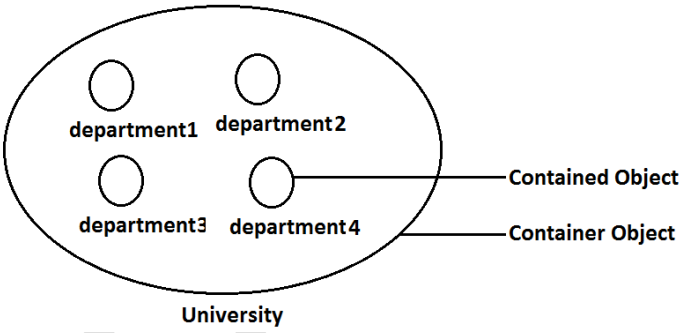
Composition:

- 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:



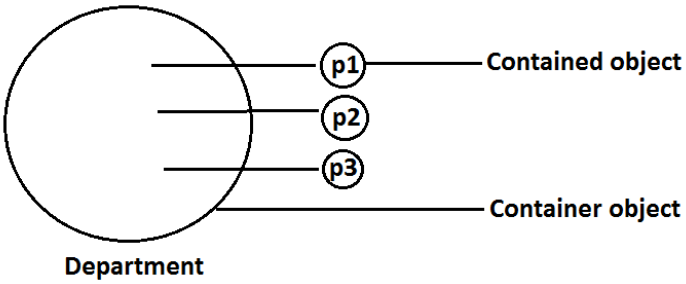
Aggregation:

- 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:

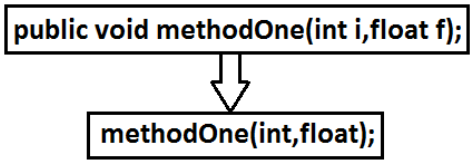
- Within a department there may be a chance of several professors will work whenever we are closing department still there may be a chance of existing professor object without existing department object the relationship between department and professor is called aggregation where the objects having weak association.

Example:



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.
- 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

Overloading:

- Two methods are said to be overload if and only if both having the same name but different argument types.
- 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

- Lack of overloading in "C" increases complexity of the programming.
- But in java we can take multiple methods with the same name and different argument types.

Example:

abs(int)
abs(long)
abs(float)
.
.
.

- Having the same name and different argument types is called method overloading.
- All these methods are considered as overloaded methods.
- Having overloading concept in java reduces complexity of the programming.

Example:

```
class Test
{
    public void methodOne()
    {
        System.out.println("no-arg method");
    }
    public void methodOne(int i)
    {
        System.out.println("int-arg method");
    }
    public void methodOne(double d)
    {
        System.out.println("double-arg method");
    }
    public static void main(String[] args)
    {
        Test t=new Test();
        t.methodOne();//no-arg method
        t.methodOne(10);//int-arg method
```

overloaded methods

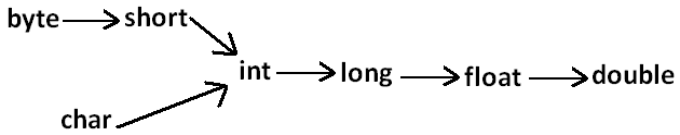
```
                t.methodOne(10.5);//double-arg method
        }
}
```

- In overloading compiler is responsible to perform method resolution(decision) based on the reference type. Hence overloading is also considered as compile time polymorphism(or) static ()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("bhaskar");//String version
                t.methodOne(new Object());//Object version
                t.methodOne(null);//String version
        }
}
```

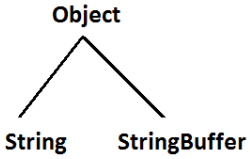
- In overloading Child will always get high priority then Parent.

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("durga");//String version
        t.methodOne(new StringBuffer("bhaskar"));//StringBuffer version
    }
}
```

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: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
    }
}
```

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:

- 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.
- The Parent class method which is overridden is called overridden method.
- The Child class method which is overriding is called overriding method.

Example 1:

```
class Parent
{
    public void property()
    {
        System.out.println("cash+land+gold");
    }
    public void marry()
    {
        System.out.println("subbalakshmi");     overridden method
    }
}
class Child extends Parent                                    overriding
{
    public void marry()
    {
        System.out.println("Trisha/nayanatara/anushka");      overriding method
    }
}
class Test
{
    public static void main(String[] args)
    {
        Parent p=new Parent();
        p.marry();//subbalakshmi(parent method)
        Child c=new Child();
        c.marry();//Trisha/nayanatara/anushka(child method)
        Parent p1=new Child();
        p1.marry();//Trisha/nayanatara/anushka(child method)
    }
}
```

- 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.
- 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:

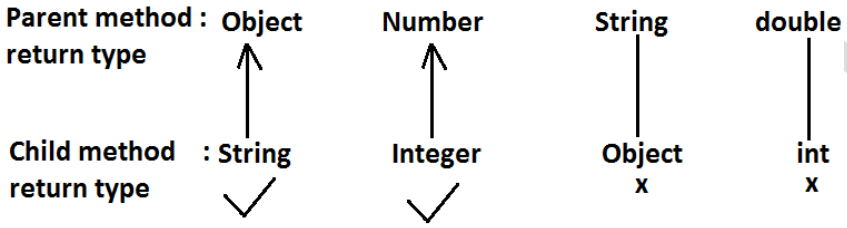
- In overriding method names and arguments must be same. That is method signature must be same.
- Until 1.4 version the return types must be same but from 1.5 version onwards co-variant return types are allowed.
- According to this Child class method return type need not be same as Parent class method return type its Child types also allowed.

Example:

```
class Parent
{
	public Object methodOne()
	{
		return null;
	}
}
class Child extends Parent
{
	public String methodOne()
	{
		return null;
	}
}
```

- 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.

Example:

```
class Parent
{
	public final void methodOne()
	{}
}
class Child extends Parent
{
	public void methodOne()
	{}
}
```

Output:

Compile time error.

Child.java:8: methodOne() in Child cannot override methodOne() in Parent; overridden method is final

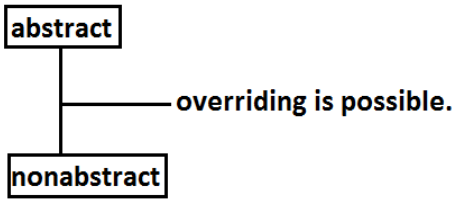
- Parent class non final methods we can override as final in child class. We can override native methods in the child classes.
- We should override Parent class abstract methods in Child classes to provide implementation.

Example:

```
abstract class Parent
{
```

```
    public abstract void methodOne();
}
class Child extends Parent
{
    public void methodOne()
    {}
}
```

Diagram:



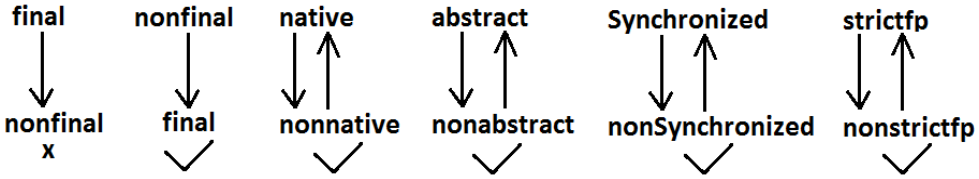
- We can override Parent class non abstract method as abstract to stop availability of Parent class method implementation to the Child classes.

Example:

```
class Parent
{
   public void methodOne()
   {}
}
abstract class Child extends Parent
{
   public abstract void methodOne();
}
```

- Synchronized, strictfp, modifiers won't keep any restrictions on overriding.

Diagram:



- While overriding we can't reduce the scope of access modifier.

Example:

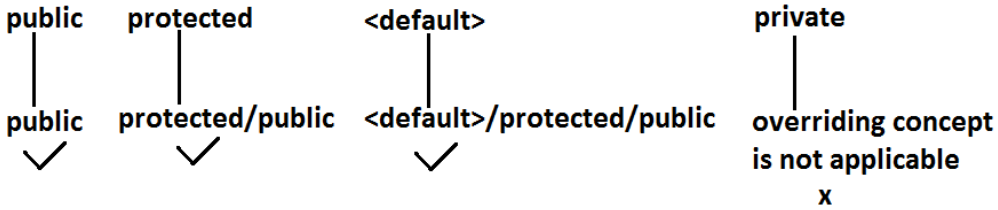
```
class Parent
{
   public void methodOne()
   {}
}
class Child extends Parent
{
   protected void methodOne()
   {}
}
```

Output:

Compile time error

methodOne() in Child cannot override methodOne() in Parent; attempting to assign weaker access privileges; was public

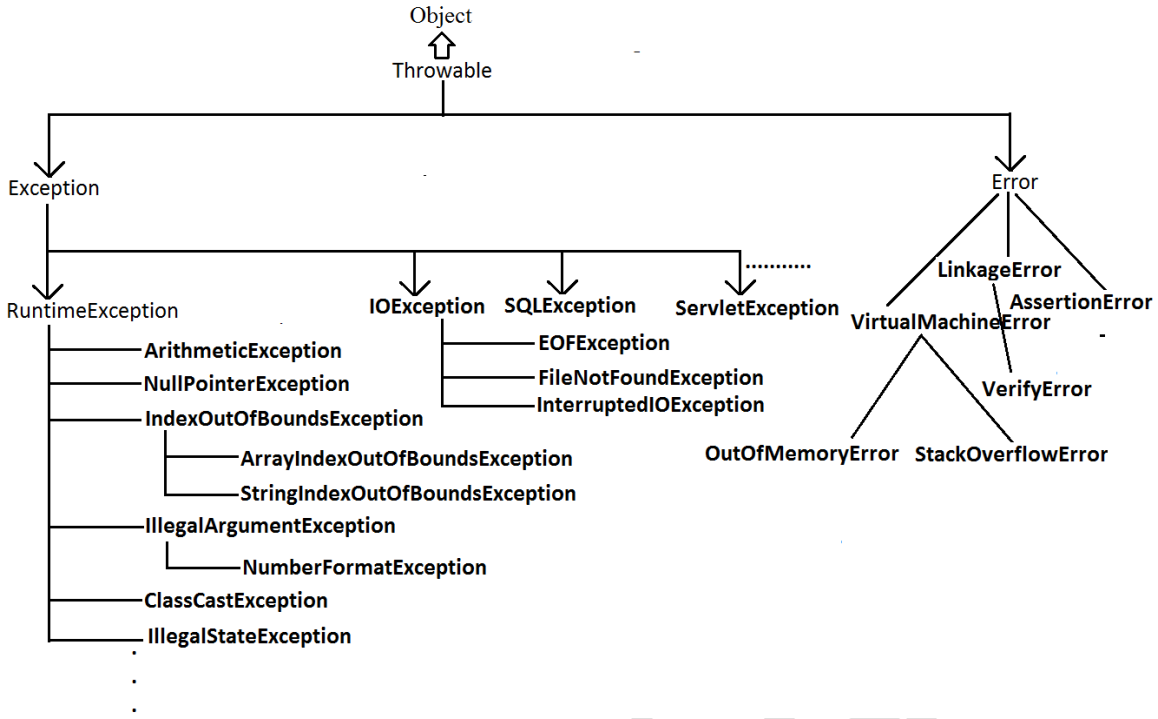
Diagram:



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
{}
}
```

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 met1hod 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	3. Method hiding is also considered as compile time polymorphism (or) static polymorphism (or) early

binding.	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
{
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
}
}
```

overloading but not overriding.

- 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	1) Must be same.	1) Must be same.
2) Argument type	2) Must be different(at least order)	2) Must be same including order.
3) Method signature	3) Must be different.	3) Must be same.
4) Return types	4) No restrictions.	4) Must be same until 1.4v but from 1.5v onwards we can take co-variant return types also.
5) private,static,final methods	5) Can be overloaded.	5) Can not be overridden.
6) Access modifiers	6) No restrictions.	6) Weakering/reducing is not allowed.
7) Throws clause	7) No restrictions.	7) 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	8) Is always takes care by compiler based on referenced type.	8) Is always takes care by JVM based on runtime object.
9) Also known as	9) Compile time polymorphism (or) static(or)early binding.	9) 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

1) 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

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 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:

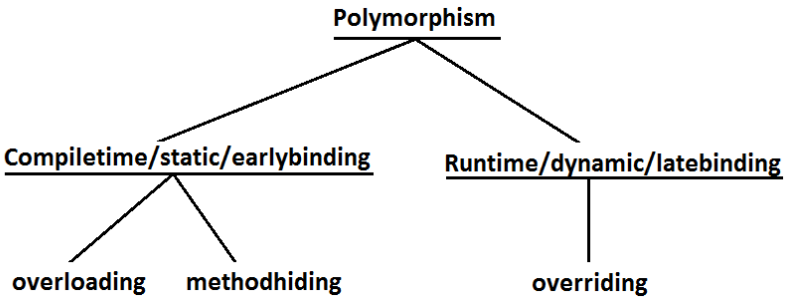
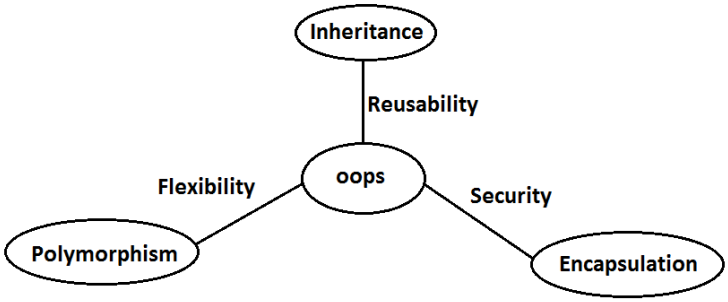


Diagram:



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.

Constructors

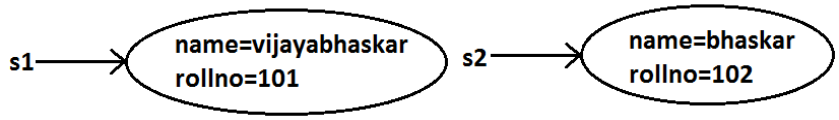
- Object creation is not enough compulsory we should perform initialization then only the object is in a position to provide the response properly.
- Whenever we are creating an object some piece of the code will be executed automatically to perform initialization of an object this piece of the code is nothing but constructor.
- 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)
    {
        this.name=name;          Constructor
        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:

- Both instance block and constructor will be executed automatically for every object creation but instance block 1st followed by constructor.
- The main objective of constructor **is to perform initialization of an object**.
- Other than initialization **if we want to perform any activity for every object creation we have to define that activity inside instance block**.
- Both concepts having different purposes hence replacing one concept with another concept is not possible.
- Constructor can take arguments but instance block can't take any arguments **hence we can't replace constructor concept with instance block**.
- 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.

Example:

```
class Test
{
    void Test()    it is not a constructor and it is a method
    {}
}
```

3) It is legal (but stupid) to have a method whose name is exactly same as class name.
4) The only applicable modifiers for the constructors are **public, default, private, protected**.
5) 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:

- For every class in java including abstract classes also constructor concept is applicable.
- If we are not writing at least one constructor then compiler will generate default constructor.
- 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:

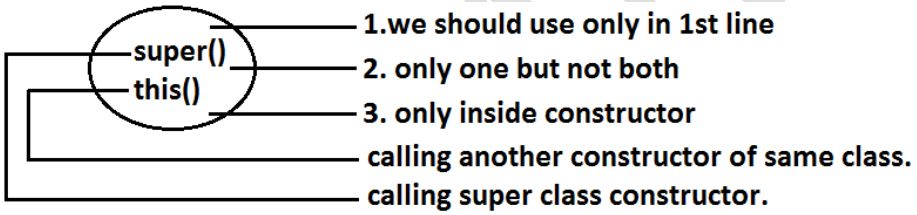
```
class Test
{
	public void methodOne()
	{
		super();
	}
}
```

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
1. These are constructors calls.	1. These are keywords which can be used to call parent class and current class instance members.
2. We should use only inside constructors.	2. We can use anywhere except static area.

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)
	{
		this(10);
		System.out.println("double-argument constructor");
	}
	Test(int i)
	{
		this();
```

```
		System.out.println("int-argument constructor");
	}
	Test()
	{
		System.out.println("no-argument constructor");
	}
	public static void main(String[] args)
	{
		Test t1=new Test(10.5);//no-argument constructor/int-argument constructor/double-argument constructor
		Test t2=new Test(10);//no-argument constructor/int-argument constructor
		Test t3=new Test();//no-argument constructor
	}
}
```

- "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 inheritance**.

Example:

```
class Test
{
    Test()
    {}
}
  valid
```

```
abstract class Test
{
    Test()
    {}
}
  valid
```

```
interface Test1
{
    Test1()
    {}
}
  invalid
```

We can't create object for abstract class but abstract class can contain constructor what is the need?

- Abstract class constructor will be executed to perform initialization of child class object.

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
    }
}
```

Case 1: 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:

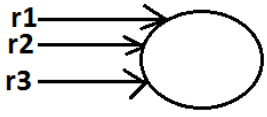
- For any java class if we are allow to crate only one object such type of class is said to be singleton class.

Example:

1) Runtime class
2) ActionServlet
3) ServiceLocator

```
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:



- 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.

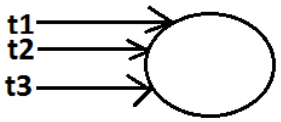
Creation of our own singleton classes:

- We can create our own singleton classes for this we have to use private constructor 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:

- We can create any xxxton classes like(double ton,trible ton….etc)

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)
				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
    }
}
```

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
{
  1 static int i=10;                                   ---- 7
  2 static
    {
        methodOne();                                   ---- 8
        System.out.println("first static block");      ---- 10
    }
  3 public static void main(String[] args)
    {
        methodOne();                                   ---- 13
        System.out.println("main method");             ---- 15
    }
  4 public static void methodOne()
        {
            System.out.println(j);                     ---- 9, 14
        }
    5 static
        {
            System.out.println("second static block");  ---- 11
        }
    6 static 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:

- 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

Example:

```
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
   }
```

Analysis:

i=0[RIWO]
j=0[RIWO]
x=0[RIWO]
y=0[RIWO]
i=10[R&w]
j=20[R&w]
x=100[R&w]
y=200[R&w]

Output:

```
E:\scjp>java Derived
0
Base static block
0
Derived first static block
Derived second static block
200
Derived 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].

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");
	}
}
```

- 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
	}
}
```

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
```

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

Without using System.out.println() statement is it possible to print some statement to the console?

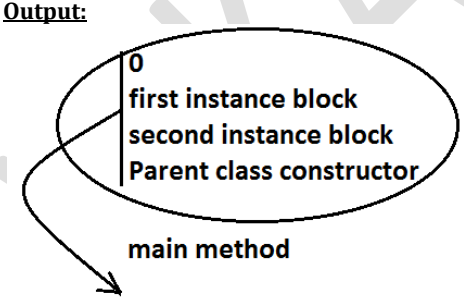
Example:

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

Instance control flow:

```
class Parent
{
③ int i= 10;                                        ⑨
④ {
        methodOne();                                ⑩
        System.out.println("first instance block");  ⑫
  }
⑤ Parent()
  {
        System.out.println("Parent class constructor"); ⑮
  }
  public static void main(String[] args)            ①
  {
        Parent p=new Parent();                      ②
        System.out.println("main method");          ⑯
  }
⑥ public void methodOne()
  {
        System.out.println(j);                      ⑪
  }
⑦ {
        System.out.println("second instance block"); ⑬
  }
⑧ int j =20;                                        ⑭
}
```

Analysis:
i=0[RIWO]
j=0[RIWO]
i=10[R&W]
j=20[R&W]
Output:



- 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 and 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.

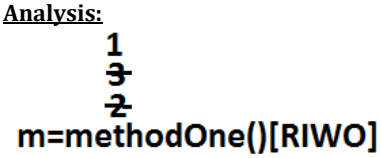
Note: Object creation is the most costly operation in java and hence if there is no specific requirement never recommended to crate objects.

Example 1:

```
public class Initilization
{
        private static String methodOne(String msg)
```

```
        {
                System.out.println(msg);
                return msg;
        }
        public Initilization()
        {
                m=methodOne("1");
        }
        {
                m=methodOne("2");
        }
        String m=methodOne("3");
        public static void main(String[] args)
        {
                Object obj=new Initilization();
        }
}
```

Analysis:



Output:

2
3
1

Example 2:

```
public class Initilization
{
        private static String methodOne(String msg)
        {
                System.out.println(msg);
                return msg;
        }
        static String m=methodOne("1");
        {
                m=methodOne("2");
        }
        static
        {
                m=methodOne("3");
        }
        public static void main(String[] args)
        {
                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:

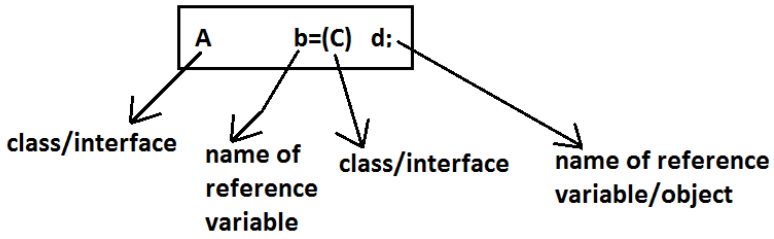
Object o=new String("bhaskar");//valid
System.out.println(o.hashCode());//valid
System.out.println(o.length());//C.E:cannot find symbol,symbol : method length(),location: class java.lang.Object

- 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;
```

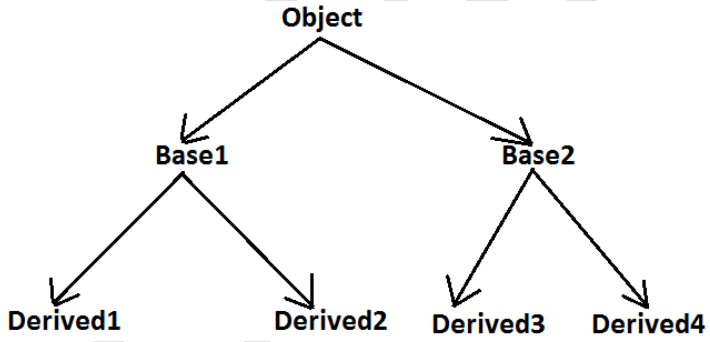
Runtime checking:

- The underlying object type of "d" must be either same (or) derived type of "C" otherwise we will get runtime exception saying ClassCastException.

Example:

```
Object o=new String("bhaskar");
StringBuffer sb=(StringBuffer)o;
```
→ Runtime Exception: ClassCastException

Diagram:



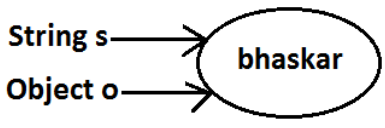

```
Base1 b=new Derived2();//valid
Object o=(Base1)b;//valid
Object o1=(Base2)o;//invalid
Object o2=(Base2)b;//invalid
Base2 b1=(Base1)(new Derived1());//invalid
Base2 b2=(Base2)(new Derived3());//valid
Base2 b2=(Base2)(new Derived1());//invalid
```

- 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.

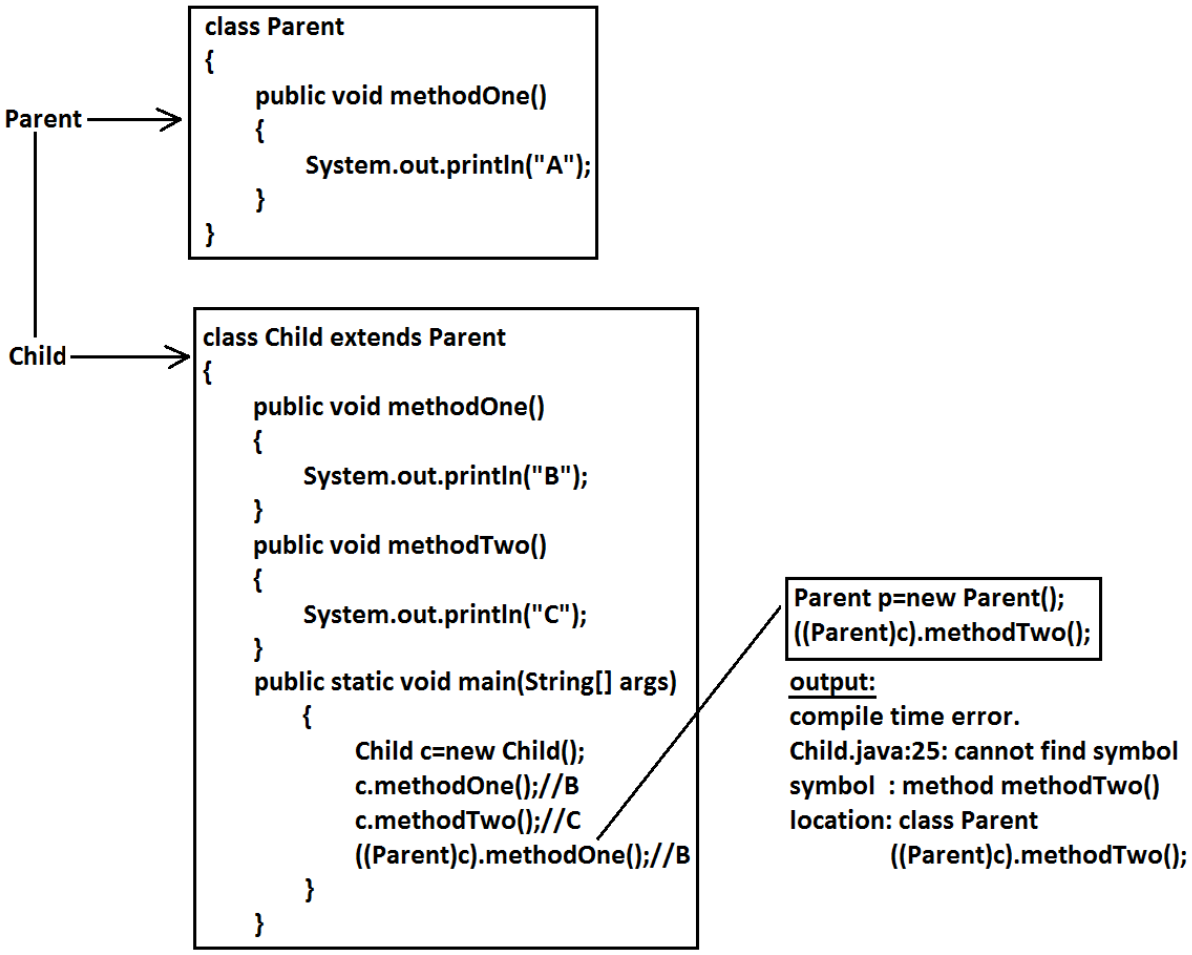
Example:

equivalent code:

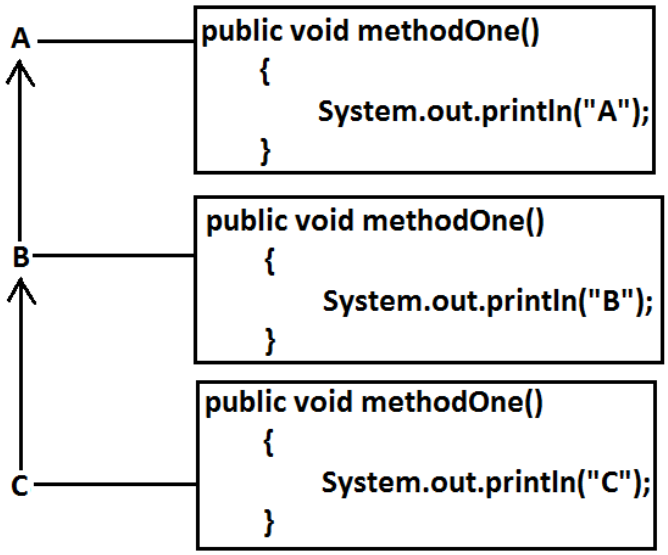
```
String s=new String("bhaskar");
Object o=(Object)s;
System.out.println(s==o);//true
Object o=new String("bhaskar");
```



Example 1:



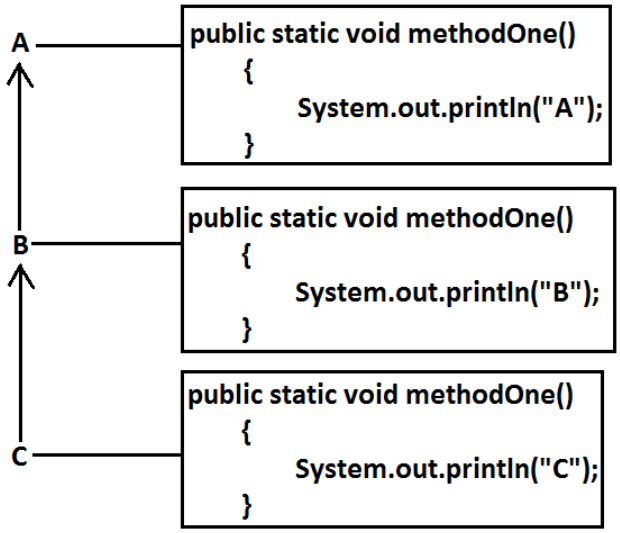
Example 2:



- 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

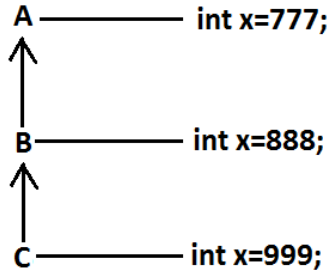
Example 3:



- 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.

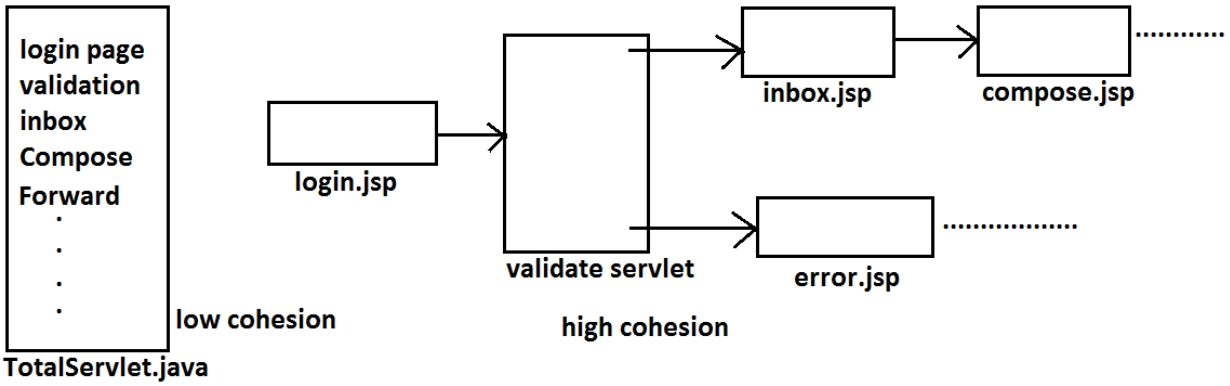
1) Without effecting remaining components we can't modify any component hence enhancement(development) will become difficult.
2) It reduces maintainability of the application.
3) It doesn't promote reusability of the code.

- 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.

Note: It is highly recommended to follow loosely coupling and high cohesion.

Multi Threading

Agenda

1) Introduction.
2) The ways to define instantiate and start a new Thread.
3) Getting and setting name of a Thread.
4) Thread priorities.
5) The methods to prevent(stop) Thread execution.
   1. yield()
   2. join()
   3. sleep()
6) Synchronization.
7) Inter Thread communication.
8) Deadlock
9) Daemon Threads.

Multitasking: Executing several tasks simultaneously is the concept of multitasking. There are two types of multitasking's.

1) **Process based multitasking.**
2) **Thread based multitasking.**

Diagram:

Process based multitasking: Executing several tasks simultaneously where each task is a separate independent process such type of multitasking is called process based multitasking.

Example:

- While typing a java program in the editor we can able to listen mp3 audio songs at the same time we can download a file from the net all these tasks are independent of each other and executing simultaneously and hence it is Process based multitasking.
- This type of multitasking is best suitable at "**os level**".

Thread based multitasking: Executing several tasks simultaneously where each task is a separate independent part of the same program, is called Thread based multitasking. And each independent part is called a "Thread".

- This type of multitasking is best suitable for "programatic level".
- When compared with "C++", developing multithreading examples is very easy in java because java provides in built support for multithreading through a rich API (Thread, Runnable, ThreadGroup, ThreadLocal....etc).
- In multithreading on 10% of the work the programmer is required to do and 90% of the work will be down by java API.
- The main important application areas of multithreading are:

1) To implement multimedia graphics.
2) To develop animations.
3) To develop video games etc.

- Whether it is process based or Thread based the main objective of multitasking is to improve performance of the system by reducing response time.

The ways to define instantiate and start a new Thread:

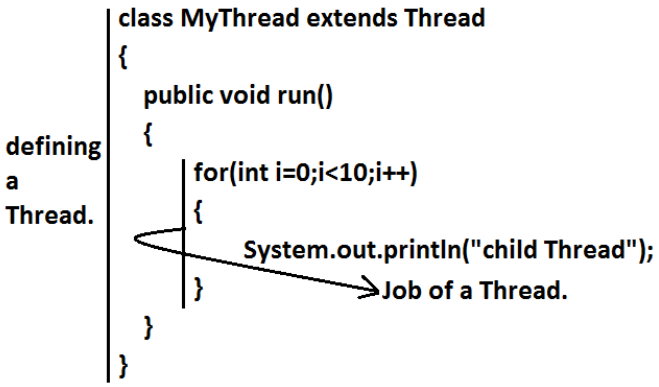
What is singleton? Give example?

- We can define a Thread in the following 2 ways.

1. By extending Thread class.
2. By implementing Runnable interface.

Defining a Thread by extending "Thread class":

Example:




```
class ThreadDemo
{
    public static void main(String[] args)
    {
            MyThread t=new MyThread();//Instantiation of a Thread
            t.start();//starting of a Thread
            for(int i=0;i<5;i++)
            {
                System.out.println("main thread");
            }
    }
}
```

Case 1: Thread Scheduler:

- If multiple Threads are waiting to execute then which Thread will execute 1st is decided by "Thread Scheduler" which is part of JVM.
- Which algorithm or behavior followed by Thread Scheduler we can't expect exactly it is the JVM vendor dependent hence in multithreading examples we can't expect exact execution order and exact output.
- The following are various possible outputs for the above program.

p1	p2	p3
main thread	main thread	main thread
main thread	main thread	main thread
main thread	main thread	main thread
main thread	main thread	main thread
main thread	main thread	main thread
child thread	child thread	child thread
child thread	child thread	child thread
child thread	child thread	child thread
child thread	child thread	child thread
child thread	child thread	child thread
child thread	child thread	child thread
child thread	child thread	child thread
child thread	child thread	child thread
child thread	child thread	child thread
child thread	child thread	child thread
child thread	child thread	child thread

Case 2: Difference between t.start() and t.run() methods.

- In the case of t.start() a new Thread will be created which is responsible for the execution of run() method. But in the case of t.run() no new Thread will be created and run() method will be executed just like a normal method by the main Thread. In the above program if we are replacing t.start() with t.run() the following is the output.

Output:

```
child thread
child thread
child thread
child thread
child thread
child thread
child thread
child thread
child thread
child thread
child thread
main thread
main thread
main thread
main thread
main thread
```

- Entire output produced by only main Thread.

Case 3: importance of Thread class start() method.

- For every Thread the required mandatory activities like registering the Thread with Thread Scheduler will takes care by Thread class start() method and programmer is responsible just to define the job of the Thread inside run() method. That is start() method acts as best assistant to the programmer.

Example:

```
start()
{
    1. Register Thread with Thread Scheduler
    2. All other mandatory low level activities.
    3. Invoke or calling run() method.
}
```

- We can conclude that without executing Thread class start() method there is no chance of starting a new Thread in java.

Case 4: If we are not overriding run() method:

- If we are not overriding run() method then Thread class run() method will be executed which has empty implementation and hence we won't get any output.

Example:

```
class MyThread extends Thread
{}
class ThreadDemo
{
    public static void main(String[] args)
    {
        MyThread t=new MyThread();
```

```
                t.start();
        }
}
```

- It is highly recommended to override run() method. Otherwise don't go for multithreading concept.

Case 5: Overriding of run() method.

- We can overload run() method but Thread class start() method always invokes no argument run() method the other overload run() methods we have to call explicitly then only it will be executed just like normal method.

Example:

```
class MyThread extends Thread
{
        public void run()
        {
                System.out.println("no arg method");
        }
        public void run(int i)
        {
                System.out.println("int arg method");
        }
}
class ThreadDemo
{
        public static void main(String[] args)
        {
                MyThread t=new MyThread();
                t.start();
        }
}
```

Output:

```
No arg method
```

Case 6: overriding of start() method:

- If we override start() method then our start() method will be executed just like a normal method call and no new Thread will be started.

Example:

```
class MyThread extends Thread
{
        public void start()
        {
                System.out.println("start method");
        }
        public void run()
        {
                System.out.println("run method");
        }
}
class ThreadDemo
{
        public static void main(String[] args)
        {
                MyThread t=new MyThread();
                t.start();
                System.out.println("main method");
        }
}
```

Output:

```
start method
main method
```

- Entire output produced by only main Thread.

Case 7:

Example 1:

```
class MyThread extends Thread
{
    public void start()
    {
        System.out.println("start method");
    }
    public void run()
    {
        System.out.println("run method");
    }
}
```

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

output:

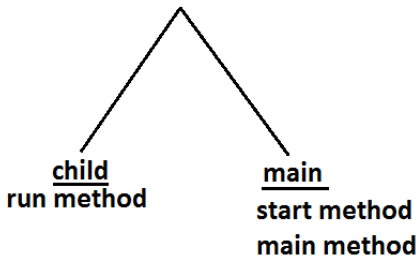
```
main thread
start method
main method
```

Example 2:

```
class MyThread extends Thread
{
    public void start()
    {
        super.start();
        System.out.println("start method");
    }
    public void run()
    {
        System.out.println("run method");
    }
}
```

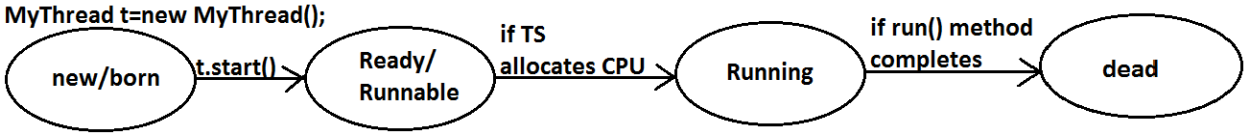
```
class ThreadDemo
{
    public static void main(String[] args)
    {
        MyThread t=new MyThread();
        t.start();
        System.out.println("main method");
    }
}
```

Output:



Case 8: life cycle of the Thread:

Diagram:



- Once we created a Thread object then the Thread is said to be in new state or born state.
- Once we call start() method then the Thread will be entered into Ready or Runnable state.
- If Thread Scheduler allocates CPU then the Thread will be entered into running state.

- Once run() method completes then the Thread will entered into dead state.

Case 9:

- After starting a Thread we are not allowed to restart the same Thread once again otherwise we will get runtime exception saying "IllegalThreadStateException".

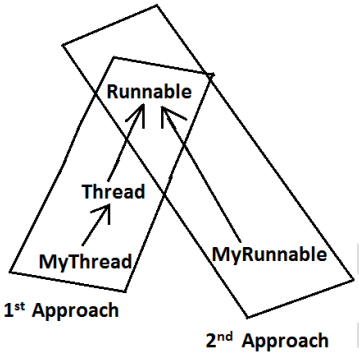
Example:

```
MyThread t=new MyThread();
t.start();//valid
;;;;;;;;
t.start();//we will get R.E saying: IllegalThreadStateException
```

Defining a Thread by implementing Runnable interface:

- We can define a Thread even by implementing Runnable interface also. Runnable interface present in java.lang.pkg and contains only one method run().

Diagram:



Example:

```
class MyRunnable implements Runnable
{
    public void run()
    {
        for(int i=0;i<10;i++)
        {
            System.out.println("child Thread");
        }
    }
}
```

defining a Thread

job of a Thread

```
class ThreadDemo
{
    public static void main(String[] args)
    {
        MyRunnable r=new MyRunnable();
        Thread t=new Thread(r);//here r is a Target Runnable
        t.start();
        for(int i=0;i<10;i++)
        {
            System.out.println("main thread");
        }
    }
}
```

Output:

```
main thread
main thread
main thread
main thread
main thread
main thread
main thread
main thread
main thread
```

main thread
child Thread
child Thread
child Thread
child Thread
child Thread
child Thread
child Thread
child Thread
child Thread
child Thread

- We can't expect exact output but there are several possible outputs.

Case study:

```
MyRunnable r=new MyRunnable();
Thread t1=new Thread();
Thread t2=new Thread(r);
```

Case 1: t1.start():

- A new Thread will be created which is responsible for the execution of Thread class run()method.

Output:

main thread
main thread
main thread
main thread
main thread

Case 2: t1.run():

- No new Thread will be created but Thread class run() method will be executed just like a normal method call.

Output:

main thread
main thread
main thread
main thread
main thread

Case 3: t2.start():

- New Thread will be created which is responsible for the execution of MyRunnable run() method.

Output:

main thread
main thread
main thread
main thread
main thread
child Thread
child Thread
child Thread
child Thread
child Thread

Case 4: t2.run():

- No new Thread will be created and MyRunnable run() method will be executed just like a normal method call.

Output:

child Thread
child Thread
child Thread
child Thread
child Thread
main thread
main thread
main thread
main thread
main thread

Case 5: r.start():

- We will get compile time error saying start()method is not available in MyRunnable class.

Output:

Compile time error
E:\SCJP>javac ThreadDemo.java
ThreadDemo.java:18: cannot find symbol
Symbol: method start()

Location: class MyRunnable

Case 6: r.run():

- No new Thread will be created and MyRunnable class run() method will be executed just like a normal method call.

Output:

```
child Thread
child Thread
child Thread
child Thread
child Thread
main thread
main thread
main thread
main thread
main thread
```

Best approach to define a Thread:

- Among the 2 ways of defining a Thread, implements Runnable approach is always recommended.
- In the 1st approach our class should always extends Thread class there is no chance of extending any other class hence we are missing the benefits of inheritance.
- But in the 2nd approach while implementing Runnable interface we can extend some other class also. Hence implements Runnable mechanism is recommended to define a Thread.

Thread class constructors:

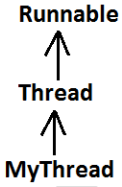
1) Thread t=new Thread();
2) Thread t=new Thread(Runnable r);
3) Thread t=new Thread(String name);
4) Thread t=new Thread(Runnable r,String name);
5) Thread t=new Thread(ThreadGroup g,String name);
6) Thread t=new Thread(ThreadGroup g,Runnable r);
7) Thread t=new Thread(ThreadGroup g,Runnable r,String name);
8) Thread t=new Thread(ThreadGroup g,Runnable r,String name,long stackSize);

Durga's approach to define a Thread(not recommended to use):

```
class MyThread extends Thread
{
    public void run()
    {
        System.out.println("run method");
    }
}
```

```
class ThreadDemo
{
    public static void main(String[] args)
    {
        MyThread t=new MyThread();
        Thread t1=new Thread(t);
        t1.start();
        System.out.println("main method");
    }
}
```

Diagram:



Output:

```
main method
run method
```

Getting and setting name of a Thread:

- Every Thread in java has some name it may be provided explicitly by the programmer or automatically generated by JVM.
- Thread class defines the following methods to get and set name of a Thread.

Methods:

1) **public final String getName()**
2) **public final void setName(String name)**

Example:

```
class MyThread extends Thread
{}
class ThreadDemo
{
```

```
        public static void main(String[] args)
        {
                System.out.println(Thread.currentThread().getName());//main
                MyThread t=new MyThread();
                System.out.println(t.getName());//Thread-0
                Thread.currentThread().setName("Bhaskar Thread");
                System.out.println(Thread.currentThread().getName());//Bhaskar Thread
        }
}
```

Note: We can get current executing Thread object reference by using Thread.currentThread() method.

Thread Priorities

- Every Thread in java has some priority it may be default priority generated by JVM (or) explicitly provided by the programmer.
- The valid range of Thread priorities is 1 to 10[but not 0 to 10] where 1 is the least priority and 10 is highest priority.
- Thread class defines the following constants to represent some standard priorities.

1) **Thread. MIN_PRIORITY----------1**
2) **Thread. MAX_PRIORITY----------10**
3) **Thread. NORM_PRIORITY--------5**

- There are no constants like Thread.LOW_PRIORITY, Thread.HIGH_PRIORITY
- Thread scheduler uses these priorities while allocating CPU.
- The Thread which is having highest priority will get chance for 1st execution.
- If 2 Threads having the same priority then we can't expect exact execution order it depends on Thread scheduler whose behavior is vendor dependent.
- We can get and set the priority of a Thread by using the following methods.

1) **public final int getPriority()**
2) **public final void setPriority(int newPriority);//the allowed values are 1 to 10**

- The allowed values are 1 to 10 otherwise we will get runtime exception saying "IllegalArgumentException".

Default priority:

- The default priority only for the main Thread is 5. But for all the remaining Threads the default priority will be inheriting from parent to child. That is whatever the priority parent has by default the same priority will be for the child also.

Example 1:

```
class MyThread extends Thread
{}
class ThreadPriorityDemo
{
    public static void main(String[] args)
    {
            System.out.println(Thread.currentThread().getPriority());//5
            Thread.currentThread().setPriority(9);
            MyThread t=new MyThread();
            System.out.println(t.getPriority());//9
    }
}
```

Example 2:

```
class MyThread extends Thread
{
    public void run()
    {
            for(int i=0;i<10;i++)
            {
                    System.out.println("child thread");
            }
    }
}
class ThreadPriorityDemo
{
    public static void main(String[] args)
    {
            MyThread t=new MyThread();
            //t.setPriority(10);          ------------> 1
            t.start();
            for(int i=0;i<10;i++)
            {
                    System.out.println("main thread");
            }
    }
}
```

- If we are commenting line 1 then both main and child Threads will have the same priority and hence we can't expect exact execution order.
- If we are not commenting line 1 then child Thread has the priority 10 and main Thread has the priority 5 hence child Thread will get chance for execution and after completing child Thread main Thread will get the chance in this the output is:

Output:

```
child thread
child thread
child thread
child thread
child thread
child thread
child thread
child thread
child thread
child thread
main thread
main thread
main thread
main thread
main thread
main thread
main thread
main thread
main thread
main thread
main thread
```

- Some operating systems(like windowsXP) may not provide proper support for Thread priorities. We have to install separate bats provided by vendor to provide support for priorities.

The Methods to Prevent a Thread from Execution:

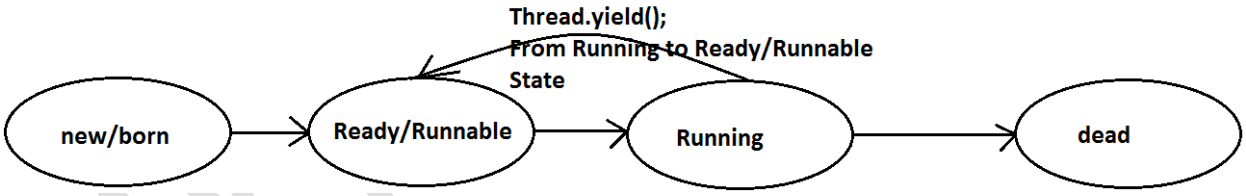
- We can prevent(stop) a Thread execution by using the following methods.

1) **yield();**
2) **join();**
3) **sleep();**

yield():

- yield() method causes "to pause current executing Thread for giving the chance of remaining waiting Threads of same priority".
- If all waiting Threads have the low priority or if there is no waiting Threads then the same Thread will be continued its execution.
- If several waiting Threads with same priority available then we can't expect exact which Thread will get chance for execution.
- The Thread which is yielded when it get chance once again for execution is depends on mercy of the Thread scheduler.
- public static native void **yield**();

Diagram:



Example:

```
class MyThread extends Thread
{
	public void run()
	{
		for(int i=0;i<5;i++)
		{
			Thread.yield();
			System.out.println("child thread");
		}
	}
}
class ThreadYieldDemo
{
	public static void main(String[] args)
	{
		MyThread t=new MyThread();
```

```
                t.start();
                for(int i=0;i<5;i++)
                {
                        System.out.println("main thread");
                }
        }
}
```

Output:

```
main thread
main thread
main thread
main thread
main thread
child thread
child thread
child thread
child thread
child thread
```

- In the above example the chance of completing main Thread 1st is high because child Thread always calling yield() method.

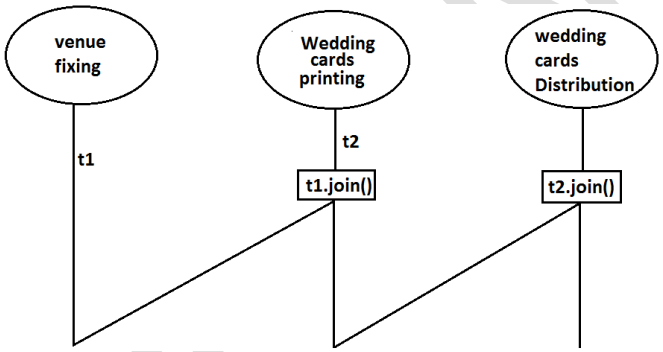
Join():

- If a Thread wants to wait until completing some other Thread then we should go for join() method.

Example:

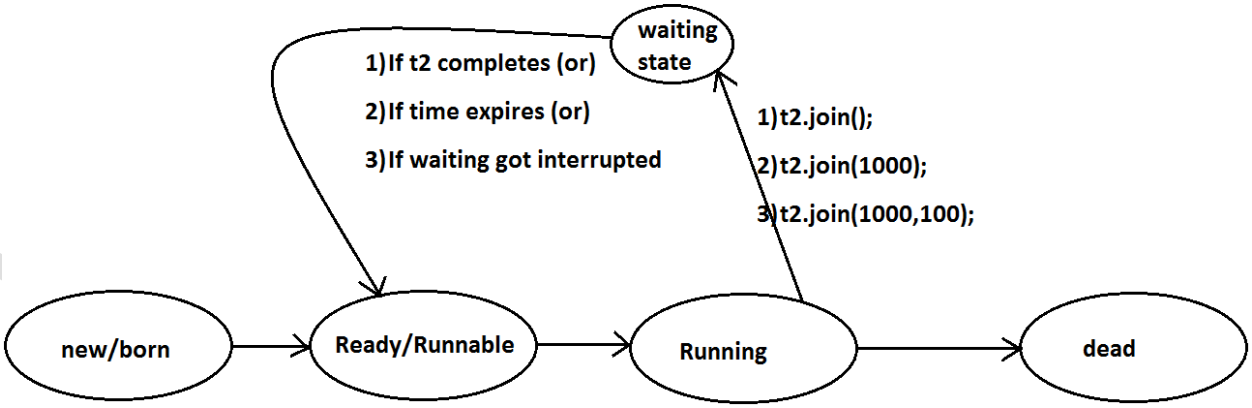
- If a Thread t1 executes t2.join() then t1 should go for waiting state until completing t2.

Diagram:



1) **public final void join()throws InterruptedException**
2) **public final void join(long ms) throws InterruptedException**
3) **public final void join(long ms,int ns) throws InterruptedException**

Diagram:



- Every join() method throws InterruptedException, which is checked exception hence compulsory we should handle either by try catch or by throws keyword.

Example:

```
class MyThread extends Thread
{
        public void run()
        {
                for(int i=0;i<5;i++)
                {
                        System.out.println("Seethe Thread");
```

```
                    try
                    {
                        Thread.sleep(2000);
                    }
                    catch (InterruptedException e){}
                }
            }
        }
        class ThreadJoinDemo
        {
            public static void main(String[] args)throws InterruptedException
            {
                MyThread t=new MyThread();
                t.start();
                //t.join();          ------------> 1
                for(int i=0;i<5;i++)
                {
                    System.out.println("Rama Thread");
                }
            }
        }
```

- If we are commenting line 1 then both Threads will be executed simultaneously and we can't expect exact execution order.
- If we are not commenting line 1 then main Thread will wait until completing child Thread in this the output is seethe Thread 5 times followed by Rama Thread 5 times.

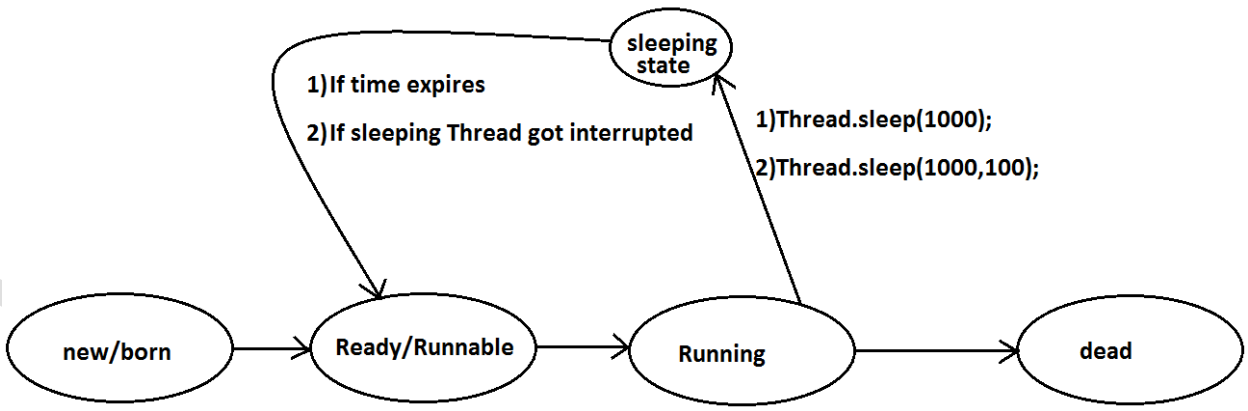
Sleep() method:

- If a Thread don't want to perform any operation for a particular amount of time then we should go for sleep() method.

1) public static native void sleep(long ms) throws InterruptedException

2) public static void sleep(long ms,int ns)throws InterruptedException

Diagram:



Example:

```
class ThreadJoinDemo
{
    public static void main(String[] args)throws InterruptedException
    {
        System.out.println("M");
        Thread.sleep(3000);
        System.out.println("E");
        Thread.sleep(3000);
        System.out.println("G");
        Thread.sleep(3000);
        System.out.println("A");
```

```
        }
    }
```

Output:

```
M
E
G
A
```

Interrupting a Thread:

- We can interrupt a sleeping or waiting Thread by using interrupt()(break off) method of Thread class.

Example:

```
class MyThread extends Thread
{
    public void run()
    {
        try
        {
            for(int i=0;i<5;i++)
            {
                System.out.println("i am lazy Thread :"+i);
                Thread.sleep(2000);
            }
        }
        catch (InterruptedException e)
        {
            System.out.println("i got interrupted");
        }
    }
}
class ThreadInterruptDemo
{
    public static void main(String[] args)
    {
        MyThread t=new MyThread();
        t.start();
        //t.interrupt();          ----------> 1
        System.out.println("end of main thread");
    }
}
```

- If we are commenting line 1 then main Thread won't interrupt child Thread and hence child Thread will be continued until its completion.
- If we are not commenting line 1 then main Thread interrupts child Thread and hence child Thread won't continued until its completion in this case the output is:

```
End of main thread
I am lazy Thread: 0
I got interrupted
```

Note:

- Whenever we are calling interrupt() method we may not see the effect immediately, if the target Thread is in sleeping or waiting state it will be interrupted immediately.
- If the target Thread is not in sleeping or waiting state then interrupt call will wait until target Thread will enter into sleeping or waiting state. Once target Thread entered into sleeping or waiting state it will effect immediately.
- In its lifetime if the target Thread never entered into sleeping or waiting state then there is no impact of interrupt call simply interrupt call will be wasted.

Example:

```
class MyThread extends Thread
{
    public void run()
    {
        for(int i=0;i<5;i++)
        {
            System.out.println("iamlazy thread");
        }
        try
        {
            Thread.sleep(3000);
        }
        catch (InterruptedException e)
```

```
                {
                        System.out.println("i got interrupted");
                }
        }
}
class ThreadInterruptDemo1
{
        public static void main(String[] args)
        {
                MyThread t=new MyThread();
                t.start();
                t.interrupt();
                System.out.println("end of main thread");
        }
}
```

- In the above program interrupt() method call invoked by main Thread will wait until child Thread entered into sleeping state.
- Once child Thread entered into sleeping state then it will be interrupted immediately.

Compression of yield, join and sleep() method?

property	Yield()	Join()	Sleep()
1) Purpose?	To pause current executing Thread for giving the chance of remaining waiting Threads of same priority.	If a Thread wants to wait until completing some other Thread then we should go for join.	If a Thread don't want to perform any operation for a particular amount of time then we should go for sleep() method.
2) Is it static?	yes	no	yes
3) Is it final?	no	yes	no
4) Is it overloaded?	No	yes	yes
5) Is it throws 6) **InterruptedException?**	no	yes	yes
7) Is it native method?	yes	no	sleep(long ms) sleep(long ms,int ns)

Synchronization

- Synchronized is the keyword applicable for methods and blocks but not for classes and variables.
- If a method or block declared as the synchronized then at a time only one Thread is allow to execute that method or block on the given object.
- The main advantage of synchronized keyword is we can resolve date inconsistency problems.
- But the main disadvantage of synchronized keyword is it increases waiting time of the Thread and effects performance of the system.
- Hence if there is no specific requirement then never recommended to use synchronized keyword.
- Internally synchronization concept is implemented by using lock concept.
- Every object in java has a unique lock. Whenever we are using synchronized keyword then only lock concept will come into the picture.
- If a Thread wants to execute any synchronized method on the given object 1st it has to get the lock of that object. Once a Thread got the lock of that object then it's allow to execute any synchronized method on that object. If the synchronized method execution completes then automatically Thread releases lock.
- While a Thread executing any synchronized method the remaining Threads are not allowed execute any synchronized method on that object simultaneously. But remaining Threads are allowed to execute any non-synchronized method simultaneously. [lock concept is implemented based on object but not based on method].

Example:

```
class Display
{
        public synchronized void wish(String name)
        {
                for(int i=0;i<5;i++)
                {
                        System.out.print("good morning:");
                        try
                        {
                                Thread.sleep(1000);
                        }
                        catch (InterruptedException e)
                        {}
```

```
                        System.out.println(name);
                }
        }
}
class MyThread extends Thread
{
        Display d;
        String name;
        MyThread(Display d,String name)
        {
                this.d=d;
                this.name=name;
        }
        public void run()
        {
                d.wish(name);
        }
}
class SynchronizedDemo
{
        public static void main(String[] args)
        {
                Display d1=new Display();
                MyThread t1=new MyThread(d1,"dhoni");
                MyThread t2=new MyThread(d1,"yuvaraj");
                t1.start();
                t2.start();
        }
}
```

- If we are not declaring wish() method as synchronized then both Threads will be executed simultaneously and we will get irregular output.

  Output:

  good morning:good morning:yuvaraj
  good morning:dhoni
  good morning:yuvaraj
  good morning:dhoni
  good morning:yuvaraj
  good morning:dhoni
  good morning:yuvaraj
  good morning:dhoni
  good morning:yuvaraj
  dhoni
- If we declare wish()method as synchronized then the Threads will be executed one by one that is until completing the 1st Thread the 2nd Thread will wait in this case we will get regular output which is nothing but

  Output:

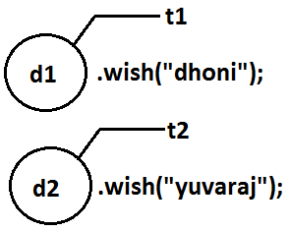
  good morning:dhoni
  good morning:dhoni
  good morning:dhoni
  good morning:dhoni
  good morning:dhoni
  good morning:yuvaraj
  good morning:yuvaraj
  good morning:yuvaraj
  good morning:yuvaraj
  good morning:yuvaraj

  Case study:

  Case 1:

```
Display d1=new Display();
Display d2=new Display();
MyThread t1=new MyThread(d1,"dhoni");
MyThread t2=new MyThread(d2,"yuvaraj");
t1.start();
t2.start();
```

Diagram:



- Even though we declared wish() method as synchronized but we will get irregular output in this case, because both Threads are operating on different objects.

Class level lock:

- Every class in java has a unique lock. If a Thread wants to execute a static synchronized method then it required class level lock.
- Once a Thread got class level lock then it is allow to execute any static synchronized method of that class.
- While a Thread executing any static synchronized method the remaining Threads are not allow to execute any static synchronized method of that class simultaneously.
- But remaining Threads are allowed to execute normal synchronized methods, normal static methods, and normal instance methods simultaneously.
- Class level lock and object lock both are different and there is no relationship between these two.

Synchronized block:

- If very few lines of the code required synchronization then it's never recommended to declare entire method as synchronized we have to enclose those few lines of the code with in synchronized block.
- The main advantage of synchronized block over synchronized method is it reduces waiting time of Thread and improves performance of the system.

Example 1: To get lock of current object we can declare synchronized block as follows.
Synchronized(this){}
Example 2: To get the lock of a particular object 'b' we have to declare a synchronized block as follows.
Synchronized(b){}
Example 3: To get class level lock we have to declare synchronized block as follows.
Synchronized(Display.class){}

- As the argument to the synchronized block we can pass either object reference or ".class file" and we can't pass primitive values as argument [because lock concept is dependent only for objects and classes but not for primitives].

Example:
Int x=b;
Synchronized(x){}
Output:
Compile time error.
Unexpected type.
Found: int
Required: reference

Questions:

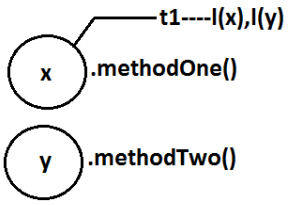
1) Explain about synchronized keyword and its advantages and disadvantages?
2) What is object lock and when a Thread required?
3) What is class level lock and when a Thread required?
4) What is the difference between object lock and class level lock?
5) While a Thread executing a synchronized method on the given object is the remaining Threads are allowed to execute other synchronized methods simultaneously on the same object?
   Ans: No.
6) What is synchronized block and explain its declaration?
7) What is the advantage of synchronized block over synchronized method?
8) Is a Thread can hold more than one lock at a time?
   Ans: Yes, up course from different objects.

Example:

```
class X
{
    synchronized void methodOne()
    {
        Y y=new Y();
        y.methodTwo();
    }
}
```

```
class Y
{
    synchronized void methodTwo()
    {}
}
```

Diagram:



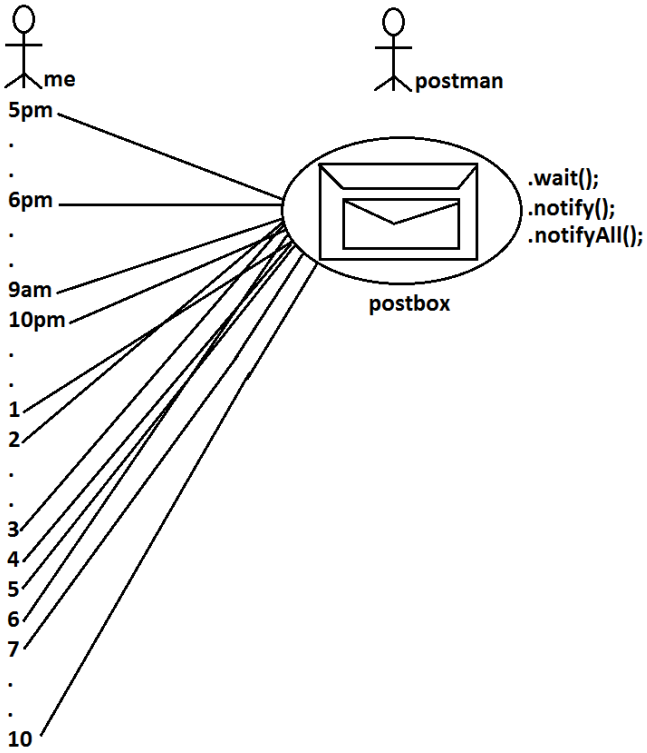
9) What is synchronized statement?

Ans: The statements which present inside synchronized method and synchronized block are called synchronized statements. [Interview people created terminology].

Inter Thread communication (wait(),notify(), notifyAll()):

- Two Threads can communicate with each other by using wait(), notify() and notifyAll() methods.
- The Thread which is excepting updation it has to call wait() method and the Thread which is performing updation it has to call notify() method. After getting notification the waiting Thread will get those updations.

Diagram:



- wait(), notify() and notifyAll() methods are available in Object class but not in Thread class because Thread can call these methods on any common object.
- To call wait(), notify() and notifyAll() methods compulsory the current Thread should be owner of that object that is current Thread should has lock of that object that is current Thread should be in synchronized area. Hence we can call wait(), notify() and notifyAll() methods only from synchronized area otherwise we will get runtime exception saying IllegalMonitorStateException.
- Once a Thread calls wait() method on the given object 1st it releases the lock of that object immediately and entered into waiting state.
- Once a Thread calls notify() (or) notifyAll() methods it releases the lock of that object but may not immediately.
- Except these (wait(),notify(),notifyAll()) methods there is no other place(method) where the lock release will be happen.

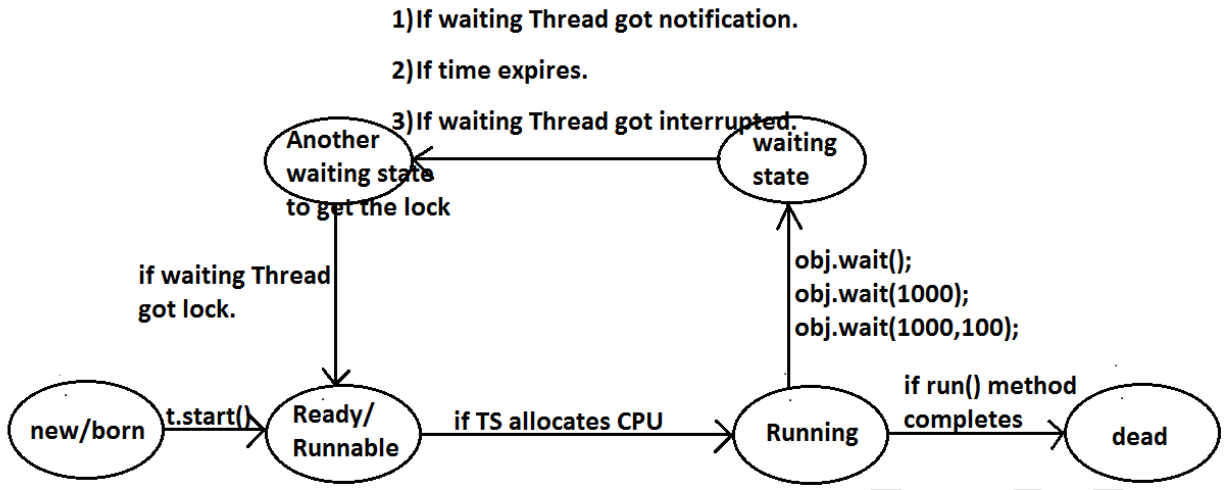
Method	Is Thread Releases Lock?
yield()	No
join()	No
sleep()	No
wait()	Yes
notify()	Yes
notifyAll()	Yes

- Once a Thread calls wait(), notify(), notifyAll() methods on any object then it releases the lock of that particular object but not all locks it has.

1) public final void wait()throws InterruptedException
2) public final native void wait(long ms)throws InterruptedException
3) public final void wait(long ms,int ns)throws InterruptedException

4) **public final native void notify()**
5) **public final void notifyAll()**

Diagram:



Example 1:

```
class ThreadA
{
	public static void main(String[] args)throws InterruptedException
	{
		ThreadB b=new ThreadB();
		b.start();
		synchronized(b)
		{
			System.out.println("main Thread calling wait() method");//step-1
			b.wait();
			System.out.println("main Thread got notification call");//step-4
			System.out.println(b.total);
		}
	}
}
class ThreadB extends Thread
{
	int total=0;
	public void run()
	{
		synchronized(this)
		{
			System.out.println("child thread starts calcuation");//step-2
			for(int i=0;i<=100;i++)
			{
				total=total+i;
			}
			System.out.println("child thread giving notification call");//step-3
			this.notify();
		}
	}
}
```

Output:

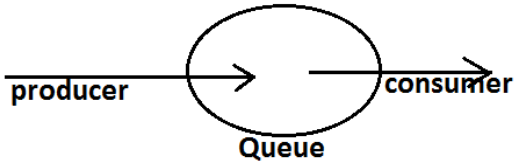
```
main Thread calling wait() method
child thread starts calculation
child thread giving notification call
main Thread got notification call
5050
```

Example 2:

Producer consumer problem:

- Producer(producer Thread) will produce the items to the queue and consumer(consumer thread) will consume the items from the queue. If the queue is empty then consumer has to call wait() method on the queue object then it will entered into waiting state.
- After producing the items producer Thread call notify() method on the queue to give notification so that consumer Thread will get that notification and consume items.

Diagram:



Example:

```
class Producer
{
    Producer()
    {
        synchronized(q)
        {
            produce items to the queue
                q.notify();
        }
    }
}
```

```
{
    Consumer()
    {
        synchronized(q)
        {
            if(q is empty)
            {
                q.wait();
            }
            else
                continue items;
        }
    }
}
```

Notify vs notifyAll():

- We can use notify() method to give notification for only one Thread. If multiple Threads are waiting then only one Thread will get the chance and remaining Threads has to wait for further notification. But which Thread will be notify(inform) we can't expect exactly it depends on JVM.
- We can use notifyAll() method to give the notification for all waiting Threads. All waiting Threads will be notified and will be executed one by one.

Note: On which object we are calling wait(), notify() and notifyAll() methods that corresponding object lock we have to get but not other object locks.

Example:

```
Stack s1=new Stack();
Stack s2=new Stack();
```

```
synchronized(s1)
{
    ;;;;;;;;;
    s2.wait();
    ;;;;;;;;;
}
(invalid)
```

```
synchronized(s1)
{
    ;;;;;;;;;
    s1.wait();
    ;;;;;;;;;
}
(valid)
```

R.E:IllegalMonitorStateException

Dead lock:

- If 2 Threads are waiting for each other forever(without end) such type of situation(infinite waiting) is called dead lock.
- There are no resolution techniques for dead lock but several prevention(avoidance) techniques are possible.
- Synchronized keyword is the cause for deadlock hence whenever we are using synchronized keyword we have to take special care.

Example:

```
class A
{
    public synchronized void foo(B b)
    {
        System.out.println("Thread1 starts execution of foo() method");
        try
        {
            Thread.sleep(2000);
        }
        catch (InterruptedException e)
        {}
        System.out.println("Thread1 trying to call b.last()");
        b.last();
```

```
        }
        public synchronized void last()
        {
                System.out.println("inside A, this is last()method");
        }
}
class B
{
        public synchronized void bar(A a)
        {
        System.out.println("Thread2 starts execution of bar() method");
        try
        {
                Thread.sleep(2000);
        }
        catch (InterruptedException e)
        {}
        System.out.println("Thread2 trying to call a.last()");
        a.last();
        }
        public synchronized void last()
        {
        System.out.println("inside B, this is last() method");
        }
}
class DeadLock implements Runnable
{
        A a=new A();
        B b=new B();
        DeadLock()
        {
                Thread t=new Thread(this);
                t.start();
                a.foo(b);//main thread
        }
        public void run()
        {
                b.bar(a);//child thread
        }
        public static void main(String[] args)
        {
                new DeadLock();//main thread
        }
}
```

Output:

```
Thread1 starts execution of foo() method
Thread2 starts execution of bar() method
Thread2 trying to call a.last()
Thread1 trying to call b.last()
//here cursor always waiting.
```

Daemon Threads:

- The Threads which are executing in the background are called daemon Threads. The main objective of daemon Threads is to provide support for non daemon Threads.

  Example:

  Garbage collector
- We can check whether the Thread is daemon or not by using isDaemon() method.

  public final boolean isDaemon();
- We can change daemon nature of a Thread by using setDaemon () method.

  public final void setDaemon(boolean b);
- But we can change daemon nature before starting Thread only. That is after starting the Thread if we are trying to change the daemon nature we will get R.E saying IllegalThreadStateException.
- Main Thread is always non daemon and we can't change its daemon nature because it's already started at the beginning only.
- Main Thread is always non daemon and for the remaining Threads daemon nature will be inheriting from parent to child that is if the parent is daemon child is also daemon and if the parent is non daemon then child is also non daemon.
- Whenever the last non daemon Thread terminates automatically all daemon Threads will be terminated.

Example:

```
class MyThread extends Thread
{
	public void run()
	{
		for(int i=0;i<10;i++)
		{
			System.out.println("lazy thread");
			try
			{
				Thread.sleep(2000);
			}
			catch (InterruptedException e)
			{}
		}
	}
}
class DaemonThreadDemo
{
	public static void main(String[] args)
	{
	MyThread t=new MyThread();
	t.setDaemon(true);          ------------> 1
	t.start();
	System.out.println("end of main Thread");
	}
}
```

Output:

End of main Thread
Lazy thread

- If we are commenting line 1 then both main and child Threads are non daemon and hence both will be executed until they completion.
- If we are not commenting line 1 then main Thread is non daemon and child Thread is daemon and hence whenever main Thread terminates automatically child Thread will be terminated.

Deadlock vs Starvation:

- A long waiting of a Thread which never ends is called deadlock.
- A long waiting of a Thread which ends at certain point is called starvation.
- A low priority Thread has to wait until completing all high priority Threads.
- This long waiting of Thread which ends at certain point is called starvation.

How to kill a Thread in the middle of the line?

- We can call stop() method to stop a Thread in the middle then it will be entered into dead state immediately.

public final void stop();

- stop() method has been deprecated and hence not recommended to use.

suspend and resume methods:

- A Thread can suspend another Thread by using suspend() method then that Thread will be paused temporarily.
- A Thread can resume a suspended Thread by using resume() method then suspended Thread will continue its execution.

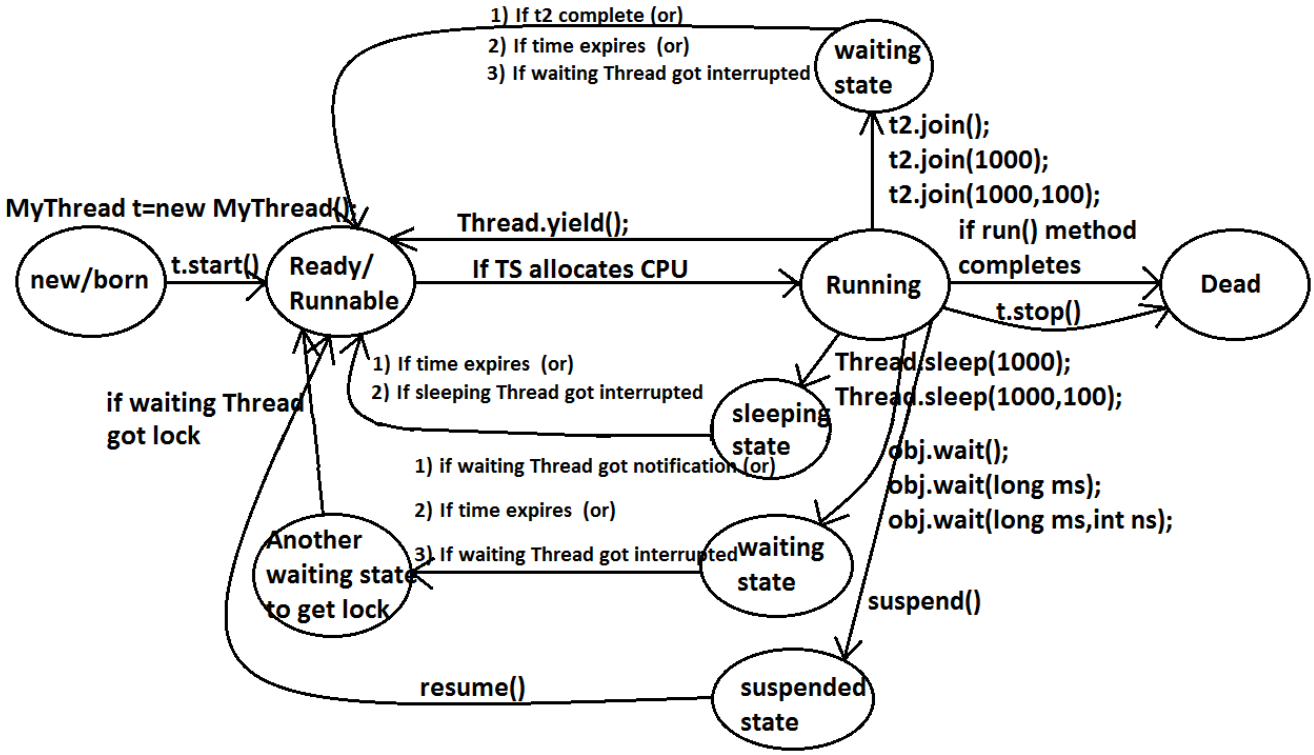
1) public final void suspend();
2) public final void resume();

- Both methods are deprecated and not recommended to use.

RACE condition:

- Executing multiple Threads simultaneously and causing data inconsistency problems is nothing but Race condition we can resolve race condition by using synchronized keyword.

Life cycle of a Thread:



What is the difference between extends Thread and implements Runnable?

- Extends Thread is useful to override the public void run() method of Thread class.
- Implements Runnable is useful to implement public void run() method of Runnable interface.

Extends Thread, implements Runnable which one is advantage?

- If we extend Thread class, there is no scope to extend another class.

Example:

Class MyClass extends Frame,Thread//invalid

- If we write implements Runnable still there is a scope to extend one more class.

Example:

1) class MyClass extends Thread implements Runnable
2) class MyClass extends Frame implements Runnable

How can you stop a Thread which is running?

Step 1:

- Declare a boolean type variable and store false in that variable.

boolean stop=false;

Step 2:

- If the variable becomes true return from the run() method.

If(stop) return;

Step 3:

- Whenever to stop the Thread store true into the variable.

System.in.read();//press enter
Obj.stop=true;

Questions:

1) **What is a Thread?**
2) **Which Thread by default runs in every java program?**
   - Ans: By default main Thread runs in every java program.
3) **What is the default priority of the Thread?**
4) **How can you change the priority number of the Thread?**
5) **Which method is executed by any Thread?**
   - Ans: A Thread executes only public void run() method.
6) **How can you stop a Thread which is running?**
7) **Explain the two types of multitasking?**
8) **What is the difference between a process and a Thread?**
9) **What is Thread scheduler?**
10) **Explain the synchronization of Threads?**
11) **What is the difference between synchronized block and synchronized keyword?**
12) **What is Thread deadlock? How can you resolve deadlock situation?**
13) **Which methods are used in Thread communication?**

14) What is the difference between notify() and notifyAll() methods?
15) What is the difference between sleep() and wait() methods?
16) Explain the life cycle of a Thread?
17) What is daemon Thread?

Java.lang Package

1) Object
2) String
3) StringBuffer
4) StringBuilder
5) Wrapper Classes
6) Autoboxing and Autounboxing

- For writing any java program the most commonly required classes and interfaces are encapsulated in the separate package which is nothing but java.lang package.
- It is not required to import java.lang package in our program because it is available by default to every java program.
- The following are some of important classes present in java.lang package.
  1. Object
  2. String
  3. StringBuffer
  4. StringBuilder
  5. All wrapper classes
  6. Execption API
  7. Thread API….etc

  What is your favorite package?
  Why java.lang is your favorite package?
- It is not required to import lang package explicitly but the remaining packages we have to import.

Java.lang.Object class: For any java object whether it is predefine or customized the most commonly required methods are encapsulated into a separate class which is nothing but object class.

- As object class acts as a root (or) parent (or) super for all java classes, by default its methods are available to every java class.
- The following is the list of all methods present in java.lang Object class.

1) public String toString();
2) public native int hashCode();
3) public boolean equals(Object o);
4) protected native Object clone()throws CloneNotSupportedException;
5) public final Class<?> getClass();
6) protected void finalize()throws Throwable;
7) public final void wait()throws InterruptedException;
8) public final native void wait()throws InterruptedException;
9) public final void wait(long ms,int ns)throws InterruptedException;
10) public final native void notify();
11) public final native void notifyAll();

toString() method: We can use this method to get string representation of an object.

- Whenever we are try to print any object reference internally toString() method will be executed.
- If our class doesn't contain toString() method then Object class toString() method will be executed.

Example: System.out.println(s1); ➔super(s1.toString());

Example 1:

```
class Student
{
String name;
int rollno;
Student(String name,int rollno)
{
this.name=name;
this.rollno=rollno;
}
public static void main(String args[]){
Student s1=new Student("vijayabhaskar",101);
Student s2=new Student("bhaskar",102);
```

```
System.out.println(s1);
System.out.println(s1.toString());
System.out.println(s2);
}
}
```

Output:

```
Student@3e25a5
Student@3e25a5
Student@19821f
```

- In the above program Object class toString() method got executed which is implemented as follows.

```
public String toString() {
      return getClass().getName() + "@" + Integer.toHexString(hashCode());
  }
```

- To provide our own String representation we have to override toString() method in our class.
- For example whenever we are try to print student reference to print his a name and roll no we have to override toString() method as follows.

```
public String toString(){
return name+"........"+rollno;
}
```

- In String class, StringBuffer, StringBuilder, wrapper classes and in all collection classes toString() method is overridden for meaningful string representation. Hence in our classes also **highly recommended to override toString() method**.

Example 2:

```
class Test
{
public String toString()
{
return "Test";
}
public static void main(String[] args){
Integer i=new Integer(10);
String s=new String("bhaskar");
Test t=new Test();
System.out.println(i);
System.out.println(s);
System.out.println(t);
}}
```

Output:

```
10
Bhaskar
Test
```

hashCode() method:

- For every object jvm will generate a unique number which is nothing but hashCode.
- hashCode of an object will be used by jvm while saving objects into HashSet, HashMap, and Hashtable etc.
- If the objects are stored according to hashCode searching will become very efficient (The most powerful search algorithm is hashing which will work based on hashCode).
- If we didn't override hashCode() method then Object class hashCode() method will be executed which generates hashCode based on address of the object but it doesn't mean hashCode represents address of the object.
- Based on our programming requirement we can override hashCode() method in our class.
- Overriding hashCode() method is said to be proper if and only if for every object we have to generate a unique number.

Example 3:

```
class Student
{
public int hashCode()
{
return 100;
}}
```

- It is improper way of overriding hashCode() method because for every object we are generating same hashcode.

```
class Student
{
int rollno;
public int hashCode()
{
return rollno;
}}
```

- It is proper way of overriding hashcode() method because for every object we are generating a different hashcode.

toString() method vs hashCode() method:

```
class Test
{
int i;
Test(int i)
{
```

```
class Test{
int i;
Test(int i){
this.i=i;
}
```

```
this.i=i;
}
public static void main(String[] args){
Test t1=new Test(10);
Test t2=new Test(100);
System.out.println(t1);
System.out.println(t2);
}
}
```

Object==➔toString() called.
Object==➔hashCode() called.

- In this case Object class toString() method got executed which is internally calls Object class hashCode() method.

```
public int hashCode(){
return i;
}
public static void main(String[] args){
Test t1=new Test(10);
Test t2=new Test(100);
System.out.println(t1);
System.out.println(t2);
}}
```

Object==➔toString() called.
Test==➔hashCode() called.

- In this case Object class toString() method got executed which is internally calls Test class hashCode() method.

Example 4:

```
class Test
{
int i;
Test(int i)
{
this.i=i;
}
public int hashCode(){
return i;
}
public String toString()
{
return i+"";
}
public static void main(String[] args){
Test t1=new Test(10);
Test t2=new Test(100);
System.out.println(t1);
System.out.println(t2);
}
}
```

Output:

```
10
100
```

- In this case Test class toString() method got executed.

Note: if we are giving optrurniuty to Object class toString() method it internally calls hashCode() method. But if we are overriding toString() method it may not call hashCode() method.

- We can use toString() method while printing object references and we can use hashCode() method while saving objects into HashSet or Hashtable or HashMap.

equals() method:

- We can use this method to check equivalence of two objects.
- If our class doesn't contain .equals() method then object class .equals() method will be executed which is always meant for reference compression[address compression].

Example 5:

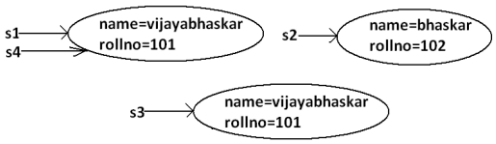
```
class Student
{
String name;
int rollno;
Student(String name,int rollno)
{
this.name=name;
this.rollno=rollno;
}
public static void main(String[] args){
Student s1=new Student("vijayabhaskar",101);
Student s2=new Student("bhaskar",102);
Student s3=new Student("vijayabhaskar",101);
Student s4=s1;
System.out.println(s1.equals(s2));
System.out.println(s1.equals(s3));
System.out.println(s1.equals(s4));
```

}}

Output:

False
False
True

Diagram:



- In the above program Object class .equals() method got executed which is always meant for reference compression that is if two references pointing to the same object then only .equals(() method returns true.
- In object class .equals() method is implemented as follows which is meant for reference compression.

```
public boolean equals(Object obj) {
    return (this == obj);
  }
```

- Based on our programming requirement we can override .equals() method for content compression purpose.
- While overriding .equals() method we have to consider the following things.

1) Meaning of content compression like whether the names are equal (or) roll numbers (or) both are equal.
2) If we are passing heterogeneous object our .equals() method should return false that is we have to handle **ClassCastException** to return false.
3) If we are providing null argument our .equals() method should return false that is we have to handle **NullPointerException** to return false.

- The following is the proper way of overriding .equals() method for content compression in Student class.

Example 6:

```
class Student
{
String name;
int rollno;
Student(String name,int rollno)
{
this.name=name;
this.rollno=rollno;
}
public boolean equals(Object obj)
{
try{
String name1=this.name;
int rollno1=this.rollno;
Student s2=(Student)obj;
String name2=s2.name;
int rollno2=s2.rollno;
if(name1.equals(name2)&&rollno1==rollno2)
{
return true;
}
else return false;
}
catch(ClassCastException e)
{
return false;
}
catch(NullPointerException e)
{
return false;
}
}
public static void main(String[] args){
Student s1=new Student("vijayabhaskar",101);
Student s2=new Student("bhaskar",102);
Student s3=new Student("vijayabhaskar",101);
Student s4=s1;
System.out.println(s1.equals(s2));
System.out.println(s1.equals(s3));
```

```
System.out.println(s1.equals(s4));
System.out.println(s1.equals("vijayabhaskar"));
System.out.println(s1.equals("null"));
}
}
```

Output:

```
False
True
True
False
False
```

Simplified version of .equals() method:

```
public boolean equals(Object o){
try{
Student s2=(Student)o;
if(name.equals(s2.name)&&rollno==s2.rollno){
return true;}
else return false;
}
catch(ClassCastException e)
{
return false;
}
catch(NullPointerException e)
{
return false;
}}
```

More simplified version of .equals() method:

```
public boolean equals(Object o)
{
if(this==o)
return true;
if(o instanceof Student)
{
Student s2=(Student)o;
if(name.equals(s2.name)&&rollno==s2.rollno)
return true;
else
return false;
}
return false;
}
```

Example 7:

```
class Student
{
String name;
int rollno;
Student(String name,int rollno)
{
this.name=name;
this.rollno=rollno;
}
public boolean equals(Object o)
{
if(this==o)
return true;
if(o instanceof Student)
{
Student s2=(Student)o;
if(name.equals(s2.name)&&rollno==s2.rollno)
return true;
else
return false;
}
return false;
}
```

```
public static void main(String[] args){
Student s=new Student("vijayabhaskar",101);
Integer i=new Integer(10);
System.out.println(s.equals(i));
}
}
```

Output:

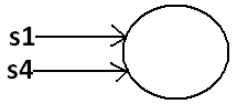
False

- To make .equals() method more efficient we have to place the following code at the top inside .equals() method.

```
if(this==o)
return true;
```

Diagram:

Student s1=new Student("vijayabhaskar",101);
Student s4=s1;



- If 2 references pointing to the same object then .equals() method return true directly without performing any content compression this approach improves performance of the system.

Relationship between .equals() method and ==(double equal operator):

1) If r1==r2 is true then r1.equals(r2) is always true that is if two objects are equal by double equal operator then these objects are always equal by .equals() method also.
2) If r1==r2 is false then we can't conclude anything about r1.equals(r2) it may return true (or) false.
3) If r1.equals(r2) is true then we can't conclude anything about r1==r2 it may returns true (or) false.
4) If r1.equals(r2) is false then r1==r2 is always false.

Differences between == (double equal operator) and .equals() method?

==(double equal operator)	.equals() method
1) It is an operator applicable for both primitives and object references.	1) It is a method applicable only for object references but not for primitives.
2) In the case of primitives == (double equal operator) meant for content compression, but in the case of object references == operator meant for reference compression.	2) By default .equals() method present in object class is also meant for reference compression.
3) We can't override== operator for content compression in object references.	3) We can override .equals() method for content compression.
4) If there is no relationship between argument types then we will get compile time error saying incompatible types.	4) If there is no relationship between argument types then .equals() method simply returns false and we won't get any compile time error and runtime error.
5) For any object reference r, r==null is always false.	5) For any object reference r, r.equals(null) is also returns false.

Note: in general we can use == (double equal operator) for reference compression whereas .equals() method for content compression.

Contract between .equals() method and hashCode() method:

1) If 2 objects are equal by .equals() method compulsory their hashcodes must be equal (or) same. That is
   If r1.equals(r2) is true then r1.hascode()==r2.hashcode() must be true.
2) If 2 objects are not equal by .equals() method then there are no restrictions on hashCode() methods. They may be same (or) may be different. That is
   If r1.equals(r2) is false then r1.hashCode()==r2.hashCode() may be same (or) may be different.
3) If hashcodes of 2 objects are equal we can't conclude anything about .equals() method it may returns true (or) false. That is
   If r1.hashCode()==r2.hashCode() is true then r1.equals(r2) method may returns true (or) false.
4) If hashcodes of 2 objects are not equal then these objects are always not equal by .equals() method also. That is
   If r1.hashCode()==r2.hashCode() is false then r1.equals(r2) is always false.

- To maintain the above contract between .equals() and hashCode() methods whenever we are overriding .equals() method compulsory we should override hashCode() method. Violation leads to no compile time error and runtime error but it is not good programming practice.
- Consider the following person class.

Program:

```
class Person
{
String name;
int age;
Person(String name,int age)
{
this.name=name;
this.age=age;
```

```
}
public boolean equals(Object o)
{
if(this==o)
return true;
if(o instanceof Person)
{
Person p2=(Person)o;
if(name.equals(p2.name)&&age==p2.age)
return true;
else
return false;
}
return false;
}
public static void main(String[] args){
Person p1=new Person("vijayabhaskar",101);
Person p2=new Person("vijayabhaskar",101);
Integer i=new Integer(102);
System.out.println(p1.equals(p2));
System.out.println(p1.equals(i));
}}
```

Output:

True

False

Which of the following is appropriate way of overriding hashCode() method?

x 1)
```
public int hashCode()
{
    return 100;
}
```

x 2)
```
public int hashCode()
{
    return age+height;
}
```

✓ 3)
```
public int hashCode()
{
    return name.hashCode()+age;
}
```

4)Any of the above.

- Based on whatever the parameters we override ".equals() method" we should use same parameters while overriding hashCode() method also.

  Note: in all wrapper classes, in string class, in all collection classes .equals() method is overridden for content compression in our classes also it is highly recommended to override .equals() method.

Which of the following is valid?

1) If hash Codes of 2 objects are not equal then .equals() method always return false.(valid)

Example:

```
class Test
{
    int i;
    Test(int i)
    {
        this.i=i;
    }
    public int hashCode()
    {
        return i;
    }
    public String toString()
    {
        return i+"";
    }
    public static void main(String[] args)
    {
        Test t1=new Test(10);
        Test t2=new Test(20);
        System.out.println(t1.hashCode());//10
        System.out.println(t2.hashCode());//20
        System.out.println(t1.hashCode()==t2.hashCode());//false
        System.out.println(t1.equals(t2));//false
    }
}
```

2) If 2 objects are equal by == operator then their hash codes must be same.(valid)

Example:

```
class Test
{
	int i;
	Test(int i)
	{
		this.i=i;
	}
	public int hashCode()
	{
		return i;
	}
	public String toString()
	{
		return i+"";
	}
	public static void main(String[] args)
	{
		Test t1=new Test(10);
		Test t2=t1;
		System.out.println(t1.hashCode());//10
		System.out.println(t2.hashCode());//10
		System.out.println(t1==t2);//true

	}
}
```

3) If == operator returns false then their hash codes(may be same (or) may be different) must be different.(invalid)

Example:

```
class Test
{
	int i;
	Test(int i)
	{
		this.i=i;
	}
	public int hashCode()
	{
		return i;
	}
	public String toString()
	{
		return i+"";
	}
	public static void main(String[] args)
	{
		Test t1=new Test(10);
		Test t2=new Test(10);
		System.out.println(t1.hashCode());//10
		System.out.println(t2.hashCode());//10
		System.out.println(t1==t2);//false
	}
}
```

4) If hashcodes of 2 objects are equal then these objects are always equal by == operator also.(invalid)

Clone () method:

- The process of creating exactly duplicate object is called cloning.
- The main objective of cloning is to maintain backup.
- That is if something goes wrong we can recover the situation by using backup copy.
- We can perform cloning by using clone() method of Object class.

protected native object clone() throws CloneNotSupportedException;

Example:

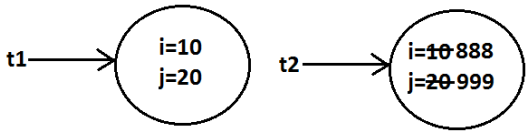
```
class Test implements Cloneable
{
	int i=10;
	int j=20;
	public static void main(String[] args)throws CloneNotSupportedException
		{
		Test t1=new Test();
```

```
                Test t2=(Test)t1.clone();
                t2.i=888;
                t2.j=999;
                System.out.println(t1.i+"----------"+t1.j);
                System.out.println(t2.i+"----------"+t2.j);
        }
}
```

Output:

10----------20
888----------999

Diagram:



- We can perform cloning only for Cloneable objects.
- An object is said to be Cloneable if and only if the corresponding class implements Cloneable interface.
- Cloneable interface present in java.lang package and does not contain any methods. It is a marker interface where the required ability will be provided automatically by the JVM.

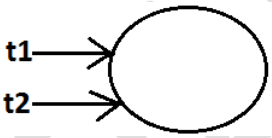
Shallow cloning vs deep cloning:

- The process of creating just duplicate reference variable but not duplicate object is called shallow cloning.

Example:

```
Test t1=new Test();
Test t2=t1;
```

Diagram:

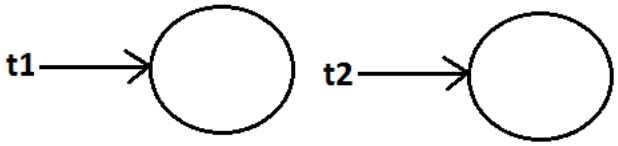


- The process of creating exactly independent duplicate object is called deep cloning.

Example:

```
Test t1=new Test();
Test t2=(Test)t1.clone();
System.out.println(t1==t2);//false
System.out.println(t1.hashCode()==t2.hashCode());//false
```

Diagram:



- Cloning by default deep cloning.

getClass() method:

- This method returns runtime class definition of an object.

Example:

```
class  Test implements Cloneable
{
        public static void main(String[] args)throws CloneNotSupportedException
        {
                Object o=new String("bhaskar");
                System.out.println("Runtime object type of o is :"+o.getClass().getName());
        }
}
```

Output:

Runtime object type of o is: java.lang. String

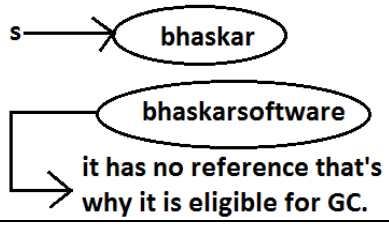
String

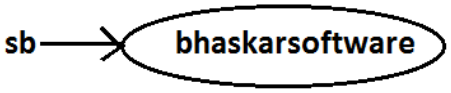
Case 1:

String s=new String("bhaskar");
s.concat("software");
System.out.println(s);//bhaskar
• Once we create a String object we can't perform any changes in the existing object. If we are try to perform any changes with those changes a new object will be created. This behavior is called immutability of the
	StringBuffer sb=new StringBuffer("bhaskar");
	sb.append("software");
	System.out.println(sb);
	• Once we created a StringBuffer object we can perform any changes in the existing object. This behavior is called mutability of the StringBuffer object.
	Diagram:

String object.

Diagram:





Case 2:

String s1=new String("bhaskar");
String s2=new String("bhaskar");
System.out.println(s1==s2);//false
System.out.println(s1.equals(s2));//true

- In String class .equals() method is overridden for content compression hence if the content is same .equals() method returns true even though objects are different.

StringBuffer sb1=new StringBuffer("bhaskar");
StringBuffer sb2=new StringBuffer("bhaskar");
System.out.println(sb1==sb2);//false
System.out.println(sb1.equals(sb2));//false

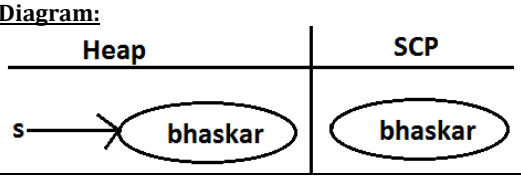
- In StringBuffer class .equals() method is not overridden for content compression hence Object class .equals() method got executed which is always meant for reference compression. Hence if objects are different .equals() method returns false even though content is same.

Case 3:

String s=new String("bhaskar");

- In this case two objects will be created one is on the heap the other one is SCP(String constant pool) and s is always pointing to heap object.

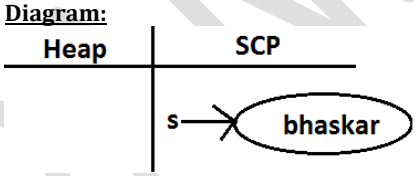
Diagram:



String s="bhaskar";

- In this case only one object will be created in SCP and s is always referring that object.

Diagram:



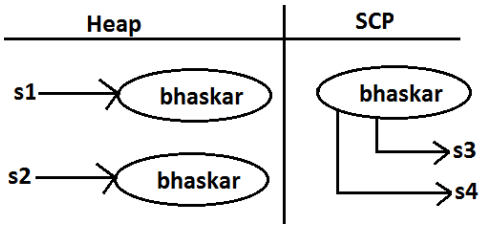
Note:

1) Object creation in SCP is always optional 1st JVM will check is any object already created with required content or not. If it is already available then it will reuse existing object instead of creating new object. If it is not already there then only a new object will be created. Hence there is no chance of existing 2 objects with same content on SCP that is duplicate objects are not allowed in SCP.
2) Garbage collector can't access SCP area hence even though object doesn't have any reference still that object is not eligible for GC if it is present in SCP.
3) All SCP objects will be destroyed at the time of JVM shutdown automatically.

Example 1:

String s1=new String("bhaskar");
String s2=new String("bhaskar");
String s3="bhaskar";
String s4="bhaskar";

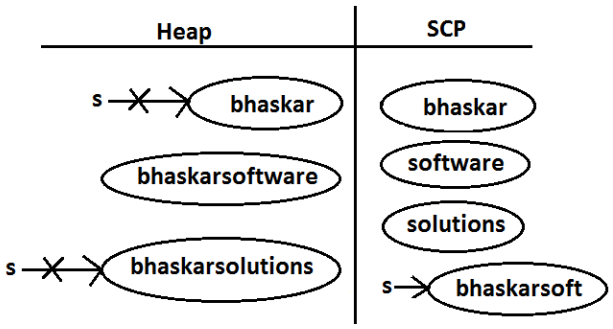
Diagram:



Example 2:

String s=new String("bhaskar");
s.concat("software");
s=s.concat("solutions");
s="bhaskarsoft";

Diagram:

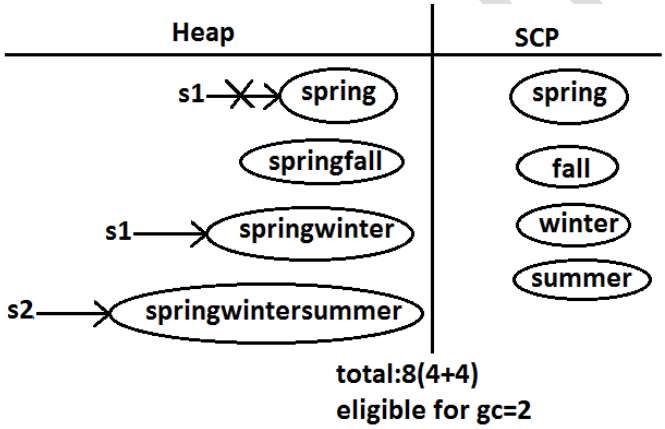


- For every String Constant one object will be created in SCP. Because of runtime operation if an object is required to create compulsory that object should be placed on the heap but not SCP.

Example 3:

```
String s1=new String("spring");
s1.concat("fall");
s1=s1+"winter";
String s2=s1.concat("summer");
System.out.println(s1);
System.out.println(s2);
```

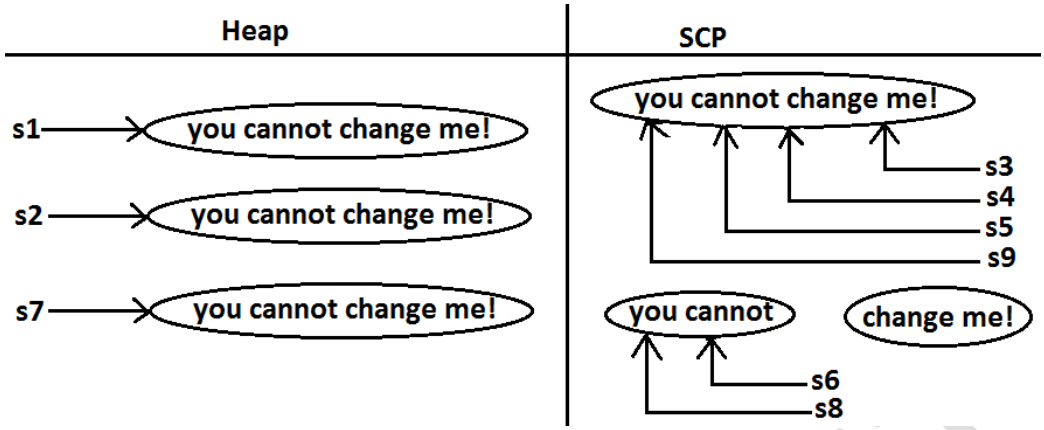
Diagram:



Example:

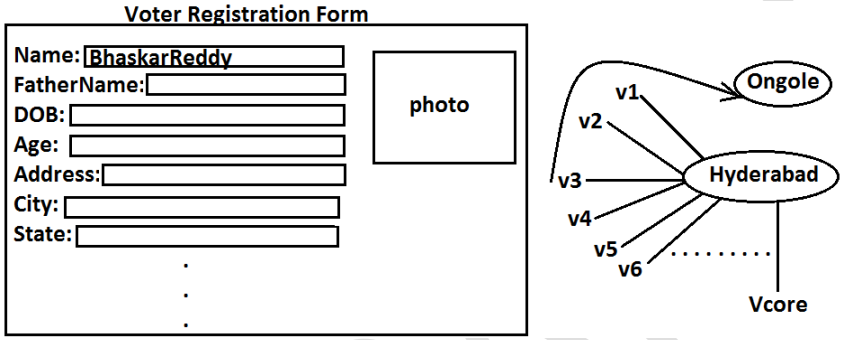
```
class StringDemo
{
	public static void main(String[] args)
	{
		String s1=new String("you cannot change me!");
		String s2=new String("you cannot change me!");
		System.out.println(s1==s2);//false
		String s3="you cannot change me!";
		System.out.println(s1==s3);//false
		String s4="you cannot change me!";
		System.out.println(s3==s4);//true
		String s5="you cannot "+"change me!";
		System.out.println(s3==s5);//true
		String s6="you cannot ";
		String s7=s6+"change me!";
		System.out.println(s3==s7);//false
		final String s8="you cannot ";
		String s9=s8+"change me!";
		System.out.println(s3==s9);//true
		System.out.println(s6==s8);//true
		}
}
```

Diagram:



Importance of String constant pool (SCP):

Diagram:



- In our program if any String object is required to use repeatedly then it is not recommended to create multiple object with same content it reduces performance of the system and effects memory utilization.
- We can create only one copy and we can reuse the same object for every requirement. This approach improves performance and memory utilization we can achieve this by"scp".
- In SCP several references pointing to same object the main disadvantage in this approach is by using one reference if we are performing any change the remaining references will be impacted. To parent this sun people declared String objects as immutable.
- According to this once we creates a String object we can't perform any changes in the existing object if we are trying to perform any changes with those changes a new String object will be created hence immutability is the main disadvantage of scp.

1) What is the main difference between String and StringBuilder?

2) What is the meaning of immutability and mutability?

3) Explain immutability and mutability with an example?

4) What is SCP?

- A specially designed memory area for the String literals.

5) What is the advantage of SCP?

- Instead of creating a separate object for every requirement we can create only one object and we can reuse same object for every requirement. This approach improves performance and memory utilization.

6) What is the disadvantage of SCP?

- In SCP as several references pointing to the same object by using one reference if we are performing any changes the remaining references will be inflected. To prevent this compulsory String objects should be immutable. That is immutability is the disadvantage of SCP.

7) Why SCP like concept available only for the String but not for the StringBuffer?

- As String object is the most commonly used object sun people provided a specially designed memory area like SCP to improve memory utilization and performance.
- But StringBuffer object is not commonly used object hence specially designed memory area is not at all required.

8) Why String objects are immutable where as StringBuffer objects are mutable?

- In the case of String as several references pointing to the same object, by using one reference if we are allowed perform the change the remaining references will be impacted. To prevent this once we created a String object we can't perform any change in the existing object that is immutability is only due to SCP.
- But in the case of StringBuffer for every requirement we are creating a separate object by using one reverence if we are performing any change in the object the remaining references won't be impacted hence immutability concept is not require for the StringBuffer.

9) Similar to String objects any other objects are immutable in java?

- In addition to String all wrapper objects are immutable in java.

10) Is it possible to create our own mutable class?

Yes.

11) Explain the process of creating our own immutable class with an example?

12) What is the difference between final and immutability?

13) What is interning of String objects?

Interning of String objects:

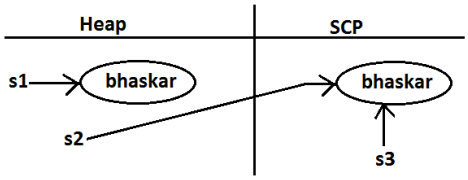
- By using heap object reference, if we want to corresponding SCP object reference we should go for intern() method.

Example 1:

```
class StringInternDemo
{
	public static void main(String[] args)
	{
		String s1=new String("bhaskar");
		String s2=s1.intern();
		String s3="bhaskar";
		System.out.println(s2==s3);//true

	}
}
```

Diagram:



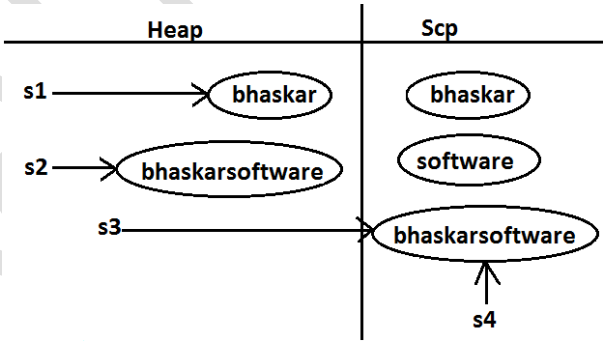
- If the corresponding object is not there in SCP then intern() method itself will create that object and returns it.

Example 2:

```
class StringInternDemo
{
	public static void main(String[] args)
	{
		String s1=new String("bhaskar");
		String s2=s1.concat("software");
		String s3=s2.intern();
		String s4="bhaskarsoftware";
		System.out.println(s3==s4);//true

	}
}
```

Diagram 2:



String class constructors:

1) String s=new String();

- Creates an empty String Object.

2) String s=new String(String literals);

- To create an equivalent String object for the given String literal on the heap.

3) String s=new String(StringBuffer sb);

- Creates an equivalent String object for the given StringBuffer.

4) String s=new String(char[] ch);

Example:

```
class StringInternDemo
{
public static void main(String[] args)
{
char[] ch={'a','b','c'};
String s=new String(ch);
```

```
System.out.println(ch);//abc
}
}
```

5) String s=new String(byte[] b);

- For the given byte[] we can create a String.

Example:

```
class StringInternDemo
{
public static void main(String[] args)
{
byte[] b={100,101,102};
String s=new String(b);
System.out.println(s);//def
}
}
```

Important methods of String class:

1) public char charAt(int index);

- Returns the character locating at specified index.

Example:

```
class StringInternDemo
{
        public static void main(String[] args)
        {
        String s="bhaskar";
        System.out.println(s.charAt(3));//s
        System.out.println(s.charAt(100));//StringIndexOutOfBoundsException
        }
}
```

2) public String concat(String str);

Example:

```
class StringInternDemo
{
        public static void main(String[] args)
        {
        String s="bhaskar";
        s=s.concat("software");
        //s=s+"software";
        //s+="software";
        System.out.println(s);//bhaskarsoftware
        }
}
```

- The overloaded "+" and "+=" operators also meant for concatenation purpose only.

3) public boolean equals(Object o);

- For content compression where case is important.
- It is the overriding version of Object class .equals() method.

4) public boolean equalsIgnoreCase(String s);

- For content compression where case is not important.

Example:

```
class StringInternDemo
{
        public static void main(String[] args)
        {
        String s="bhaskar";
        System.out.println(s.equals("bhaskar"));//true
        System.out.println(s.equalsIgnoreCase("BHASKAR"));//true
        }
}
```

Note: We can validate username by using equalsIgnoreCase() method where case is not important and we can validate password by using .equals() method where case is important.

5) public String substring(int begin);

- Return the substring from begin index to end of the string.

Example:

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

```
        String s="vijayabhaskar";
        System.out.println(s.substring(6));//bhaskar
        }
}
```

6) public String substring(int begin, int end);

- Returns the substring from begin index to end-1 index.

Example:

```
class StringInternDemo
{
        public static void main(String[] args)
        {
        String s="vijayabhaskar";
        System.out.println(s.substring(6));//bhaskar
        System.out.println(s.substring(3,7));//ayab
        }
}
```

7) public int length();

- Returns the no of characters present in the string.

Example:

```
class StringInternDemo
{
        public static void main(String[] args)
        {
        String s="vijayabhaskar";
        System.out.println(s.length());//13
        //System.out.println(s.length);//compile time error
        //StringInternDemo.java:7: cannot find symbol
        //symbol  : variable length
        //location: class java.lang.String
        }
}
```

Note: length is the variable applicable for arrays where as length() method is applicable for String object.

8) public String replace(char old,char new);

- To replace every old character with a new character.

Example:

```
class StringInternDemo
{
        public static void main(String[] args)
        {
        String s="abababab";
        System.out.println(s.replace('a','b'));//bbbbbb
        }
}
```

9) public String toLowerCase();

- Converts the all characters of the string to lowercase.

Example:

```
class StringInternDemo
{
        public static void main(String[] args)
        {
        String s="BHASKAR";
        System.out.println(s.toLowerCase());//bhaskar
        }
}
```

10) public String toUpperCase();

- Converts the all characters of the string to uppercase.

Example:

```
class StringInternDemo
{
        public static void main(String[] args)
        {
        String s="bhaskar";
        System.out.println(s.toUpperCase());//BHASKAR
        }
}
```

11) public String trim()

- We can use this method to remove blank spaces present at beginning and end of the string but not blank spaces present at middle of the String.

Example:

```
class StringInternDemo
{
        public static void main(String[] args)
        {
        String s="  bha skar  ";
        System.out.println(s.trim());//bha skar
        }
}
```

12) public int indexOf(char ch);

- It returns index of 1st occurrence of the specified character if the specified character is not available then return -1.

Example:

```
class StringInternDemo
{
        public static void main(String[] args)
        {
        String s="vijayabhaskarreddy";
        System.out.println(s.indexOf('a'));//3
        System.out.println(s.indexOf('z'));-1
        }
}
```

13) public int lastIndexOf(Char ch);

- It returns index of last occurrence of the specified character if the specified character is not available then return -1.

Example:

```
class StringInternDemo
{
        public static void main(String[] args)
        {
        String s="vijayabhaskarreddy";
        System.out.println(s.lastIndexOf('a'));//11
        System.out.println(s.indexOf('z'));//-1
        }
}
```

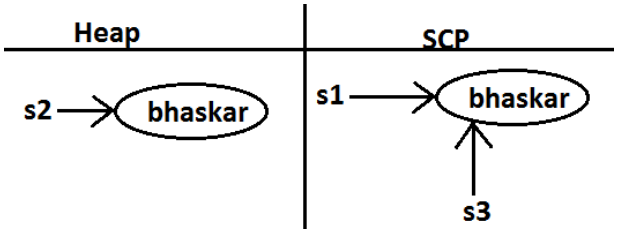
Note:

- Because runtime operation if there is a change in content with those changes a new object will be created only on the heap but not in SCP.
- If there is no change in content no new object will be created the same object will be reused.

Example 1:

```
class StringInternDemo
{
        public static void main(String[] args)
        {
        String s1="bhaskar";
        String s2=s1.toUpperCase();
        String s3=s1.toLowerCase();
        System.out.println(s1==s2);//false
        System.out.println(s1==s3);//true
        }
}
```

Diagram:

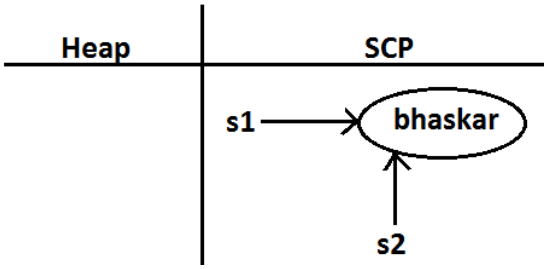


Example 2:

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

```
        {
        String s1="bhaskar";
        String s2=s1.toString();
        System.out.println(s1==s2);//true
        }
}
```

Diagram:



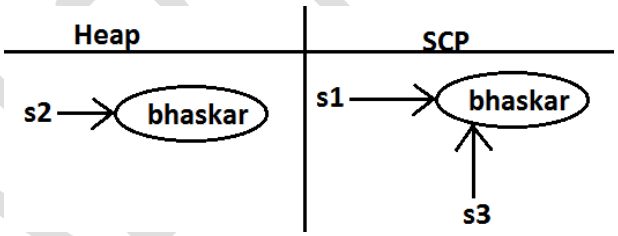
Creation of our own immutable class:

- We can create our own immutable classes.
- Once we created an object we can't perform any changes in the existing object. If we are trying to perform any changes with those changes a new object will be created. If there is no change in the content then existing object will be reused. This behavior is called immutability.

Example:

```
class StringInternDemo
{
        public static void main(String[] args)
        {
        String s1="bhaskar";
        String s2=s1.toUpperCase();
        String s3=s1.toLowerCase();
        System.out.println(s1==s2);//false
        System.out.println(s1==s3);//true
        }
}
```

Diagram:

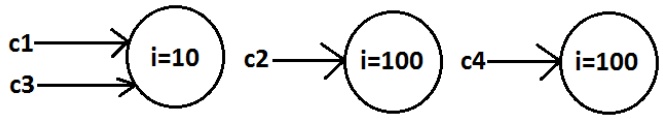


Immutable program:

```
final class CreateImmutable
{
        private int i;
        CreateImmutable(int i)
        {
                this.i=i;
        }
        public CreateImmutable modify(int i)
        {
                if(this.i==i)
                        return this;
                else
                        return (new CreateImmutable(i));
        }
        public static void main(String[] args)
        {
        CreateImmutable c1=new CreateImmutable(10);
        CreateImmutable c2=c1.modify(100);
        CreateImmutable c3=c1.modify(10);
        System.out.println(c1==c2);//false
        System.out.println(c1==c3);//true
        CreateImmutable c4=c1.modify(100);
```

```
        System.out.println(c2==c4);//false
        }
}
```

Diagram:



Final vs immutability:

- If we declare a variable as final then we can't perform reassignment for that variable. It doesn't mean in the corresponding object we can't perform any changes. That is through final keyword we won't get any immutability that is final and immutability concepts are different.

Example:

```
class Test
{
        public static void main(String[] args)
        {
                final StringBuffer sb=new StringBuffer("bhaskar");
                sb.append("software");
                System.out.println(sb);//bhaskarsoftware
                sb=new StringBuffer("solutions");//C.E: cannot assign a value to final variable sb
        }
}
```

- In the above example even though "sb" is final we can perform any type of change in the corresponding object. That is through final keyword we are not getting any immutability nature.

StringBuffer

- If the content will change frequently then never recommended to go for String object because for every change a new object will be created internally.
- To handle this type of requirement we should go for StringBuffer concept.
- The main advantage of StringBuffer over String is, all required changes will be performed in the existing object only instead of creating new object.

Constructors:

1) StringBuffer sb=new StringBuffer();

- Creates an empty StringBuffer object with default initialcapacity "16".
- Once StringBuffer object reaches its maximum capacity a new StringBuffer object will be created with **Newcapacity=(currentcapacity+1)*2.**

Example:

```
class StringBufferDemo
{
    public static void main(String[] args)
    {
    StringBuffer sb=new StringBuffer();
    System.out.println(sb.capacity());//16
    sb.append("abcdefghijklmnop");
    System.out.println(sb.capacity());//16
    sb.append("q");
    System.out.println(sb.capacity());//34
    }
}
```

2) StringBuffer sb=new StringBuffer(int initialcapacity);

- Creates an empty StringBuffer object with the specified initial capacity.

Example:

```
class StringBufferDemo
{
    public static void main(String[] args)
    {
    StringBuffer sb=new StringBuffer(19);
    System.out.println(sb.capacity());//19
    }
}
```

3) StringBuffer sb=new StringBuffer(String s);

- Creates an equivalent StringBuffer object for the given String with capacity=s.length()+16;

Example:

```
class StringBufferDemo
```

```
{
    public static void main(String[] args)
    {
    StringBuffer sb=new StringBuffer("bhaskar");
    System.out.println(sb.capacity());//23
    }
}
```

Important methods of StringBuffer:

1) public int length();

- Return the no of characters present in the StringBuffer.

2) public int capacity();

- Returns the total no of characters but a StringBuffer can accommodate(hold).

3) public char charAt(int index);

- It returns the character located at specified index.

Example:

```
class StringBufferDemo
{
    public static void main(String[] args)
    {
    StringBuffer sb=new StringBuffer("vijayabhaskarreddy");
    System.out.println(sb.length());//18
    System.out.println(sb.capacity());//34
    System.out.println(sb.charAt(14));//e
    }
}
```

4) public void setCharAt(int index,char ch);

- To replace the character locating at specified index with the provided character.

Example:

```
class StringBufferDemo
{
    public static void main(String[] args)
    {
    StringBuffer sb=new StringBuffer("vijayabhaskarreddy");
    sb.setCharAt(6,'A');
    System.out.println(sb);
    }
}
```

5) public StringBuffer append(String s);

```
public StringBuffer append(int i);
public StringBuffer append(long l);
public StringBuffer append(boolean b);
public StringBuffer append(double d);
public StringBuffer append(float f);
```

All these are overloaded methods.

Example:

```
class StringBufferDemo
{
    public static void main(String[] args)
    {
    StringBuffer sb=new StringBuffer();
    sb.append("PI value is :");
    sb.append(3.14);
    sb.append(" this is exactly ");
    sb.append(true);
    System.out.println(sb);//PI value is :3.14 this is exactly true
    }
}
```

6) public StringBuffer insert(int index,String s);

```
public StringBuffer insert(int index,int i);
public StringBuffer insert(int index,long l);
public StringBuffer insert(int index,double d);
public StringBuffer insert(int index,boolean b);
public StringBuffer insert(int index,float f);
```

All are overloaded methods

- To insert at the specified location.

Example:

```
class StringBufferDemo
{
```

```
        public static void main(String[] args)
        {
        StringBuffer sb=new StringBuffer("vijaya");
        sb.insert(6,"bhaskar");
        sb.insert(13,"9");
        System.out.println(sb);//vijayabhaskar9
        }
}
```

7) public StringBuffer delete(int begin,int end);

- To delete characters from begin index to end n-1 index.

8) public StringBuffer deleteCharAt(int index);

- To delete the character locating at specified index.

Example:

```
class StringBufferDemo
{
        public static void main(String[] args)
        {
        StringBuffer sb=new StringBuffer("vijayabhaskar");
        System.out.println(sb);//vijayabhaskar
        sb.delete(6,13);
        System.out.println(sb);//vijaya
        sb.deleteCharAt(5);
        System.out.println(sb);//vijay
        }
}
```

9) public StringBuffer reverse();

Example:

```
class StringBufferDemo
{
        public static void main(String[] args)
        {
        StringBuffer sb=new StringBuffer("vijayabhaskar");
        System.out.println(sb);//vijayabhaskar
        System.out.println(sb.reverse());//raksahbayajiv
        }
}
```

10) public void setLength(int length);

- Consider only specified no of characters and remove all the remaining characters.

Example:

```
class StringBufferDemo
{
        public static void main(String[] args)
        {
        StringBuffer sb=new StringBuffer("vijayabhaskar");
        sb.setLength(6);
        System.out.println(sb);//vijaya
        }
}
```

11) public void trimToSize();

- To deallocate the extra free memory such that capacity and size are equal.

Example:

```
class StringBufferDemo
{
        public static void main(String[] args)
        {
        StringBuffer sb=new StringBuffer(1000);
        System.out.println(sb.capacity());//1000
        sb.append("bhaskar");
        System.out.println(sb.capacity());//1000
        sb.trimToSize();
        System.out.println(sb.capacity());//7
        }
}
```

12) public void ensureCapacity(int initialcapacity);

- To increase the capacity dynamically based on our requirement.

Example:

```
class StringBufferDemo
{
    public static void main(String[] args)
    {
    StringBuffer sb=new StringBuffer();
    System.out.println(sb.capacity());//16
    sb.ensureCapacity(1000);
    System.out.println(sb.capacity());//1000
    }
}
```

StringBuilder (1.5)

- Every method present in StringBuffer is declared as synchronized hence at a time only one thread is allowed to operate on the StringBuffer object due to this, waiting time of the threads will be increased and effects performance of the system.
- To overcome this problem sun people introduced StringBuilder concept in 1.5v.
- StringBuilder is exactly same as StringBuffer except the following differences.

	StringBuffer		StringBuilder
1)	Every method present in StringBuffer is synchronized.	1)	No method present in StringBuilder is synchronized.
2)	At a time only one thread is allow to operate on the StringBuffer object hence StringBuffer object is Thread safe.	2)	Multiple Threads are allowed to operate simultaneously on the StringBuilder object hence StringBuilder is not Thread safe.
3)	It increases waiting time of the Thread and hence relatively performance is low.	3)	Threads are not required to wait and hence relatively performance is high.
4)	Introduced in 1.0 version.	4)	Introduced in 1.5 versions.

String vs StringBuffer vs StringBuilder:

- If the content is fixed and won't change frequently then we should go for String.
- If the content will change frequently but Thread safety is required then we should go for StringBuffer.
- If the content will change frequently and Thread safety is not required then we should go for StringBuilder.

Method chaining:

- For most of the methods in String, StringBuffer and StringBuilder the return type is same type only. Hence after applying method on the result we can call another method which forms method chaining.

Example:

```
s.m1().m2().m3()……..
```

- In method chaining all methods will be evaluated from left to right.

Example:

```
class StringBufferDemo
{
    public static void main(String[] args)
    {
    StringBuffer sb=new StringBuffer();
    sb.append("vijaya").insert(6,"bhaskarreddy").delete(13,17).reverse().append("solutions").insert(22,"abcdf").reverse();
    System.out.println(sb);//sfdcbanoitulosvijayabhaskary
    }
}
```

Wrapper classes

- The main objectives of wrapper classes are:
- To wrap primitives into object form so that we can handle primitives also just like objects.
- To define several utility functions which are required for the primitives.

Constructors:

- All most all wrapper classes define the following 2 constructors one can take corresponding primitive as argument and the other can take String as argument.

Example:

1) **Integer i=new Integer(10);**
2) **Integer i=new Integer("10");**

- If the String is not properly formatted then we will get runtime exception saying "**NumberFormatException**".

Example:

```
class WrapperClassDemo
{
    public static void main(String[] args)throws Exception
    {
    Integer i=new Integer("ten");
    System.out.println(i);//NumberFormatException
    }
}
```

- Float class defines 3 constructors with float, String and double arguments.

1) **Float f=new Float (10.5f);**
2) **Float f=new Float ("10.5f");**

3) **Float f=new Float(10.5);**
4) **Float f=new Float ("10.5");**

- Character class defines only one constructor which can take char primitive as argument there is no String argument constructor.
  Character ch=new Character('a');//valid
  Character ch=new Character("a");//invalid
- Boolean class defines 2 constructors with boolean primitive and String arguments.
- If we want to pass boolean primitive the only allowed values are true, false where case should be lower case.
  Example:

```
        Boolean b=new Boolean(true);
    //Boolean b1=new Boolean(True);//C.E
    //Boolean b=new Boolean(False);//C.E
```

- If we are passing String argument then case is not important and content is not important. If the content is case insensitive String of true then it is treated as true in all other cases it is treated as false.
  Example 1:

```
class WrapperClassDemo
{
    public static void main(String[] args)throws Exception
    {
    Boolean b1=new Boolean("true");
    Boolean b2=new Boolean("True");
    Boolean b3=new Boolean("false");
    Boolean b4=new Boolean("False");
    Boolean b5=new Boolean("bhaskar");
    System.out.println(b1);//true
    System.out.println(b2);//true
    System.out.println(b3);//false
    System.out.println(b4);//false
    System.out.println(b5);//false
    }
}
```

Example 2(for exam purpose):

```
class WrapperClassDemo
{
    public static void main(String[] args)throws Exception
    {
    Boolean b1=new Boolean("yes");
    Boolean b2=new Boolean("no");
    System.out.println(b1);//false
    System.out.println(b2);//false
    System.out.println(b1.equals(b2));//true
    System.out.println(b1==b2);//false
    }
}
```

Wrapper class	Constructor summery
Byte	byte, String
Short	short, String
Integer	Int, String
Long	long, String
Float	float, String, double
Double	double, String
Character	char,~~String~~
Boolean	boolean, String

Note:

1) In all wrapper classes toString() method is overridden to return its content.
2) In all wrapper classes .equals() method is overridden for content compression.

Utility methods:

1) valueOf() method.
2) XXXValue() method.
3) parseXxx() method.
4) toString() method.

valueOf() method: We can use valueOf() method to create wrapper object for the given primitive or String this method is alternative to constructor.

Form 1: Every wrapper class except Character class contains a static valueOf() method to create wrapper object for the given String.

public static wrapper valueOf(String s);

Example:

```
class WrapperClassDemo
{
        public static void main(String[] args)throws Exception
        {
        Integer i=Integer.valueOf("10");
        Double d=Double.valueOf("10.5");
        Boolean b=Boolean.valueOf("bhaskar");
        System.out.println(i);//10
        System.out.println(d);//10.5
        System.out.println(b);//false
        }
}
```

Form 2: Every integral type wrapper class (Byte, Short, Integer, and Long) contains the following valueOf() method to convert specified radix string to wrapper object.

public static Integer valueOf(String s, int radix)

base

Note: the allowed radix range is 2-36.

base 2 ——— **0 To 1**
base 8 ——— **0 To 7**
base 10 ——— **0 To 9**
base 11 ——— **0 To 9,a**
base 16 ——— **0 To 9,a To f**
base 36 ——— **0 To 9,a to z**

Example:

```
class WrapperClassDemo
{
      public static void main(String[] args)
      {
      Integer i=Integer.valueOf("100",2);
      System.out.println(i);//4
      }
}
```

Analysis:

1 0 0
$2^2 2^1 2^0$

$2^0*0+2^1*0+2^2*1$

$1*0+2*0+4*1$

$0+0+4=4$

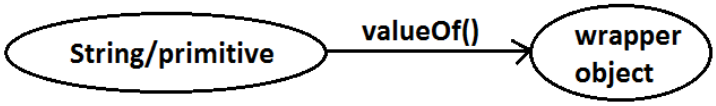
Form 3: Every wrapper class including Character class defines valueOf() method to convert primitive to wrapper object.

public static wrapper valueOf(primitive p);

Example:

```
class WrapperClassDemo
{
      public static void main(String[] args)throws Exception
      {
      Integer i=Integer.valueOf(10);
      Double d=Double.valueOf(10.5);
      Boolean b=Boolean.valueOf(true);
      System.out.println(i);//10
      System.out.println(d);//10.5
      System.out.println(b);//true
      }
}
```

Diagram:



xxxValue() method: We can use xxxValue() methods to convert wrapper object to primitive.

- Every number type wrapper class (Byte, Short, Integer, Long, Float, Double) contains the following 6 xxxValue() methods to convert wrapper object to primitives.

1) **public byte byteValue()**
2) **public short shortValue()**
3) **public int intValue()**
4) **public long longValue()**
5) **public float floatValue()**
6) **pblic double doubleValue();**

Example:

```
class WrapperClassDemo
{
    public static void main(String[] args)throws Exception
    {
    Integer i=new Integer(130);
    System.out.println(i.byteValue());//-126
    System.out.println(i.shortValue());//130
    System.out.println(i.intValue());//130
    System.out.println(i.longValue());//130
    System.out.println(i.floatValue());//130.0
    System.out.println(i.doubleValue());//130.0
    }
}
```

charValue() method: Character class contains charValue() method to convert Character object to char primitive.

public char charValue();

Example:

```
class WrapperClassDemo
{
        public static void main(String[] args)
        {
        Character ch=new Character('a');
        char c=ch.charValue();
        System.out.println(c);//a
        }
}
```

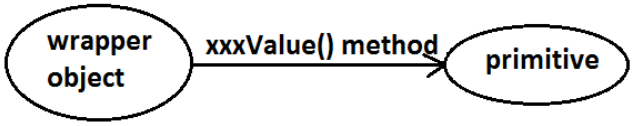
booleanValue() method: Boolean class contains booleanValue() method to convert Boolean object to boolean primitive.

public boolean booleanValue();

Example:

```
class WrapperClassDemo
{
    public static void main(String[] args)
    {
    Boolean b=new Boolean("bhaskar");
    boolean b1=b.booleanValue();
    System.out.println(b1);//false
    }
}
```

Diagram:



- In total there are 38(6*6+1+1) xxxValue() methods are possible.

parseXxx() method: We can use this method to convert String to corresponding primitive.

Form1: Every wrapper class except Character class contains a static parseXxx() method to convert String to corresponding primitive.

public static primitive parseXxx(String s);

Example:

```
class WrapperClassDemo
{
    public static void main(String[] args)
    {
    int i=Integer.parseInt("10");
    boolean b=Boolean.parseBoolean("bhaskar");
    double d=Double.parseDouble("10.5");
    System.out.println(i);//10
    System.out.println(b);//false
```

```
        System.out.println(d);//10.5
        }
}
```

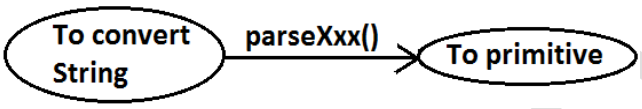
Form 2: integral type wrapper classes(Byte, Short, Integer, Long) contains the following parseXxx() method to convert specified radix String form to corresponding primitive.

public static primitive parseXxx(String s,int radix);

Example:

```
class WrapperClassDemo
{
        public static void main(String[] args)
        {
        int i=Integer.parseInt("100",2);
        System.out.println(i);//4
        }
}
```

Diagram:



toString() method: We can use toString() method to convert wrapper object (or) primitive to String.

Form 1:

public String toString();

- Every wrapper class including Character class contains the above toString() method to convert wrapper object to String.
- It is the overriding version of Object class toString() method.
- Whenever we are trying to print wrapper object reference internally this toString() method only executed.

Example:

```
class WrapperClassDemo
{
        public static void main(String[] args)
        {
        Integer i=Integer.valueOf("10");
        System.out.println(i);//10
        System.out.println(i.toString());//10
        }
}
```

Form 2: Every wrapper class contains a static toString() method to convert primitive to String.

public static String toString(primitive p);

Example:

```
class WrapperClassDemo
{
        public static void main(String[] args)
        {
        String s1=Integer.toString(10);
        String s2=Boolean.toString(true);
        System.out.println(s1);//10
        System.out.println(s2);//true
        }
}
```

Form 3:

- Integer and long classes contains the following static toString() method to convert the primitive to specified radix String form.

public static String toString(primitive p,int radix);

Example:

```
class WrapperClassDemo
{
        public static void main(String[] args)
        {
        String s1=Integer.toString(7,2);
        String s2=Integer.toString(17,2);
        System.out.println(s1);//111
        System.out.println(s2);//10001
        }
}
```

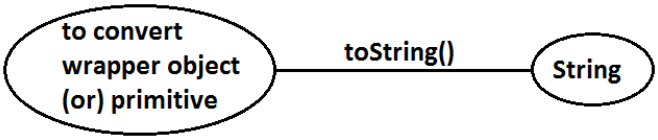
Form 4: Integer and Long classes contains the following toXxxString() methods.

public static String toBinaryString(primitive p);
public static String toOctalString(primitive p);
public static String toHexString(primitive p);

Example:

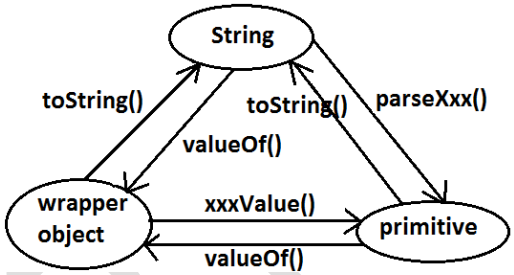
```
class WrapperClassDemo
{
	public static void main(String[] args)
	{
	String s1=Integer.toBinaryString(7);
	String s2=Integer.toOctalString(10);
	String s3=Integer.toHexString(20);
	System.out.println(s1);//111
	System.out.println(s2);//12
	System.out.println(s3);//14
	}
}
```

Diagram:



Dancing between String, wrapper object and primitive:

Diagram:



- String, StringBuffer, StringBuilder and all wrapper classes are final classes.
- The wrapper classes which are not child class of **Number** of Boolean and Character.
- The wrapper classes which are not direct class of Object of Byte, Short, Integer, Long, Float, Double.
- Sometimes we can consider **Void** is also as wrapper class.
- In addition to String all wrapper objects also immutable in java.

Autoboxing and Autounboxing

- Until 1.4 version we can't provide wrapper object in the place of primitive and primitive in the place of wrapper object all the required conversions should be performed explicitly by the programmer.

Example 1:

Program 1:

```
class AutoBoxingAndUnboxingDemo
{
	public static void main(String[] args)
	{
		Boolean b=new Boolean(true);
		if(b)
		{
			System.out.println("hello");
		}}}
```

1.4V →

```
E:\scjp>javac -source 1.4 AutoBoxingAndUnboxingDemo.java
AutoBoxingAndUnboxingDemo.java:6: incompatible types
found   : java.lang.Boolean
required: boolean
```

Program 2:

```
class AutoBoxingAndUnboxingDemo
{
	public static void main(String[] args)
	{
		Boolean b=new Boolean(true);
		if(b)
		{
			System.out.println("hello");
```

}}}

Output:

Hello

Example 2:

Program 1:

```
import java.util.*;
class AutoBoxingAndUnboxingDemo
{
    public static void main(String[] args)
    {
        ArrayList l=new ArrayList();
        l.add(10);
    }
}
```

```
E:\scjp>javac -source 1.4 AutoBoxingAndUnboxingDemo.java
AutoBoxingAndUnboxingDemo.java:7: cannot find symbol
symbol  : method add(int)
location: class java.util.ArrayList
```

Program 2:

```
import java.util.*;
class AutoBoxingAndUnboxingDemo
{
    public static void main(String[] args)
    {
        ArrayList l=new ArrayList();
        Integer i=new Integer(10);
        l.add(i);
    }
}
```

- But from 1.5 version onwards we can provide primitive value in the place of wrapper and wrapper object in the place of primitive all required conversions will be performed automatically by compiler. These automatic conversions are called Autoboxing and Autounboxing.

  Autoboxing: Automatic conversion of primitive to wrapper object by compiler is called Autoboxing.

  Example:

  Integer i=10; [compiler converts "int" to "Integer" automatically by Autoboxing]
- After compilation the above line will become.

  Integer i=Integer.valueOf(10);
- That is internally Autoboxing concept is implemented by using valueOf() method.

  Autounboxing: automatic conversion of wrapper object to primitive by compiler is called Autounboxing.

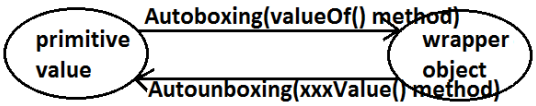
  Example:

  Integer i=new Integer(10);

  Int i=I; [compiler converts "Integer" to "int" automatically by Autounboxing]
- After compilation the above line will become.

  Int i=I.intValue();
- That is Autounboxing concept is internally implemented by using xxxValue() method.

  Diagram:



Example:

```
import java.util.*;
class AutoBoxingAndUnboxingDemo
{
    static Integer I=10;————(1) Autoboxing.
    public static void main(String[] args)
    {
        int i=I;————(2) Autounboxing
        methodOne(i);
    }                    (3) Autoboxing.
    public static void methodOne(Integer I)
    {
        int k=I;————(4) Autounboxing
        System.out.println(k);//10
    }
```

Note: From 1.5 version onwards we can use primitives and wrapper objects interchangly the required conversions will be performed automatically by compiler.

Example 1:

```
import java.util.*;
class AutoBoxingAndUnboxingDemo
{
    static Integer I=0;
    public static void main(String[] args)
    {
        int i=I;
        System.out.println(i);//0
    }
}
```

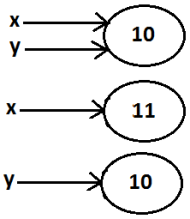
Example 2:

```
import java.util.*;
class AutoBoxingAndUnboxingDemo
{
    static Integer I;   null
    public static void main(String[] args)
    {
        int i=I;————>R.E:NullPointerException
        System.out.println(i);
                              ————> int i=I.intValue();
    }
}
```

Example 3:

```
import java.util.*;
class AutoBoxingAndUnboxingDemo
{
    public static void main(String[] args)
    {
        Integer x=10;
        Integer y=x;
        ++x;
        System.out.println(x);//11
        System.out.println(y);//10
        System.out.println(x==y);//false
    }
}
```

Diagram:

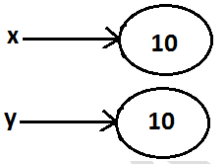


Note: All wrapper objects are immutable that is once we created a wrapper object we can't perform any changes in the existing object. If we are trying to perform any changes with those changes a new object will be created.

Example 4:

```
import java.util.*;
class AutoBoxingAndUnboxingDemo
{
	public static void main(String[] args)
	{
		Integer x=new Integer(10);
		Integer y=new Integer(10);
		System.out.println(x==y);//false
	}
}
```

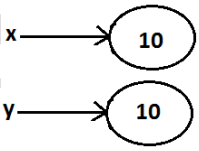
Diagram:



Example 5:

```
import java.util.*;
class AutoBoxingAndUnboxingDemo
{
	public static void main(String[] args)
	{
		Integer x=new Integer(10);
		Integer y=10;
		System.out.println(x==y);//false
	}
}
```

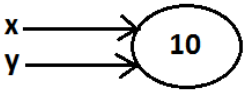
Diagram:



Example 6:

```
import java.util.*;
class AutoBoxingAndUnboxingDemo
{
	public static void main(String[] args)
	{
		Integer x=new Integer(10);
		Integer y=x;
		System.out.println(x==y);//true
	}
}
```

Diagram:

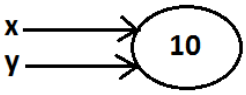


Example 7:

```
import java.util.*;
class AutoBoxingAndUnboxingDemo
{
	public static void main(String[] args)
	{
		Integer x=10;
```

```
                Integer y=10;
                System.out.println(x==y);//true
        }
}
```

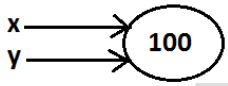
Diagram:



Example 8:

```
import java.util.*;
class AutoBoxingAndUnboxingDemo
{
        public static void main(String[] args)
        {
                Integer x=100;
                Integer y=100;
                System.out.println(x==y);//true
        }
}
```

Diagram:



Example 9:

```
import java.util.*;
class AutoBoxingAndUnboxingDemo
{
        public static void main(String[] args)
        {
                Integer x=1000;
                Integer y=1000;
                System.out.println(x==y);//false
        }
}
```

Diagram:

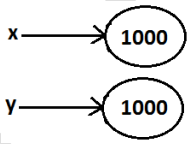
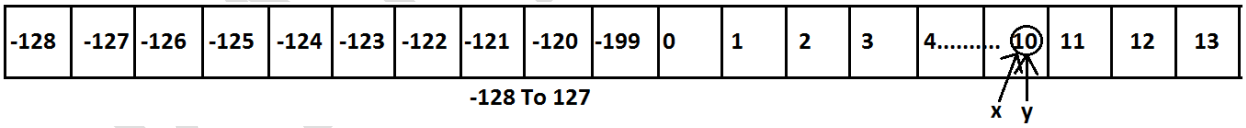


Diagram:



Conclusions:

- To implement the Autoboxing concept in every wrapper class a buffer of objects will be created at the time of class loading.
- By Autoboxing if an object is required to create 1st JVM will check whether that object is available in the buffer or not. If it is available then JVM will reuse that buffered object instead of creating new object. If the object is not available in the buffer then only a new object will be created. This approach improves performance and memory utilization.
- But this buffer concept is available only in the following cases.

Byte	Always
Short	-128 To 127
Integer	-128 To 127
Long	-128 To 127
Character	0 To 127
Boolean	Always

- In all the remaining cases compulsory a new object will be created.

Examples:

```
① Integer x=127;
  Integer y=127;
  System.out.println(x==y);//true
② Integer x=128;
  Integer y=128;
  System.out.println(x==y);//false
```

```
③ Boolean b1=true;
  Boolean b2=true;
  System.out.println(b1==b2);//true
④ Double d1=10.0;
  Double d2=10.0;
  System.out.println(d1==d2);//false
```

- Internally Autoboxing concept is implemented by using valueOf() method hence the above rule applicable even for valueOf() method also.

Examples:

```
① Integer x=new Integer(10);
  Integer y=new Integer(10);
  System.out.println(x==y);//false
② Integer x=10;
  Integer y=10;
  System.out.println(x==y);//true
```

```
③ Integer x=Integer.valueOf(10);
  Integer y=Integer.valueOf(10);
  System.out.println(x==y);//true
④ Integer x=10;
  Integer y=Integer.valueOf(10);
  System.out.println(x==y);//true
```

Note: When compared with constructors it is recommended to use valueOf() method to create wrapper object.

Overloading with respect to widening, Autoboxing and var-arg methods:

Case 1: Widening vs Autoboxing.

Widening: Converting a lower data type into a higher data type is called widening.

Example:

```
import java.util.*;
class AutoBoxingAndUnboxingDemo
{
	public static void methodOne(long l)
	{
		System.out.println("widening");
	}
	public static void methodOne(Integer i)
	{
		System.out.println("autoboxing");
	}
	public static void main(String[] args)
	{
		int x=10;
		methodOne(x);
	}
}
```

Output:

Widening

- Widening dominates Autoboxing.

Case 2: Widening vs var-arg method.

Example:

```
import java.util.*;
class AutoBoxingAndUnboxingDemo
{
  public static void methodOne(long l)
  {
	System.out.println("widening");
  }
  public static void methodOne(int... i)
  {
	System.out.println("var-arg method");
  }
  public static void main(String[] args)
  {
	int x=10;
	methodOne(x);
  }
}
```

Output:

Widening

- Widening dominates var-arg method.
  Case 3: Autoboxing vs var-arg method.
  Example:

```
import java.util.*;
class AutoBoxingAndUnboxingDemo
{
	public static void methodOne(Integer i)
	{
		System.out.println("Autoboxing");
	}
	public static void methodOne(int... i)
	{
		System.out.println("var-arg method");
	}
	public static void main(String[] args)
	{
		int x=10;
		methodOne(x);
	}
}
```

  Output:
  Autoboxing
- Autoboxing dominates var-arg method.
- In general var-arg method will get least priority. That is if no other method matched then only var-arg method will get chance. It is exactly same as "default" case inside a switch.

1) Widening
2) Autoboxing
3) **Var-arg method.**

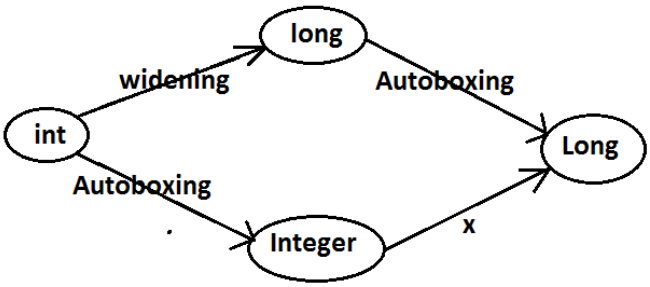
Case 4:

```
import java.util.*;
class AutoBoxingAndUnboxingDemo
{
	public static void methodOne(Long l)
	{
		System.out.println("Long");
	}
	public static void main(String[] args)
	{
		int x=10;
		methodOne(x);
	}
}
```

Output:

- methodOne(java.lang.Long) in AutoBoxingAndUnb oxingDemo cannot be applied to (int)

Diagram:



- Widening followed by Autoboxing is not allowed in java but Autoboxing followed by widening is allowed.

Case 5:

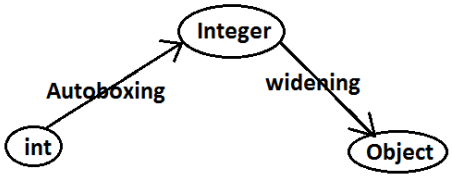
```
import java.util.*;
class AutoBoxingAndUnboxingDemo
{
```

```
                                        public static void methodOne(Object o)
                                        {
                                        System.out.println("Object");
                                        }
                                        public static void main(String[] args)
                                        {
                                        int x=10;
                                        methodOne(x);
}
}
```

Output:

Object

Diagram:



Which of the following declarations are valid?

1) Longl=10;(valid)
2) Long l=10;(invalid)(C.E)
3) Long l=10l;(Autoboxing)
4) Number n=10; (valid)
5) Object o=10.0; (valid)
6) int i=10l; (invalid)(C.E)

Java.IO Package

Agenda:

1) File
2) FileWriter
3) FileReader
4) BufferedWriter
5) BufferedReader
6) PrintWriter

File:

File f=new File("abc.txt");

- This line 1st checks whether abc.txt file is already available (or) not if it is already available then "f" simply refers that file. If it is not already available then it won't create any physical file just creates a java File object represents name of the file.

Example:

```
import java.io.*;
class  FileDemo
{
      public static void main(String[] args)throws IOException
      {
                File f=new File("cricket.txt");
                System.out.println(f.exists());//false
                f.createNewFile();
                System.out.println(f.exists());//true
                }
}
```

- A java File object can represent a directory also.

Example:

```
import java.io.*;
class  FileDemo
{
      public static void main(String[] args)throws IOException
      {
                File f=new File("cricket123");
                System.out.println(f.exists());//false
                f.mkdir();
                System.out.println(f.exists());//true
      }
}
```

Note: in UNIX everything is a file, java "file IO" is based on UNIX operating system hence in java also we can represent both files and directories by File object only.

File class constructors:

1) **File f=new File(String name);**
- Creates a java File object that represents name of the file or directory.

2) **File f=new File(String subdirname,String name);**
- Creates a File object that represents name of the file or directory present in specified sub directory.

3) **File f=new File(File subdir,String name);**

Requirement: Write code to create a file named with demo.txt in current working directory.

Program:

```
import java.io.*;
class  FileDemo
{
	public static void main(String[] args)throws IOException
	{
		File f=new File("demo.txt");
		f.createNewFile();
	}
}
```

Requirement: Write code to create a directory named with bhaskar123 in current working directory and create a file named with abc.txt in that directory.

Program:

```
import java.io.*;
class  FileDemo
{
	public static void main(String[] args)throws IOException
	{
		File f1=new File("bhaskar123");
		f1.mkdir();
		File f2=new File("bhaskar123","abc.txt");
		f2.createNewFile();
	}
}
```

Requirement: Write code to create a file named with demo.txt present in c:\xyz folder.

Program:

```
import java.io.*;
class  FileDemo
{
	public static void main(String[] args)throws IOException
	{
		File f=new File("c:\\bhaskar","demo.txt");
		f.createNewFile();
	}
}
```

Import methods of file class:

1) **boolean exists();**
- Returns true if the physical file or directory available.

2) **boolean createNewFile();**
- This method 1st checks whether the physical file is already available or not if it is already available then this method simply returns false. If this file is not already available then it will create a new file and returns true

3) **boolean mkdir();**

4) **boolean isFile();**
- Returns true if the File object represents a physical file.

5) **boolean isDirectory();**

6) **String[] list();**
- It returns the names of all files and subdirectories present in the specified directory.

7) **long length();**
- Returns the no of characters present in the file.

8) **boolean delete();**
- To delete a file or directory.

FileWriter:
- By using FileWriter we can write character data to the file.

Constructors:

FileWriter fw=new FileWriter(String name);
FileWriter fw=new FileWriter(File f);
- The above 2 constructors meant for overriding.
- Instead of overriding if we want append operation then we should go for the following 2 constructors.

FileWriter fw=new FileWriter(String name,boolean append);
FileWriter fw=new FileWriter(File f,boolean append);

- If the specified physical file is not already available then these constructors will create that file.

Methods:

1) write(int ch);

- To write a single character to the file.

2) write(char[] ch);

- To write an array of characters to the file.

3) write(String s);

- To write a String to the file.

4) flush();

- To give the guarantee the last character of the data also added to the file.

5) close();

- To close the stream.

Example:

```
import java.io.*;
class FileWriterDemo
{
	public static void main(String[] args)throws IOException
	{
		FileWriter fw=new FileWriter("cricket.txt",true);
		fw.write(98);//adding a single character
		fw.write("haskar\nsoftware solutions");
		fw.write("\n");
		char[] ch={'a','b','c'};
		fw.write(ch);
		fw.write("\n");
		fw.flush();
		fw.close();
	}
}
```

Output:

```
Bhaskar
Software solutions
ABC
```

FileReader: By using FileReader we can read character data from the file.

Constructors:

FileReader fr=new FileReader(String name);
FileReader fr=new FileReader (File f);

Methods:

1) int read();

- It attempts to read next character from the file and return its Unicode value. If the next character is not available then we will get -1.

2) int read(char[] ch);

- It attempts to read enough characters from the file into char[] array and returns the no of characters copied from the file into char[] array.

3) void close();

Approach 1:

```
import java.io.*;
class FileReaderDemo
{
	public static void main(String[] args)throws IOException
	{
		FileReader fr=new FileReader("cricket.txt");
		int i=fr.read();
		while(i!=-1)
		{
			System.out.print((char)i);
			i=fr.read();
		}
	}
}
```

Output:

```
Bhaskar
Software solutions
ABC
```

Approach 2:

```
import java.io.*;
class FileReaderDemo
{
	public static void main(String[] args)throws IOException
	{
		File f=new File("cricket.txt");
		FileReader fr=new FileReader(f);
		char[] ch=new char[(int)f.length()];
		fr.read(ch);
		for(char ch1:ch)
		{
			System.out.print(ch1);
		}
	}
}
```

Output:

VBR
Software solutions.

Usage of FileWriter and FileReader is not recommended because:

1) While writing data by FileWriter compulsory we should insert line separator(\n) manually which is a bigger headache to the programmer.
2) While reading data by FileReader we have to read character by character which is not convent to the programmer.
3) To overcome these limitations we should go for BufferedWriter and BufferedReader concepts.

BufferedWriter:

- By using BufferedWriter object we can write character data to the file.

Constructors:

BufferedWriter bw=new BufferedWriter(writer w);
BufferedWriter bw=new BufferedWriter(writer w,int buffersize);

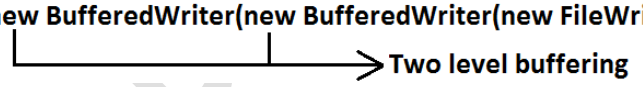
Note:

- BufferedWriter never communicates directly with the file it should communicates via some writer object.

Which of the following declarations are valid?

1) BufferedWriter bw=new BufferedWriter("cricket.txt"); (invalid)
2) BufferedWriter bw=new BufferedWriter (new File("cricket.txt")); (invalid)
3) BufferedWriter bw=new BufferedWriter (new FileWriter("cricket.txt")); (valid)

4)BufferedWriter bw=new BufferedWriter(new BufferedWriter(new FileWriter("cricket.txt"))); (valid)



Methods:

1) write(int ch);
2) write(char[] ch);
3) write(String s);
4) flush();
5) close();
6) newline();

- **Inserting a new line character to the file.**
- When compared with FileWriter which of the following capability(facility) is available as method in BufferedWriter.

1) Writing data to the file.(x)
2) Closing the writer.(x)
3) Flush the writer.(x)
4) Inserting newline character. (✓)

Example:

```
import java.io.*;
class BufferedWriterDemo
{
	public static void main(String[] args)throws IOException
	{
		FileWriter fw=new FileWriter("cricket.txt");
		BufferedWriter bw=new BufferedWriter(fw);
		bw.write(100);
		bw.newLine();
		char[] ch={'a','b','c','d'};
		bw.write(ch);
		bw.newLine();
		bw.write("bhaskar");
		bw.newLine();
		bw.write("software solutions");
```

```
                bw.flush();
                bw.close();
                }
}
```

Output:

```
d
abcd
bhaskar
software solutions
```

BufferedReader:

- This is the most enhanced(better) Reader to read character data from the file.

Constructors:

BufferedReader br=new BufferedReader(Reader r);
BufferedReader br=new BufferedReader(Reader r,int buffersize);

Note:

- BufferedReader can not communicate directly with the File it should communicate via some Reader object. The main advantage of BufferedReader over FileReader is we can read data line by line instead of character by character.

Methods:

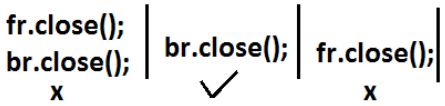
1) **int read();**
2) **int read(char[] ch);**
3) **String readLine();**
   - It attempts to read and return next line from the File. if the next line is not available then this method returns null.
4) **void close();**

Example:

```
import java.io.*;
class BufferedReaderDemo
{
    public static void main(String[] args)throws IOException
    {
                FileReader fr=new FileReader("cricket.txt");
                BufferedReader br=new BufferedReader(fr);
                String line=br.readLine();
                while(line!=null)
                {
                        System.out.println(line);
                        line=br.readLine();
                }
                br.close();
                }
}
```

Note:

- Whenever we are closing BufferedReader automatically underlying FileReader will be closed it is not required to close explicitly.



- Even this rule is applicable for BufferedWriter also.

PrintWriter:

- This is the most enhanced Writer to write text data to the file.
- By using FileWriter and BufferedWriter we can write only character data to the File but by using PrintWriter we can write any type of data to the File.

Constructors:

PrintWriter pw=new PrintWriter(String name);
PrintWriter pw=new PrintWriter(File f);
PrintWriter pw=new PrintWriter(Writer w);

- PrintWriter can communicate either directly to the File or via some Writer object also.

Methods:

```
write(int ch);
write (char[] ch);
write(String s);
flush();
close();
print(char ch);
print (int i);
```

```
print (double d);
print (boolean b);
print (String s);
println(char ch);
println (int i);
println(double d);
println(boolean b);
println(String s);
```

Example:

```
import java.io.*;
class PrintWriterDemo
{
    public static void main(String[] args)throws IOException
    {
            FileWriter fw=new FileWriter("cricket.txt");
            PrintWriter out=new PrintWriter(fw);
            out.write(100);
            out.println(100);
            out.println(true);
            out.println('c');
            out.println("bhaskar");
            out.flush();
            out.close();
    }
}
```

Output:

```
d100
true
c
bhaskar
```

What is the difference between write(100) and print(100)?

- In the case of write(100) the corresponding character "d" will be added to the File but in the case of print(100) "100" value will be added directly to the File.

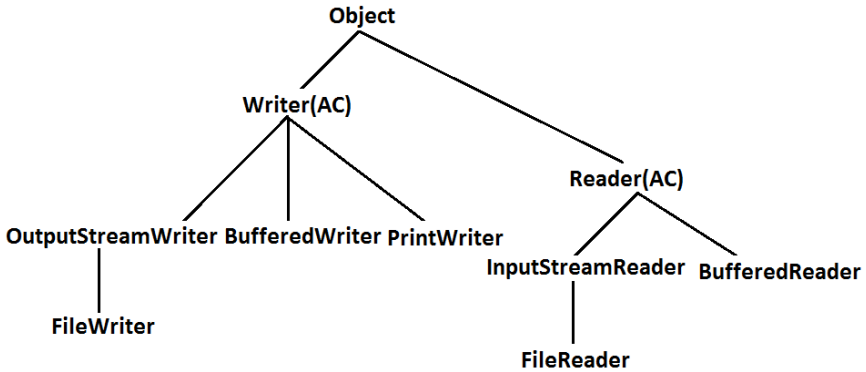
Note 1:

1) The most enhanced Reader to read character data from the File is BufferedReader.
2) The most enhanced Writer to write character data to the File is PrintWriter.

Note 2:

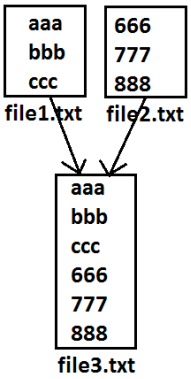
1) In general we can use Readers and Writers to handle character data. Where as we can use InputStreams and OutputStreams to handle binary data(like images, audio files, video files etc).
2) We can use OutputStream to write binary data to the File and we can use InputStream to read binary data from the File.

Diagram:



Requirement: Write a program to perform File merge(combine) operation.

Diagram:

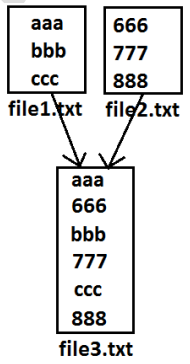


Program:

```
import java.io.*;
class FileWriterDemo1
{
	public static void main(String[] args)throws IOException
	{
		PrintWriter pw=new PrintWriter("file3.txt");
		BufferedReader br=new BufferedReader(new FileReader("file1.txt"));
		String line=br.readLine();
		while(line!=null)
		{
			pw.println(line);
			line=br.readLine();
		}
		br=new BufferedReader(new FileReader("file2.txt"));   reuse
		line=br.readLine();
		while(line!=null)
		{
			pw.println(line);
			line=br.readLine();
		}
		pw.flush();
		br.close();
		pw.close();
		}
}
```

Requirement: Write a program to perform file merge operation where merging should be performed line by line alternatively.

Diagram:



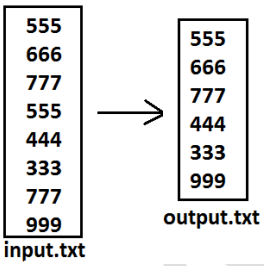
Program:

```
import java.io.*;
class FileWriterDemo1
{
	public static void main(String[] args)throws IOException
	{
		PrintWriter pw=new PrintWriter("file3.txt");
		BufferedReader br1=new BufferedReader(new FileReader("file1.txt"));
		BufferedReader br2=new BufferedReader(new FileReader("file2.txt"));
		String line1=br1.readLine();
		String line2=br2.readLine();
		while(line1!=null||line2!=null)
		{
```

```
                    if(line1!=null)
                    {
                    pw.println(line1);
                    line1=br1.readLine();
                    }
                    if(line2!=null)
                    {
                    pw.println(line2);
                    line2=br2.readLine();
                    }
            }
            pw.flush();
            br1.close();
            br2.close();
            pw.close();
            }
}
```

Requirement: Write a program to delete duplicate numbers from the file.

Diagram:

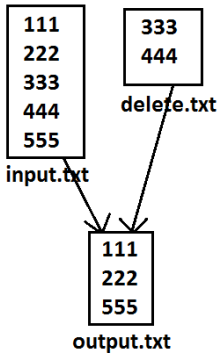


Program:

```
import java.io.*;
class FileWriterDemo1
{
    public static void main(String[] args)throws IOException
    {
            BufferedReader br1=new BufferedReader(new FileReader("input.txt"));
            PrintWriter out=new PrintWriter("output.txt");
            String target=br1.readLine();
            while(target!=null)
            {
                    boolean available=false;
                    BufferedReader br2=new BufferedReader(new FileReader("output.txt"));
                    String line=br2.readLine();
                    while(line!=null)
                    {
                            if(target.equals(line))
                            {
                                    available=true;
                                    break;
                            }
                            line=br2.readLine();
                    }
                    if(available==false)
                    {
                            out.println(target);
                            out.flush();
                    }
                    target=br1.readLine();
            }
    }
}
```

Requirement: write a program to perform file extraction operation.

Diagram:



Program:

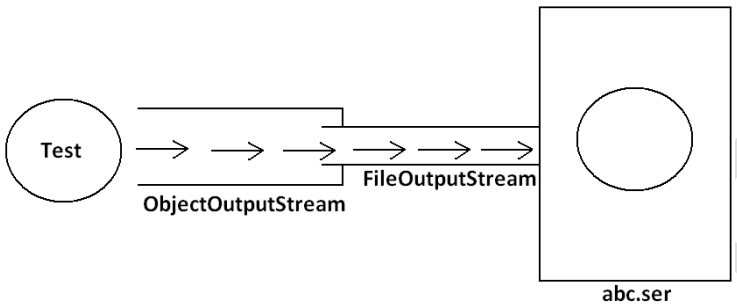
```
import java.io.*;
class FileWriterDemo1
{
	public static void main(String[] args)throws IOException
	{
		BufferedReader br1=new BufferedReader(new FileReader("input.txt"));
		PrintWriter pw=new PrintWriter("output.txt");
		String line=br1.readLine();
		while(line!=null)
		{
			boolean available=false;
			BufferedReader br2=new BufferedReader(new FileReader("delete.txt"));
			String target=br2.readLine();
			while(target!=null)
			{
				if(line.equals(target))
				{
					available=true;
					break;
				}
				target=br2.readLine();
			}
			if(available==false)
			{
				pw.println(line);
			}
			line=br1.readLine();
		}
		pw.flush();
	}
}
```

1. Introduction.
2. Object graph in serialization.
3. customized serialization.
4. Serialization with respect inheritance.

Serialization: The process of saving (or) writing state of an object to a file is called serialization but strictly speaking it is the process of converting an object from java supported form to either network supported form (or) file supported form.

- By using FileOutputStream and ObjectOutputStream classes we can achieve serialization process.

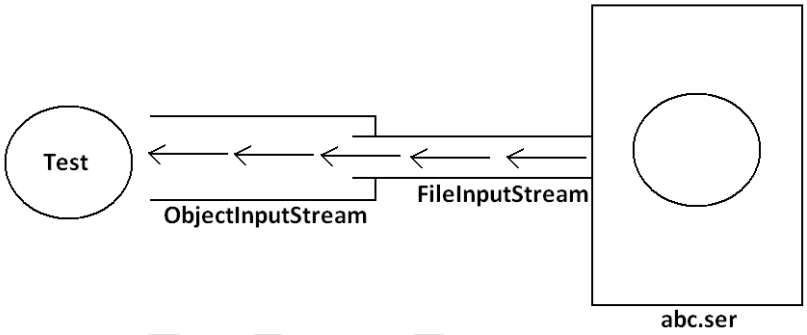
Diagram:



De-Serialization: The process of reading state of an object from a file is called DeSerialization but strictly speaking it is the process of converting an object from file supported form (or) network supported form to java supported form.

- By using FileInputStream and ObjectInputStream classes we can achieve DeSerialization.

Diagram:



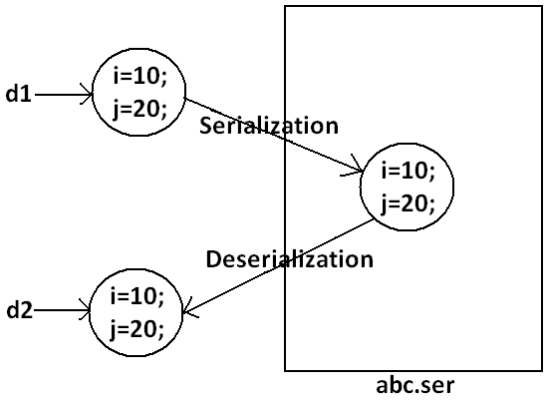
Example 1:

```
import java.io.*;
class Dog implements Serializable
{
int i=10;
int j=20;
}
class SerializableDemo
{
public static void main(String args[])throws Exception{
Dog d1=new Dog();
System.out.println("Serialization started");
FileOutputStream fos=new FileOutputStream("abc.ser");
ObjectOutputStream oos=new ObjectOutputStream(fos);
oos.writeObject(d1);
System.out.println("Serialization ended");
System.out.println("Deserialization started");
FileInputStream fis=new FileInputStream("abc.ser");
ObjectInputStream ois=new ObjectInputStream(fis);
Dog d2=(Dog)ois.readObject();
System.out.println("Deserialization ended");
System.out.println(d2.i+"....."+d2.j);
}
}
```

Output:

```
Serialization started
Serialization ended
Deserialization started
Deserialization ended
10.....20
```

Diagram:



Note: We can perform Serialization only for Serilizable objects.

- An object is said to be Serilizable if and only if the corresponding class implements Serializable interface.
- Serializable interface present in **java.io package** and does not contain any methods. It is marker interface. The required ability will be provided automatically by JVM.
- We can add any no. Of objects to the file and we can read all those objects from the file but in which order we wrote objects in the same order only the objects will come back. That is order is important.

Example2:

```
import java.io.*;
class Dog implements Serializable
{
int i=10;
int j=20;
}
class Cat implements Serializable
{
int i=30;
int j=40;
}
class SerializableDemo
{
public static void main(String args[])throws Exception{
Dog d1=new Dog();
Cat c1=new Cat();
System.out.println("Serialization started");
FileOutputStream fos=new FileOutputStream("abc.ser");
ObjectOutputStream oos=new ObjectOutputStream(fos);
oos.writeObject(d1);
oos.writeObject(c1);
System.out.println("Serialization ended");
System.out.println("Deserialization started");
FileInputStream fis=new FileInputStream("abc.ser");
ObjectInputStream ois=new ObjectInputStream(fis);
Dog d2=(Dog)ois.readObject();
Cat c2=(Cat)ois.readObject();
System.out.println("Deserialization ended");
System.out.println(d2.i+"....."+d2.j);
System.out.println(c2.i+"....."+c2.j);
}
}
```

Output:

```
Serialization started
Serialization ended
Deserialization started
Deserialization ended
10.....20
30.....40
```

Transient keyword:

- While performing serialization if we don't want to serialize the value of a particular variable then we should declare that variable with "transient" keyword.
- At the time of serialization JVM ignores the original value of transient variable and save default value.
- That is transient means "not to serialize".

Static Vs Transient:

- static variable is not part of object state hence they won't participate in serialization because of this declaring a static variable as transient these is no use.

Transient Vs Final:

- final variables will be participated into serialization directly by their values. Hence declaring a final variable as transient there is no use.

Example 3:

```
import java.io.*;
class Dog implements Serializable
{
static transient int i=10;
final transient int j=20;
}
class SerializableDemo
{
public static void main(String args[])throws Exception{
Dog d1=new Dog();
FileOutputStream fos=new FileOutputStream("abc.ser");
ObjectOutputStream oos=new ObjectOutputStream(fos);
oos.writeObject(d1);
FileInputStream fis=new FileInputStream("abc.ser");
ObjectInputStream ois=new ObjectInputStream(fis);
Dog d2=(Dog)ois.readObject();
System.out.println(d2.i+"....."+d2.j);
}
}
```

Output:

10.....20

Diagram:

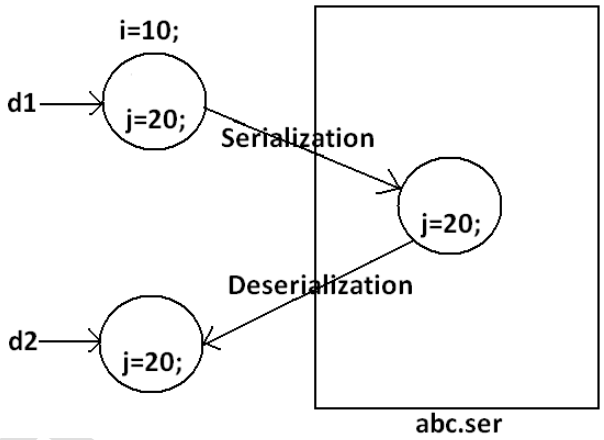


Table:

declaration	output
int i=10; int j=20;	10.....20
transient int i=10; int j=20;	0.....20
transient int i=10; transient static int j=20;	0.....20
transient final int i=10; transient int j=20;	10.....0
transient final int i=10; transient static int j=20;	10.....20

Object graph in serialization:

- Whenever we are serializing an object the set of all objects which are reachable from that object will be serialized automatically. This group of objects is nothing but object graph in serialization.
- In object graph every object should be Serializable otherwise we will get runtime exception saying "NotSerializableException".

Example 4:

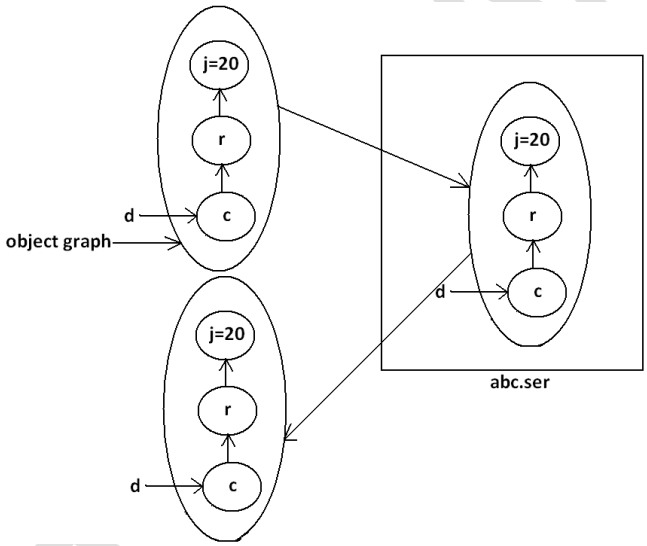
```
import java.io.*;
class Dog implements Serializable
{
Cat c=new Cat();
```

```
}
class Cat implements Serializable
{
Rat r=new Rat();
}
class Rat implements Serializable
{
int j=20;
}
class SerializableDemo
{
public static void main(String args[])throws Exception{
Dog d1=new Dog();
FileOutputStream fos=new FileOutputStream("abc.ser");
ObjectOutputStream oos=new ObjectOutputStream(fos);
oos.writeObject(d1);
FileInputStream fis=new FileInputStream("abc.ser");
ObjectInputStream ois=new ObjectInputStream(fis);
Dog d2=(Dog)ois.readObject();
System.out.println(d2.c.r.j);
}
}
```

Output:

20

Diagram:



- In the above example whenever we are serializing Dog object automatically Cat and Rat objects will be serialized because these are part of object graph of Dog object.
- Among Dog, Cat, Rat if at least one object is not serializable then we will get runtime exception saying "NotSerializableException".

Customized serialization:

Example 5:

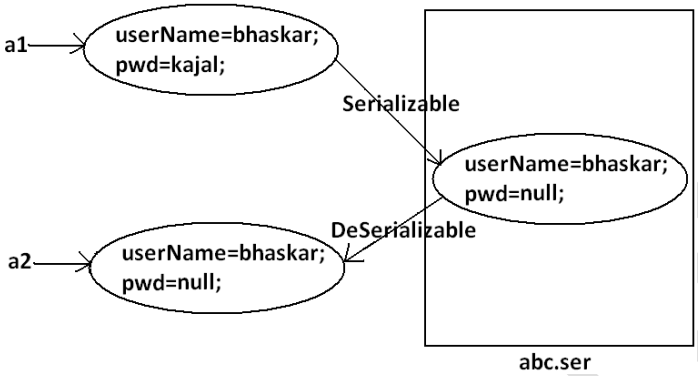
```
import java.io.*;
class Account implements Serializable
{
String userName="Bhaskar";
transient String pwd="kajal";
}
class CustomizedSerializeDemo
{
public static void main(String[] args)throws Exception{
Account a1=new Account();
System.out.println(a1.userName+"....."+a1.pwd);
FileOutputStream fos=new FileOutputStream("abc.ser");
ObjectOutputStream oos=new ObjectOutputStream(fos);
oos.writeObject(a1);
FileInputStream fis=new FileInputStream("abc.ser");
ObjectInputStream ois=new ObjectInputStream(fis);
Account a2=(Account)ois.readObject();
```

```
System.out.println(a2.userName+"....."+a2.pwd);
}
}
```

Output:

```
Bhaskar.....kajal
Bhaskar.....null
```

Diagram:



- In the above example before serialization Account object can provide proper username and password. But after Deserialization Account object can provide only username bur not password. This is due to declaring password as transient. Hence doing default serialization there may be a chance of loss of information due to transient keyword.
- We can recover this loss of information by using customized serialization.
- We can implements customized serialization by using the following two methods.

1) private void writeObject(OutputStream os) throws Exception.

- This method will be executed automatically by jvm at the time of serialization.
- It is a callback method. Hence at the time of serialization if we want to perform any extra work we have to define that in this method only.

2) private void readObject(InputStream is)throws Exception.

- This method will be executed automatically by JVM at the time of Deserialization. Hence at the time of Deserialization if we want to perform any extra activity we have to define that in this method only.

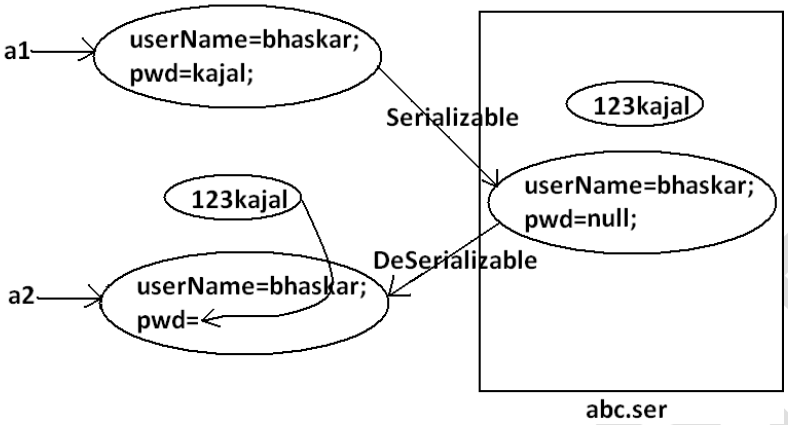
Example 6: Demo program for customized serialization to recover loss of information which is happen due to transient keyword.

```
import java.io.*;
class Account implements Serializable
{
String userName="Bhaskar";
transient String pwd="kajal";
private void writeObject(ObjectOutputStream os)throws Exception
{
os.defaultWriteObject();
String epwd="123"+pwd;
os.writeObject(epwd);
}
private void readObject(ObjectInputStream is)throws Exception{
is.defaultReadObject();
String epwd=(String)is.readObject();
pwd=epwd.substring(3);
}
}
class CustomizedSerializeDemo
{
public static void main(String[] args)throws Exception{
Account a1=new Account();
System.out.println(a1.userName+"....."+a1.pwd);
FileOutputStream fos=new FileOutputStream("abc.ser");
ObjectOutputStream oos=new ObjectOutputStream(fos);
oos.writeObject(a1);
FileInputStream fis=new FileInputStream("abc.ser");
ObjectInputStream ois=new ObjectInputStream(fis);
Account a2=(Account)ois.readObject();
System.out.println(a2.userName+"....."+a2.pwd);
}
}
```

Output:

Bhaskar.....kajal
Bhaskar.....kajal

Diagram:



- At the time of Account object serialization JVM will check is there any writeObject() method in Account class or not. If it is not available then JVM is responsible to perform serialization(default serialization). If Account class contains writeObject() method then JVM feels very happy and executes that Account class writeObject() method. The same rule is applicable for readObject() method also.

Serialization with respect to inheritance:

Case 1:

- If parent class implements Serializable then automatically every child class by default implements Serializable. That is Serializable nature is inheriting from parent to child.

Example 7:

```
import java.io.*;
class Animal implements Serializable
{
int i=10;
}
class Dog extends Animal
{
int j=20;
}
class SerializableWRTInheritance
{
public static void main(String[] args)throws Exception{
Dog d1=new Dog();
System.out.println(d1.i+"....."+d1.j);
FileOutputStream fos=new FileOutputStream("abc.ser");
ObjectOutputStream oos=new ObjectOutputStream(fos);
oos.writeObject(d1);
FileInputStream fis=new FileInputStream("abc.ser");
ObjectInputStream ois=new ObjectInputStream(fis);
Dog d2=(Dog)ois.readObject();
System.out.println(d2.i+"....."+d2.j);
}
}
```

Output:
10.....20
10.....20

- Even though Dog class does not implements Serializable interface explicitly but we can Serialize Dog object because its parent class animal already implements Serializable interface.

Case 2:

- Even though parent class does not implements Serializable we can serialize child object if child class implements Serializable interface.
- At the time of serialization JVM ignores the values of instance variables which are coming from non Serializable parent JVM saves default values for those variables.
- At the time of Deserialization JVM checks whether any parent class is non Serializable or not. If any parent class is non Serializable JVM creates a separate object for every non Serializable parent and shares its instance variables to the current object.
- For this JVM always calls no arg constructor(default constructor) of that non Serializable parent hence every non Serializable parent should compulsory contain no arg constructor otherwise we will get runtime exception.

Example 8:

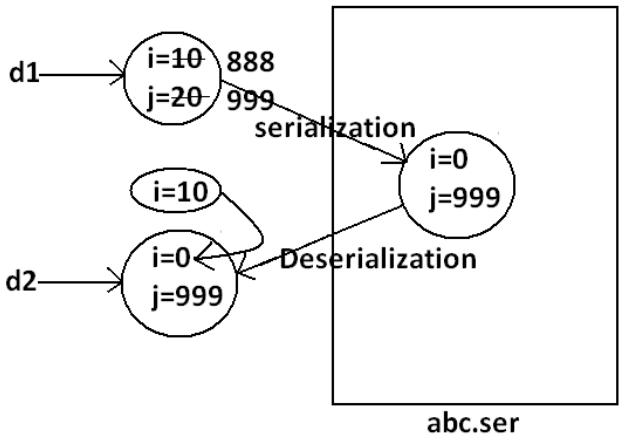
```
import java.io.*;
```

```
class Animal
{
int i=10;
Animal(){
System.out.println("Animal constructor called");
}
}
class Dog extends Animal implements Serializable
{
int j=20;
Dog(){
System.out.println("Dog constructor called");
}
}
class SerializableWRTInheritance
{
public static void main(String[] args)throws Exception{
Dog d1=new Dog();
d1.i=888;
d1.j=999;
FileOutputStream fos=new FileOutputStream("abc.ser");
ObjectOutputStream oos=new ObjectOutputStream(fos);
oos.writeObject(d1);
System.out.println("Deserialization started");
FileInputStream fis=new FileInputStream("abc.ser");
ObjectInputStream ois=new ObjectInputStream(fis);
Dog d2=(Dog)ois.readObject();
System.out.println(d2.i+"....."+d2.j);
}
}
```

Output:

```
Animal constructor called
Dog constructor called
Deserialization started
Animal constructor called
10.....999
```

Diagram:



Collections

- An array is an indexed collection of fixed no of homogeneous data elements. (or)
- An array represents a group of elements of same data type.
- The main advantage of array is we can represent huge no of elements by using single variable. So that readability of the code will be improved.

Limitations of Object[] array:

1) Arrays are fixed in size that is once we created an array there is no chance of increasing (or) decreasing the size based on our requirement hence to use arrays concept compulsory we should know the size in advance which may not possible always.
2) Arrays can hold only homogeneous data elements.

Example:

```
Student[] s=new Student[10000];
s[0]=new Student();//valid
s[1]=new Customer();//invalid(compile time error)
```

Compile time error:

```
Test.java:7: cannot find symbol
Symbol: class Customer
Location: class Test
s[1]=new Customer();
```

3) But we can resolve this problem by using object type array(Object[]).

Example:

```
Object[] o=new Object[10000];
o[0]=new Student();
o[1]=new Customer();
```

4) Arrays concept is not implemented based on some data structure hence ready-made methods support we can't expert. For every requirement we have to write the code explicitly.

- To overcome the above limitations we should go for collections concept.

1) Collections are growable in nature that is based on our requirement we can increase (or) decrease the size hence memory point of view collections concept is recommended to use.
2) Collections can hold both homogeneous and heterogeneous objects.
3) Every collection class is implemented based on some standard data structure hence for every requirement ready-made method support is available being a programmer we can use these methods directly without writing the functionality on our own.

Differences between Arrays and Collections?

Arrays	Collections
1) Arrays are fixed in size.	1) Collections are growable in nature.
2) Memory point of view arrays are not recommended to use.	2) Memory point of view collections are highly recommended to use.
3) Performance point of view arrays are recommended to use.	3) Performance point of view collections are not recommended to use.
4) Arrays can hold only homogeneous data type elements.	4) Collections can hold both homogeneous and heterogeneous elements.
5) There is no underlying data structure for arrays and hence there is no readymade method support.	5) Every collection class is implemented based on some standard data structure and hence readymade method support is available.
6) Arrays can hold both primitives and object types.	6) Collections can hold only objects but not primitives.

Collection: If we want to represent a group of objects as single entity then we should go for collections.

Collection framework: It defines several classes and interfaces to represent a group of objects as a single entity.

java	C++
Collection	Containers
Collection framework	STL(Standard Template Library)

9(Nine) key interfaces of collection framework:

1. Collection 2. List 3. Set
4. SortedSet 5. NavigableSet 6. Queue
7. Map 8. SortedMap 9. NavigableMap

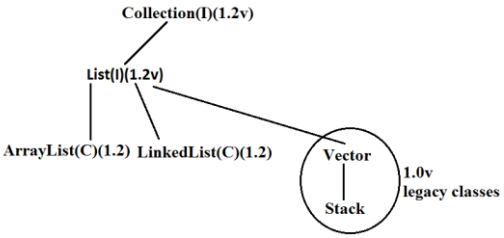
Collection:

1) If we want to represent a group of "individual objects" as a single entity then we should go for collection.
2) In general we can consider collection as root interface of entire collection framework.
3) Collection interface defines the most common methods which can be applicable for any collection object.

List:

1) It is the child interface of Collection.
2) If we want to represent a group of individual objects as a single entity where "duplicates are allow and insertion order must be preserved" then we should go for List interface.

Diagram:

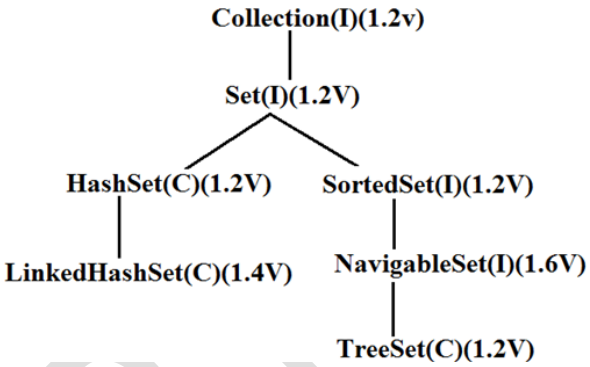


- Vector and Stack classes are reengineered in 1.2 versions to implement List interface.

Set:

1) It is the child interface of Collection.
2) If we want to represent a group of individual objects as single entity "where duplicates are not allow and insertion order is not preserved" then we should go for Set interface.

Diagram:



SortedSet:

1) It is the child interface of Set.
2) If we want to represent a group of "unique objects" according to some sorting order then we should go for SortedSet.

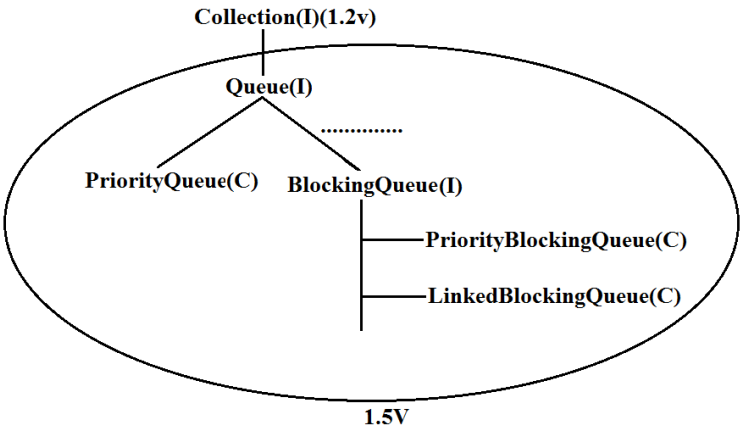
NavigableSet:

1) It is the child interface of SortedSet.
2) It provides several methods for navigation purposes.

Queue:

1) It is the child interface of Collection.
2) If we want to represent a group of individual objects prior to processing then we should go for queue concept.

Diagram:



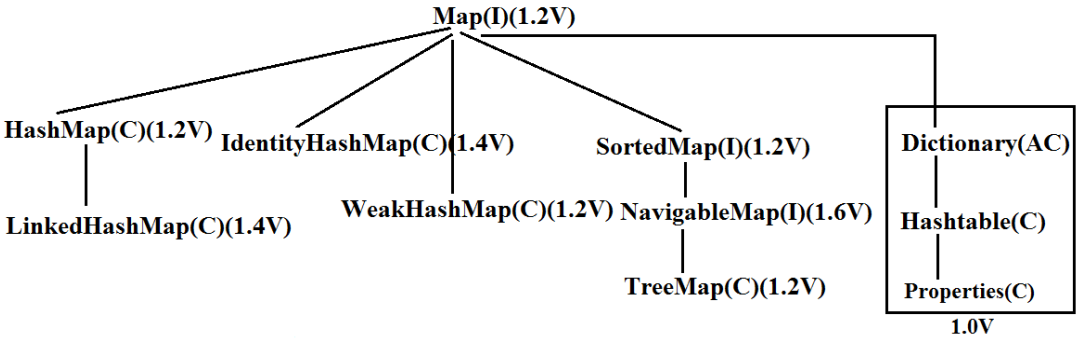
Note: All the above interfaces (Collection, List, Set, SortedSet, NavigableSet, and Queue) meant for representing a group of individual objects.

- If we want to represent a group of objects as key-value pairs then we should go for Map.

Map:

1) Map is not child interface of Collection.
2) If we want to represent a group of objects as key-value pairs then we should go for Map interface.
3) Duplicate keys are not allowed but values can be duplicated.

Diagram:



SortedMap:

1) It is the child interface of Map.
2) If we want to represent a group of objects as key value pairs "according to some sorting order of keys" then we should go for SortedMap.

NavigableMap:

1) It is the child interface of SortedMap and defines several methods for navigation purposes.

What is the difference between Collection and Collections?

- "Collection is an "interface" which can be used to represent a group of objects as a single entity. Whereas "Collections is an utility class" present in java.util package to define several utility methods for Collection objects.

Collection----------interface

Collections----------class

- In collection framework the following are legacy characters.

1) Enumeration(I)
2) Dictionary(AC)
3) Vector(C)
4) Stack(C)
5) Hashtable(C)
6) Properties(C)

Diagram:

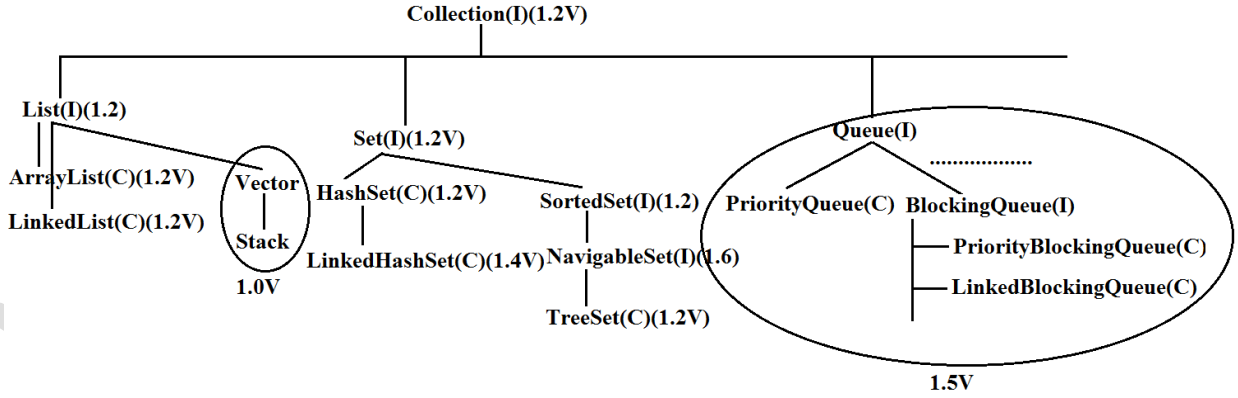
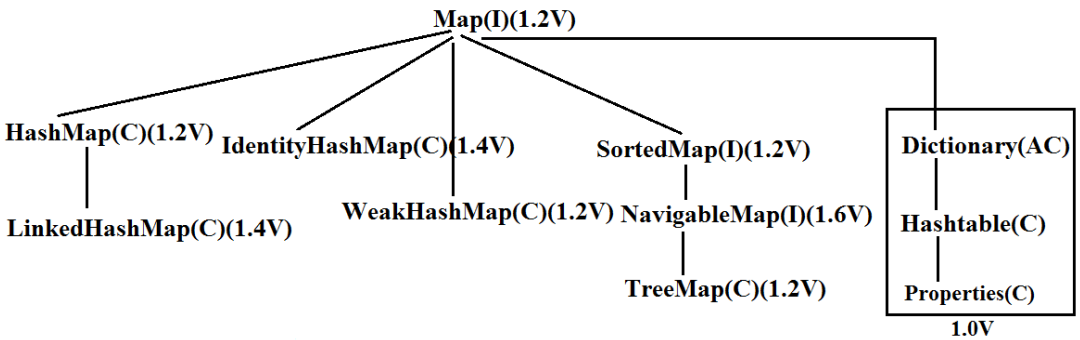


Diagram:



Collection interface:

- If we want to represent a group of individual objects then we should go for Collection interface. This interface defines the most common general methods which can be applicable for any Collection object.
- The following is the list of methods present in Collection interface.

1) boolean add(Object o);
2) boolean addAll(Collection c);
3) boolean remove(Object o);
4) boolean removeAll(Object o);
5) boolean retainAll(Collection c);

- To remove all objects except those present in c.

6) Void clear();
7) boolean contains(Object o);
8) boolean containsAll(Collection c);
9) boolean isEmpty();
10) Int size();
11) Object[] toArray();
12) Iterator iterator();

- There is no concrete class which implements Collection interface directly.

List interface:

- It is the child interface of Collection.
- If we want to represent a group of individual objects where duplicates are allow and insertion order is preserved. Then we should go for List.
- We can differentiate duplicate objects and we can maintain insertion order by means of index hence "index play very important role in List".
- List interface defines the following specific methods.

1) boolean add(int index,Object o);
2) boolean addAll(int index,Collectio c);
3) Object get(int index);
4) Object remove(int index);
5) Object set(int index,Object new);//to replace
6) Int indexOf(Object o);

- Returns index of first occurrence of "o".

7) Int lastIndexOf(Object o);
8) ListIterator listIterator();

ArrayList:

1) The underlying data structure is resizable array (or) growable array.
2) Duplicate objects are allowed.
3) Insertion order preserved.
4) Heterogeneous objects are allowed.
5) Null insertion is possible.

Constructors:

1) ArrayList a=new ArrayList();

- Creates an empty ArrayList object with default initial capacity "10" if ArrayList reaches its max capacity then a new ArrayList object will be created with

New capacity=(current capacity*3/2)+1

2) ArrayList a=new ArrayList(int initialcapacity);

- Creates an empty ArrayList object with the specified initial capacity.

3) ArrayList a=new ArrayList(collection c);

- Creates an equivalent ArrayList object for the given Collection that is this constructor meant for inter conversation between collection objects. That is to dance between collection objects.

Demo program for ArrayList:

```
import java.util.*;
class ArrayListDemo
{
    public static void main(String[] args)
    {
        ArrayList a=new ArrayList();
        a.add("A");
        a.add(10);
        a.add("A");
        a.add(null);
        System.out.println(a);//[A, 10, A, null]
        a.remove(2);
        System.out.println(a);//[A, 10, null]
        a.add(2,"m");
        a.add("n");
```

```
        System.out.println(a);//[A, 10, m, null, n]
        }
}
```

- Usually we can use collection to hold and transfer objects to provide support for this requirement every collection class implements Serializable and Cloneable interfaces.
- ArrayList and Vector classes implements RandomAccess interface so that any random element we can access with the same speed.
- RandomAccess interface present in util package and doesn't contain any methods. It is a marker interface.

Differences between ArrayList and Vector?

	ArrayList		Vector
1)	No method is synchronized	1)	Every method is synchronized
2)	At a time multiple Threads are allow to operate on ArrayList object and hence ArrayList object is not Thread safe.	2)	At a time only one Thread is allow to operate on Vector object and hence Vector object is Thread safe.
3)	Relatively performance is high because Threads are not required to wait.	3)	Relatively performance is low because Threads are required to wait.
4)	It is non legacy and introduced in 1.2v	4)	It is legacy and introduced in 1.0v

Getting synchronized version of ArrayList object:

- Collections class defines the following method to return synchronized version of List.

Public static List synchronizedList(list l);

Example:

```
ArrayList a=new arrayList();
List l1=collections.synchronizedList(a);
```

synchronized version

nonsynchronized version

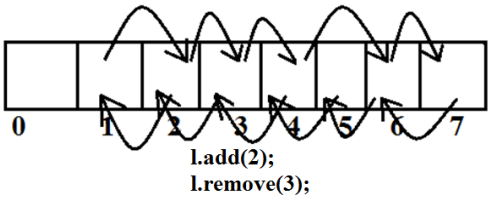
- Similarly we can get synchronized version of Set and Map objects by using the following methods.

1) public static Set synchronizedSet(Set s);

2) public static Map synchronizedMap(Map m);

- ArrayList is the best choice if our frequent operation is retrieval.
- ArrayList is the worst choice if our frequent operation is insertion (or) deletion in the middle because it requires several internal shift operations.

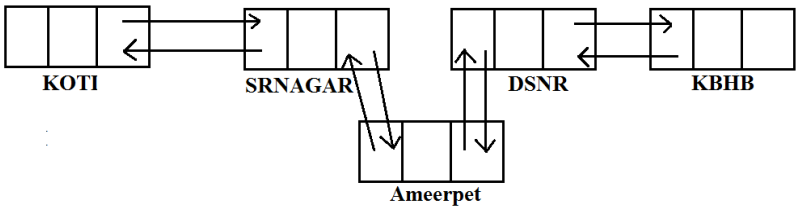
Diagram:



LinkedList:

1) The underlying data structure is double LinkedList
2) If our frequent operation is insertion (or) deletion in the middle then LinkedList is the best choice.
3) If our frequent operation is retrieval operation then LinkedList is worst choice.
4) Duplicate objects are allowed.
5) Insertion order is preserved.
6) Heterogeneous objects are allowed.
7) Null insertion is possible.
8) Implements Serializable and Cloneable interfaces but not RandomAccess.

Diagram:



- Usually we can use LinkedList to implement Stacks and Queues to provide support for this requirement LinkedList class defines the following 6 specific methods.

1) void addFirst(Object o);
2) void addLast(Object o);
3) Object getFirst();

4) Object getLast();
5) Object removeFirst();
6) Object removeLast();

- We can apply these methods only on LinkedList object.

Constructors:

1) LinkedList l=new LinkedList();

- Creates an empty LinkedList object.

2) LinkedList l=new LinkedList(Collection c);

- To create an equivalent LinkedList object for the given collection.

Example:

```
import java.util.*;
class LinkedListDemo
{
    public static void main(String[] args)
    {
        LinkedList l=new LinkedList();
        l.add("bhaskar");
        l.add(30);
        l.add(null);
        l.add("bhaskar");
        System.out.println(l);//[bhaskar, 30, null, bhaskar]
        l.set(0,"software");
        System.out.println(l);//[software, 30, null, bhaskar]
        l.set(0,"venky");
        System.out.println(l);//[venky, 30, null, bhaskar]
        l.removeLast();
        System.out.println(l);//[venky, 30, null]
        l.addFirst("vvv");
        System.out.println(l);//[vvv, venky, 30, null]
    }
}
```

Vector:

1) The underlying data structure is resizable array (or) growable array.
2) Duplicate objects are allowed.
3) Insertion order is preserved.
4) Heterogeneous objects are allowed.
5) Null insertion is possible.
6) Implements Serializable, Cloneable and RandomAccess interfaces.

- Every method present in Vector is synchronized and hence Vector is Thread safe.

Vector specific methods:

To add objects:

1) add(Object o);-----Collection
2) add(int index,Object o);-----List
3) addElement(Object o);-----Vector

To remove elements:

1) remove(Object o);--------Collection
2) remove(int index);----------List
3) removeElement(Object o);----Vector
4) removeElementAt(int index);-----Vector
5) removeAllElements();-----Vector
6) clear();-------Collection

To get objects:

1) Object get(int index);----------List
2) Object elementAt(int index);-----Vector
3) Object firstElement();----------Vector
4) Object lastElement();----------Vector

Other methods:

1) Int size();//How many objects are added
2) Int capacity();//Total capacity
3) Enumeration elements();

Constructors:

1) Vector v=new Vector();

- Creates an empty Vector object with default initial capacity 10.
- Once Vector reaches its maximum capacity then a new Vector object will be created with double capacity. That is "newcapacity=currentcapacity*2".

2) Vector v=new Vector(int initialcapacity);

3) Vector v=new Vector(int initialcapacity, int incrementalcapacity);
4) Vector v=new Vector(Collection c);

Example:

```
import java.util.*;
class VectorDemo
{
	public static void main(String[] args)
	{
		Vector v=new Vector();
		System.out.println(v.capacity());//10
		for(int i=1;i<=10;i++)
		{
			v.addElement(i);
		}
	System.out.println(v.capacity());//10
	v.addElement("A");
	System.out.println(v.capacity());//20
	System.out.println(v);//[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, A]
	}
}
```

Stack:

1) It is the child class of Vector.
2) Whenever last in first out order required then we should go for Stack.

Constructor:

- It contains only one constructor.

Stack s= new Stack();

Methods:

1) Object push(Object o);
- To insert an object into the stack.

2) Object pop();
- To remove and return top of the stack.

3) Object peek();
- To return top of the stack without removal.

4) boolean empty();
- Returns true if Stack is empty.

5) Int search(Object o);
- Returns offset if the element is available otherwise returns "-1"

Example:

```
import java.util.*;
class StackDemo
{
	public static void main(String[] args)
	{
		Stack s=new Stack();
		s.push("A");
		s.push("B");
		s.push("C");
		System.out.println(s);//[A, B, C]
		System.out.println(s.pop());//C
		System.out.println(s);//[A, B]
		System.out.println(s.peek());//B
		System.out.println(s.search("A"));//2
		System.out.println(s.search("Z"));//-1
		System.out.println(s.empty());//false
	}
}
```

The 3 cursors of java:

- If we want to get objects one by one from the collection then we should go for cursor. There are 3 types of cursors available in java. They are:

1) Enumeration
2) Iterator
3) ListIterator

Enumeration:

1) We can use Enumeration to get objects one by one from the legacy collection objects.
2) We can create Enumeration object by using elements() method.

Enumeration e=v.elements();

Vector Object

- Enumeration interface defines the following two methods

1) public boolean hasMoreElements();
2) public Object nextElement();

Example:

```
import java.util.*;
class  EnumerationDemo
{
	public static void main(String[] args)
	{
		Vector v=new Vector();
		for(int i=0;i<=10;i++)
		{
			v.addElement(i);
		}
		System.out.println(v);//[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
		Enumeration e=v.elements();
		while(e.hasMoreElements())
		{
			Integer i=(Integer)e.nextElement();
			if(i%2==0)
				System.out.println(i);//0 2 4 6 8 10
		}
		System.out.print(v);//[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
	}
}
```

Limitations of Enumeration:

1) We can apply Enumeration concept only for legacy classes and it is not a universal cursor.
2) By using Enumeration we can get only read access and we can't perform remove operations.
3) To overcome these limitations sun people introduced Iterator concept in 1.2v.

Iterator:

1) We can use Iterator to get objects one by one from any collection object.
2) We can apply Iterator concept for any collection object and it is a universal cursor.
3) While iterating the objects by Iterator we can perform both read and remove operations.

- We can get Iterator object by using iterator() method of Collection interface.

Iterator itr=c.iterator();

- Iterator interface defines the following 3 methods.

1) public boolean hasNext();
2) public object next();
3) public void remove();

Example:

```
import java.util.*;
class IteratorDemo
{
	public static void main(String[] args)
	{
		ArrayList a=new ArrayList();
		for(int i=0;i<=10;i++)
		{
			a.add(i);
		}
		System.out.println(a);//[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
		Iterator itr=a.iterator();
		while(itr.hasNext())
		{
			Integer i=(Integer)itr.next();
			if(i%2==0)
				System.out.println(i);//0, 2, 4, 6, 8, 10
			else
				itr.remove();
		}
		System.out.println(a);//[0, 2, 4, 6, 8, 10]
	}
}
```

Limitations of Iterator:

1) Both enumeration and Iterator are single direction cursors only. That is we can always move only forward direction and we can't move to the backward direction.
2) While iterating by Iterator we can perform only read and remove operations and we can't perform replacement and addition of new objects.
3) To overcome these limitations sun people introduced listIterator concept.

ListIterator:

1) ListIterator is the child interface of Iterator.
2) By using listIterator we can move either to the forward direction (or) to the backward direction that is it is a bi-directional cursor.
3) While iterating by listIterator we can perform replacement and addition of new objects in addition to read and remove operations

- By using listIterator method we can create listIterator object.

ListIterator itr=l.listiterator();

- ListIterator interface defines the following 9 methods.

1) public boolean hasNext();
2) public Object next(); — forward
3) public int nextIndex();
4) public boolean hasPrevious();
5) public Object previous(); — backward
6) public int previousIndex();
7) public void remove();
8) public void set(Object o);
9) public void add(Object new);

Example:

```
import java.util.*;
class  ListIteratorDemo
{
	public static void main(String[] args)
	{
		LinkedList l=new LinkedList();
		l.add("balakrishna");
		l.add("venki");
		l.add("chiru");
		l.add("nag");
		System.out.println(l);//[balakrishna, venki, chiru, nag]
		ListIterator itr=l.listIterator();
		while(itr.hasNext())
		{
			String s=(String)itr.next();
			if(s.equals("venki"))
			{
				itr.remove();
			}
		}
		System.out.println(l);//[balakrishna, chiru, nag]
	}
}
```

Case 1:

```
if(s.equals("chiru"))
{
itr.set("chran");
}
```

Output:

```
[balakrishna, venki, chiru, nag]
[balakrishna, venki, chran, nag]
```

Case 2:

```
if(s.equals("nag"))
{
itr.add("chitu");
}
```

Output:

```
[balakrishna, venki, chiru, nag]
[balakrishna, venki, chiru, nag, chitu]
```

- The most powerful cursor is listIterator but its limitation is it is applicable only for "List objects".

Compression of Enumeration Iterator and ListIterator?

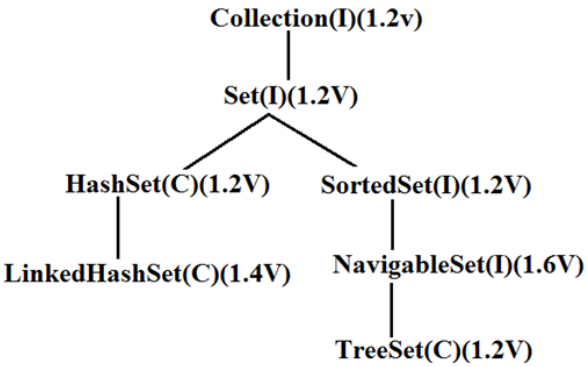
	Property	Enumeration	Iterator	ListIterator
1)	Is it legacy?	yes	no	no
2)	It is applicable for?	Only legacy classes.	Applicable for any	Applicable for only list

			collection object.	objects.
3)	Moment?	Single direction cursor(forward)	Single direction cursor(forward	Bi-directional.
4)	How to get it?	By using elements() method.	By using iterator()method.	By using listIterator() method.
5)	Accessibility?	Only read.	Both read and remove.	Read/remove/replace/add.
6)	methods	hasMoreElement() nextElement()	hasNext() next() remove()	9 methods.

Set interface:

1) It is the child interface of Collection.
2) If we want to represent a group of individual objects where duplicates are not allow and insertion order is not preserved then we should go for Set interface.

Diagram:



- Set interface does not contain any new method we have to use only Collection interface methods.

HashSet:

1) The underlying data structure is Hashtable.
2) Insertion order is not preserved and it is based on hash code of the objects.
3) Duplicate objects are not allowed.
4) If we are trying to insert duplicate objects we won't get compile time error and runtime error add() method simply returns false.
5) Heterogeneous objects are allowed.
6) Null insertion is possible.
7) Implements Serializable and Cloneable interfaces but not RandomAccess.

Constructors:

1) HashSet h=new HashSet();

- Creates an empty HashSet object with default initial capacity 16 and default fill ratio 0.75(fill ratio is also known as load factor).

2) HashSet h=new HashSet(int initialcapacity);

- Creates an empty HashSet object with the specified initial capacity and default fill ratio 0.75.

3) HashSet h=new HashSet(int initialcapacity,float fillratio);

4) HashSet h=new HashSet(Collection c);

Example:

```
import java.util.*;
class HashSetDemo
{
    public static void main(String[] args)
    {
        HashSet h=new HashSet();
        h.add("B");
        h.add("C");
        h.add("D");
        h.add("Z");
        h.add(null);
        h.add(10);
        System.out.println(h.add("Z"));//false
        System.out.println(h);//[null, D, B, C, 10, Z]
        }
}
```

LinkedHashSet:

1) It is the child class of HashSet.
2) LinkedHashSet is exactly same as HashSet except the following differences.

HashSet	LinkedHashSet
1) The underlying data structure is Hashtable.	1) The underlying data structure is a combination of LinkedList and Hashtable.

2)	Insertion order is not preserved.	2)	Insertion order is preserved.
3)	Introduced in 1.2 v.	3)	Introduced in 1.4v.

- In the above program if we are replacing HashSet with LinkedHashSet the output is [B, C, D, Z, null, 10].That is insertion order is preserved.

Example:

```
import java.util.*;
class LinkedHashSetDemo
{
    public static void main(String[] args)
    {
            LinkedHashSet h=new LinkedHashSet();
            h.add("B");
            h.add("C");
            h.add("D");
            h.add("Z");
            h.add(null);
            h.add(10);
            System.out.println(h.add("Z"));//false
            System.out.println(h);//[B, C, D, Z, null, 10]
            }
}
```

Note: LinkedHashSet and LinkedHashMap commonly used for implementing "cache applications" where insertion order must be preserved and duplicates are not allowed.

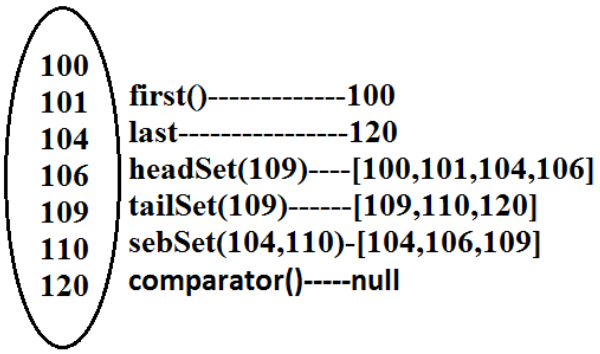
SortedSet:

1) It is child interface of Set.
2) If we want to represent a group of "unique objects" according to some sorting order then we should go for SortedSet interface.
3) That sorting order can be either default natural sorting (or) customized sorting order.

- SortedSet interface define the following 6 specific methods.

1) Object first();
2) Object last();
3) SortedSet headSet(Object o);
- Returns the SortedSet whose elements are <=o.
4) SortedSet tailSet(Object o);
- It returns the SortedSet whose elements are >=o.
5) SortedSet subset(Object o1,Object o2);
- Returns the SortedSet whose elements are >=o1but <o2.
6) Comparator comparator();
- Returns the Comparator object that describes underlying sorting technique.
- If we are following default natural sorting order then this method returns null.

Diagram:



TreeSet:

1) The underlying data structure is balanced tree.
2) Duplicate objects are not allowed.
3) Insertion order is not preserved and it is based on some sorting order of objects.
4) Heterogeneous objects are not allowed if we are trying to insert heterogeneous objects then we will get ClassCastException.
5) Null insertion is possible(only once).

Constructors:

1) TreeSet t=new TreeSet();
- Creates an empty TreeSet object where all elements will be inserted according to default natural sorting order.

2) TreeSet t=new TreeSet(Comparator c);

- Creates an empty TreeSet object where all objects will be inserted according to customized sorting order specified by Comparator object.

3) TreeSet t=new TreeSet(SortedSet s);

4) TreeSet t=new TreeSet(Collection c);

Example 1:

```
import java.util.*;
class TreeSetDemo
{
	public static void main(String[] args)
	{
		TreeSet t=new TreeSet();
		t.add("A");
		t.add("a");
		t.add("B");
		t.add("Z");
		t.add("L");
		//t.add(new Integer(10));//ClassCastException
		//t.add(null);//NullPointerException
		System.out.println(t);//[A, B, L, Z, a]
		}
}
```

Null acceptance:

- For the empty TreeSet as the 1st element "null" insertion is possible but after inserting that null if we are trying to insert any other we will get NullPointerException.
- For the non empty TreeSet if we are trying to insert null then we will get NullPointerException.

Example 2:

```
import java.util.*;
class TreeSetDemo
{
	public static void main(String[] args)
	{
		TreeSet t=new TreeSet();
		t.add(new StringBuffer("A"));
		t.add(new StringBuffer("Z"));
		t.add(new StringBuffer("L"));
		t.add(new StringBuffer("B"));
		System.out.println(t);
		}
}
```

Output:

Runtime Exception.

- Exception in thread "main" java.lang.ClassCastException: java.lang.StringBuffer cannot be cast to java.lang.Comparable
- If we are depending on default natural sorting order compulsory the objects should be homogeneous and Comparable otherwise we will get ClassCastException.
- An object is said to be Comparable if and only if the corresponding class implements Comparable interface.
- String class and all wrapper classes implements Comparable interface but StringBuffer class doesn't implement Comparable interface hence in the above program we are getting ClassCastException.

Comparable interface:

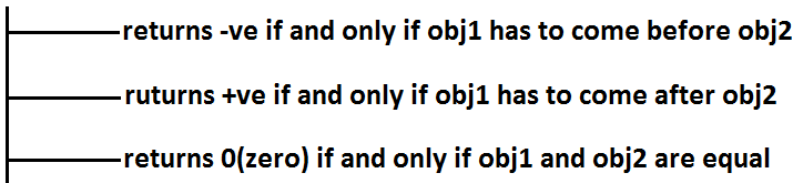
- Comparable interface present in java.lang package and contains only one method compareTo() method.

public int compareTo(Object obj);

Example:

obj1.compareTo(obj2);

Diagram:



Example 3:

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

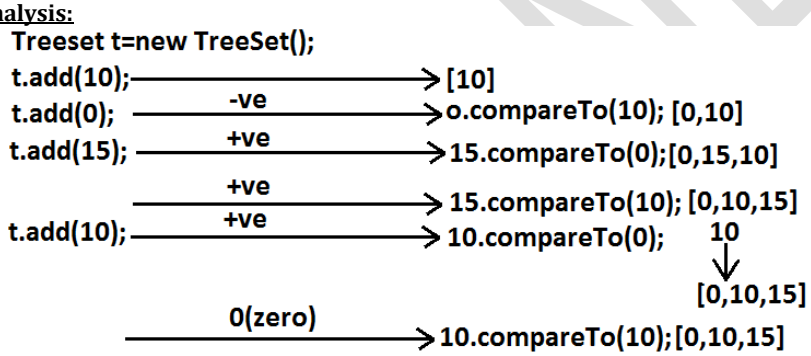
```
                System.out.println("A".compareTo("Z"));//-25
                System.out.println("Z".compareTo("K"));//15
                System.out.println("A".compareTo("A"));//0
                //System.out.println("A".compareTo(new Integer(10)));//Test.java:8: compareTo(java.lang.String) in
java.lang.String cannot be applied to (java.lang.Integer)
                //System.out.println("A".compareTo(null));//NullPointerException
        }
}
```

- If we are depending on default natural sorting order then internally JVM will use compareTo() method to arrange objects in sorting order.

Example 4:

```
import java.util.*;
class Test
{
        public static void main(String[] args)
        {
                TreeSet t=new TreeSet();
                t.add(10);
                t.add(0);
                t.add(15);
                t.add(10);
                System.out.println(t);//[0, 10, 15]
        }
}
```

compareTo() method analysis:



- If we are not satisfying with default natural sorting order (or) if default natural sorting order is not available then we can define our own customized sorting by Comparator object.

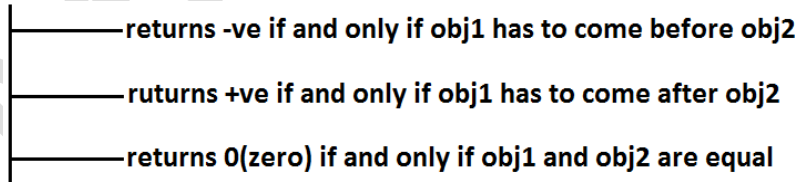
• Comparable meant for default natural sorting order. • Comparator meant for customized sorting order.

Comparator interface:

- Comparator interface present in java.util package this interface defines the following 2 methods.

1) public int compare(Object obj1,Object Obj2);

Diagram:



2) public boolean equals(Objectobj);

- Whenever we are implementing Comparator interface we have to provide implementation only for **compare()** method.
- Implementing equals() method is optional because it is already available from Object class through inheritance.

Requirement: Write a program to insert integer objects into the TreeSet where the sorting order is descending order.

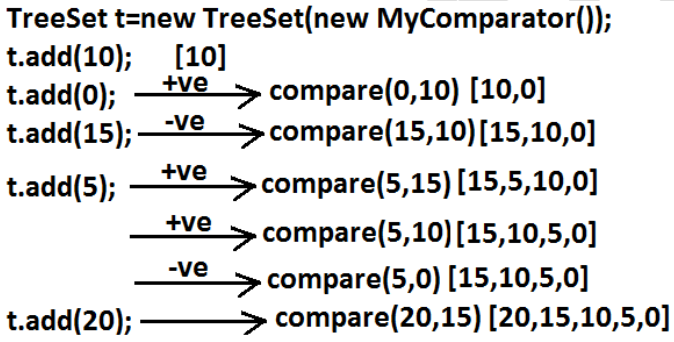
Program:

```
import java.util.*;
class Test
{
        public static void main(String[] args)
        {
                TreeSet t=new TreeSet(new MyComparator());----------(1)
                t.add(10);
                t.add(0);
```

```
                t.add(15);
                t.add(5);
                t.add(20);
                System.out.println(t);//[20, 15, 10, 5, 0]
        }
}
class MyComparator implements Comparator
{
        public int compare(Object obj1,Object obj2)
        {
                Integer i1=(Integer)obj1;
                Integer i2=(Integer)obj2;
                if(i1<i2)
                        return +1;
                else if(i1>i2)
                        return -100;
                else return 0;
        }
}
```

- At line "1" if we are not passing Comparator object then JVM will always calls compareTo() method which is meant for default natural sorting order(ascending order)hence in this case the output is [0, 5, 10, 15, 20].
- At line "1" if we are passing Comparator object then JVM calls compare() method of MyComparator class which is meant for customized sorting order(descending order) hence in this case the output is [20, 15, 10, 5, 0].

Diagram:



Various alternative implementations of compare() method:

```
public int compare(Object obj1,Object obj2)
        {
                Integer i1=(Integer)obj1;
                Integer i2=(Integer)obj2;
                //return i1.compareTo(i2);//[0, 5, 10, 15, 20]
                //return -i1.compareTo(i2);//[20, 15, 10, 5, 0]
                //return i2.compareTo(i1);//[20, 15, 10, 5, 0]
                //return -i2.compareTo(i1);//[0, 5, 10, 15, 20]
                //return -1;//[20, 5, 15, 0, 10]//reverse of insertion order
                //return +1;//[10, 0, 15, 5, 20]//insertion order
                //return 0;//[10]and all the remaining elements treated as duplicate.
        }
```

Requirement: Write a program to insert String objects into the TreeSet where the sorting order is reverse of alphabetical order.

Program:

```
import java.util.*;
class TreeSetDemo
{
        public static void main(String[] args)
        {
                TreeSet t=new TreeSet(new MyComparator());
                t.add("Roja");
                t.add("ShobaRani");
                t.add("RajaKumari");
                t.add("GangaBhavani");
                t.add("Ramulamma");
                System.out.println(t);//[ShobaRani, Roja, Ramulamma, RajaKumari, GangaBhavani]
        }
}
class MyComparator implements Comparator
```

```
{
	public int compare(Object obj1,Object obj2)
	{
		String s1=obj1.toString();
		String s2=(String)obj2;
		//return s2.compareTo(s1);
		return -s1.compareTo(s2);
	}
}
```

Requirement: Write a program to insert StringBuffer objects into the TreeSet where the sorting order is alphabetical order.

Program:

```
import java.util.*;
class TreeSetDemo
{
	public static void main(String[] args)
	{
		TreeSet t=new TreeSet(new MyComparator());
		t.add(new StringBuffer("A"));
		t.add(new StringBuffer("Z"));
		t.add(new StringBuffer("K"));
		t.add(new StringBuffer("L"));
		System.out.println(t);// [A, K, L, Z]
	}
}
class MyComparator implements Comparator
{
	public int compare(Object obj1,Object obj2)
		{
		String s1=obj1.toString();
		String s2=obj2.toString();
		return s1.compareTo(s2);
		}
}
```

Note: Whenever we are defining our own customized sorting by Comparator then the objects need not be Comparable.

Example: StringBuffer

Requirement: Write a program to insert String and StringBuffer objects into the TreeSet where the sorting order is increasing length order. If 2 objects having the same length then consider they alphabetical order.

Program:

```
import java.util.*;
class TreeSetDemo
{
	public static void main(String[] args)
	{
		TreeSet t=new TreeSet(new MyComparator());
		t.add("A");
		t.add(new StringBuffer("ABC"));
		t.add(new StringBuffer("AA"));
		t.add("xx");
		t.add("ABCD");
		t.add("A");
		System.out.println(t);//[A, AA, xx, ABC, ABCD]
	}
}
class MyComparator implements Comparator
{
	public int compare(Object obj1,Object obj2)
		{
		String s1=obj1.toString();
		String s2=obj2.toString();
		int l1=s1.length();
		int l2=s2.length();
		if(l1<l2)
			return -1;
		else if(l1>l2)
			return 1;
		else
			return s1.compareTo(s2);
```

```
		}
}
```

Note: If we are depending on default natural sorting order then the objects should be "homogeneous and comparable" otherwise we will get ClassCastException. If we are defining our own sorting by Comparator then objects "need not be homogeneous and comparable".

Comparable vs Comparator:

- For predefined Comparable classes default natural sorting order is already available if we are not satisfied with default natural sorting order then we can define our own customized sorting order by Comparator.
- For predefined non Comparable classes [like StringBuffer] default natural sorting order is not available we can define our own sorting order by using Comparator object.
- For our own classes [like Customer, Student, and Employee] we can define default natural sorting order by using Comparable interface. The person who is using our class, if he is not satisfied with default natural sorting order then he can define his own sorting order by using Comparator object.

Example:

```
import java.util.*;
class Employee implements Comparable
{
String name;
int eid;
Employee(String name,int eid)
{
	this.name=name;
	this.eid=eid;
}
public String toString()
{
	return name+"----"+eid;
}
public int compareTo(Object o)
{
	int eid1=this.eid;
	int eid2=((Employee)o).eid;
	if(eid1<eid2)
	{
		return -1;
	}
	else if(eid>eid2)
	{
		return 1;
	}
	else return 0;
}
}
class CompComp
{
	public static void main(String[] args)
	{
		Employee e1=new Employee("nag",100);
		Employee e2=new Employee("balaiah",200);
		Employee e3=new Employee("chiru",50);
		Employee e4=new Employee("venki",150);
		Employee e5=new Employee("nag",100);
		TreeSet t1=new TreeSet();
		t1.add(e1);
		t1.add(e2);
		t1.add(e3);
		t1.add(e4);
		t1.add(e5);
		System.out.println(t1);//[chiru----50, nag----100, venki----150, balaiah----200]
		TreeSet t2=new TreeSet(new MyComparator());
		t2.add(e1);
		t2.add(e2);
		t2.add(e3);
		t2.add(e4);
		t2.add(e5);
		System.out.println(t2);//[balaiah----200, chiru----50, nag----100, venki----150]
```

```
        }
    }
    class MyComparator implements Comparator
    {
        public int compare(Object obj1,Object obj2)
        {
                Employee e1=(Employee)obj1;
                Employee e2=(Employee)obj2;
                String s1=e1.name;
                String s2=e2.name;
                return s1.compareTo(s2);
                }
    }
```

Compression of Comparable and Comparator?

Comparable	Comparator
1) Comparable meant for default natural sorting order.	1) Comparator meant for customized sorting order.
2) Present in java.lang package.	2) Present in java.util package.
3) Contains only one method. compareTo() method.	3) Contains 2 methods. Compare() method. Equals() method.
4) String class and all wrapper Classes implements Comparable interface.	4) No predefined class implements Comparator.

Compression of Set implemented class objects:

Property	HashSet	LinkedHashSet	TreeSet
1) Underlying Data structure.	Hashtable.	LinkedList +Hashtable.	Balanced Tree.
2) Insertion order.	Not preserved.	Preserved.	Not preserved (by default).
3) Duplicate objects.	Not allowed.	Not allowed.	Not allowed.
4) Sorting order.	Not applicable.	Not applicable.	Applicable.
5) Heterogeneous objects.	Allowed.	Allowed.	Not allowed.
6) Null insertion.	Allowed.	Allowed.	For the empty TreeSet as the 1st element null insertion is possible in all other cases we will get NPE.

Map:

1) If we want to represent a group of objects as "key-value" pair then we should go for Map interface.
2) Both key and value are objects only.
3) Duplicate keys are not allowed but values can be duplicated
4) Each key-value pair is called "one entry".

Diagram:

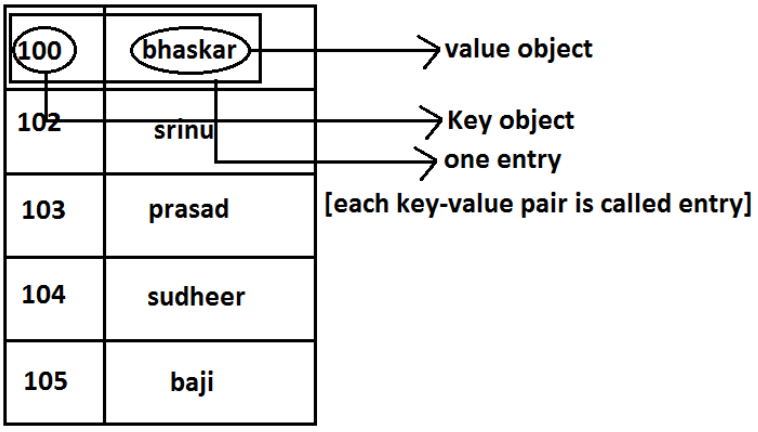
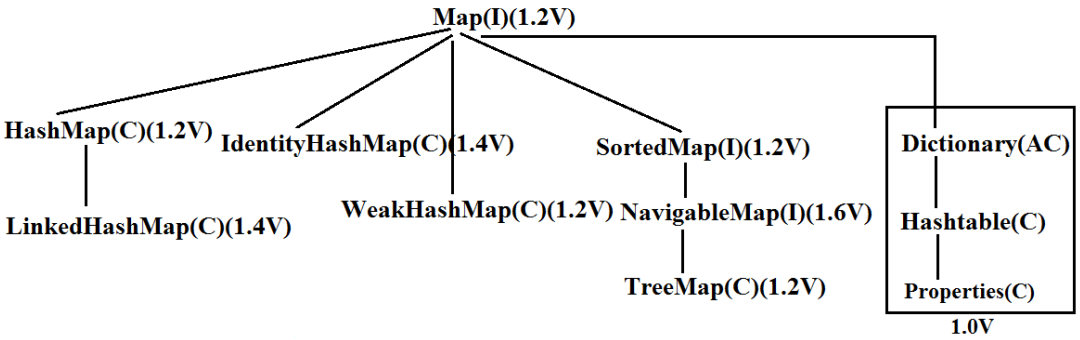


Diagram:



- Map interface is not child interface of Collection and hence we can't apply Collection interface methods here.
- Map interface defines the following specific methods.

1) Object put(Object key,Object value);

- To add an entry to the Map, if key is already available then the old value replaced with new value and old value will be returned.

Example:

```
import java.util.*;
class Map
{
    public static void main(String[] args)
    {
        HashMap m=new HashMap();
        m.put("100","vijay");
        System.out.println(m);//{100=vijay}
        m.put("100","bhaskar");
        System.out.println(m);//{100=bhaskar}
    }
}
```

2) void putAll(Map m);
3) Object get(Object key);
4) Object remove(Object key);

- It removes the entry associated with specified key and returns the corresponding value.

5) boolean containsKey(Object key);
6) boolean containsValue(Object value);
7) boolean isEmpty();
8) Int size();
9) void clear();
10) Set keySet();

- The set of keys we are getting.

11) Collection values();

- The set of values we are getting.

12) Set entrySet();

- The set of entryset we are getting.

Entry interface:

- Each key-value pair is called one entry without existing Map object there is no chance of existing entry object hence interface entry is define inside Map interface(inner interface).

Example:

```
interface Map
{
    .....;
    .....;
    .....;
    interface Entry
    {
        Object getKey();
        Object getValue();          on Entry we can apply these 3 methods.
        Object setValue(Object new);
    }
}
```

HashMap:

1) The underlying data structure is Hashtable.
2) Duplicate keys are not allowed but values can be duplicated.
3) Insertion order is not preserved and it is based on hash code of the keys.
4) Heterogeneous objects are allowed for both key and value.
5) Null is allowed for keys(only once) and for values(any number).

Differences between HashMap and Hashtable?

	HashMap		Hashtable
1)	No method is synchronized.	1)	Every method is synchronized.
2)	Multiple Threads can operate simultaneously on HashMap object and hence it is not Thread safe.	2)	Multiple Threads can't operate simultaneously on Hashtable object and hence Hashtable object is Thread safe.
3)	Relatively performance is high.	3)	Relatively performance is low.
4)	Null is allowed for both key and value.	4)	Null is not allowed for both key and value otherwise we will get NullPointerException.
5)	It is non legacy and introduced in 1.2v.	5)	It is legacy and introduced in 1.0v.

How to get synchronized version of HashMap:

- By default HashMap object is not synchronized. But we can get synchronized version by using the following method of Collections class.

public static Map synchronizedMap(Map m1)

Constructors:

1) HashMap m=new HashMap();

- Creates an empty HashMap object with default initial capacity 16 and default fill ratio "075".

2) HashMap m=new HashMap(int initialcapacity);

3) HashMap m =new HashMap(int initialcapacity, float fillratio);

4) HashMap m=new HashMap(Map m);

Example:

```
import java.util.*;
class HashMapDemo
{
    public static void main(String[] args)
    {
        HashMap m=new HashMap();
        m.put("chiranjeevi",700);
        m.put("balaiah",800);
        m.put("venkatesh",200);
        m.put("nagarjuna",500);
        System.out.println(m);//{nagarjuna=500, venkatesh=200, balaiah=800, chiranjeevi=700}
        System.out.println(m.put("chiranjeevi",100));//700
        Set s=m.keySet();
        System.out.println(s);//[nagarjuna, venkatesh, balaiah, chiranjeevi]
        Collection c=m.values();
        System.out.println(c);//[500, 200, 800, 100]
        Set s1=m.entrySet();
        System.out.println(s1);//[nagarjuna=500, venkatesh=200, balaiah=800, chiranjeevi=100]
        Iterator itr=s1.iterator();
        while(itr.hasNext())
        {
            Map.Entry m1=(Map.Entry)itr.next();
            System.out.println(m1.getKey()+"....."+m1.getValue());//nagarjuna.....500
                                                                  //venkatesh.....200
                                                    //balaiah.....800
                                                //chiranjeevi.....100
            if(m1.getKey().equals("nagarjuna"))
            {
                m1.setValue(1000);
            }
        }
        System.out.println(m);//{nagarjuna=1000, venkatesh=200, balaiah=800, chiranjeevi=100}
    }
}
```

LinkedHashMap:

	HashMap		LinkedHashMap
1)	The underlying data structure is Hashtable.	1)	The underlying data structure is a combination of Hashtable+ LinkedList.
2)	Insertion order is not preserved.	2)	Insertion order is preserved.
3)	introduced in 1.2.v.	3)	Introduced in 1.4v.

Note: in the above program if we are replacing HashMap with LinkedHashMap then the output is {chiranjeevi=100, balaiah.....800, venkatesh.....200, nagarjuna.....1000} that is insertion order is preserved.

Note: in general we can use LinkedHashSet and LinkedHashMap for implementing cache applications.

IdentityHashMap:

1) It is exactly same as HashMap except the following differences.
2) In the case of HashMap JVM will always use ".equals()"method to identify duplicate keys.
3) But in the case of IdentityHashMap JVM will use== (double equal operator) to identify duplicate keys.

Example:

```
import java.util.*;
class HashMapDemo
{
	public static void main(String[] args)
	{
		HashMap m=new HashMap();
		Integer i1=new Integer(10);
		Integer i2=new Integer(10);
		m.put(i1,"pavan");
		m.put(i2,"kalyan");
		System.out.println(m);
	}
}
```

- In the above program i1 and i2 are duplicate keys because i1.equals(i2) returns true.
- In the above program if we replace HashMap with IdentityHashMap then i1 and i2 are not duplicate keys because i1==i2 is false hence in this case the output is {10=pavan, 10=kalyan}.

```
System.out.println(m.get(10));//null
10==i1------false
10==i2------false
```

WeakHashMap:

- It is exactly same as HashMap except the following differences.
- In the case of normal HashMap, an object is not eligible for GC even though it doesn't have any references if it is associated with HashMap. That is HashMap dominates garbage collector.
- But in the case of WeakHashMap if an object does not have any references then it's always eligible for GC even though it is associated with WeakHashMap that is garbage collector dominates WeakHashMap.

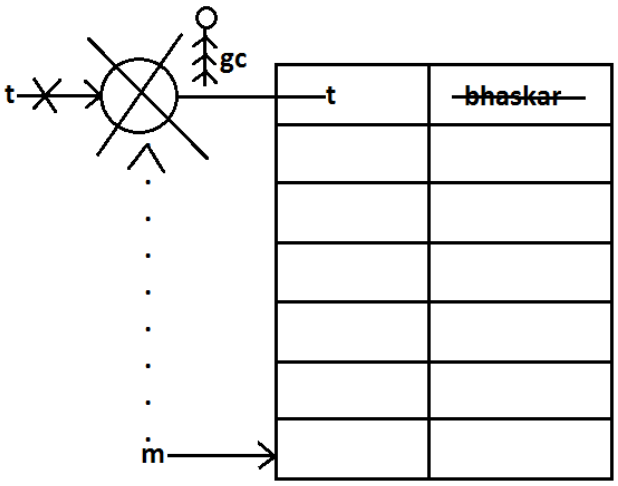
Example:

```
import java.util.*;
class WeakHashMapDemo
{
	public static void main(String[] args)throws Exception
	{
		WeakHashMap m=new WeakHashMap();
		Temp t=new Temp();
		m.put(t,"bhaskar");
		System.out.println(m);//{Temp=bhaskar}
		t=null;
		System.gc();
		Thread.sleep(5000);
		System.out.println(m);//{}
	}
}
class Temp
{
	public String toString()
	{
		return "Temp";
	}
	public void finalize()
	{
		System.out.println("finalize() method called");
	}
}
```

Output:

```
{Temp=bhaskar}
finalize() method called
{}
```

Diagram:



- In the above program if we replace WeakHashMap with normal HashMap then object won't be destroyed by the garbage collector in this the output is
  {Temp=bhaskar}
  {Temp=bhaskar}

SortedMap:

- It is the child interface of Map.
- If we want to represent a group of key-value pairs according to some sorting order of keys then we should go for SortedMap.
- Sorting is possible only based on the keys but not based on values.
- SortedMap interface defines the following 6 specific methods.

1) Object firsyKey();
2) Object lastKey();
3) SortedMap headMap(Object key);
4) SortedMap tailMap(Object key);
5) SortedMap subMap(Object key1,Object key2);
6) Comparator comparator();

TreeMap:

1) The underlying data structure is RED-BLACK Tree.
2) Duplicate keys are not allowed but values can be duplicated.
3) Insertion order is not preserved and all entries will be inserted according to some sorting order of keys.
4) If we are depending on default natural sorting order keys should be homogeneous and Comparable otherwise we will get ClassCastException.
5) If we are defining our own sorting order by Comparator then keys can be heterogeneous and non Comparable.
6) There are no restrictions on values they can be heterogeneous and non Comparable.
7) For the empty TreeMap as first entry null key is allowed but after inserting that entry if we are trying to insert any other entry we will get NullPointerException.
8) For the non empty TreeMap if we are trying to insert an entry with null key we will get NullPointerException.
9) There are no restrictions for null values.

Constructors:

1) TreeMap t=new TreeMap();
- For default natural sorting order.

2) TreeMap t=new TreeMap(Comparator c);
- For customized sorting order.

3) TreeMap t=new TreeMap(SortedMap m);
4) TreeMap t=new TreeMap(Map m);

Example 1:

```
import java.util.*;
class TreeMapDemo
{
    public static void main(String[] args)
    {
            TreeMap t=new TreeMap();
            t.put(100,"ZZZ");
            t.put(103,"YYY");
            t.put(101,"XXX");
            t.put(104,106);
            t.put(107,null);
            //t.put("FFF","XXX");//ClassCastException
            //t.put(null,"xxx");//NullPointerException
            System.out.println(t);//{100=ZZZ, 101=XXX, 103=YYY, 104=106, 107=null}
```

```
        }
    }
```

Example 2:

```
import java.util.*;
class TreeMapDemo
{
    public static void main(String[] args)
    {
        TreeMap t=new TreeMap(new MyComparator());
        t.put("XXX",10);
        t.put("AAA",20);
        t.put("ZZZ",30);
        t.put("LLL",40);
        System.out.println(t);//{ZZZ=30, XXX=10, LLL=40, AAA=20}
    }
}
class MyComparator implements Comparator
{
    public int compare(Object obj1,Object obj2)
    {
        String s1=obj1.toString();
        String s2=obj2.toString();
        return s2.compareTo(s1);
    }
}
```

Hashtable:

1) The underlying data structure is Hashtable.
2) Insertion order is not preserved and it is based on hash code of the keys.
3) Heterogeneous objects are allowed for both keys and values.
4) Null key (or) null value is not allowed otherwise we will get NullPointerException.
5) Duplicate keys are allowed but values can be duplicated.

Constructors:

1) Hashtable h=new Hashtable();

- Creates an empty Hashtable object with default initialcapacity 11 and default fill ratio 0.75.

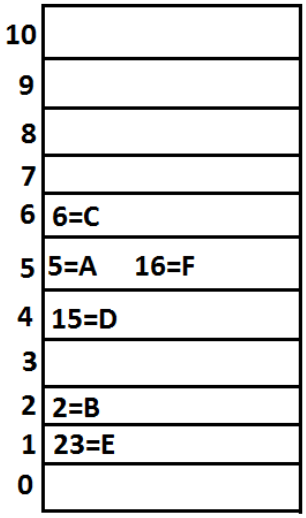
2) Hashtable h=new Hashtable(int initialcapacity);
3) Hashtable h=new Hashtable(int initialcapacity,float fillratio);
4) Hashtable h=new Hashtable (Map m);

Example:

```
import java.util.*;
class HashtableDemo
{
    public static void main(String[] args)
    {
        Hashtable h=new Hashtable();
        h.put(new Temp(5),"A");
        h.put(new Temp(2),"B");
        h.put(new Temp(6),"C");
        h.put(new Temp(15),"D");
        h.put(new Temp(23),"E");
        h.put(new Temp(16),"F");
        System.out.println(h);//{6=C, 16=F, 5=A, 15=D, 2=B, 23=E}
    }
}
class Temp
{
    int i;
    Temp(int i)
    {
        this.i=i;
    }
    public int hashCode()
    {
        return i;
    }
    public String toString()
    {
        return i+"";
```

```
    }
}
```

Diagram:

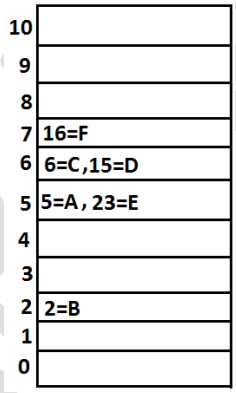


Note: if we change hasCode() method of Temp class as follows.

```
public int hashCode()
    {
        return i%9;
    }
```

- Then the output is {16=F, 15=D, 6=C, 23=E, 5=A, 2=B}.

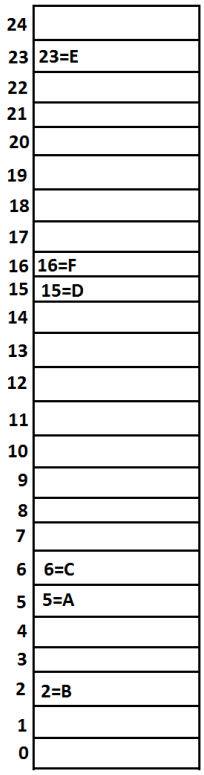
Diagram:



Note: if we change initial capacity as 25.

Hashtable h=new Hashtable(25);

Diagram:



Properties:

1) Properties class is the child class of Hashtable.
2) If anything which changes frequently such type of values not recommended to hardcode in java application because for every change we have to recompile, rebuild and redeployed the application and even server restart also required sometimes it creates a big business impact to the client.
3) Such type of variable things we have to hardcode in property files and we have to read the values from the property files.
4) The main advantage in this approach is if there is any change in property files automatically those changes will be available to java application just redeployment is enough.
5) By using Properties object we can read and hold properties from property files into java application.

Constructor:

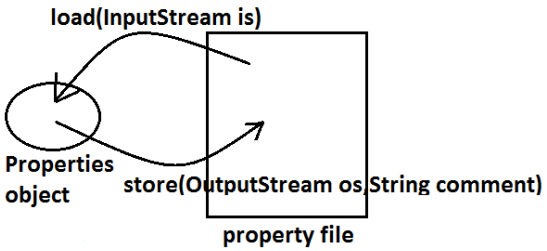
Properties p=new Properties();

- In properties both key and value "should be String type only".

Methods:

1) String getPrperty(String propertyname) ;
- Returns the value associated with specified property.
2) String setproperty(String propertyname,String propertyvalue);
- To set a new property.
3) Enumeration propertyNames();
4) void load(InputStream is);//Any InputStream we can pass.
- To load Properties from property files into java Properties object.
5) void store(OutputStream os,String comment);//Any OutputStream we can pass.
- To store the properties from Properties object into properties file.

Diagram:



Example:

```
import java.util.*;
import java.io.*;
class PropertiesDemo
{
    public static void main(String[] args)throws Exception
    {
```

```
                Properties p=new Properties();
                FileInputStream fis=new FileInputStream("abc.properties");
                p.load(fis);
                System.out.println(p);//{user=scott, password=tiger, venki=8888}
                String s=p.getProperty("venki");
                System.out.println(s);//8888
                p.setProperty("nag","9999999");
                Enumeration e=p.propertyNames();
                while(e.hasMoreElements())
                {
                        String s1=(String)e.nextElement();
                        System.out.println(s1);//nag
                                                //user
                                                //password
                                                //venki
                }
                FileOutputStream fos=new FileOutputStream("abc.properties");
                p.store(fos,"updated by bhaskar for scjp demo class");
        }
}
```

Property file:

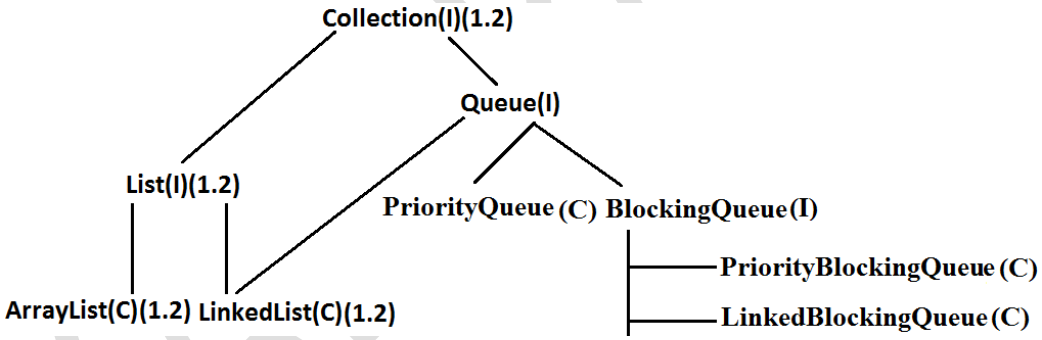
```
user=scott
nag=9999999
password=tiger
venki=8888
```

abc.properties

1.5 enhancements (Queue interface)

1) If we want to represent a group of individual objects prior (happening before something else) to processing then we should go for Queue interface.

Diagram:



2) Usually Queue follows **first in first out order** but based on our requirement we can implement our own order also.
3) From 1.5v onwards LinkedList also implements Queue interface.
4) LinkedList based implementation of Queue always follows first in first out order.

Queue interface methods:

1) boolean after(Object o);
- To add an object to the Queue.

2) Object poll() ;
- To remove and return head element of the Queue, if Queue is empty then we will get null.

3) Object remove();
- To remove and return head element of the Queue. If Queue is empty then this method raises Runtime Exception saying NoSuchElementException.

4) Object peek();
- To return head element of the Queue without removal, if Queue is empty this method returns null.

5) Object element();
- It returns head element of the Queue and if Queue is empty then it will raise Runtime Exception saying NoSuchElementException.

PriorityQueue:

1) We can use PriorityQueue to represent a group of individual objects prior to processing according to some priority.
2) The priority order can be either default natural sorting order (or) customized sorting order specified by Comparator object.
3) If we are depending on default natural sorting order then the objects must be homogeneous and Comparable otherwise we will get ClassCastException.
4) If we are defining our own customized sorting order by Comparator then the objects need not be homogeneous and Comparable.
5) Duplicate objects are not allowed.
6) Insertion order is not preserved but all objects will be inserted according to some priority.

7) Null is not allowed even as the 1st element for empty PriorityQueue.

Constructors:

1) PriorityQueue q=new PriorityQueue();
- Creates an empty PriorityQueue with default initial capacity 11 and default natural sorting order.

2) PriorityQueue q=new PriorityQueue(int initialcapacity,Comparator c);
3) PriorityQueue q=new PriorityQueue(int initialcapacity);
4) PriorityQueue q=new PriorityQueue(Collection c);
5) PriorityQueue q=new PriorityQueue(SortedSet s);

Example 1:

```
import java.util.*;
class PriorityQueueDemo
{
	public static void main(String[] args)
	{
		PriorityQueue q=new PriorityQueue();
		//System.out.println(q.peek());//null
		//System.out.println(q.element());//NoSuchElementException
		for(int i=0;i<=10;i++)
		{
			q.offer(i);
		}
		System.out.println(q);//[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
		System.out.println(q.poll());//0
		System.out.println(q);//[1, 3, 2, 7, 4, 5, 6, 10, 8, 9]
		}
}
```

Note: Some platforms may not provide proper supports for PriorityQueue [windowsXP].

Example 2:

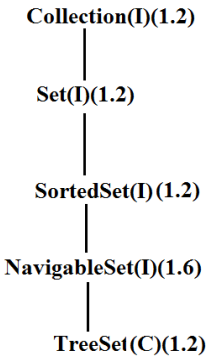
```
import java.util.*;
class PriorityQueueDemo
{
	public static void main(String[] args)
	{
		PriorityQueue q=new PriorityQueue(15,new MyComparator());
		q.offer("A");
		q.offer("Z");
		q.offer("L");
		q.offer("B");
		System.out.println(q);//[Z, B, L, A]
		}
}
class MyComparator implements Comparator
{
	public int compare(Object obj1,Object obj2)
	{
		String s1=(String)obj1;
		String s2=obj2.toString();
		return s2.compareTo(s1);
	}
}
```

1.6v Enhancements (NavigableSet and NavigableMap)

NavigableSet:

1) It is the child interface of SortedSet.
2) It provides several methods for navigation purposes.

Diagram:



- NavigableSet interface defines the following methods.

1) ceiling(e);
- It returns the lowest element which is >=e.

2) higher(e);
- It returns the lowest element which is >e.

3) floor(e);
- It returns highest element which is <=e.

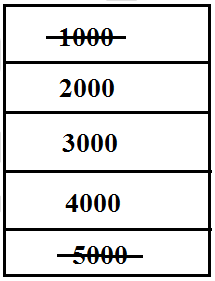
4) lower(e);
- It returns height element which is <e.

5) pollFirst ();
- Remove and return 1st element.

6) pollLast ();
- Remove and return last element.

7) descendingSet ();
- Returns SortedSet in reverse order.

Diagram:



Example:

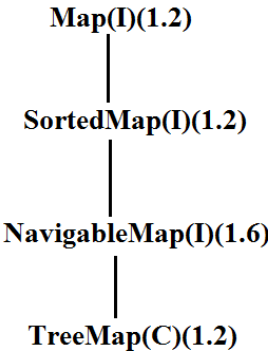
```
import java.util.*;
class NavigableSetDemo
{
	public static void main(String[] args)
	{
		TreeSet<Integer> t=new TreeSet<Integer>();
		t.add(1000);
		t.add(2000);
		t.add(3000);
		t.add(4000);
		t.add(5000);
		System.out.println(t);//[1000, 2000, 3000, 4000, 5000]
		System.out.println(t.ceiling(2000));//2000
		System.out.println(t.higher(2000));//3000
		System.out.println(t.floor(3000));//3000
		System.out.println(t.lower(3000));//2000
		System.out.println(t.pollFirst());//1000
		System.out.println(t.pollLast());//5000
		System.out.println(t.descendingSet());//[4000, 3000, 2000]
		System.out.println(t);//[2000, 3000, 4000]
		}
```

}

NavigableMap:

- It is the child interface of SortedMap and it defines several methods for navigation purpose.

Diagram:



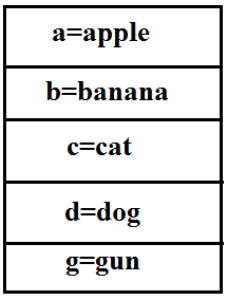
- NavigableMap interface defines the following methods.

1) ceilingKey(e);
2) higherKey(e);
3) floorKey(e);
4) lowerKey(e);
5) pollFirstEntry();
6) pollLastEntry();
7) descendingMap();

Example:

```
import java.util.*;
class NavigableMapDemo
{
	public static void main(String[] args)
	{
		TreeMap<String,String> t=new TreeMap<String,String>();
		t.put("b","banana");
		t.put("c","cat");
		t.put("a","apple");
		t.put("d","dog");
		t.put("g","gun");
		System.out.println(t);//{a=apple, b=banana, c=cat, d=dog, g=gun}
		System.out.println(t.ceilingKey("c"));//c
		System.out.println(t.higherKey("e"));//g
		System.out.println(t.floorKey("e"));//d
		System.out.println(t.lowerKey("e"));//d
		System.out.println(t.pollFirstEntry());//a=apple
		System.out.println(t.pollLastEntry());//g=gun
		System.out.println(t.descendingMap());//{d=dog, c=cat, b=banana}
		System.out.println(t);//{b=banana, c=cat, d=dog}
		}
}
```

Diagram:



Collections class:

- Collections class defines several utility methods for collection objects.

Sorting the elements of a List:

- Collections class defines the following methods to perform sorting the elements of a List.

public static void sort(List l);

- To sort the elements of List according to default natural sorting order in this case the elements should be homogeneous and comparable otherwise we will get ClassCastException.
- The List should not contain null otherwise we will get NullPointerException.

public static void sort(List l,Comparator c);

- To sort the elements of List according to customized sorting order.

Program 1: To sort elements of List according to natural sorting order.

```
import java.util.*;
class CollectionsDemo
{
	public static void main(String[] args)
	{
		ArrayList l=new ArrayList();
		l.add("Z");
		l.add("A");
		l.add("K");
		l.add("N");
		//l.add(new Integer(10));//ClassCastException
		//l.add(null);//NullPointerException
		System.out.println("Before sorting :"+l);//[Z, A, K, N]
		Collections.sort(l);
		System.out.println("After sorting :"+l);//[A, K, N, Z]
	}
}
```

Program 2: To sort elements of List according to customized sorting order.

```
import java.util.*;
class CollectionsDemo
{
	public static void main(String[] args)
	{
		ArrayList l=new ArrayList();
		l.add("Z");
		l.add("A");
		l.add("K");
		l.add("L");
		l.add(new Integer(10));
		//l.add(null);//NullPointerException
		System.out.println("Before sorting :"+l);//[Z, A, K, L, 10]
		Collections.sort(l,new MyComparator());
		System.out.println("After sorting :"+l);//[Z, L, K, A, 10]
		}
}
class MyComparator implements Comparator
{
	public int compare(Object obj1,Object obj2)
	{
		String s1=(String)obj1;
		String s2=obj2.toString();
		return s2.compareTo(s1);
	}
}
```

Searching the elements of a List:

- Collections class defines the following methods to search the elements of a List.

public static int binarySearch(List l,Object obj);

- If the List is sorted according to default natural sorting order then we have to use this method.

public static int binarySearch(List l,Object obj,Comparator c);

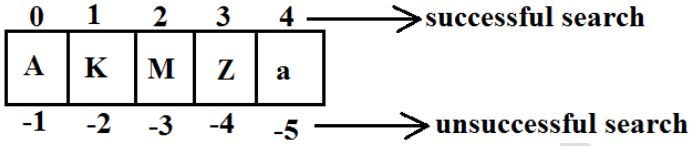
- If the List is sorted according to Comparator then we have to use this method.

Program 1: To search elements of List.

```
import java.util.*;
class CollectionsSearchDemo
{
	public static void main(String[] args)
```

```
        {
                ArrayList l=new ArrayList();
                l.add("Z");
                l.add("A");
                l.add("M");
                l.add("K");
                l.add("a");
                System.out.println(l);//[Z, A, M, K, a]
                Collections.sort(l);
                System.out.println(l);//[A, K, M, Z, a]
                System.out.println(Collections.binarySearch(l,"Z"));//3
                System.out.println(Collections.binarySearch(l,"J"));//-2
                }
}
```

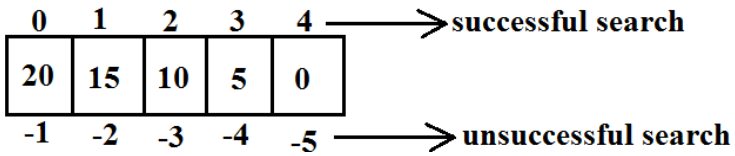
Diagram:



Program 2:

```
import java.util.*;
class CollectionsSearchDemo
{
        public static void main(String[] args)
        {
                ArrayList l=new ArrayList();
                l.add(15);
                l.add(0);
                l.add(20);
                l.add(10);
                l.add(5);
                System.out.println(l);//[15, 0, 20, 10, 5]
                Collections.sort(l,new MyComparator());
                System.out.println(l);//[20, 15, 10, 5, 0]
                System.out.println(Collections.binarySearch(l,10,new MyComparator()));//2
                System.out.println(Collections.binarySearch(l,13,new MyComparator()));//-3
                System.out.println(Collections.binarySearch(l,17));//-6
                }
}
class MyComparator implements Comparator
{
        public int compare(Object obj1,Object obj2)
        {
                Integer i1=(Integer)obj1;
                Integer i2=(Integer)obj2;
                return i2.compareTo(i1);
        }
}
```

Diagram:



Conclusions:

1) Internally these search methods will use binary search algorithm.
2) Successful search returns **index** unsuccessful search returns **insertion point**.
3) Insertion point is the location where we can place the element in the sorted list.
4) Before calling binarySearch() method compulsory the list should be sorted otherwise we will get unpredictable results.
5) If the list is sorted according to Comparator then at the time of search operation also we should pass the same Comparator object otherwise we will get unpredictable results.

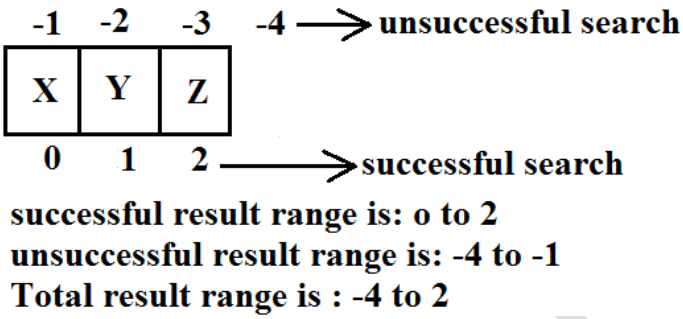
Note:

- For the list of n elements with respect to binary Search() method.

Successful search range is: 0 to n-1. Unsuccessful search results range is: -(n+1)to -1.

Total result range is: -(n+1)to n-1.

Example:



Reversing the elements of List:

public static void reverse(List l);

reverse() vs reverseOrder() method

- We can use reverse() method to reverse the elements of List.
- Where as we can use reverseOrder() method to get reversed Comparator.

Comparator c1=Collections.reverseOrder(Comparator c);

↓ **Descending order** ↓ **Ascending order**

Program: To reverse elements of list.

```
import java.util.*;
class CollectionsReverseDemo
{
    public static void main(String[] args)
    {
        ArrayList l=new ArrayList();
        l.add(15);
        l.add(0);
        l.add(20);
        l.add(10);
        l.add(5);
        System.out.println(l);//[15, 0, 20, 10, 5]
        Collections.reverse(l);
        System.out.println(l);//[5, 10, 20, 0, 15]
    }
}
```

Arrays class:

- Arrays class defines several utility methods for arrays.

Sorting the elements of array:

public static void sort(primitive[] p);//any primitive data type we can give

- To sort the elements of primitive array according to default natural sorting order.

public static void sort(object[] o);

- To sort the elements of object[] array according to default natural sorting order.
- In this case objects should be homogeneous and Comparable.

public static void sort(object[] o,Comparator c);

- To sort the elements of object[] array according to customized sorting order.

Note: We can sort object[] array either by default natural sorting order (or) customized sorting order but we can sort primitive arrays only by default natural sorting order.

Program: To sort elements of array.

```
import java.util.*;
class ArraySortDemo
{
    public static void main(String[] args)
    {
```

```
                int[] a={10,5,20,11,6};
                System.out.println("primitive array before sorting");
                for(int a1:a)
                {
                        System.out.println(a1);
                }
                Arrays.sort(a);
                System.out.println("primitive array after sorting");
                for(int a1: a)
                {
                        System.out.println(a1);
                }
                String[] s={"A","Z","B"};
                System.out.println("Object array before sorting");
                for(String s1: s)
                {
                        System.out.println(s1);
                }
                Arrays.sort(s);
                System.out.println("Object array after sorting");
                for(String s1:s)
                {
                        System.out.println(s1);
                }
                Arrays.sort(s,new MyComparator());
                System.out.println("Object array after sorting by Comparator:");
                for(String s1: s)
                {
                        System.out.println(s1);
                }
        }
}
class MyComparator implements Comparator
{
        public int compare(Object obj1,Object obj2)
        {
                String s1=obj1.toString();
                String s2=obj2.toString();
                return s2.compareTo(s1);
        }
}
```

Searching the elements of array:

- Arrays class defines the following methods to search elements of array.

1) **public static int binarySearch(primitive[] p,primitive key);**
2) **public static int binarySearch(Object[] p, object key);**
3) **public static int binarySearch(Object[] p,Object key,Comparator c);**

- All rules of Arrays class binarySearch() method are exactly same as Collections class binarySearch() method.

Program: To search elements of array.

```
import java.util.*;
class ArraysSearchDemo
{
        public static void main(String[] args)
        {
                int[] a={10,5,20,11,6};
                Arrays.sort(a);
                System.out.println(Arrays.binarySearch(a,6));//1
                System.out.println(Arrays.binarySearch(a,14));//-5
                String[] s={"A","Z","B"};
                Arrays.sort(s);
                System.out.println(Arrays.binarySearch(s,"Z"));//2
                System.out.println(Arrays.binarySearch(s,"S"));//-3
                Arrays.sort(s,new MyComparator());
                System.out.println(Arrays.binarySearch(s,"Z",new MyComparator()));//0
                System.out.println(Arrays.binarySearch(s,"S",new MyComparator()));//-2
                System.out.println(Arrays.binarySearch(s,"N"));//-4(unpredictable result)
                }
```

}

Converting array to List:

- Arrays class defines the following method to view array as List.

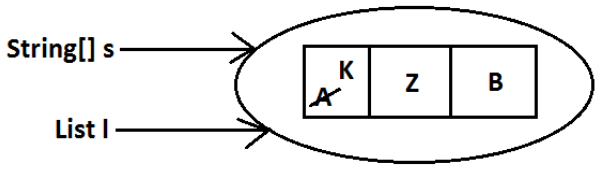
public static List asList (Object[] o);

- Strictly speaking we are not creating an independent List object just we are viewing array in List form.
- By using List reference if we are performing any change automatically these changes will be reflected to array reference similarly by using array reference if we are performing any change automatically these changes will be reflected to the List reference.
- By using List reference if we are trying to perform any operation which varies the size then we will get runtime exception saying **UnsupportedOperationException**.
- By using List reference if we are trying to insert heterogeneous objects we will get runtime exception saying **ArrayStoreException**.

Program: To view array in List form.

```
import java.util.*;
class ArraysAsListDemo
{
	public static void main(String[] args)
	{
		String[] s={"A","Z","B"};
		List l=Arrays.asList(s);
		System.out.println(l);//[A, Z, B]
		s[0]="K";
		System.out.println(l);//[K, Z, B]
		l.set(1,"L");
		for(String s1: s)
		System.out.println(s1);//K,L,B
		//l.add("bhaskar");//UnsupportedOperationException
		//l.remove(2);//UnsupportedOperationException
		//l.set(1,new Integer(10));//ArrayStoreException
		}
}
```

Diagram:



Generics

Agenda:

1) **Introduction**
2) **Generic Classes**
3) **Bounded Types**
4) **Generic methods and wild card character**
5) **Communication with non generic code**
6) **Conclusions**

Introduction:

Case 1:

- Arrays are always type safe that is we can provide the guarantee for the type of elements present inside array.
- For example if our programming requirement is to hold String type of objects it is recommended to use String array. In the case of string array we can add only string type of objects by mistake if we are trying to add any other type we will get compile time error.

Example:

```
String[] s=new String[600];
s[0]="durga";
s[1]="pavan";
s[2]=new Integer(10);(invalid)  --C.E-->  E:\SCJP>javac Test.java
                                          Test.java:8: incompatible types
                                          found   : java.lang.Integer
                                          required: java.lang.String
```

- That is we can always provide guarantee for the type of elements present inside array and hence arrays are safe to use with respect to type that is arrays are type safe.
- But collections are not type safe that is we can't provide any guarantee for the type of elements present inside collection.
- For example if our programming requirement is to hold only string type of objects it is never recommended to go for ArrayList.
- By mistake if we are trying to add any other type we won't get any compile time error but the program may fail at runtime.

Example:

```
ArrayList l=new ArrayList();
l.add("vijaya");
l.add("bhaskara");
l.add(new Integer(10));
.
.
.
String name1=(String)l.get(0);
String name2=(String)l.get(1);
String name3=(String)l.get(2);(invalid)  --R.E-->  Exception in thread "main" java.lang.ClassCastException:
                                                   java.lang.Integer cannot be cast to java.lang.String
```

- Hence we can't provide guarantee for the type of elements present inside collections **that is collections are not safe to use with respect to type**.

Case 2: In the case of array at the time of retrieval it is not required to perform any type casting.

Example:

```
String[] s=new String[600];
s[0]="vijaya";
s[1]="bhaskara";
----------------------
----------------------
----------------------
String name1=s[0];  --->  At the time of retrieval
                          type casting is not required.
```

- But in the case of collection at the time of retrieval compulsory we should perform type casting otherwise we will get compile time error.

Example:

```
ArrayList l=new ArrayList();
l.add("vijaya");
l.add("bhaskara");
String name1= l.get(0);
```

C.E →
```
Test.java:9: incompatible types
found   : java.lang.Object
required: java.lang.String
          String name1=l.get(0);
```

```
At the time of retrieval type
casting is Mandatory.
String name1=(String)l.get(0);
```

- That is in collections type casting is bigger headache.
- To overcome the above problems of collections(type-safety, type casting)sun people introduced generics concept in 1.5v hence the main objectives of generics are:

1) To provide type safety to the collections.
2) To resolve type casting problems.

- To hold only string type of objects we can create a generic version of ArrayList as follows.

Example:

```
ArrayList<String> l=new ArrayList<String>();
l.add("vijaya");
l.add(10);(invalid)
```

C.E →
```
Test.java:8: cannot find symbol
symbol  : method add(int)
location: class java.util.ArrayList<java.lang.String>
          l.add(10);
```

- For this ArrayList we can add only string type of objects by mistake if we are trying to add any other type we will get compile time error that is through generics we are getting type safety.
- At the time of retrieval it is not required to perform any type casting we can assign elements directly to string type variables.

Example:

```
ArrayList<String> l=new ArrayList<String>();
l.add("A");
.
.
.
String name1= l.get(0);
```

Type casting is not required

- That is through generic syntax we can resolve type casting problems.

Conclusion1:

- Polymorphism concept is applicable only for the base type but not for parameter type[usage of parent reference to hold child object is called polymorphism].

Example:

```
ArrayList<String> l1=new ArrayList<String>();
List<String> l2=new  ArrayList<String>();
Collection<String> l3=new  ArrayList<String>();
ArrayList<Object> l4=new ArrayList<String>();
```

parameter type

base Type

C.E →
```
Test.java:9: incompatible types
found   : java.util.ArrayList<java.lang.String>
required: java.util.ArrayList<java.lang.Object>
    ArrayList<Object> l4=new ArrayList<String>();
```

Concluson2:

- For the parameter type we can use any class or interface but not primitive value(type).

Example:

```
ArrayList<int> l=new ArrayList<int>();
```

```
Test.java:6: unexpected type
found   : int
required: reference
            ArrayList<int> l=new ArrayList<int>();
```

Generic classes:

- Until1.4v ArrayList class is declared as follows.

Example:

```
class ArrayList
{
    add(Object o);
    Object get(int index);
}
```

- add() method can take object as the argument and hence we can add any type of object to the ArrayList. Due to this we are not getting type safety.
- The return type of get() method is object hence at the time of retrieval compulsory we should perform type casting.
- But in 1.5v a generic version of ArrayList class is declared as follows.

Example:

```
                                  Type parameter
class ArrayList<T>
{
    add(T t);
    T get(int index);
}
```

- Based on our requirement **T** will be replaced with our provided type. For Example
- To hold only string type of objects we can create ArrayList object as follows.

Example:

```
ArrayList<String> l=new ArrayList<String>();
```

- For this requirement compiler considered ArrayList class is

Example:

```
class ArrayList<String>
{
    add(String s);
    String get(int index);
}
```

- add() method can take only string type as argument hence we can add only string type of objects to the List. By mistake if we are trying to add any other type we will get compile time error.

Example:

```
import java.util.*;
class Test
{
    public static void main(String[] args)
    {
        ArrayList<String> l=new ArrayList<String>();
        l.add("A");
        l.add(10);   C.E
    }
}
```

```
Test.java:8: cannot find symbol
symbol  : method add(int)
location: class java.util.ArrayList<java.lang.String>
            l.add(10);
```

- Hence through generics we are getting type safety.
- At the time of retrieval it is not required to perform any type casting we can assign its values directly to string variables.

Example:

```
import java.util.*;
class Test
{
    public static void main(String[] args)
    {
        ArrayList<String> l=new ArrayList<String>();
        l.add("A");
        l.add("10");
        String name1= l.get(0);
    }                          ->Type casting is not required
}
```

- Based on our requirement we can create our own generic classes also.

Example:

```
class Account<T>
{}
Account<Gold> g1=new Account<Gold>();
Account<Silver> g2=new Account<Silver>();
```

Example:

```
class UDGenerics<T>
{
    T obj;
    UDGenerics(T obj)
    {
        this.obj=obj;
    }
    public void show()
    {
        System.out.println("The type of object is :"+obj.getClass().getName());
    }
    public T getObject()
    {
        return obj;
    }
}
class GenericsDemo
{
    public static void main(String[] args)
    {
        UDGenerics<Integer> g1=new UDGenerics<Integer>(10);
        g1.show();
        System.out.println(g1.getObject());
      UDGenerics<String> g2=new UDGenerics<String>("bhaskar");
        g2.show();
        System.out.println(g2.getObject());
      UDGenerics<Double> g3=new UDGenerics<Double>(10.5);
        g3.show();
        System.out.println(g3.getObject());
    }
}
```

Output:

```
The type of object is: java.lang.Integer
10
The type of object is: java.lang. String
Bhaskar
The type of object is: java.lang. Double
10.5
```

Bounded types:

- We can bound the type parameter for a particular range by using extends keyword such types are called bounded types.

Example 1:

```
class Test<T>
{}
Test <Integer> t1=new Test < Integer>();
```

Test <String> t2=new Test < String>();

- Here as the type parameter we can pass any type and there are no restrictions hence it is unbounded type.

Example 2:

class Test<T extends X>
{}

- If x is a class then as the type parameter we can pass either x or its child classes.
- If x is an interface then as the type parameter we can pass either x or its implementation classes.

Example 1:

```
class Test<T extends Number>
{}
class Test1
{
    public static void main(String[] args)
    {
        Test<Integer> t1=new Test<Integer>();
        Test<String> t2=new Test<String>();
    }
}
```

→ type parameter java.lang.String is not within its bound
Test<String> t2=new Test<String>();

Example 2:

```
class Test<T extends Runnable>
{}
class Test1
{
    public static void main(String[] args)
    {
        Test<Thread> t1=new Test<Thread>();
        Test<String> t2=new Test<String>();
    }
}
```

C.E → Test1.java:8: type parameter java.lang.String is not within its bound
Test<String> t2=new Test<String>();

- We can't define bounded types by using implements and super keyword.

Example:

class Test<T implements Runnable>	class Test<T super String>
{}	{}
(invalid)	(invalid)

- But implements keyword purpose we can replace with extends keyword.
- As the type parameter we can use any valid java identifier but it convention to use **T** always.

Example:

class Test<X>	class Test<bhaskar>
{}	{}

- We can pass any no of type parameters need not be one.

Example:

```
class HashMap<K,V>
{}
```

K → key type
V → value type

HashMap<Integer,String> h=new HashMap<Integer,String>();

- We can define bounded types even in combination also.

Example 1:

class Test<T extends Number&Runnable>
{}(valid)

- As the type parameter we can pass any type which extends Number class and implements Runnable interface.

Example 2:

class Test<T extends Number&Runnable&Comparable>
{}(valid)

Example 3:

class Test<T extends Number&String>

{}(invalid)

- We can't extend more than one class at a time.
  Example 4:
  class Test<T extends Runnable&Comparable>
  {}(valid)
  Example 5:
  class Test<T extends Runnable&Number>
  {}(invalid)
- We have to take 1st class followed by interface.
  Generic methods and wild-card character (?):
  methodOne(ArrayList<String> l): This method is applicable for ArrayList of only String type.
  Example:
  l.add("A");
  l.add(null);
  l.add(10);//(invalid)
- Within the method we can add only String type of objects and null to the List.
  methodOne(ArrayList<?> l): We can use this method for ArrayList of any type but within the method we can't add anything to the List except null.
  Example:
  l.add(null);//(valid)
  l.add("A");//(invalid)
  l.add(10);//(invalid)
- This method is useful whenever we are performing only read operation.
  methodOne(ArrayList<? Extends x> l):
- If x is a class then this method is applicable for ArrayList of either x type or its child classes.
- If x is an interface then this method is applicable for ArrayList of either x type or its implementation classes.
- In this case also within the method we can't add anything to the List except null.
  methodOne(ArrayList<? super x> l):
- If x is a class then this method is applicable for ArrayList of either x type or its super classes.
- If x is an interface then this method is applicable for ArrayList of either x type or super classes of implementation class of x.
- But within the method we can add x type objects and null to the List.

Which of the following declarations are allowed?

1) ArrayList<String> l1=new ArrayList<String>();//(valid)
2) ArrayList<?> l2=new ArrayList<String>();//(valid)
3) ArrayList<?> l3=new ArrayList<Integer>();//(valid)
4) ArrayList<? extends Number> l4=new ArrayList<Integer>();//(valid)
5) ArrayList<? extends Number> l5=new ArrayList<String>();(invalid)
   Output:
   Compile time error.
   Test.java:10: incompatible types
   Found : java.util.ArrayList<java.lang.String>
   Required: java.util.ArrayList<? extends java.lang.Number>
   ArrayList<? extends Number> l5=new ArrayList<String>();
6) ArrayList<?> l6=new ArrayList<? extends Number>();
   Output:
   Compile time error
   Test.java:11: unexpected type
   found : ? extends java.lang.Number
   required: class or interface without bounds
   ArrayList<?> l6=new ArrayList<? extends Number>();
7) ArrayList<?> l7=new ArrayList<?>();
   Output:
   Test.java:12: unexpected type
   Found :?
   Required: class or interface without bounds
   ArrayList<?> l7=new ArrayList<?>();

- We can declare the type parameter either at class level or method level.
  Declaring type parameter at class level:
  class Test<T>
  {
  We can use anywhere this 'T'.
  }
  Declaring type parameter at method level:
- We have to declare just before return type.
  Example:
  public <T> void methodOne1(T t){}//valid

```
public <T extends Number> void methodOne2(T t){}//valid
public <T extends Number&Comparable> void methodOne3(T t){}//valid
public <T extends Number&Comparable&Runnable> void methodOne4(T t){}//valid
public <T extends Number&Thread> void methodOne(T t){}//invalid
```

Output:

```
Compile time error.
Test.java:7: interface expected here
    public <T extends Number&Thread> void methodOne(T t){}//valid
public <T extends Runnable&Number> void methodOne(T t){}//invalid
```

Output:

```
Compile time error.
Test.java:8: interface expected here
    public <T extends Runnable&Number> void methodOne(T t){}//valid
```

Communication with non generic code:

- To provide compatibility with old version sun people compramized the concept of generics in very few area's the following is one such area.

Example:

```
import java.util.*;
class Test
{
    public static void main(String[] args)
    {
        ArrayList<String> l=new ArrayList<String>();
        l.add("A");
        //l.add(10);//C.E:cannot find symbol,method add(int)
        methodOne(l);
        l.add(10.5);//C.E:cannot find symbol,method add(double)
        System.out.println(l);//[A, 10, 10.5, true]
    }
    public static void methodOne(ArrayList l)
    {
        l.add(10);
        l.add(10.5);
        l.add(true);
    }
}
```

Note:

- Generics concept is applicable only at compile time, at runtime there is no such type of concept. Hence the following declarations are equal.

ArrayList l=new ArrayList<String>();
ArrayList l=new ArrayList<Integer>(); } **All are equal.**
ArrayList l=new ArrayList();

Example 1:

```
import java.util.*;
class Test
{
    public static void main(String[] args)
    {
        ArrayList l=new ArrayList<String>();
        l.add(10);
        l.add(10.5);
        l.add(true);
        System.out.println(l);// [10, 10.5, true]
    }
}
```

Example 2:

```
import java.util.*;
class Test
{
    public void methodOne(ArrayList<String> l){}
    public void methodOne(ArrayList<Integer> l){}
}
```

Output:

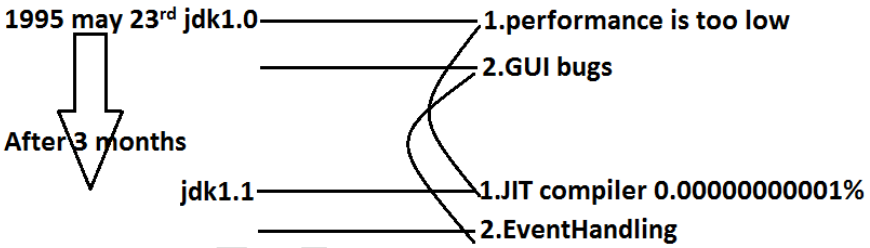
```
Compile time error.
Test.java:4: name clash: methodOne(java.util.ArrayList<java.lang.String>) and me
thodOne(java.util.ArrayList<java.lang.Integer>) have the same erasure
```

public void methodOne(ArrayList<String> l){}

- The following 2 declarations are equal.
  ArrayList<String> l1=new ArrayList();
  ArrayList<String> l2=new ArrayList<String>();
- For these ArrayList objects we can add only String type of objects.
  Example:
  l1.add("A");//valid
  l1.add(10); //invalid

Inner Classes

- Sometimes we can declare a class inside another class such type of classes are called inner classes.
  Diagram:



- Sun people introduced inner classes in 1.1 version as part of "**EventHandling**" to resolve GUI bugs.
- But because of powerful features and benefits of inner classes slowly the programmers starts using in regular coding also.
- Without existing one type of object if there is no chance of existing another type of object then we should go for inner classes.
  Example: Without existing University object there is no chance of existing Department object hence we have to define Department class inside University class.
  Example1:

```
class University------------Outer class
{
    class Department -------inner class
    {
    }
}
```

Example 2: Without existing Bank object there is no chance of existing Account object hence we have to define Account class inside Bank class.
Example:

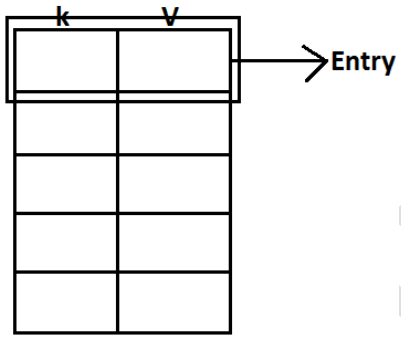
```
class Bank------------Outer class
{
    class Account-------inner class
    {
    }
}
```

Example 3: Without existing Map object there is no chance of existing Entry object hence Entry interface is define inside Map interface.

Example:

```
interface Map ————— outer interface
{
      interface Entry —— inner interface
      {
      }
}
```

Diagram:



Note: The relationship between outer class and inner class is not IS-A relationship and it is Has-A relationship.

- Based on the purpose and position of declaration all inner classes are divided into 4 types. They are:

1) Normal or Regular inner classes
2) Method Local inner classes
3) Anonymous inner classes
4) Static nested classes.

1. Normal (or) Regular inner class: If we are declaring any named class inside another class directly without static modifier such type of inner classes are called normal or regular inner classes.

Example:

```
class Outer
{
      class Inner
      {
      }
}
```

Output:

```
javac Outer.java
Outer.class        Outer$Inner.class
E:\scjp>java Outer
Exception in thread "main" java.lang.NoSuchMethodError: main
E:\scjp>java Outer$Inner
Exception in thread "main" java.lang.NoSuchMethodError: main
```

Example:

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

Output:

```
javac Outer.java
Outer.class
Outer$Inner.class
E:\scjp>java Outer
outer class main method
E:\scjp>java Outer$Inner
Exception in thread "main" java.lang.NoSuchMethodError: main
```

- Inside inner class we can't declare static members. Hence it is not possible to declare main() method and we can't invoke inner class directly from the commend prompt.

Example:

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

Output:

```
E:\scjp>javac Outer.java
Outer.java:5: inner classes cannot have static declarations
		public static void main(String[] args)
```

Accessing inner class code from static area of outer class:

Example:

```
class Outer
{
	class Inner
	{
		public void methodOne(){
			System.out.println("inner class method");
		}
	}
	public static void main(String[] args)
	{
		Outer o=new Outer();
		Outer.Inner i=o.new Inner();
		i.methodOne();
		(or)
		Outer.Inner i=new Outer().new Inner();
		i.methodOne();
		(or)
		new Outer().new Inner().methodOne();
	}
}
```

Accessing inner class code from instance area of outer class:

Example:

```
class Outer
{
	class Inner
	{
		public void methodOne()
		{
			System.out.println("inner class method");
		}
	}
	public void methodTwo()
	{
		Inner i=new Inner();
		i.methodOne();
	}
```

```
        public static void main(String[] args)
        {
                Outer o=new Outer();
                o.methodTwo();
        }
}
```

Output:

```
E:\scjp>javac Outer.java
E:\scjp>java Outer
Inner class method
```

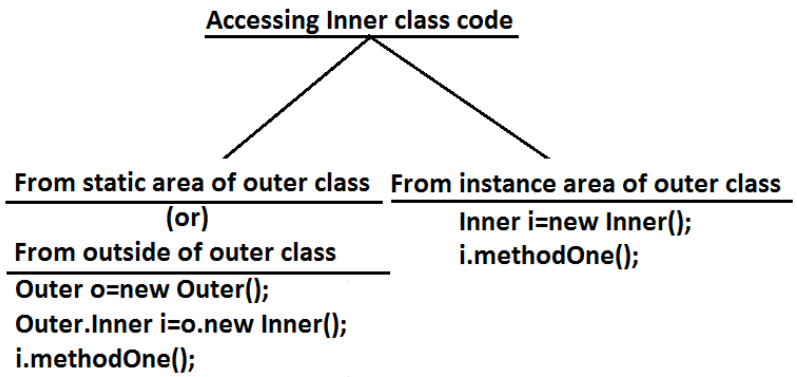
Accessing inner class code from outside of outer class:

Example:

```
class Outer
{
        class Inner
        {
                public void methodOne()
                {
                        System.out.println("inner class method");
                }
        }
}
class Test
{
        public static void main(String[] args)
        {
                new Outer().new Inner().methodOne();
        }
}
```

Output:

```
Inner class method
```



- From inner class we can access all members of outer class (both static and non-static, private and non private methods and variables) directly.

Example:

```
class Outer
{
        int x=10;
        static int y=20;
        class Inner{
                public void methodOne()
                {
                        System.out.println(x);//10
                        System.out.println(y);//20
                }
        }
        public static void main(String[] args)
        {
                new Outer().new Inner().methodOne();
```

```
        }
    }
```

- Within the inner class "**this**" always refers current inner class object. To refer current outer class object we have to use "**outer class name.this**".

Example:

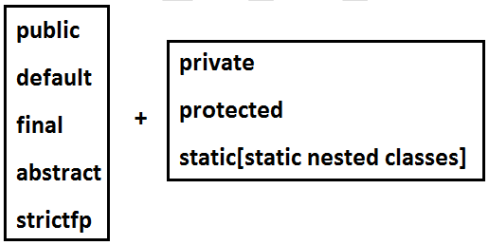
```
class Outer
{
    int x=10;
    class Inner
    {
        int x=100;
        public void methodOne()
        {
            int x=1000;
            System.out.println(x);//1000
            System.out.println(this.x);//100
            System.out.println(Outer.this.x);//10
        }
    }
    public static void main(String[] args)
    {
        new Outer().new Inner().methodOne();
    }
}
```

- The applicable modifiers for outer classes are:

1) **public**
2) **default**
3) **final**
4) **abstract**
5) **strictfp**

- But for the inner classes in addition to this the following modifiers also allowed.

Diagram:



Method local inner classes:

- Sometimes we can declare a class inside a method such type of inner classes are called method local inner classes.
- The main objective of method local inner class is to define method specific repeatedly required functionality.
- We can access method local inner class only within the method where we declared it. That is from outside of the method we can't access. As the scope of method local inner classes is very less, this type of inner classes are most rarely used type of inner classes.

Example:

```
class Test
{
    public void methodOne()
    {
        class Inner
        {
            public void sum(int i,int j)
            {
                System.out.println("The sum:"+(i+j));
            }
        }
        Inner i=new Inner();
        i.sum(10,20);
        ;;;;;;;;;;;;;;
        i.sum(100,200);
        ;;;;;;;;;;;;;;;;
        i.sum(1000,2000);
        ;;;;;;;;;;;;;;;;;
    }
    public static void main(String[] args)
```

```
    {
        new Test().methodOne();
    }
}
```

Output:

```
The sum: 30
The sum: 300
The sum: 3000
```

- If we are declaring inner class inside instance method then we can access both static and non static members of outer class directly.
- But if we are declaring inner class inside static method then we can access only static members of outer class directly and we can't access instance members directly.

Example:

```
class Test
{
    int x=10;
    static int y=20;
    public void methodOne()
    {
        class Inner
        {
            public void methodTwo()
            {
                System.out.println(x);//10
                System.out.println(y);//20
            }
        }
        Inner i=new Inner();
        i.methodTwo();
    }
    public static void main(String[] args)
    {
        new Test().methodOne();
    }
}
```

- If we declare methodOne() method as static then we will get compile time error saying "non-static variable x cannot be referenced from a static context".
- From method local inner class we can't access local variables of the method in which we declared it. But if that local variable is declared as final then we won't get any compile time error.

Example:

```
class Test
{
    int x=10;
    public void methodOne()
    {
        int y=20;
        class Inner
        {
            public void methodTwo()
            {
                System.out.println(x);//10
                System.out.println(y); //C.E: local variable y is accessed from within inner class; needs to be
declared final.
            }
        }
        Inner i=new Inner();
        i.methodTwo();
    }
    public static void main(String[] args)
    {
        new Test().methodOne();
    }
}
```

- If we declared y as final then we won't get any compile time error.
- Consider the following declaration.

```
class Test
{
```

```
        int i=10;
        static int j=20;
        public void methodOne()
        {
                int k=30;
                final int l=40;
                class Inner
                {
                        public void methodTwo()
                        {
                                System.out.println(i);
                                System.out.println(j);
                                System.out.println(k);
                                System.out.println(l);
                        }
                }
                Inner i=new Inner();
                i.methodTwo();
        }
        public static void main(String[] args)
        {
                new Test().methodOne();
        }
}
```

line 1

- **At line 1 which of the following variables we can access?**

1) i 2) j 3) k 4) l

- **If we declare methodOne() method as static then which variables we can access at line 1?**

1) i 2) j 3) k 4) l

- If we declare methodTwo() as static then we will get compile time error because we can't declare static members inside inner classes.
- The only applicable modifiers for method local inner classes are:

1) final
2) abstract
3) strictfp

- By mistake if we are declaring any other modifier we will get compile time error.

Anonymous inner classes:

- Sometimes we can declare inner class without name such type of inner classes are called anonymous inner classes.
- The main objective of anonymous inner classes is "**just for instant use**".
- There are 3 types of anonymous inner classes

1) Anonymous inner class that extends a class.
2) Anonymous inner class that implements an interface.
3) Anonymous inner class that defined inside method arguments.

Anonymous inner class that extends a class:

```
class PopCorn
{
        public void taste()
        {
                System.out.println("spicy");
        }
}
class Test
{
        public static void main(String[] args)
        {
                PopCorn p=new PopCorn()
                {
                        public void taste()
                        {
                                System.out.println("salty");
                        }
                };
                p.taste();//salty
                PopCorn p1=new PopCorn();
                p1.taste();//spicy
```

```
    }
}
```

Analysis:

1) PopCorn p=new PopCorn();

- We are just creating a PopCorn object.

2) PopCorn p=new PopCorn()

```
        {
        };
```

- We are creating child class without name for the PopCorn class and for that child class we are creating an object with Parent PopCorn reference.

3) PopCorn p=new PopCorn()

```
    {
        public void taste()
        {
        System.out.println("salty");
        }
    };
```

1) We are creating child class for PopCorn without name.
2) We are overriding taste() method.
3) We are creating object for that child class with parent reference.

Note: Inside Anonymous inner classes we can take or declare new methods but outside of anonymous inner classes we can't call these methods directly because we are depending on parent reference.[parent reference can be used to hold child class object but by using that reference we can't call child specific methods]. These methods just for internal purpose only.

Example 1:

```
class PopCorn
{
    public void taste()
    {
        System.out.println("spicy");
    }
}
class Test
{
    public static void main(String[] args)
    {
        PopCorn p=new PopCorn()
        {
            public void taste()
            {
                methodOne();//valid call(internal purpose)
                System.out.println("salty");
            }
            public void methodOne()
            {
                System.out.println("child specific method");
            }
        };
        //p.methodOne();//here we can not call(outside inner class)
        p.taste();//salty
        PopCorn p1=new PopCorn();
        p1.taste();//spicy
    }
}
```

Output:

```
Child specific method
Salty
Spicy
```

Example 2:

```
class Test
{
    public static void main(String[] args)
    {
        Thread t=new Thread()
        {
            public void run()
            {
```

```
                                        for(int i=0;i<10;i++)
                                        {
                                                System.out.println("child thread");
                                        }
                                }
                        };
                        t.start();
                        for(int i=0;i<10;i++)
                        {
                                System.out.println("main thread");
                        }
        }
}
```

Anonymous Inner Class that implements an interface:

Example:

```
class InnerClassesDemo
{
        public static void main(String[] args)
        {
                Runnable r=new Runnable()//here we are not creating for Runnable interface, we are creating implements class object.
                {
                        public void run()
                        {
                                for(int i=0;i<10;i++)
                                {
                                        System.out.println("Child thread");
                                }
                        }
                };
                Thread t=new Thread(r);
                t.start();
                for(int i=0;i<10;i++)
                {
                        System.out.println("Main thread");
                }
        }
}
```

Anonymous Inner Class that define inside method arguments:

Example:

```
class Test
{
        public static void main(String[] args)
        {
                new Thread(
                        new Runnable()
                {
                        public void run()
                        {
                                for(int i=0;i<10;i++)
                                {
                                        System.out.println("child thread");
                                }
                        }
                }).start();
                for(int i=0;i<10;i++)
                {
                        System.out.println("main thread");
                }
        }
}
```

Output:

- This output belongs to example 2, anonymous inner class that implements an interface example and anonymous inner class that define inside method arguments example.

```
Main thread
Main thread
Main thread
```

```
Main thread
Main thread
Main thread
Main thread
Main thread
Main thread
Main thread
Child thread
Child thread
Child thread
Child thread
Child thread
Child thread
Child thread
Child thread
Child thread
Child thread
```

Difference between general class and anonymous inner classes:

General Class	Anonymous Inner Class
1) A general class can extends only one class at a time.	1) Ofcource anonymous inner class also can extends only one class at a time.
2) A general class can implement any no. Of interfaces at a time.	2) But anonymous inner class can implement only one interface at a time.
3) A general class can extends a class and can implement an interface simultaneously.	3) But anonymous inner class can extends a class or can implements an interface but not both simultaneously.

Static nested classes:

- Sometimes we can declare inner classes with static modifier such type of inner classes are called static nested classes.
- In the case of normal or regular inner classes without existing outer class object there is no chance of existing inner class object.
- But in the case of static nested class without existing outer class object there may be a chance of existing static nested class object.

Example:

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

- Inside static nested classes we can declare static members including main() method also. Hence it is possible to invoke static nested class directly from the command prompt.

Example:

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

Output:

```
E:\SCJP>javac Test.java
```

```
E:\SCJP>java Test
Outer class main method
E:\SCJP>java Test$Nested
Nested class main method
```

- From the normal inner class we can access both static and non static members of outer class but from static nested class we can access only static members of outer class.

Example:

```
class Test
{
    int x=10;
    static int y=20;
    static class Nested
    {
        public void methodOne()
        {
            System.out.println(x);//C.E:non-static variable x cannot be referenced from a static context
            System.out.println(y);
        }
    }
}
```

Compression between normal or regular class and static nested class?

Normal /regular inner class	Static nested class
1) Without existing outer class object there is no chance of existing inner class object. That is inner class object is always associated with outer class object.	1) Without existing outer class object there may be a chance of existing static nested class object. That is static nested class object is not associated with outer class object.
2) Inside normal or regular inner class we can't declare static members.	2) Inside static nested class we can declare static members.
3) Inside normal inner class we can't declare main() method and hence we can't invoke regular inner class directly from the command prompt.	3) Inside static nested class we can declare main() method and hence we can invoke static nested class directly from the command prompt.
4) From the normal or regular inner class we can access both static and non static members of outer class directly.	4) From static nested class we can access only static members of outer class directly.

Internationalization

- The process of designing a web application such that it supports various countries, various languages without performing any changes in the application is called Internationalization.
- We can implement Internationalization by using the following classes. They are:

1) Locale
2) NumberFormat
3) DateFormat

1. Locale: A Locale object can be used to represent a geographic (country) location (or) language.

- Locale class present in **java.util package**.
- It is a final class and direct child class of Object implements Cloneable and Serializable Interfaces.

How to create a Locale object:

- We can create a Locale object by using the following constructors of Locale class.

1) Locale l=new Locale(String language);
2) Locale l=new Locale(String language,String country);

- Locale class already defines some predefined Locale constants. We can use these constants directly.

Example:
Locale. UK
Locale. US
Locale. ITALY
Locale. CHINA

Important methods of Locale class:

1) public static Locale getDefault()
2) public static void setDefault(Locale l)
3) public String getLanguage()
4) public String getDisplayLanguage(Locale l)
5) public String getCountry()
6) public String getDisplayCountry(Locale l)
7) public static String[] getISOLanguages()
8) public static String[] getISOCountries()
9) public static Locale[] getAvailableLocales()

Example for Locale:

```
import java.util.*;
class LocaleDemo{
public static void main(String args[]){
Locale l1=Locale.getDefault();
//System.out.println(l1.getCountry()+"....."+l1.getLanguage());
//System.out.println(l1.getDisplayCountry()+"....."+l1.getDisplayLanguage());
Locale l2=new Locale("pa","IN");
Locale.setDefault(l2);
String[] s3=Locale.getISOLanguages();
for(String s4:s3)
{
//System.out.print("ISO language is           :");
//System.out.println(s4);
}
String[] s4=Locale.getISOCountries();
for(String s5:s4)
{
System.out.print("ISO Country is:");
System.out.println(s5);
}
Locale[] s=Locale.getAvailableLocales();
for(Locale s1:s)
{
//System.out.print("Available locales is:");
//System.out.println(s1.getDisplayCountry()+"....."+s1.getDisplayLanguage());
}}}
```

NumberFormat:

- Various countries follow various styles to represent number.

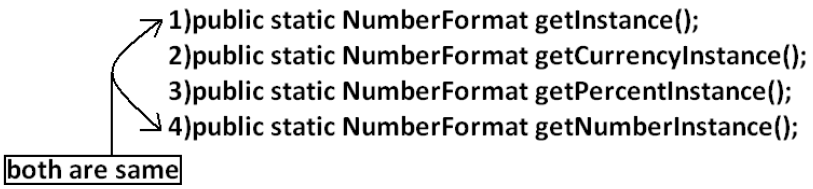
Example:
1,23,456.789----------INDIA
123,456.789----------US
123.456,789----------ITALY

- By using NumberFormat class we can format a number according to a particular Locale.
- NumberFormat class present in java.Text package and it is an abstract class.
- Hence we can't create an object by using constructor.

NumberFormat nf=new NumberFormat(); --------invalid

Getting NumberFormat object for the default Locale:

- NumberFormat class defines the following methods for this.



Getting NumberFormat object for the specific Locale:

- The methods are exactly same but we have to pass the corresponding Locale object as argument.

Example: public static NumberFormat getNumberInstance(Locale l);

- Once we got NumberFormat object we can call the following methods to format and parse numbers.

1) public String format(long l);
2) public String format(double d);

- To convert a number from java form to Locale specific form.

1) public Number parse(String source)throws ParseException

- To convert from Locale specific String form to java specific form.

Example:

```
import java.util.*;
import java.text.*;
class NumberFormatDemo
{
public static void main(String args[]){
double d=123456.789;
NumberFormat nf=NumberFormat.getInstance(Locale.ITALY);
System.out.println("ITALY form is :"+nf.format(d));
}
}
```

Output:

```
ITALY form is :123.456,789
```

Requirement: Write a program to print a java number in INDIA, UK, US and ITALY currency formats.

Program:

```
import java.util.*;
import java.text.*;
class NumberFormatDemo
{
public static void main(String args[]){
double d=123456.789;
Locale INDIA=new Locale("pa","IN");
NumberFormat nf=NumberFormat.getCurrencyInstance(INDIA);
System.out.println("INDIA notation is :"+nf.format(d));
NumberFormat nf1=NumberFormat.getCurrencyInstance(Locale.UK);
System.out.println("UK notation is :"+nf1.format(d));
NumberFormat nf2=NumberFormat.getCurrencyInstance(Locale.US);
System.out.println("US notation is :"+nf2.format(d));
NumberFormat nf3=NumberFormat.getCurrencyInstance(Locale.ITALY);
System.out.println("ITALY notation is :"+nf3.format(d));
}}
```

Output:

```
INDIA notation is: INR 123,456.79
UK notation is: ú123,456.79
US notation is: $123,456.79
ITALY notation is: Ç 123.456,79
```

Setting Maximum, Minimum, Fraction and Integer digits:

- NumberFormat class defines the following methods for this purpose.

1) public void **setMaximumFractionDigits**(int n);
2) public void **setMinimumFractionDigits**(int n);
3) public void **setMaximumIntegerDigits**(int n);
4) public void **setMinimumIntegerDigits**(int n);

Example:

```
import java.text.*;
public class NumberFormatExample
{
public static void main(String[] args){
NumberFormat nf=NumberFormat.getInstance();
nf.setMaximumFractionDigits(3);
System.out.println(nf.format(123.4));
System.out.println(nf.format(123.4567));
nf.setMinimumFractionDigits(3);
System.out.println(nf.format(123.4));
System.out.println(nf.format(123.4567));
nf.setMaximumIntegerDigits(3);
System.out.println(nf.format(1.234));
System.out.println(nf.format(123456.789));
```

```
nf.setMinimumIntegerDigits(3);
System.out.println(nf.format(1.234));
System.out.println(nf.format(123456.789));
}}
```

Output:

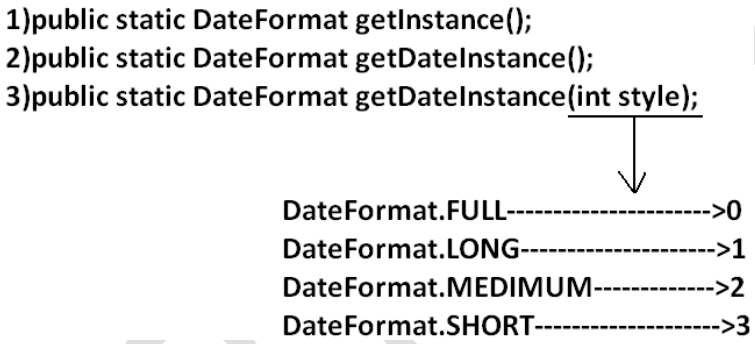
```
123.4
123.457
123.400
123.457
1.234
456.789
001.234
456.789
```

DateFormat: Various countries follow various styles to represent Date. We can format the date according to a particular locale by using DateFormat class.

- DateFormat class present in java.text package and it is an abstract class.

Getting DateFormat object for default Locale:

- DateFormat class defines the following methods for this purpose.



Getting DateFormat object for the specific Locale:

1) public static DateFormat getDateInstance(int style, Locale l);

- Once we got DateFormat object we can format and parse Date by using the following methods.

1) public String format(Date date);

- To convert the date from java form to locale specific string form.

1) public Date parse(String source)throws ParseException

- To convert the date from locale specific form to java form.

Requirement: Write a program to represent current system date in all possible styles of us format.

Program:

```
import java.text.*;
import java.util.*;
public class DateFormatDemo
{
public static void main(String args[]){
System.out.println("full form is :"+DateFormat.getDateInstance(0).format(new Date()));
System.out.println("long form is :"+DateFormat.getDateInstance(1).format(new Date()));
System.out.println("medium form is :"+DateFormat.getDateInstance(2).format(new Date()));
System.out.println("short form is :"+DateFormat.getDateInstance(3).format(new Date()));
}
}
```

Output:

```
Full form is: Wednesday, July 20, 2011
Long form is: July 20, 2011
Medium form is: Jul 20, 2011
Short form is: 7/20/11
```

Note: The default style is medium style.

Requirement: Write a program to represent current system date in UK, US and ITALY styles.

Program:

```
import java.text.*;
import java.util.*;
public class DateFormatDemo
```

```
{
public static void main(String args[]){
DateFormat UK=DateFormat.getDateInstance(0,Locale.UK);
DateFormat US=DateFormat.getDateInstance(0,Locale.US);
DateFormat ITALY=DateFormat.getDateInstance(0,Locale.ITALY);
System.out.println("UK style is :"+UK.format(new Date()));
System.out.println("US style is :"+US.format(new Date()));
System.out.println("ITALY style is :"+ITALY.format(new Date()));
}
}
```

Output:

UK style is: Wednesday, 20 July 2011
US style is: Wednesday, July 20, 2011
ITALY style is: mercoled∞ 20 luglio 2011

Getting DateFormat object to get both date and time:

- DateFormat class defines the following methods for this.

1)public static DateFormat getDateTimeInstance();
2)public static DateFormat getDateTimeInstance(int dateStyle, int timeStyle);
3)public static DateFormat getDateTimeInstance(int dateStyle, int timeStyle,Locale l);

dateStyle: 0 to 3
timeStyle: 0 to 3

Example:

```
import java.text.*;
import java.util.*;
public class DateFormatDemo
{
public static void main(String args[]){
DateFormat ITALY=DateFormat.getDateTimeInstance(0,0,Locale.ITALY);
System.out.println("ITALY style is:"+ITALY.format(new Date()));
}
}
```

Output:

ITALY style is: mercoled∞ 20 luglio 2011 23.21.30 IST

Development

Javac: we can use Javac to compile a single or group of ".java files".

Syntax:

javac [options] Test.java (valid)
Test.java Demo.java (valid)
*.java (valid)

[options]
-source
-version
-cp <path>
-d <directory>
.
.
.
.

Java: we can use java command to run a single ".class file".

Syntax:

```
java    [options]    classfile    arg[0]    arg[1].....
        -version
        -ea/-esa/-da/-dsa
        -D
        -cp/-classpath
              .
              .
              .
              .
```

Classpath: Class path describes the location where the required ".class files" are available. We can set the class path in the following 3 ways.

1) Permanently by using environment variable "classpath". This class path will be preserved after system restart also.
2) Temporary for a particular command prompt level by using "set" command.

Example:

```
set classpath=%classpath%;D:\durga_classes;.;
```

- Once if you close the command prompt automatically this class path will be lost.

3) We can set the class path for a particular command level by using "–cp" (or) "–class path". This class path is applicable only for that command execution. After executing the command this classpath will be lost.

- Among the 3 ways of setting the class path the most common way is setting class path at command level by using "–cp".

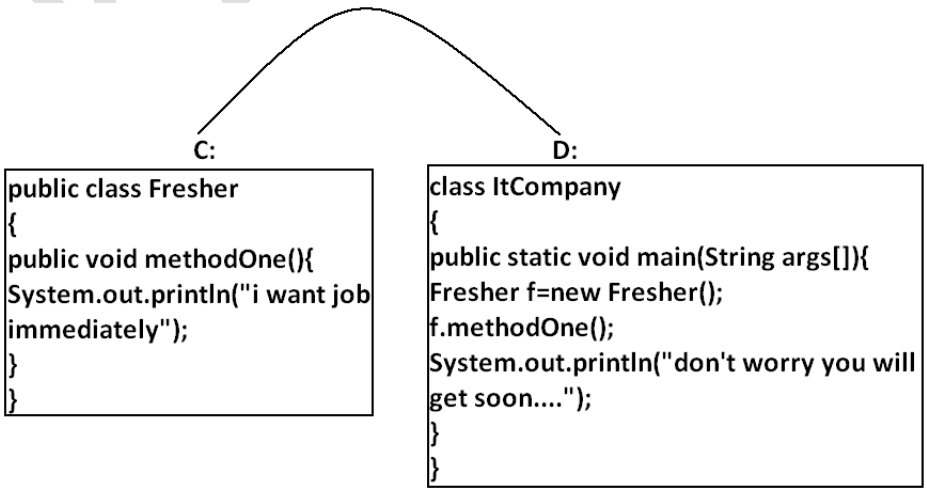
Example 1:

```
class Rain
{
public static void main(String args[]){
System.out.println("Raining of jobs these days");
}
}
```

Analysis:

```
D:\Java>javac Rain.java (valid)
D:\Java>java Rain  (valid)
Raining of jobs these days.
D:\>java Rain  (invalid)
Exception in thread "main" java.lang.NoClassDefFoundError: Rain
D:\>java -cp D:\java Rain  (valid)
Raining of jobs these days.
D:\>java Rain  (invalid)
C:\>java -cp d:\java Rain  (valid)
Raining of jobs these days
```

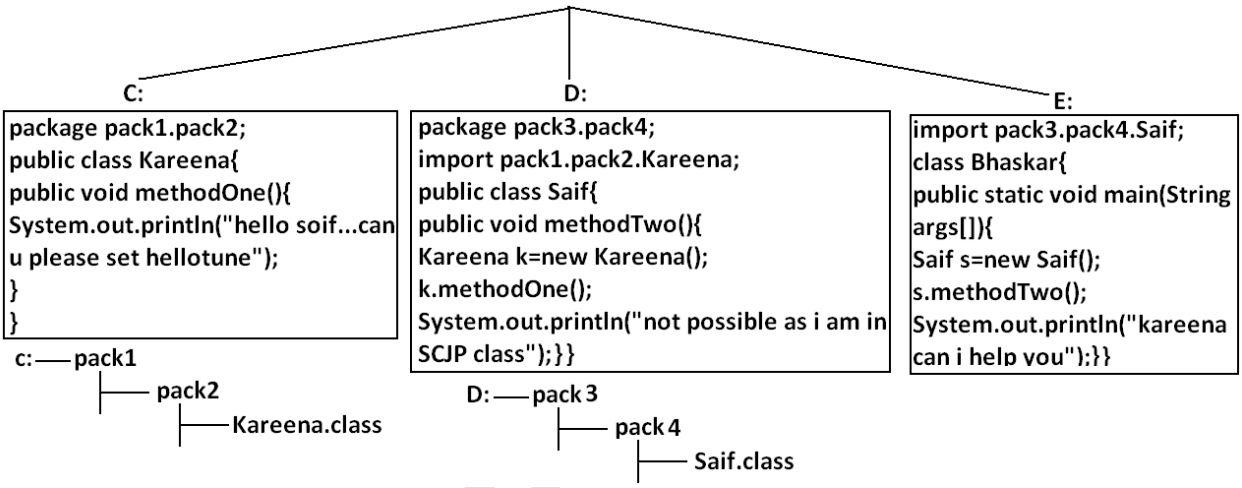
Example 2:



Analysis:

C:\>javac Fresher.java (valid)
D:\>javac ItCompany.java (invalid)
compile time error: ItCompany.java:4: cannot find symbol
symbol : class Fresher
location: class ItCompany
Fresher f=new Freaher();
D:\>javac -cp c: ItCompany.java (valid)
D:\>java ItCompany
Runtime error:NoClassDefFoundError: Fresher
D:\>java -cp c: ItCompany
Runtime error:NoClassDefFoundError: ItCompany
D:\>java -cp .;c: ItCompany (valid)
i want job immediately
don't worry you will get soon....
E:\>java -cp D:;C: ItCompany (valid)

Example 3:



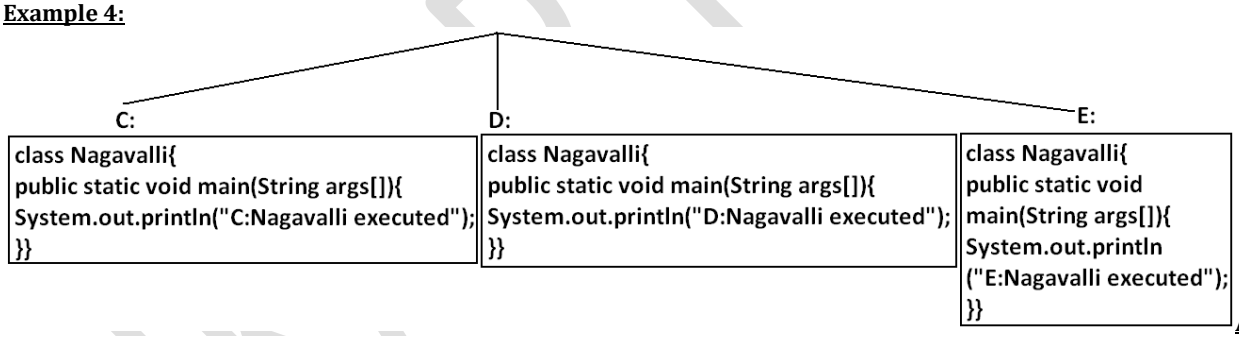
Analysis:

C:\>javac -d . Kareena.java (valid)
D:\>javac -d . Saif.java (invalid) ⟶ compile time error
Saif.java:5: cannot find symbol
symbol : class Kareena
location: class pack3.pack4.Saif
Kareena k=new Kareena();
D:\>javac -cp c: -d . Saif.java (valid)
E:\>javac Bhaskar.java (invalid) ⟶ compile time error
Bhaskar.java:4: cannot find symbol
symbol : class Saif
location: class Bhaskar
Saif s=new Saif();
E:\>javac -cp d: Bhaskar.java (valid)
E:\>java Bhaskar (invalid) ⟶ Runtime error
NoClassDefFoundError: pack3/pack4/Saif
E:\>java -cp d: Bhaskar (invalid) ⟶ Runtime error
NoClassDefFoundError: Bhaskar

E:\>java -cp .;d: Bhaskar (invalid) ⟶ Runtime error
NoClassDefFoundError: pack1/pack2/Kareena
E:\>java -cp .;d:;c: Bhaskar (valid) ⟶ output:
hello soif...can u please set hellotune
not possible as i am in SCJP class
kareena can i help you
F:\>java -cp E:;D:;C: Bhaskar (valid) ⟶ output:
hello soif...can u please set hellotune
not possible as i am in SCJP class
kareena can i help you

Note: if any folder structure created because of package statement. It should be resolved by import statement only and the location of base package should be make it available in class path.
Note: in classpath the order of locations is very important and it should be from left to right.

Example 4:



Analysis:

C:\>javac Nagavalli.java (valid)
D:\>javac Nagavalli.java (valid)
E:\>javac Nagavalli.java (valid)
C:\>java Nagavalli output:
C:Nagavalli executed
C:\>java -cp e:;d:;c: Nagavalli output:
E:Nagavalli executed
C:\>java -cp d:;e:;c: Nagavalli output:
D:Nagavalli executed

Jar file: If several dependent classes present then it is never recommended to set the classpath individual for every component. We have to group all these ".class files" into a single jar file and we have to make that jar file available to the classpath.
Example: All required classes to develop a Servlet are grouped into a single jar file (Servlet-api.jar) hence while compiling Servlet classes we have to make this jar file available in the classpath.
What is the difference between Jar, War and Ear?
Jar (java archive): Represents a group of ".class files".
War (web archive): Represents a web application which may contains Servlets, JSP, HTML pages, JavaScript files etc.
Ear (Enterprise archive): it represents an enterprise application which may contain Servlets, JSP, EJB'S, JMS component etc.

- In generally an ear file consists of a group of war files and jar files.

Ear=war+ jar

Various Commands:

To create a jar file:

D:\Enum>jar -cvf bhaskar.jar Beer.class Test.class X.class
D:\Enum>jar -cvf bhaskar.jar *.class

To extract a jar file:

D:\Enum>jar -xvf bhaskar.jar

To display table of contents of a jar file:

D:\Enum>jar -tvf bhaskar.jar

Example 5:

```
public class BhaskarColorFulCalc{
public static int add(int x,int y){
return x*y;
}
public static int multiply(int x,int y){
return 2*x*y;
}}
```

Analysis:

C:\>javac BhaskarColorFulCalc.java
C:\>jar -cvf bhaskar.jar BhaskarColorFulCalc.class

Example 6:

```
class Client{
public static void main(String args[]){
System.out.println(BhaskarColorFulCalc.add(10,20));
System.out.println(BhaskarColorFulCalc.multiply(10,20));
}}
```

Analysis:

D:\Enum>javac Client.java (invalid)
D:\Enum>javac -cp c: Client.java (invalid)
D:\Enum>javac -cp c:\bhaskar.jar Client.java (valid)
D:\Enum>java -cp .;c:\bhaskar.jar Client (valid)

Note: Whenever we are placing jar file in the classpath compulsory we have to specify the name of the jar file also and just location is not enough.

System properties:

- For every system some persistence information is available in the form of system properties. These may include name of the os, java version, vendor of jvm etc.
- We can get system properties by using getProperties() method of system class. The following program displays all the system properties.

Example 7:

```
import java.util.*;
class Test{
public static void main(String args[]){
//Properties is a class in util package.
//here getPropertes() method returns the Properties object.
Properties p=System.getProperties();
p.list(System.out);
}
}
```

How to set system property from the command prompt:

- We can set system property from the command prompt by using –D option.

Command:

D:\Enum>java -Dbhaskar=scjp Test
propertyname propertyvalue

What is the difference between path and classpath?

Path: We can use "path variable" to specify the location where required binary executables are available.

- If we are not setting path then "java" and "Javac" commands won't work.

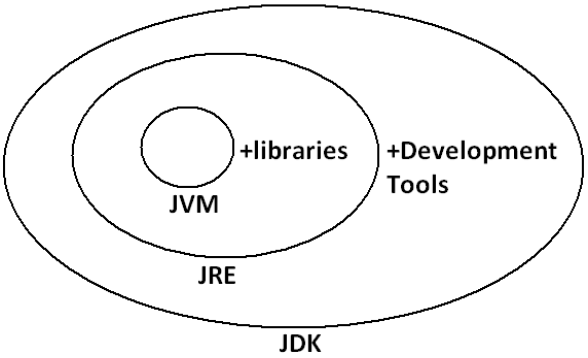
Classpath: We can use "classpath variable" to describe location where required class files are available.

- If we are not setting classpath then our program won't compile and run.

What is the difference between JDK, JRE and JVM?

JDK (java development kit): To develop and run java applications the required environment is JDK.
JRE (java runtime environment): To run java application the required environment is JRE.
JVM (java virtual machine): To execute java application the required virtual machine is JVM.

Diagram:



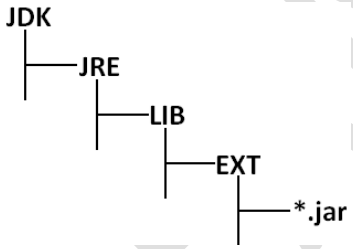
- JDK=JRE+Development Tools.
- JRE=JVM+Libraries.
- JRE is the part of JDK.
- Jvm is the part of JRE.

Note: At client side JRE is required and at developers side JDK is required.

Shortcut way to place a jar files:

- If we are placing jar file in the following location then it is not required to set classpath explicitly.

Diagram:



ENUM

- We can use enum to define a group of named constants.

Example 1:

```
enum Month
{
JAN,FEB,MAR,DEC;
}
```

Example 2:

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

- Enum concept introduced in 1.5 versions.
- When compared with old languages enum java's enum is more powerful.
- By using enum we can define our own data types which are also come enumerated data types.

Internal implementation of enum:

- Internally enum's are implemented by using class concept. Every enum constant is a reference variable to that enum type object.
- Every enum constant is implicitly **public static final** always.

Example 3:

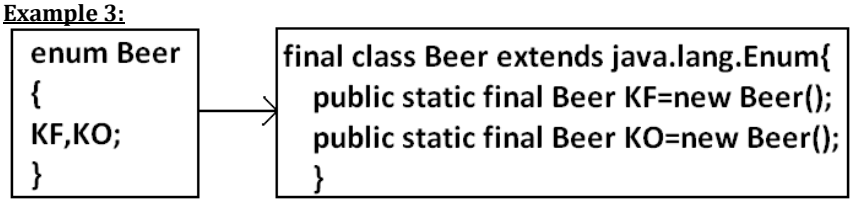


Diagram:

Declaration and usage of enum:

Example 4:

```
enum Beer
{
KF,KO,RC,FO;//here semicolon is optional.
}
class Test
{
public static void main(String args[]){
Beer b1=Beer.KF;
System.out.println(b1);
}
}
```

Output:

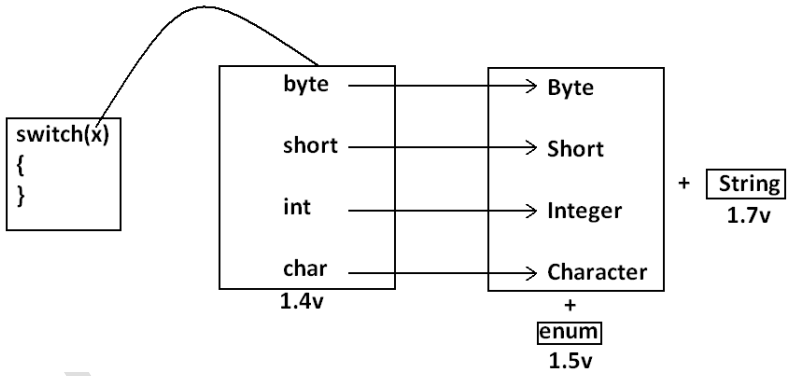
```
D:\Enum>java Test
KF
```

Note: Every enum constant internally static hence we can access by using "enum name".

Enum vs switch statement:

- Until 1.4 versions the allowed types for the switch statement are byte, short, char int. But from 1.5 version onwards in addition to this the corresponding wrapper classes and enum type also allowed. That is from 1.5 version onwards we can use enum type as argument to switch statement.

Diagram:



Example:

```
enum Beer
{
KF,KO,RC,FO;
}
class Test{
public static void main(String args[]){
Beer b1=Beer.RC;
switch(b1){
case KF:
      System.out.println("it is childrens brand");
      break;
case KO:
      System.out.println("it is too lite");
      break;
case RC:
      System.out.println("it is too hot");
      break;
case FO:
      System.out.println("buy one get one");
      break;
```

```
default:
    System.out.println("other brands are not good");
    }
}}
```

Output:

```
D:\Enum>java Test
It is too hot
```

- If we are passing enum type as argument to switch statement then every case label should be a valid enum constant otherwise we will get compile time error.

Example:

```
enum Beer
{
KF,KO,RC,FO;
}
class Test{
public static void main(String args[]){
Beer b1=Beer.RC;
switch(b1){
case KF:
case RC:
case KALYANI:
}}}
```

Output:

```
Compile time error.
D:\Enum>javac Test.java
Test.java:11: unqualified enumeration constant name required
case KALYANI:
```

- We can declare enum either outside the class or within the class but not inside a method. If we declare enum outside the class the allowed modifiers are:

1) public
2) default
3) strictfp.

- If we declare enum inside a class then the allowed modifiers are:

1) public private
2) default + protected
3) strictfp static

Example:

```
enum X
{ }
class Y
{ }
(valid)
```

```
class X
{
enum Y
{}
}
(valid)
```

```
class X
{
public void methodOne(){
enum X
{}
}
}
```

```
output:
compile time error.
D:\Enum>javac X.java
X.java:4: enum types must not be local
enum X
```

Enum vs inheritance:

- Every enum in java is the direct child class of java.lang.Enum class hence it is not possible to extends any other enum.
- Every enum is implicitly final hence we can't create child enum.
- Because of above reasons we can conclude inheritance concept is not applicable for enum's explicitly.
- But enum can implement any no. Of interfaces simultaneously.

Example:

```
enum X
{}
enum Y extends X
{}
(invalid)
```

```
enum X extends Enum
{}
(invalid)
```

```
class X
{}
enum Y extends X
{}
(invalid)
```

Example:

```
enum X
{}
class Y extends X
{}
output:
compile time error.
D:\Enum>javac Y.java
Y.java:3: cannot inherit from final X
class Y extends X
 Y.java:3: enum types are not extensible
class Y extends X
                (invalid)
```

```
interface X
{}
enum Y implements X
{}
        (valid)
```

Java.lang.Enum: Every enum in java is the direct child class of java.lang.Enum. The power of enum is inheriting from this class only.

- It is abstract class and it is direct child class of "Object class" it implements **Serializable** and **Comparable**.

values() method: Every enum implicitly contains a static values() method to list all constants of enum.

Example: Beer[] b=Beer.values();

ordinal() method: Within enum the order of constants is important we can specify by its ordinal value.

- We can find ordinal value(index value) of enum constant by using ordinal() method.

Example: public int ordinal();

Example:

```
enum Beer
{
KF,KO,RC,FO;
}
class Test{
public static void main(String args[]){
Beer[] b=Beer.values();
for(Beer b1:b)//this is forEach loop.
{
System.out.println(b1+"......."+b1.ordinal());
}}}
```

Output:

```
D:\Enum>java Test
KF.......0
KO.......1
RC.......2
FO.......3
```

Specialty of java enum: When compared with old languages enum java's enum is more powerful because in addition to constants we can take normal variables, constructors, methods etc which may not possible in old languages.

- Inside enum we can declare main method and even we can invoke enum directly from the command prompt.

Example:

```
enum Fish{
GOLD,APOLO,STAR;
public static void main(String args[]){
System.out.println("enum main() method called");
}}
```

Output:

```
D:\Enum>java Fish
enum main() method called
```

- In addition to constants if we are taking any extra members like methods then the list of constants should be in the 1st line and should ends with semicolon.
- If we are taking any extra member then enum should contain at least one constant. Any way an empty enum is always valid.

Example:

```
enum X{
A,B,C;//here semicolon mandatory.
public void methodOne(){
}
}          (valid)
```

```
enum X{
public void methodOne(){
}
A,B,C;
}
      (invalid)
```

```
enum X
{
public void methodOne(){
}
}
```
(invalid)

```
enum X
{

}
```
(valid)

```
enum X
{
  ;
public void methodOne(){
}
}
```
(valid)

Enum vs constructor: Enum can contain constructor. Every enum constant represents an object of that enum class which is static hence all enum constants will be created at the time of class loading automatically and hence constructor will be executed at the time of enum class loading for every enum constants.

Example:

```
enum Beer{
KF,KO,RC,FO;
Beer(){
System.out.println("Constructor called.");
}
}
class Test{
public static void main(String args[]){
Beer b=Beer.KF;
System.out.println("hello.");
}}
```

Output:

```
D:\Enum>java Test
Constructor called.
Constructor called.
Constructor called.
Constructor called.
Hello.
```

- We can't create enum object explicitly and hence we can't invoke constructor directly.

Example:

```
enum Beer{
KF,KO,RC,FO;
Beer(){
System.out.println("constructor called");
}
}
class Test{
public static void main(String args[]){
Beer b=new Beer();
System.out.println(b);
}}
```

Output:

```
Compile time error.
D:\Enum>javac Test.java
Test.java:9: enum types may not be instantiated
Beer b=new Beer();
```

Example:

```
KF==>public static final Beer KF=new Beer();
KF(100)==>public static final Beer KF=new Beer(100);
```

```
enum Beer
{
KF(100),KO(70),RC(65),Fo(90),KALYANI;
int price;
Beer(int price){
this.price=price;
}
Beer()
{
this.price=125;
}
public int getPrice()
{
return price;
}
}
```

```
class Test{
public static void main(String args[]){
Beer[] b=Beer.values();
for(Beer b1:b)
{
System.out.println(b1+"......."+b1.getPrice());
}}}
```

- Inside enum we can take both instance and static methods but it is not possible to take abstract methods.

Case 1:

- Every enum constant represents an object hence whatever the methods we can apply on the normal objects we can apply the same methods on enum constants also.

Which of the following expressions are valid?

1) Beer.KF==Beer.RC----------> false
2) Beer.KF.equals(Beer.RC) ---------->false
3) Beer.KF<Beer.RC---------->invalid
4) Beer.KF.ordinal()<Beer.RC.ordinal()------>valid

Case 2:

Example 1:

```
package pack1;
public enum Fish
{
STAR,GUPPY;
}
```

Example 2:

```
package pack2;
//import static pack1.Fish.*;
import static pack1.Fish.STAR;
class A
{
public static void main(String args[]){
System.out.println(STAR);
}
}
```

1) Import pack1.*; ---------->invalid
2) Import pack1.Fish; ---------->invalid
3) import static pack1.Fish.*; ---------->valid
4) import static pack1.Fish.STAR; ---------->valid

Example 3:

```
package pack3;
//import pack1.Fish;
import pack1.*;
//import static pack1.Fish.GUPPY;
import static pack1.Fish.*;
class B
{
public static void main(String args[]){
Fish f=Fish.STAR;
System.out.println(GUPPY);
}
}
```

Case 3:

```
enum Color
{
BLUE,RED
{
public void info(){
System.out.println("Dangerous color");
}
},GREEN;
public void info()
{
System.out.println("Universal color");
}}
class Test{
public static void main(String args[]){
Color[] c=Color.values();
```

```
for(Color c1:c)
{
c1.info();
}}}
```

Output:

```
Universal color
Dangerous color
Universal color
```

Regular expression

- A Regular Expression is a expression which represents a group of Strings according to a particular pattern.

Example:

- We can write a Regular Expression to represent all valid mail ids.
- We can write a Regular Expression to represent all valid mobile numbers.

The main important application areas of Regular Expression are:

- To implement validation logic.
- To develop Pattern matching applications.
- To develop translators like compilers, interpreters etc.
- To develop digital circuits.
- To develop communication protocols like TCP/IP, UDP etc.

Example:

```
import java.util.regex.*;
class RegularExpressionDemo
{
    public static void main(String[] args)
    {
            int count=0;
            Pattern p=Pattern.compile("ab");
            Matcher m=p.matcher("abbbabbaba");
            while(m.find())
            {
                count++;
                System.out.println(m.start()+"------"+m.end()+"------"+m.group());
            }
            System.out.println("The no of occurences :"+count);
    }
}
```

Output:

```
0------2------ab
4------6------ab
7------9------ab
The no of occurrences: 3
```

Pattern class:

- A Pattern object represents “compiled version of Regular Expression”.
- We can create a Pattern object by using compile() method of Pattern class.

public static Pattern compile(String regex);

Example:

Pattern p=Pattern.compile("ab");

Note: if we refer API we will get more information about pattern class.

Matcher:

- A Matcher object can be used to match character sequences against a Regular Expression. We can create a Matcher object by using matcher() method of Pattern class.

public Matcher matcher(String target);

Matcher m=p.matcher("abbbabbaba");

Important methods of Matcher class:

1) boolean find();

- It attempts to find next match and returns true if it is available otherwise returns false.

2) int start();

- Returns the start index of the match.

3) int end();

- Returns the offset(equalize) after the last character matched.(or)
- Returns the end index of the matched.

4) String group();

- Returns the matched Pattern.

Note: Pattern and Matcher classes are available in **java.util.regex** package.

Character classes:

[abc]----------Either ‘a’ or ‘b’ or ‘c’

[^abc] ----------Except 'a' and 'b' and 'c'
[a-z] ----------Any lower case alphabet symbol
[A-Z] ----------Any upper case alphabet symbol
[a-zA-Z] ----------Any alphabet symbol
[0-9] ----------Any digit from 0 to 9
[a-zA-Z0-9] ----------Any alphanumeric character

Example:

```
import java.util.regex.*;
class RegularExpressionDemo
{
	public static void main(String[] args)
	{
		Pattern p=Pattern.compile("x");
		Matcher m=p.matcher("a1b7@z#");
		while(m.find())
		{
			System.out.println(m.start()+"-------"+m.group());
		}
	}
}
```

Output:

x=[abc]	x=[^abc]	x=[0-9]	x=[a-z]
0-------a	1-------1	1-------1	0-------a
2-------b	3-------7	3-------7	2-------b
	4-------@		5-------z
	5-------z		
	6-------#		

Predefined character classes:

\s----------space character
\d----------Any digit from o to 9[o-9]
\w----------Any word character[a-zA-Z0-9]
. ----------Any character including special characters.

Example:

```
import java.util.regex.*;
class RegularExpressionDemo
{
	public static void main(String[] args)
	{
		Pattern p=Pattern.compile("x");
		Matcher m=p.matcher("a1b7@z#");
		while(m.find())
		{
			System.out.println(m.start()+"-------"+m.group());
		}
	}
}
```

Output:

x=\\s	x=\\d	x=\\w	x=.
4-------	1-------1	0-------a	0-------a
	3-------7	1-------1	1-------1
		2-------b	2-------b
		3-------7	3-------7
		6-------z	4-------
			5-------@
			6-------z
			7-------#

Quantifiers:

- Quantifiers can be used to specify no of characters to match.

a----------Exactly one 'a'
a+----------At least one 'a'
a*----------Any no of a's including zero number
a? ----------At most one 'a'

Example:

```
import java.util.regex.*;
class RegularExpressionDemo
{
	public static void main(String[] args)
	{
		Pattern p=Pattern.compile("x");
		Matcher m=p.matcher("abaabaaab");
		while(m.find())
		{
			System.out.println(m.start()+"-------"+m.group());
		}
	}
}
```

Output:

x=a	x=a+	x=a*	x=a?
0-------a	0-------a	0-------a	0-------a
2-------a	2-------aa	1-------	1-------
3-------a	5-------aaa	2-------aa	2-------a
5-------a		4-------	3-------a
6-------a		5-------aaa	4-------
7-------a		8-------	5-------a
		9-------	6-------a
			7-------a
			8-------
			9-------

Pattern class split() method:

- Pattern class contains split() method to split the given string against a regular expression.

Example 1:

```
import java.util.regex.*;
class RegularExpressionDemo
{
	public static void main(String[] args)
	{
		Pattern p=Pattern.compile("\\s");
		String[] s=p.split("bhaskar software solutions");
		for(String s1:s)
		{
			System.out.println(s1);//bhaskar
					       //software
					       //solutions
		}
	}
}
```

Example 2:

```
import java.util.regex.*;
class RegularExpressionDemo
{
	public static void main(String[] args)
	{
		Pattern p=Pattern.compile("\\."); (or)[.]
		String[] s=p.split("www.dugrajobs.com");
		for(String s1:s)
		{
			System.out.println(s1);//www
					       //dugrajobs
					       //com
		}
	}
}
```

String class split() method:

- String class also contains split() method to split the given string against a regular expression.

Example:

```
import java.util.regex.*;
class RegularExpressionDemo
{
	public static void main(String[] args)
	{
		String s="www.durgajobs.com";
		String[] s1=s.split("\\.");
		for(String s2:s1)
		{
			System.out.println(s2);//www
									//durgajobs
									//com
		}
	}
}
```

Note:

- String class split() method can take regular expression as argument where as pattern class split() method can take target string as the argument.

StringTokenizer:

- This class present in java.util package.
- It is a specially designed class to perform string tokenization.

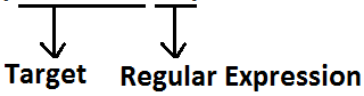
Example 1:

```
import java.util.*;
class RegularExpressionDemo
{
	public static void main(String[] args)
	{
		StringTokenizer st=new StringTokenizer("durga software solutions");
		while(st.hasMoreTokens())
		{
			System.out.println(st.nextToken());//durga
								//software
								//solutions
		}
	}
}
```

- The default regular expression for the StringTokenizer is space.

Example 2:

```
import java.util.*;
class RegularExpressionDemo
{
	public static void main(String[] args)
	{
		StringTokenizer st=new StringTokenizer("1,99,988",",");
```




```
		//StringTokenizer st=new StringTokenizer("1,99,988",",");
		while(st.hasMoreTokens())
		{
			System.out.println(st.nextToken());//1
								//99
								//988
		}
	}
}
```

Requirement: Write a regular expression to represent all valid identifiers in yava language.

Rules:

The allowed characters are:

1) a to z, A to Z, 0 to 9, -,#
2) The 1st character should be alphabet symbol only.
3) The length of the identifier should be at least 2.

Program:

```
import java.util.regex.*;
class RegularExpressionDemo
```

```
{
    public static void main(String[] args)
    {
        Pattern p=Pattern.compile("[a-zA-Z][a-zA-Z0-9-#]+"); (or)
        Pattern p=Pattern.compile("[a-zA-Z][a-zA-Z0-9-#][a-zA-Z0-9-#]*");
        Matcher m=p.matcher(args[0]);
        if(m.find()&&m.group().equals(args[0]))
        {
            System.out.println("valid identifier");
        }
        else
        {
            System.out.println("invalid identifier");
        }

    }
}
```

Output:

```
E:\scjp>javac RegularExpressionDemo.java
E:\scjp>java RegularExpressionDemo bhaskar
Valid identifier
E:\scjp>java RegularExpressionDemo ?bhaskar
Invalid identifier
```

Requirement: Write a regular expression to represent all mobile numbers.

Rules:

1) Should contain exactly 10 digits.
2) The 1st digit should be 7 to 9.

Program:

```
import java.util.regex.*;
class RegularExpressionDemo
{
    public static void main(String[] args)
    {
        Pattern p=Pattern.compile("[7-9][0-9][0-9][0-9][0-9][0-9][0-9][0-9][0-9][0-9]");
        //Pattern p=Pattern.compile("[7-9][0-9]{9}");
        Matcher m=p.matcher(args[0]);
        if(m.find()&&m.group().equals(args[0]))
        {
            System.out.println("valid number");
        }
        else
        {
            System.out.println("invalid number");
        }
    }
}
```

Analysis:

10 digits mobile:

[7-9][0-9][0-9][0-9][0-9][0-9][0-9][0-9][0-9][0-9] (or)
[7-9][0-9]{9}

Output:

```
E:\scjp>javac RegularExpressionDemo.java
E:\scjp>java RegularExpressionDemo 9989308279
Valid number
E:\scjp>java RegularExpressionDemo 6989308279
Invalid number
```

10 digits (or) 11 digits:

(0?[7-9][0-9]{9})

Output:

```
E:\scjp>javac RegularExpressionDemo.java
E:\scjp>java RegularExpressionDemo 9989308279
Valid number
E:\scjp>java RegularExpressionDemo 09989308279
Valid number
E:\scjp>java RegularExpressionDemo 919989308279
Invalid number
```

10 digits (0r) 11 digit (or) 12 digits:

(0|91)?[7-9][0-9]{9} (or)
(91)?(0?[7-9][0-9]{9})

```
E:\scjp>javac RegularExpressionDemo.java
E:\scjp>java RegularExpressionDemo 9989308279
Valid number
E:\scjp>java RegularExpressionDemo 09989308279
Valid number
E:\scjp>java RegularExpressionDemo 919989308279
Valid number
E:\scjp>java RegularExpressionDemo 69989308279
Invalid number
```

Requirement: Write a regular expression to represent all Mail Ids.

Program:

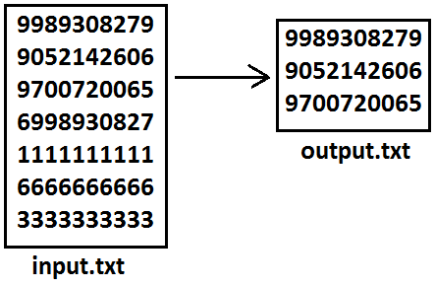
```
import java.util.regex.*;
class RegularExpressionDemo
{
	public static void main(String[] args)
	{
		Pattern p=Pattern.compile("[a-zA-Z][a-zA-Z0-9-.]*@[a-zA-Z0-9]+([.][a-zA-Z]+)+");
		Matcher m=p.matcher(args[0]);
		if(m.find()&&m.group().equals(args[0]))
		{
			System.out.println("valid mail id");
		}
		else
		{
			System.out.println("invalid mail id");
		}
	}
}
```

Output:

```
E:\scjp>javac RegularExpressionDemo.java
E:\scjp>java RegularExpressionDemo bhaskar86.vaka@gmail.com
Valid mail id
E:\scjp>java RegularExpressionDemo 999bhaskar86.vaka@gmail.com
Invalid mail id
E:\scjp>java RegularExpressionDemo 999bhaskar86.vaka@gmail.co9
Invalid mail id
```

Requirement: Write a program to extract all valid mobile numbers from a file.

Diagram:



Program:

```
import java.util.regex.*;
import java.io.*;
class RegularExpressionDemo
{
	public static void main(String[] args)throws IOException
	{
		PrintWriter out=new PrintWriter("output.txt");
		BufferedReader br=new BufferedReader(new FileReader("input.txt"));
		Pattern p=Pattern.compile("[7-9][0-9]{9}");
		String line=br.readLine();
		while(line!=null)
		{
			Matcher m=p.matcher(line);
			while(m.find())
			{
```

```
                                        out.println(m.group());
                                }
                                line=br.readLine();
                        }
                        out.flush();
        }
}
```

Requirement: Write a program to extract all Mail IDS from the File.
Note: In the above program replace mobile number regular expression with MAIL ID regular expression.
Requirement: Write a program to display all .txt file names present in **E:\scjp** folder.
Program:

```
import java.util.regex.*;
import java.io.*;
class RegularExpressionDemo
{
        public static void main(String[] args)throws IOException
        {
                int count=0;
                Pattern p=Pattern.compile("[a-zA-Z0-9-$.]+[.]txt");
                File f=new File("E:\\scjp");
                String[] s=f.list();
                for(String s1:s)
                {
                        Matcher m=p.matcher(s1);
                        if(m.find()&&m.group().equals(s1))
                        {
                                count++;
                                System.out.println(s1);
                        }
                }
                System.out.println(count);
        }
}
```

Output:

```
input.txt
output.txt
outut.txt
3
```