
Sun Certified
Programmer for
java 1.5
Be Competitive
Raj

SUN CERTIFIED PROFESSIONAL FOR JAVA 1.5

CONTENTS:

1

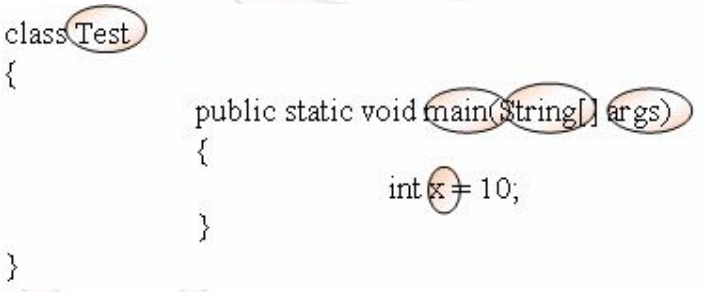
LANGUAGE FUNDAMENTALS

1) Identifiers
2) Keywords
3) Datatypes
4) Literals
5) Arrays
6) Types of variables
7) var – arg methods
8) command line arguments & main method
9) java coding standards

Identifiers

A name in the program is an identifier it may be class name or method name,variable name or label name.

Ex:-



Rules for defining Identifiers

1) A java identifier is a sequence of characters, where each character may be a letter from *a-z* or *A-Z* or a digit form *0-9* or currency symbol *$* or connecting punctuation – , if we are using any other symbol we will get Compile time error "*IllegalCharacter*".
2) Identifier should not be starts with digit.
3) There is no length limit for java identifiers but it is not recommended to take more than 15 length.
4) Java Identifiers are case sensitive

Ex:

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

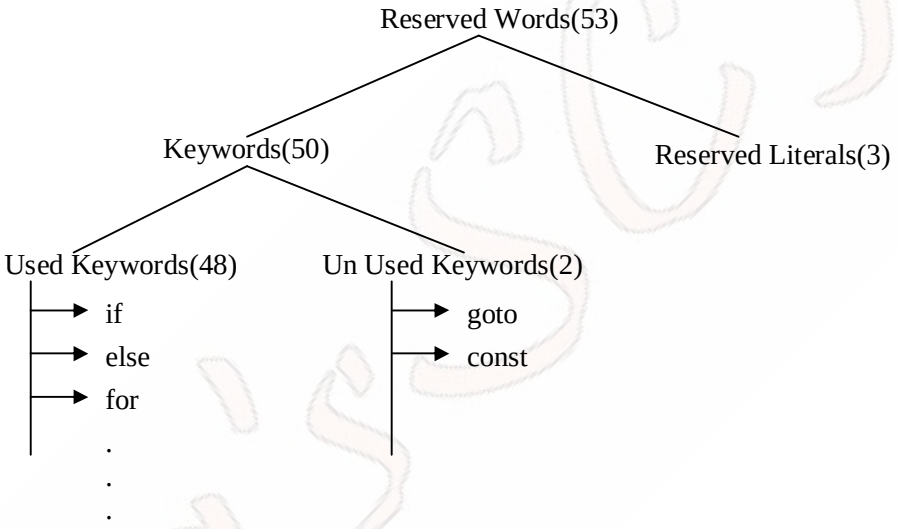
O/P:- 10

Which of the following are valid Identifiers

1) total# ✗
2) all@hands ✗
3) 123total ✗
4) break ✗
5) String ✓
6) total_number ✓
7) $_$ ✓
8) cah ✓

Reserved Words

Some identifiers are reserved to associate some functionality or to represent values, such type of reserved identifiers are called "*ReservedWords*".



Keywords for Datatypes

byte	float
short	double
int	char
long	boolean

Keywords for FlowControl

if	do	break
else	while	continue
switch	for	return
case		
default		

Keywords for ExceptionHandling

try
catch
finally
throw
throws
assert ⟶ 1.4 Version (Debugging

Keywords for Modifers

public	native
private	synchronized
protected	volatile
final	transient
abstract	strictfp
static	

Class Related Keywords

class
interface
package
extends
implements
import

Object Related Keywords

new
instanceof
super
this

void return type Keywords

if a method doesn't return anything compulsory that method should be with void return type

UnUsed Keywords

goto → in java usage is considered as harmful.
const → alternatively we should use final keyword.

Enum Keyword

This keyword has introduced in 1.5 version, to define user defined data types.

Ex:

```
enum Month
{
        JAN,FEB,MAR,....DEC;
}
```

```
enum Beer
{
        KO,KF,RC,FO,BR;
}
```

Reserved Literals

true
false } Allowed values for boolean data types

null → Default value for object reference

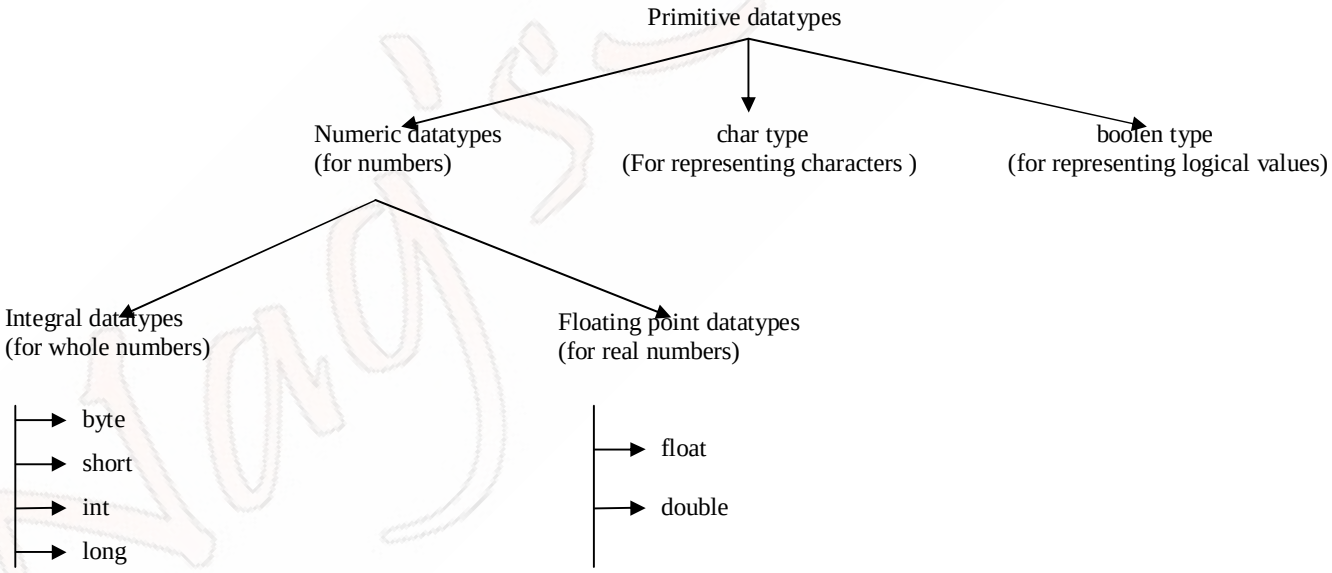
Which of the following are valid java Reserved words ?

1) int, float, signed, double ✗
2) abstract, final, volatile, virtual ✗
3) new, delete ✗
4) goto, constant, static ✗
5) byte, short, int, long ✓

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

Datatypes

In java every variable has a type, every expression has a type and all types are strictly defined.
All the assignments should be checked by the compiler for the type compatibility. Hence java language considers as strongly typed language.
Java is not considered as pure object oriented programming language because several OOP features(like multiple inheritance, operator overloading) are not supported by java. Even java contains non-object primitive datatypes.



Except boolean and char all the remaining datatypes are signed datatypes i.e we can represent both +ve and –ve numbers.

byte

Size : 8-bits
Range: -128 to 127

-ve numbers can represented in 2's compliment form.

Ex:

```
byte b = 10;
byte b = 127;
byte b = 130;   → C.E: possible loss of precision
```

byte b = true; → C.E: Incompatible types found: boolean
required: byte

byte datatype is best suitable if we are handling data either from file or form network.

short

size = 2 bytes
range = -2^{15} to 2^{15} -1 (-32768 to 32767)

Ex:

short s = 10;
short s = 32767;
short s = 65535;→ C.E: possible loss of precision.
short s = true; → C.E: Incompatible types

short is best suitable datatype for 16-bit process. But currently these are completely out dated and hence the corresponding datatypes also no one is using.

int

The most commonly used datatype is int.

size = 4 bytes
range = -2^{31} to $2^{31} - 1$(-2147483648 to 2147483747)

The size of int is always fixed irrespective of platform hence the chance of failing java program is very less if u r changing the platform hence the java is considered as Robust.

long

if int is not enough to hold big values then we should go for long-datatype

size = 8 bytes
range = -2^{63} to $2^{63} - 1$
Ex:
The amount of distance traveled by light in 1000days can be represented by long datatype only and int is not enough.

floating-point

for representing real numbers(numbers with decimal points)

floating –point datatypes

float	double
size = 4 bytes range = -3.4e38 to 3.4e38 If 6 to 7 decimal places of accuracy required Considered as single precision	size = 8 bytes range = -1.7e308 to 1.7e308 If 14 to 15 decimal places of accuracy required Considered as double precision

boolean datatye

size = not applicable(virtual machine dependent).
range = not applicable but allowed values are true/false.

Which of the following boolean declarations are valid

boolean b1 = true; ✓
boolean b2 = 0; ✗ → Incompatible types found:int required : boolean
boolean b3 = TRUE; ✗ →capital TRUE is not valid.

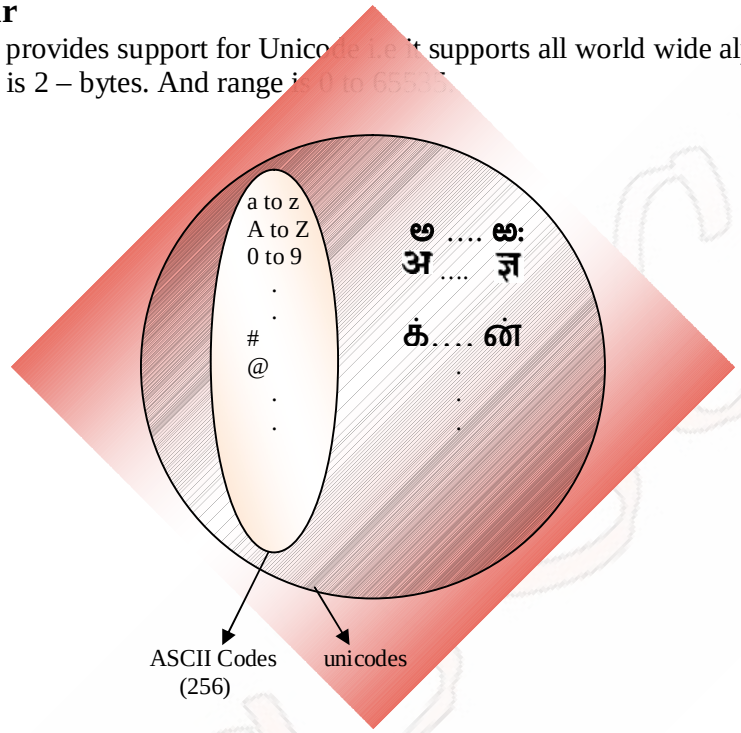
Ex:

```
int x = 0;
if(x)
{
        System.out.println("Hello");
}
else
{
        System.out.println("Hai");
}
```

C.E: Incompatible types found :int
required: boolean.

char

java provides support for Unicode i.e it supports all world wide alphabets. Hence the size of char in java is 2 – bytes. And range is 0 to 65535

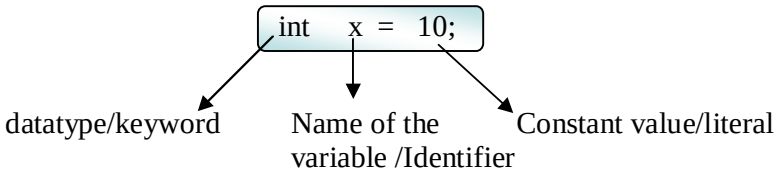


Comparison table for java primitive datatypes

datatype	size	range	Default value	Wrapper class
byte	1 byte	-128 to 127	0	Byte
short	2 bytes	-32768 to 32767	0	Short
int	4 bytes	-2^{31} to $2^{31}-1$	0	Integer
long	8 bytes	-2^{63} to $2^{63}-1$	0	Long
float	4 bytes	-3.4e38 to 3.4e38	0.0	Float
double	8 bytes	-1.7e308 to 1.7e308	0.0	Double
boolean	NA	NA(But allowed values are true, false)	false	Boolean
char	2 bytes	0 to 65535	0(balnk spaces)	Character

Literals

A literal represents a constant value which can be assigned to the variables



Integral Literal

We can specify an integral literal in the following ways.

Decimal literals: allowed digits are 0 to 9

Ex: int x = 10;

Octal literals: allowed digits are 0 to 7 but here literal value should be prefixed with 0(zero)

Ex: int x = 010;

Hexadecimal literals: the allowed digits are 0 to 9, A- F (Both lower, Upper case) literals should be prefixed with 0x or oX

Ex: int x = 0x10;

Ex:

```
class Test
{
    public static void main(String arg[])
    {
        int x = 10;
        int y = 010;
        int z = 0x10;
        System.out.println(x + "..." + y + "..." + z);
    }
}
```

O/P:- 10…8…16

Except decimal, octal, hexadecimal there is no other way to represents constant values for the integral datatype.

By default every integral lateral is of int datatype we can specify explicitly. An integral literal is of long type by suffixing with l or L.

Ex:

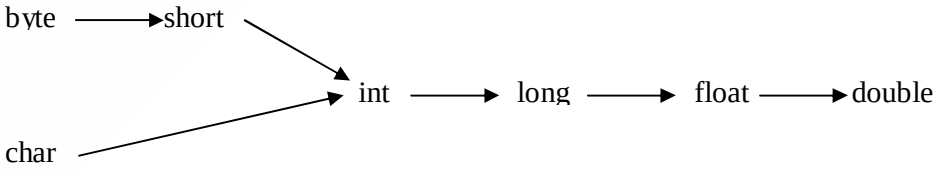
10 → int value.

10l → long value.

```
long l = 10l;
int i = 10l;
```

C.E: possible loss of precision found : long

Required:int



There is no way to specify explicitly an integral literal is of type byte and short.

If the integral literal is with in the range of byte then the JVM by default treats it as byte literal. Similarly short literal also.

Floating – point literals

By default floating-point literals are double type we can specify explicitly as float type by suffixing with ‘f’ or ‘F’.

Which of the following are valid declarations

A. float f = 10.5; ✗ → C.E possible loss of precision
B. float f = 10.5f; ✓
C. double d = 10.5; ✓
D. double d = 10.5f; ✓
E. double d = 10.5D; ✓

we can specify explicitly a floating point literal of double type by suffixing with d or D.
we can also represent float-point literals by using scientific notation.

Ex:

```
double d = 10e23;
int i = 10e23;  → C.E possible loss of precision found : double
                                    required : int.
```

Floating point literals can be specified only in decimal form. i.e we can't use octal and hexa decimal representation for floating point literals.

Ex:

```
Double d = 0x123.456;
C.E: Malformed floating-point literal.
```

Which of the following are valid declarations

A. float f = 123.456; ✗
B. float f = 0x123.456F; ✓
C. float f = 0x123; ✓
D. float f = 1.2e36; ✗
E. double d = 1.2e36; ✓

Boolean Literals

The only allowed values for boolean datatype are true, false.

Which of the following are valid declarations

1. boolean b = true; ✓
2. boolean b = FALSE; ✗
3. boolean b = 0; ✗

character literal

A char literal can be represented as a single character with in single quotes.

Ex:

```
char ch = 'a';  ✓
char ch = 'ab'; ✗   C.E: unclosed character literal.
char ch = a;    ✗
```

we can represent a char literal by using it's Unicode value. For the allowed Unicode values are 0 to 65535.

Ex:

```
char ch = 97;
System.out.println(ch);          → O/P: a
char ch = 65535;
char ch = 65536;        → C.E : possible loss of precision found : int
                                          required :char
```

we can represent a char literal by using Unicode representation which is nothing but
\uxxxx'

Ex:

```
char ch = '\u0061'
System.out.println(ch);          → O/P:a
char ch = '\ubeef';   ✓
```

char ch = '\uface'; ✓
char ch = '\iface'; ✗
char ch = '\uface'; ✗

we can also represent a char literal by using escape character.

Ex:

char ch = '\b'; ✓
char ch = '\n'; ✓
char ch = '\l'; ✗

The following is the list of all possible escape characters in java.

\b → backspace
\n → new line
\r → carriage return
\f → formfeed
\t → horizontal tab
\' → single quote
\" → double quote
\\ → back slash

Which of the following char declarations are valid?

char ch = 'c'; ✓
char ch = 'abc'; ✗
char ch = a ; ✗
char ch = 65123; ✓
char ch = 65537; ✗
char ch = \uabcd; ✗
char ch = '\uanuska'; ✗
char ch = '\ubeef'; ✓
char ch = '\r'; ✓
char ch = '\t'; ✓
char ch = '\d'; ✗

String literal

A sequence of character with in double quotes is String literal.

String s = "Durga"; ✓
String s = 'software'; ✗
String s = 'a'; ✗

Arrays

Introduction

An array is a data structure that represents an index collection of fixed no of homogeneous data elements. The main advantage of arrays is we can represent a group of values with single name hence readability of the code will be improved. The main limitation of arrays is they are fixed in size. i.e once we constructed an array there is no chance of increasing or decreasing bases on our requirement hence with respect to memory arrays shows worst performance we can overcome this problem by using collections.

Array Declaration

The following are the ways to declare an array.

1) int [] a;
2) int a[];
3) int [] a;

The first one is recommended to use because the type is clearly separated from the name.
At the first time of declarations we are not allowed to specify the size. Violation leads to C.E.

Ex:

int[] a; ✓
int[6] a; ✗

Declaring Multidimensional Arrays

The following are the valid declarations for multidimensional arrays.

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

we can specify the dimension before name of variable also, but this facility is available only for the first variable.

int[] a[],b[]; ✓
int[] []a,[]b; ✗

Which of the following declarations are valid?

i. int[2][3] a; ✗
ii. int[] a,b; ✓
iii. int[] a[],b[]; ✓
iv. int []a,[]b; ✗
v. int []a,b[]; ✓

Construction of Arrays

Single Dimension : Arrays are internally implemented as object hence by using new operator we can construct an array.
Compulsory at the time of construction we should specify the size otherwise compile time error.

Ex:

int[] a = new int[10]; → ✓
int[] a = new int[]; → ✗ C.E

It is legal to have an error with size 0 there is no C.E or R.E

int[] a = new int[0];

If we are specifying array size with some –ve integer

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

we will get R.E saying NegativeArraySizeException.

The only allowed Data type to allow the size are byte, short, char, int. if we are using any other datatype we will get a C.E.

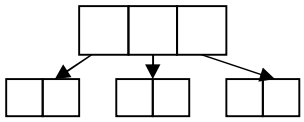
int[] a = new int[10];
int[] a1 = new int[10l]; ✓
int[] a = new int[10l]; → C.E possible loss of precision found: long
required: int
int[] a = new int[10.5];→ C.E
int[] a = new int[true]; → C.E Incompatible types found : boolean required:int.

The maximum allowed Array Size in java is 2147483648.

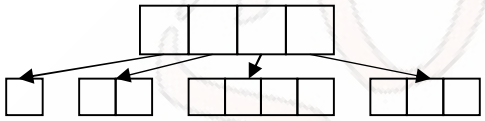
Multi Dimension: In java multidimensional arrays are implemented as single dimension arrays. This approach improves performance with respect to memory.

Ex:

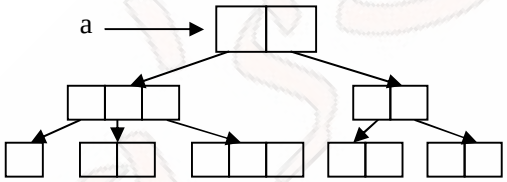
int[][] a = new int[3][2];



int[][] a = new int[4][]
a[0] = new int[1];
a[1] = new int[2];
a[2] = new int[4];
a[3] = new int[3];

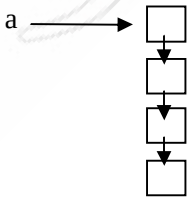


How to declare an array for the following diagrams?



Ans:

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



Ans: int [][][][]a = new int[1][1][1][1];

Which of the following Array declarations are valid?

int [][] a = new int[3][4]; → ✓
int [][] a = new int[3][]; → ✓
int [][] a = new int[][4]; → ✗

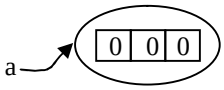
```
int [][] a = new int[][];          →  ✗
int [][][] a = new int[3][4][];    →  ✓
int [][][] a = new int[3][][5];    →  ✗
```

Initialization of arrays

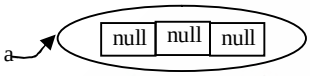
Once we created an array all it's elements initialized with default values.

Ex:

```
int[] a = new int[3];
System.out.println(a[0]);      → O/P: 0
System.out.println(a);         → O/P: [I@10b62c9
```




```
int[][] a = new int[3][2];
System.out.println(a);         → [I@10b62c9
System.out.println(a[0]);      → [[I@82ba41
System.out.println(a[0][0]);   → 0

int[][] a = new int[3][];
System.out.println(a);         → [I@10b62c9
System.out.println(a[0]);      → null
System.out.println(a[0][0]);   → NullPointerException
```



Once we created an array all it's elements are initialized with default values.
If we are providing any explicit initialization then default values will be overridden with our provided values.

Ex:

```
int[] a = new int[3];
a[0] = 10;
a[1] = 20;
System.out.println(a[0] + "---" + a[1] + "---" + a[2]);
a[10] = 100;   → R.E: ArrayIndexOutOfBoundsException.
a[-10] = 100;  → R.E: ArrayIndexOutOfBoundsException.
a[10.5] = 100; → C.E: PossibleLossOfPrecision found : double
                                                required : int
```

when ever we are trying to access an array with int index which is not in valid range then we will get runtime exception saying "ArrayIndexOutOfBoundsException". But there is no C.E.
If we are trying to access an array index with the following datatype we will get C.E.
float, double, long, boolean.

Declaration Construction and Initialization in a single line

```
int []a;
a = new int[3];
a[0] = 10;
a[1] = 20;
a[2] = 30;
```

All these statements we can replace with a single line as follows.

```
int[] a = {10,20,30};
String[] s = {"Chiru","Allu","Ram","Akil"}
```

If we want to use the above shortcut technique compulsory we should perform declaration, construction initialization in a single line only. If we are dividing into 2 lines we will get C.E.

Ex:

```
int[] a;
a = {10,20,30,40};
C.E: illegal start of expression.

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

length Vs length();

length: 1) It is the final variable applicable for array objects.
2) It represents the size of the array.

Ex:

```
int [] a = new int[5];
System.out.println(a.length());        → C.E
System.out.println(a.length);          → 6
```

length(): 1) It is the final method applicable only for String Objects.
2) It represents the no of characters present in the String.

Ex:

```
String s = "raju";
System.out.println(s.length);          → C.E
System.out.println(s.length());        → 4
```

In the case of Multidimensional array length variable always represent base size but not total no of elements.

Ex:

```
int[][] a = new [3][2];
System.out.println(a.length);
System.out.println(a[0].length);
```

There is no variable which represents the total no of elements present in multidimensional arrays.

Anonymous Arrays

Some times we can declare an array with out name also such type of arrays are called anonymous arrays.

The main objective of anonymous arrays is just for temporary usage. We can create an anonymous arrays as follows.

Ex:

```
new int[] {10,20,30}  → X

class Test
{
        public static void main(String arg[])
        {
                System.out.println(sum(new int[]{10,20,30,40}));
        }
        public static int sum(int[] a)
        {
                int total = 0;
                for(int i = 0; i< a.length;i++)
                {
                        total = total + a[i];
                }
                return total;
        }
}
```

Array Element Assignments

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

Ex:

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

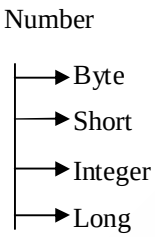
in this case the following datatypes are allowed as array elements.

byte, short, int, char.

```
a[0] = 10;
a[1] = 'a';
byte b = 20;
a[2] = b;
a[3] = 10.5;    → possible loss of precision. found : double
                                              required: int
```

in the case of object arrays as array elements we can provide either declared type object or it's child class objects.

```
Number[] n = new Number[6];
n[0] = new Integer(10);          ✓
n[1] = new Long(10l);            ✓
n[2] = new String("raju");       ✗ Incompatible types found : String.
                                   required : Number.
```



If we declare an array of interface type we are allowed to provide it's implementation class object as elements.

```
Runnable[] = new Runnable[]
r[0] = new Thread();
```

Array Variable Assignments

A char element can be promoted as the int element but a char array can't be prompted to int array.

Ex:

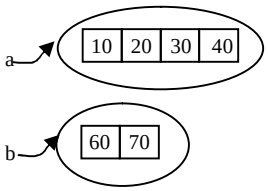
```
int [] a = new int[6];
int [] b = a;
char[] ch = {'a','b','c'};
int [] c = ch;    → Incompatible types found : char[]
                                  required:int[]
```

which of the following promotions are possible?

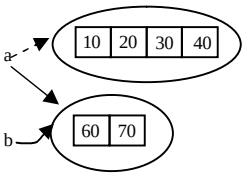
```
int      → double     ✓
float    → long       ✗
int[]    → float[]    ✗
char[]   → int[]      ✗
boolean  → int        ✗
int      → Boolean    ✗
```

when ever we are assigning one array to another array compiler will check only the types instead of sizes

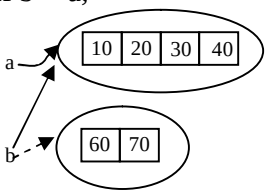
```
int[] a = {10,20,30,40};
int[] b = {60,70};
```



If a = b;



If b = a;



See the following example.

Ex:

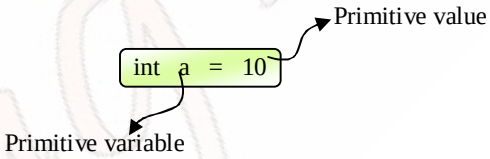
```
int [][] a = new int[3][];
a[0] = new int[4];
a[1] = new int[4][5];   → Incompatible types found: int[][]
                                                  required: int[]
a[2] = 10;              → Incompatible types found: int
                                                  required: int[]
```

Types of Variables

Based on the type of value represented by the variable all the variables are divided into two types.

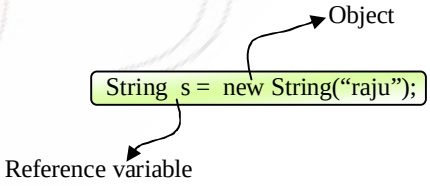
1) primitive variables

Ex:



2) Reference variables

Ex:



Base on the purpose and position of declaration all variables are divided into 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 crated at the time of object creation and will be destroyed at the time of object destruction i.e the scope of instance variables is exactly same as the scope of object.

We have to declare instance variables with in the class but outside of any method (or) constructor or block.

For the instance variables no need to perform initialization. JVM will always provide values.
Ex:

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

Instance variables also known as 'attributes'.

static variable

if the value of a variable is fixed for all objects then it is not recommended to declare that variable at instance level. Such type of variables we have to declare at class level by using static keyword.

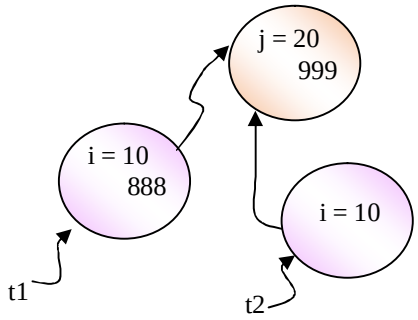
For the static variables a single copy will be created at class level and shared by all objects of that class.

Static variables should be declared with in the class but outside of any method or block or constructor.
Static variables will be created at the time of class loading and destroyed at the time of unloading.

We can access static variables either by using class name or by using object reference using class name is recommended.
Ex:

```
class Test
{
        int i;
        static int j = 10;
        public static void main(String arg[])
        {
                Test t1 = new Test();
                t1.i = 888;
                t1.j = 999;
                Test t2 = new Test();
                System.out.println(t2.i+" ----"+t2.j);
        }
}
```

O/P:- 10----999



For the static variables no need to perform initialization JVM will always provide default values.

Ex:

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

LocalVariables

If we are declaring a variable with in a method or constants or block such type of variables are called local variables. Local variables also known as *temporary variable* or '*automatic variable*'. Local variables will be created as the part of method execution and will be destroyed once the method completes.

For the local variables JVM won't provide any default values. Before using a local variable compulsory we should perform initialization explicitly otherwise compile time error.

Ex:

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

C.E: variable 'i' might not have been initialized.

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

O/P:- Hello

Because we didn't use 'i'.

Ex:

```
class Test
{
    public static void main(String arg[])
    {
        int i;
```

```
                if(arg.length > 0)
                {
                        i = 10;
                }
                System.out.println(i);
        }
}
```

C.E: variable 'i' might not have been initialized because we may give or not arguments at runtime so before that compiler will generate error.

Ex:

```
class Test
{
        public static void main(String arg[])
        {
                int i;
                if(arg.length > 0)
                {
                        i = 10;
                }
                else
                {
                        i = 20;
                }
                System.out.println(i);
        }
}
```

O/P:- Because if we give arguments 10 will be initialized other wise 20 will be initialized. It is not good programming practice to perform initialization in logical blocks for local variables because they may not execute at runtime.

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

For the static and instance variables JVM will always provide default values. But for the local variables JVM won't provide default values compulsory we should perform initialization before using.

Un initialized Arrays

Ex:

```
class Test
{
        int[] a;
        public static void main(String arg[])
        {
                Test t = new Test();
                System.out.println(t.a);        → null
                System.out.println(t.a[0]);     → NullPointerException
        }
}
```

Instance Level

1) int[] a;

```
System.out.println(objectref.a)        → null
System.out.println(objectref.a[0])     → NullPointerException.
```

2) int[] a = new int[6];

```
System.out.println(objectref.a);       → [I@123
System.out.println(objectref.a[0]);    → 0;
```

Satic level

1) static int[] a;
   System.out.println(a); → null
   System.out.println(a[0]); → NPE(NullPointerException)
2) static int[] a=new int[6];
   System.out.println(a); → [I@123
   System.out.println(a[0]); → 0

Local level:

1)

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

} C.E: varialbe 'a' might not have been initialized

2)

```
int [] a = new int[6];
System.out.println(a);          → [I@123
System.out.println(a[0])        → 0
```

Note: If we are creating an array all it's elements are automatically initialized with default values. Even though it is local array.

var- arg methods

From 1.5 version on words we are allowed to declare a method with variable no of arguments such type of methods are called var-arg methods.

We can declare a var-arg methoda as follows

```
m1(int… i);
```

this methods is applicable for any no of int arguments including zero no of arguments.

Ex:

```
class Test
{
    public static void m1(int... i)
    {
        System.out.println("var-arg methods");
    }
    public static void main(String arg[])
    {
        m1();          → var-arg methods
        m1(10);        → var-arg methods
        m1(10,20);     → var-arg methods
        m1(10,20,30); → var-arg methods
    }
}
```

'var-arg' methods internally implemented by using single dimensional arrays. Hence we can differentiate arguments by using index.

Ex:

```
class Test
{
    public static void main(String arg[])
    {
        sum(10,20);
        sum(10,20,30,40);
```

```
                sum(10);
                sum();
        }
        public static void sum(int... a)
        {
                int total = 0;
                for (int i=0;i<a.length ;i++ )
                {
                        total = total+a[i];
                }
                System.out.println("The sum is:"+total);
        }
}
```

O/P:-

```
The sum is:30
The sum is:100
The sum is:10
The sum is:0
```

We can mix general parameter with var-arg parameter.

Ex:

m1(char ch, int…a)

If we are mixing general parameters with var-arg parameter then var-arg parameter must be the last parameter otherwise compile time error.

Ex:

m1(int… a, float f) ✗

m1(float f, int… a) ✓

we can't take more than one var-arg parameter in var-arg method otherwise C.E.

m1(int… a double… d) ✗

which of the following are valid var-arg declarations.

m1(int... a) ✓

m1(int. ..a) ✗ C.E: malformed function

m1(char ch, int... a) ✓

m1(int... a, char ch) ✗

m1(int...a, boolean... b) ✓

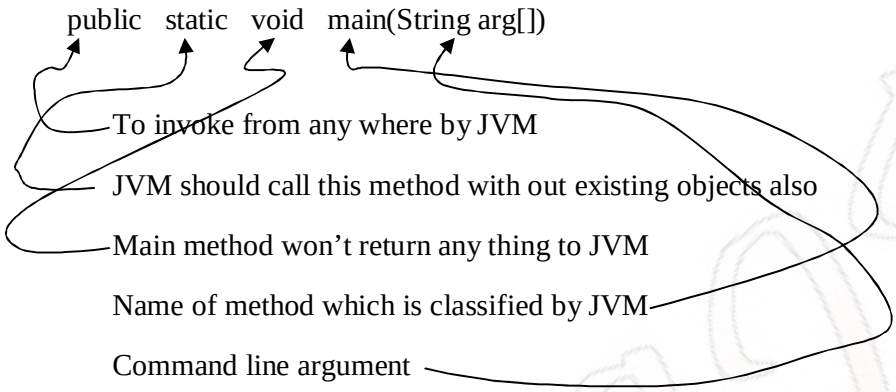
Ex:

```
class Test
{
        public static void m1(int i)
        {
                System.out.println("General method");
        }
        public static void m1(int... i)
        {
                System.out.println("var-arg method");
        }
        public static void m1(String arg[])
        {
                m1();
                m1(10,20);
                m1(10);

        }
}
```

Var-arg method will always get least priority i.e if no other method matched then only var-arg method will be executed.

Command line arguments & main() method

JVM always calls main method to start the program. Compiler is not responsible to check whether the class contain main() or not. Hence if we are not the main method we won't get any C.E. But at runtime JVM raises NoSuchMethodError:main



If we perform any changes in main() we will get NoSuchMethodError:main() at runtime. But no C.E.
The main() can be declared as final and synchronized.

Which of the following main() are valid?

public static void main(String arg){} → ✗
public static void main(String arg[]){return 10;} → ✗
public void main(String arg[]){} → ✗
public static final synchronized main(String arg[]){} → ✗
public static final synchronized void main(String arg[]){} → ✓
public static final void main(String arg[]){} → ✓

Command line arguments

We can send command line argument to the main() by using

arg[0] → first command line argument
arg[0] → first command line argument
-
-
-
args.length → no of command line arguments

write a code to display all command line arguments?

```
for (int i = 0;i < args.length ;i++ )
{
        System.out.println(arg[i]);
}

for (int i = 0;i <= args.length ;i++ )
{
        System.out.println(arg[i]);     → C.E ArrayIndexOutOfBoundsException
}
```

Java coding Standards

Coding standards for classes

Usually class name should be noun. Should starts with upper case letter and if it contain multiple words every inner words also should start with capital letters.

Ex:

String
StringBuffer
NumberFormat
CustomerInformation

Coding standards for Interfaces

Usually interface named should be adjective, starts with capital letters and if it contains multiple words, every inner word also should starts with capital letter.

Ex:

Runnable
Serializable
Clonable
Movable
Transferable
Workable

Coding standards with methods

Values should be either verbs or verb + noun combination.
Starts with lower case and every inner words starts with upper case(this convention is also called camel case convention).

Ex:

getName(), getMessage(), toString(), show(), display().

Coding standards for variables

Usually the variable starts with noun and every inner word should start with upper case i.e camel case convention.

Ex:

Name, rollno, bandwidth, totalNumber.

Coding standards for constants

It should be noun, it should contain only upper case letters and works are separated with underscores.

Ex:

MAX_SIZE, MIN_PRIORITY, COLLEGE_NAME.

Java Been Coding Conventions

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

Ex:

```
public class StudentBeen
{
        private String name;
        public String getName()
        {
                return name;
        }
        public void setName(String name)
        {
                this.name = name;
        }
}
```

Syntax for getterMethod

Compulsory it should be public & should not contain any arguments.

For the non boolean property xxx the following is syntax of getter method

```
public datatype getXxx()
{
        return xxx;
}
```

For the boolean property xxx the following is the syntax

```
public boolean getXxx() or isXxx()
{
        return xxx;
}
```

Syntax of setter Method

It should be public and return type should be void. For any propertyxxx

```
public void setXxx(datatype xxx)
{
        This.xxx = xxx;
}
```

To register a listener

To register myActionListener x which of the following is valid coding convention.

public void addMyActionListener(myActionListener x); ✓
public void addActionListener(myActionListener x); ✗
public void registerMyActionListener(myActionListener x); ✗

Similarly to un register myActionListener x which of the following are valid coding convention.

public void removeMyActionListener(myActionListener x);
public void removeActionListener(myActionListener x);
public void registerMyActionListener(myActionListener x);

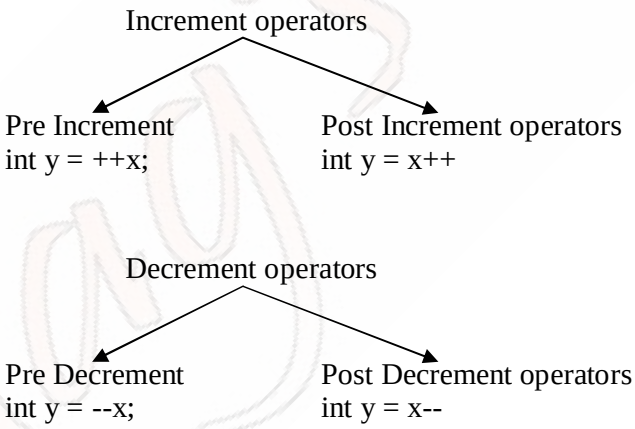
2

OPERATORS & ASSIGNMENTS

Introduction

1) increment & decrement operators.
2) arithmetic operators.
3) string concatenation operators.
4) Relational operators
5) Equality operators
6) Bitwise operators
7) Short circuit operators
8) instance of operators
9) type cast operators
10) assignment operator
11) conditional operator
12) new operator
13) [] operator
14) Precedence of java operators
15) Evaluation order of java operands

Increment & Decrement operators



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

Expression	initial value of x	final value of x	final value of x
y = ++x	4	5	5
y = x++	4	5	4
y = --x	4	3	3
y = x--	4	3	4

```
int x = 4;                  int x = 4;
int y = ++x;                int y = ++4;  → C.E: unexpected type found : value
System.out.pritnln(y);      System.out.pritnln(y);        required: varialbe
```

Increment & decrement operators we can apply only for variables but not for constant values.
We can't perform nesting of increment or decrement operator.

Ex:

```
int x= 4;
int y = ++(++x);
System.out.println(y);  → C.E: unexpected type found : value
                                required: varialbe
```

For the final variables we can't apply increment or decrement operators.

Ex:

```
final int x = 4;
x++;    → x = x + 1
System.out.println(x);
```

O/P:- C.E –can't assign a value to final variable 'x'.

We can apply increment or decrement operators even for floating point data types also.

Ex:

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

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

If we are applying any arithmetic operators b/w 2 operands 'a' & 'b' the result type is

Increment & Decrement Operators x and y are numeric (FP & integer) or char types.	

`x++`	Post-increment : add 1 to the value.
	The value is returned *before* the increment is made, e.g.
	`x = 1;`
	`y = x++;`
	Then y will hold 1 and x will hold 2

`x--`	Post-decrement : subtract 1 from the value.
	The value is returned *before* the decrement is made, e.g. :
	`x = 1;`
	`y = x--;`
	Then y will hold 1 and x will hold 0.

`++x`	Pre-increment : add 1 to the value.
	The value is returned *after* the increment is made, e.g.
	`x = 1;`
	`y = ++x;`
	Then y will hold 2 and x will hold 2.

`--x`	Pre-decrement : subtract 1 from the value.
	The value is returned *after* the decrement is made, e.g.
	`x = 1;`
	`y = --x;`
	Then y will hold 0 and x will hold 0.

Assignment Operators

The java assignment operator statement has the following syntax:

<variable> = *<expression>*

If the value already exists in the variable it is overwritten by the assignment operator (=).

Ex:

```
public class AssignmentOperatorsDemo {
public AssignmentOperatorsDemo( ) {

//          Assigning Primitive Values
            int j, k;
            j = 10;       // j gets the value 10.
            j = 5;        // j gets the value 5. Previous value is overwritten.
            k = j;        // k gets the value 5.
            System.out.println("j is : "+j);
            System.out.println("k is : "+k);

//          Assigning References
            Integer i1 = new Integer("1");
            Integer i2 = new Integer("2");
            System.out.println("i1 is : "+i1);
            System.out.println("i2 is : "+i2);
            i1 = i2;
            System.out.println("i1 is : "+i1);
            System.out.println("i2 is : "+i2);

//          Multiple Assignments
            k = j = 10;       // (k = (j = 10))
            System.out.println("j is : "+j);
            System.out.println("k is : "+k);

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

Assignment Operators

```
x operation= y
```

is equivalent to

```
x = x operation y
```

x and y must be numeric or char types except for "=", which allows x and y also to be object references. In this case, x must be of the same type of class or interface as y. If mixed floating-point and integer types, the rules for mixed types in expressions apply.

=	Assignment operator. `x = y;` y is evaluated and x set to this value. The value of x is then returned.

`+=, -=, *=, /=, %=`	Arithmetic operation and then assignment, e.g.
	`x += y;`
	is equivalent to
	`x = x + y;`

`&=, |=, ^=`	Bitwise operation and then assignment, e.g.
	`x &= y;`
	is equivalent to
	`x = x & y;`

`<<=, >>=, >>>=`	Shift operations and then assignment, e.g.
	`x <<= n;`
	is equivalent to
	`x = x << n;`

Arithmetic Operators

Java provides eight Arithmetic operators. They are for addition, subtraction, multiplication, division, modulo (or remainder), increment (or add 1), decrement (or subtract 1), and negation. An example program is shown below that demonstrates the different arithmetic operators in java.

The binary operator + is overloaded in the sense that the operation performed is determined by the type of the operands. When one of the operands is a String object, the other operand is implicitly converted to its string representation and string concatenation is performed.

String message = 100 + "Messages"; //"100 Messages"

Ex:

```
public class ArithmeticOperatorsDemo {

public ArithmeticOperatorsDemo() {
        int x, y = 10, z = 5;
        x = y + z;
        System.out.println("+ operator resulted in " + x);
        x = y - z;
        System.out.println("- operator resulted in " + x);
        x = y * z;
        System.out.println("* operator resulted in " + x);
        x = y / z;
        System.out.println("/ operator resulted in " + x);
        x = y % z;
        System.out.println("% operator resulted in " + x);
        x = y++;
        System.out.println("Postfix ++ operator resulted in " + x);
        x = ++z;
        System.out.println("Prefix ++ operator resulted in " + x);
        x = -y;
        System.out.println("Unary operator resulted in " + x);
        // Some examples of special Cases
        int tooBig = Integer.MAX_VALUE + 1; // -2147483648 which is
        // Integer.MIN_VALUE.
        int tooSmall = Integer.MIN_VALUE - 1; // 2147483647 which is
        // Integer.MAX_VALUE.
        System.out.println("tooBig becomes " + tooBig);
        System.out.println("tooSmall becomes " + tooSmall);
```

```
                System.out.println(4.0 / 0.0); // Prints: Infinity
                System.out.println(-4.0 / 0.0); // Prints: -Infinity
                System.out.println(0.0 / 0.0); // Prints: NaN
                double d1 = 12 / 8; // result: 1 by integer division. d1 gets the value
                // 1.0.
                double d2 = 12.0F / 8; // result: 1.5
                System.out.println("d1 is " + d1);
                System.out.println("d2 iss " + d2);
        }
        public static void main(String args[]) {
                new ArithmeticOperatorsDemo();
        }
}
```

Arithmetic Operators

x and y are numeric or char types. If mixed floating-point and integer types, then floating-point arithmetic used and a floating-point value returned. If mixed integer types, the wider type is returned. If double and float mixed, double is returned.

`x + y`	Addition
`x - y`	Subtraction
`x * y`	Multiplication
`x / y`	Division If FP arithmetic and y = 0.0, then infinity returned if x is not zero, NaN if x is zero. ArthmeticException thrown if x & y are integer types and y is zero.
`x % y`	Modulo - remainder of x/y returned. If FP arithmetic and y = 0.0 or infinity, then NaN returned ArthmeticException thrown if x & y are integer types and y is zero.
`-x`	Unary minus Negation of x value

Boolean Operators

x and y are boolean types. x and y can be expressions that result in a boolean value.
Result is a boolean `true` or `false` value.

`x && y`	Conditional AND	If both x and y are true, result is true. If either x or y are false, the result is false If x is false, y is not evaluated.
`x & y`	Boolean AND	If both x and y are true,the result is true. If either x or y are false, the result is false Both x and y are evaluated before the test.
`x || y`	Conditional OR	If either x or y are true, the result is true. If x is true, y is not evaluated.
`x | y`	Boolean OR	If either x or y are true, the result is true. Both x & y are evaluated before the test.

!x	Boolean NOT	If x is true, the result is false. If x is false, the result is true.
x ^ y	Boolean XOR	If x is true and y is false, the result is true. If x is false and y is true, the result is true. Otherwise, the result is false. Both x and y are evaluated before the test.

Comparison Operators

Relational operators in Java are used to compare 2 or more objects. Java provides six relational operators: greater than (>), less than (<), greater than or equal (>=), less than or equal (<=), equal (==), and not equal (!=).
All relational operators are binary operators, and their operands are numeric expressions.
Binary numeric promotion is applied to the operands of these operators. The evaluation results in a boolean value. Relational operators have precedence lower than arithmetic operators, but higher than that of the assignment operators. An example program is shown below that demonstrates the different relational operators in java.

Ex:

```
public class RelationalOperatorsDemo {

   public RelationalOperatorsDemo( ) {

     int x = 10, y = 5;
     System.out.println(”x > y : “+(x > y));
     System.out.println(”x < y : “+(x < y));
     System.out.println(”x >= y : “+(x >= y));
     System.out.println(”x <= y : “+(x <= y));
     System.out.println(”x == y : “+(x == y));
     System.out.println(”x != y : “+(x != y));

   }

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

Logical operators return a true or false value based on the state of the Variables. There are six logical, or boolean, operators. They are AND, conditional AND, OR, conditional OR, exclusive OR, and NOT. Each argument to a logical operator must be a boolean data type, and the result is always a boolean data type. An example program is shown below that demonstrates the different Logical operators in java.

Ex:

```
public class LogicalOperatorsDemo {
        public LogicalOperatorsDemo() {
                boolean x = true;
                boolean y = false;
                System.out.println("x & y : " + (x & y));
                System.out.println("x && y : " + (x && y));
                System.out.println("x | y : " + (x | y));
                System.out.println("x || y: " + (x || y));
                System.out.println("x ^ y : " + (x ^ y));
```

```
                System.out.println("!x : " + (!x));
        }
        public static void main(String args[]) {
                new LogicalOperatorsDemo();
        }
}
```

x	y	!x	x & y x && y	x | y x || y	x ^ y
true	true	false	true	true	false
true	false	false	false	true	true
false	true	true	false	true	true
false	false	true	false	false	false

Comparison Operators	
x and y are numeric or char types only except for "==" and "!=" operators, which can also compare references. If mixed types, then the narrower type converted to wider type. Returned value is boolean `true` or `false`.	
`x < y`	Is x less than y ?
`x <= y`	Is x less than or equal to y ?
`x > y`	Is x greater than y ?
`x >= y`	Is x greater than or equal to y ?
`x == y`	Is x equal to y ?
`x != y`	Is x not equal to y ?

Bitwise Operators

Java provides Bit wise operators to manipulate the contents of variables at the bit level.
These variables must be of numeric data type (char, short, int, or long). Java provides seven bitwise operators. They are AND, OR, Exclusive-OR, Complement, Left-shift, Signed Right-shift, and Unsigned Right-shift. An example program is shown below that demonstrates the different Bit wise operators in java.

Ex:

```
public class BitwiseOperatorsDemo {
```

```
        public BitwiseOperatorsDemo() {
                int x = 0xFAEF; //1 1 1 1 1 0 1 0 1 1 1 0 1 1 1 1
                int y = 0xF8E9; //1 1 1 1 1 0 0 0 1 1 1 0 1 0 0 1
                int z;
                System.out.println("x & y : " + (x & y));
                System.out.println("x | y : " + (x | y));
                System.out.println("x ^ y : " + (x ^ y));
                System.out.println("~x : " + (~x));
                System.out.println("x << y : " + (x << y));
                System.out.println("x >> y : " + (x >> y));
                System.out.println("x >>> y : " + (x >>> y));
                //There is no unsigned left shift operator
        }
        public static void main(String args[]) {
                new BitwiseOperatorsDemo();
        }
}
```

The result of applying bitwise operators between two corresponding bits in the operands is shown in the Table below.

A	B	~A	A & B	A | B	A ^ B
1	1	0	1	1	0
1	0	0	0	1	1
0	1	1	0	1	1
0	0	1	0	0	0

Bitwise Operators

x and y are integers. If mixed integer types, such as `int` and `long`, the result will be of the wider type.

Note: Operations on byte and short types may give unexpected results since operands are promoted to integers during intermediate operations. For example,

```
byte x = (byte)0xFF;
x >>>= 1;
```

will result in 0xFF in x rather than 0x7F. That is because the operation is carried out on a signed integer rather than simply on 8 bits. Here the signed byte is promoted to the signed integer 0xFFFFFFFF.

Use of an integer would go as follows:

```
int i = 0xFF;
i >>>= 1;
```

This results in 0x7F in the variable i.

`~x`	Compliment	Flip each bit, ones to zeros, zeros to ones
`x & y`	AND	AND each bit a with corresponding bit in b
`x | y`	OR	OR each bit in a with corresponding bit in b
`x ^ y`	XOR	XOR each bit in x with corresponding bit in y
`x << y`	Shift left	Shift x to the left by y bits. High order bits lost. Zero bits fill in right bits.
`x >> y`	Shift Right - Signed	Shift x to the right by y bits. Low order bits lost. Same bit value as sign (0 for positive numbers, 1 for negative) fills in the left bits.
`x >>> y`	Shift Right - Unsigned	Shift x to the right by y bits. Low order bits lost. Zeros fill in left bits regardless of sign.

Class and Object Operators

`x instanceof` *c*	Class Test Operator	The first operand must be an object reference. *c* is the name of a class or interface. If x is an instance of type c or a subclass of c, then `true` returned. If x is an instance of interface type c or a sub-interface, then `true` is returned. Otherwise, `false` is returned.
`new c(args)`	Class Instantiation	Create an instance of class c using constructor `c(args)`
`"."`	Class Member Access	Access a method or field of a class or object : `o.f` - field access for object o `o.m()` - method access for object o

`()`	Method Invocation	Parentheses after a method name invokes (i.e. calls) the code for the method, e.g.
		`o.m()`
		`o.m(x,y)`

`(c)`	Object Cast	Treat an object as the type of class or interface c: `c x=(c)y;` Treat y as an instance of class or interface c

+	String Concatenation	This binary operator will concatenate one string to another. E.g.
		
		`String str1 = "abc";`
		`String str2 = "def";`
		`String str3 = str1 + str2`
		
		results in `str3` holding `"abcdef"`.
		
		For mixed operands, if either a or b in (a + b) is a string, concatenation to a string will occur. Primitives will be converted to strings and the toString() methods of objects will be called.
		
		(This is the only case of operator *overloading* in Java.)
		
		Note that the equivalence operator "+=" will also perform string concatenation.

`[]`	Array Element Access	In Java, arrays are classes. However, the bracket operators work essentially the same as in the C/C++.
		
		To access a given element of an array, place the number of the element as an `int` value (`long` values cannot be used in Java arrays) into the brackets, e.g.
		
		`float a = b[3];`
		`int n = 5;`
		`char c=c[n];`
		
		where `b` is a `float` array and `c` is a char array.

Other Operators

The Conditional operator is the only ternary (operator takes three arguments) operator in Java. The operator evaluates the first argument and, if true, evaluates the second argument. If the first argument evaluates to false, then the third argument is evaluated. The conditional operator is the expression equivalent of the if-else statement. The conditional expression can be nested and the conditional operator associates from right to left: **(a?b?c?d:e:f:g) evaluates as (a?(b?(c?d:e):f):g)**

An example program is shown below that demonstrates the Ternary operator in java.

Ex:

```
public class TernaryOperatorsDemo {

    public TernaryOperatorsDemo() {
        int x = 10, y = 12, z = 0;
        z = x > y ? x : y;
        System.out.println("z : " + z);
    }
```

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

    public class BooleanEx1 {

        static String m1(boolean b) {
            return b ? "T" : "F";
        }
        public static void main(String[] args) {
            boolean t1 = false ? false : true ? false : true ? false : true;
            boolean t2 = false ? false
                    : (true ? false : (true ? false : true));
            boolean t3 = ((false ? false : true) ? false : true) ? false
                    : true;
            System.out.println(m1(t1) + m1(t2) + m1(t3));
        }
    }
```

O/P:- FFT

Other Operators		
x=*boolean*?y:x	Conditional Operator	The first operand - *boolean* - is a boolean variable or expression. First this boolean operand is evaluated. If it is `true` then the second operator evaluated and x is set to that value. If the boolean operator is `false`, then the third operand is evaluated and x is set to that value.
(*primitive type*)	Type Cast	To assign a value of one primitive numeric type a more narrow type, e.g. long to int, an explicit cast operation is required, e.g. `long a = 5; int b = (int)a;`

Operators Precedence

The order in which operators are applied is known as precedence. Operators with a higher precedence are applied before operators with a lower precedence. The operator precedence order of Java is shown below. Operators at the top of the table are applied before operators lower down in the table. If two operators have the same precedence, they are applied in the order they appear in a statement.
That is, from left to right. You can use parentheses to override the default precedence.

Operators Precedence	
postfix	**[] . () expr++ expr–**
unary	**++expr –expr +expr -expr ! ~**
creation/caste	**new (type)expr**
multiplicative	*** / %**

additive	**+ -**
shift	**>> >>>**
relational	**< <= > >= instanceof**
equality	**== !=**
bitwise AND	**&**
bitwise exclusive OR	^
bitwise inclusive OR	|
logical AND	**&&**
logical OR	||
ternary	**?:**
assignment	**= "op="**

Example

In an operation such as,

result = 4 + 5 * 3

First (5 * 3) is evaluated and the result is added to 4 giving the Final Result value as 19. Note that '*' takes higher precedence than '+' according to chart shown above. This kind of precedence of one operator over another applies to all the operators.

Summary of Operators

The following quick reference summarizes the operators supported by the Java programming language.

Simple Assignment Operator

= Simple assignment operator

Arithmetic Operators

+ Additive operator (also used for String concatenation)
- Subtraction operator
* Multiplication operator
/ Division operator
%Remainder operator

Unary Operators

+ Unary plus operator; indicates positive value (numbers are positive without this, however)
- Unary minus operator; negates an expression
++ Increment operator; increments a value by 1
-- Decrement operator; decrements a value by 1
! Logical compliment operator; inverts the value of a boolean

Equality and Relational Operators

==	Equal to
!=	Not equal to
>	Greater than
>=	Greater than or equal to
<	Less than
<=	Less than or equal to

Conditional Operators

&&	Conditional-AND
||	Conditional-OR
?:	Ternary (shorthand for if-then-else statement)

Type Comparison Operator

instanceof Compares an object to a specified type

Bitwise and Bit Shift Operators

~	Unary bitwise complement
<<	Signed left shift
>>	Signed right shift
>>>	Unsigned right shift
&	Bitwise AND
^	Bitwise exclusive OR
|	Bitwise inclusive OR

3

DECLARATION AND ACCESS CONTROL

Introduction

In this chapter the topics going to cover are

1) Java Source File Structure
2) Class Modifiers
3) Member Modifiers
4) Interface

Java Source File Structure

A java Source file can contain any no of classes but at most one class can be declared as the public. if there is any public class then compulsory the name of the source file and the name of the public must be matched other wise compile time error. It there is no public class then any name we can use for the source file

Ex:

```
class A
{
}
public class B
{
}
class c
{
}
```

If we save this file as B.java we won't get any compile time error. Other wise will get compile time error. For example if we declare 'c' class as public we will get Compile time error saying class c is public should be declared in a file named c.java.

Note: It is recommended to take only one class for source file.

import Statement

The following Example will explains the use of import statement.

Ex:

```
class Test
{
        public static void main(String arg[])
        {
                ArrayList l = new ArrayList();
        }
}
```

C.E: can't find symbol class ArrayList

We can resolve this problem by using **fully qualified** name i.e

java.ArrayList = new java.util.ArrayList();

using fully qualified name every time reduces readability of the code. we can resolve this by using import statement. just declare import statement only once and use short names directly.

Ex:

```
import java.util.ArrayList;
class Test
{
        public static void main(String arg[])
        {
                ArrayList l = new ArrayList();
        }
}
```

Hence the main objective of import statement is it acts as typing shortcut.

Types of import statement

There are 2 types of import statements.

1) implicit class import.
2) explicit class import.

implicit class imports

import java.util.;*

- it reduces readability and hence it is not recommended to use in real time coding.
- It acts as a typing shortcut

explicit class imports

import java.util.ArrayList;

It is recommended to use because it improves readability.

Case1: **which of the following import statements are valid ?**

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

Case2: consider the following class

```
class MyObject extends java.util.util.UnicastRemoteObject
{
}
```

The code compiles fine even though we are not using import statement because we are providing fully qualified name.

Case3:

```
import java.util.*;
import java.sql.*;
class Test
{
        public static void main(String arg[])
        {
```

```
            Date d = new Date();
        }
    }
```

C.E: reference to Date is ambiguous even in the list case also we will get the same problem because it is available in both util and awt packages.

Case4:

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

Compiler resolves the import statement in the following order.

1) Explicit class imports.
2) Classes present in current working directory.
3) Implicit class imports.

Case5:

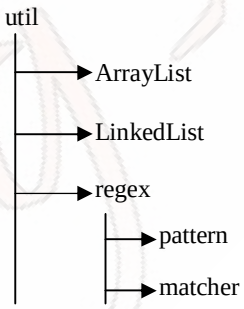
Import statement is totally compile time issue. If more no of imports, more will be compile time. But there is no effect on execution time.

When ever we are importing a package all the classes present in that package will be available but not sub package classes.

Ex:

To use pattern class compulsory we should use

```
import java.util.regex.*;
import java.util.*;
```



static imports

This concept has introduced in 1.5 version. According to sun people static imports improves readability of the code but according to world wide java expert static imports increases confusion instead of improving readability.

Hence if there is no specific requirement it is not recommended to use static imports. The main objective of static imports is to import static members of a particular class. So that with out using class name we can access directly static members.

Ex:

With out static import

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

With static import

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

Ex:

```
import static java.lang.Integer.*;
import static java.lang.Byte.*;
class Test
{
        public static void main(String arg[])
        {
                System.out.println(MAX_VALUE);
        }
}
```

Reference to MAX_VALUE is ambiguous. Compiler always resolves static members in the following order.

1) Current class static member
2) Explicit static member
3) Implicit static member

Ex:

```
import static java.lang.Integer.MAX_VALUE;      → 2
import static java.lang.Byte.*;
class Test
{
        static int MAX_VALUE = 999;      → 1
        public static void main(String arg[])
        {
                System.out.println(MAX_VALUE);
        }
}
```

O/P:- 999

If we are commenting line 1 then the o/p is 2147483647. if we are commenting both lines 1 & 2 then the O/P is 127.

Note:- It is not recommended to extend import concept up to static members because several classes can contain static members with same name. which may cause compile time error.

Static imports also reduces readability of the code. It is recommended to access static members by using class name only.

Which of the following import statements are valid?

import java.lang.Math.*; ✗
import static java.lang.Math.*; ✓
import java.lang.Math; ✓
import static java.lang.Math; ✗

package concept

package in java is an encapsulation mechanism to group related classes and interfaces into a single module. The main objective of package is to resolve naming conflicts.
There is universally accepted convention to name the package i.e to use internet domain name in reverse.

Ex:

com.IBM, com.ICICIBANK, com.AirtelIndia

```
package com.rajusoft;
class Test
{
        public static void main(String arg[])
        {
                System.out.println("package name");
        }
}
```

```
javac Test.java
```

The generated .class file will be placed in current working directory.

```
javac –d . Test.java
```

Here '.' Represents current place.

The generated .class file will be placed in corresponding package structure.
If the required package structure is not already available this command itself create the needed packages.

If the specified destination is not available we will get compile time error. Saying the system can't find the path specified.

```
javac –d Z: Test.java
```

if Z: is not available we will get C.E

```
java com.rajusoft Test
```

for any java source file we are allowed to take at most 1 package statement i.e more than one package statement is not allowed violation leads to C.E.

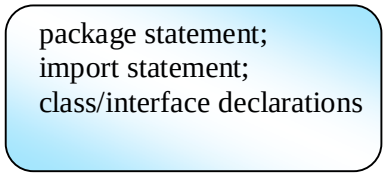
Ex:

```
package pack1;
package pack2;
class student
{

}
```

If we are not taking any package statement then the current working directory Acts as default package. The first non-comment statement in any java source file is package statement.

```
import java.util.*;
package pack1;
class Test
{

}
```

The following is the current structure of java source file

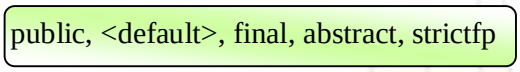


class modifier

when ever we are writing our own classes compulsory we should provide information about our class to the JVM like.

1) child class creation is possible or not
2) Instantiation is possible or not
3) Whether we can access this class from any where

We can specify this information by declaring the corresponding modifier. The only applicable modifier for the top level classes are



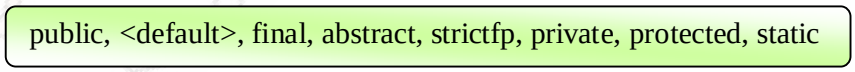
If we are using any other modifier we will get compile time error.

Ex:

```
private class Test
{
        public static void main(String arg[])
        {
                int x = 10;
                if(x == 10)
                {
                        System.out.println("Hi");
                }
                else
                {
                        System.out.println("Hello");
                }
        }
}
```

C.E: modifier private not allowed here

For the inner classes the applicable modifiers are



public classes

If a class declared as the public then we can access that class from any where.

Ex:

```
package pack1;
public class A
{
        public void m1()
        {
                System.out.println("Hai hello");
        }
}
```

```
package pack2;
import pack1.A;
class B
{
        public static void main(String arg[])
        {
                A a1 = new A();
                a1.m1();
        }
}
```

If we are not declaring class 'A' as public then while compiling 'B' class we will get C.E saying pack1.A is not public in pack1. can't be accessed from outside package.

<default> classes

If a class declared as default then we can access that class only in the current package. <default> access is also known as package level access.

abstract

abstract is the modifier applicable only for classes and methods i.e we can't use abstract for variables.
If we don't know about implementation still we are allowed to declare methods with abstract modifier.

Ex:

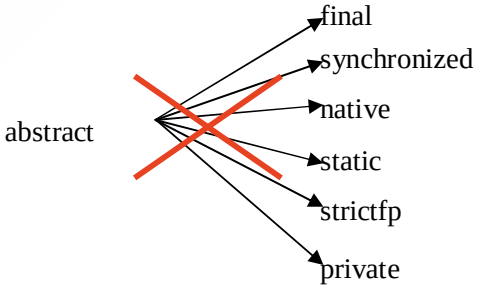
```
abstract class Vehicle
{
        public abstract int getNoOfWheels();
}
class Bus extends Vehicle
{
        public int getNoOfWheels()
        {
                return 7;
        }
}
```

abstract method should have only declaration but not implementation hence abstract method declaration should ends with ;(semicolon)

Ex:

```
public abstract void m1();      ✓
public abstract void m1(){}     ✗
```

child class is responsible to provide implementation for abstract methods.
abstract never talks about implementation, if any modifier talks about implementation it is considered as illegal combination with abstract.
Hence the following 6 combinations are illegal.



If we don't want instantiation for any class such type of classes we have to declare with abstract modifier. i.e for abstract classes instantiation is not possible
If a class contain at least one abstract method then the corresponding class should be declared as abstract otherwise C.E.
Even though class doesn't contain any abstract method still we can declare that class with abstract modifier.
i.e abstract class can contain zero no of abstract methods.

Ex:

HTTP servlet is a abstract class but it doesn't contain only abstract method.

Case1:

```
class Test
{
        public abstract void m1()
        {

        }
}
```

C.E: abstract methods can't have a body

Case2:

```
class Test
{
        public abstract void m1();
}
```

C.E: Test is not abstract and doesn't override abstract method m1 in Test()

Case3:

```
abstract class Test
{
        public abstract void m1();
}
class SubTest extends Test
{
}
```

C.E: SubTest is not abstract and doesn't override abstract method m1() in Test.

If we are creating any child class for abstract class then compulsory we should provide implementation for all abstract methods of parent class other wise the child class also should be declared as abstract

Ex:

```
abstract class Test
{
        public static void main(String arg[])
        {
                Test t = new Test();
        }
}
```

C.E: Test is abstract can't be instantiated.

final

final is the modifier applicable for classes, methods and variables.
If a method is not allowed to override in the child class then we should declare that method as 'final'.
i.e for the final methods overriding is not possible.

Ex:

```
class P
{
        public final void marry()
        {
                System.out.println("Bujji");
```

```
        }
}
class C extends P
{
        public void marry()
        {
                System.out.println("Preeti");
        }
}
```
C.E: marry in 'C' can't override, overridden method is final.

for any class if we are not allow to create child class, such type of classes we have to declare with final i.e for final classes inheritance is not possible.

Ex:

```
final class P
{
}
class C extends P
{
}
```
C.E: can't inherit from final 'p'.

Relation Ship between final and abstract

final method should not be overridden but we should override abstract method to provide implementation. Hence final & abstract combination is always illegal.

public abstract final void m1(){}

Illegal combination of modifiers : abstract, final.

For the final classes we are not allowed to create the child class. But for abstract class we should create child class to provide implementation. Hence abstract and final are Illegal combination of modifiers for that classes.

final classes can't contain abstract methods but abstract classes can contain final methods.

Ex:

```
final class P                    abstract class P
{                                {
        abstract void m1();              final void m1(){}
}                                }
```

It is a good programming practice to use abstract keyword but If there is no specific requirement it is not recommended to use 'final' keyword.

strictfp

This modifier is applicable only for methods and classes but not for variables. If a method declared as a strictfp all floating point calculations in that method IEEE754 standard so that we can get platform independent results.

strictfp and abstract is always illegal combination for methods. If a class declared as strictfp all concrete methods in that class will follow IEEE754 standard for floating point arithmetic.

abstract and strictfp is a legal combination for the classes(Illegal for the methods)

member modifiers

public members

we can access public members from any where but the corresponding class must be visible.

Ex:

```
package pack1;
public class A
{
        public void m1()
        {
                System.out.println("To day Sunday");
        }
}

package pack1;
import pack1.A;
class B
{
        public static void main(String arg[])
        {
                A a = new A();
                a.m1();
        }
}
```

If class A is not public then we are not allowed to access m1 from outside the package even though m1 is public.

<default> members

If a member declared as a default we can access that member only in the current package.

Ex:

In the above program if we declare m1() as default then we can't access that method from pack2 even though class A is public.

private members

If a member declared as private we can access that member only in the current class.

Ex: singleton classes we can implement by using private constructor. Private members are not visible in the child class to provide implementation. Hence private and abstract combination is illegal for methods.

protected members

If a member declared as protected then we can access that member from any where with in the current package and only in child classes from outside package.

protected = <default> + kids(childclasses)

We can access protected members from outside package only in child class. And we should use child class reference only. i.e parent class reference is not allowed to access protected members from outside package violation leads to C.E but in the current package we can access protected members either by using parent class reference or by child class reference.

Ex:

```
package pack1;
public class A
{
        protected void m1()
        {
```

```
                System.out.println("This is very imp point");
        }
}
package pack2;
import pack1.A;
class B extends A
{
        public static void main(String arg[])
        {
        Case1: A a1 = new A();          ✗
                a1.m1();

        Case2: B b1 = new B();          ✓
                b1.m1();

        Case3: A a2 = new B()           ✗
                a2.m1();
        }
}
```

If we can take B- class in the same package pack1, then there is no C.E.

Summarization of public, protected, default and private

Visibility	public	protected	<default>	private
With in the same class	✓	✓	✓	✓
With in the same package but from out side class	✓	✓	✓	✓
Outside package but from child class	✓	✓	✗	✗
Outside package but from non-child class	✓	✗	✗	✗

- The Most Restricted Modifier is **Private.**
- The Modifier Which provides wide range of access is **Public** .
- **Public > Protected > Default > Private**
- Recommended Modifier for data members is Private.
- And for methods is public .

Final Variables

For the Static and instance variables no need to perform initialization, JVM Will Provide default initialization.

For the local variables compulsory we should perform initialization before using.

Final Instance Variables

For the final instance variables JVM won't provide any default values, compulsory we should perform initialization **before completion of constructor.**

The following are the places to perform this

1) At the time of declaration:.

```
final int i = 0;
```

2) Inside instance initialization class

```
final int i;
{
        i = 0;
}
```

3) Inside constructor

```
final int i;
    test()
    {
        i = 0;
    }
```

If u r performing initialization any where else we will get compile time error.

Ex:

```
Class Test
{
    final int i;
}
```

O/P:- Generates Compile time error, variable "i" might not have been initialized.

```
Class Test
{
    final int i;
    public void m1()
    {
        i=20;
    }
}
```

O/P:- Generates Compile time error, can't assign a value to final variable.

```
Class Test
{
    final int i;
    {
        i=20;
    }
}
```

O/P:- Won't generate compile time error.

Final static Variables

For the final static variables compulsory we should perform initialization.

Ex:

```
class Test
{
    final static int i;
}
```

O/P:-generates Compile time error. variable i might not have been assigned.

We can perform initialization for the final static variables in the following places.

1) At the time of declaration

```
final static int i = 0;
```

2) Inside static blocks

```
static
{
    i = 0;
}
```

3) If u r performing initialization anywhere else we will get compile time error

```
class Test
{
	final static int i;
	public static void main(String arg[])
	{
		i = 20;
	}
}
```

O/P:- will produce compile time error.

Final local Variables

Before using a local variable(whether it is final or non-final) we should perform initialization. If we are not using local variable then no need of perform initialization even though it is final.

Ex:

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

This is valid Because we didn't use "i" here.

```
class Test
{
	public static void main(String arg[])
	{
		final int i;
		System.out.println("hello");
	}
}
```

This is also valid because we didn't use "i" here.

```
class Test
{
	public static void main(String arg[])
	{
		final int i = 0;
		System.out.println("hello");
	}
}
```

This is valid..

```
class Test
{
	public static void main(String arg[])
	{
		final int i;
		System.out.println(i);
	}
}
```

This is not valid. Because Before going to use we didn't initialize "i".

The variables which are declared as formal are acts as local variables of the method.

if a *formal parameter* declared as the final then with in the method we are not allowed to change it's values.

Ex:

```
class Test
{
	public static void main(String arg[])
	{
		m1(10,20);
	}
	public static void m1(final int i,final int j)
	{
		//i=100;
		//j=100;
		System.out.println(i+"----"+j);
	}
}
```

If you use final you can't change i, j values.

Final parameter 'i' may not be assigned.

For the local variables the only applicable modifier is final.

Ex:

```
class Test
{
	public static void main(String arg[])
	{
		private int i = 20;
		System.out.println(i);
	}
}
```

O/P:- Provides compile time error because we declare int as private.

static Modifier

static is the modifier is applicable for methods and variables.
we can't declare top level classes as static but inner classes can be declares as static(static nested class).
Instance variables(methods) can't be accessed from static area directly, But Static members can be accessed from any where.

Ex:

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

O/P:- C.E: non static variable can't be referred from static context.

Q) Consider the following 4 declarations

1) int i = 0;
2) static int i = 0;
3) public void m1()
   {

```
        System.out.println(i)
    }
```

4)
```
    public static void main()
    {
        System.out.println(i)
    }
```

Which of the following r not allowed to take simultaneously in any class

A) 1 & 3 B) 1 & 4 C) 2 & 3 D) 2 & 4

Ans : B

for static methods compulsory we should provide complete implementation But for Abstract methods we can't provide implementation hence ***abstract*** and ***static*** *combination* is illegal for methods.

Overloading is possible for static methods.

Ex:-

```
class  StringTest
{
    public static void main(String[] args)
    {
    System.out.println("Haiiiiii");
    }
    public static void main(int [] arg)
    {
    System.out.println("Hellllllllllo");
    }
}
```

inheritance concept is applicable for the static method.

overriding is not applicable for static methods but seems to be overriding is possible, but it is method hiding.

Ex:

```
class Foo
{
    public static void method()
    {
        System.out.println("in Foo");
    }
}

class Bar extends Foo
{
    public static void method()
    {
        System.out.println("in Bar");
    }
}
```

Ex:

```
class Foo
{
    public static void classMethod()
    {
        System.out.println("classMethod() in Foo");
    }
```

```
        public void instanceMethod()
        {
                System.out.println("instanceMethod() in Foo");
        }
}

class Bar extends Foo
{
        public static void classMethod()
        {
                System.out.println("classMethod() in Bar");
        }
        public void instanceMethod()
        {
                System.out.println("instanceMethod() in Bar");
         }
}

class Test
{
        public static void main(String[] args)
        {
                Foo f = new Bar();
                f.instanceMethod();
                f.classMethod();
        }
}
```

If you run this, the output is
instanceMethod() in Bar
classMethod() in Foo

native method

native is the modifier applicable only for methods
we can't declare classes and variables as native.
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 communicate with already existing legacy systems.

psudocode:

```
class Native
{
        static
        {
                System.loadLibrary("Path of native library");//Loading the native
library.
        }
        public native void m1(); //Declaring a native method.
}
class client
```

```
{
        Native n = new Native();
        n.m1();//Invoking a native method.
}
```

native method should end with ;(semicolon).because we r not responsible to provide implementation, it is already available.

- The use of native keyword breaks the platform independent nature of java.
- For the native methods ***overloading***, ***Inheritance*** and ***overriding*** concepts are applicable.
- For the native methods implementation is already available. But for abstract methods implementation won't be available. Hence *abstract* and *native* is *illegal combination* of modifiers.

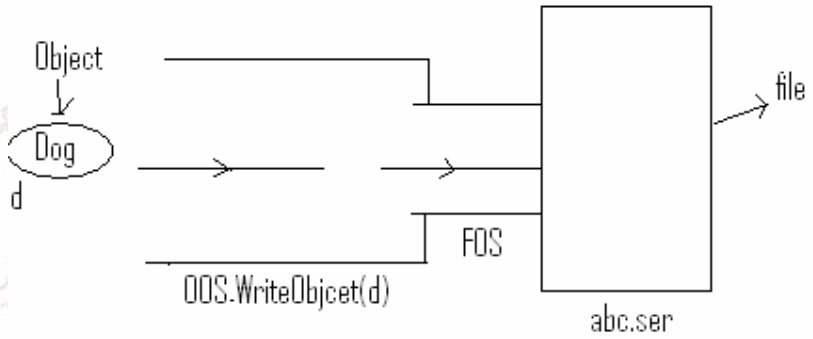
Synchronized

- It is a keyword applicable only for methods and blocks. We can't declare variables and classes with synchronized keyword.
- If a method declared as synchronized at a time only one thread is allowed to execute on the given object. Hence the main advantage of synchronized keyword is we can overcome data inconsistency problem.
- Use of synchronized keyword may effect performance of the system.
- Synchronized and abstract combination is illegal for the methods.

Transient Modifier

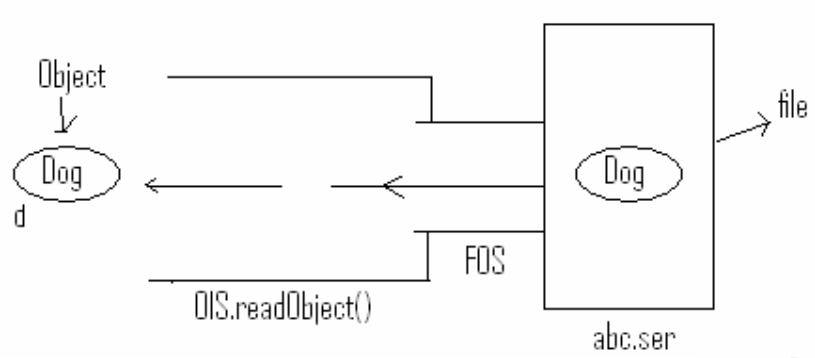
Transient is the keyword applicable only for variables. i.e we can't apply transient for methods and classes.

Serialization The Process of Saving an object to a file is called "serialization". But strictly speaking serialization is the process of converting an object from *java supported* format to *network* or *file* supported format.
By using FileOutPutStream, ObjectOutPutStream classes we can achieve serialization



DeSerialization The process of reading an object from a file is called deserialization. But strictly speaking it is the process of Converting an object from *network* supported format or *file* supported format to *java* supported format.

By using FileInputStream, ObjectInputStream we can achieve deserialization.



Ex:

```
import java.io.*;
class TransientDemo implements Serializable
{
	int i = 10;
	int j = 20;
	public static void main(String args[])throws Exception
	{
		TransientDemo d = new TransientDemo();
		FileOutputStream fos = new FileOutputStream("abc.ser");
		ObjectOutputStream oos = new ObjectOutputStream(fos);
		oos.writeObject(d);

		FileInputStream fis = new FileInputStream("abc.ser");
		ObjectInputStream ois = new ObjectInputStream(fis);
		TransientDemo d1 = (TransientDemo)ois.readObject();
		System.out.println(d1.i+"----"+d1.j);
	}
}
```

Serialization

DeSerialization

We can perform serialization only serialization objects.

- An Object is said to be serializable if and only if the corresponding class should implement serializable interface. serializable interface present in java.io package and doesn't contain any method, it is marker interface.

- If u r trying to perform serialization of a non serialization Object. we will get runtime exception saying NonSerializableException.

- While performing serialization if u don't want to save the value of a particular variable, that variable we have declared with transient keyword.

- At the time of serialization, JVM ignores the value of transient variable and saves it's default value.

i.e Transient means not to serialize.

- static variables never part of object state hence they won't participate in serialization process.
- Declaring a static variable as transient there is no impact.
- Similarly declaring final variables with transient keyword creates no impact.

Declaration	O/P
int i = 10; int j = 20;	10…20
transient int i = 10; int j = 20;	0…20
transient int i = 10; transient int j = 20;	0…0
transient static int i = 10; transient int j = 20;	10…0
transient static int i = 10; transient final int j = 20;	10…20

VolatileModifier

- Volatile is the keyword applicable only for variables. We can't declare methods and classes with volatile modifier.
- If the value of a variable keep on changing then we have to declare that variable as volatile.
- For the volatile variable JVM will create a separate local copy for every thread.
- All the intermediate calculation performed by that thread will be referred in local copy instead of master copy ,once the value got finalized just before terminating the thread the local copy value updated in the master copy. So that remaining threads will always get a stable value.
- The main advantage of volatile keyword is we can overcome "Data Inconsistency" problems. Creating and maintaining separate copy for every thread will increase complexity and effect performance of the system.
- volatile and final is illegal combination for the variables.

Modifier	Variables	Method	Top level class	Inner class	Blocks	constructor
public	✓	✓	✓	✓	X	✓
protected	✓	✓	X	✓	X	✓
<default>	✓	✓	✓	✓	X	✓
private	✓	✓	X	✓	X	✓
abstract	X	✓	✓	✓	X	X
final	✓	✓	✓	✓	X	X
strictfp	X	✓	✓	✓	X	X
static	✓	✓	X	✓	✓	X
synchronized	X	✓	X	X	✓	X
native	X	✓	X	X	X	X
transient	✓	X	X	X	X	X
volatile	✓	X	X	X	X	X

Interfaces

1) Introduction.
2) Declaring Interface.
3) Interface Methods.
4) Interface variables.
5) Naming conflicts in interface.
6) Marker/Tag interface.

Intoduction

From the client point of view interface defines the set of services what he is getting. From the service provider point of view an interface defines the set of services what he is offering.
Hence an interface acts as a contract between **client** and **service provider**.

The Main Advantages of interface are

- **Security :** The third party person not-allowed to know internal implementation details.
- **Enhancement:** With out effecting end user or client we can perform any modification in the internal implementation.
- **Improves Maintainability.**
- We can achieve communication between 2 different systems(A java application can communicate with Dotnet through interfaces.
  interface never allowed to contain any implementation details. Hence all the methods declared inside interfaces must be abstract and interfaces is considered as 100% pure abstract class.

Declaring An Interface

```
interface sample
{
        public void m1();
        public void m2();

}
```

The allowed modifiers for interface are

```
public
abstraction
strictfp
<default>
```

When ever a class implementing an interface complexity it should provide the implementation for all the interface methods. Other wise the class must be declared as abstract. Violation leads to compile time error.

```
interface sample
{
        public void m1();
        public void m2();

}
class test implements sample
{
}
```

C.E:- Test is not abstract and does not override abstract methods in same
If test class declared as abstract then we won't get any compilation error.

If the test class declared as abstract then the child class of test is responsible to provide implementation for m2() method.

When ever we are implementing an interface method compulsory we should declare that method as public otherwise compile time error.

```
interface sample
{
        void m1();
}
class test implements sample
{
        void m1(){}
}
```

m1() in **test** can't implement m1() in **sample** attempting to assign weaker access privileges was public.

```
Public  void m1(){}
```

If we declared public then no compile time error

extends vs implements

A class can extends only one class at a time. But an interface can extends any no of interfaces simultaneously.
But an interface can't implement another interface.

Q) Which of the following statements are valid.

A class can extends any no of classes simultaneously. X
A class can extends only one class at a time. ✓
An interface can extend only one interface at a time. X
An interface can extend any no of interfaces simultaneously. ✓
A class can implement any no of interfaces at a time. ✓
A class can extend an interface. X
An interface can implement another interface. X

interface Methods

Every Interface method is by default public and abstract whether we r declaring or not.
Hence the following declarations are equal inside interface.

```
void m1();
public void m1();
public abstract void m1();
```

As the interface method are by default public and abstract, we r not allowed to use the following modifiers.

```
private
protected
static
final
native
strictfp
synchronized.
```

Which of the following method declarations are valid inside interface.

1) private void m1(); X

2) void m1(){} X
3) final void m1(); X
4) static synchronized void m1(); X
5) native void m1(); X
6) public abstract void m1(); ✓

Interface variables

An interface can contain variables also every interface variable is by default public static and final whether we r declaring or not.
Hence the following declarations are equal inside interface.

```
                int i = 10;
          public int i = 10;
     public static int i = 10;
public static final int i = 10;
```

As interface variables already public static and final we r not allowed to declare with the following modifiers.

private, protected, volatile, transient.

For the interface variables compulsory we should perform initialization at the time of declarations only.

Eg:-

```
interface inter
{
    int i ;   → C.E = expected.
}
```

Which of the following variable declarations are allowed inside an interface

1) int i = 10; ✓
2) private int i = 10 ; X
3) volatile int i = 10 ;X
4) transient int i = 10 ;X
5) int i ; X
6) public static final int i = 10 ; ✓

interface variables are by default available in the implemented classes.
From the implementation classes we r allowed to access interface variables but we r not allowed to change their values i.e reassignment is not possible because these are final.

Ex:

```
interface inter
{
    int i = 10;
}
class test implements inter
{
    public static void main(String arg[])
    {
        //i = 20;        →C.E should not assign final variable.
        System.out.println(i);
    }
}
```

If place int i = 20 instead of i = 20; no compiler error will come because it is local variable to main method.

Naming conflicts in interfaces

Case1:

If two interfaces contain a method with same signature and same return type in the implementation class, only one method implementation is enough.

Ex:

```
interface Left
{
	void m1();
}

interface Right
{
	void m1();
}

class test implements Left, Right
{
	public void m1()
	{
		System.out.println("haiiiiiii");
	}
}
```

Case2:

If two interfaces contains a method with the same name but different arguments. In the implementation class compulsory we should provide implementation for both methods and these methods act as overloads method.

Ex:

```
interface Right
{
	void m1(int i);
}
class test implements Left, Right
{
	public void m1()
	{
		System.out.println("m1 method  no args");
	}

	public void m1(int i)
	{
		System.out.println("m1 method with args");
	}
}
```

Case3:

If two interfaces contains a method with same signature but different return type then we can't implement those two interfaces simultaneously.

Ex:

```
interface Left
{
	int m1();
}

interface Right
{
```

```
void m1();
}
class test implements Left, Right
{
        …
}
```

Variable naming conflicts

Ex:

```
interface Left
{
        int i = 10;
}

interface Right
{
        int i = 100;
}

class test implements Left, Right
{

        public static void main(String ar[])
        {
                //System.out.println(i);→ C.E reference to i is ambiguous
                System.out.println(Left.i);   ✓
                System.out.println(Right.i);  ✓
        }
}
```

Marker Interface

By implementing an interface if our objects will get some ability, Such type of interfaces are called "*marker*" or "*taginterface*".

Ex:

Serializable, Clonable interfaces are marked for some ability.

If an interface doesn't contain any method it is always interface.
Even though an interface contains some methods by implementing that interface if our objects will get some ability. Such type of interfaces are called 'marker' or 'taginterfaces'

Ex:

Comparable, Runnable.

4

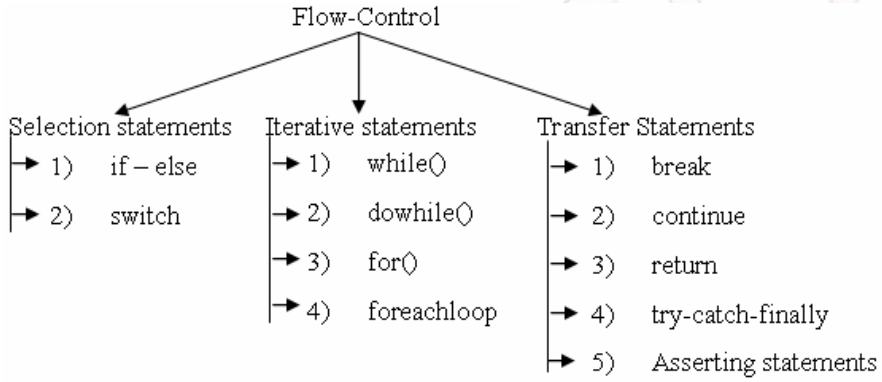
FLOW CONTROL

Introduction

Flow control describes the order in which all the statements will execute at run time.
Flow controls are categorized into 3 types those are

1) Selection statements
2) Iterative statements
3) Transfer statements

The following fig will explains the flow controls categories.



Selection statements

if-else statement

In the case of if-else statements else part & curly braces are optional.
Without curly braces we are allowed to take 1 statement under 'if', that statement never be declarative statement other wise compile time error.

```
if(true)
System.out.println("Hello"); ✓
```

```
if(true)
int i = 10; ✗
```

```
if(true)
{
        int i = 10; ✓
}
```

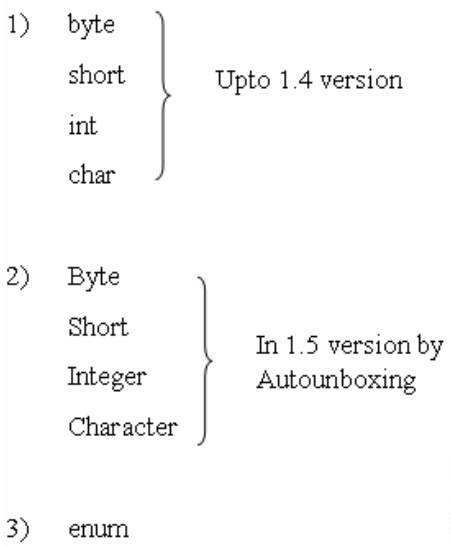
switch statement

Syntax:-

```
switch (i)
{
case 1  :System.out.println("Hi");
         break;
case 2  :System.out.println("Hello");
         break;
case 3  :System.out.println("Hai");
```

```
                    break;
        default :

        }
```

Case 1:- The valid arguments to the switch statement are



The following data types are not allowed for the switch argument

long
float
double
boolean

curly braces are mandatory.

Inside switch both case and default are optional (i.e)

Ex:

```
int i = 10;
switch(i)
{
}
```

With in switch every statement should be under some case or default i.e independent statements are not allowed inside switch.

Ex:

```
switch(i)
{
        System.out.println("Hello");
}
C.E: case or default expected.

switch(i)
{
        case 1:
        default: System.out.println("Hello");
}
```

Case labels must be compiled time constants.

Ex:

```
int i = 10;
int j = 2;

switch (i)
{
case j   :System.out.println("Hello");
         break;
}
```

C.E:

```
fcdemo.java:22: constant expression required
                case j   :System.out.println("Hello");
```

In this example if we declare ‘j’ as final then there is no compile time error.

```
int i = 2;
final int j = 2;
switch (i)
{
case j   :System.out.println("Hello");
         break;
}
```

The ‘case’ labels must be in the range supported by switch argument.

Ex:

```
byte b = 100;
switch (b)
{
case 10          :System.out.println("10");
                  break;
case 100         :System.out.println("100");
                  break;
case 1000        :System.out.println("1000");
                  break;
}
```

C.E:

```
fcdemo.java:25: possible loss of precision
found   : int
required: byte
                case 1000        :System.out.println("1000");
                     ^
```

The case labels and the switch arguments can be expressions also but case label must be constant expression.

Ex:

```
byte b = 100;
switch(b+1)
{
        case 10:System.out.println("10");
        case 20:System.out.println("20");
        case 30+40:System.out.println("30+40 = 70");
}
```

Duplicate case labels are not allowed.

Ex:

```
byte b = 100;
switch(b+1)
{
        case 10:System.out.println("10");
        case 20:System.out.println("20");
```

```
            case 20:System.out.println("Duplicate 20");
        }
```

C.E:

```
fcdemo.java:33: duplicate case label
                        case 20:System.out.println("Duplicate 20");
                        ^
```

Caselabel:

- case label must be compile time constants.
- case label must be in the range of switch argument.
- case label can be expression also (only constant expression).
- Duplicates are not allowed.

default case:

In the switch statement we can place default case any where but it is convention to take default case always at least.

Ex:

```
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");
}
```

Here if 'x' is 0 then output is 0,1.
if 'x' is 1 then output is 1.
if 'x' is 2 then output is 2,default.

In side switch once we got matched case from that statement on words all the statements will execute from top to bottom until break or end of switch.

Ex:

```
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");
}
```

Here if 'x' is 0 then output is 0.
if 'x' is 1 then output is 1,2.
if 'x' is 2 then output is 2.
if 'x' is 3 then output is default.

Iterative statements

while

Ex:

1)

```
while(true)
{
        System.out.println("hi");
}
System.out.println("Hello");
```

C.E:

```
fcdemo.java:48: unreachable statement
                System.out.println("Hello");
                ^
```

2)

```
while(false)
{
        System.out.println("hi");
```

```
}
System.out.println("Hello");
```

C.E:

```
fcdemo.java:45: unreachable statement
                {
                ^
```

3)
```
int a = 10;
int b = 20;
while(a<b)
{
        System.out.println("Hi");
}
System.out.println("Hello");
```

O/P:-
```
Hi
Hi
Hi
.
.
```

4)
```
final int a = 10;
final int b = 20;
while(a<b)
{
        System.out.println("Hi");
}
System.out.println("Hello");
```

C.E:

```
fcdemo.java:63: unreachable statement
                System.out.println("Hello");
                ^
```

dowhile loop

In the loop body has to execute at least once then we should go for do-while loop.

Syntax:
```
do
{
}while (boolean);
```

Here ';' is mandatory. But in C++ semicolon(;) is optional.
Curly braces are optional, with out curly braces we should take only 1 statement b/w do-while, that statement never be declarative statement.

Ex:

1)
```
int a = 10;
int b = 20;
do
System.out.println("In do while loop");
while (a>b);
```

2)
```
do
int i = 10;
while();
```

3)
```
int a = 10;
int b = 20;
do;
while(a>b);
```

Here semicolon(;) after do is equivalent to empty statement in java.

Case 1:
```
do
{
        System.out.println("Hi");
}
while (true);
System.out.println("Hello");
```
C.E:
```
fcdemo.java:81: unreachable statement
                System.out.println("Hello");
                ^
```

Case 2:
```
do
{
        System.out.println("Hi");
}
while (false);
System.out.println("Hello");
```
O/P:-
```
Hi
Hello
```

Case 3:
```
int a = 10;
int b = 20;
do
{
        System.out.println("Hi");
}while (a<b);
System.out.println("Hello");
```
O/P:-
```
Hi
Hi
Hi
.
.
```

Case 4:
```
final int a = 10;
final int b = 20;
do
{
        System.out.println("Hi");
}
while (a<b);
System.out.println("Hello");
```
C.E:
```
fcdemo.java:105: unreachable statement
                System.out.println("Hello");
                ^
```

for loop

The most commonly used loop

Syntax:
```
for(initialization section; conditional expression; increment/decrement section)
{
        //loop body
}
```
Initialization section: It is not possible to declare more than 1 variable of different types.

Ex:
```
for(int i = 10,j = 20)
```

```
for(int i = 0, long j = 0)      X
for(int i = 0; long j = 0)      X
for(int i = 0; i+j = 0;)        X
```

in the initialization section we are allowed to take any valid java statement including sop also.

Ex:

```
int i = 0;
for(System.out.println("Hi");i<5;i++)
{
        System.out.println("Hello");
}
```

O/P:-

```
Hi
Hello
Hello
Hello
Hello
Hello
```

Conditional Expression: Any expression which should return boolean value.
It is optional & default value is true by compiler.

Ex:

```
for(int i = 0; ;i++)
{
}
```

→defaultly true

Increment/Decrement section: Any valid java statement is allowed including System.out.println also

Ex:

```
for(System.out.println("Hi");;System.out.println("Hello"))
{
        System.out.println("xyz");
}
```

O/P:-

```
Hi
xyz
Hello
.
.
.
```

All the 3 parts of for loop are independent of each other & optional.

```
For( ; ;);
```

Case1:

```
for(int i = 0;true;i++)
{
        System.out.println("Hello");
}
System.out.println("Hi");
```

C.E:

```
fcdemo.java:128: unreachable statement
                System.out.println("Hi");
                ^
```

Case2:

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

C.E:

```
fcdemo.java:128: unreachable statement
                System.out.println("Hi");
                ^
```

Because by default conditional expression is true.

Case3:

```
for(int i = 0;false;i++)
{
        System.out.println("Hello");
}
System.out.println("Hi");
```

C.E:

```
fcdemo.java:128: unreachable statement
                    System.out.println("Hi");
                    ^
```

Case4:

```
int a = 10,b = 20;
for(int i = 0;a<b; i++)
{
        System.out.println("Hello");
}
System.out.println("Hi");
```

O/P:-

```
Hello
Hello
Hello
.
.
```

Case5:

```
final int a = 10,b = 20;
for(int i = 0;a<b; i++)
{
        System.out.println("Hello");
}
System.out.println("Hi");
```

C.E:

```
fcdemo.java:128: unreachable statement
                    System.out.println("Hi");
                    ^
```

for each loop

The most convenient loop for accessing the elements of only arrays & collections this loop has introduced in 1.5 version.

Ex:

General for loop

```
int []a = {10,20,30};
for(int i=0;i<a.length;i++)
{
        System.out.println(a[i]);
}
```

for-each loop

```
int []a = {10,20,30};
for(int x : a)
{
        System.out.println(x);
}
```

The program to display the elements of two dimensional string array by using for-each loop is

General for loop

```
String[] s[] = {{"A","B"},{"C","D","E"}};
for (String[] x: s)
{
        for(String y:x)
        {
        System.out.println(y);
        }
}
```

for-each loop

```
String[] s[] = {{"A","B"},{"C","D","E"}};
for(int i=0;i<s.length;i++)
{
        for(int j=0;j<s[i].length;j++)
        {
        System.out.print(s[i][j]);
        }
}
```

for-each loop is the most convenient loop to access the elements of arrays & collections but the limitation of this loop is applicable for arrays & collections and it is not general loop.

Transfer statements

break

It can be used in the following places.

1) with in the loops to come out of the loop.
2) Inside switch statement to come out of the switch .
3) If we are using break any where else we will get a compile time error.

Ex:

```
int x = 0;
if(x!=5)
        break;
System.out.println("if");
```

C.E:

```
fcdemo.java:180: break outside switch or loop
                        break;
                        ^
```

continue

1) we should use ‘continue’ only in the loops to skip current iteration & continue for the next iteration.
2) If we are using ‘continue’ any where except loops we will get compile time error saying “continue out side of loop”.

Ex:

```
for(int i=0;i<10;i++)
{
        if((i%2) == 0)
        continue;
        System.out.print(i);
}
```

O/P:- 13579

5

EXCEPTIONHANDLING

Introduction

It is an unexpected unwanted event which disturbs entire flow of the program.

RealTimeExample:

1) SleepingException
2) TirePuncharedException

If we are not handling exception, the program may terminate abnormally with out releasing allocated resources. This is not a graceful termination. Being a good programming practice compulsory we should handle exceptions for graceful termination of the program.

Exception handling means it is not repairing an exception we are providing alternative way to continue the program normally. For example if out programming requirement is to read the data from London file, if at runtime London file is not available, we have to provide a local file as the part of exception handling. So that respect of the program will be continued normally.

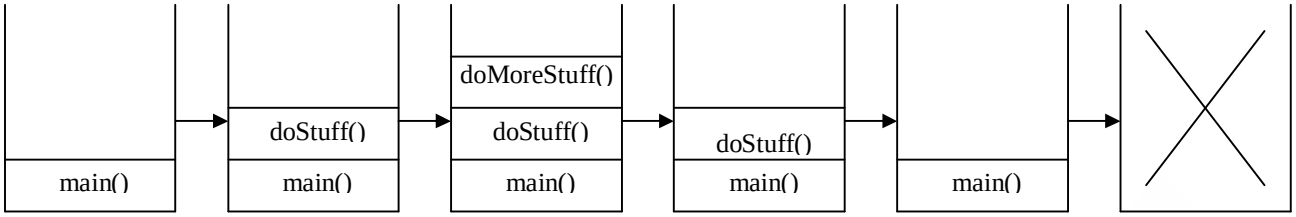
Runtime Stack Mechanism

For every thread JVM will create a runtime stack. All the method calls performed by the thread will be sorted in the corresponding runtime stack. If a method terminates normally the corresponding entry from the stack will be removed.

After completing all the method calls the stack is empty. Just before terminating the thread JVM will destroy the corresponding stack.

Ex:

```
class ExceptionDemo
{
        public static void main(String[] args)
        {
                doStuff();
        }
        public static void doStuff()
        {
                doMoreStuff();
        }
        public static void doMoreStuff()
        {
                System.out.println("Hi this is Exception .....Thread");
        }
}
```



Default Exception Handling

Ex:

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

O/P:-

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at ExceptionDemo.doMoreStuff(ExceptionDemo.java:30)
        at ExceptionDemo.doStuff(ExceptionDemo.java:25)
        at ExceptionDemo.main(ExceptionDemo.java:21)
```

When ever an exception raised the method in which it is raised is responsible for the preparation of exception object by including the following information

- Name of Exception.
- Description.
- Location of Exception.

After preparation of Exception Object, The method handovers the object to the JVM, JVM will check for Exception handling code in that method if the method doesn't contain any exception handling code then JVM terminates that method abnormally and removes corresponding entry from the stack.

JVM will check for exception handling code in the caller and if the caller method also doesn't contain exception handling code then JVM terminates that caller method abnormally and removes corresponding entry from the stack.

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 abnormally.

Just before terminating the program JVM handovers the responsibilities of exception handling to default exception handler. Default exception handler prints the error in the following format.

Name of Exception : Description
stackTrace

Exception Hierarchy

Throwable is the parent of entire java exception hierarchy. It has 2 child classes

1) Exception.
2) Error.

Exception
These are recoverable. Most of the cases exceptions are raised due to program code only.

Error
Errors are non-recoverable. Most of the cases errors are due to lack of system resources but not due to our programs.

Checked Vs UnChecked
The Exceptions which are checked by the compiler for smooth execution of the program at runtime are called '*checked exception*'

Ex:- IOException, ServletException, InterruptedException.

The Exceptions which are unable to checked by the compiler are called '*unchecked exceptions*'

Runtimeexception and it's child classes, ***Error*** and it's child classes are considered as unchecked exceptions and all the remaining considered as checked.

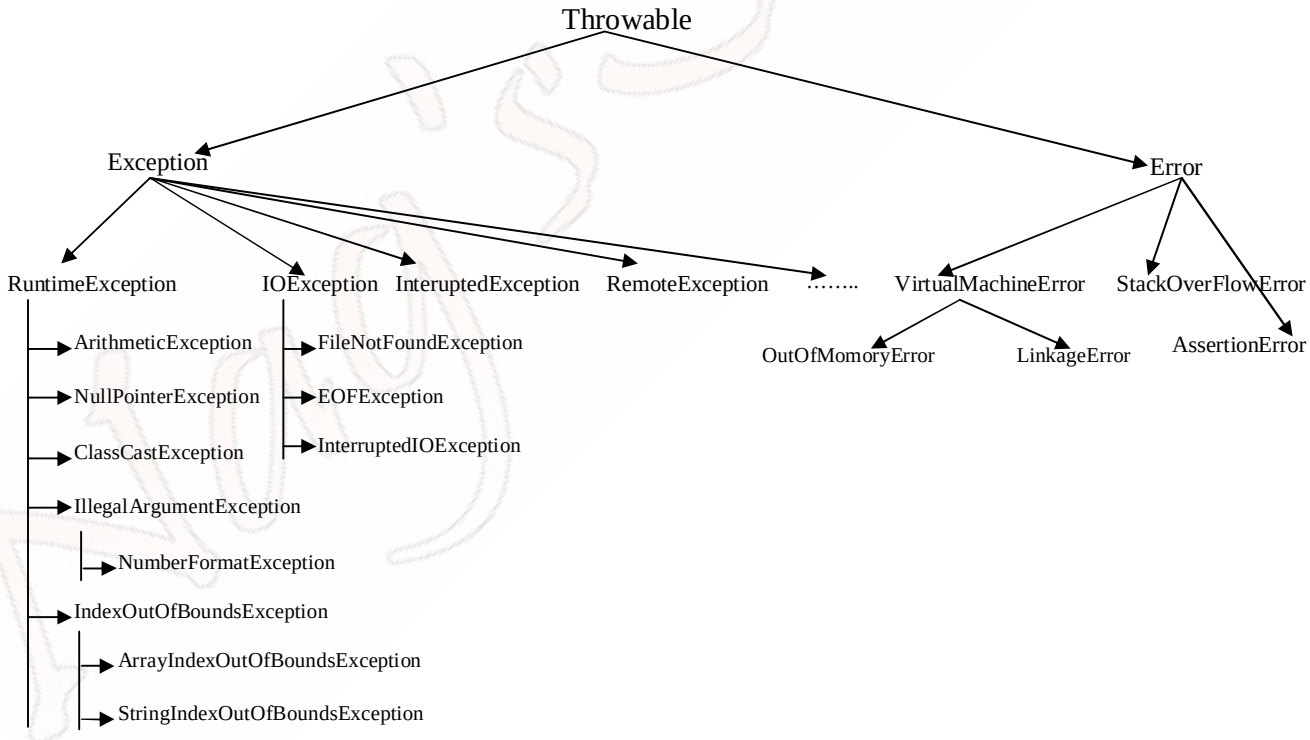
Whether the exception is checked or unchecked it always occur at runtime only.

Partially checked Vs fully checked
A checked exception is said to be fully checked iff all it's child classes also checked.

Ex:- IOException.

A checked exception is said to be partially checked if some of it's child classes are not checked.

Ex:- Exception, Throwable.



Exception Handling By Using try ,catch

We have to place the risky code inside the try block and the corresponding exception handling code inside catch block.

```
try
{
        //Risky code
}
catch (X e)
{
        //handling code
}
```

Without Exception handling code

```
class Test
{
        public static void main(String arg[])
        {
                System.out.println("Statement 1");
                System.out.println(10/0);
                System.out.println("Statement 2");
        }
}
```

Abnormal termination

```
class Test
{
        public static void main(String arg[])
        {
                System.out.println("Statement 1");
                try
                {
                        System.out.println(10/0);
                }
                catch (ArithmeticException e)
                {
                        System.out.println(10/2);
                }

                System.out.println("Statement 2");
        }
}
```

Normal termination

Control Flow in try, catch

```
try
{
        statement 1;
        statement 2;
        statement 3;
}
catch (X e)
{
        statement 4;
}
statement 5;
```

Case1: if there is no exception
        1,2,3,5 normal termination.
Case2: if there is an exception raised at statement 2 and the corresponding catch block matched.
        1,4,5 normal termination.
Case3: if an exception raised at statement 2 but the corresponding catch block is not matched
        1 Abnormal termination.
Case4: if an exception raised at statement 4 or statement 5 it is always abnormal termination.

The Methods to display Exception Information

Throwable class contains the following methods to display error information.

printStackTrace: It displays error information in the following format.

Name of Exception : *Description*
StackTace.

toString: it displays error in the following format.

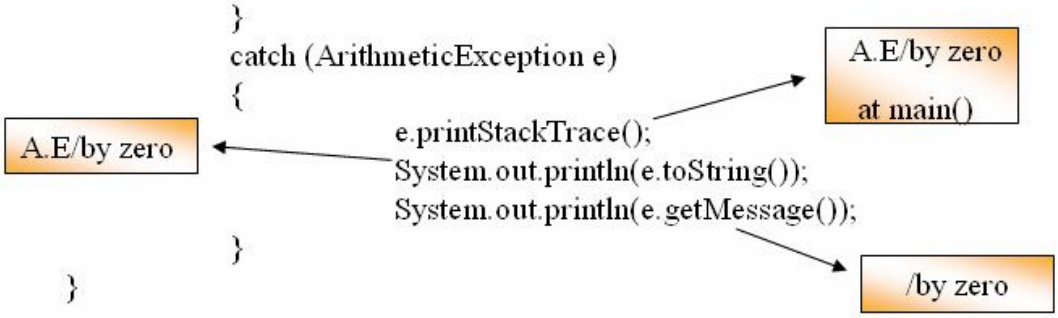
Name of Exception : *Description*

getMessage:it displays error information in the following format.

Description

Ex:-

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



Note: Default Exception handler always uses printStackTrace method only.

try with multiple catch blocks

The way of handling exception is valid from exception to exception. Hence for every exception we should define corresponding catch blocks hence try with multiple catch blocks is possible.

```
try
{
        risky code
}
catch (ArithmeticException e )
{
        //handler to A.E
}
catch (NullPointerException e)
{
        //handler for N.P.E
}
catch(IOException e)
```

```
{
        //handler for IOException
}
catch(Exception e)
{
        //handler for Exception
}
```

In the case of try with multiple catch blocks the order of catch blocks is important. And it should be from child to parent other wise Compiler Error. Saying Exception xxx has already been caught

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

}
catch (ArithmeticException e)
{

}
```

✓

```
try
{
        risky code
}
catch (ArithmeticException e )
{

}
catch (Exception e)
{

}
```

✗

C.E java.lang.ArithmeticException has already been caught

If there is no chance of raising an exception in try statement then we r not allowed to maintain catch block for that exception violation leads to compile time error but this rule is applicable only for fully checked exceptions.

```
try
{
        System.out.println("Hi");
}
catch (ArithmeticException e)
{
}
```

✓

```
try
{
        System.out.println("Hi");
}
catch (Exception e)
{
}
```

✓

```
try
{
        System.out.println("Hi");
}
catch (IOException e)
{
}
```

✗

```
try
{
        System.out.println("Hi");
}
catch (InterruptedException e)
{
}
```

✗

C.E: java.io.IOException is never thrown in body of corresponding try statement

finally

It is not recommended to place cleanup code inside **try statement** because there is no guarantee for execution of all statements inside try block.

It is not recommended to maintain cleanup code with in the **catch block** because if there is no execution the catch blocks won't be executed.
We required a block to maintain cleanup code which should execute always irrespective of whether the exception is raised or not whether it is handled or not , such block is nothing but "*finally block*"
Hence the main objective of finally block is to maintain cleanup code.

```
try
{
        //open the database connection
        //read the data
}
catch (X e)
{

}
finally
{
        close the Connection
}
```

Ex:

```
try
{
        System.out.println("try");
}
catch (ArithmeticException e)
{
        System.out.println("catch");
}
finally
{
        System.out.println("finally");
}
```

O/P: try
finally
Normal
Termination

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

O/P: catch
finally
Normal
Termination

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

O/P: finally
Abnormal termination

Hence finally block should always execute irrespective of whether the execution is raised or not raised or handled or not handled.
The finally block won't be executed if the system it self exists(JVM shutdown) i.e in the case of System.exit() finally block won't be executed.

```
try
{
        System.out.println("Hi");
        System.exit(0);
}
catch (ArithmeticException e)
{
        System.out.println("catch");
}
finally
{
        System.out.println("finally");
}
```

O/P:-Hi

Difference Between final, finally, finalize

final: It is the modifier applicable for classes methods and variables. For final classes we can't create child classes i.e inheritance is not possible.

final() methods can't be override in child classes for final variables reassignments is not possible because they are constants.

finally: It is a block associated with try catch the main objective of finally block is to maintain cleanup code which should execute always.

finalize: It is a method should be executed by the "*Garbage Collector*" just before destroying an object. The main objective of finalize method is to maintain cleanup code.

Note:- when compared with finalize, finally is always recommended to maintain cleanup code because there is no guarantee for the exact behavior of "*Garbage Collector*" it is Virtual Machine Dependent.

Possibe combinations of try, catch, finally

1 ✓
```
try
{
}
catch (X e)
{
}
finally
{
}
```

2 ✓
```
try
{
}
catch (X e)
{
}
```

3 ✓
```
try
{
}
finally
{
}
```

4 ✗
```
try
{
}
```
C.E: try with out catch or finally

5 ✗
```
catch(X e)
{
}
```
C.E: catch with out try

6 ✗
```
finally
{
}
```
C.E: finally with out try

7 ✗
```
try
{
}
System.out.println("Hello");
catch (X e)
{
}
```
C.E: try with out catch
Catch with out try

8 ✗
```
try
{
}
catch (X e)
{
}
System.out.println("Hello");
finally
{
}
```
C.E: finally with out try

9 ✓
```
try
{
}
catch (X e)
{
}
catch (Y e)
{
}
finally
{
}
```

10 ✗
```
try
{
}
catch (X e)
{
}
finally
{
}
finally
{
}
```
C.E: finally with out try

11 ✗
```
try
{
}
finally
{
}
catch (X e)
{
}
```
C.E: catch with out try

12 ✗
```
try
{
}
catch(X e)
{
}
System.out.println("Hello");
catch(Y e)
{
}
```
C.E: catch with out try

Control flow in try - catch – finally

The following program will demonstrates the control flow in different cases.

Ex:

```
class ExceptionDemo
{
    public static void main(String arg[])
    {
        try
        {
            statement1;
            statement2;
            statement3;
        }
        catch (X e)
        {
            statement4;
        }
        finally
        {
            statement5;
        }
        statement6;
    }
}
```

Case1: if there is no exception, then the statements 1, 2, 3, 5, 6 will execute with normal termination.

Case2: if an exception raised at statement-2 and the corresponding catch block matched, then the statements 1, 4, 5, 6 will execute with normal termination.

Case3: if an exception raised at statement-2 but the corresponding catch block not matched then the statements 1, 5, 6 will execute with abnormal termination.

Case4: if an exception raised at statement-2 and while executing the corresponding catch block at statement–4 an exception raised then the statements 1, 5 will execute with abnormal termination.

Case5: if an exception raised at statement-5 or at statement-6 then it is always abnormal condition.

Control flow in nested try – catch – finally

The following program will demonstrates the flow of nested try –catch – finally.

Ex:

```
try
{
    statement 1;
    statement 2;
    statement 3;
    try
    {
        statement 4;
        statement 5;
        statement 6;
    }
    catch (X e)
    {
        statement 7;
    }
    finally
```

```
        {
                statement 8;
        }
        statement 9;
}
catch (Y e)
{
        statement 10;
}
finally
{
        statement 11;
}
statement 12;
```

Case1: if there is no exception then the statements 1, 2, 3, 4, 5, 6, 7, 8, 9, 11, 12 will execute with normal termination.

Case2: if an exception raised at statement-2 and the corresponding catch block matched then the statements 1, 10, 11, 12 will execute with normal termination.

Case3: if an exception raised at statement-2 but the corresponding catch block not matched then the statements 1, 11, 12 will execute with abnormal termination.

Case4: if an exception raised at statement-5 and the corresponding inner catch has matched then the statements 1, 2, 3, 4, 7, 8, 9, 11, 12 will execute with normal termination.

Case5: if an exception raised at statement-5 and the inner catch has not matched but outer catch has matched then the statements 1, 2, 3, 4, 8, 10, 11, 12 will execute with normal termination.

Case6: if an exception raised at statement-5 but the inner and outer catch blocks are not matched then the statements 1, 2, 3, 4, 8, 11 will execute with abnormal termination.

Case7: if an exception raised at statement-7 and

i) If outer catch block has matched then the statements 1, 2, 3, - - - 8, 10, 11, 12 will execute with normal termination.
ii) If the outer catch block has not matched then the statements 1, 2, 3, - - - 8, 11 will execute with abnormal termination.

Case8: if an exception raised at statement-8 and

i) If outer catch has matched then the statements 1, 2, 3, - - - will execute with normal termination.
ii) If outer catch has not matched then the statements 1, 2, 3, - - - 11 will execute with abnormal termination.

Case9: if an exception raised at statement-9 and

i) If the outer catch has matched then the statements 1, 2, 3 - - -8, 10, 11, 12 will execute with normal termination.
ii) If the outer catch has not matched then the statements 1, 2, 3 - - -8, 11 will execute with abnormal termination.

Case10: if an exception raised at statement-10 it is always abnormal termination but before termination compulsory the finally block should be executed.

Case11: if an exception raised at statement-11 or 12 it is always abnormal termination.

throw keyword

By using throw we can hand – over exception object to the JVM. The out put of the following two programs is same.

①
```
class Test
{
        public static void main(String arg[])
        {
                System.out.println(10/0);
        }
}
```

②
```
class Test
{
        public static void main(String arg[])
        {
                throw new ArithmeticException ("/ by zero...!");
        }
}
```

Here in *first* case main method is responsible for the creation of exception object and hand – over that object to the JVM.
In the *second* case we created object explicitly and hand – over that object to the JVM programmatically by throw key – word.

Syntax:

```
throw e;
```

Where 'e' → Any throwable object

①
```
class Test
{
        public static void main(String arg[])
        {
                throw new Test();
        }
}
```

C.E:
```
ExceptionDemo.java:217: incompatible types
found   : Test
required: java.lang.Throwable
                throw new Test();
                ^
```

②
```
class Test extends RuntimeException
{
        public static void main(String arg[])
        {
                throw new Test();
        }
}
```

R.E:
```
Exception in thread "main" Test
        at Test.main(ExceptionDemo.java:217)
```

③
```
class Test
{
        static ArithmeticException e = new ArithmeticException();
        public static void main(String arg[])
        {
                throw e;
        }
}
```

R.E:
```
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at Test.main(ExceptionDemo.java:225)
```

Here Explicitly we created a object to the ArithmeticExcption class and that object was thrown by thow to the JVM.

④
```
class Test
{
        static ArithmeticException e;
        public static void main(String arg[])
        {
                throw e;
        }
}
```

R.E:
```
Exception in thread "main" java.lang.NullPointerException
        at Test.main(ExceptionDemo.java:235)
```

Here we didn't create Object to the AritmeticExcepiton class just we created a reference, so reference variable is not pointing to any object and we thrown only reference variable that's why only it shows NullPointerException.

After throw keyword we are not allowed to place any statements directly other wise compile time error.

Ex:

```
class Test
{
        public static void main(String arg[])
        {
                throw new  ArithmeticException();
                System.out.println("After throw statement...!");
        }
}
```

C.E:
```
ExceptionDemo.java:244: unreachable statement
                        System.out.println("After throw statement...!");
                        ^
```

Directly in the sense indirectly we can place any statements after throw. See the following example.

Ex:

```
class Test
{
        public static void main(String arg[])
        {
                if(false)
                {
                        throw new  ArithmeticException();
                }
                else                                                    ✓
                {
                        System.out.println("After throw statement...!");
                }
        }
}
```

① To hand – over exception object to JVM

② 'e' is of type throwable other wise C.E

③ If 'e' is not points to object. Then we will get NullpointerException

④ After throw we can't take any statement directly other wise C.E

throws

If our code may be a chance of raising *checked exception* then compulsory we should handle that checked exception either by using try, catch or we have to delegate that responsibility to the caller using throws keyword other wise C.E saying

UnreportedException : XXXException must be caught or declared to be thrown

Ex:

```
class Test
{
    public static void main(String arg[])
    {
        Thread.sleep(1000);
    }
}
```

C.E:

```
ExceptionDemo.java:267: unreported exception java.lang.InterruptedException; mus
t be caught or declared to be thrown
                Thread.sleep(1000);
                             ^
```

We can resolve this problem either by using try catch or by using throws keyword as follows

```
class Test
{
    public static void main(String arg[])
    {
        try
        {
            Thread.sleep(1000);
        }
        catch (InterruptedException e)
        {
        }
    }
}
```

```
class Test
{
    public static void main(String arg[])throws
InterruptedException
    {
        Thread.sleep(1000);
    }
}
```

Hence the main objective of throws keyword is to delegate the responsibilities of exception handling to the caller.

Ex:

```
class Test
{
    public static void main(String arg[]) throws InterruptedException
    {
        doStuff();
    }
    public static void doStuff() throws InterruptedException
    {
        doMoreStuff();
    }
    public static void doMoreStuff() throws InterruptedException
    {
        Thread.sleep(500);
        System.out.println("I am in office");
    }
}
```

If we are not taking at least one throws keyword we will get Compiler Error.

If the parent class constructor throws some checked exception then the child class constructor must throw same checked exception or its parent other wise compiler error.

Ex:

```
class p
{
        p() throws Exception
        {
        }
}
class c extends p
{
        c() throws Exception
        {
                super();
        }
}
```

If we don't take this we will get C.E

We can use throws keyword only for throwable classes other wise C.E

Ex:

```
class Test
{
        public static void main() throws Test
        {
                System.out.println("Hello");
        }
}
```

C.E:

```
ExceptionDemo.java:325: incompatible types
found   : Test
required: java.lang.Throwable
        public static void main() throws Test
                                         ^
```

Summarization of Exception Handling Keywords

try: To maintain risky code.

Catch: To maintain exception handling code.

finally: To maintain cleanup code.

throw: To hand – over exception object to the JVM programmatically.

throws: To delegate the responsibilities of exception handling to the caller.

Summarization of various compile time errors in Exception Handling

1) Exception has already been caught.
2) Exception never thrown in the body of corresponding try statement.
3) try with out catch or finally.
4) catch with out try.
5) finally with out try.
6) unreachable statement.
7) Incompatible types found: types

required: throwable

8) unreported Exception must be caught or declared to be thrown.

Customized Exception

Based on our programming requirement some times we have to create our own exception, which are nothing but "*Customized Exception*".

Ex: TooYoungException.
TooOldException.
InSufficiantFundsTransfer.

```
class TooYoungException extends RuntimeException
{
        TooYoungException(String s)
        {
                super(s);
        }
}

class TooOldException extends RuntimeException
{
        TooOldException(String s)
        {
                super(s);
        }
}
class CustomExceptionDemo
{
        public static void main(String arg[])
        {
                int age = Integer.parseInt(arg[0]);
                if(age > 60)
                {
                        throw new TooOldException("Younger age is already over");
                }
                else if(age <18)
                {
                        throw new TooYoungException("Please wait same more time");
                }
                System.out.println("Thanks for register");
        }
}
```

O/P:-

```
E:\Rjava\SCJP\PracticePrg>java CustomExceptionDemo 2
Exception in thread "main" TooYoungException: Please wait same more time
        at CustomExceptionDemo.main(ExceptionDemo.java:357)
```

```
E:\Rjava\SCJP\PracticePrg>java CustomExceptionDemo 62
Exception in thread "main" TooOldException: Younger age is already over
        at CustomExceptionDemo.main(ExceptionDemo.java:353)
```

Note: It is recommended to define customized exceptions as unchecked. i.e our custom exceptions class should extends R.E either directly or indirectly.

Top 10 Exceptions

All Exceptions are divided into two categories

JVM Exceptions: Raised automatically by the JVM when ever a particular condition occurs.
Ex: ArithmeticException, NullPointerException.

ProgramaticExceptions: These are raised programmatically because of programmers code or API's developers Code.
Ex: IllegalArgumentException, NumberFormatException.

1) NullPointerException:- It is the direct child class of *RuntimeException* and it is *unchecked*. Thrown *automatically* by the JVM when ever we are performing any operation on null.

Ex:
```
String s = null
System.out.println(s.length());
```
→ NullPointerException.

2) StackOverFlowError:- It is the child class of *Error* and it is *unchecked*. Raised *automatically* by the JVM when ever we are performing recursive method invocation.

Ex:
```
class Test
{
    public static void m1()
    {
        m1();
    }
    public static void main(String arg[])
    {
        m1();
    }
}
```

3) ArrayIndexOutOfBoundsException:- It is the child class of *RuntimeException* and it is *unchecked* thrown *automatically* by the JVM when ever we are accessing an array element with invalid int index.(Out of range index)

Ex:
```
int [] a = {10, 20, 30};
System.out.println(a[0]);
System.out.println(a[20]);
```
→ ArrayIndexOutOfBoundsException.

4)ClassCastException:- It is the child class of *RuntimeException* and it is *unchecked*. Thrown *automatically* by the JVM when ever we are trying to typecast parent class object to the child type.

Ex:
```
String s = "raju";
Object o = (Object)s
```
✓

```
Object o = new Object();
String s = (String)o;
```
✗

R.E: ClassCastException.

5) NoClassDefFoundError:- It is the child class of *Error* and it is *unchecked*. Thrown *automatically* by the JVM if the required .class file is not available.

Ex: java beebi

If beebi.class file is not available we will get NoClassDefFoundError.

6) ExceptionInInitializerError:- It is child class of *Error* and it is *unchecked*. Raised *automatically* by the JVM when ever an Exception occur during initialization of static variables or execution of static blocks.

Ex:

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

O/P:-

```
Exception in thread "main" java.lang.ExceptionInInitializerError
Caused by: java.lang.ArithmeticException: / by zero
        at Test.<clinit>(ExceptionDemo.java:383)
```

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

O/P:-

```
Exception in thread "main" java.lang.ExceptionInInitializerError
Caused by: java.lang.NullPointerException
        at Test.<clinit>(ExceptionDemo.java:391)
```

7) IllegalArgumentException:- It is the child class of *RuntimeException* and it is *unchecked* thrown *explicitly* by the programmer or API developer when ever we are invoking a method with inappropriate or invalid argument.

Ex:

The valid range of Thread priority is 1 – 10 , if we are trying to invoke setpriority method with 100 as argument we will get IllegalArgumentException.

```
public void setPriority(int i)
{
        if (i>10 || i<11)
        {
                throw new IllegalArgumentException();
        }
}
```

8) NumberFormatException:- It is the child class of *Illegal Argument* and it is *unchecked*. Thrown *programmatically* when ever we are attempting to convert String to Number type but the String is not formatted properly

Ex:

```
String s = 10;
int i = Integer.parseInt(s);      ✓

String s = "ten";
int i = Integer.parseInt(s);      ✗      → NullPointerException
```

9) IlleaglStateExcepiton:- It is the child class of *RuntimeException* and it is *unchecked*. Thrown *programmatically* to indicate that a method has invoked at an inappropriate time.

Ex:

After invalidating a session object we are not allowed to call any method on that object other wise IllegalStateException.
After comiting the response we are not allowed to redirect or forward otherwise IlleagalStateException
After starting a thread we are not allowed to start the same thread once again other wise IlleagalStateException.

```
MyThread t = new MyThread();
t.start();
t.start();          → IllegalThreadStateException.
```

10) AssertionError:- It is the child class of *Error* and it is *unchecked* thrown *programmatically* to indicate Assertion fails.

Ex:

```
Assert(false);  → AssertionError
```

Top 10 Exceptions in Table Format

Exception/Error	Child of	checked/unchecked	Thrown by
1) NullPointerException	RuntimeException	Unchecked	JVM
2) StackOverFlowError	Error	Unchecked	
3) ArrayIndexOutOfBoundsException	RuntimeException	Unchecked	
4)ClassCastException	RuntimeException	Unchecked	
5) NoClassDefFoundError	Error	Unchecked	
6) ExceptionInInitializerError	Error	Unchecked	
7) IllegalArgumentException	RuntimeException	Unchecked	Programmer/ API Developer
8) NumberFormatException	IllegalArgument	Unchecked	
9) IlleaglStateExcepiton	RuntimeException	Unchecked	
10) AssertionError	Error	Unchecked	

6

ASSERTIONS

Introduction

Assertions has introduced in 1.4 version. The main objective of assertions is to perform ***debugging***.

The traditional way of debugging is to use System.out.println's. But the main disadvantage of this approach is compulsory we should remove these S.O.P's after fixing the problem other wise these will execute at run time. Which may effect performance of the system. It may reduce readability of the code and disturbs logging mechanism.

To resolve all these problems sun people has introduces Assertions concept in 1.4 version.

The main advantage of assertions is we can enable or disable assertions based on our requirement. But by default assertions are disabled.

Assertions concept we have to use in Test environment or in Development environment but not in the production.

Identifier Vs Keyword

Assert keyword has introduced in 1.4 version hence from 1.4 version on words we are not allowed to use assert as identifier.

Ex:-

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

in 1.4 or 1.5 if we compile
javac assert.java then we will get the following C.E as of release 1.4 assert is a keyword and may not used as an identifier.

```
javac –source 1.3 assert.java
java Test
```

Types of assert statement

There are 2 types of assert statement

1) simple assert.
2) Augmented assert.

Simple assert

Syntax: assert <*boolean expression*> ;

assert(b);

If b is true then normal continuation follows. Else the program will terminate abnormally can rise assertion error.

Ex:-

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

1) javac Test.java
2) java Test
   in this case assert statement won't be executed because it is disable by default.
3) java –ea Test

   Then generates assertion error.

Augmented Version

Syntax: assert <*boolean expression*> : <*message expression*> ;

Assert e1:e2;

'e1' → should be boolean type.
'e2' → any thing is allowed including method calls also

Ex:-

```
class Test
{
    public static void main(String[] args)
    {
        int x = 10;
            ;
            ;
        assert(x>10):"here the value of x should be > 10 but it is "+x;
        System.out.println(x);
    }
}
```

javac Test.java
java –ea Test

O/P:-

```
D:\rjava\scjpExamples>java -ea Test
Exception in thread "main" java.lang.AssertionError: here the value of x should
be > 10 but it is 10
        at Test.main(assert.java:8)
```

Note: assert e1:e2

Here 'e2' will be executed iff 'e1' is false.

Ex:-

```
class Test
{
    public static void main(String[] args)
    {
        int x = 10;
            ;
            ;
        assert(x==0):++x;
        System.out.println(x);
    }
}
```

javac Test.java

java –ea Test

O/P:-

```
D:\rjava\scjpExamples>java -ea Test
Exception in thread "main" java.lang.AssertionError: 11
        at Test.main(assert.java:8)
```

→ assert(e1):e2;

for e2 any thing is allowed including method calls also But void return type method calls

are not allowed. Violation leads to compile time error.

Ex:-

```
class Test
{
    public static void main(String[] args)
    {
        int x = 10;
        ;
        ;
        assert(x>0):m1();
        System.out.println(x);
    }
    public static void m1()
    {
        return;
    }
}
```

javac Test.java

java –ea Test

O/P:-

```
D:\rjava\scjpExamples>javac assert.java
assert.java:8: 'void' type not allowed here
                assert(x>0):m1();
                            ^
```

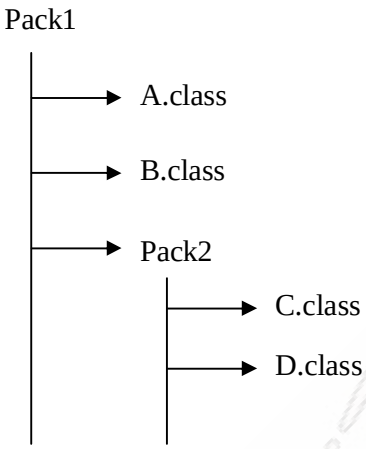
Various Run Time Flags

1) *-ea* → To enable assertions in every non-system class(i.e user defined class)
2) *-enableassertions* → To enable assertions in every non-system class(Exactly similar to *-ea*)
3) *-da* → To disable assertions in every non-system class.

4) *-isableassertions* → To disable assertions in every non-system class.(Exactly similar to *-da*)
5) *-esa* → To enable assertions in every system class
6) *-enablesystemassertion* → similar to *-esa*
7) *–dsa* → To disable assertions in every system class.
8) *-disablesystemassertions* → similar to '-dsa'

Ex:-
java *–ea –esa –da –dsa –ea* Test

All the flags will execute from **left to right** and there is no priority difference b/w enabling and disabling



→ To enable assertions only in the 'A- Class'
java -ea:Pack1.A

→ To enable assertions only in B and D
java -ea:Pack1.B -ea:Pack1.Pack2.D

→ To enable assertion in all classes of Pack1 and its sub package classes also.
java –ea:Pack1…

→ To enable assertions in all classes present in pack1 but not those present in pack2
java -ea:Pack1… -da:Pack1.Pack2…

Note:- we can enable assertions either class wise or package wise also.

Proper and Improper Use of assertions

1) It is improper to use assert statement for validating the arguments of a public method.

```
public void withdraw(double amount)
{
    assert(amount >= 100);
}
```

C.E:- No C.E, R.E it is improper use.

2) It is improper to use assertions for validating command line argument also, because these are arguments to public main method.

3) It is proper to use assertions for validating private method argument.
4) It is improper to mix programming language with assert statement.

5) In our code if there is any place where the control is not allowed to reach. It is the best place to use the assert statement.

```
Ex:-  switch (month)
      {
            case 1:
                  ....
                  ....
            case 2:
                  ....
                  ....
            case 3:
                  ....
                  ....
            case 4:
                  ....
                  ....
            :
            :
            :
            case 12:
                  ....
                  ....
            default:
                  assert(false);

            }
```

AssertionError

It is the child class of Error and it is unchecked.
It is not recommended to catch AssertionError by using catch Block. It is stupid type of activity.

```
Ex:-
      class Test
      {
            public static void main(String arg[])
            {
                  int x = 10;
                        ;
                        ;
                        ;
                  try
                  {
                        assert(x>10);
                  }
                  catch (AssertionError e)
                  {
                  System.out.println("I am stupid...Because I am catching
                   Assertion");
                  }
            }
      }
```

O/P:-

```
D:\rjava\scjpExamples>javac assert.java

D:\rjava\scjpExamples>java -ea Test
I am stupid...Because I am catching Assertion
```

7

OO CONCEPTS

1) Data hiding
2) Abstraction
3) Encapsulation
4) Tightly Encapsulated class
5) Is – A Relation Ship
6) HAS – A Relation ship
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) typecasting

Data Hiding

The data should not go out directly i.e outside person is not allowed to access the data this is nothing but "*Data Hiding*".
The main advantage of data hiding is we can achieve security.
By using private modifier we can achieve this.

Ex:

```
Class datademo
{
        private  double amount;
        ……
}
```

It is highly recommended to declare data members with private modifier.

Abstraction

Hiding implementation details is nothing but abstraction. The main advantages of abstraction are we can achieve **security** as we r not highlighting internal implementation.
Enhancement will become easy. With out effecting outside person we can change our internal implementation.

It improves **maintainability**.

Note:- 1) If we *don't know about implementation* just we have to represent the specification then we should go for ***interface***

2) If we don't know about complete implementation just we have partial implementation then we should go for ***abstract.***

3) If we know complete implementation and if we r ready to provide service then we should go for concrete class

Encapsulation

If a class follows data hiding and abstraction such type of class is said to be 'Encapsulated' class.

Encapsulation = Data Hiding + Abstraction

Ex:

```
class Account
{
        private int balance;
        public void setBalance(int balance)
        {
                //validating the user & his permissions.
                this.balance = balance;
        }
        public int getBalance()
        {
                //validating the user and his permissions.
                return balance;
        }
}
```

The data members we have to declared as private. So that outside person is not allowed to access to directly we have to provide Access to our data by defining setter and getter methods. i.e hiding data behind methods is the central concept of encapsulation.

The main advantages of encapsulation are security, enhancement, maintainability.

Tightly Encapsulated Class

A class is said to be tightly encapsulated iff all the data members declared as private.

Ex:-

```
class student
{
        private String name;
        public void setName(String name)
        {
                this.name = name;
        }
        public String getName()
        {
                return name;
        }
}
```

Q) which of the following classes are tightly encapsulated classes?

```
class x
{
        private int i =10;
        private int getI();
        {
```

✓

```
                return i;
        }
}

class y extends x
{
        public int i = 30;        X
}

class z extends x
{
        private int k = 40;       ✓
}
```

Q)

```
class x
{
int i = 0;
}

class y extends x
{
private int j = 20;
}

class z extends y
{
private int k = 30;
}
```

no class is tightly encapsulated if the parent class is not tightly encapsulated then no child class is tightly encapsulated.

IS – A RelationShip

- Also known as 'inheritance'.
- By using extends keyword we can implement inheritance.
- The main advantage is reusability.

Note:- parent class reference can be used to hold child class object but by using that reference we r not allowed to call child class specific methods.

```
class test
{
public static void main(String arg[])
{
p p1 = new p();
p1.m1();
p1.m2();
p1.m3();        →C.E: Child class methods are not available to the parent class.

c c1 = new c();
c1.m1();
c1.m2();
c2.m3();

p p2 = new c();
p2.m1();
```

```
p2.m2();
p2.m3();        →C.E: By using parent class reference we can't call child class specific
                  method

c c4 = new p() →C.E: Child class reference can't be used to hold parent class object.
}
}
```

A class can extend only one class at a time but an interface can extend any no of interfaces simultaneously.

HAS – A RelationShip

- Also known as Composition or Aggregation .
- There is no specific keyword, but most of the cases we can implemented by using new keyword.
- Main advantage is reusability.

Ex:

```
class engine
{
        m1(){}
        m2(){}
        .
        .
        .
}
class car
{
        engine e = new engine();
                .
                .
}
```

Class car has engine.

HAS – A relationship increases dependency b/w components and creates maintainability problems.

Method Signature

In java method signature consists of method name and arguments list(including order also)

```
public void m1(int i, float f)
            {
            }
```

method signature

Compiler uses method signature to resolve method calls.

Two methods with the same signature are not allowed in any java class, violation leads to compile time error.

Ex:

```
class Test
{
        public void m1(int i)
        {
        }
        public void m2()
        {
        }
        public int m1(int i)
        {
        }
```

```
}
```
C.E:-m1(int) is already defined in Test

In java return type is not part of method signature.

OverLoading

Two methods r said to be overloaded iff the method names are same, But arguments are different(Atleast Order) Lack of overloading increases complexity of the program in the case of 'c' language.
For the same requirement we should maintain different method names if arguments are same.

In C

abs(int i) → only for int arguments.
fabs(float f) → only for float arguments.
labs(long l) → only for long arguments.

But in **java** two methods with the same name is allowed eventhough arguments r different.

```
abs(int i)
abs(float f)
abs(long l)
```

Ex:-

Case1:

```
class Test
{
    public void m1()
    {
        System.out.println("no-args");
    }
    public void m1(int i)
    {
        System.out.println("int-args");
    }
    public void m2(double d)
    {
        System.out.println("double-args");
    }
    public static void main(String[] args)
    {
        Test t = new Test();
        t.m1();
        t.m1(10);
        t.m1(10.5);
    }
}
```

The overloading method resolution is the responsibility of compiler based on reference type and method arguments. Hence overloading is considered as compile-time polymorphism or EarlyBinding.

Automatic promotion in overloading

Ex:-

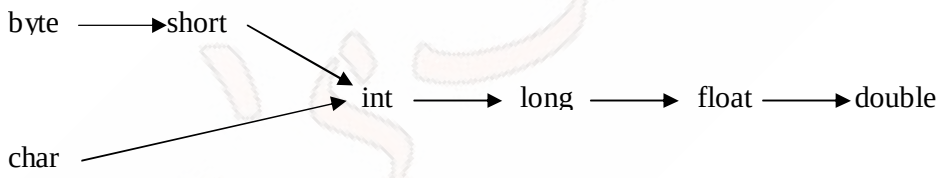
Case2:

```
class Test
{
    public void m1()
    {
        System.out.println("no-args");
    }
    public void m1(int i)
```

```
        {
                System.out.println("int-args");
        }
        public void m1(float f)
        {
                System.out.println("float-args");
        }
        public static void main(String[] args)
        {
                Test t = new Test();
                t.m1();
                t.m1(10);
                t.m1(10l);
                t.m1(10f);
                t.m1('a');
                t.m1(10.5);
        }
}
```

In the case of overloading if there is no method with the required argument then the compiler won't raise immediately compile time error. First it will promote arguments to next level and checks is there any matched method with promoted arguments, if there is no such method compiler will promote the argument to the next level and checks for the matched method. After all possible promotions still the compiler unable to find the matched method then it raises compile time error.

The following is the list of all possible Automatic promotion.



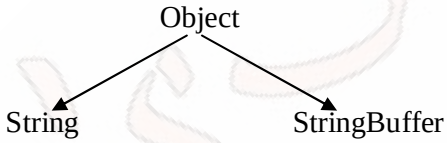
Ex:-

```
class Test
{
        public void m1(String s)
        {
                System.out.println("String Version");
        }
        public void m1(Object o)
        {
                System.out.println("Object Version");
        }
        public static void main(String arg[])
        {
                Test t = new Test();
                t.m1("raju");           →String Version
                t.m1(new Object());     →Object Version
                t.m1(null);
        }
}
```

In the case of overloading the more specific version will get the chance first.

Ex:-

```
Case3:
class Test
{
        public void m1(String s)
        {
                System.out.println("String Version");
        }
        public void m1(StringBuffer sb)
        {
                System.out.println("StringVersion");
        }
        public static void main(String arg[])
        {
                Test t = new Test();
                t.m1("raju");
                t.m1(null);
        }
}
```



So it produces Ambiguity error.

Note:- we can define a single method which can be applicable for any type of primitive argument, as follows

```
m1(double d)
```

By automatic promotion instead of double we can give char, byte, short, int, long, float, double.
Based on the same approach square root method in math class is declared.

```
public static double sqrt(double d)
```

If we want to declare a method which can be applicable for byte, short, char, int, long we should declare as follows.

```
m1(long)
```

Ex:-

```
class Test
{
        public void m1(int... i)
        {
                System.out.println("var-arg");
        }
        public void m1(int i)
        {
                System.out.println("int-arg");
        }
        public static void main(String arg[])
```

```
        {
            Test t = new Test();
            t.m1();         //var-arg
            t.m1(10,20);    //var-arg
            t.m1(10);       //int-arg
        }
    }
```

var-arg method will always get least priority i.e if no other method matched then only var-arg method will get chance for execution.

Ex:-

```
class Test
{
    public void m1(int i, float f)
    {
        System.out.println("int, float");
    }
    public void m1(float f, int i)
    {
        System.out.println("float, int");
    }
    public static void main(String arg[])
    {
        Test t = new Test();
        t.m1(10.5f, 10);
        t.m1(10, 10.5f);
        t.m1(10, 10);   //      C.E : reference to m1() is ambiguous
        t.m1(10.5f, 10.5f);//   C.E : can't find symbol method m1(float,float)
    }
}
```

Ex:

```
class Animal
{
}
class Monkey extends Animal
{
}
class Test
{
    public void m1(Animal a)
    {
        System.out.println("Animal Version");
    }
    public void m1(Monkey m)
    {
        System.out.println("Monkey Version");
    }
    public static void main(String arg[])
    {
        Test t = new Test();
Case1:  Animal a = new Animal();
        t.m1(a);        //Animal Version

Case2:  Monkey m = new Monkey();
        t.m1(m);        //Monkey Version
```

```
Case3:      Animal a1 = new Monkey();
            t.m1(a1);       //Animal Version
        }
}
```

Overloading method resolution will always take care by compiler based on the reference type but not based on runtime object.

Overriding

What ever the parent has by default available to the child class through inheritance, If the child class is not satisfied with the parent class implementation then the child is allowed to overwrite that parent class method to provide it's own specific implementation, this concept is nothing but *"overriding"*.

Ex:

```
class  P
{
        public void property()
        {
                System.out.println("Land, gold, cash");
        }
        public void mary()
        {
                System.out.println("Subbalakshmi");
        }
}
class C extends P
{
        public void mary()
        {
                System.out.println("Priyanka");
        }
}
class Test
{
        public static void main(String[] args)
        {
                P p1 =  new P();
Case1:          p1.mary();      //Subbalakshmi

                C c1 =  new C();
Case2:          c1.mary();      //Priyanka

                P p2 =  new C();
Case3:          p2.mary();      //Priyanka
        }
}
```

Overriding method resolution will take care by JVM based on runtime Object. Hence overriding is considered as runtime polymorphism or dynamic polymorphism or latebinding.
The process of overriding method resolution is also known as "*dynamic method dispatch*".

Rules for overriding

1) The method names and arguments(including order) must be same. i.e signatures of the methods must be same.

2) In overriding the return types must be same but this rule is applicable only until 1.4 version but from 1.5 version onwards *co-variant return types* also allowed. i.e the child class method return type need not to be same as parent class method return type it's child class also allowed.

Ex:

Parent : Object(covariant returntype).
Child : String, StringBuffer, Object….. ✓

Parent : String
Child : Object X

Because String to Object is not a covariant.

Parent : double
Child : int X

Because co-variant return types are not applicable for primitive types.

Which of the following are valid overriding methods……?

Parent : public void m1()
Child : public void m1() ✓

Parent : public int m1()
Child : public long m1() X

Parent : public Object m1()
Child : public String m1() ✓

Parent : public String m1()
Child : public Object m1() X

3) final method can't be overridden in child classes. Private methods are not visible in the child classes. Hence they won't participate in overriding. Based on our requirement we can take exactly same declaration in child class, But It is not overriding.

Ex:

```
class P
{
        private void m1()
        {
        }
}
class C extends P
{
        private void m1()
        {
        }
}
```

Here it is not overriding child class specific method.

4) native methods can be overridden as non-native, similarly we can override abstract methods, synchronized methods.
We can override a non-abstract method as abstract also

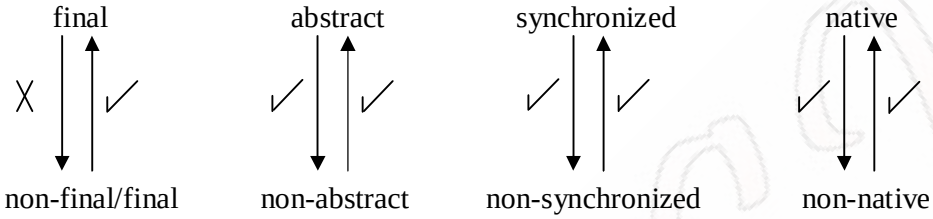
```
class P
{
        public  void m1()
        {
        }
}
abstract class C extends P
{
        public abstract void m1();
}
```

✓

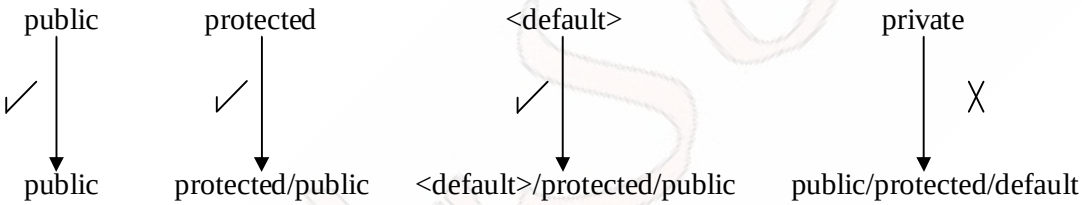
```
abstract class P
{
        public abstract void m1();
}
class C extends P
{
        public  void m1()
        {
        }
}
```

✓

The following are possible types



5) while overriding we are not allowed o decrease access privileges. Otherwise compile time error but we can increase access privileges. The following is the list of valid with respect to access privileges.



6) while overriding the size of checked exceptions we are not allowed to *increase* in throws class but we can *decrease* the size of checked exceptions. But there are no restrictions on unchecked exceptions.

Overriding in static methods

A static method can't be overridden as non-static

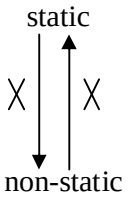
Ex:

```
class P
{
        public static void m1()
        {

        }
}
class C extends P
{
        public void m1()
        {
        }
}
```

O/P:-

```
E:\Rjava\SCJP\PracticePrg>javac OverrideDemo.java
OverrideDemo.java:79: m1() in C cannot override m1() in P; overridden method is
static
        public void m1()
                    ^
1 error
```

Similarly a non-static method can't be overridden as static method.



If both parent and child class methods are static then there is no compile time error or run time error it seems that overriding is happened but it is not overriding this concept is called "**method hiding"**.
All the rules of method hiding are exactly similar to overriding, except both methods declared as static.
In the case of method hiding method resolution will take care by compiler based on reference type(But not runtime object).

Ex:

```
class P
{
        public static void m1()
        {
                System.out.println("parent method");
        }
}
class C extends P
{
        public static void m1()
        {
                System.out.println("child method");
        }
}
class Test
{
        public static void main(String arg[])
        {
                P p = new P();
Case1:          p.m1();         //parent method

                C c = new C();
Case2:          c.m1();         //child method

                P p1 = new P();
Case3:          p1.m1();        //parent method
        }
}
```

In the case of method hiding the method resolution will take care by compiler based on reference type. Hence *method hiding* is considered as '*static polymorphism*' or '*compile time polymorphism*' or '*early binding*'.

Overriding in the case of Variable

Overriding concept is not applicable for variables. And it is applicable only for methods.
Variable resolution always takes care by compiler based on reference type.

Ex:

```
class P
{
        int i = 888;
```

```
}
class C extends P
{
        int i = 999;
}
class Test
{
        public static void main(String arg[])
        {
                P p = new P();
Case1:          System.out.println(p.i);        //888

                C c = new C();
Case2:          System.out.println(c.i);        //999

                P p1 = new C();
Case3:          System.out.println(p1.i);       //888
        }
}
```

Comparison between Overloading and Overriding

Property	Overloading	Overriding
1) Method names	Same	Same
2) Arguments	Different(at least order)	Same(including order)
3) signature	Different	Same
4) return type	No rule or no restriction	Must be same until 1.4 version. But from 1.5 version onwards co-variant return types are allowed
5) throws clause	No restrictions	The size of the checked exception should not be increased. But no restrictions for unchecked exceptions.
6) Access Modifier	No restrictions	Should be decreased
7) Private/final/static	Can't be overloaded	Can't be overridden
8) Method resolution	Method resolution takes care by compiler based on reference type and arguments	Takes care by JVM based on runtime object
9) Other names	Compile time (or) static polymorphism (or) early binding	Runtime (or) dynamic polymorphism (or) late binding

Consider the following parent class method declaration….!

Parent : public void m1(int i)

Which of the following methods are allowed in the child class.

Child :

1) private void m1(float f)throws Exception ✓ (overloading)
2) public void m1(int i)throws classCastException ✓ (overloading)
3) public final void m1(int i) ✓ (overloading)
4) private abstract void m1(long l)throws Exception X (private & abstract)
5) public synchronized int m1() throws IOException ✓ (over loading)
6) public int m1(int i) X
7) public void m1(int i)throws Exception X (Exception Increased)

Static Control Flow

```
class StaticDemo
{
    1   static int i = 0;                                   7
    2   static
        {
                m1();                                       8
                System.out.println("First Static Block");    10
        }
    3   public static void main(String[] args)
        {
                m1();                                       13
                System.out.println("Main Method");           15
        }
    4   public static void m1()
        {
                System.out.println(j);                      9  14
        }
    5   static
        {
                System.out.println("Second Static Block");   11
        }
    6   static int j = 20;                                  12
}
```

O/P:-

```
0
First Static Block
Second Static Block
20
Main Method
```

Process of static control flow

1) Identification of static members from top to bottom
   - i = 0 (RIWO) Read Indirect Write Only
   - j = 0 (RIWO)
   - (1-6) Steps.
2) Execution of static variable assignments and static block from top to bottom
   - i = 0 (R & W)
   - j = 0 (R & W)
   - (7-12) Steps.
3) Execution of Main Method.
   - (12-15) Steps.

If the variable is in RIWO state then we r not allowed to perform read operation directly, violation leads to C.E Saying "*Illegal forward reference*".

static Blocks

If we want to perform some activity at the time of class loading, Then we should define that activity at static blocks because these(static blocks) will execute at the time of class loading only.

We have to load native libraries at the time of class loading. Hence we have to define this activity inside the "*static block*".

Ex:

```
class Native
{
        static
        {
        System.loadLibrary("native Library path");
        }
}
```

After loading database driver compulsory we should register with the driver manager. But it is not required to perform explicitly this registration because in every driver class there should be one static block to perform this registration.

Hence at the time of loading the database driver automatically registration will be performed.

Ex:

```
class DatabaseDriver
{
        static
        {
                //Register Driver with Driver Manager
        }
}
```

Case1:- with out using main method we can able to print some statement to the console.

Ex:

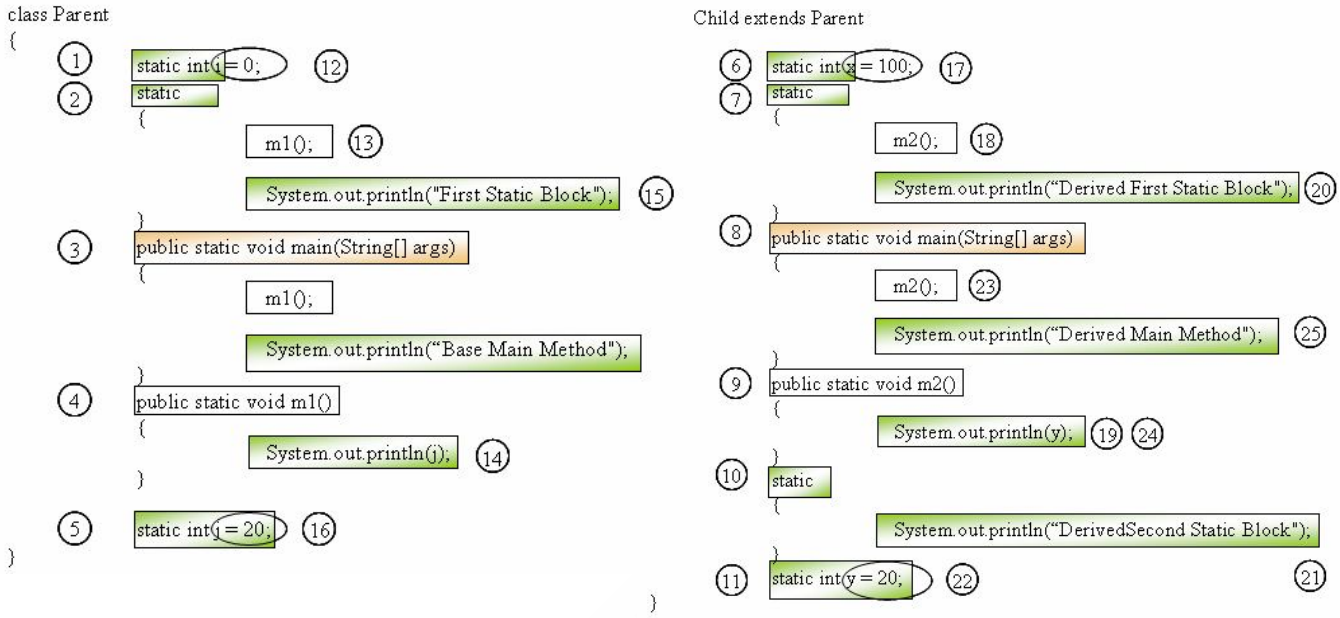
```
class StaticDemo
{
        static
        {
                System.out.println("Hello....I can print  ");
                System.exit(0);
        }
}
```

With out using main method, static block still we are able to print some statement on the console.

Ex:

```
class StaticDemo
{
        static int i = m1();
        public static int m1()
        {
                System.out.println("Hello ....I am able to print");
                System.exit(0);
                return 1;
        }
}
```

static control flow in parent and child classes



E:\> javac StaticDemo.java

Then it generates two class files Parent.class and Child.class.

→ **What is the O/P if I am execute java Child?**

Process:

1) Identification of static members from parent to child.

i = 0(RIWO)
j = 0(RIWO)
x = 0(RIWO)
y = 0(RIWO)

(0 - 11) steps.

2) Execution of static variable assignments and static blocks from parent to child.

i = 10(R&W)
j = 20(R&W)
x = 100(R&W)
y = 200(R&W)

(12 - 22) steps.

3) Execution of child class main method.

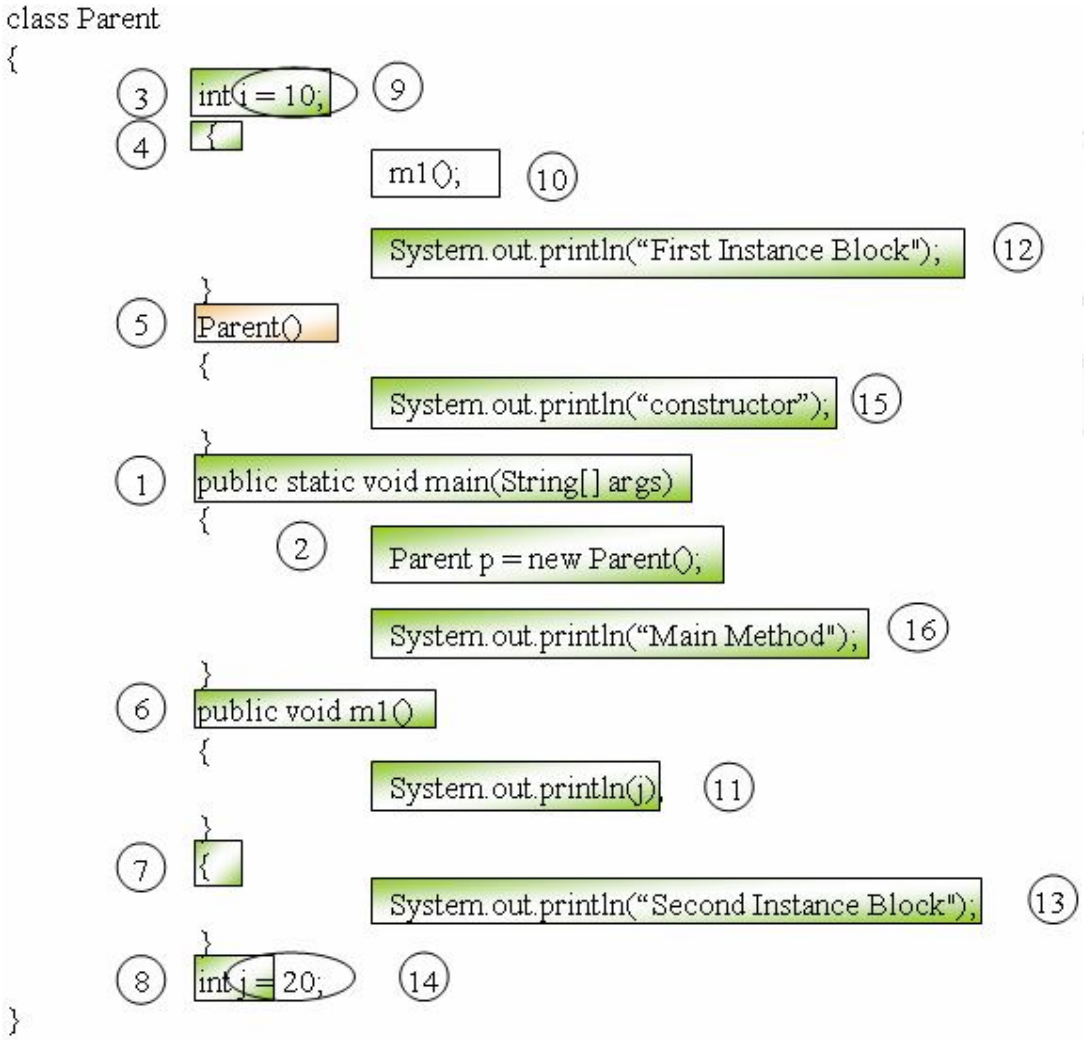
O/P:- 0
Base static Block
0
Derived First static Block
Derived Second static Block
200
Derived Main Method

→ If we execute java Parent

Executes only Parent class members.

If we remove main method in child, parent main is available because it is not overriding concept.

Instance Control Flow



O/P:-

```
---------- Run Java Program ----------
0
First Instance Block
Second Instance Block
constructor
Main Method
```

Process of static control flow

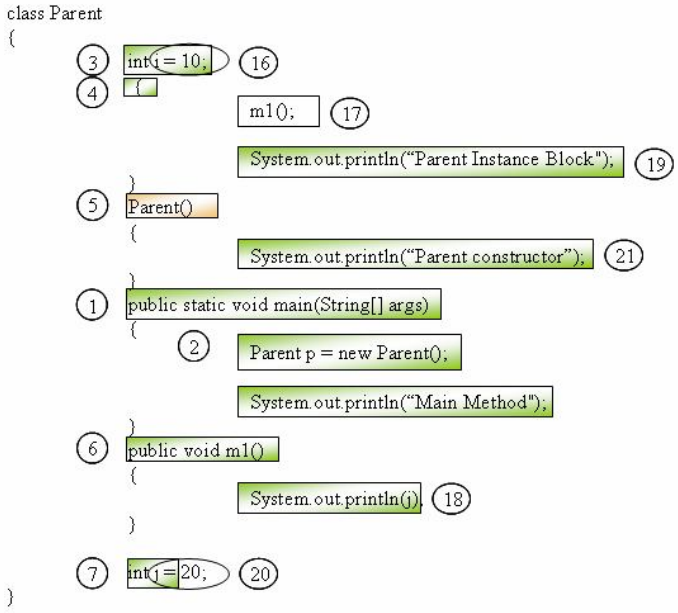
Instance control flow will execute at the time of object creation the following is the sequence of statements which will execute at the time of object creation.

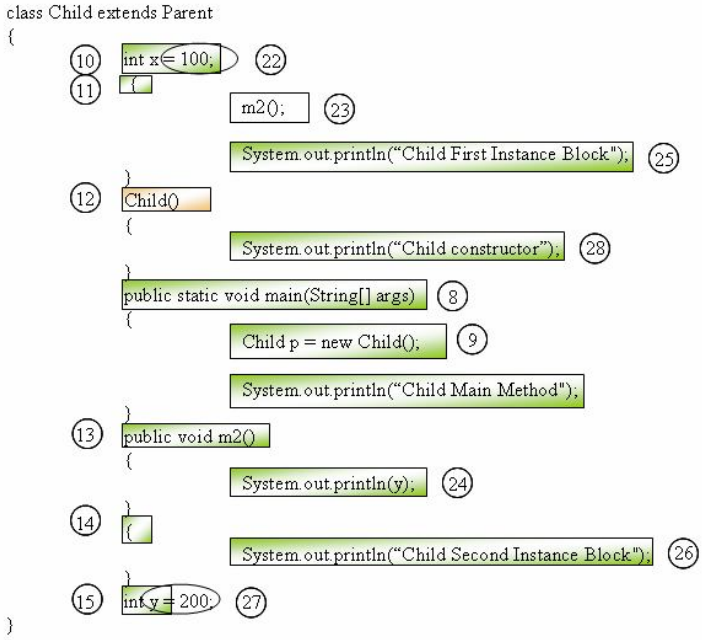
1) Identification of Instance members
   - i = 0 (RIWO) Read Indirect Write Only
   - j = 0 (RIWO)
   - (1-8) Steps.
2) Execution of Instance variable assignments and Instance block from top to bottom
   - i = 0 (R & W)
   - j = 0 (R & W)
   - (9-14) Steps.
3) Execution of Main Method.
   - (15) Steps.

Note:- Static control flow is only one time activity and that will execute at the time of class loading. But instance control flow is not one time activity. And it will execute for every object creation separately.

If the variable is in RIWO state then we r not allowed to perform read operation directly, violation leads to C.E Saying "*Illegal forward reference*"

Instance control flow in parent and child classes





E:\> javac StaticDemo.java

Then it generates two class files Parent.class and Child.class.

→ What is the O/P if I execute java Child?

Process:

1) Identification of Instance members from parent to child.
   - i = 0(RIWO)
   - j = 0(RIWO)
   - x = 0(RIWO)
   - y = 0(RIWO)

   (1 - 15) steps.
2) Execution of Instance variable assignments and instance blocks only in parent class.
   - i = 10(R&W)
   - j = 20(R&W)

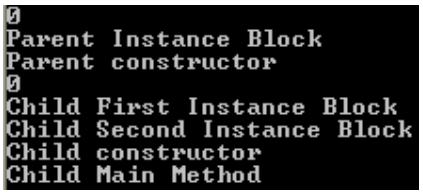
   (16 - 20) steps.
3) Execution of Parent class constructor.

   21 step
4) Execution of Instance variable assignments and instance blocks in the child class
   - x = 100
   - y = 200

   (22-27) steps.
5) Execution of child class constructor.

   28 step

O/P:-



Note:-

Object Creation is the most costly operation. Unnecessarily we r not allowed to create more no of Objects. It will effect performance of the system.

Constructor

After Creation of Object compulsory we should perform initialization then only that object in a position to provide service to others.
At the time of Object Creation some peace of code will execute automatically to perform initialization that peace of code is nothing but "*Constructor*".
Hence the main Objective of constructor is to perform initialization.
If we don't have constructor we will face burden. For example see the following code.

Ex:

```
class Student
{
	String name;
	int rollno;

	public static void main(String[] args)
	{
		Student s1 = new Student();
		Student s2 = new Student();
		Student s3 = new Student();
		Student s4 = new Student();
		s1.name = "Malli";s1.rollno = 101;
		s2.name = "Sankar";s2.rollno = 102;
		s3.name = "Kiran";s3.rollno = 103;
		s4.name = "Sai";s4.rollno = 104;
		System.out.println(s1.name+"-----"+s1.rollno);
		System.out.println(s2.name+"-----"+s2.rollno);
		System.out.println(s3.name+"-----"+s3.rollno);
		System.out.println(s4.name+"-----"+s4.rollno);
	}
}
```

To over come this type of burden constructor was introduced.

Ex:-

```
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("raju",101);
		Student s2 = new Student("mani",102);
```

```
                System.out.println(s1.name+"-----"+s1.rollno);
                System.out.println(s2.name+"-----"+s2.rollno);
        }
}
```

O/P:-

```
raju-----101
mani-----102
```

Rules for Constructor

While writing constructors we should follow the following rules.

1) The name of the constructor and name of the class must be same.
2) The only allowed modifiers for the constructors are public, private, protected, <default>. If we are using any other modifier we will get C.E(Compiler Error).

Ex:

```
class Test
{
        static Test()
        {
                ----
        }
}
```

C.E:- modifier static not allowed here.

3) return type is not allowed for the constructors even void also. If we r declaring return type then the compiler treats it as a method and hence there is no C.E and R.E(RuntimeError).

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

It is legal(But Stupid) to have a method whose name is exactly same as class name. Here it was treated as method.

Default Constructor

If we r not writing any constructor then the compiler always generate default constructor.
If we r writing at least one constructor then the compiler won't generate any constructor.
Hence every class contains either programmer written constructor or compiler generated default constructor but not both simultaneously.

Prototype of default constructor:

1) It is always no-arg constructor.
2) It contains only one – line **super();**
   This is a call to superclass – constructor it is no-argument call only.
3) The modifier of the default constructor is same as class modifier(either public or default).

Programmers Code	Compiler Generated Code

class Test
{
 	
}
	class Test
	{
	Test()
	{
	super();
	}
	}

public class Test
{
 	
}
	public class Test
	{
	public Test()
	{
	super();
	}
	}

class Test
{
private Test()
{
}
}
	class Test
	{
	private Test()
	{
	super();
	}
	}

class Test
{
void Test()
{
}
}
	class Test
	{
	void Test()
	{
	}
	Test()
	{
	super();
	}
	}

class Test
{
Test()
{
this(10);
}
Test(int i)
{
}
}
	class Test
	{
	Test()
	{
	this(10);
	}
	Test(int i)
	{
	super();
	}
	}

The first line inside any constructor must be a call to super class constructor(super()) or a call to overloaded constructor of the same class(this()). If we are not taking any thing then the compiler will always place super()

super() & this() in constructor

we should use as first statement in constructor. We can use either super or this but not both simultaneously. Outside constructs we can't use i.e we can invoke a constructor directly from another constructor only.

Case1:

```
class Test
{
        Test()
        {
                super()
                System.out.println("constructor")
        }
}
```

Case2:

```
class Test
{
        Test()
        {
                System.out.println("constructor")
                super();
        }
}
```

C.E: Call to super() must be first statement in constructor

Case3:

```
class Test
{
        Test()
        {
                super();
                this(10);
                super("Constructor");
        }
        Test(int i)
        {
                System.out.println("Constructor")
        }
}
```

C.E: Call to to this must be first statement in

Case4

```
class Test
{
        public void m1()
        {
                super();
        }
}
```

C.E: Call to to Super must be first statement in

Overloaded Constructor

A class can contain more than one constructors with different arguments. This type of constructors are called "***overloaded constructor***".

Ex:-

```
class Test
{
        Test()
        {
                this(10);
                System.out.println("No-arg constructor");
        }
        Test(int i)
        {
                this(10.5);
                System.out.println("int-arg");
        }
        Test(double d)
        {
                System.out.println("double-arg")
        }
        public static void main(String arg[])
        {
                Test t1 = new Test();
                Test t2 = new Test(10);
```

```
                Test t3 = new Test(20.5);
                Test t4 = new Test('a');
                Test t5 = new Test(10l);
            }
        }
```

Inheritance concept is not applicable for constructor and hence overriding is also not applicable.

```
class Test
{
    public static void m1()
    {
        System.out.println("m1 method");
        m2();
    }
    public static void m2()
    {
        System.out.println("m2 method");
        m1();
    }
    public static void main(String arg[])
    {
        m1();
        System.out.println("Hello......hai");
    }
}
```

No Compilar Error and No Runtime Error

```
class Test
{
    Test()
    {
        this(10);
    }
    Test(int i)
    {
        this();
    }
    public static void main(String arg[])
    {
        System.out.println("Hai.....hello");
    }
}
```

C.E: Recursive constructor invokation

Recursive Method invocation is runtime exception saying stack over flow error.

✓
```
class p
{
}
class c extends p
{
}
```

✓
```
class p
{
    p()
    {
    }
}
class c extends p
{
}
```

```
class p
{
    p(int i)
    {
    }
}
class c extends p
{
}
```

C.E:

Because super calls no-argument constructor in parent.
when ever we r writing any constructor. It is remanded to provide no-arument constructor also. If the parent class contains some constructors then while writing child class constructor we should take a bit care.

If the parent class constructor *throws* some *checked exception*. Compulsory the child class constructor should throw the same checked exception or it's parent other wise compile time error.

```
class p
{
}
class c extends p
{
}
```

✓

```
class p
{
        p()
        {
        }
}
class c extends p
{
}
```

✓

```
class p
{
        p() throws IOException
        {
        }
}
class c extends p
{
        c()
        {
                super();
        }
}
```

If we r not declaring this then we will get C.E saying UnReported Exception in default constructor

If the parent class constructor *throws unchecked exception* then child class constructor not required to throw that exception

Ex:

```
class p
{
        p() throws ArithmeticException
        {
        }
}
class c extends p
{
        c()
        {
                super();
        }
}
```

Which of the following statements are true?

The first line inside constructor either super() or this() but not both simultaneously. ✓

If we are not taking either super() or this() as the first line in the constructor then the compiler will always place this(). ✗

Compiler will always generate default constructor. ✗

Constructor overriding is possible but overloading is not possible ✗

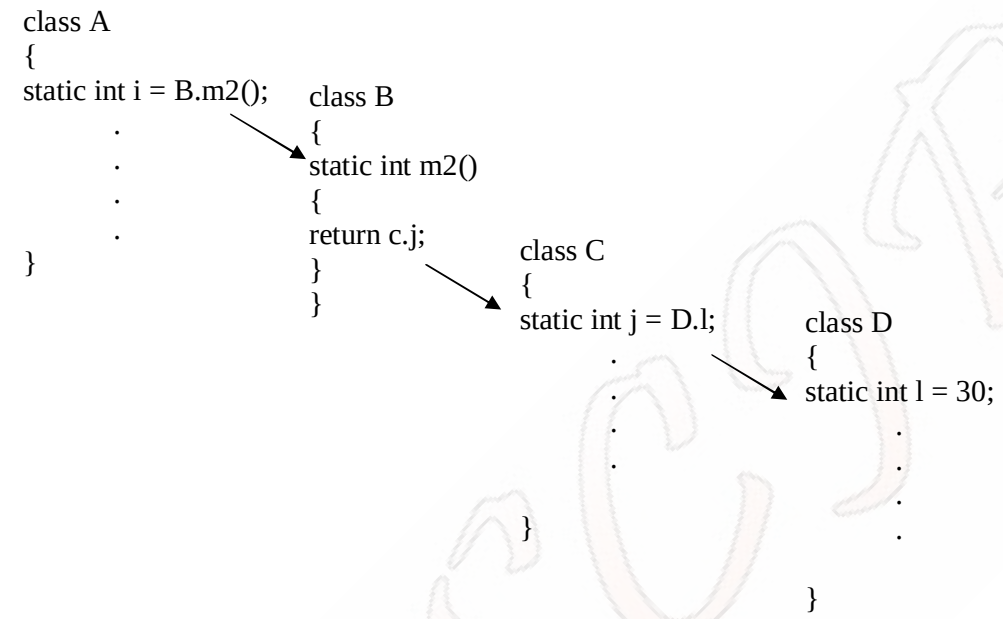
Even for constructor also inheritance possible ✗

If the parent class constructor throws some checked exception compulsory child class constructor should throw same exception or its parent. ✓

Coupling

It is also one of the feature of OOPs. The degree of dependency between the components is called *coupling*.

Ex:



These components are tightly coupled with each other. With out effecting any component we can't modify any component. This type of programming is not recomanded.
When ever we are developing the components compulsory we should maintain very less dependency between the components i.e we have to maintain loosely coupling.

The main advantages of loosely coupling are:

1) It imporves maintainability.
2) It makes enhancements easy.
3) It produces reusability.

Cohesion

For every component we have to define a well defined **single purpose**. Such type of components are said to be followed high "*cohesion*".

Ex:-
If we define a single servlet for all required functionalities of mail application like displaying login page, Validating the user, Displaying Inbox page etc…

Then for every small change entire component will be disturbed and it reduces maintainability and suppresses reusability of code.

To resolve this problem. For every task we can define a separate component like login.jsp for displaying login page, validate servlet for validation purpose, inbox servlet, For displaying mails etc….

By using this approach we can perform the modification in any component with out effecting the remaining. And it imporves maintainability. It also promotes reusability. When ever validation required the same validate servlet we can use.

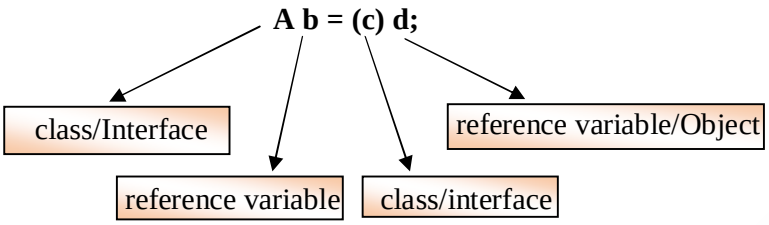
The main advantages of High Cohesion are:

1) It imporves maintainability.

2) Enhancements is easy.
3) Promotes reusability.

typeCasting

General syntax of typecasting



For type casting we have to check at 3 regains those are

Compile time checking 1

There should be some relation ship between the type of 'd' and 'c'. other wise we will get compile time error saying *inconvertable* types

found : d
required : c

Ex:

String s = "raju";
Object o = (Object)s; ✓

Because Object is parent to String

String s = "raju";.
Object o =(StringBuffer)s; ✗

```
TypeCastDemo.java:8: inconvertible types
found   : java.lang.String
required: java.lang.StringBuffer
                Object o = (StringBuffer)s;
                           ^
```

Compile time checking 2

'c' must be same or derived type of 'A' other wise compile time error. Saying *incompatable* types

found : c
required : A

Ex:

String s = "raju";
Object o = (String)s; ✓

Because String is child of Object.

String s = "raju";
StringBuffer sb = (Object)s; ✗

```
TypeCastDemo.java:9: incompatible types
found   : java.lang.Object
required: java.lang.StringBuffer
                StringBuffer sb = (Object)s;
                                  ^
```

Runtime Checking

The internal object type of 'd' must be same or derived type of 'c' other wise we will get RuntimeException saying

ClassCastException : 'd' type

Ex:

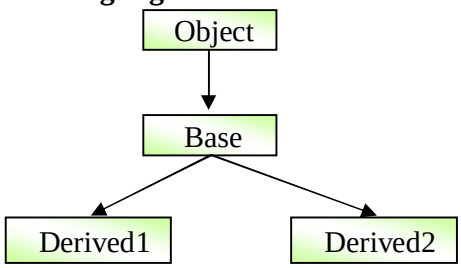
```
Object o = new String("raju");
StringBuffer sb = (StringBuffer)o;
```

```
Exception in thread "main" java.lang.ClassCastException: java.lang.String
        at TypeCastDemo.main(TypeCastDemo.java:11)
```

```
Object o = new String("raju");
String s = (String)o;
```
✓

Because internal Object type of o is String , so 'c'(String), 'd'(String) are same.

Consider the following Figure




```
Base ob1 = new Derived2();
```
✓

```
Object ob2 = (Base)ob1;
```
✓

```
Derived2 ob3 = (Derived2)ob2;
```
✓

```
Base ob4 = (Derived2)ob3;
```
✓

```
Derived2 ob5 = (Derived1)ob4;
```
✗ //R.E ClassCastException

Conseder the following Code

```
class Dog
{
}
class Beagle extends Dog
{
}
class Test
{
        public static void main(String arg[])
        {
                Beagle b1 = new Beagle();
                Dog d1 = new Dog();
                Dog d2 = b1;
                //insertcode
        }
}
```

Which of the following are allowed to the line 9(//insertcode)?

1) Beagle b2 = (Beagle)d1;
2) Beagle b3 = (Beagle)d2;
3) Beagle b4 = d2;
4) Non of the above

Ans: in 1 Runtime checking failed(3rd case) and in 3 Comiple time error Incompatable types(2nd Case)

Conseder the following Code

```
class X
{
        void do1()
        {
        }
}
class Y extends X
{
        void do2()
        {
        }
}
class Test
{
        public static void main(String arg[])
        {
                X  x1 = new X();
                X x2 = new Y();
                Y y1 = new Y();
                //insertcode here to compile fine
        }
}
```

Which of the following are allowed to compile fine

A) x2.do2(); ✗
B) ((Y)x2).do2(); ✓
C) ((Y)x2).do1(); ✓
D) ((Y)x1).do1(); //C.E

8

INNER CLASSES

Introduction

We can declare a class inside another class such type of classes are called innerclasses.
This innerclasses concept has introduced in 1.1 version as the part of event handling. By observing the utilities and functionalities of inner class slowly the programmers are using this concept even in regular coding also.

Ex1:

With out existing car object there is no chance of existing wheel object. Hence we have to declare wheel class inside car class.

```
class car
{
        class wheel
        {

        }
}
```

Ex:2

A map is a collection of entry objects. With out existing Map object there is no chance of existing entry object. Hence we have to define entry interface inside map interface.

```
interface Map
{
        interface Entity
        {

        }
}
```

Based on the purpose and position of inner class all the inner classes are divided into 4 categories.

1) Normal/Regular inner classes
2) Method local inner classes
3) Anonymous inner classes
4) static nested classes

Normal/Regular inner classes

A class declared inside another class directly with out static modifier is called normal or regular inner class.

Ex:

9

THREADS AND CONCURRENCY

Introduction

Multitasking Executing several tasks simultaneously is called '*Multitasking*', There are 2 types of multitasking.

1) Process based Multitasking.
2) Thread based Multitasking.

Process based Multi Tasking

Executing several tasks simultaneously where each task is a separate independent process is called '*Process based Multitasking*'.

Ex:

While writing java program in the editor we can run *MP3 player*. At the same time we can *download a file* from the net. All these tasks are executing simultaneously and independent of each other. Hence it is process based Multitasking. Process based Multitasking is best suitable at O.S level.

Process based Multi Tasking

Executing several tasks simultaneously where each task is a separate independent part of the same program is called Thread based Multitasking. This type of multitasking is best suitable at programmatic level.

Java provides in built support for thread based multitasking by providing rich library (Thread, ThreadGroup, Runnable ..etc)

Whether it is Processbased or Threadbased the main objective of multitasking is to reduce response time and improve performance of the system.

The main application area of multithreading is videogames implementation, ,Multimedia graphics …etc.

Defining Instantiating, Starting the Thread

We can define instantiate and starting a thread by using the following 2- ways.

1) By extending Thread Class.
2) By implementing Runnable interface.

By extending Thread Class

Ex:

```
class MyThread extends Thread
{
        public void run()
        {
                for (int i=0; i<10; i++ )
                {
                        System.out.println("Child Thread");
                }
        }
}
```

```
class MultiThreadDemo
{
	public static void main(String[] args) throws InterruptedException
	{
		MyThread t = new MyThread();
		t.start();
		for(int i = 0; i<10; i++)
		{
			System.out.println("Main Thread");
		}
	}
}
```

Case1 :

Thread Shedular:

If multiple threads are there then which thread will get chance first for execution will be decided by "Thread Scheduler". Thread Scheduler is the post of JVM. The behavior of thread scheduler is vendor dependent and hence we can't expect exact O/P for the above program.

The possible Outputs are:

①	②	③
Child	Main	Main
.	.	Child
.	.	Main
.	.	Child
.	.	.
main	Child	main

……...

Case2:

The difference between t.start() & t.run():

In the case of t.start() a new thread will be created and which is responsible for the execution of run. 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.

Hence in the above program if we are replacing t.start() with t.run() then the O/P is

```
Child
Child
Child
Child
```

```
Main
Main
  .
  .
  .
```

Case3:

Importance of Thread Class start() method:

After Creating thread object compulsory we should perform registration with the thread

scheduler. This will take care by start() of Thread class, So that the programmers has to concentrate on only job. With out executing Thread class start() method there is no chance of start a new Thread in java.

```
start()
{
        Register our thread with the thread scheduler.
        Invoke run method.
}
```

Case4:

If we are not overriding run() method:

Then the thread class run method will be executed which has empty implementation.

Ex:

```
class MyThread extends Thread
{

}
class ThreadDemo
{
        public static void main(String arg[])
        {
                MyThread t = new MyThread();
                t.start();
        }
}
```

O/P:- no output

Case5:

If we are overriding start() method:

Ex:

```
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 arg[])
        {
                MyThread t = new MyThread();
                t.start();
        }
}
```

O/P:- start() method

In this case start() method will be executed just like a normal method.

Case6:

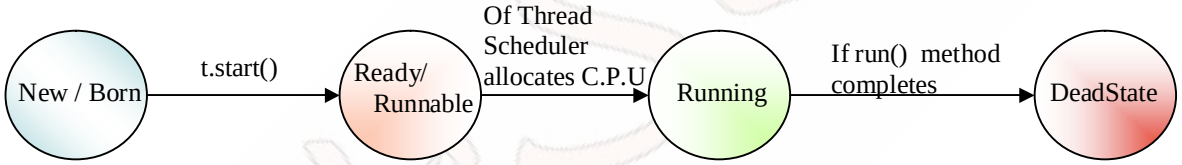
```
class MyThread extends Thread
{
        public void start()
        {
```

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

O/P:-
```
start() method.
run method.
main method.
```

Life cycle of Thread

When u write new MyThread()



Note:- we can't stop a running Thread explicitly. But until 1.2 version we can achieve this by using stop() method but it is deprecated method. Similarly suspend() and resume() methods also deprecated.

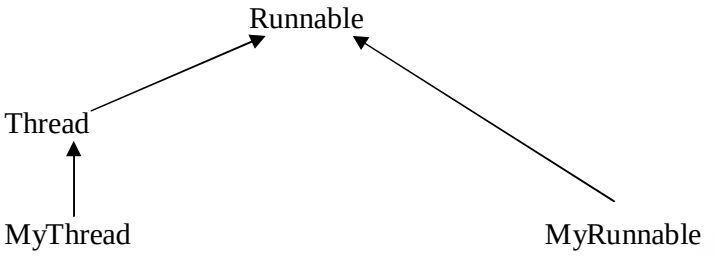
After starting a thread we are not allowed to restart the same thread once again, violation leads to Runtime Error saying "*IllegalThreadStateException*".

```
MyThread t = new MyThread();
t.start();
t.start();        ➔ Illegal Thread state Exception.
```

By Implementing Runnable Interface

We can define a Thread by implementing runnable interface also. Runnable interface available in java.lang package and contains only one method

```
public void run();
```



1st approach **2nd approach**

Ex:

```
class MyRunnable implements Runnable
{
        public void run()
        {
                for(int i = 0;i<=10;i++)
                {
                        System.out.println("Child Thread");
                }
        }
}
class ThreadDemo
{
        public static void main(String arg[])
        {
                MyRunnable r = new MyRunnable();
                Thread t  = new Thread(r);
                t.start();
                for(int i = 0;i<=10;i++)
                {
                        System.out.println("Main Thread");
                }
        }
}
```

Possible O/P:

```
Main Thread
Child Thread
Main Thread
Child Thread
Main Thread
Child Thread
Main Thread
Child Thread
Main Thread
Child Thread
Main Thread
Child Thread
Main Thread
Child Thread
Main Thread
Child Thread
Main Thread
Child Thread
Main Thread
Child Thread
Main Thread
Child Thread
Main Thread
Child Thread
```

Ex:

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

Case1:

```
t1.start();
```

A new thread will be started and executes thread class run method(Which is having empty implementation).

Case2:

t1.run();

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

Case3:

t2.start();

A new thread will be created and responsible for execution of MyRunnable run method.

Case4:

t2.run();

No new thread will be created and MyRunnable run method will be executed just like a normal method call.

Case5:

r.run();

No new thread will be created and MyRunnable run method will be executed just like a normal method call.

Case6:

r.start();

Compiler Error: MyRunnable doesn't contain start method.

In which of the following cases a new thread will be started.

1) t1.start(); ✓
2) t1.start(); ✓
3) r.start();
4) t1.run();
5) t2.run();
6) r.run();

Note:- Among 2- approaches of defining a thread – implements Runnable is always recommended to use in the 1st approach as we are already extending thread class. There is no chance of extending any other class. Hence we are missing the key benefits of oops inheritance(reusability) . hence this approach is not recommended to use.

Hybrid Mechanism To define a thread(not recommended to use)

Ex:

```
class MyThread  extends Thread
{
        public void run()
        {
                System.out.println("Child Thread");
        }
}
class HybridThreadDemo
{
        public static void main(String arg[])
        {
                MyThread t1 = new MyThread();
                Thread t  = new Thread(t1);
                t.start();
                System.out.println("Main Thread");

        }
}
```

O/P:- Child Thread
Main Thread

Or
Main Thread
Child Thread

Thread class construtors

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);

setting and getting the name of a Thread

Thread class defines the following methods to set and get the name of a Thread.

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

Ex:

```
class Test
{
	public static void main(String arg[])
	{
		System.out.println(Thread.currentThread().getName());
		Thread.currentThread().setName("New Thread");
		System.out.println(Thread.currentThread().getName());
	}
}
```

O/P:-

```
main
New Thread
```

Thread Priorities

Every Thread in java having some priority. The range of valid thread priorities is (1-10) (1 is least & 10 is Highest). Thread class defines the following constant for representing some standard priorities.

Thread.MIN_PRIORITY →1
Thread.NORM_PRIORITY →2
Thread.MAX_PRIORITY →3

Thread scheduler use these priorities while allocating C.P.U. The Thread which is having highest priority will get chance first for execution. If two threads having the same priority then which thread will get chance first for execution is decided by Thread Scheduler, which is vendor dependent i.e we can't expect exactly.

The default priority for the main thread only the 5, but for all the remaining threads the priority will be inherit from parent i.e what ever the parent has the same priority the child thread also will get.

Thread class contains the following methods to set and get priorities of thread.

1) public final void setPriority(int priority)

where priority should be from 1-10 other wise R.E: IllegalArgumentException.

2) public final int getPriority();

Ex:

```
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 arg[])
	{
		MyThread t = new MyThread();
		System.out.println(t.getPriority());
		t.setPriority(10);
		t.start();
		for(int i =0;i<10;i++)
		{
			System.out.println("Main Thread");
		}
	}
}
```

O/P:-

```
5
Main Thread
Child Thread
Main Thread
Child Thread
Main Thread
Child Thread
Main Thread
Child Thread
Main Thread
Child Thread
Main Thread
Child Thread
Main Thread
Child Thread
Main Thread
Child Thread
Main Thread
Child Thread
Main Thread
Child Thread
```

Some Operating Systems may not provide support for thread priorities.

Preventing Thread from execution

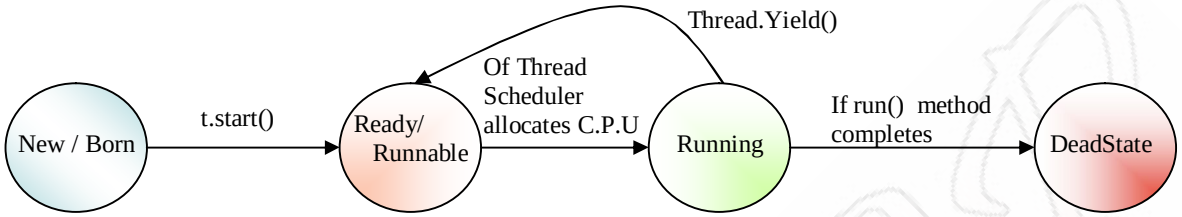
We can prevent a thread from execution by using the following methods.

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

yield()

The thread which is called yield() method temporarily *pause* the execution to give the chance for remaining threads of same priority. If there is no waiting thread or all waiting threads having low priority. Then the same thread will get the chance immediately for the execution.

public static native void yield();



Ex:

```
class MyThread extends Thread
{
        public void run()
        {
                for (int i=0; i< 10 ; i++ )
                {
                        System.out.println("Child Thread");
                        Thread.yield();
                }
        }
}
class YieldDemo
{
        public static void main(String arg[])
        {
                MyThread t = new MyThread();
                t.start();
                for(int i =0;i<10;i++)
                {
                        System.out.println("Main Thread");
                }
        }
}
```

In this case main thread will get chance more no of times for execution. Because child thread intentionally calling "*yield()*" method. As the yield method is native method some Operating system may not provide the support for this.

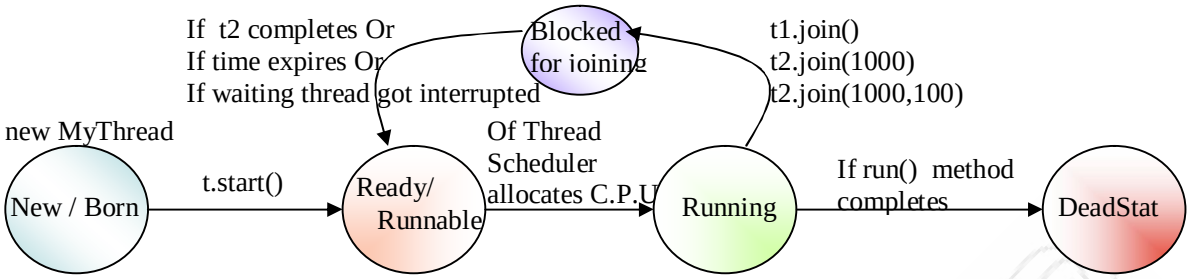
join()

If a thread wants to wait until some other thread completion then we should go for join method.

Ex:

If a thread t1 executes t2.join(), then t1 will be entered into waiting state until t2 completion.

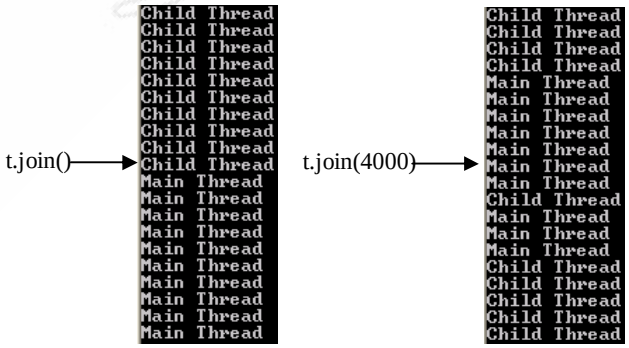
public final void join() throws InterruptedException
public final void join(long ms) throws InterruptedException
public final void join(long ms, int ns) throws InterruptedException



Ex:

```
class MyThread extends Thread
{
	public void run()
	{
		for (int i=0; i< 10 ; i++ )
		{
			System.out.println("Child Thread");
			try
			{
				Thread.sleep(1000);
			}
			catch (InterruptedException e)
			{
			}
		}
	}
}

class JoinDemo
{
	public static void main(String arg[])throws InterruptedException
	{
		MyThread t = new MyThread();
		t.start();
		t.join();		→ 1
		//t.join(4000)
		for(int i =0;i<10;i++)
		{
			System.out.println("Main Thread");
		}
	}
}
```

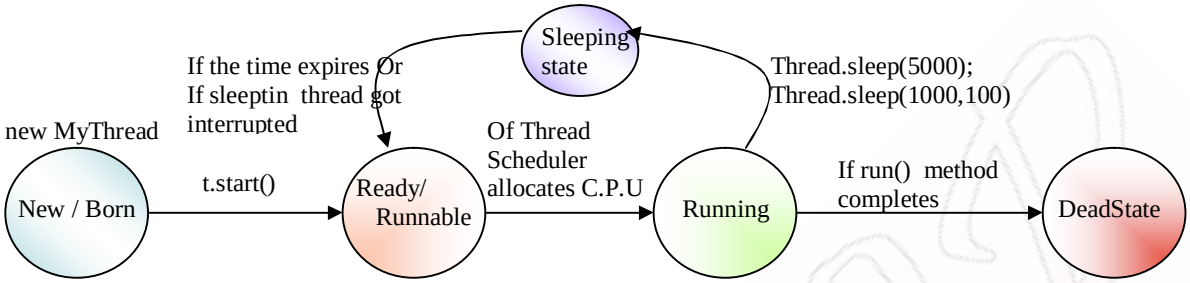
O/P:-



sleep()

If a method has to wait some predefined amount of time with out execution then we should go for sleep() method.

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



Ex:

```
class MyThread extends Thread
{
	public void run()
	{
		try
		{
			for (int i = 0;i<10;i++)
			{
				System.out.println("This is Lazy Method");
				Thread.sleep(1000);
			}
		}
		catch (InterruptedException e)
		{
			System.out.println(e);
		}

	}
}
class SleepDemo
{
	public static void main(String arg[])throws InterruptedException
	{
		MyThread t = new MyThread();
		t.start();
		System.out.println("Main Thread");
	}
}
```

O/P:-

```
Main Thread
This is Lazy Method
This is Lazy Method
This is Lazy Method
This is Lazy Method
This is Lazy Method
This is Lazy Method
This is Lazy Method
This is Lazy Method
This is Lazy Method
This is Lazy Method
```

interrupting a thread

we can interrupt a sleeping or waiting thread by using interrupt method of thread class.

public void interrupt()

when ever we are calling *interrupt()* the target thread may not be effected immediately. At the time of calling interrupt if the target thread is not in sleeping or in waiting state interrupt call will wait until target thread entered into sleeping or waiting state.

Ex:

```
class MyThread extends Thread
{
	public void run()
	{
		try
		{
			for (int i = 0;i<10;i++)
			{
				System.out.println("This is Lazy Method");
			//	Thread.sleep(3000);   → 1
			}
				Thread.sleep(3000);   → 2
		}
		catch (InterruptedException e)
		{
			System.out.println(e);
		}

	}
}
class InterruptDemo
{
	public static void main(String arg[])throws InterruptedException
	{
		MyThread t = new MyThread();
		t.start();
		t.interrupt();
		System.out.println("Main Thread");
	}
}
```

O/P:-

Sleep() used Inside for loop →

```
This is Lazy Method
Main Thread
java.lang.InterruptedException: sleep interrupted
```

Sleep() used Outside for loop →

```
Main Thread
This is Lazy Method
This is Lazy Method
This is Lazy Method
This is Lazy Method
This is Lazy Method
This is Lazy Method
This is Lazy Method
This is Lazy Method
This is Lazy Method
This is Lazy Method
java.lang.InterruptedException: sleep interrupted
```

Synchronization table for *yield(), join() & sleep()*

Property	yield()	join()	sleep()
Is it overloaded?	No	Yes	Yes
Is it static?	Yes	No	Yes
Is it final?	No	Yes	No
Is it throws InterruptedException?	No	Yes	Yes
Is it native?	Yes	No	Sleep(long ms)→native Sleep(long ms, int ms)→not native

Synchronization

'synchronized' is the keyword applicable for the *methods* and *blocks*. We can't apply this keyword for *variables* and *classes*. If a method declared as synchronized at a time only one thread is allowed to execute that method on the given object. The main advantage of synchronized keyword is we can overcome data inconsistency problem. The main limitation of synchronized keyword is it may create preference problems. Hence if there is no specific requirement it's not recommended to use synchronized keyword.
Every object in java has a unique lock. When ever are using synchronized keyword then ***object level lock*** concept will come into picture. If a thread want to execute any synchronized method on the object first it should require the lock of that object. Once a thread got the lock then it is allowed to execute any synchronized method on that object.
While a thread executing a synchronized method on the object, then the remaining threads are not allowed to execute any synchronized method on the same object. But the remaining threads are allowed to execute any non-synchronized method on the same object.

Every object in java has unique lock but a thread can acquire more than one lock at a time.

Ex:

```
class Display
{
    public synchronized void wish(String name)
    {
        for(int i =0;i<10;i++)
        {
            System.out.print("Hai.....!");
            try
            {
                Thread.sleep(2000);
            }
            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 arg[])
        {
                Display d = new Display();
                MyThread t1 = new MyThread(d,"YS");
                MyThread t2 = new MyThread(d,"Babu");
                t1.start();
                t2.start();
        }
}
```

O/P:-

If we are not declaring wish method as "*synchronized*" we will get irregular o/p because both threads will execute simultaneously. If we are declaring wish method as synchronized we will get regular o/p because at a time only one thread is allowed to execute wish method.

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

O/P:-

Even though wish method is synchronized we will get irregular o/p only because both threads are operating on different objects.

Class level lock: If a thread want to execute any static synchronized method then compulsory that thread should require class level lock. While a thread executing any static system method then the remaining threads are not allowed to execute any static synchronized method of the same class simultaneously. But the remaining threads are allowed to execute any non-synchronized static methods, synchronized – instance method, non – synchronized instance method simultaneously.

Declare static synchronized in display method and try the above example we will get regular p/p because there is class level lock.

Synchronized Blocks

It is not recommended to declare entire method as synchronized if very few lines of code causes the problem that code we can declare inside Synchronized Block so that we can improve performance of the system.

Syntax:

```
synchronized(b)
{
        //critical section.
}
```

Where 'b' is an object reference.

To get the lock for the current object we can define synchronized block as follows

```
synchronized(this)
{
        //critical section.
}
```

We can define synchronized block to get class level as follows

```
synchronized(Display.class)
```

we can define synchronized block either for *object references* or for *class references* But not for primitives violation leads to Compile time error.

```
int i =10;
synchronized(i)
{
        //------
}
```

C.E:- unexpected type
found : int.
required : reference.

Synchronized statement(Only for Interview Purpose):

The statement which are defined in inside synchronized method or synchronized blocks are called 'synchronized statement'.

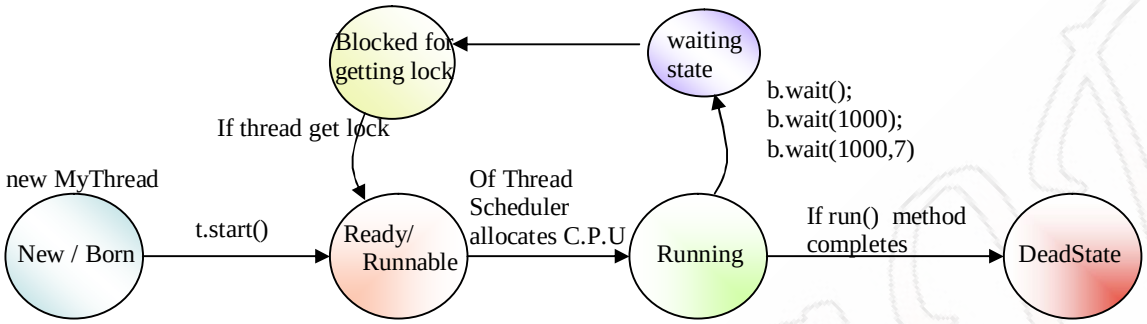
Inter Thread Communication

Two threads can communicate with each other by using wait(), notify(), notifyAll().
These methods are available in object class but not in thread class. Because threads are calling these methods on any object.

We should call these methods only from synchronized area other wise we get runtime exception saying IllegalMonitorStateException.

If a thread executes wait() method it immediately releases the lock of that object(But not all locks) and entered into waiting state.
After giving the notification also the thread releases the lock but may not be immediately.

public final void wait() throws InterruptedException
public final void wait(long ms) throws InterruptedException
public final void wait(long ms, int ns) throws InterruptedException
public final void notify();
public final void notifyAll();



Method	Is lock released
yield()	X
join()	X
sleep()	X
wait()	✓
notify()	✓
notifyall()	✓

Ex:

```
class ThreadA
{
        public static void main(String arg[])throws InterruptedException
        {
                ThreadB b = new ThreadB();
                b.start();
                System.out.println("Main Method");
                Thread.sleep(100);      ----------→ 1
                System.out.println(b.total);

        }
}
class ThreadB extends Thread
{
        int total = 0;
        public void run()
        {
                System.out.println("Child Starting calculation");
                for(int i = 1; i<=100; i++)
                {
                        if(i == 1)System.out.println("in for loop ");
                        total = total + i;
                }
        }
}
```

O/P:-

Using sleep() →
```
Main Method
Child Starting calculation
in for loop
5050
```
With out using →
```
Main Method
Child Starting calculation
0
in for loop
```

Here we used sleep() method to give chance to child thread. But we can't say child thread will finish his work with in given time. So we may get unusually outputs. There is no Guaranteed Output.

Ex:

```
class ThreadA
{
    public static void main(String arg[])throws InterruptedException
    {
        ThreadB b = new ThreadB();
        b.start();
        synchronized(b)
        {
            System.out.println("Main Method calling wait method ");
            b.wait();
            System.out.println("Main Got Notification");
            System.out.println(b.total);
        }
    }
}
class ThreadB extends Thread
{
    int total = 0;
    public void run()
    {
        synchronized(this)
        {
            System.out.println("Child Starting calculation");
            for(int i = 1; i<=100; i++)
            {
                total = total + i;
            }
            System.out.println("Child giving notification");
            this.notify();
        }
    }
}
```

O/P:-

```
Main Method calling wait method
Child Starting calculation
Child giving notification
Main Got Notification
5050
```

Dead Lock

If two threads are waiting for each other forever, then the threads are said to be in "*deadlock*".
There is no deadlock resolution technique but prevention technique are available

Ex:

Banker's Algorithm.

Ex:

```
class A
{
    synchronized void foo(B b)
    {
        System.out.println("Threas 1 entered foo() method");
```

```
                try
                {
                        Thread.sleep(600);
                }
                catch (InterruptedException e)
                {
                }
                System.out.println("Thread 1 is trying to call b.last()");
                b.last();
        }
        synchronized void last()
        {
                System.out.println("Inside A, This is last() method");
        }
}

class B
{
        synchronized void bar(A a)
        {
                System.out.println("Threas 2 entered bar() method");
                try
                {
                        Thread.sleep(600);
                }
                catch (InterruptedException e)
                {
                }
                System.out.println("Thread 2 is trying to call b.last()");
                a.last();
        }
        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();
                b.bar(a);
        }
        public void run()
        {
                a.foo(b);
        }
        public static void main(String arg[])
        {
                new DeadLock();
        }
}
```

O/P:-

```
Threas 2 entered bar() method
Threas 1 entered foo() method
Thread 2 is trying to call b.last()
Thread 1 is trying to call b.last()
```

DaemonThread

The threads which hare running in the background to provide support for user defined threads are called "*Daemon Thread*". Usually daemon thread are running with *low priority* but based on our requirement we can increase their priority also.

We can check whether the given thread is daemon or not by using the following thread class thread.

public boolean isDaemon()

we can change the daemon nature of a thread by using setDaemon() method of thread class.

public void setDaemon(Boolean b)

the daemon nature of a thread is *inheriting* from the parent. i.e if the parent is daemon then the child is also daemon and if the parent is non – daemon then the child is also non – daemon.

After starting a thread we are not allowed to change the daemon nature violation leads to runtime exception saying IllegalThreadStateException.

Ex:

```
class MyThread extends Thread
{
}
class Test
{
        public static void main(String arg[])
        {
                System.out.println(Thread.currentThread().isDaemon());
                MyThread t = new MyThread();
                System.out.println(t.isDaemon());
                t.setDaemon(true);
                System.out.println(t.isDaemon());
                t.start();
                //t.setDaemon(false); → 1
        }
}
```

O/P:-

```
false
false
true
```

If we don't comment line 1 we will get the IlleagalThreadStateException, see the following o/p.

```
false
false
true
Exception in thread "main" java.lang.IllegalThreadStateException
        at java.lang.Thread.setDaemon(Unknown Source)
        at Test.main(ThreadDemo.java:416)
```

We can't change the daemon nature of main thread because it has started already before main() method only.
All the daemon threads will be terminated automatically when ever last non – daemon thread terminates.

Ex:

```
class MyThread extends Thread
{
        public void run()
        {
                for(int i = 0;i<10;i++)
```

```
                {
                        System.out.println("Child Thread");
                        try
                        {
                                Thread.sleep(1000);
                        }
                        catch (InterruptedException e)
                        {
                        }
                }
        }
}
class DaemonThreadDemo
{
        public static void main(String arg[])
        {
                MyThread t = new MyThread();
                t.setDaemon(true);              → 1
                t.start();
                System.out.println("The end of main");
        }
}
```

O/P:-

```
Child Thread
The end of main
```

If we are commenting line 1, then both child and main threads are non – Daemon, hence they will execute until their completion.

If we are not commenting line 1, then the child thread is daemon and hence it will terminate automatically when ever main() thread terminates.

10

FUNDAMENTAL CLASSES IN java.lang.package

Introduction

The most commonly used and general purpose classes which are required for any java program are grouped into a package which is nothing but a "**java.lang.package".**
All the classes and interfaces which are available in this package are by default available to any java program. There is no need to import this class.

Object Class

The most common general methods which can be applicable on any java object are defined in object class. Object class is the parent class of any java class, whether it is predefined or programmer defined, hence all the object class methods are by default available to any java class.

Object class define the following 11 methods

- **clone**()
  Creates a new object of the same class as this object.
- **equals**(Object)
  Compares two Objects for equality.
- **finalize**()
  Called by the garbage collector on an object when garbage collection determines that there are no more references to the object.
- **getClass**()
  Returns the runtime class of an object.
- **hashCode**()
  Returns a hash code value for the object.
- **notify**()
  Wakes up a single thread that is waiting on this object's monitor.
- **notifyAll**()
  Wakes up all threads that are waiting on this object's monitor.
- **toString**()
  Returns a string representation of the object.
- **wait**()
  Waits to be notified by another thread of a change in this object.
- **wait**(long)
  Waits to be notified by another thread of a change in this object.
- **wait**(long, int)
  Waits to be notified by another thread of a change in this object.

toString()

To return String representation of an object.

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

Ex:

```
class Student
{
        String name;
        int rollno;
        Student(String name,int rollno)
        {
                this.name = name;
                this.rollno = rollno;
        }

        public static void main(String arg[])
        {
                Student s1 = new Student ("raju", 101);
                Student s2 = new Student ("giri", 102);
                System.out.println(s1);         →Student@10b62c9
                System.out.println(s2);         →Student@82ba41

        }
}
```

When ever we are passing object reference as argument to s.o.p() internally JVM will call toString() on that object.

If we are not providing toString() then Object class toString() will be executed which is implemented as follows

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

Based on our requirement to provide our own String representation we have to override toString()

Ex:

If we are printing Student Object reference to return name & roll no we have to override toString() as follows

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

If we can place this toString() in student class then the O/P is

raju-----101

giri-----102

It is highly recommended to override toString() in our classes.

hashCode()

The hashCode of an Object just represents a random number which can be used by JVM while saving/adding Objects into Hashsets, Hashtables or Hashmap.
hashCode() of Object class implemented to return hashCode based on address of an object, but based on our requirement we can override hashCode() to generate our own numbers as hashCodes

Case1:

```
class Test
{
        int i;
        Test(int i)
        {
                this.i = i;
        }
        public int hashCode()
        {
                return i;
        }
        public static void main(String arg[])
        {
                Test t1 = new Test(100);
                Test t2 = new Test(110);
                System.out.println(t1);         → 64
                System.out.println(t2);         → 6e
        }
}
```

Case2:

```
class hashCodeDemo
{
        int i;
        hashCodeDemo(int i)
        {
                this.i = i;
        }
        public int hashCode()
        {
                return i;
        }
        public String toString()
        {
                return i + "";
        }
        public static void main(String arg[])
        {
                hashCodeDemo h1 = new hashCodeDemo(100);
                hashCodeDemo h2 = new hashCodeDemo(110);
                System.out.println(h1);         → 100
                System.out.println(h2);         → 110
        }

}
```

equals()

Ex:

```
class Student
{
        String name;
        int rollno;
        Student(String name,int rollno)
        {
                this.name = name;
                this.rollno = rollno;
        }

        public static void main(String arg[])
        {
                Student s1 = new Student ("raju", 101);
                Student s2 = new Student ("giri", 102);
                Student s3 = new Student ("giri", 102);
                System.out.println(s1.equals(s2));      → false
                System.out.println(s2.equals(s3));      →false

        }
}
```

r1 == r2 → reference Comparision.
r1.equals(r2) → reference.

Note: In this case Object class .equals() has executed which is meant for reference comparison but based on our requirement it is recommended to override .equals() for content comparison.
By over loading .equals() we have to consider the following 3 cases
Case1: The meaning of equality

Case2: In the case of heterogeneous objects we have to return false. (i.e) we have to handle ClassCastException to return false.

Case3: If we are passing null as the argument we have return false. (i.e) we have to handle NullPointerException to return false.

Overridden methods of equals

Ex:

```
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 c)
                {
                    return false;
                }
                catch (NullPointerException e)
                {
                    return false;
                }

            }
            public static void main(String arg[])
            {
                Student s1 = new Student ("raju", 101);
                Student s2 = new Student ("giri", 102);
                Student s3 = new Student ("giri", 102);

                System.out.println(s1.equals(s2));     → false
                System.out.println(s2.equals(s3));     → true
                System.out.println(s1.equals(null));   → false

            }
        }
```

Comparison between '==' operator and '.equals()'

==	.equals()
1) It is an operator and applicable for both primitive and Object references.	1) It is a method applicable for Only Object reference.
2) For Object reference r1, r2 r1 == r2 is true iff both r1 and r2 pointing to the same object on the heap i.e it is meant for reference comparision.	2) Default implementation available in Object class is meant for reference comparison only i.e r1.equals(r2) is true iff both r1,r2 are pointing to the same object on the heap.
3) We can't override == operator.	3) We can override equals() for content comparison.
4) IF r1 & r2 are incompatible then r1 == r2 results Compile Time Error.	4) If r1 & r2 are incompatible then r1.equals(r2) is always false.
5) For any Object reference r, r == null returns false.	5) For any Object reference r, r.equals(null) returns false.

Relationship between '==' and .equals()

If r1 == r2 is true then r1.equals(r2) is always true.
If r1.equals(r2) is true, then r1 == r2 need not to be true.

Contract Between .equals() and hashCode()

1) If two objects are equal by .equals() then their hashcodes must be equal.
   Ex:
   If r1.equals(r2) is true then
   r1.hashCode() == r2.hashCode is also true.
2) If two Objects are not equal by .equals() then their hashCode may or may not be same.
3) If the hashCodes of two Objects are equal then the objects may or may not be equal by .equals()
4) If the hashCodes of two Objects are not equal then the Objects are always not equal by .equals().

To satisfy above contract when ever we are overriding .equals it is highly recommended to override hashCode() also.

clone()

The process of creating exactly duplicate Object is called Clonning.
Object class contains the clone method to perform this

protected native Object clone() throws CloneNotSupportedException

Ex:

```
class Test implements Cloneable
{
    int i = 10;
    int j = 20;
    public static void main(String arg[])throws CloneNotSupportedException
    {
        Test t1 = new Test();
        Test t2 = (Test)t1.clone();
        t1.i = 100;
        t1.j = 200;
        System.out.println(t2.i+"----"+t2.j);   → 10----20
    }
}
```

All the Objects can't have the capability to produce cloned Objects. Only clonaeble objects having that capability.

An Object is said to be cloneable iff the corresponding class has to implement java.lang.cloneable interface.

It doesn't contain any methods it is a marker interface.

Protected members can be accessible from outside package in the child classes but we should invoke them by using child class reference only. That is parent class reference is not allowed to invoke protected members from outside package.

```
class Test implements Cloneable
{
```

```
        public static void main(String arg[])throws CloneNotSupportedException
        {

        Case1: Object o = new Object();          ✗
               Object o2 = o.clone();

        Case2: Object o = new Test();            ✗
               Object o2 = o.clone();

        Case3: Test t = new Test();
               Object o = t.clone();             ✓
        }
}
```

It is highly recommended to override clone() in our class like doGet(), doPost() methods.

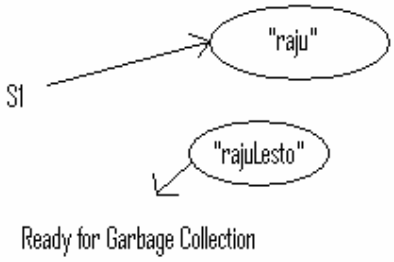
Strings Class

Once we created String objects we are not allowed to perform any changes in the existing object. If u r trying to perform any changes with those changes a new String object will be created this behavior is nothing but '*immutability*' Of the String Objects.

Ex:

```
String s1 = new String("raju");
s1.concat("Lesto");
System.out.println(s1);
```

O/P:- raju



Once we created a StringBuffer object we are allowed perform any changes in the existing object only. This behavior is nothing but '*mutability*' of the StringBuffer object.

Ex:

```
StringBuffer  sb1 = new StringBuffer("raju");
sb1.append("Lesto");
System.out.println(sb1);
```

O/P:- rajuLesto

Ex:

```
String s1 = new String("lesto");
String s2 = new String("lesto");
System.out.println(s1 == s2);      → false
System.out.println(s1.equals(s2)); → true
```

In String class .equals method is overridden for content comparision.

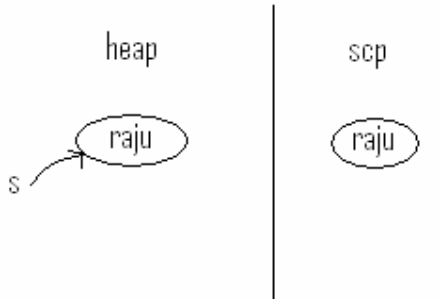
```
StringBuffer sb1 = new StringBuffer("lesto");
StringBuffer sb2 = new StringBuffer("lesto");
System.out.println(sb1 == sb2)          →false
System.out.println(sb1.equals(sb2));→ false
```

In the StringBuffer .equals method is not overridden for content comparison.
Hence Object class .equals method will be executed which is meant for reference comparison.

Difference Between *(String s = new String("raju")) and (String s = "raju")?*

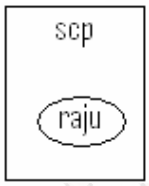
String s = new String("raju"):-

In this case 2 objects will be created one is on the heap and the second one is on the scp(String constant pool) and 's' is referring to heap object.



→ Object Creation in SCP is Optional.
→ If the object is not available then only a new object will be created.

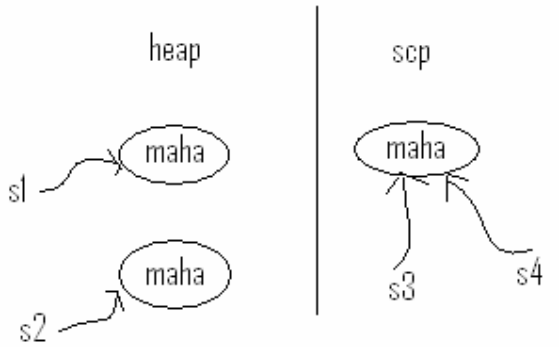
String s = "raju":-
In this case only one object will be created if it is not already available and 's' is referring to that object. If an object is already available with this can in scp then 's' will refer that existing object only instead of creating new object.



Note:- The Object present in the scp is not eligible for garbage collection even though the object doesn't have any reference variables.
All the scp objects will be destroyed at the time of JVM shutdown.

Ex:

```
String s1 = new String("maha");
String s2 = new String("maha");
String s3 = "maha";
String s4 = "maha";
```



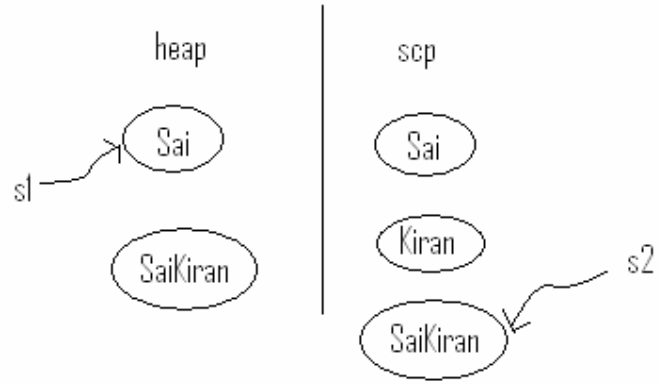
Note:- In the scp there is no chance of 2 string objects with the same content i.e duplicate string objects won't present in scp.

But in the case of heap there may be a chance of duplicate stringObjects

Ex:

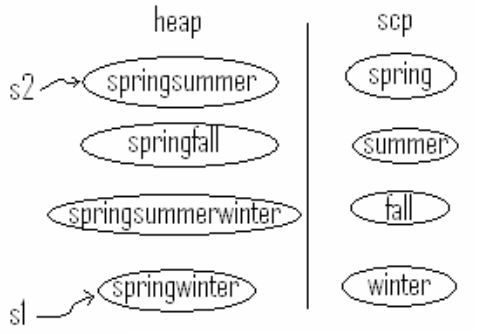
```
String s1 = new String("Sai");
1.concat("Kiran");
String s2 = "SaiKiran";
System.out.println(s1);
System.out.println(s2);
```

O/P:- Sai
SaiKiran




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

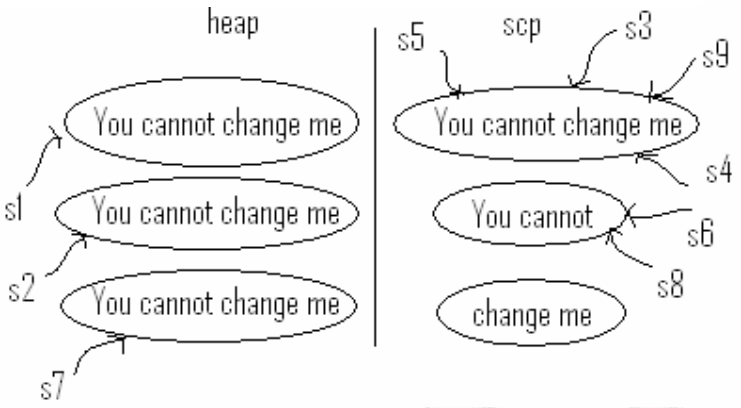
O/P:- springwinter
springsummer



Ex:

```
class  StringTest
{
        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);
        String s3 = "You cannot change me";
        System.out.println(s1==s3);
```

```
        String s4 = "You cannot change me";
        System.out.println(s3==s4);
        String s5 = "You cannot"+" change me";
        System.out.println(s4==s5);
        String s6 = "You cannot";
        String s7 =  s6 + " change me";
        System.out.println(s4==s7);
        final String s8 = "You cannot";
        String s9 =  s8 + " change me";
        System.out.println(s4==s9);
        }
}
```



Importance of String Constant Pool(SCP)

In our program if any string object is repeatedly going to use, we can create only one object in the string constant pool and shared by several required references.

Instead of creating several string objects with the same content creating only one object improves performance of the system and memory utilization also. This is biggest advantage of SCP.

As the Several references pointing to the same object in SCP by using one reference if u r allowed to change the content the remaining references may be effected. Hence once we created a String object. We r not allowed to change it's content. If u r trying to change with those changes a new object will be created and there is no impact on the remaining references. This behavior is nothing but immutability of the string objects.

SCP is the only reason why the String Objects are immutable.

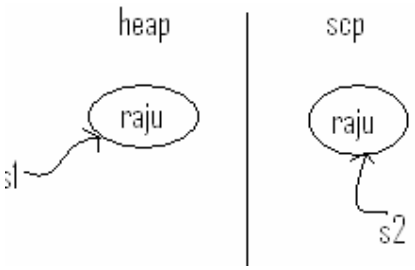
Interview point of questions:

1) What is the difference Between String and StringBuffer?
2) What is the meaning of immutability?
3) Why the String Objects are declared as immutable?

Interning of String Objects

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

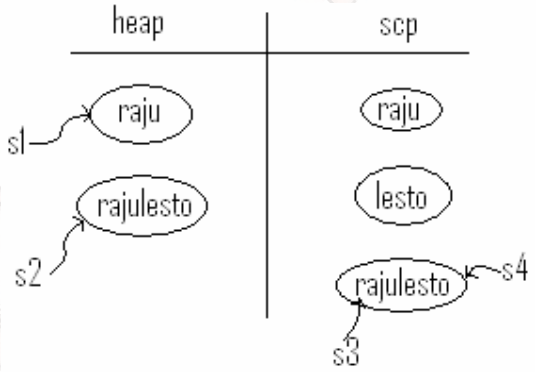
```
    }
}
```



At any point of time if we have heap object reference we can find it's equivalent SCP Object by using intern() method.
If the equivalent SCP Object is not already available then intern() method will create a new object in the SCP.

Ex:

```
class  StringTest
{
  public static void main(String[] args)
  {
  String s1 = new String("raju");
  String s2 = s1.concat("lesto");
  String s3 = s2.intern();
  String s4 = "rajulesto";
  System.out.println(s3 == s4);
  }
}
```



String Class Constructor

1) String s = new String();
Creates an empty String Object.

2) String s = new String(String literal);
3) String s = new String(StringBuffer s);
4) String s = new String(char [] ch);

```
char[] ch = {'a','b','c','d'};
String s = new String(ch);
System.out.println(s);
O/P:--abcd
```

5) String s = new String(byte[] b);
byte[] b = {100,101,102,103,104};
String s = new String(b);
System.out.println(s);
O/P:--defgh.

Important Methods of String Class

1) public char charAt(int index);
Ex:-
String s = “raju”;
System.out.println(s.chatAt(2)); O/P----j
System.out.println(s.charAt(10)); O/P----StringIndexOutOfBoundsException.

2) The overloaded ‘+’ and ‘+=’ operators acts as concatenation operation only.
Public String concat(String s)
Ex:- String s = ”raju”;
s = s.concat(“lesto”);
s = s+”lesto”;
s += “lesto”;
System.out.println(s);

3)
(i) Public Boolean equals(Object o)
It checks for content comparision. in this case of hetrogenious objects this method returns false.
String s = "lesto";
System.out.println(s.equals("LESTO"));
System.out.println(s.equals("lesto"));

(ii) Public Boolean equalsIgnoreCase(String s);
String s = "lesto";
System.out.println(s.equalsIgnoreCase("LESTO"));

Note:- Usually for validating username(where the case is not important method, we can use equals ignore case method) But for validating passwords we can use eauals method where the case is important.

4) public int length();
String s = "lesto";
System.out.println(s.length); O/P:-C.E
System.out.println(s.length()); O/P:-5
length is the variable applicable for array objects,But length() is the method applicable for String Object.

5) public String replace(char old, char new);
String s = "ababa";
System.out.println(s.replace('a','b')); O/P:-bbbbb

6) (i)public String substring(int begin);
String s = "abcdefgh";
System.out.println(s.substring(3));O/P:-defgh
(ii)public String substring(int begin, int end);
Returns the characters from (begin) index to (end-1) index.

String s = "abcdefgh";
System.out.println(s.substring(3,6)); O/P:-def.

7) public String toLowerCase();
public String toUpperCase();
Ex:- String s = “RoyalChallange”;
System.out.println(s.toLowerCase());

System.out.println(s.toUpperCase());

8) public String trim()
to remove blank spaces present at Starting and Ending of the Sting. But not middle blank spaces.

```
String s = "   raju 12  ";
System.out.println(s.trim()); O/P:-raju 12
```

9) public int indexOf(char ch);
returns first occurance index
public int lastIndexOf(char ch);

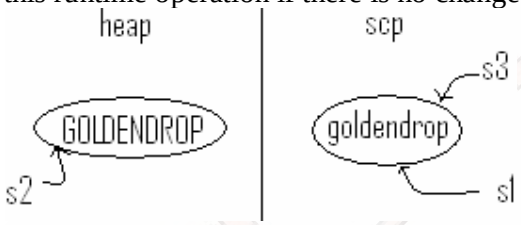
Q)Check the following question

```
String s1 = "golden drop";
String s2 = s1.toUpperCase();
String s3 = s1.toLowerCase();
System.out.println(s1 == s2);
System.out.println(s1 == s3);
```

What is the output?

O/P :- false, true

Because of some runtime operation, if a String object is required to create it should always create on the heap only.
Due to this runtime operation if there is no change in the content then new object won't be created.



StringBuffer Class

Constructors of StringBuffer:

StringBuffer sb = new StringBuffer();

Creates an empty StringBuffer object with default initial capacity 16.
If it reaches max capacity then a new StringBuffer object will be created with new capacity is

capacity = (currentcapacity + 1) * 2

Ex:-

```
StringBuffer sb = new StringBuffer();
System.out.println(sb.capacity());
sb.append("abcdefghijklmnop");
sb.append("q");
System.out.println(sb.capacity()); -→ 34
```

Ex:-

```
StringBuffer sb = new StringBuffer(40);
System.out.println(sb.capacity());
sb.append("abcdefghijklmnop");
        .
        .
        .
        .
```

Creates an empty StringBuffer with the required initial capacity.

StringBuffer sb = new StringBuffer(String s);
Creates an equivalent StringBuffer object for the given string with

capacity = s.length() + 16

Ex:-

```
StringBuffer sb = new StringBuffer("Lesto");
System.out.println(sb.capacity());  → 16 + 5 = 21
```

→Q) Which of the following is a valid constructor for the StringBuffer but not for the String.

1) ( )
2) (int capacity)
3) (String s)

Important Methods of StringBuffer class

1)public int length();

2)public int capacity();

3)public char charAt(int index);

Ex:-

```
StringBuffer sb = new StringBuffer("Lesto");
System.out.println(sb.charAt(3));
System.out.println(sb.chatAt(10));      → StringIndexOutOfBoundsException.
```

4)public void set chatAt(int index, char ch);

Replaces the character present at specified index with the provided char.

Ex:-

```
StringBuffer sb = new StringBuffer("");
sb.setCharAt(3,'a');
System.out.println(sb);
```

5)public StringBuffer append(String s)

Allows all overloaded methods….int, float, byte, boolean, char[], byte[],….etc

6)public StringBuffer insert(int index, String s)

Ex:-

```
StringBuffer sb = new StringBuffer("Lesto");
sb.insert(2,"raju");
System.out.println(sb); StringBuffer sb = new StringBuffer(String s);
```

7) public StringBuffer delete(int start, int end)

delete characters from **start** to **(end-1)**

```
StringBuffer sb = new StringBuffer("abcdefgh")
sb.delete(2,6);  O/P:-abgh
```

8) public StringBuffer delete charAt(int index)

deleting Character located at specified index.

9) public StringBuffer reverse();

Ex:-

```
StringBuffer sb = new StringBuffer("raju");
Syetem.out.println(sb.reverse()); O/P:-ujar.
```

10) public void setLength(int requiredlength);

Ex:-
```
StringBuffer sb = new StringBuffer(" aishwaryaabhishak");
sb.setLength(8);
System.out.println(sb) O/P:-aishwarya
```

Note:-All the methods which are available in the StringBuffer are Sunchronized or it may effect the performance of the system.

We can overcome this problem by using StringBuilder.

StringBuilder

StringBuilder class exactly similar to StringBuffer (including constructors and methods) Except the following.

StringBuilder	StringBuffer
No method is synchronized	All methods are Synchronized
StringBuilder is not thread safe	StringBuffer is thread safe
Performance is high	Performance is very low
Available only in 1.5 version	Available from 1.0 version

If the content is not changing frequently then we should go for the **string**.
If the content will change frequently and thread safety is required then we should go for **StringBuffer.**
If the content is changing frequently and thread safety is not required then we should go for **StringBuilder.**

Chaining of Methods

For most of the methods in String and StringBuffer the returntypes are the same String and StringBuffer objects only.

After Applying a method we are allowed to call another method on the result which forms a method change.

```
sb.m1().m2().m3().m4().m5()…..;
```

All these method calls will execute from left to right.

```
StringBuffer sb = new StringBuffer(“raju”);
sb.append("software").reverse().insert(2,"abc").delete(2,5).append("xyz");
```

Wrapper Classes

The main objectives of wrapper classes are:
1) To Wrap primitives into object form. So that we can handle primitives also just like objects.
2) To Define several utility functions for the primitives(like converting primitive to the string form etc.)

Constructors:
valueOf()
xxxValue()
parseXxx()
toString()

Constructing Wrapper objects by using constructors

Every Wrapper class contains 2 constructors one can take the corresponding primitive as the argument and the other can take the corresponding string as the argument.

Ex:

```
Integer I = new Integer(10);
Integer I = new Integer(“10”);
```

If the String is unable to convert into the number form then we will get runtime exception saying “NumberformatException”.

Ex: `Integer I = new Integer(“ten”);`

Float class contains 2 constructors which can take double String as argument.

```
Float f = new Float(10.5f);
Float f = new Float(“10.5f”);
Float f = new Float(10.5);
Float f = new Float(10.5);
```

Character class contain only one constructor which can take char as the argument i.e character class doesn’t contain a constructor which can take string as the argument.

Ex:

```
Character ch1 = new Character('a');   → valid.
Character ch1 = new Character("a");   → not valid.
```

Boolean class contains 2 constructors one can take boolean primitive. Other can take string argument. If u r providing boolean primitive as the argument the. The allowed values are true or false.
Case is not important if the content contains ‘TRUE’ then it is considered as true other wise it considered as false.

Q) ***Which of the following are valid ?***

```
Boolean b = new Boolean(true);      ✓   O/P:-true
Boolean b = new Boolean(FALSE);     X
Boolean b = new Boolean(“false”);   ✓   O/P:-false
Boolean b = new Boolean(“TrUE”);    ✓   O/P:-true
Boolean b = new Boolean(“raju”);    ✓   O/P:-false
Boolean b = new Boolean(“yes”);     ✓   O/P:-false
```

Q) ***Consider The Following*:**

Boolean b1 = new Boolean("yes");
Boolean b2 = new Boolean("No");

System.out.println(b1.equals(b2));
1) C.E
2) R.E
3) true
4) false
Ans: 3 (true)

WrapperClasses	constructor arguments
Byte	byte (or) string
Short	short (or) string
Integer	int (or) string
Long	long (or) string
Float	float (or) string (or) double
Double	double (or) string
Character	char
Boolean	boolean (or) string

valueOf( ) methods

version1:
All the wrapper classes **except character** class contains the valueOf() method for converting ***string*** to corresponding Wrapper Object.
public static wrapper valueOf(String s);
Ex:

Integer I = Integer.valueOf('10'); ✓
Float F = Float.valueOf("10.5"); ✓
Boolean B = Boolean.valueOf("raju"); ✓
Character ch = Character.valueOf("10"); X→ C.E

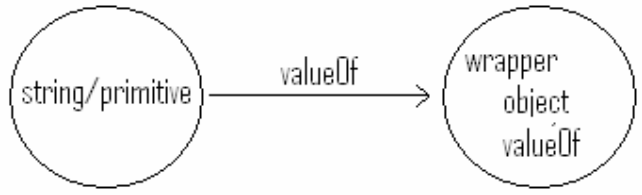
Version2:
All Integral wrapper classes "Byte, Short, Long, Integer" Contains the following valueOf() method.
public static wrapper valueOf(String s, int radix);
The allowed base of radix is 1 to 36.Because Numerics(10) ,Alphabets(26) finally 10+26 =36
Ex:

Integer I = Integer.valueOf("101011", 2); ✓
System.out.println(I); **O/P:- 43**

Version3:
Every Wrapper class including character class contains the following valueOf() method to convert ***primitive*** to wrapper object form.
public static wrapper valueOf(primitive p);
Ex:

Integer I = Integer.valueOf(10); ✓
Character ch = Character.valueOf('a'); ✓
Boolean B = Boolean.valueOf(true); ✓

Version1, Version2 → String to wrapper object.
Version3 → primitive to wrapper object.



xxxValue() method

Every wrapper class Except character and Boolean classes contains the following xxxValue() methds for converting wrapperObject to primitive.

```
public int intValue();
public byte byteValue();
public short shortValue();
public long longValue();
public float floatValue();
public int doubleValue();
```

Ex:

```
Integer I = Integer.valueOf(130);
System.out.println(I.byteValue());
System.out.println(I.shortValue());   →-126
System.out.println(I.intValue());     →130
System.out.println(I.langValue());    →130
System.out.println(I.floatValue());   →130.0
System.out.println(I.doubleValue()); →130.0
```

Character class contain **charValue()** method to return char primitive for the given character wrapper object.

```
public char charValue();
```

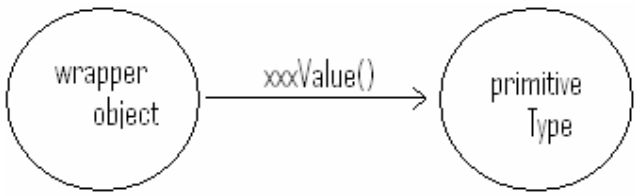
Ex:

```
character ch = new character('a');
char ch = ch.charValue();
System.out.println(ch1);        O/P:- a
```

Boolean class contains booleanValue() method to return boolean primitive for the given boolean objective.

```
Boolean B = Boolean.valueOf("Tea Break");
boolean b1 = B.booleanValue();
System.out.println(b1);  O/P:-false
```

Note:- In total 38 xxxValue methods are possible ((6 X 6) +1 + 1) = 38



parseXxx() methods

version1:

Every wrapper class except character class contains the following parseXxx() method for converting ***String*** to ***primitive*** type

public static primitive parseXxx(String s);

Ex:

int i= Integer.valueOf("10");

Boolean b = Boolean.parseBoolean("true");

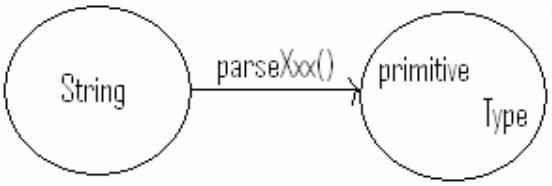
Version2:

Integral wrapper classes(Byte, Short, Integer and long) contains the following parseXxx() method.

public static primitive parseXxx(String s, int radix);

int i = Integer.parseInt("101011", 2);

System.out.println(i); **O/P:-43**



toSting()methods

version1:

All wrapper Classes contains an instance method toString() for converting wrapper object to String type. This is overriding method of object class toString() method.

public String toString();

Integer I = new Integer(10);

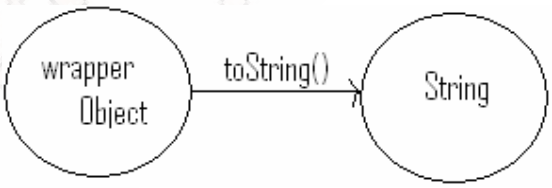
String s = I.toString();

System.out.println(s); **O/P:-10**

Boolean B = new Boolean("raju");

String s = B.toString();

System.out.println(s); **O/P:-false**



Version2:

Every wrapper class contains the following static toString() for converting primitive to String representation..

public static String toString(primitive p);

String s = Integer.toString(10);

System.out.println(s); **O/P:-10**

String s = Boolean.toString(true);
System.out.println(s); **O/P:-true**

Version3:

Integer and long classes contains the following toString() to return the String in the specified radix.

public static String toString(int/long, int radix);

String s = Integer.toString(43, 2);
System.out.println(s); **O/P:-"101011"**

String s = Integer.toString(43, 8);
System.out.println(s); **O/P:-"53"**

Version4:

Integer and Long classes contains the following methods to return specified radix String form.

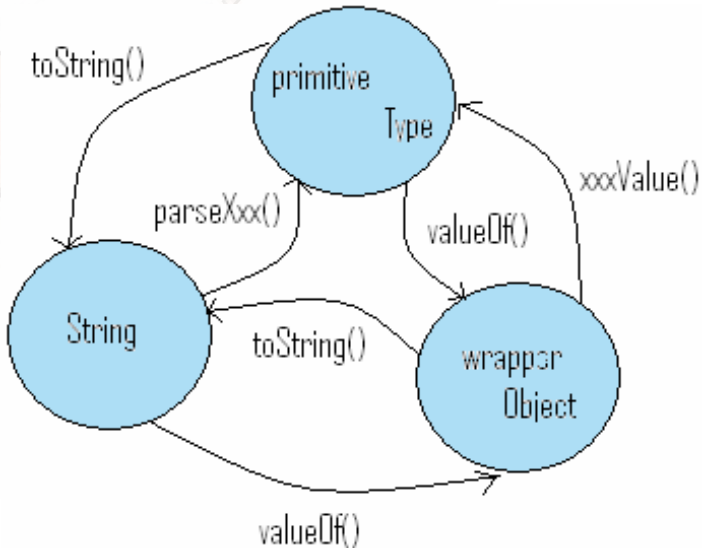
public static String toBinaryString(int/long, l);
public static String toOctalString(int/long, l);
public static String toHexString(int/long, l);

String s = Integer.toBinaryString(43);
System.out.println(s); **O/P:-"101011"**

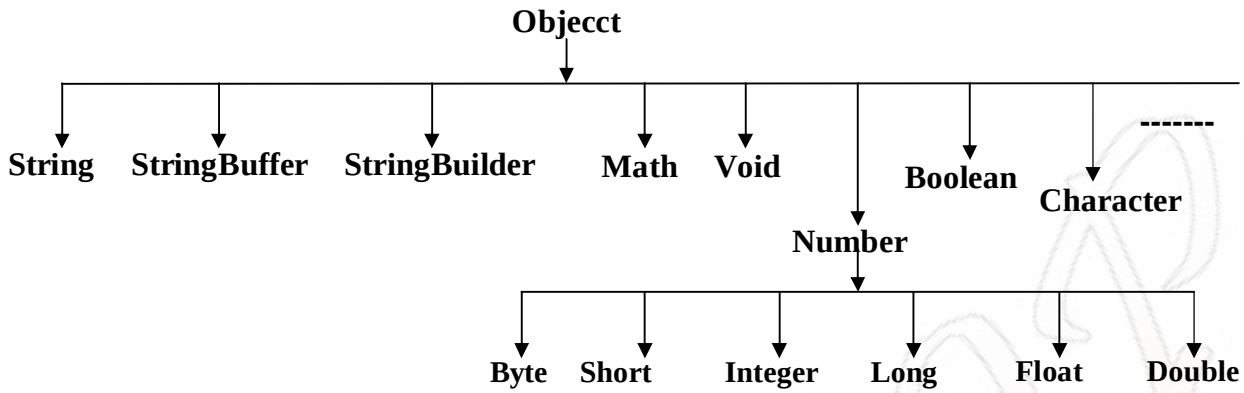
String s = Integer.toOctalString(43);
System.out.println(s); **O/P:-"53"**

String s = Integer.toHexString(43);
System.out.println(s); **O/P:-"262"**

Diagrammatic form of wrapper classes



Partial Hirarchy of java.lang Package



- Void is also considered as wrapper class.
- String, StingBuffer, StringBuilder and all wrapper class are final.
- In Addition to String Object all wrapper class objects also immutable.
- Boolean and Character wrapper classes are not child classes or Number class

11

COLLECTIONS FRAME WORK & GENERICS

Limitations of Object Arrays

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

The main limitations of Object Arrays are

1) Arrays are ***fixed in size*** i.e once we created an array there is no chance of increasing or decreasing it's size based on our requirement.

2) Arrays can hold ***only homogeneous*** data elements.

Ex:-

```
Student [] s = new Student [600];
s[0] = new Student;          ✓
s[1] = new Integer(10);      X
s[2] = "raju";               X
```

We can resolve this problem by using object Arrays.

```
Object [] o = new Object [600];
o[0] = new Student;          ✓
o[1] = new Integer(10);      ✓
o[2] = "raju";               ✓
```

i.e By using Object Arrays we can hold heterogeneous data elements.

3) For the Arrays there is ***no underlying Data Structure*** i.e for every requirement we have to code explicitly and there is no default ready made support(like sorting, searching).
we can't represent array elements in some sorting order by default. We ***can't prevent duplicate*** object insertion etc…

To resolve the above problems of arrays, Sun people has introduced **'collections concept'**

- Collections are grow able in nature i.e based on requirement we can increase or decrease the size.
- Collections can hold heterogeneous data elements also.
- Every collection class has implemented based on some data structure

Hence for every requirement ready made method support is possible.
We can represent all the elements in some sorting order. We can prevent duplicate object insertion by default.

Comparison Between collections and Arrays

Collections	Arrays
1) Collections are not fixed In size.	1) Arrays are fixed In size.
2) With respect to memory collections are good.	2) with respect to memory arrays are not good.
3) With respect to performance collections shows worst performance.	3) With respect to performance the arrays are recommended to use.
4) Collections can hold heterogeneous data elements.	4) Arrays can only hold homogeneous data elements.
5) Every Collection class has built by using some Data structure.	5) There is no underlying Data Structure.
6) Collection can hold only Objects but not primitives.	6) Arrays can hold both Objects and primitives.

Collections Frame Work

It defines group of classes and interfaces which can be used for representing a collection of Objects as single entity.

Java: Collection Frame Work collection

C++: STL(standard Template Library) container

7 key Interfaces of collection Frame Work

1) Collection :

This interface is the root interface for entire collection framework.
This interface can be used for representing a group of objects as single entity.
This interface defines the most common general methods which can be applicable for any collection object.
There is no concrete class which implements collection interface directly.

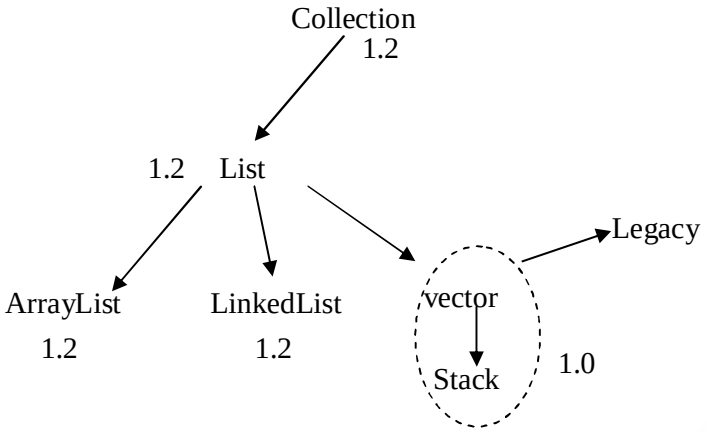
Difference Between Collection and Collections

Collection is an **interface** available in **java.util** package for representing a group of objects as single entity.
Collections is an **utility class** available in **java.util** package and defines several utility methods for collection implemented class object

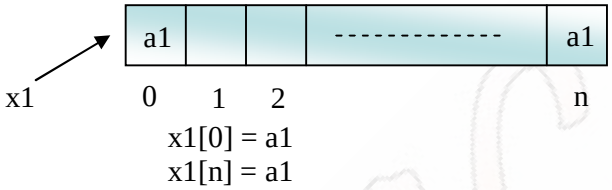
2) List interface :

This can be used for representing a group of individual objects as a single entity where **insertion order is preserved** and **duplicate** objects allowed. This is child interface of collection.
The following classes implemented List interface directly.

ArrayList, LinkedList, Vector and Stack.



Vector and stack classes are reengineered in 1.2 version to fit into collection frame work.
By means of Indexes we can preserve insertion order and we can differentiate duplicate objects.



3) Set interface :

This can be used for representing a group of individual objects where **duplicate objects are not allowed** and **insertion operation is not preserved.**

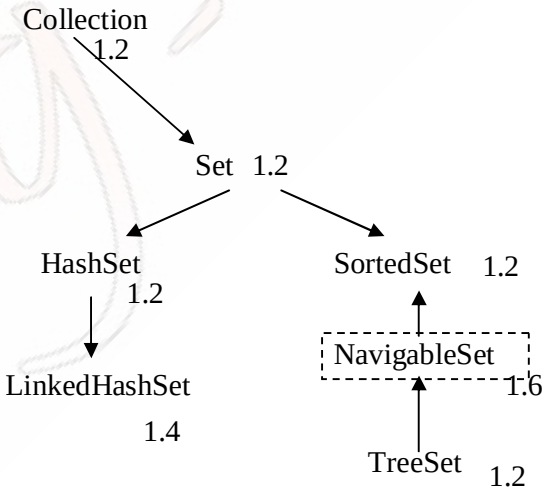
This is the child interface of collection

hashset and *linkedHashSet* are the classes which implements Set interface directly.

4) SortedSet interface :

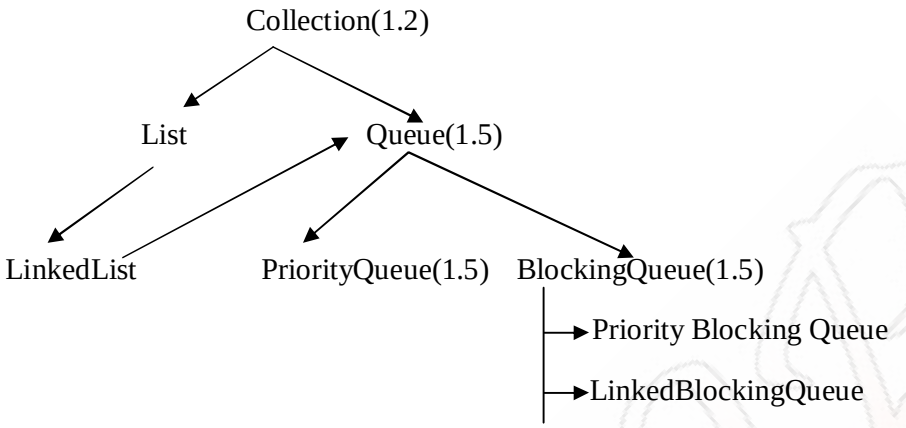
This can be used for representing a group of individual and unique objects. Where all the objects are inserted in some sorting order. It is the child interface of Set interface

TreeSet is the implemented class of SortedSet



5) Queue interface:

It is the child interface of collection it has introduced in 1.5 version this interface can be used for representing a group of individual objects prior to processing.

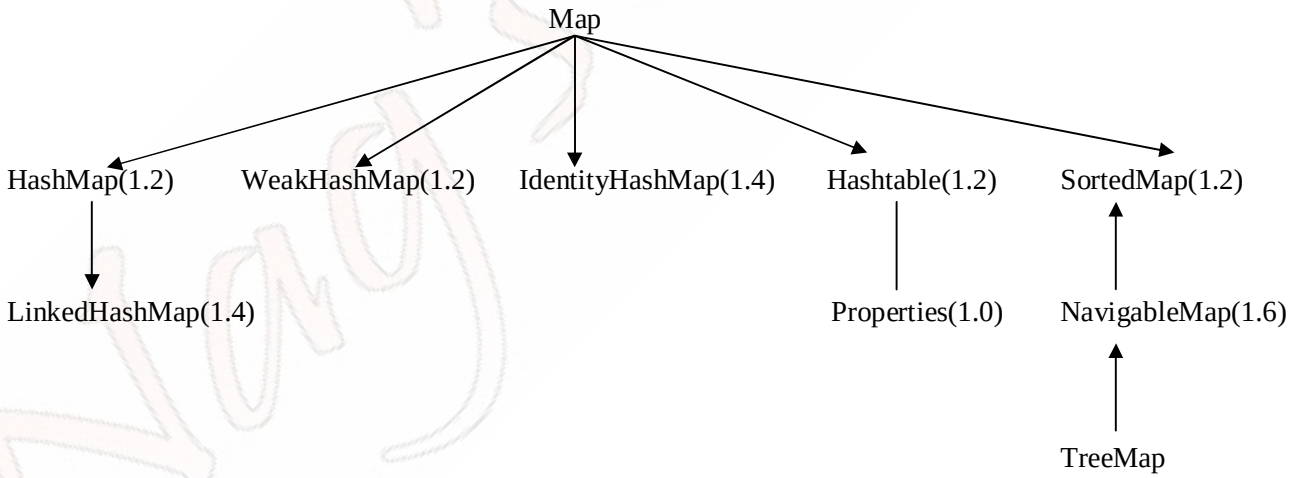


All the above interfaces(collection, List, Set, SortedSet, Queue) can be used for representing a group of individual objects.
If u want to represent a group of objects as key value pair than we can't use above interfaces.
To handle this requirement sun people has introduced **map** interface.

6) Map:
This can be used for representing a group of objects as key value pairs. Both key and value are objects only.

StudentName → StudentRollNo
phoneNumber → contactdetails
word → meaning
IP Address → Domain-name

Map interface is not child interface of collection.



7) SortedMap:
This can be used for representing a group of objects as key value pairs where all the entries are arranged in some sorting order of keys.
TreeMap is the class which implements SortedMap interface.

The following are legacy Characters in collection Frame Work.

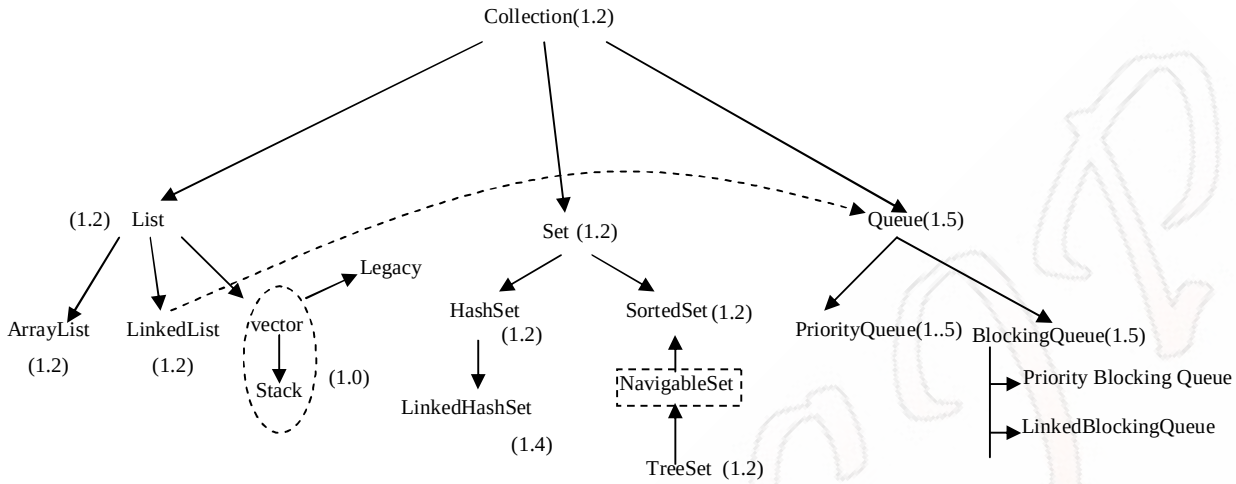
1) vector.
2) stack
3) Hashtable
4) properties

→ Concrete classes

5) Dictionary → abstract classes
6) Enumaration → interface

Collection

This can be used for representing a group of individual objects as a single entity. This interface defines the most common general methods. Which can be applicable for any collection implemented class object.



Colletion Interface methods

1)boolean add(Object obj)
2) boolean addAll(Collection c)
3) boolean remove(Object obj)

4) boolean removeAll(Collection c)
(Removes particular group of objects.)

5) boolean retainAll(Collection c)
(Removes all the elements except those present in 'c')

6) void clear()
(Removies all objects.)

7) boolean contains(Object obj)
(Checks object is there or not.)

8) boolean contains (Collection c)
9) boolean isEmpty()
10) int size()
11) Object [] toArray()

12) Iterator iterator()
(to retrieve the objects one by one.)

List interface

This can be used for representing a group of individual objects where insertion order is preserved and duplicate objects are allowed. By means of index we can preserve insertion order and we can differentiate duplicate objects.

List Interface Defines the following methods

1) boolean add(Object obj)

2) boolean add(int index, Object obj)
3) boolean addAll(Collection c)
4) boolean addAll(int index, Collection c)
5) boolean remove(Object obj)
6) Object remove(int index)
7) Object set(int index, Object new)
Old object. It replaces with the existing objected located at specified index with the new object. And it returns object.

8) Object get(int index)
9) int indexOf(Object obj)
10) int lastIndexOf(Object obj)
11) ListIterator listIterator()

ArrayList()

- The underlying data Structure for ArrayList() is resizable Array or "*Growable Array*".
- Duplicate objects are allowed.
- Insertion order is preserved.
- Heterogeneous objects are allowed.
- 'null' insertion is possible.

Constructors of Array List

ArrayList l = new ArrayList()

Creates an empty ArrayList object with default intitial capacity 10.
When ever ArrayList reaches its max capacity a new ArrayList Object will be created with new capacity.

capacity = (current capacity * 3/2) + 1

ArrayList l = new ArrayList(int initial capacity)

Creates an empty ArrayList Object with the specified initial capacity.

ArrayList l = new ArrayList(Collection c)

For inter conversion between collection objects.

Ex:

```
import java.util.*;
class ArrayListDemo
{
        public static void main(String arg[])
        {
                ArrayList a = new ArrayList();
                a.add("A");
                a.add(new Integer(10));
                a.add("A");
                a.add(null);
                System.out.println(a);
                a.remove(2);
                System.out.println(a);
                a.add(2,"M");
                a.add("N");
                System.out.println(a);

        }
}
```

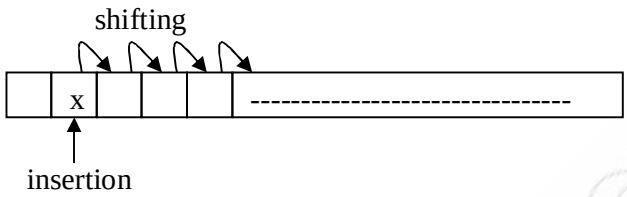
O/P:-

```
[A, 10, A, null]
[A, 10, null]
[A, 10, M, null, N]
```

ArrayList and vector classes implement RandomAccess interface. So that we can access any element with the same speed. Hence ArrayList is best suitable if our frequent operation is retrieval operation.

Usually the collection objects can be used for data transport purpose and hence every collection implemented class already implemented serializable and cloneable interfaces.

ArrayList is the worst choice if u want to perform insertion or deletion in the middle.



Note:- ArrayList is not recommended if the frequent operation is insertion or deletion in the middle. To handle this requirement we should go for linked list.

LinkedList()

- The underlying Data Structure for linked list is doubly linked list.
- Duplicate objects are allowed.
- Insertion order is preserved.
- Heterogeneous objects are allowed.
- 'null' insertion is possible.
- Implements List, Queue, serializable, clonealbe Interfaces But not RandomAccess.

LinkedList is the bestchoice if our frequent operation is insertion or deletion in the middle(no shift operations are required)

LinkedList is the worst choice if our frequent operation is retrieval operation.
LinkedList class usually used for implementing stacks and Queues to provide support for this requirement, LinkedList class contains the following specific methods.

1. void addFirst(Object obj)
2. void addLast(Object obj)
3. Object removeFirst()
4. Object removeLast()
5. Object getFirst()
6. Object getLast()

Constructors

1. LinkedList l = new LinkedList()
2. LinkedList l = new LinkedList(Collection c)

For inter conversion between collection objects

Ex:

```
import java.util.*;
class LinkedListDemo
{
        public static void main(String arg[])
        {
```

```
                LinkedList l = new LinkedList();
                l.add("raju");
                l.add(new Integer(10));
                l.add(null);
                l.add("raju");
                l.set(0, "chinna");
                l.add(0, "Kiran");
                l.addFirst("AAAA");
                l.addLast("ZZZZ");
                System.out.println(l);
        }
}
```

Inter conversion between collection objects.

```
ArrayList al = new ArrayList();
        l1.add(10);
        l1.add(20);
        l1.add(30);
System.out.println("l1--->"+l1);
LinkedList l2 = new LinkedList(l1);
        l2.add(1,5);
        l2.add(3,5);
        l2.add(5,15);
System.out.println("l2--->"+l2);
ArrayList l3 = new ArrayList(l2)
System.out.println("l3--->"+l3);
```

O/P:-

```
l1--->[10, 20, 30]
l2--->[10, 5, 20, 5, 30, 15]
l3--->[10, 5, 20, 5, 30, 15]
```

VectorClass

- The underlying Data structure for the vector is resizable array or growable array.
- Insertion order is preserved.
- Duplicate objects are allowed.
- 'null' insertion is possible.
- Heterogeneous objects are allowed.
- Best choice if the frequent operation is retrieval.
- Worst choice if the frequent operation is insertion or deletion in the middle.
- Vector class implemented serializable, cloneable and RandomAccess Interfaces.

Difference between vector and ArrayList?

Vector	ArrayList
1) All methods of vector are synchronized	1) no method is synchronized
2) vector object is thread safe	2) vector object is not thread safe by default
3) performance is low	3) performance is High
4) 1.0 version(Legacy)	4) 1.2 version(non Legacy)

We can get synchronized version of ArrayList of by using the following method of collection class.

Ex:

```
ArrayList l1 = new ArrayList();
List l2 = Collections.synchronizedList(l1);
```

synchronized Non synchronized

Similarly we can find synchronized versions of set and Map objects by using the collections class method.

```
public static Set synchronizedSet(Set s1)
public static Map synchronizedMap(Map m1)
```

Vector methods

For adding objects.

```
add(Object obj)
add(int index, Object obj)
addElement(Object obj)
```

For removing Objects

```
remove(Object obj)
removeElement(Object obj)
remove(int index)
removeElementAt(int index)
clear()
removeAllElements()
```

For Accessing Elements

```
Object get(int index)
Object elementAt(int index)
Object firstElement();
Object lastElement();
```

OtherMethods

```
int size();
int capacity();
enumaration elements();
```

constructors

```
Vector v = new Vector();
```

Creates an empty vector object with default initial capacity 10, when ever vector reaches it's max capacity a new vector object will be created with.

new capacity = 2 * current capacity

```
Vector v = new Vector(int initialCapacity)
Vector v = new Vector(int initialCapacity, int incrementalCapacity)
Vector v = new Vector(Collection c)
```

Ex:

```
import java.util.*;
class VectorDemo
{
    public static void main(String arg[])
    {
        Vector v = new Vector();
        System.out.println(v.capacity());
        for (int i = 0;i<10 ;i++ )
        {
            v.addElement(i);
        }
```

```
                System.out.println(v.capacity());
                v.addElement("Aa");
                System.out.println(v.capacity());
                System.out.println(v);
        }
    }
```

O/P:-

```
10
10
20
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, Aa]
```

Stack

It is the child class of Vector contains only one constructor.

```
Stack s = new Stack();
```

Methods

Object push(Object obj);
: For inserting an object to the stack

Object pop();
: It removes and returns top of the stack.

Object peak();
: Returns the top of the stack with out removal of object.

int search(Object obj);
: If the specified object is available it returns its offset from top of the stack
: If the object is not available then it returns -1.

boolean empty();
: returns true if the stack is empty otherwise false.

Ex:

```
import java.util.*;
class StackDemo
{
    public static void main(String arg[])
    {
        Stack s = new Stack();
        s.push("A");
        s.push("B");
        s.push("C");
        System.out.println(s);
        System.out.println(s.search("A"));
        System.out.println(s.search("Z"));
    }
}
```

O/P:-

```
[A, B, C]
3
-1
```

Cursors Available in collection frame work

From the collection object to retrieve object we can use the following 3 cursors.

1. Enumeration
2. Iterator
3. ListIterator

Enumeration

This interface has introduced in 1.0 version it contains the following 2 methods.

```
boolean hasMoreElements();
Object nextElement();
```

Ex:

```
import java.util.*;
class EnumaretionDemo
{
    public static void main(String arg[])
    {
        Vector v = new Vector();
        for (int i =0;i<=10 ;i++ )
        {
            v.addElement(i);
        }
        System.out.println(v);
        Enumeration e = v.elements();
        while (e.hasMoreElements())
        {
            Integer i = (Integer)e.nextElement();
            if((i%2) == 0)
            System.out.println(i);
        }
        System.out.println(v);
    }
}
```

O/P:-

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
0
2
4
6
8
10
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Limitations of Enumeration

1. It is applicable only for legacy classes and it is not a universal cursor.
2. While iterating the elements by using enumeration we can perform only read operation and we can't perform any modify/removal operations.

To overcome these problems we should go for Iterator interface.

Iterator

- Introduced in 1.2 version.
- We can get Iterator Object for any collection incremented class i.e it is universal cursor.
- By iterating the elements we can perform remove operation also in addition to read operation.

This interface contains the following 3 methods.

```
boolean hasNext();
Object next();
void remove();
```

Ex:

```
import java.util.*;
class IteratorDemo
{
    public static void main(String arg[])
    {
        ArrayList al = new ArrayList();
        for (int i =0;i<=10 ;i++ )
        {
```

```
                    al.add(i);
                }
                System.out.println(al);
                Iterator itr = al.iterator();
                while (itr.hasNext())
                {
                    Integer i = (Integer)itr.next();
                    if((i%2) == 0)
                    {
                        System.out.println(i);
                    }
                    else
                    {
                    itr.remove();
                    }
                }
                System.out.println(al);
            }
        }
```

O/P:-

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
0
2
4
6
8
10
[0, 2, 4, 6, 8, 10]
```

Note:-

1. Enumeration and Iterator are single directional cursors. They can always move to words forward direction only.
2. By using Iterator we can perform read and remove operations. And we can't perform any replace or addition of new objects

To over come these limitations we should go for list Iterator interface().

ListIterator

1.It has introduced in 1.2 version and it is child interface of Iterator.
2.It is a bidirectional cursor.
3.i.e Based on requirement we can move either to the forward or backward direction.
4.While Iterating we can perform replace and add operation in addition to read and remove this interface defines the following 9 months.

1. boolean hasNext();
2. boolean hasPrevious();
3. Object next();
4. Object previous();
5. int nextIndex();
6. int previousIndex();
7. void remove();
8. void set(Object new)
9. void set(Object new)

nextIndex(): If there is no next element it returns size of the list.
previousIndex(): If there is no previous element it returns -1 .

```
import java.util.*;
class ListIteratorDemo
{
    public static void main(String arg[])
```

```
        {
            LinkedList l = new LinkedList();
            l.add("balakrishna");
            l.add("chiru");
            l.add("venky");
            l.add("nag");
            System.out.println(l);
            ListIterator ltr = l.listIterator();
            while (ltr.hasNext())
            {
                String s = (String)ltr.next();
                if(s.equals("nag"))
                {
                    ltr.add("chaitanya");
                }
            }
            System.out.println(l);
        }
    }
```

The most powerful cursor is listIterator. But it's main limitation is it is applicable only for list implemented classes (ArrayList, LinkedList, Vector, Stack).

Comparision between All the Three cursors

Properties	Enumeration	Iterator	ListIterator
1) It is Legacy?	Yes	No	No
2) It is applicable	Only for legacy classes	For any collection	Only for list implemented classes
3) How to get?	By using elements()	By Using iterator()	By using listIterator()
4) Accessibility	Only read	read & remove	read/remove/replace/add
5) Movement	Single direction (Only Forward)	Single direction (Only Forward)	Biderctional
6) Methods	hasMoreElements() nextElements()	hasNext() next() remove()	hasNext() hasPrevious() . .
7) version	1.0	1.2	1.2

SetInterface

This can be used for representing a group of Individual objects where insertion order is not preserved and duplicate objects are not allowed.
Set interface is child interface of Collection.
This interface doesn't contain any new method and we have to use only collection Interface methods.

HashSet

- The underlying Data Structure for HashSet is Hashtable.
- Insertion order is not preserved and it is based on has code of the Object.
- Duplicate objects are not allowed. Violation leads to no CompileTimeError or RuntimeError, add method simply returns false.

- 'null' insertion is possible.
- Heterogeneous objects are allowed.
- HashSet is the best choice if the frequent operation is Search Operation.

Constructors

1. HashSet h = new HashSet()
   Creates an empty HashSet object with default initial value 16 and default fill ratio 0.75
2. HashSet h = new HashSet(int initialcapacity)

3. HashSet h = new HashSet(int initialCapacity, float fillratio)
   Here fillratio is 0 or 1.
4. HashSet h = new HashSet(Collection c)
   Ex:

```
import java.util.*;
class HashSetDemo
{
        public static void main(String arg[])
        {
                HashSet h = new HashSet();
                h.add("B");
                h.add("C");
                h.add("D");
                h.add("Z");
                h.add(null);
                h.add(new Integer(10));
                System.out.println(h.add("Z"));
                System.out.println(h);
        }
}
```

O/P:-

```
false
[Z, D, null, C, B, 10]
```

LinkedHashSet

It is the child class of HashSet. It is Exactly similar to HashSet. Except the following differences.

HashSet	LinkedHashSet
1) The underlying Data Structure is Hashtable	1) The underlying Data Structures are Hashtable and LinkedList
2) Insertion Order is not preserved	2) Insertion Order is preserved.
3) Introduced in 1.2 version	3) Introduced in 1.4 version

In the above program if u r replacing 'HashSet' with 'LinkedHashSet' the following is the output.

```
false
[B, C, D, Z, null, 10]
```

Note:- For implementing *caching* application the best suitable Data structure is LinkedHashSet and LinkedHashMap where duplicate objects are not allowed and insertion order Must be preserved.

CacheMemory

SortedSet

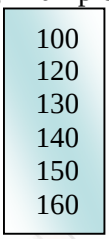
- This can be used for representing a group of individual objects where duplicate objects are not allowed.
- Insertion order is not preserved but all the elements are inserted according to some sorting order of elements. The sorting order may be default natural sorting order or customized sorting order.

SortedSet Interface contains the following more specific methods

1. Object first()
   Returns first element in the SortedSet
2. Object last()
3. SortedSet headSet(Object obj)
   Returns the SortedSet contains the elements which are less than object obj.
4. SortedSet tailSet(Object obj)
   Returns the SortedSet whose elements are greater than or equal to object obj
5. SortedSet subSet(Object obj1, Object obj2)
   Returns the SortedSet whose elements are >= obj1 but < Obj2
6. Comparator comparator();
   Returns the comparator object describes the underlying sorting technique.

If you are using default(Assending) natural sorting order it returns null.

Observe the following Example.



first() → 100
last() → 160
headSet(140) → {100,120,130}
tailSet(140) → {140,150,160}
subset(130,150)→{130,140}
comparator() → null

TreeSet

- The underlying Data structure for the TreeSet is Balanced tree.
- Duplicate objects are not allowed. If we are trying to insert duplicate object we won't get any compile time error or Run time error, add method simply returns false.
- Insertion order is not preserved but all the elements are inserted according to some sorting order.
- Heterogeneous objects are not allowed, violation leads to Run time error saying class cast Exception

Constructors

1. TreeSet t = new TreeSet();
   Creates an empty TreeSet Object where the sorting order is default natural sorting order.
2. TreeSet t= new TreeSet(Comparator c)
   Creates an empty TreeSet Object where the Sorting order is specified by comparator object.
3. TreeSet t= new TreeSet(Collection c)
4. TreeSet t= new TreeSet(SortedSet s)
   Creates TressSet Object for a given SortedSet.

Ex:

```
import java.util.*;
class TreeSetDemo
{
	public static void main(String arg[])
	{
		TreeSet t = new TreeSet();
		t.add("A");
		t.add("B");
		t.add("Z");
		t.add("L");
		//t.add(new Integer(10));	→ ClassCastException.
		//t.add(null);			→ NullPointerException.
		t.add("A");			→ false.
		System.out.println(t);
	}
}
```

null Acceptance

For the empty TreeSet as the first element null insertion is possible. But after inserting null if we are trying to insert any other element we will get NullPointerException.

If the TreeSet already contains some elements if we are trying to insert null we will get NullPointerException.

Ex:

```
import java.util.*;
class TreeSetDemo
{
	public static void main(String arg[])
	{
		TreeSet t = new TreeSet();
		t.add(new StringBuffer("A"));
		t.add(new StringBuffer("B"));
		t.add(new StringBuffer("T"));
		t.add(new StringBuffer("Z"));

		System.out.println(t);
	}
}
```

O/P:- R.E: ClassCastException.

If we are depending on natural sorting order compulsory the objects should be homogeneous and comparable other wise we will get class cast Exception.

An object is said to be comparable(if and only if) the corresponding class has to implement comparable interface.

All wrapper classes and String class already implemented comparable interface. But the String buffer doesn't implement comparable interface. Hence in the above program we got class cast exception.

Comparable Interface

It is available in *java.lang package*. This interface contains only one method compareTo()

```
public int compareTo(Object obj)
obj1.compareTo(obj2)
```

return –ve if obj1 has to come before obj2.
+ ve if obj1 has to come after obj2.
0 if obj1 and obj2 are equal(Duplicate Objects).

Ex:

```
System.out.println("A".compareTo("Z"));	→ -ve
System.out.println("K".compareTo("A"));	→ _ve
```

System.out.println("K".compareTo("K")); → 0
System.out.println("a".compareTo("A")); → +ve
System.out.println("A".compareTo(new Integer(10))); → classCastException
System.out.println("A".compareTo(null)); → NullPointerException

Note: while Inserting the objects into the TreeSet JVM internally uses compareTo() method if we are depending on natural sorting order.

Sometimes we have to define our own customized sorting order, then we should go for comparator Interface.

Comparator Iterface

By using comparator object we can define our own customized sorting.

Comparator Interface is available in *java.util package*. This interface defines the following 2 methods.

1) public int compare(Object obj1, Object obj2)

returns –ve if obj1 has to come before obj2
+ve if obj1 has to come after obj2
0 if obj1 and obj2 are equal.

2) public boolean equals(Object obj)

When ever we are implementing comparator interface compulsory we should provide implementation for compare method. Implementing equals method is optional because it is already available from object class through inheritance.

Write a program to insert integer objects into the Treeset where the sorting order is descending order.

```
import java.util.*;
class TreeSetDemo
{
	public static void main(String arg[])
	{
		TreeSet t = new TreeSet(new MyComparator());
		t.add(10);
		t.add(5);
		t.add(15);
		t.add(20);
		t.add(0);
		System.out.println(t);
	}
}
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 -1;
		else
			return 0;
	}
}
```

O/P:- [20,15,10,5,0]

Flow of control in MyComparator

compare(10, 5) → 10, 5
compare(15, 10) → 15, 10, 5
compare(20, 15) → 20, 15, 10, 5
compare(0, 20)

compare(0, 15)
compare(0, 10)
compare(0, 5) → 20, 15, 10, 5, 0

if we are not passing comparator object JVM will always call compareTo() method which is meant for default natural sorting order.

Case1: If we implement compare method as follows the outputs are

```
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 I2.compareTo(I1);      → [20,15,10,5,0]
    return -I1.compareTo(I2);     → [20,15,10,5,0]
    return -I2.compareTo(I1);     → [0,5,10,15,20]
    return I2-I1;                 → [20,15,10,5,0]
    return I1-I2;                 → [0,5,10,15,20]
}
```

Case2: If we implement compare method as follows

```
public int compare(Object obj1, Object obj2)
{
    return -1;    → [0,20,15,5,10] (reverse of Insertion order)
    return  1;    → [10,5,15,20,0] (Insertion Order)
    return  0;    → [10] (All the remaining elements considered as duplicate objects)
}
```

Write a program to insert String Objects to the TreeSet Where the Sorting order is reverse of Alphabetical order.

```
import java.util.*;
class TreeSetDemo
{
    public static void main(String arg[])
    {
        TreeSet t = new TreeSet(new MyComparator());
        t.add("chiru");
        t.add("venky");
        t.add("nagaruna");
        t.add("balaiah");
        t.add("saikumar");
        t.add("suman");
        System.out.println(t);

    }
}
class MyComparator implements Comparator
{
    public int compare(Object obj1, Object obj2)
    {
        String s1 = (String)obj1;
        String s2 = (String)obj2;
        return s2.compareTo(s1);
    }
}
```

O/P:-

```
[venky, suman, saikumar, nagaruna, chiru, balaiah]
```

Map Interface

The Map interface is not an extension of Collection interface. Instead the interface starts of it's own interface hierarchy, for maintaining key-value associations. The interface describes a mapping from keys to values, without duplicate keys, by defination.

The Map interface provides three collection views, which allow a map's contents to be viewed as a set of keys, collection of values, or set of key-value mappings. The order of a map is defined as the order in which the iterators on the map's collection views return their elements. Some map implementations, like the TreeMap class, make specific guarantees as to their order; others, like the HashMap class, do not.

Key-Value pairs

Key-value pairs are stored in *maps*.

Map interfaces

- *Map* implemented by HashMap and TreeMap
- *SortedMap* implemented by TreeMap.
- *Map.Entry* which describes access methods to the key-value pairs.

Implementing classes

A number of classes implement the Map interface, including HashMap, TreeMap, LinkedHashMap, WeakHashMap, ConcurrentHashMap, and Properties. The most generally useful class is HashMap.

- java.util.**HashMap** is implemented with a hash table. Access time is O(1). Entries are unsorted.
- java.util.**LinkedHashMap** is implemented with a hash table. Access time is O(1). Entries are sorted in either entry order or order of last access, which is useful for implementing a LRU (least recently used) caching policy.
- java.util.**TreeMap** is implemented as a balanced binary tree. Access time is O(log N). Entries are sorted.

Map interface methods

Here are some of the most useful Map methods. `m` is a Map, `b` is a boolean, `i` is an int, `set` is a Set, `col` is a Collection, `key` is an Object used as the key used to store a value, `val` is an Object stored as the value associated with the key.

Result	Method	Description
Adding key-value pairs to a map		
`obj =`	`m.put(key, val)`	Creates mapping from `key` to `val`. It returns the previous value (or null) associated with the key.
	`m.putAll(map2)`	Adds all key-value entries from another map, *map2*.
Removing key-value pairs from a map		
	`m.clear()`	Removes all elements from a Map
`obj =`	`m.remove(key)`	Deletes mapping from `key` to anything. Returns previous value (or null) associated with the key.
Retrieving information from the map		
`b =`	`m.containsKey(key)`	Returns true if `m` contains a key `key`
`b =`	`m.containsValue(val)`	Returns true if `m` contains `val` as one of the values
`obj =`	`m.get(key)`	Returns value corresponding to `key`, or `null` if there is no mapping. If

		`null` has been stored as a value, use `containsKey` to see if there is a mapping.
b =	*m*.**isEmpty**()	Returns `true` if *m* contains no mappings.
i =	*m*.**size**()	Returns number of mappings in *m*.
Retrieving all keys, values, or key-value pairs (necessary for iteration)		
set =	*m*.**entrySet**()	Returns set of Map.Entry values for all mappings.
set =	*m*.**keySet**()	Returns Set of keys.
col =	*m*.**values**()	Returns a Collection view of the values in *m*.

Map.Entry interface methods

Each element is a map has a *key* and *value*. Each key-value pair is saved in a `java.util.Map.Entry` object. A set of these map entries can be obtained by calling a map's `entrySet()` method. Iterating over a map is done by iterating over this set.

Assume in the following table that `me` is a `Map.Entry` object.

Result	Method	Description
obj =	*me*.**getKey**()	Returns the key from the pair.
obj =	*me*.**getValue**(*key*)	Returns the value from the Map pair.
obj =	*me*.**setValue**(*val*)	This is an *optional* operation and may not be supported by all `Map.Entry` objects. Sets the value of the pair, which modifies the `Map` which it belongs to. Returns the orginal value.

The interface methods can be broken down into three sets of operations: *altering*, *querying* and providing *alternative* views

The *alteration operation* allows you to add and remove key-value pairs from the map. Both the key and value can be null. However you should not add a Map to itself as a key or value.

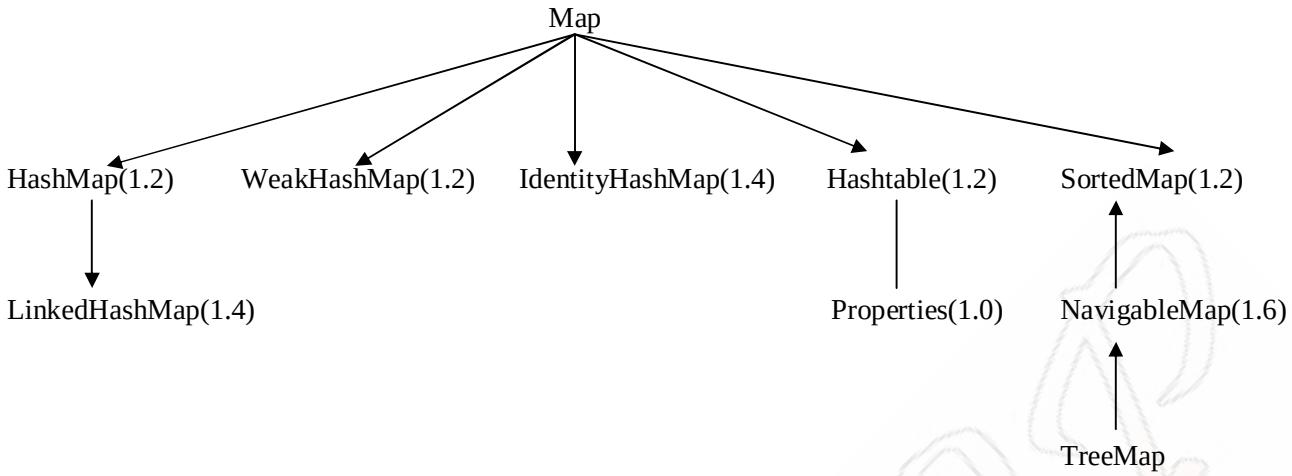
```
Object put(Object key, Object value)
Object remove(Object key)
void putAll(Map t)
void clear()
```

The *query operations* allow you to check on the contents of the map

```
Object get(Object key)
boolean containsKey(Object key)
boolean containsValue(Object value)
int size()
boolean isEmpty()
```

The set methods allow you to work with the *group of keys* or values as a collection

```
Set keySet()
Collection values()
Set entrySet()
```



HashMap

HashMap is not sorted or ordered. If you just need a Map, and you don't care about the order of the elements while iterating through it, you can use a HashMap. That keeps it simple. The values can be null. But the key should ne unique, so you can have one null value for key. (Allows only one null key).

HashMap class constructors

In addition to implemented the *Map* interface methods, HashMap has the following constructors.

Result	Constructor	Description
hmap =	`new HashMap()`	Creates a new HashMap with default initial capacity 16 and load factor 0.75.
hmap =	`new HashMap(`*`initialCapacity`*`)`	Creates a new HashMap with the specified initial int capacity.
hmap =	`new HashMap(`*`initialCapacity, loadFactor`*`)`	Creates a new HashMap with the specified capacity which will not exceed a specified (float) load factor.
hmap =	`new HashMap(`*`mp`*`)`	Creates a new HashMap with elements from the *Map mp*

LinkedHashMap

LinkedHashMap will keep the order in which the elements are inserted into it. If you will be adding and removing elements a lot, it will be better to use HashMap, because LinkedHashMap will be slower to do those operations. But, you can iterate faster using LinkedHashMap. So, if you will be iterating heavily, it may be a good idea to use this.

WeakHashMap

A map of *weak keys* that allow objects referred to by the map to be released; designed to solve certain types of problems. If no references outside the map are held to a particular key, it may be garbage collected.

IdentityHashMap

A hash map that uses == instead of **equals( )** to compare keys. Only for solving special types of problems; not for general use.

Hashtable

Hashtable is almost the same as HashMap. The main differences are:
1. Hashtable does not let you have null value for key.
2. The key methods of Hashtable are synchronized. So, they may take a longer time to execute, compared to HashMap's methods.

SortedMap

If you have a **SortedMap** (of which **TreeMap** is the only one available), the keys are guaranteed to be in sorted order, which allows additional functionality to be provided with these methods in the **SortedMap** interface:

Comparator comparator( ): Produces the comparator used for this **Map**, or **null** for natural ordering.

Object firstKey( ): Produces the lowest key.

Object lastKey( ): Produces the highest key.

SortedMap subMap(fromKey, toKey): Produces a view of this **Map** with keys from **fromKey**, inclusive, to **toKey**, exclusive.

SortedMap headMap(toKey): Produces a view of this **Map** with keys less than **toKey**.

SortedMap tailMap(fromKey): Produces a view of this **Map** with keys greater than or equal to **fromKey**.

SortedMap interface methods

The *SortedMap* interface is used by TreeMap and adds additional methods to reflect that a TreeMap is sorted.

Result	Method	Description
comp =	`comparator()`	Returns Comparator used to compare keys. null if natural ordering used (eg, String).
obj =	`firstKey()`	Key of first (in sorted order) element.
obj =	`lastKey()`	Key of last (in sorted order) element.
smp =	`headMap(`*obj*`)`	Returns *SortedMap* of all elements less than *obj*.
smp =	`tailMap(`*obj*`)`	Returns *SortedMap* of all elements greater than or equal to *obj*.
smp =	`subMap(`*fromKey*`, `*toKey*`)`	Returns *SortedMap* of all elements greater than or equal to *fromKey* and less than *toKey*.

TreeMap

TreeMap is a sorted Map and will be sorted by the natural order of the elements. If you want, you can define a custom sort order by means of a Comparator passed to the constructor. So, when you need a custom sort order, it is clear which class you should be using!

TreeMap class constructors

TreeMap implements the *Map* and *SortedMap* interface methods. In constrast to HashMap, TreeMap keeps the balanced binary tree in sorted order by key. If the key has a natural order (eg, String) this is ok, but often you will supply a Comparator object that tells how two keys compare. It has the following constructors.

Result	Constructor	Description
tmap =	`new TreeMap()`	Creates new TreeMap. Keys sorted by natural order.

tmap =	`new TreeMap(`*comp*`)`	Creates new TreeMap using Comparator *comp* to sort keys.
tmap =	`new TreeMap(`*mp*`)`	Creates new TreeMap from *Map mp* using natural ordering.
tmap =	`new TreeMap(`*smp*`)`	Creates new TreeMap from *SortedMap smp* using key ordering from *smp*.

Generics Introduction

Array objects are by default typesafe. i.e we declare String Array we can insert only String Objects. By Mistake if we r trying to insert any other elements we will get compile time error.

Ex:

```
String [] s = new String[100];
s[0] = "raju";
s[1] = 10;        →C.E
```

But Collection objects are not typesafe by default. If our requirement is to add only String objects to the ArrayList , By mistake if we r trying to insert any other element we won't get any compile time error.

```
ArrayList l = new ArrayList();
l.add("raju");
l.add(new Integer(10));          → No Compile Time Error.
```

While retrieving Array elements there is no need to perform typecasting.

```
String name = s[0];              → No typecasting required here.
```

But while retrieving the elements from ArrayList compulsory we should perform typecasting

```
String name = l.get(0);          → C.E
String name = (String)l.get(0);  ✓
```

To resolve the above 2 problems (typesafety, typecasting) sun people introduced generics concept in the 1.5 version.If we want to create ArrayList object to hold any String Objects we have to define as follows.

```
ArrayList<String> l = new ArrayList<String>();
```

For this ArrayList we have to add only String objects. By mistake if we r trying to add any other type we will get compile time error.

```
l.add("valid");
l.add(new Integer(10));          X
```

C.E:- can't find the symbol add(Integer)

At the time of retrieval no need to perform any typecasting.

```
String name = l.get(0);→ No typecasting is required.
```

Hence by using generics we can provide typesafety and we can resolve typecasting problems.

By using generics we can define parameter for the collection. These parameterized collection classes are nothing but "Generic collection classes".

Polymorphism concept is not applicable for the parameter type but applicable for basetype

```
List<String> l = new ArrayList<String>()
```

Base type Parameter type ✓

List<*object*> l = new ArrayList<*String*>(); X

C.E:- Incompatable Types
Found:ArrayList<String>
Required :List<Object>

The type parameter must be object type(any class or interface name). we can't apply generic concept for primitive datatype.

Ex:- ArrayList<*int*> l = new ArrayList<*int*>();
C.E:- unexpected type found : int
Required : Integer

→ **Which of the following declarations are valid**

1) ArrayList<*Integer*> l = new ArrayList<*Integer*>(); ✓
2) ArrayList<*Number*> l = new ArrayList<*Integer*>(); X
3) ArrayList<Runnable> l = new ArrayList<*Runnable*>(); ✓
4) ArrayList<*long*> l = new ArrayList<*Long*>(); X
5) ArrayList<*Object*> l = new ArrayList<*StringBuffer*>(); X

Generic Classes

Until 1.4 version we have ArrayList class with the declarations as follows.

```
Class ArrayList
{
        add(Object o);
        Object get(int index);
}
```

add method contain Object as argument, hence we can add any kind of object.

As a result we can't get any type safety. The return type of get() method is object hence at the time of retrieval we should perform typecasting.
But in the 1.5 version we have generic ArrayList class with the definition as follows

```
class ArrayList<T>
{
        add(T t);
        T get(int);
}
```

Based on runtime requirement the corresponding version of ArrayList will loaded.

ArrayList<*String*> l = new ArrayList<*String*>();

For the following declarations the corresponding loaded class is

```
class ArrayList<String>
{
        add(String);
        String get(int);
}
```

The argument to the add method is String hence we should add only String Object as the result we will get type safety.
The return type of get() method is String. Hence at the time of retrieval no need to perform typecasting.

We can define our own generic classes also

Ex:

```
class gen<T>
{
```

```
    T ob;
    gen(T ob)
    {
        this.ob = ob;
    }
    public void show()
    {
        System.out.println("The type of Object is : "+ob.getClass().getName());
    }
    public T getOb()
    {
        return ob;
    }
}
class GenericsDemo
{
    public static void main(String[] args)
    {
        gen<String> g1 = new gen<String>("raju");
        g1.show();
        System.out.println(g1.getOb());

        gen<Integer> g2 = new gen<Integer>(10);
        g2.show();
        System.out.println(g2.getOb());
    }
}
```

O/P:-
```
The type of  Object is : java.lang.String
raju
The type of  Object is : java.lang.Integer
10
```

BoundedTypes

We can bound the type parameter for a particular range. Such type of types is called *bounded types*.

We can achieve this by using extends keyword.

Ex:

```
class Gen<T>
{
}
```

Here we can pass any types as the type parameter and there are no restrictions.

```
Gen<String> g1 = new Gen<String>();      ✓
Gen<Integer> g2 = new Gen<Integer>();    ✓
```

Ex:

```
Class Gen<T extends X>
```

If 'X' is a class then any type which is the child class of 'X' is allowed as the type parameter.
If 'X' is an interface then any type which is the implementation class of 'X' is allowed as the type parameter.

Ex:

```
class Gen<T extends Number>
{
}
```

In this case as the type parameter we can take either number or it's child classes.

```
Gen<Integer> g1 = new Gen<Integer>();   ✓
Gen<String> g2 = new Gen<String>();    ✗
```

Because type parameter String is not with in it's bound

```
class Gen<T extends Runnable>
{
}
```

In this case as the type parameter we can take any implementation class of Runnable interface.

```
Gen<Thread> t1 = new Gen<Thread>();
```

Because Thread is implementation class of Runnable

```
Gen<String>   t2 = new Gen<String>();
```

Because String is not implementation class of Runnable interface .

In generics we have only extends keyword and there is no implements keyword. It's purpose is also survived by using extends keyword only.

Ex:

```
class Gen<T extends Number>
{
    T ob;
    Gen(T ob)
    {
        this.ob = ob;
    }
    void show()
    {
        System.out.println("The int value is :" + ob.intValue());
    }
}
class GenDemo
{
    public static void main(String arg[])
    {
        Gen<Integer> t1 = new Gen<Integer>(new Integer(10)); ✓
        t1.show();
        Gen<Double> t2 = new Gen<Double>(10.5);          ✓
        t2.show();
        Gen<String> t3 = new Gen<String>("raju");        ✗
        t3.show();
    }
}
```

Generic Methods

1) m1(ArrayList<String>)
   It is applicable for ArrayList of only String type.
2) m1(ArrayList<? extends x> l)
   Here if 'x' is a class then this method is applicable for ArrayList of either x or it's child classes.
   If 'x' is an interface then this method is applicable for ArrayList of any implementation class of x
3) m1(ArrayList <? Super x> l)
   If 'x' is a class then this method is applicable for ArrayList of either x or it's super classes.
   If 'x' is an interface then this method is applicable for ArrayList of any super classes of implemented class of x.
4) m1(ArrayList <?> l)
   This method is applicable for ArrayList of any type.

In the method declaration if we can use '?' in that method we are not allowed to insert any element except null. Because we don't know exactly what type of object is coming.

Ex:

```
import java.util.*;
class Test
{
	public static void main(String arg[])
	{
		ArrayList<String> l1 = new ArrayList<String>();
		l1.add("A");
		l1.add("B");
		l1.add("C");
		l1.add("D");
		m1(l1);
		ArrayList<Integer> l2 = new ArrayList<Integer>();
		l2.add(10);
		l2.add(20);
		l2.add(30);
		l2.add(40);
		m1(l2);
	}
	public static void m1(ArrayList<?> l)
	{
		//l.add("D");   → C.E: Because we can't expect what type of value
		                   will come
		l.remove(1);
		l.add(null);
		System.out.println(l);
	}
}
```

O/P:- [A, C, D, null]
[10, 30, 40, null]

Which of the following declarations are valid?

```
ArrayList<String> l = new ArrayList<String>();                → ✓
ArrayList<Object> l = new ArrayList<String>();                → ✗
ArrayList<? extends Object> l = new ArrayList<String>();      → ✓
ArrayList<? extends String> l = new ArrayList<String>();      → ✓
ArrayList<? Super String> l = new ArrayList<String>();        → ✓
ArrayList<? Super Runnable> l = new ArrayList<Object>();      → ✓
ArrayList<? Super Runnable> l = new ArrayList<Thread>();      → ✗
ArrayList<?> l = new ArrayList<Integer>();                    → ✓
ArrayList<?> l = new ArrayList<? extends Number>();           → ✗
ArrayList<?> l = new ArrayList<? super Number>();             → ✗
ArrayList<?> l = new ArrayList<?>();                          → ✗
```

We can use Unicode character '?' in the declaration part but not in the construction part.

Communication with legacy non-generic code

To provide compatibility with old non-generic versions sun people compromised the generics concept in some places, the following is one such place

Ex:

```
import java.util.*;
class Test
{
	public static void main(String arg[])
	{
```

```
                ArrayList<String> l = new ArrayList<String>();
                l.add("A");
                l.add("B");
                l.add("C");
                //l.add(10);
                m1(l);
                System.out.println(l);
        }
        public static void m1(ArrayList l)
        {
                l.add(new Integer(10));
                l.add(new StringBuffer("raju"));
        }
}
```

O/P:- [A, B, C, 10, raju]

Note:

Generics is the concept applicable only at compile time to provide type safety, at runtime there is no generic concept at all.

Ex:

```
ArrayList l = new ArrayList<String>();
l.add("A");                     → ✓
l.add(new Integer(10));         → ✓
l.add(new StringBuffer("raju"));
ArrayList<String>l = new ArrayList();
l.add("A");     → ✓
l.add(10);      → ✗
l.add("B");     → ✓
l.add(null);    → ✓
l.add(20);      → ✗
```

12

FILE I/O & SERIALIZATION

File I/O Introduction

The following is the list methods going to cover in File I/O Concept.

1) File
2)FileWriter
3) FileReader
4) BufferedWriter
5)BufferedReader
6) printWriter

File

A java file object represent just name of the file/directory.

File f = new File("abc.txt");

If 'abc.txt' is already available then 'f' will represent that physical file.
If it is not already available, It won't create any new file and 'f' simply represents the name of the file.

Ex:

```
import java.io.*;
class test
{
        public static void main(String[] args)
        {
                File f =  new File("cba.txt");
                System.out.println(f.exists()); → false at first time.
                f.createNewFile();
                System.out.println(f.exists()); → true

        }
}
```

I Run:

```
false
true
```

II Run:

```
true
true
```

A java file Object can represent directories also

Ex:

```
File f =  new File("bbc");
System.out.println(f.exists());
f.mkdir();
System.out.println(f.exists());
```

I Run:

false
true

II Run:

true
true

The constructors of the file class

1) File f = new File(String name)

Here name may be file or directory name.

Creates a java file object that represents a file or directory name.

2) File f = new File(String subdirec, String name)

Creates a java file object that represents file or directory name present in specified subdirectory.

3) File f = new File(File subdir, String name)

Important methods of File Class

1) boolean exists():

returns true if the physical file/directory presents other wise false.

2) boolean createNewFile():

returns ture if it creates a new file, if the required file is already available then it won't create any new file and returns false.

3) booelan mkdir()

For creation of directory.

4) boolean isFile():

returns true if the java file object represents a file.

5) boolean isDirectory():

returns true if the java file object represents a directory.

6) String [] list():

returns the names of files and directories present in the directories represented by the file object.

If the java file object represents a file instead of directory this method returns null.

7) Boolean delete():

for deleting the file or directory represented by java file object.

write a program to create a file named with 'xyz.txt' in the current working directory.

```
File f = new File("xyz.txt");
f.createNewFile();
```

Write a program to create a directory 'raju123' in the current working directory and create a file 'file1.txt' in that directory.

```
File f = new File("raju123");
f.mkdir();
//  File f1 = new File("raju123","file1.txt");
File f1 = new File(f,"file1.txt");
F1.createNewFile();
```

Write a program to list the names of files and directories in 'jdk' directory.

```
File f = new File("jdk");
String [] s = f.list();
For(String s1: s)
{
        System.out.println(s1);
}
```

FileWriter

This class can be used for writing character data to the file.

Constructors

1) FileWriter fw = new FileWriter(String fname)
2) FileWriter fw = new FileWriter(File f);

The above 2 constructors creates a file object to write character data to the file.
If the file already contains some data it will overwrite with the new data.
Instead of overriding if u have to perform append then we have to use the following constructors.

FileWriter fw = new FileWriter(String name, boolean append);
FileWriter fw = new FileWriter(File f, boolean append);

If the underlying physical file is not already available then the above constructors will create the required file also.

Important methods of FileWriter Class

1) void write(int ch) throws IOException
   for writing single character to the file.
2) void write(String s)throws IOException.
3) void write(char [] ch) throws IOException.
4) void flush():-To guaranteed that the last character of the data should be required to the file.

Ex:-

```
class test
{
    public static void main(String arg[])throws Exception
    {
        File f = new File("pongal.txt");
        System.out.println(f.exists());
        FileWriter fw = new FileWriter(f,true);
        System.out.println(f.exists());
        fw.write(97);
        fw.write("run\nsoftware\n");
        char [] ch1 = {'a','b','c'};
        fw.write(ch1);
        fw.flush();
        fw.close();
    }
}
```

FileReader

This class can be used for reading character data from the file.

Constructors

1) FileReader fr = new FileReader(String name);
2) FileReader fr = new FileReader(File f);

Important methods of FileReader Class

1) int read():
   for reading next character from the file. If there is no next character this method returns -1

2) int read(char[] ch):
   to read data from the file into char array.
3) void close():
   to close FileReader

Ex:

```
class test
{
    public static void main(String arg[])throws Exception
    {
        File f = new File("pongal.txt");
        FileReader fr = new FileReader(f);
        System.out.println(fr.read());
        char [] ch2 = new char[(int) (f.length())];
        System.out.println(ch2.length);
        fr.read(ch2);
        for(char ch1: ch2)
        {
            System.out.print(ch1);
        }
    }
}
```

The usage of FileReader and FileWriter is in efficient because

- While writing the data by using FileWriter, program is responsible to insert line separators manually.
- We can read the data character by character only by using FileReader. If increases the number of I/O operations and effect performance.

To overcome these problems sun people has introduced **BufferedReader** and **BufferedWriter** classes.

BufferedWriter

This can be used for writing character data to the file.

Constructors

1) BufferedWriter bw = new BufferedWriter(writer w)
2) BufferedWriter bw = new BufferedWriter(writer r, int size)
   BufferedWriter never communicates directly with the file. It should Communicate through some writer object only.

Q) Which of the following are valid declarations

1) BufferedWriter bw = new BufferedWriter(“abc.txt”); X
2) BufferedWriter bw = new BufferedWriter(new File(“abc.txt”)); X
3) BufferedWriter bw = new BufferedWriter(new FileWriter(“abc.txt”)); ✓
4) BufferedWriter bw = new BufferedWriter(new BufferedWriter(new FileWriter(“abc.txt”))); ✓
5)

Important methods of BufferedWriter Class

1) void write(int ch) thorows IOException
2) void write(String s) throws IOException
3) void write(char[] ch) throws IOException
4) void newLine()
   for inserting a new line character.
5) void flush()
6) void close()
7)

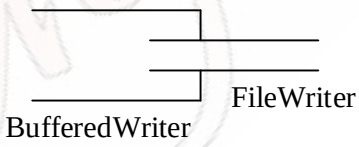
Which method is available in BufferedWriter and not available in FileWriter

Ans: newLine() method

Ex:-

```
class test
{
    public static void main(String arg[])throws Exception
    {
        File f = new File("pongal.txt");
        System.out.println(f.exists());
        FileWriter fw = new FileWriter(f);
        BufferedWriter bw = new BufferedWriter(fw);
        bw.write(97);
        bw.newLine();
        char [] ch1 = {'a','b','c','d'};
        bw.write(ch1);
        bw.newLine();
        bw.write("raju");
        bw.newLine();
        bw.write("software");
        bw.flush();
        bw.close();
    }
}
```

Note:- When ever we r closing BufferedWriter ,automatically underlying FileWriter object will be closed.



BufferedReader

By using this class we can read character data from the file.

Constructors

1) BufferedReader br = new BufferedReader(Reader r)
2) BufferedReader br = new BufferedReader(Reader r, int buffersize)
   BufferedReader never communicates directly with the file. It should Communicate through some reader object only.

Important methods of BufferedReader Class

1) int read()
2) int read(char [] ch)
3) String readLine();
   Reads the next line present in the file. If there is no nextline this method returns null.
4) void close()

Ex:

```
class test
{
    public static void main(String arg[])throws Exception
    {

        FileReader fr = new FileReader("pongal.txt");
        BufferedReader br = new BufferedReader(fr);
        String s = br.readLine();
        while(s != null)
        {
            System.out.println(s);
            s = br.readLine();
        }
        br.close();
    }
}
```

Note:- When ever we r closing BufferedReader ,automatically underlying FileReader object will be closed.

PrintWriter

The most enhanced writer for writing character data to the file is PrintWriter()

Constructors

1) PrintWriter pw = new PrintWriter(String fname)
2) PrintWriter pw = new PrintWriter(File f);
3) PrintWriter pw = new PrintWriter(Writer w);

Important methods

1) write(int ch)
2) write(char [] ch)
3) write(String s)
4) print(int i)
   print(double d)
   print(char ch)
   print(Boolean b)
   print(char ch[])
5) void flush()
6) close()

Ex:

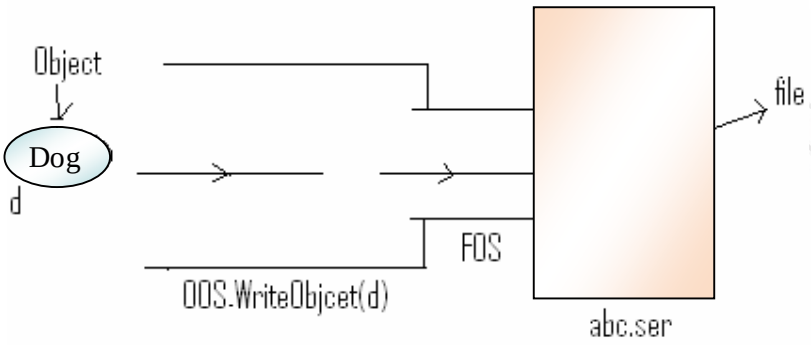
```
class test
{
    public static void main(String arg[])throws Exception
    {

        FileWriter fw = new FileWriter("pongal.txt");
```

```
                PrintWriter out = new PrintWriter(fw);
                out.write(97);
                out.println(100);
                out.println(true);
                out.println('c');
                out.println("FDGH");
                out.flush();
                out.close();
        }
}
```

Serialization Introduction

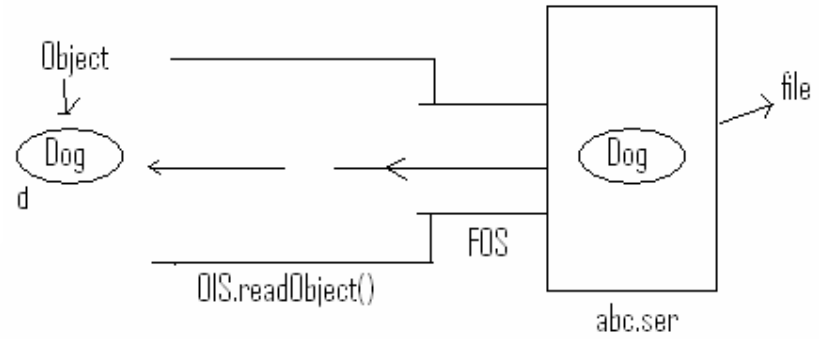
Serialization: The Process of Saving an object to a file is called “serialization”. But strictly speaking serialization is the process of converting an object from *java supported* format to *network* or *file* supported format.

By using FileOutPutStream, ObjectOutPutStream classes we can achieve serialization



Deserialization: The process of reading an object from a file is called deserialization. But strictly speaking it is the process of Converting an object from *network* supported format or *file* supported format to *java* supported format.

By using FileInputStream, ObjectInputStream we can achieve deserialization.



Ex:-

```
class Dog implements Serializable
{
        int i= 10;
        int j= 20;
}
class serializedemo
{
        public static void main(String arg[])
        {
                Dog d = new Dog();
                FileOutputStream fos = new FileOutputStream("abc.ser");
                ObjectOutputStream oos = new ObjectOutputStream(fos);
                oos.writeObject(d);
                FileInputStream fis = new FileInputStream("abc.ser");
                ObjectInputStream ois = new ObjectInputStream(fis);
                Dog d1 = (Dog)ois.readObject();
                System.out.println(d1.i+"---"+d2.j);
```

```
        }
    }
```

We can perform serialization only for serialization objects.

An Object is said to be serializable if and only if the corresponding class should implement serializable interface. serializable interface present in java.io package and doesn't contain any method, it is marker interface.

If u r trying to perform serialization of a non serialization Object. we will get runtime exception saying NonSerializableException.

If we don't want to Serialize the value of a particular variable (To meet security constraints) we should declare those variables as transient. While performing serialization JVM ignores the value of *transient* variables and saves default values instead of original values.

static variables are not part of object state and hence they won't participate in serialization process Declaring static variables as transient there is no impact similarly declaring final variables as transient creates no impact.

Declaration	**O/P**
int i = 10; int j = 20;	10…20
staic int i = 10; staic transient int j = 20;	10…20
transient int i = 10; staic transient int j = 20;	0…20
transient static int i = 10; transient int j = 20;	10…0
transient int i = 10; transient final int j = 20;	0…20

Serialization in the case of Object Graphs

When ever we are saving an object to a file all the objects which are reachable from that object will be saved by default. This group of objects is called '*Object Graph*'.
In the Object Graph if any Object is non-Serialzable we will get runtime Exception saying not- serializable Exception.

Ex:-

```
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 SerializeDemo
{
        public static void main(String arg[])throws Exception
        {
                Dog d = new Dog();
                FileOutputStream fos = new FileOutputStream("abc.ser");
                ObjectOutputStream oos = new ObjectOutputStream(fos);
                oos.writeObject(d);

                FileInputStream fis = new FileInputStream("abc.ser");
                ObjectInputStream ois = new ObjectInputStream(fis);
                Dog d1 = (Dog)ois.readObject();
                System.out.println(d1.c.r.j);
        }
}
```

O/P:- 20

In the above program among Dog, Cat and Rat classes if any class is not Serializable we will get runtime Exception saying *java.io.NotSerializableException.*

Customized Serialization

During default Serialization there may be a chance of loss of information *because of transient* variables.

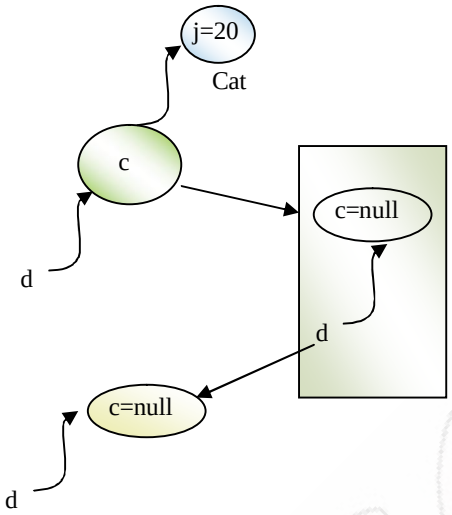
Ex:-

```
import java.io.*;
class Dog implements Serializable
{
        transient Cat c = new Cat();
}
class Cat
{
        int j= 20;
}

class SerializeDemo
{
        public static void main(String arg[])throws Exception
        {
                Dog d = new Dog();
                System.out.println("Before Serialization:"+d.c.j);
                FileOutputStream fos = new FileOutputStream("abc.ser");
                ObjectOutputStream oos = new ObjectOutputStream(fos);
                oos.writeObject(d);

                FileInputStream fis = new FileInputStream("abc.ser");
                ObjectInputStream ois = new ObjectInputStream(fis);
                Dog d1 = (Dog)ois.readObject();
                System.out.println(d1.c.j);
        }
}
```

O/P:-

```
D:\rjava\scjpExamples>java SerializeDemo
Before Serialization:20
Exception in thread "main" java.lang.NullPointerException
        at SerializeDemo.main(serialize.java:74)
```

Internal Process in Diagramatic Form



In the above program before serialization dog object tell 'j' value. But after deserialization dog object can't tell j value(d1.c.j) will rise NullPointerException.
i.e during Serialization there may be a chance of loss of information. To recover this information we should customize Serialization process. which is nothing but "*Customized Serialization*"

We can implement customized Serialization by using the following two methods.

```
private void writeObject(OutputStream os)throws Exception
{
        ….
        ….
        ….
}
```

This method will be executed automatically by the JVM at the time of Serialization.

```
private void readObject(InputStream is)throws Exception
{
        ….
        ….
        ….
}
```

This method will be executed automatically by the JVM at the time of deSerialization.

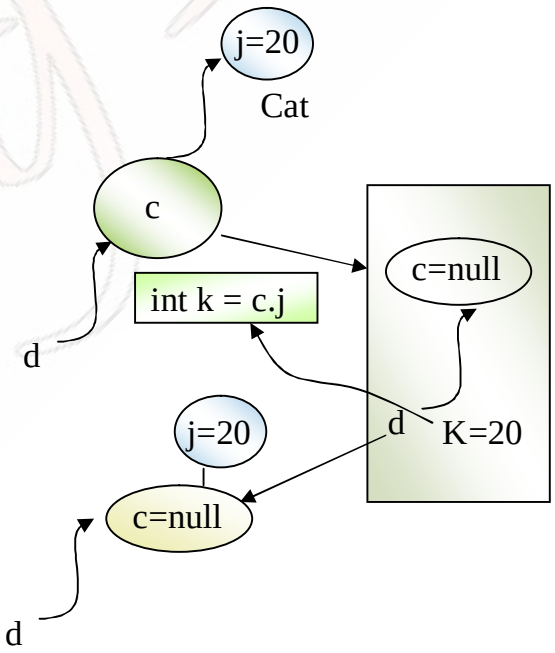
Ex:-

```
import java.io.*;
class Dog implements Serializable
{
        transient Cat c = new Cat();
        private void writeObject(ObjectOutputStream os)throws IOException
        {
                int x = c.j;
                os.writeInt(x);
        }
        private void readObject(ObjectInputStream is)throws IOException,
        ClassNotFoundException
        {
                is.defaultReadObject();
```

```
                int k = is.readInt();
                c = new Cat();
                c.j = k;
        }
}
class Cat
{
        int j= 20;
}

class SerializeDemo
{
        public static void main(String arg[])throws Exception
        {
                Dog d = new Dog();
                System.out.println("Before Serialization:"+d.c.j);
                FileOutputStream fos = new FileOutputStream("abc.ser");
                ObjectOutputStream oos = new ObjectOutputStream(fos);
                oos.writeObject(d);

                FileInputStream fis = new FileInputStream("abc.ser");
                ObjectInputStream ois = new ObjectInputStream(fis);
                Dog d1 = (Dog)ois.readObject();
                System.out.println(d1.c.j);
        }
}
```

O/P:- 20

Internal Process in Diagramatic Form:



Inheritance in Serialization

If the parent class is Serializable bydefault all the child classes also Serializable. i.e Serializable nature is inherited from *parent to child*.

Ex:-

```
import java.io.*;
class Animal implements Serializable
{
	int i = 10;
}
class Dog extends Animal
{
	int j = 20;
}

class SerializeDemo
{
	public static void main(String arg[])throws Exception
	{
		Dog d = new Dog();
		FileOutputStream fos = new FileOutputStream("abc.ser");
		ObjectOutputStream oos = new ObjectOutputStream(fos);
		oos.writeObject(d);

		FileInputStream fis = new FileInputStream("abc.ser");
		ObjectInputStream ois = new ObjectInputStream(fis);
		Dog d1 = (Dog)ois.readObject();
		System.out.println(d1.i+"-----"+d1.j);
	}
}
```

If the child class is Serializable and some of the parent classes are not Serializable, Still we are allowed to serialize child class Objects. While performing serialization JVM ignores the inherited variables which are coming from non-serializable parents.
While performing de-serialization JVM will check is there any parent class is non-serializable or not.
If any parent is non-serializable JVM will create an object for every non-serializable parent and share it's instance variables for the current child object.

The non-serializable parent class should *compulsory contain no-argument constructor* other wise we will get runtime error.

Ex:-

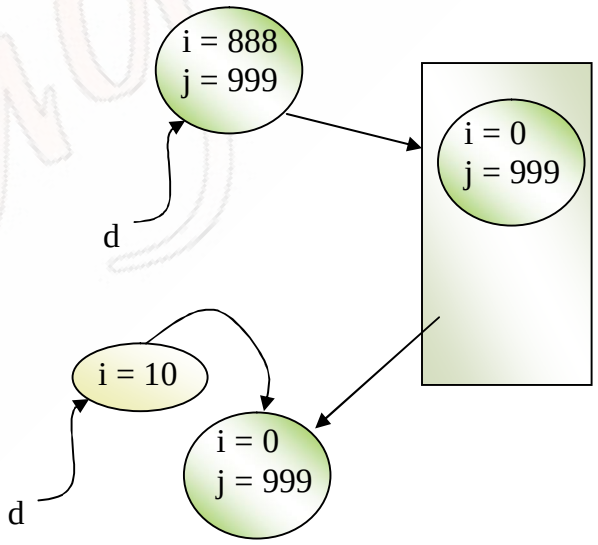
```
import java.io.*;
class Animal
{
	int i = 10;
	Animal()
	{
		System.out.println("Animal Constructor");
	}
}
class Dog extends Animal implements Serializable
{
	int j = 20;
	Dog()
```

```
        {
                System.out.println("Dog Constructor");
        }
}

class SerializeDemo
{
        public static void main(String arg[])throws Exception
        {
                Dog d = new Dog();
                d.i = 888;
                d.j = 999;
                System.out.println(d.i+"-----"+d.j);
                System.out.println("Serilization Started");
                FileOutputStream fos = new FileOutputStream("abc.ser");
                ObjectOutputStream oos = new ObjectOutputStream(fos);
                oos.writeObject(d);

                System.out.println("Deserialization Started");
                FileInputStream fis = new FileInputStream("abc.ser");
                ObjectInputStream ois = new ObjectInputStream(fis);
                Dog d1 = (Dog)ois.readObject();
                System.out.println(d1.i+"-----"+d1.j);
        }
}
```

O/P:-

```
D:\rjava\scjpExamples>java SerializeDemo
Animal Constructor
Dog Constructor
888-----999
Serilization Started
Deserialization Started
Animal Constructor
10-----999
```

Internal Process in Diagramatic Form:



13

GARBAGE COLLECTION

Introduction

In 'C++' the programmer is responsible for both creation and destruction of objects but usually the programmer is giving very much importance for creation of objects and he is ignoring the destruction of objects. Due to this at creation point of time there may not be sufficient memory for the creation of new objects and entire program may fails. But in java programmer is responsible only for creation of objects but sun people has introduced one assistance which is running continuously in the background for destruction of objects. Due to this assistance there is no chance of failing the java program due to memory problem, this assistance is nothing but "*Garbage Collector*".

The ways to make an object eligible for Garbage Collector

Even though the programmer is not responsible for destruction of objects it's good programming practice to make an object eligible for Garbage Collector if it is no longer required.

The following are different ways for this

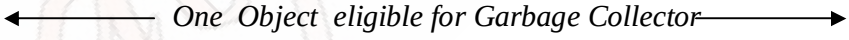
Nullifying the reference Variable

If an object is no longer required assign null to all its reference variables.

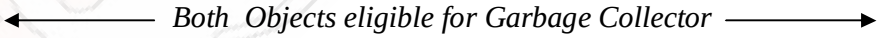
Ex:

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


```
s1 = null;
```



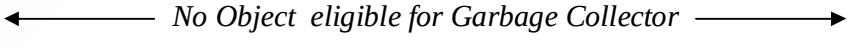

```
s2 = null;
```



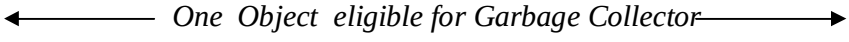
Reassigning the reference Variable

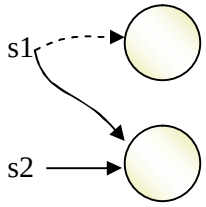
Ex:

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




```
s1 = s2;
```





The Objects Created inside a method

The objects which are created in a method are by default eligible for Garbage Collector once the method completes

Ex:

```
1)      class Test
        {
                public static void main(String arg[])
                {
                        m1();
```

Both Objects eligible for Garbage Collector

```
                }
                public static void m1()
                {
                Student s1 = new Student();
                Student s1 = new Student();
                }
        }
```

```
2)      class Test
        {
                public static void main(String arg[])
                {
                        Student s = m1();
```

One Object eligible for Garbage Collector

```
                }
                public static Student m1()
                {
                        Student s1 = new Student();
                        Student s1 = new Student();
                        return s1;
                }
        }
```

```
3)      class Test
        {
                public static void main(String arg[])
                {
                        m1();
```

Both Objects eligible for Garbage Collector

```
                }
                public static Student m1()
                {
                        Student s1 = new Student();
```

```
                Student s1 = new Student();
                return s1;
            }
        }
```

The Island of Isolation

Ex:

```
class Test
{
    Test i;
    public static void main(String[] args)
    {
        Test t1 = new Test();
        Test t2 = new Test();
        Test t3 = new Test();
        t1.i = t2;
        t2.i = t3;
        t3.i = t1;
```

← *No Object eligible for Garbage Collector* →

```
        t1 = null;
```

← *No Object eligible for Garbage Collector* →

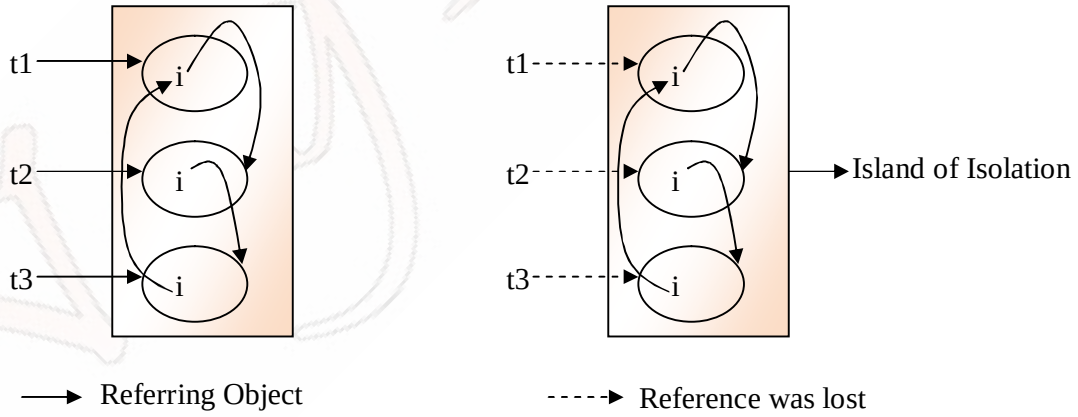
```
        t2 = null;
```

← *No Object eligible for Garbage Collector* →

```
        t3 = null;
```

← *All Object eligible for Garbage Collector* →

```
    }
}
```



Note: 1) If an object doesn't have any reference variable that object is always eligible for Garbage Collection

3) Even though object having the reference variable still there may be a chance of that object eligible for Garbage Collection (Island of Isolation …Here 'i' is internal reference)

The methods to request JVM to run Garbage Collector

We can request JVM to run Garbage Collector but there is no guarantee whether JVM accepts our request or not. We can do this by using the following ways.

By System class

'System' class contains a static 'gc' method for requesting JVM to run Garbage Collector.

```
System.gc();
```

By Using Runtime Class

A java application can communicate with JVM by using Runtime Object. We can get Runtime Object as follows.

```
Runtime r = Runtime.getRuntime();
```

Once we get Runtime Object we can apply the following methods on that object.

freeMemory(): returns the free memory available in the loop

totalMemory(): returns heap size

gc(): for requesting JVM to run GarbageCollector

Ex:

```
import java.util.*;
class RuntimeDemo
{
    public static void main(String arg[])
    {
        Runtime r = Runtime.getRuntime();
        System.out.println(r.totalMemory()); →    2031616
        System.out.println(r.freeMemory());  →    1870416
        for(int i= 0; i<= 10000; i++)
        {
            Date d = new Date();
            d = null;
        }
        System.out.println(r.freeMemory());  →    1633032
        r.gc();
        System.out.println(r.freeMemory());  →    1923992
    }
}
```

Note:- gc() method available in the system class is static method but gc() method available in Runtime class is an instance method.

Q) Which of the following is the valid way for requesting JVM to run gc?

```
System.gc();                    ✓
Runtime.gc();                   ✗    → not a static
(new Runtime()).ge();           ✗    → we can't create Object
Runtime.getRuntime().gc();      ✓
```

finalization

Just before destroying any object Garbage Collector always calls finalize() to perform clean up activities. finalize() is available in the Object class which is declared as follows.

```
protected void finalize() throws Throwable
{
}
```

case1: Garbage Collector always calls finalize() on the Object which is eligible for Garbage Collector and the corresponding class finalize method will be executed.

Ex:

```
class Test
{
    public static void main(String arg[])
    {
        String s = new String("raju");
        //Test s = new Test();
        s = null;
        System.gc();
        System.out.println("end of main method");
    }
    public void finalize()
    {
        System.out.println("finalize method called");
    }
}
```

O/P:- end of main method.

In this case String Object is eligible for G.C and hence String class finalize() method has been executed.
In the above program if we are replacing String Object with Test Object then Test class finalize() will be executed.

In this case **O/P** is

```
end of main method
finalize method called
        or
finalize method called
end of main method
```

case2: we can call finalize() explicitly in that case it will execute just like a normal method and object won't be destroyed.
While executing finalize() method if any exception is uncaught it is simply ignored by the JVM but if we are calling finalize method explicitly and if an exceptions is uncaught then the program will be terminated abnormally.

Ex:

```
class Test
{
    public static void main(String arg[])
    {
        Test s =  new Test();
        //s.finalize();
        s = null;
        System.gc();
        System.out.println("End of main method");
    }
```

```
        public void finalize()
        {
                System.out.println("finalize method");
                System.out.println(10/0);
        }
}
```

O/P:- finalize method
end of main method

Q) which of the following statements are true

1) JVM ignores all exceptions which are raised while executing finalize()
2) JVM ignores only uncaught exceptions which are raised during execution of finalize()

*case3:-*1) Garbage Collector calls finalize() only once on any object i.e it won't call more than once.
2) While executing finalize() there maybe a chance of object getting reference variable at that time G.C won't destroy that object after completing finalize()
3) If the same object is eligible for G.C second time with out executing finalize() method G.C will destroy that object.

Ex:

```
class FinalizeDemo
{
        //Test s;
        public static void main(String arg[])
        {
                FinalizeDemo f = new FinalizeDemo();
                System.out.println(f.hashcode());
                f = null;
                System.gc();
                Thread.sleep(5000);
                System.out.println(s.hashCode());
                s = null;
                System.gc();
                Thread.sleep(5000);
                System.out.println("End of main method");
        }
        public void finalize()
        {
                System.out.println("finalize method called");
                s = this;
        }
}
```

The behavior of G.C is vendor dependent hence there is no guarantee for the following

1) Whether the G.C follows mark & sweep algorithm or not
2) What exact Algorithm followed by Garbage Collector
3) In which order Garbage Collector destroys Object
4) Whether Garbage destroys all eligible objects or not
5) At what time exactly Garbage Collector will run.

14

1.5 VERSION NEW FEATURES

1. Enum
2. For-Each Loop
3. Var-Arg Methods
4. Auto Boxing & Un Boxing
5. Static Imports

Enum(enumeration)

An enum is a group of named constants.
enum can be used for defining user defined datatypes.

Ex:

```
enum Month
{
        JAN,FEB,MAR,.....DEC;
}
```

enum concept has introduced in 1.5 version. When compared with old languages enum, the java enum is more powerful because, in java enum we are allowed to take instance members, constructors ..etc which may not be possible in old languages enum.

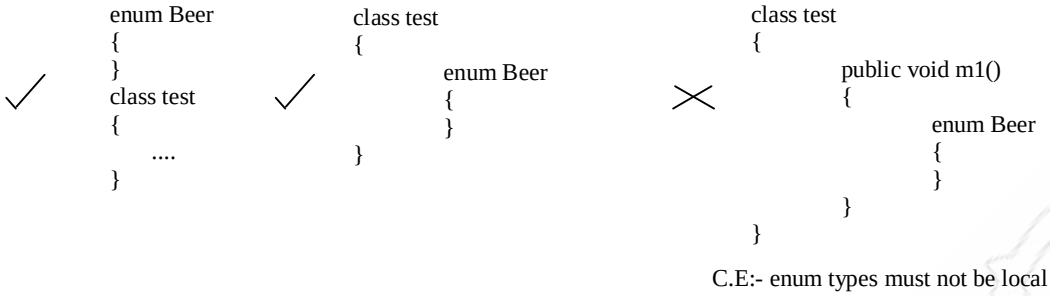
Every constant inside enum is implicitly **public static** and **final** by default. And we can access enum constants by using enum name.

Ex:

```
enum Beer
{
        ko,rc,kf,fo;
}
class test
{
        public static void main(String [] args)
        {
                Beer b1 = Beer.fo;
                System.out.println(b1);
        }
}
```

we can take enum either outside the class or with in the class but not inside method. Violation leads to C.E.

Ex:



```
enum Beer
{
}
class test
{
    ....
}
```

```
class test
{
        enum Beer
        {
        }
}
```

```
class test
{
        public void m1()
        {
                enum Beer
                {
                }
        }
}
```

C.E:- enum types must not be local

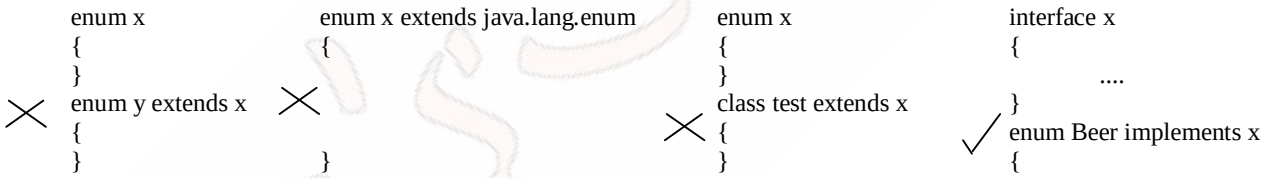
Because By Default enum constants are public static final. So in method we can't declare any static members

If we are declaring the enum outside the class the allowed modifiers are public, <default>, and strictfp.

If we are declaring enum with in a class the allowed modifiers are public, private, <default>, strictfp and static.

Enum Vs Inheritance

Every enum in java should be the direct child class of java.lang.enum.
As every enum is already extending java.lang.enum there is no chance of extending any other enum.
Hence inheritance concept is not applicable to the enum. This is the reason abstract and final modifiers are not applicable for enum.



```
enum x
{
}
enum y extends x
{
}
```

C.E:-can't inherit from final x.

```
enum x extends java.lang.enum
{
}
```

```
enum x
{
}
class test extends x
{
}
```

C.E:-can't inherit from final x.
And enum types are extensible.

```
interface x
{
        ....
}
enum Beer implements x
{
        .....
}
```

java.lang.enum is an abstract class and it is direct child class of Object.
it implements comparable and serializble interfaces. For java enums the required functionalities defined in this class only.

Enum Vs switch

We can pass enumtype as an argument to switch statement.
From 1.5 versions on words the following is the list of valid argument to the switch.

byte
short
int
char
} Upto 1.4 Version

+ Byte
Short
Integer
Character
} 1.5 Version

+ enum

Ex:

```
enum Beer
{
        kf,ko,fo;
}
class test
{
        public static void main(String arg[])
        {
                Beer b1 = Beer.fo;
                switch(b1)
                {
                        case kf : System.out.println("kf is not found");
                                  break;
                        case ko : System.out.println("ko is childrens brand");
                                  break;
                        case fo : System.out.println("Buy one get one free");
                                  break;
                        default  : System.out.println("The Other Brands are not good");
                }
        }
}
```

O/P:- Bye one get one free.

Note:-

If we are passing enum constants as the switch argument all the case labels should be valid enum constants otherwise we will get compiler time error.

values() method()

Every enum implicitly contains values method to list all it's constants.

Ex:

```
enum Months
{
        JAN,FEB,MAR,APR;
}
class test
{
        public static void main(String arg[])
        {
                Months [] m = Months.values();
                for (Months m1 : m )
                {
                        System.out.println(m1);
                }
        }
```

```
}
```

O/P:-

```
JAN
FEB
MAR
APR
```

Ordinal value()

The position of enum constant is important and it is described with ordinalvalue() we can find ordinal value of an enum constant by using following method.

Ex:

```
enum Months
{
	JAN,FEB,MAR,APR;
}
class test
{
	public static void main(String arg[])
	{
		Months [] m = Months.values();
		for (Months m1 : m )
		{
		System.out.println(m1+"....."+m1.ordinal());
		}
	}
}
```

O/P:-

```
JAN.....0
FEB.....1
MAR.....2
APR.....3
```

Note:- ordinal values are 0 based.

Speciality of java enum

A java enum can contain constructors, instance members, static members …etc in addition to constants which may not be possible in the old languages enum.

Ex:

```
enum Beer
{
	kf(100),ko(120),rc(150),fo;
	int price;
	Beer(int price)
	{
		this.price = price;
		System.out.println("constructor");
	}
	Beer()
	{
		this.price = 130;
		System.out.println("No Argumentconstructor");
	}
	public int getPrice()
	{
```

```
                return price;
        }
}
class test
{
        public static void main(String arg[])
        {
                Beer [] b = Beer.values();
                for(Beer b1 : b)
                {
                System.out.println(b1+"......"+b1.getPrice());
                }
        }
}
```

O/P:-

```
constructor
constructor
constructor
No Argumentconstructor
kf......100
ko......120
rc......150
fo......130
```

An enum can contain constructors also and these will execute at the time of enum class loading. Programmer is not responsible to call constructors explicitly.
With in enum we can't take abstract methods.

If the enum contains any thing other than constants(like instance variables, static variables.. etc) then the list of constants should end with semicolon.

Ex:

```
enum Month
{
        JAN,FEB,MAR;   <-------- Mandatory
        int num;
}
```

```
enum Month
{
        JAN,FEB,MAR;   <-------- Optional
}
```

If the enum contains anything else other than constants then the list of constants should be the first line.

Ex:

```
enum Month
{
        int num;        → C.E
        JAN,FEB,MAR;
}
```

enum in java internally implemented as class concept only. We are allowed to declare main() method in enum and we can involve directly from the command prompt.

```
enum Month
```

```
{
        JAN,FEB,MAR;
        public static void main(String arg[])
        {
                System.out.println("Hai This is enum class method");
        }
}
```

E:\> javac Month.java
java Month;

O/P:- Hai This is enum class method

Between enum constants, we can apply equality operators (== & !=) and we can't apply relational operators.
Consider the following enum declarations

```
enum Color
{
        RED,GREEN,BLUE;
}
```

1) Color.RED == Color.GREEN ✓
2) Color.RED > Color.GREEN X
3) Color.RED.ordinal() > Color.BLUE.ordinal(); ✓

Ex:

Consider the following enum declaration

```
package pack1;
enum color
{
        RED,GREEN,BLUE;
}
```

Assume the following Test class is available outside of pack1

```
class Test
{
        public static void main(String arg[])
        {
                color c = color.RED;
                System.out.println(c);
        }
}
```

To Compile Test Class Successfully, which of the following import statements we have to take.

1) import pack1.color;
2) import pack1.color.*;
3) import static pack1.color;
4) import static pack1.color.*;

```
class Test
{
    enum color
    {
```

```
        RED,GREEN;
    }
}
```

Which of the following declarations are valid outside of Test Class.

color c = RED; X

color c = color.RED; X

Test.color.c = Test.color.RED; ✓

Ex:

```
enum color
{
    RED,GREEN,BLUE
    {
        public void m1()
        {
            System.out.println("Too Good");
        }
    };
    public void m1()
    {
        System.out.println("Too Danger");
    }
}
class Test
{
    public static void main(String arg[])
    {
        color.RED.m1();
        color.BLUE.m1();
        color.GREEN.m1();
    }
}
```

O/P:- Too Danger
Too Good
Too Danger

Because we didn't put semicolon beside BLUE, and immediately we started the instance block so BLUE executes the instance block.

var- arg methods

From 1.5 version on words we are allowed to declare a method with variable no of arguments such type of methods are called var-arg methods.

We can declare a var-arg methoda as follows

```
m1(int… i);
```

this methods is applicable for any no of int arguments including zero no of arguments.

Ex:

```
class Test
{
    public static void m1(int... i)
```

```
        {
            System.out.println("var-arg methods");
        }
        public static void main(String arg[])
        {
            m1();          → var-arg methods
            m1(10);        → var-arg methods
            m1(10,20);     → var-arg methods
            m1(10,20,30);  → var-arg methods
        }
    }
```

'var-arg' methods internally implemented by using single dimensional arrays. Hence we can differentiate arguments by using index.

Ex:

```
class Test
{
    public static void main(String arg[])
    {
        sum(10,20);
        sum(10,20,30,40);
        sum(10);
        sum();
    }
    public static void sum(int... a)
    {
        int total = 0;
        for (int i=0;i<a.length ;i++ )
        {
            total = total+a[i];
        }
        System.out.println("The sum is:"+total);
    }
}
```

O/P:-

```
The sum is:30
The sum is:100
The sum is:10
The sum is:0
```

We can mix general parameter with var-arg parameter.

Ex:

m1(char ch, int…a)

If we are mixing general parameters with var-arg parameter then var-arg parameter must be the last parameter otherwise compile time error.

Ex:

m1(int… a, float f) ✗
m1(float f, int… a) ✓

we can't take more than one var-arg parameter in var-arg method otherwise C.E.

m1(int… a double… d) ✗

which of the following are valid var-arg declarations.

m1(int... a) ✓
m1(int. ..a) ✗ C.E: malformed function
m1(char ch, int... a) ✓
m1(int... a, char ch) ✗
m1(int...a, boolean... b) ✓

Ex:

```
class Test
{
	public static void m1(int i)
	{
		System.out.println("General method");
	}
	public static void m1(int... i)
	{
		System.out.println("var-arg method");
	}
	public static void m1(String arg[])
	{
		m1();
		m1(10,20);
		m1(10);

	}
}
```

Var-arg method will always get least priority i.e if no other method matched then only var-arg method will be executed.

AutoBoxing and AutoUnBoxing

Until 1.4 version, in the place of ***wrapper object*** we are not allowed to pass ***primitive*** values.

Ex:
```
ArrayList l = new ArrayList();
l.add(10); → C.E in 1.4 version.

Integer I = new Integer(10);
l.add(I); → No C.E
```

Similarly in the place of ***primitive*** we are not allowed to pass ***wrapper object***.

Ex:-
```
Boolean B = new Boolean("true");
if(B) → Generates C.E here
{
System.out.println("Hello");
}
else
{
System.out.println("Hai");
}

In Compatible types   found :Boolean
                      required:Boolean


The Following code is valid in 1.4 version.
boolean b = b.booleanValue();
if(b)
{
System.out.println("Hello");
}
else
{
System.out.println("Hai");
```

}

But from 1.5 version on words we can pass primitives in the place of objects and wrapper objects in the place of primitives. ***The required conversion will take by compiler Automatically.*** This concept is nothing but "AutoBoxing" and "AutoUnBoxing".

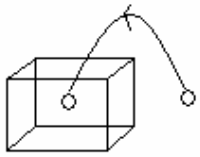
Eg:- ArrayList l = new Arraylist();
l.add(10);
Eg:- Integer I = 10;

Eg:- Boolean b = new Boolean("true");
If(b)
{
}

Because of this AutoBoxing and AutoUnBoxing concept, the importance of wrapper class is bit low from 1.5 version on words

AutoBoxing

Automatic conversion from primitive to wrapper object by the compiler is called *"AutoBoxing."*

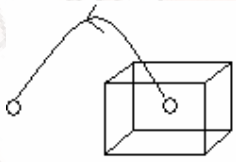


Integer I = 10;

[Compiler Automatically converts primitive to Integer Object form]

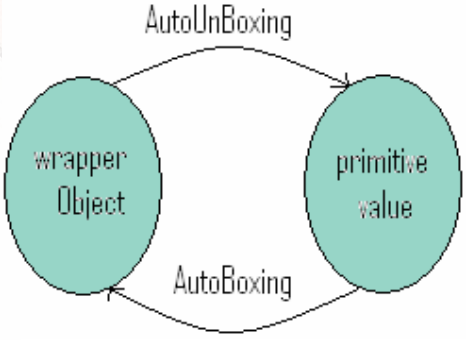
AutoUnBoxing

Automatic conversion from wrapper object to primitive type by the compiler is calles *"AutoUnBoxing"*

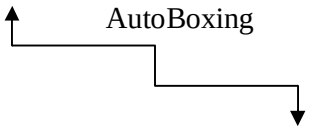


int i = new Integer(10);

[Compiler Automatically converts Integer Object to int primitive.]



Ex:- class Test
{
public static Integer I = 10;
public static void main(String[] args)
{
int i = I; →AutoUnBoxing
m1(i);




```
        }
        public static void m1(Integer I)
        {
                System.out.println(I);
        }
}
```

From 1.5 version on words there is no difference Between wrapper Object and primitive. We can use interchangeably.

Ex:

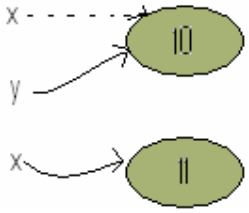
```
class Test
{
        public static Integer I = 0;
        public static void main(String[] args)
        {
        int i = I;
        System.out.println(i);

        }
}
```

Ex:

```
class Test
{
        public static Integer I;
        public static void main(String[] args)
        {
                int i = I;          → NPE(NullPointerException)
                System.out.println(i);

        }
}
```

Ex:

```
class test
{
        public static void main(String [] a)
        {
        Integer x = 10;
        Integer y = x;
        x++;
        System.out.println(x);
        System.out.println(y);
        System.out.println(x==y);

        }
}
```

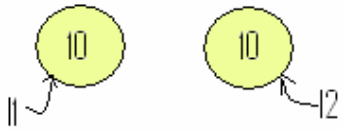


Because wrapper objects are immutable i.e once we constructed wrapper object we are not allowed to change it's content.
If u r trying to change, with those changes a new object will be created.

Case1:-

Integer I1 = new Integer(10);
Integer I2 = new Integer(10);
System.out.println(I1 == I2); → false

Because new creates new Object for every one.



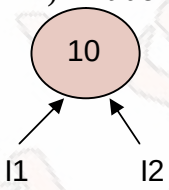
Case2:-

Integer I1 = new Integer(10);
Integer I2 = 10;
System.out.println(I1 == I2); →false

Because one is following normal and another is following AutoBoxing Concept.

Case3:-

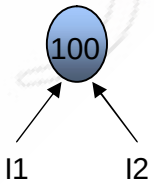
Integer I1 = 10;
Integer I2 = 10;
System.out.println(I1 == I2) →true



Because Both are following AutoBoxing Concept.

Case4:-

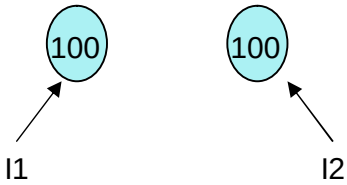
Integer I1 = 100;
Integer I2 = 100;
System.out.println(I1 == I2); →true



Because these are also following AutoBoxing concept.

Case5:-

Integer I1 = 1000;
Integer I2 = 1000;
System.out.println(I1 == I2); →false



Because for this concept the range is -127 to 128 only. So different objects will be created.

By autoboxing if an object is required to create ,compiler won't create that object immediately. First it will check .is there any object already created with the same content by autoboxing or not.
If it is already created. Compiler will gives existing object only instead of creating new object.
But this rule is applicable 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

But in all the remaining cases new object will be created.

```
Long l1 = 126l;
Long l2 = 126l;
System.out.println(l1 == l2);  →true

Boolean b1 = true;
Boolean b2 = true;
System.out.println(b1 == b2);→true

Float f1 = 10.5f;
Float f2 = 10.5f;
System.out.println(f1 == f2);  →false
```

Because it's not specified in the above list.

OverLoading in AutoBoxing, Widening, and var-args methods

Case1:-

AutoBoxing Vs Widening

```
class test {
        public void m1(Integer I)
        {
AutoBoxing      System.out.println("Integer");
        }
        public void m1(long l)
        {
                System.out.println("long");
Widening }
        public static void main(String [] a) {
                test t = new test();
                int i = 10;
                t.m1(i);
        }
```

O/P:- long

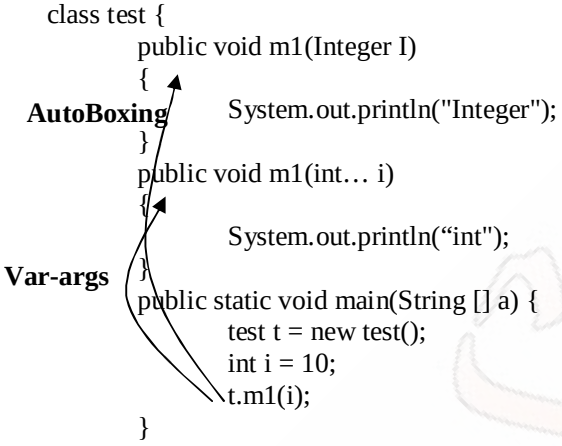
Because compiler gives the precedence for widening when compared with autoboxing.

Case2:-

AutoBoxing Vs var-args

Compiler gives the *precedence* for the autoboxing over var-arg method. i.e var-args methods will always get least priority if there is no other method. Then only var-arg method will get a chance.

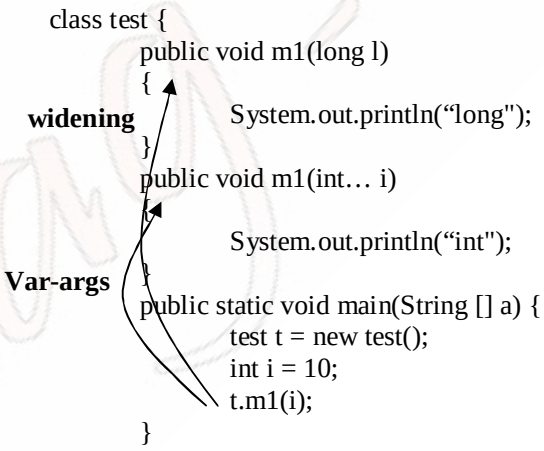
The following example will demonstrates this concept.



O/P:- Integer

Case3:-

widening Vs var-args



O/P:- long

Because the compiler will always gives chance to widening over var-arg.

Compiler gives the precedence in the following order in the case of overloading method resolution.

widening > autoboxing > var-args

Observe the following example carefully.

Ex:

class test

{

```
        public void m1(Long l)
        {
                System.out.println("Long");
        }
        public static void main(String[] args)
        {
                test t = new test();
                int i =10;
                t.m1(i);
        }
}
```

O/P:- Generates compiler error.
Because widening followed by autoboxing is not possible in java

int —widening→ **long** —autoboxing→ **Long**

Like that Integer to Long conversion is also not possible because ***Integer*** is not child class of ***Lang***

int —autoboxing→ **Integer** —widening→ **Long**

Consider the following Example:

```
class test
{
        public void m1(Object o)
        {
                System.out.println("Object");
        }
        public static void main(String[] args)
        {
                test t = new test();
                int i =10;
                t.m1(i);
        }
}
```

O/P:- Object
Because Integer is a child class of Object.

int —autoboxing→ **Integer** —widening→ **Object**

Q) Which of the following are valid declarations.

1)int i = 'a';	✓	(widening)
2)int i = new Integer(10);	✓	(AutoBoxing)
3)Long l = 10l;	✓	(AutoBoxing)
4)Long l = 10;	X	(widening followed by autoboxing)
5)Integer I = 10;	✓	(AutoBoxing)
6)Integer I = 'a';	X	char → int → Integer

static imports

This concept has introduced in 1.5 version. According to sun people static imports improves readability of the code but according to world wide java expert static imports increases confusion instead of improving readability.

Hence if there is no specific requirement it is not recommended to use static imports. The main objective of static imports is to import static members of a particular class. So that with out using class name we can access directly static members.

Ex:

With out static import

```
class Test
{
   public static void main(String arg[])
          {
            System.out.println(Math.sqrt(4));

           System.out.println(Math.random());

            System.out.println(Math.max(10,20));
          }
}
```

With static import

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

Ex:

```
import static java.lang.Integer.*;
import static java.lang.Byte.*;
class Test
{
        public static void main(String arg[])
        {
                System.out.println(MAX_VALUE);
        }
}
```

Reference to MAX_VALUE is ambiguous. Compiler always resolves static members in the following order.

4) Current class static member
5) Explicit static member
6) Implicit static member

Ex:

```
import static java.lang.Integer.MAX_VALUE;        → 2
import static java.lang.Byte.*;
class Test
{
        static int MAX_VALUE = 999;        → 1
        public static void main(String arg[])
        {
                System.out.println(MAX_VALUE);
        }
}
```

O/P:- 999

If we are commenting line 1 then the o/p is 2147483647. if we are commenting both lines 1 & 2 then the O/P is 127.

Note:- It is not recommended to extend import concept up to static members because several classes can contain static members with same name. which may cause compile time error.

Static imports also reduces readability of the code. It is recommended to access static members by using class name only.

Which of the following import statements are valid?

import java.lang.Math.*;	✗
import static java.lang.Math.*;	✓
import java.lang.Math;	✓
import static java.lang.Math;	✗

15

INTERNATIONALIZATION(I18N)

Introduction

It is the process of developing web application so that it can be used in any environment. That supports various languages and various countries, with out making any changes in the application we can achieve this by using following classes.

1) Locale:
   To represent a particular region.
2) NumberFormat:
   For formatting Numbers.
3) DateFormat:
   For formatting Dates.

Locale Class

A Locale object represents a particular region with respect to country (or) language.
It is a final class available in **java.util** package and implements Serializable and Clonable interfaces.

Construction of Locale Objects

We can construct the Locale Object by using the following Locale class constructor.

1) Locale l = new Locale(String Language);

2) Locale l = new Locale(String Language, String Country);

Locale class already defined some standard locale objects in the form of constants.

Ex: public static final Locale UK;
public static final Locale ITALY;

Important methods of Locale Class

1) public tatic Locale getDefault():
   Returns the default locale configure in JVM.
2) public static void setDefault(Locale l)
   To set our own Locale.
3) public String getCountry();
4) public String getDisplayCountry();
5) public String getLanguages();
6) public String getDisplayLanguages();
7) public static String[] getISOCountries()
   Returns ISO countries supported by the JVM.
8) public static String[] getISOLanguages();
9) public static Locale[] getAvailableLocales();

Ex

```
import java.util.*;
class LocaleDemo1
{
    public static void main(String arg[])
    {
        Locale l = Locale.getDefault();
        System.out.println(l.getCountry()+"..."+l.getLanguage());
    System.out.println(l.getDisplayCountry()+"..."+l.getDisplayLanguage());
        Locale l2 = new Locale("pa","IN");
        Locale.setDefault(l2);
        String s3[] = Locale.getISOLanguages();
        for (String s4: s3)
        {
            System.out.println(s4);
        }
        String s5[] = Locale.getISOCountries();
        for (String s6: s5)
        {
            System.out.println(s6);
        }
        Locale l3[] = Locale.getAvailableLocales();
        for (Locale l4: l3)
        {
            System.out.println(l4);

System.out.println(l4.getDisplayCountry()+"....."+l4.getDisplayLanguage());
        }
    }
}
```

NumberFormat Class

We can use this class for formatting the numbers according to a particular Locale. This class is available in **java.text** package.
Getting Number Format object for default locate
NumberFormat class contains the following factory methods to get NumberFormat Object.

Important methods of NumberFormat Class

1) public static NumberFormat getInstance();
2) public static NumberFormat getCurrencyInstance();
3) public static NumberFormat getPercentInstance();
4) public static NumberFormat getNumberInstance();

Getting NumberFormat Object for a particular locale we have to pass the corresponding locale object as the argument to the above methods.

```
public static NumberFormat getCurrencyInstance(Locale l)
```

NumberFormat class contains the following methods for converting a java number to the Locale Specific number form.

- → String format(long l)
- → String format(double d)

NumberFormat class contains the following method for converting Locale specific form to java NumberForm

Number parse(String s) throws parseException.

Consider the java number 123456.789 represents this no in ITALY, US, CHINA, Specific Forms

Ex:

```
import java.text.*;
import java.util.*;

class NumberFormatDemo
{
    public static void main(String[] args)
    {
        double d1 = 123456.789;
        Locale india = new Locale("pa","IN");
        NumberFormat nf = NumberFormat.getCurrencyInstance(india);
        System.out.println("India Notation is ....." + nf.format(d1));

        NumberFormat nf1 =
NumberFormat.getCurrencyInstance(Locale.ITALY);
        System.out.println("Italy Notation is ....." + nf1.format(d1));

        NumberFormat nf2 =
NumberFormat.getCurrencyInstance(Locale.CHINA);
        System.out.println("China Notation is ....." + nf2.format(d1));

        NumberFormat nf3 =
NumberFormat.getCurrencyInstance(Locale.US);
        System.out.println("US Notation is ....." + nf3.format(d1));
    }
}
```

O/P:-

```
India Notation is .....INR123,456.79
Italy Notation is .....Ç 123.456,79
China Notation is .....?123,456.79
US Notation is .....$123,456.79
```

Setting max/min integer and fraction digits:

NumberFormat class contains the following methods for specifying max and min fraction and integer digits.

1) public void setMaximamIntegerDigits(int n);
2) public void setMinimumIntegerDigits(int n);
3) public void setMaximamFractionDigits(int n);
4) public void setMinimumFractionDigits(int n);

NumberFormat nf = NumberFormat.getInstance();

Case1:-

nf.setMaximumIntegerDigits(4);
System.out.println(nf.format(123456.789)); → 3,456.789

Case2:-

nf.setMinimumIntegerDigits(4);
System.out.println(nf.format(12.456)); → 0012.456

Case3:-

nf.setMaximumFractionDigits(2);
System.out.println(nf.format(123456.789)); → 123,456.79

Case4:-

nf.setMinimumFractionDigits(3);
System.out.println(nf.format(123.4)); → 123.400

DateFormat Class

DateFormat class can be used for formatting the dates according to a particular locale. DateFormat class is available in java.text package. DateFormat class is an abstract class we can't create an object by using the constructor.

DateFormat df = new DateFormat(); → C.E

Creation of DateFormat Object for a default loacale

DateFormat class contains the following methods to get DateFormat Objects.

public static DateFormat getDateInstance();
public static DateFormat getDateInstance();
public static DateFormat getDateInstance(int style)

Where style is DateFormats

FULL – 0
LONG-1
MEDIUM-2
SHORT-3

Creation of DateFormat Object for a specific Locale

public static DateFormat getDateInstance(int style, Locale l)

DateFormat class contain the following method for converting java Date form to Locale specific form.

String format(Date d)

DateFormat class contain the following method for converting Locale Specific form to java Date form.

Date parse(String s)throws parseException

The program to display the current system date in all possible styles according to US Locale

Ex:

```
import java.text.*;
import java.util.*;
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()));
```

```
        }
    }
```

O/P:-

```
Full form is --Tuesday, February 24, 2009
Long form is --February 24, 2009
Medium form is --Feb 24, 2009
Short form is --2/24/09
```

The Demo program for displaying the current date according to UK,US and ITALY specific ways.

```
import java.text.*;
import java.util.*;
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()));
    }
}
```

O/P:-

```
UK Style is --Tuesday, 24 February 2009
US Style is --Tuesday, February 24, 2009
ITALY Style is --martedì 24 febbraio 2009
```

Creation of DateFormat Object for representing both Date and Time

The following are the constructors for representing both Data and Time.

```
public static DateFormat getDateTimeInstance();
public static DateFormat getDateTimeInstance(int dateStyle, int timeStyle);
public static DateFormat getDateTimeInstance(int dateStyle, int timeStyle, Locale l)
```

Ex:

```
import java.text.*;
import java.util.*;
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()));
    }
}
```

O/P:-

```
ITALY Style is --martedì 24 febbraio 2009 13.54.06 IST
```

QUICK REFERECE

Summary

Language Fundamentals
1) Identifiers
2) Keywords
3) Datatypes
4) Literals
5) Arrays
6) Types of variables
7) var – arg methods
8) command line arguments & main method
9) java coding standards

Operators
Assignment Operators: `=`
Arithmetic Operators: `-, +, *, /, %, ++, --`
Relational Operators: `>, <, >=, <=, ==, !=`
Logical Operators: `&&, ||, &, |, !, ^`
Bit wise Operator: `&, |, ^, >>, >>>`
Compound Assignment Operators: `+=, -=, *=, /=, %=`
Conditional Operator: `?:`

Declaration and Access Control
1) Java Source File Structure
2) Class Modifiers
3) Member Modifiers
4) Interface

FlowControl
Selection statements: *if, if-else and switch.*
Loop statements: *while, do-while and for.*
Transfer statements: *break, continue, return, try-catch-finally and assert.*

ExceptionHandling
Exception Hierarchy
Exception Handling By Using try ,catch
The Methods to display Exception Information
try with multiple catch blocks
finally
Possibe combinations of try, catch, finally
Control flow in try - catch – finally
throw keyword
throws
Customized Exception
Top 10 Exceptions
Assertions

OO Concepts
1) Data hiding
2) Abstraction
3) Encapsulation
4) Tightly Encapsulated class
5) Is – A Relation Ship

6) HAS – A Relation ship
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) typecasting

InnerClasses
1) Normal/Regular inner classes
2) Method local inner classes
3) Anonymous inner classes
4) static nested classes

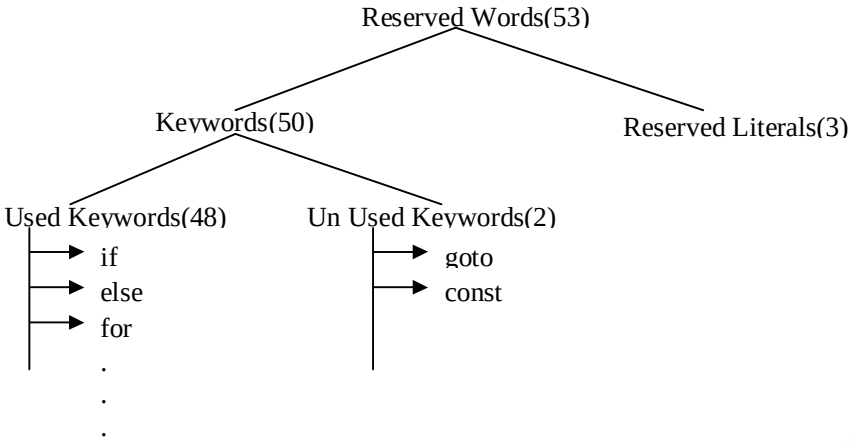
ThreadandConcurancy

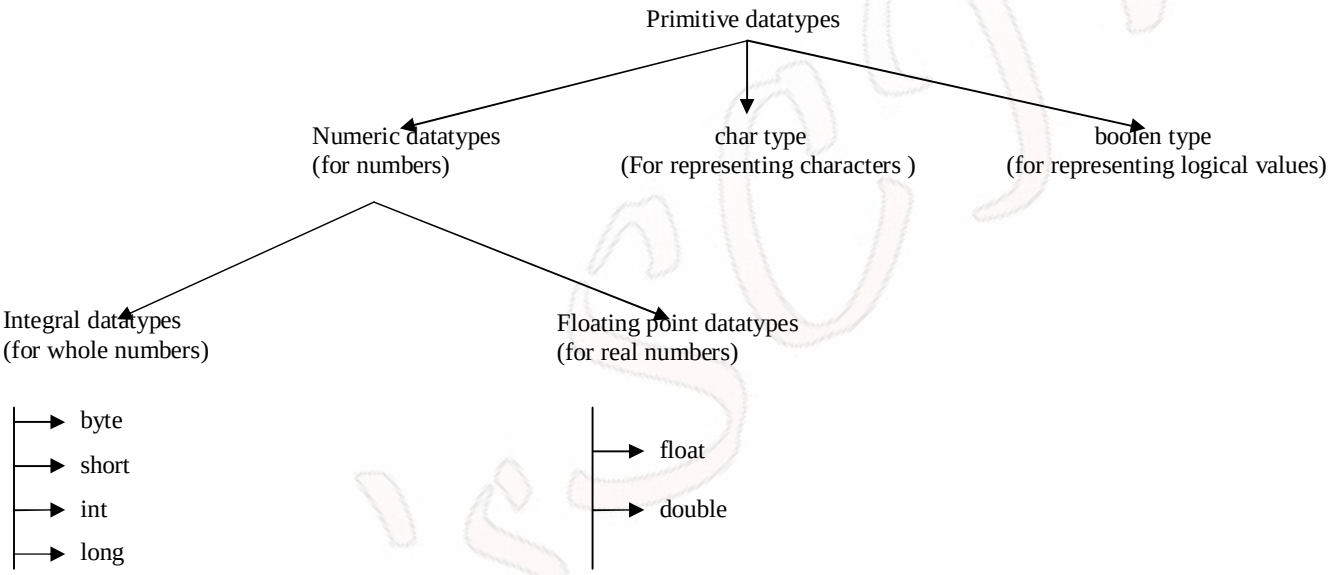
Fundamental classes in lang Package
Object Class
Strings Class
StringBuffer Class
StringBuilder
Chaining of Methods
Wrapper Classes

Collections frame work &Generics
File i/o & Serialization
1) File
2)FileWriter
3) FileReader
4) BufferedWriter
5)BufferedReader
6) printWriter
Garbage collection

1.5 version new features
1. Enum
2. For-Each Loop
3. Var-Arg Methods
4. Auto Boxing & Un Boxing
5. Static Imports

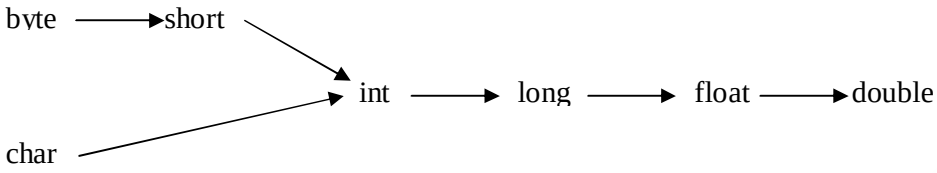
Internationalization(i18n)
1) Locale: To represent a particular region.
2) NumberFormat: For formatting Numbers.
3) DateFormat: For formatting Dates.

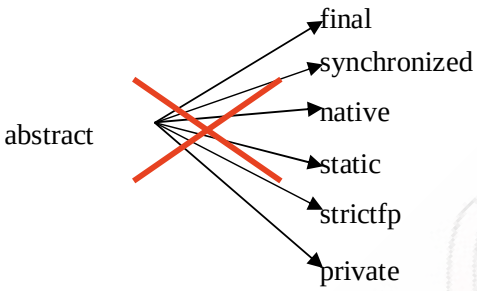




Comparison table for java primitive datatypes

datatype	size	range	Default value	Wrapper class
byte	1 byte	-128 to 127	0	Byte
short	2 bytes	-32768 to 32767	0	Short
int	4 bytes	-2^{31} to $2^{31}-1$	0	Integer
long	8 bytes	-2^{63} to $2^{63}-1$	0	Long
float	4 bytes	-3.4e38 to 3.4e38	0.0	Float
double	8 bytes	-1.7e308 to 1.7e308	0.0	Double
boolean	NA	NA(But allowed values are true, false)	false	Boolean
char	2 bytes	0 to 65535	0(balnk spaces)	Character





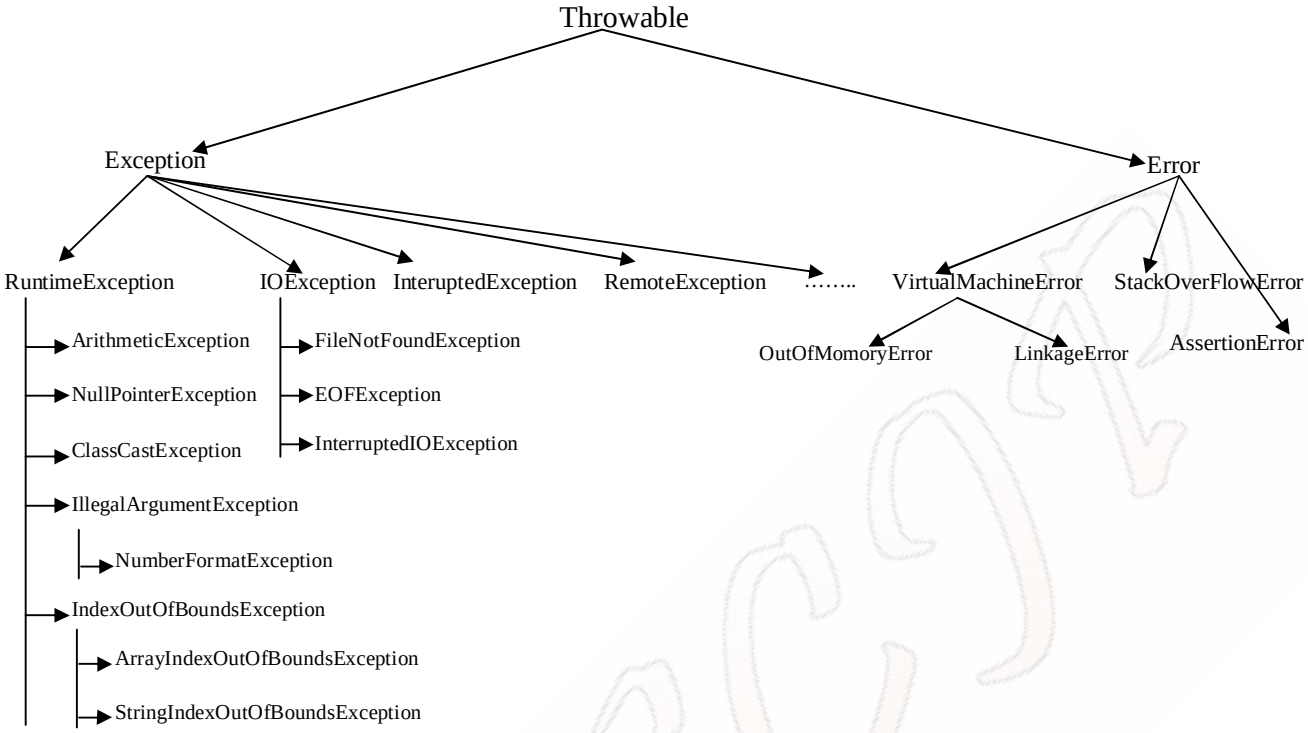
Summarization of public, protected, default and private

Visibility	public	protected	<default>	private
With in the same class	✓	✓	✓	✓
With in the same package but from out side class	✓	✓	✓	✓
Outside package but from child class	✓	✓	X	X
Outside package but from non-child class	✓	X	X	X

Access Modifiers

Modifier	Variables	Method	Top level class	Inner class	Blocks	constructor
public	✓	✓	✓	✓	X	✓
protected	✓	✓	X	✓	X	✓
<default>	✓	✓	✓	✓	X	✓
private	✓	✓	X	✓	X	✓
abstract	X	✓	✓	✓	X	X
final	✓	✓	✓	✓	X	X
strictfp	X	✓	✓	✓	X	X
static	✓	✓	X	✓	✓	X
synchronized	X	✓	X	X	✓	X
native	X	✓	X	X	X	X
transient	✓	X	X	X	X	X
volatile	✓	X	X	X	X	X

Exception Hierarchy



Top 10 Exceptions in Table Format

Exception/Error	Child of	checked/unchecked	Thrown by
1) NullPointerException	RuntimeException	Unchecked	JVM
2) StackOverFlowError	Error	Unchecked	
3) ArrayIndexOutOfBoundsException	RuntimeException	Unchecked	
4)ClassCastException	RuntimeException	Unchecked	
5) NoClassDefFoundError	Error	Unchecked	
6) ExceptionInInitializerError	Error	Unchecked	
7) IllegalArgumentException	RuntimeException	Unchecked	Programmer/ API Developer
8) NumberFormatException	IllegalArgument	Unchecked	
9) IlleaglStateExcepiton	RuntimeException	Unchecked	
10) AssertionError	Error	Unchecked	

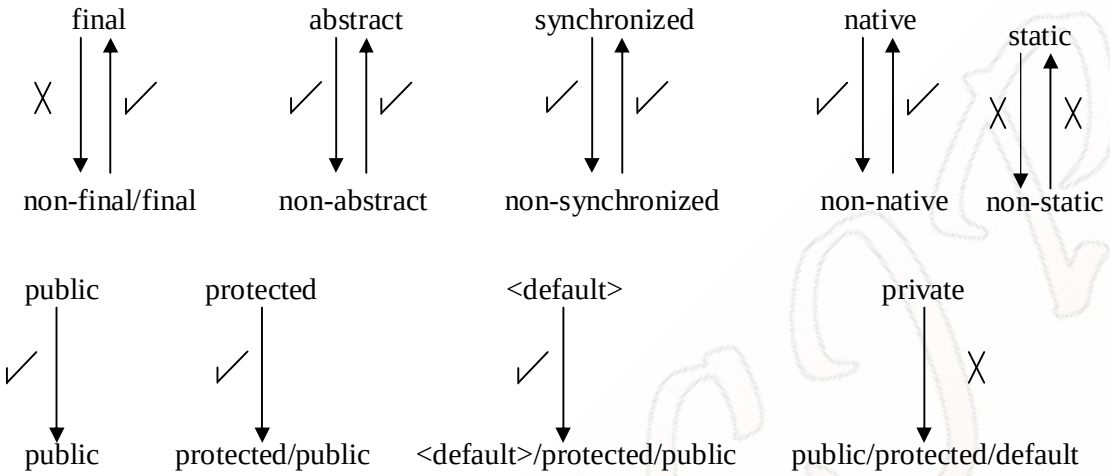
Simple assert

Syntax: assert <*boolean expression*> ;
assert(b);s

Augmented Version

Syntax: assert <*boolean expression*> : <*message expression*> ;
Assert e1:e2;

In Overriding



Comparison between Overloading and Overriding

Property	Overloading	Overriding
1) Method names	Same	Same
2) Arguments	Different(at least order)	Same(including order)
3) signature	Different	Same
4) return type	No rule or no restriction	Must be same until 1.4 version. But from 1.5 version onwards co-variant return types are allowed
5) throws clause	No restrictions	The size of the checked exception should not be increased. But no restrictions for unchecked exceptions.
6) Access Modifier	No restrictions	Should be decreased
7) Private/final/static	Can't be overloaded	Can't be overridden
8) Method resolution	Method resolution takes care by compiler based on reference type and arguments	Takes care by JVM based on runtime object
9) Other names	Compile time (or) static polymorphism (or) early binding	Runtime (or) dynamic polymorphism (or) late binding

Synchronization table for *yield(), join() & sleep()*

Property	yield()	join()	sleep()
Is it overloaded?	No	Yes	Yes
Is it static?	Yes	No	Yes
Is it final?	No	Yes	No
Is it throws InterruptedException?	No	Yes	Yes
Is it native?	Yes	No	Sleep(long ms)→native Sleep(long ms, int ms)→not native

Method	Is lock released
yield()	X
join()	X
sleep()	X
wait()	✓
notify()	✓
notifyall()	✓

Comparison between '==' operator and '.equals()'

==	.equals()
1) It is an operator and applicable for both primitive and Object references.	1) It is a method applicable for Only Object reference.
2) For Object reference r1, r2 r1 == r2 is true iff both r1 and r2 pointing to the same object on the heap i.e it is meant for reference comparision.	2) Default implementation available in Object class is meant for reference comparison only i.e r1.equals(r2) is true iff both r1,r2 are pointing to the same object on the heap.
3) We can't override == operator.	3) We can override equals() for content comparison.
4) IF r1 & r2 are incompatible then r1 == r2 results Compile Time Error.	4) If r1 & r2 are incompatible then r1.equals(r2) is always false.
5) For any Object reference r, r == null returns false.	5) For any Object reference r, r.equals(null) returns false.

Contract Between .equals() and hashCode()

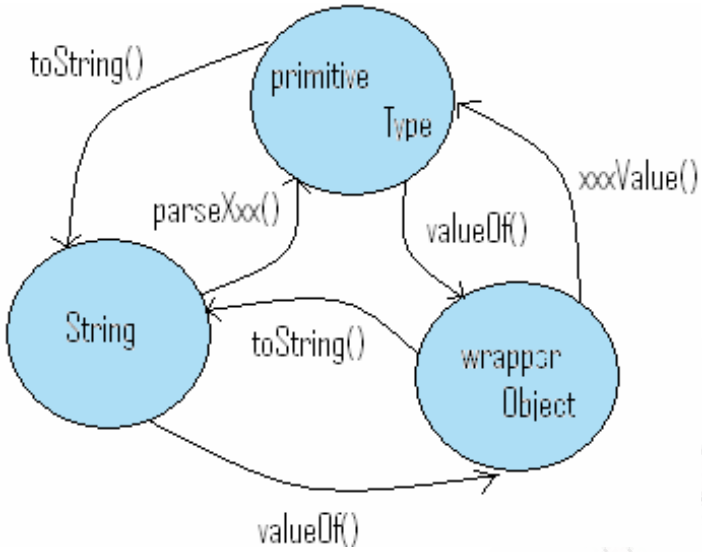
5) If two objects are equal by .equals() then their hashcodes must be equal.
   Ex:
   If r1.equals(r2) is true then
   r1.hashCode() == r2.hashCode is also true.
6) If two Objects are not equal by .equals() then their hashCode may or may not be same.
7) If the hashCodes of two Objects are equal then the objects may or may not be equal by .equals()
8) If the hashCodes of two Objects are not equal then the Objects are always not equal by .equals().

To satisfy above contract when ever we are overriding .equals it is highly recommended to override hashCode() also.

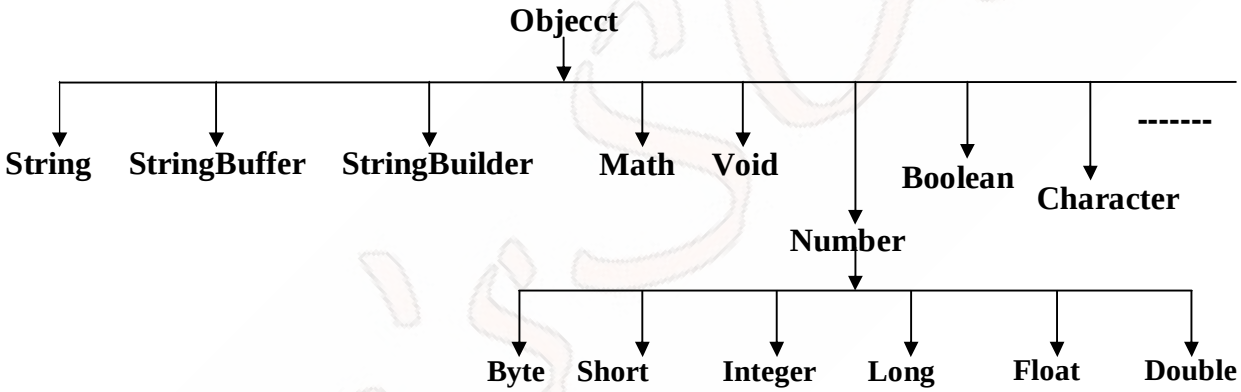
Differece Between StringBuilder and StringBuffer

StringBuilder	StringBuffer
No method is synchronized	All methods are Synchronized
StringBuilder is not thread safe	StringBuffer is thread safe
Performance is high	Performance is very low
Available only in 1.5 version	Available from 1.0 version

WrapperClasses	constructor arguments
Byte	byte (or) string
Short	short (or) string
Integer	int (or) string
Long	long (or) string
Float	float (or) string (or) double
Double	double (or) string
Character	char
Boolean	boolean (or) string



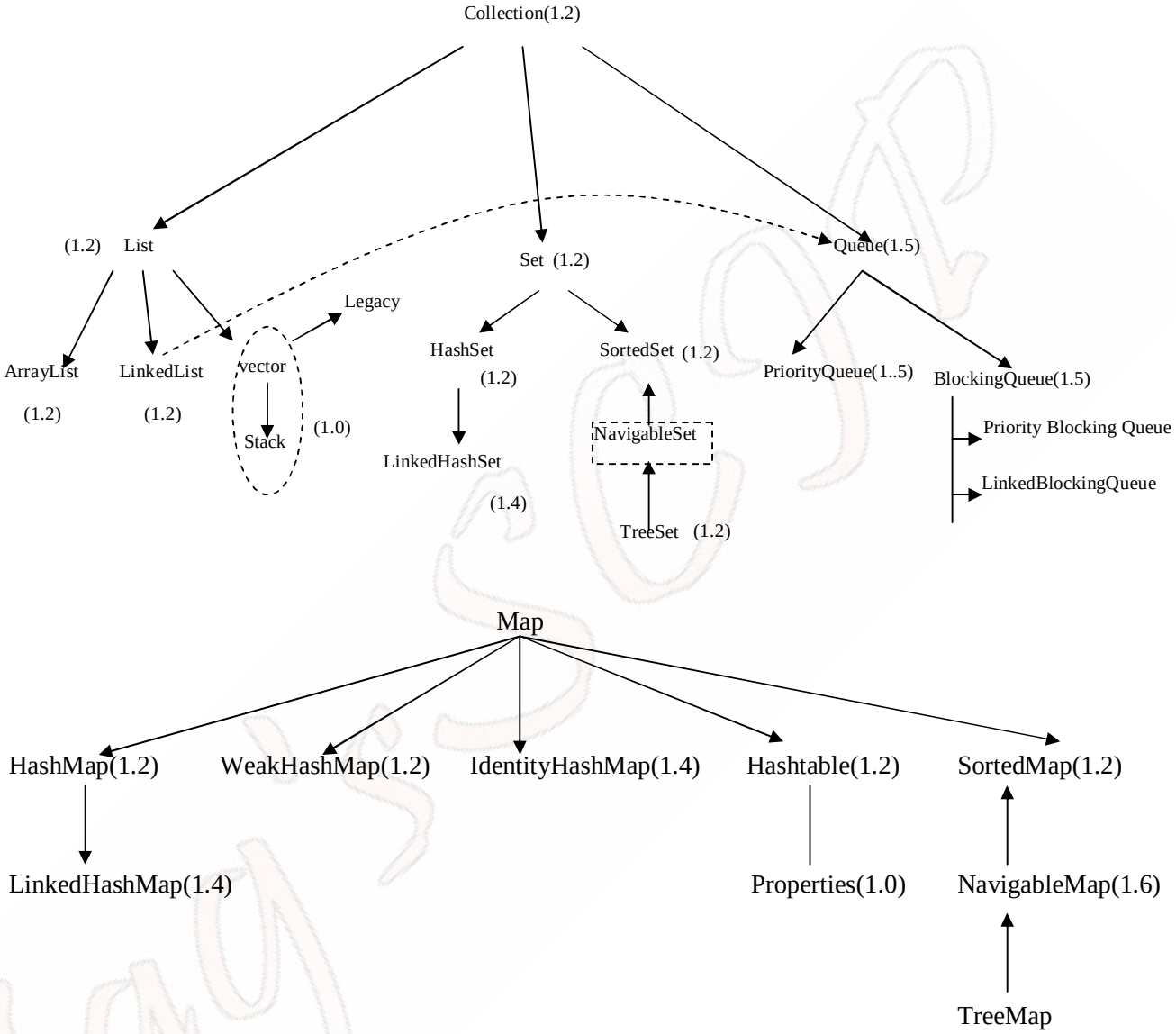
Partial Hirarchy of java.lang Package



Comparison Between collections and Arrays

Collections	Arrays
1) Collections are not fixed In size.	1) Arrays are fixed In size.
2) With respect to memory collections are good.	2) with respect to memory arrays are not good.
3) With respect to performance collections shows worst performance.	3) With respect to performance the arrays are recommended to use.
4) Collections can hold heterogeneous data elements.	4) Arrays can only hold homogeneous data elements.
5) Every Collection class has built by using some Data structure.	5) There is no underlying Data Structure.
6) Collection can hold only Objects but not primitives.	6) Arrays can hold both Objects and primitives.

Collection Frame Work



Difference between vector and ArrayList?

Vector	ArrayList
1) All methods of vector are synchronized	1) no method is synchronized
2) vector object is thread safe	2) vector object is not thread safe by default
3) performance is low	3) performance is High
4) 1.0 version(Legacy)	4) 1.2 version(non Legacy)

Comparision between All the Three cursors

Properties	Enumeration	Iterator	ListIterator
1) It is Legacy?	Yes	No	No
2) It is applicable	Only for legacy classes	For any collection	Only for list implemented classes
3) How to get?	By using elements()	By Using iterator()	By using listIterator()
4) Accessibility	Only read	read & remove	read/remove/replace/add
5) Movement	Single direction (Only Forward)	Single direction (Only Forward)	Biderctional
6) Methods	hasMoreElements() nextElements()	hasNext() next() remove()	hasNext() hasPrevious() . .
7) version	1.0	1.2	1.2

Differece between HashSet and LinkedHashSet

HashSet	LinkedHashSet
1) The underlying Data Structure is Hashtable	1) The underlying Data Structures are Hashtable and LinkedList
2) Insertion Order is not preserved	2) Insertion Order is preserved.
3) Introduced in 1.2 version	3) Introduced in 1.4 version

AutoBoxing

widening > autoboxing > var-args

Element type	Initial value
boolean	false
byte	0
short	0
int	0
long	0L
char	'u0000'
float	0.0F
double	0.0D
Object reference	null

Operator Precedence

Operators	Precedence
postfix	*expr++ expr--*
unary	*++expr --expr +expr -expr ~ !*

multiplicative	`* / %`
additive	`+ -`
shift	`<< >> >>>`
relational	`< > <= >= instanceof`
equality	`== !=`
bitwise AND	`&`
bitwise exclusive OR	`^`
bitwise inclusive OR	`|`
logical AND	`&&`
logical OR	`||`
ternary	`? :`
assignment	`= += -= *= /= %= &= ^= |= <<= >>= >>>=`

Differece Between List Interface and Set Interface

List Interface				*Set Interface*			
ArrayList	**LinkedList**	**VectorClass**	**Stack**	**HashSet**	**LinkedHashSet**	**Sorted Set**	**TreeSet**
1. The underlying data Structure for ArrayList() 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 List, Queue, serializable, clonealbe, RandomAccess Interfaces. 7. ArrayList is not recommended if the frequent operation is insertion or deletion in the middle.	1. The underlying Data Structure for linked list is doubly linked list. 2. Duplicate objects are allowed. 3. Insertion order is preserved. 4. Heterogeneous objects are allowed. 5. 'null' insertion is possible. 6. Implements List, Queue, serializable, clonealbe Interfaces But not RandomAccess.	1. The underlying Data structure for the vector is resizable array or growable array. 2. Insertion order is preserved. 3. Duplicate objects are allowed. 4. 'null' insertion is possible. 5. Heterogeneous objects are allowed. 6. Best choice if the frequent operation is retrieval. 7. Worst choice if the frequent operation is insertion or deletion in the middle. 8. Vector class implemented serializable, cloneable and RandomAccess Interfaces.	It is the child class of Vector contains only one constructor.	1. The underlying Data Structure for HashSet is Hashtable. 2. Insertion order is not preserved and it is based on has code of the Object. 3. Duplicate objects are not allowed. Violation leads to no CompileTimeError or RuntimeError, 4. add method simply returns false. 5. 'null' insertion is possible. 6. Heterogeneous objects are allowed. 7. HashSet is the best choice if the frequent operation is Search Operation.	1. The underlying Data Structure for HashSet is Hashtable, LinkedList 2. Insertion order is preserved and it is based on has code of the Object. 3. Duplicate objects are not allowed. Violation leads to no CompileTimeError or RuntimeError, 4. add method simply returns false. 5. 'null' insertion is possible. 6. Heterogeneous objects are allowed. 7. HashSet is the best choice if the frequent operation is Search Operation.	Insertion order is not preserved but all the elements are inserted according to some sorting order of elements. The sorting order may be default natural sorting order or customized sorting order.	1. The underlying Data structure for the TreeSet is Balanced tree. 2. Duplicate objects are not allowed. If we are trying to insert duplicate object we won't get any 3. compile time error or Run time error, add method simply returns false. 4. Insertion order is not preserved but all the elements are inserted according to some sorting 5. order. 6. Heterogeneous objects are not allowed, violation leads to Run time error saying class cast 7. Exception