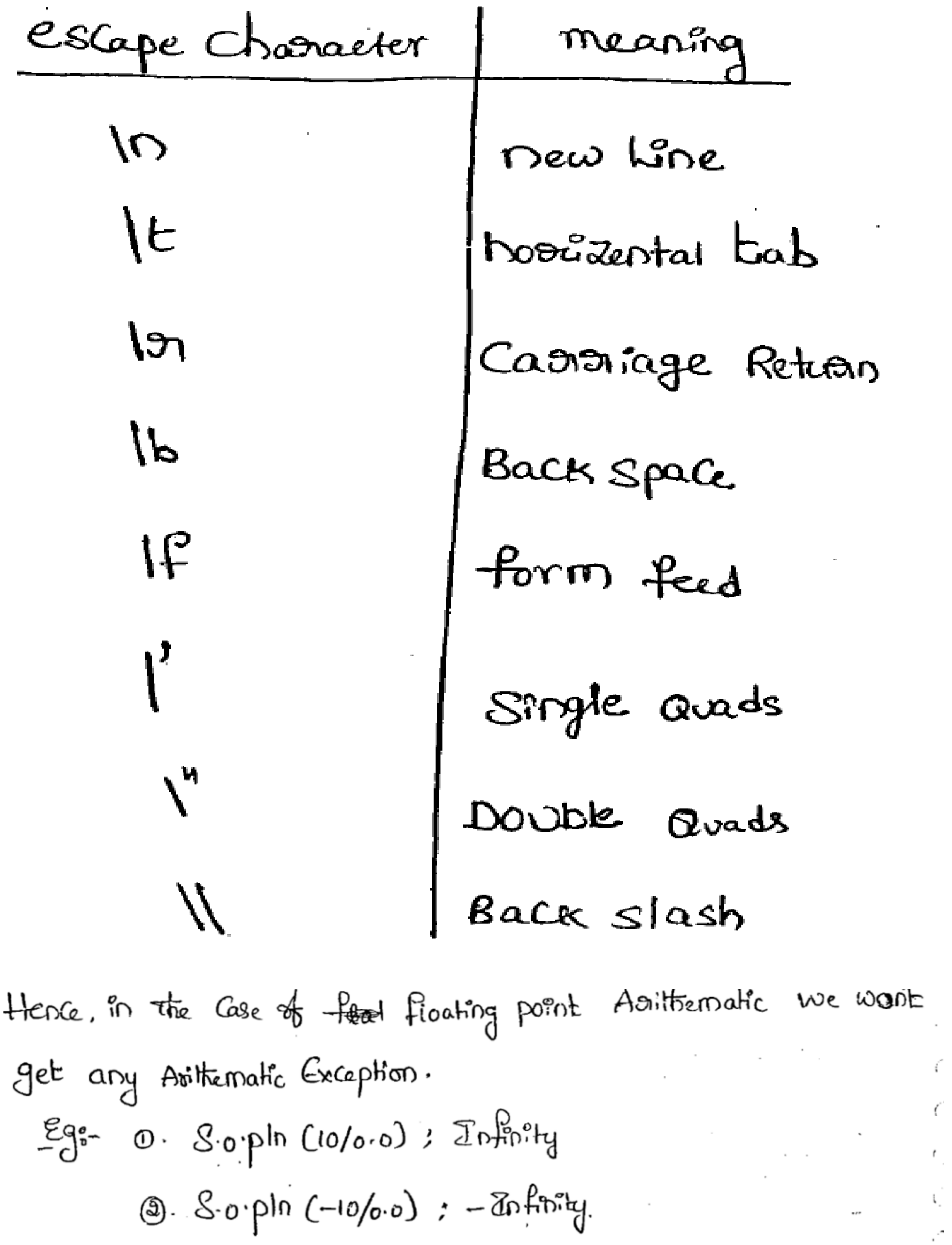
Java7_Notes:

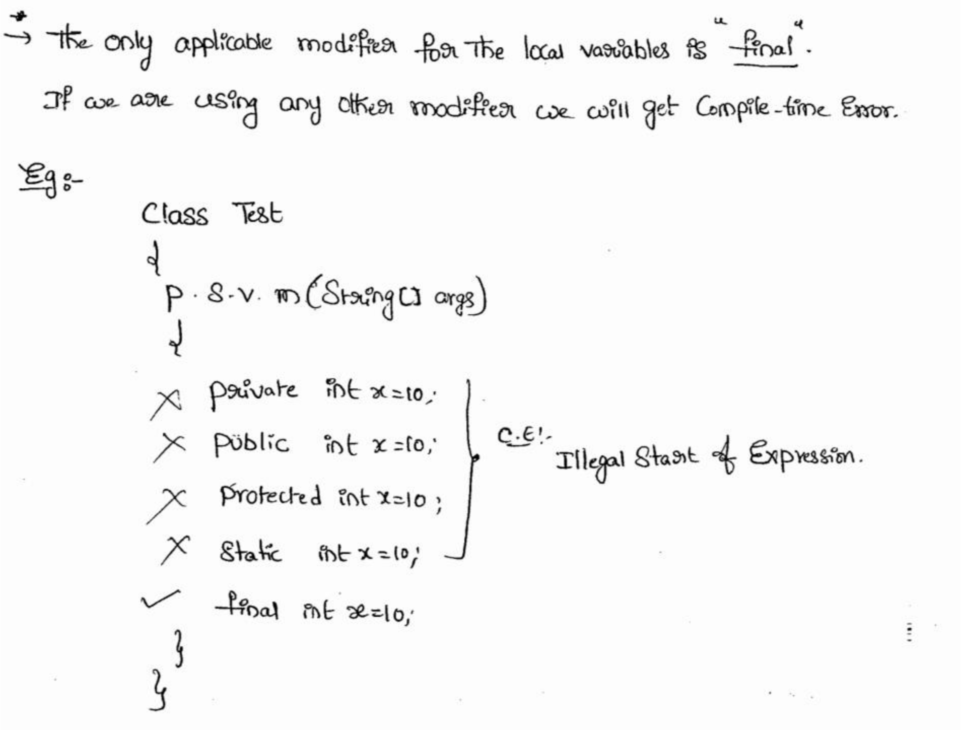


escape character	meaning
\n	new Line
\t	horizental tab
\r	Carriage Retuan
\b	Back Space
\f	form feed
\'	Single Quads
\"	Double Quads
\\	Back slash

Hence, in the Case of ~~feal~~ floating point Arithematic we wont get any Arithematic Exception.

Eg:- ①. S.o.pln (10/0.0) ; Infinity

②. S.o.pln (-10/0.0) ; -Infinity.

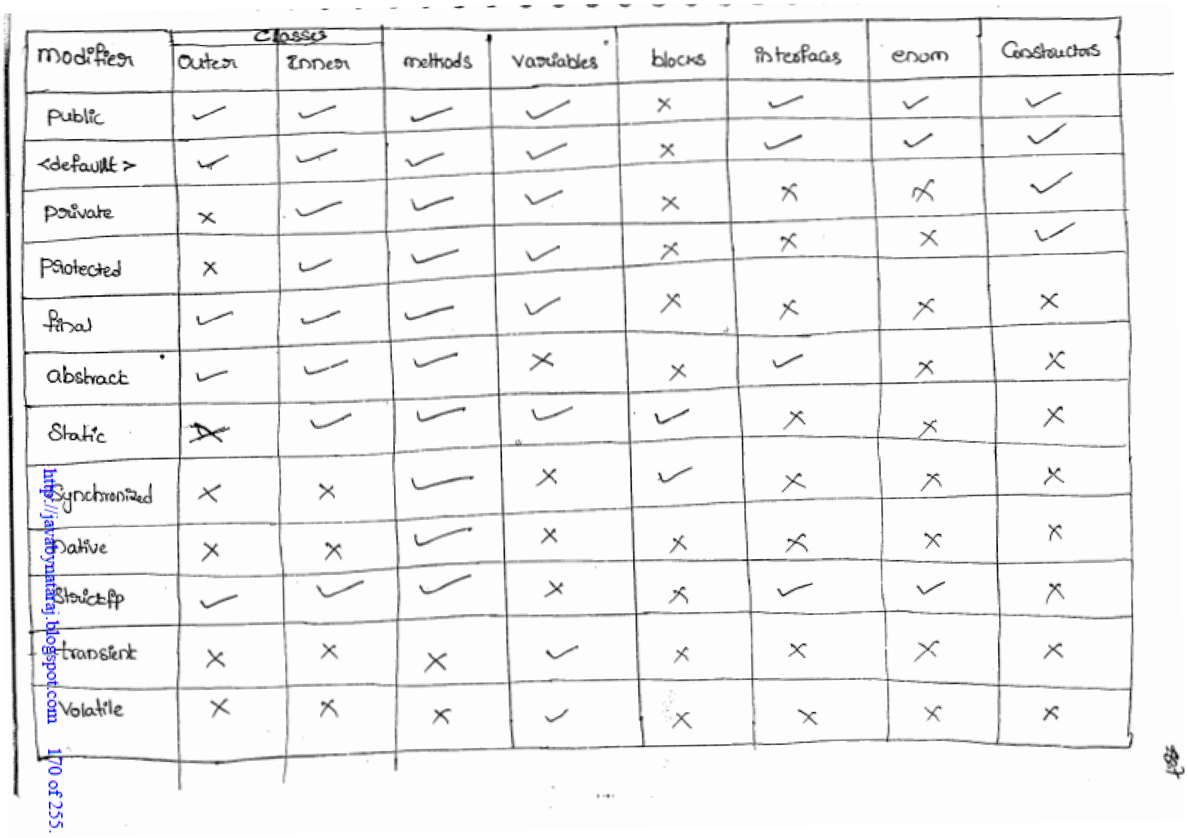


→* The only applicable modifier for the local variables is "final".

If we are using any other modifier we will get Compile-time Error.

Eg:-

```
Class Test
{
 p.s.v.m(String[] args)
 {
  ✗ private int x=10;   }
  ✗ public int x=10;    }  C.E:- Illegal Start of Expression.
  ✗ protected int x=10; }
  ✗ Static int x=10;    }
  ✓ final int x=10;
 }
}
```



Modifier	Classes		methods	variables	blocks	interfaces	enum	Constructors
	Outer	Inner						
Public	✓	✓	✓	✓	×	✓	✓	✓
<default>	✓	✓	✓	✓	×	✓	✓	✓
private	×	✓	✓	✓	×	×	×	✓
Protected	×	✓	✓	✓	×	×	×	✓
final	✓	✓	✓	✓	×	×	×	×
abstract	✓	✓	✓	×	×	✓	×	×
Static	×	✓	✓	✓	✓	×	×	×
Synchronized	×	×	✓	×	✓	×	×	×
native	×	×	✓	×	×	×	×	×
Strictfp	✓	✓	✓	×	×	✓	✓	×
transient	×	×	×	✓	×	×	×	×
Volatile	×	×	×	✓	×	×	×	×

Collection terminology.

List: Insertion odder present and duplicate are allowed

- ArrayList : Resizable Array and default capacity 10 [new capacity= current capacity * (3/2)+1] / Frequent operation is retrieval
- Vector : Resizable Array and all methods are synchronized / Frequent operation is retrieval/ default capacity 10 [new capacity = current capacity * 2]
- LinkedList: Duble Linked list and Frequent operation insertion or deletion
- Stack :

Set : No insertion order and duplicates are not allowed

- HashSet : Underlying DS hashtable and default capacity 16 [default fill ratio 0.75 (75%)] /
- LinkedHashSet: Child class of hashset and underlying DS is HashTable and Linked List
- Sortedset :
- TreeSet: Underlying DS is Balanced Tree and

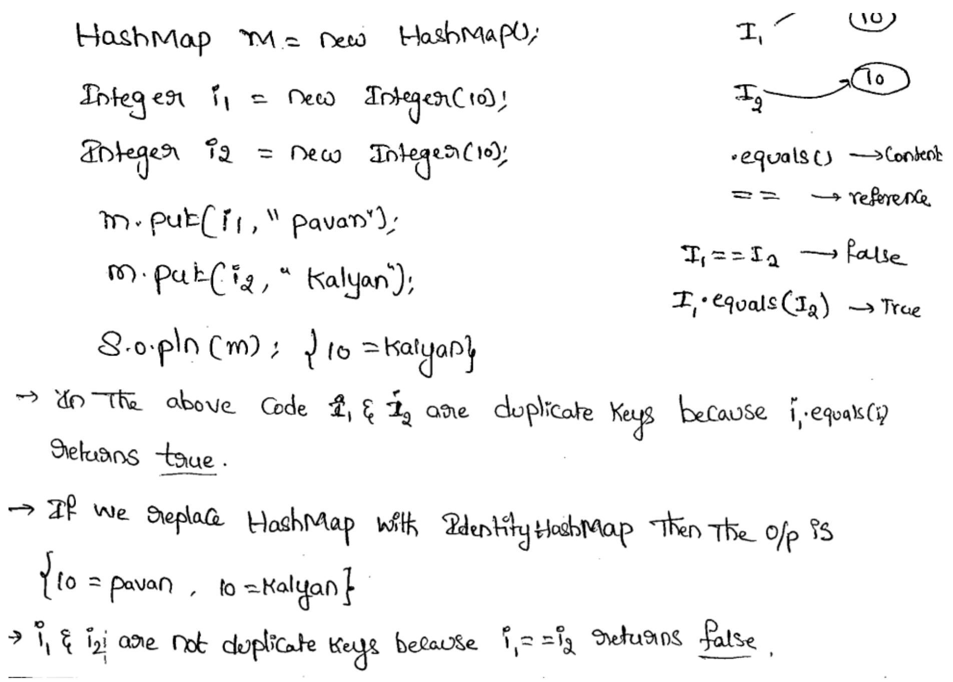
Map:

- HashMap : Underlygin DS HashTable and default capacity 16 [default fill ratio 0.75 (75%)]
- HashTable : All methods as synchonized and default capacity 11 [default fill ratio 0.7(70%)]
-
- LinkedHashMap : Child clas of hashMap and underlying DS is HashTable and Linked List

→ In the Case of HashMap to identify duplicate Keys JVM always uses .equals(), which is mostly ment for Content Comparision.

→ If we want to use == operator instead of .equals() to identify duplicate Keys we have to use IdentityHashMap. (== operator always ment for reference Comparision).

- IdentityMap :



```
HashMap m = new HashMap();
Integer i1 = new Integer(10);
Integer i2 = new Integer(10);
 m.put(i1, "pavan");
 m.put(i2, "Kalyan");
 S.o.pln(m); {10=Kalyan}
```

.equals() → Content

== → reference

$I_1 == I_2$ → false

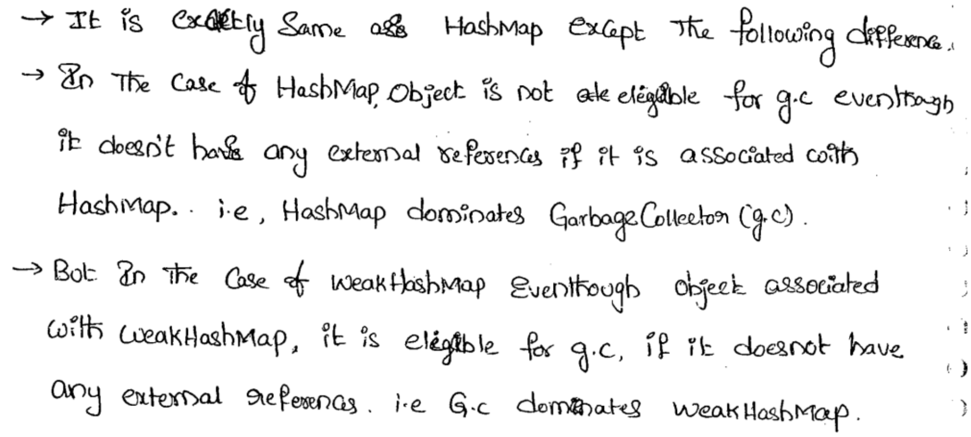
I_1.equals(I_2) → True

→ In The above Code I_1 & I_2 are duplicate Keys because i_1.equals(i_2) returns true.

→ If we replace HashMap with IdentityHashMap Then The o/p is {10 = pavan, 10 = Kalyan}

→ i_1 & i_2 are not duplicate Keys because $i_1 == i_2$ returns false.

- WeakHashMap:



→ It is exactly Same as HashMap except The following difference.

→ In The Case of HashMap, Object is not ~~ale~~ elegible for g.c eventhough it doesn't have any external references if it is associated with HashMap. i.e, HashMap dominates Garbage Collector (g.c).

→ But In The Case of WeakHashMap Eventhough object associated with WeakHashMap, it is elegible for g.c, if it doesnot have any external references. i.e G.c dominates WeakHashMap.

- SortedMap:
- TreeMap: underlying DS RED-Black Tree

Queue: where objects are inserted into one end of the queue, and taken off the queue in the other end of the queue

Sorted* : Some sorting order

Navigable* : added methods for Navigation

Linked* : Insertion order preserved

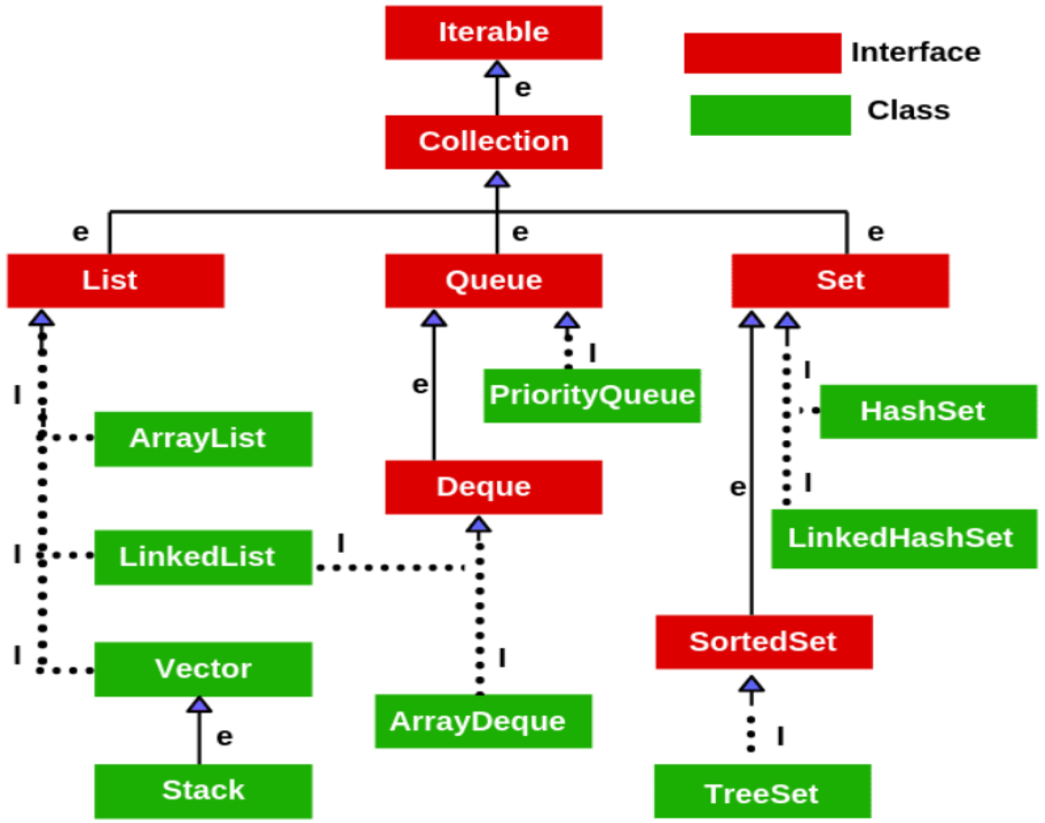
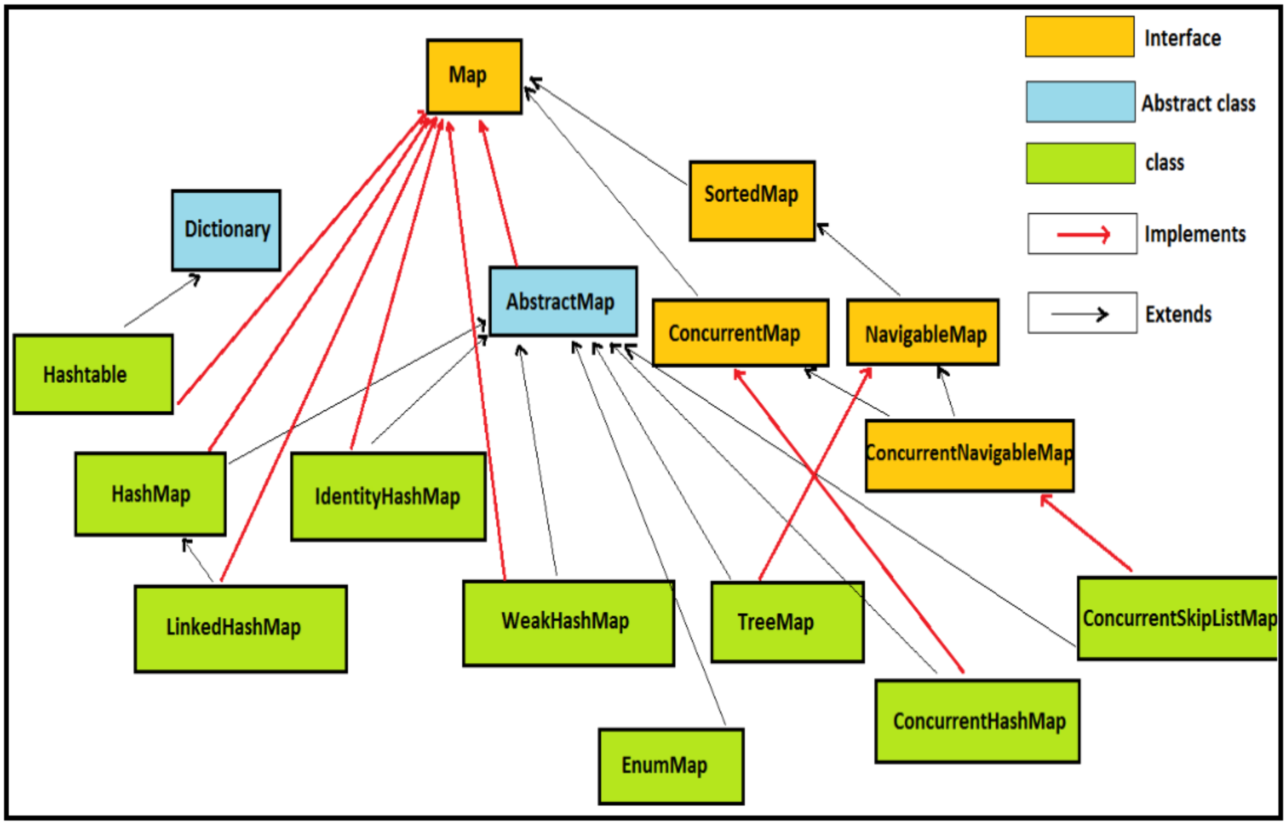
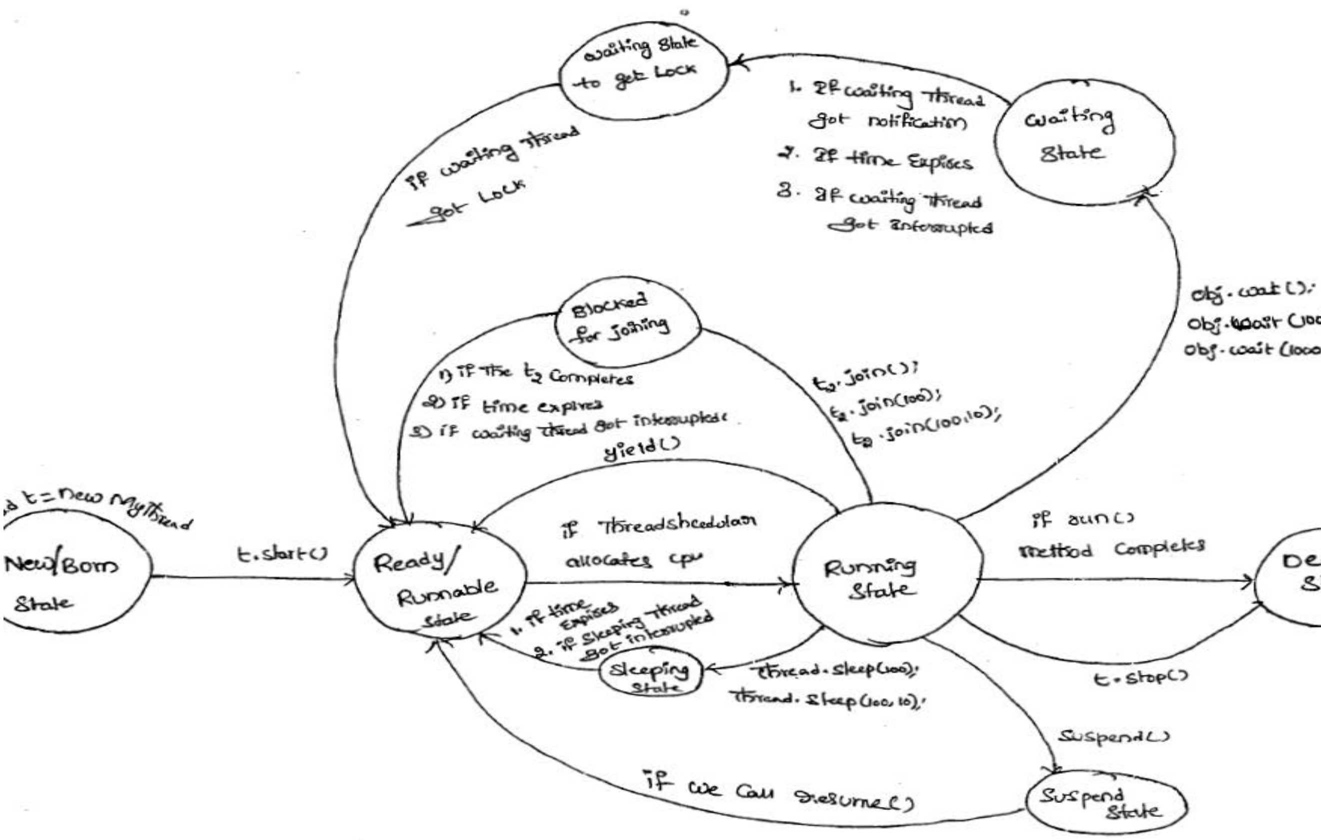


Fig: Collection Hierarchy in Java


Map
Interface
Abstract class
class
Implements
Extends
SortedMap
Dictionary
AbstractMap
ConcurrentMap
NavigableMap
Hashtable
ConcurrentNavigableMap
HashMap
IdentityHashMap
LinkedHashMap
WeakHashMap
TreeMap
ConcurrentSkipListMap
ConcurrentHashMap
EnumMap


waiting State to get Lock
1. If waiting Thread got notification
2. If time Expires
3. If waiting Thread got Interrupted
Waiting State
If waiting Thread got Lock
obj.wait();
obj.wait(100
obj.wait(1000
Blocked for Joining
1) If The t_2 Completes
2) If time expires
3) if waiting thread got interrupted
t_2.join();
t_2.join(100);
t_2.join(100,10);
yield()
d t=new MyThread
New/Born State
t.start()
Ready/ Runnable State
if Threadsheedular allocates cpu
Running State
if run() method Completes
De S
1. If time Expires
2. if Sleeping Thread got interrupted
Sleeping State
Thread.sleep(100);
Thread.sleep(100,10);
t.stop()
Suspend()
if we Call resume()
Suspend State

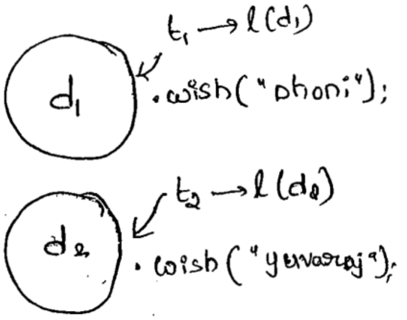
* Comparision table for yield(), join(), Sleep() :-

Property	yield()	join()	Sleep()
1) Purpose ?	to pause Current executeing thread to give the chance for the remaining threads of Same priority.	if a Thread want to wait until Completing Some other thread then we should go for join	If a Thread don't want to perform any operation for a perticular amount of time (pasusing) go for sleep()
2) Static	yes	NO	yes
3) IS it over-loaded	NO	yes	yes
4) IS it final	No	yes	No
5) IS it throws Interrupted Exception	No	yes	yes
6) IS it native method	yes	No	Sleep(long ms) ↳ native Sleep(long ms, int ns) ↳ non-native

method	is Thread releases lock ?
yield()	No
join()	No
Sleep()	No
wait()	yes
notify()	yes
notifyAll()	yes

Case Study:-

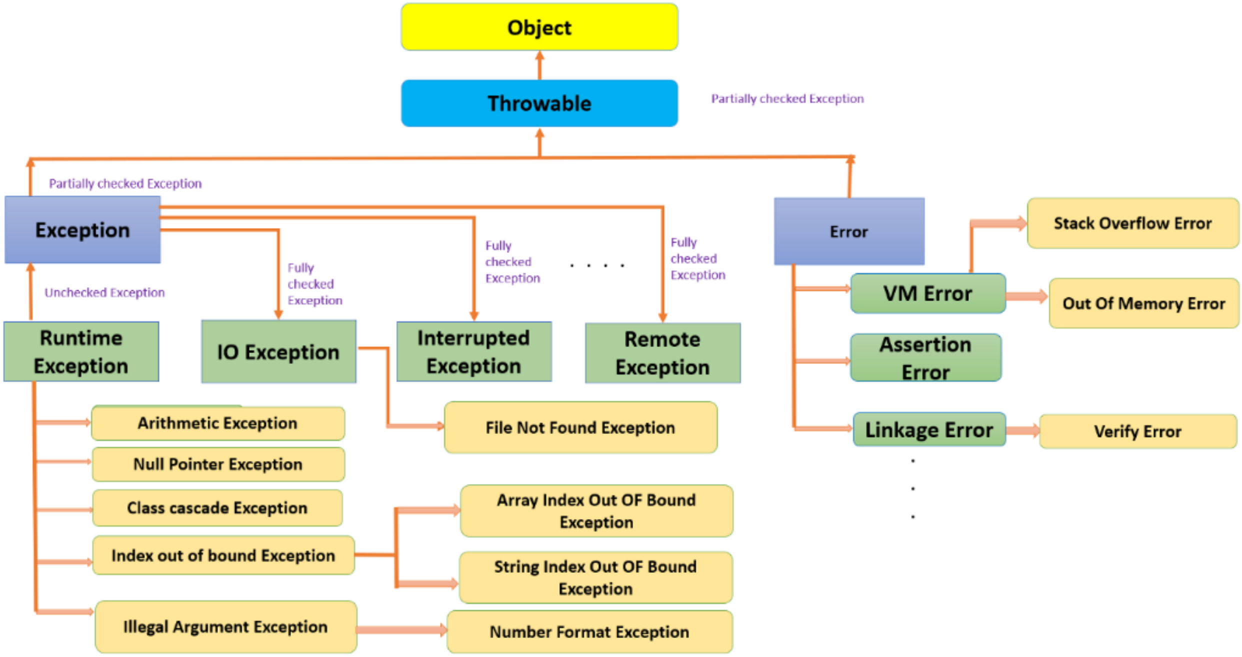
```
Display d1 = new Display();
Display d2 = new Display();
MyThread t1 = new MyThread(d1, "Dhoni");
MyThread t2 = new MyThread(d2, "Yuvaraj");
t1.start();
t2.start();
```



→ Eventhough wish() method is Synchronized we will get irregular o/p in this case. Because, the Threads are operating ~~on~~ different Objects.

Reason:-

→ Whenever multiple threads are operating on same Object then only Synchronization play the role. If multiple Threads are operating on multiple Objects then there is no impact of Synchronization.



return vs finally:-

→ finally block dominates return statement also. Hence, if there is any return statement present inside Try or Catch block, first finally will be Executed & then return statement will be Considered.

*difference b/w final, finally & finalize :-

final :-

→ It is a modifier applicable for Classes, methodes & variables.

→ If a Class declared as final, then child class Creation is not possible.

→ If a method declared as final, then overridding of that method is not possible.

→ If a variable declared as the final, then reassignment (changing the value) is not allowed because, it is a Constant.

finally :-

→ It is block always associated with try-catch to maintain Clean-up Code which should be Executed always irrespective of wheather exception raised or not raised & wheather handleded or not handeled.

finalize() :-

→ It is a method which should be Executed by Grabage Collector before destroying any object to perform Clean-up activities.

Note:-

→ if we are giving apportuenity to Object Class toString() method than it will call internally hashCode() method.

→ if we are giving apporchunity to our class toString() method than it may not Call hashCode() method.

Contract b/w .equals() & hashCode() :-

1. If two Objects are equal by .equals() Compulsary thire hashCodes must be Same.

2. If two objects are not equal by .equals() then there are no restructions on hashCode(), they Can be Same or different.

3. If hashCodes of 2 objects are equal, then we Can't Conclude above .equals(), It may returns True or false.

4. If hashCodes of 2 objects are not equals then we Can always Conclude .equals() returns false.

Conclusion :-

→ To Satisfy the above Contract b/w .equals() and hashCode(), whenever we are overriding .equals() Compulsary we should override hashCode().

→ If we are not overriding we won't get any Compile time & run-time errors.

→ But it is not a good program practice.

Q) Consider the following .equals()

```
public boolean equals(Object obj)
{
  if ( ! (obj instanceof person))
  {
     return false;
  }
  person p = (person) Obj;
  if (name . equals (p.name) && (age = = p.age))
       return true;
  else   return false;
}
```

1) Which of the following hashcode() are Said to be properly implemented.

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

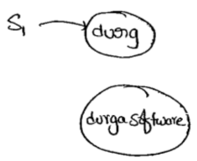
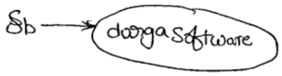
```
✗ ②  public int hashCode()
       {
         return age + (int)height;
       }
✓ ③  public int hashCode()
       {
         return name.hashCode() + age;
       }
✗ ④  public int hashCode()
       {
         return (int)height;
       }
  ⑤  public int hashCode()
       {
         return age + name.length();
       }
```

Note:-

To maintain a Contract b/w .equals() and hashCode(), what ever the parameters we are using while over riding .equals() we have to use the Same parameters while Overriding hashCode() also.

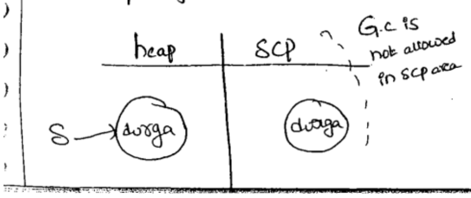
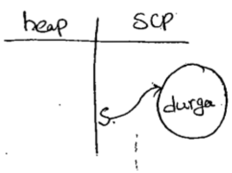
Immutable	mutable

String s = new String("durga");
s.concate("software");
s.o.p(s); durga
	SB s = new SB("durga");
	s.append("software");
	s.o.pln(s); // durgasoftware

	
→ Once we created a String Object we can't perform any changes in the Existing object. if we are trying to Perform any changes with those Changes a new object will be Created this behaviour is nothing but, "immutability of String object"	→ Once we Created a StringBuffer Object we Can perform any changes in the existing object. This behaviour is nothing but "mutability of String-Buffer object".

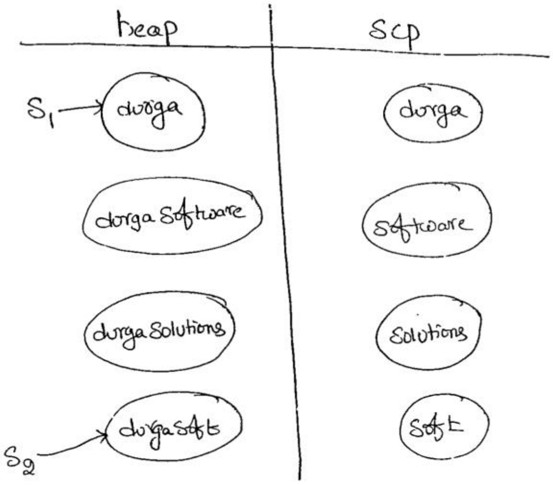
Case (3) :-

* What is the difference b/w following?

String s = new String("durga");	String s = "durga";
→ In this Case two objects will be Created one is in heap, & the other is in SCP. and 's' is always pointing to heap object	→ In this Case only one Object will be Created in SCP and 's' is always Pointing to that Object
 heap SCP G.c is not allowed in SCP area s durga durga	

Ex③ :-

```
String  s1 = new String ("durga");
        s1. concate ("software");
        s1. concate ("solutions");
String  s2 = new s1. concate ("soft");
```



→ we Can resolve this problem by Creating only one object & Share the Same object with all required references.

→ This approach improves memory utilization & Performance. we Can acheive this by using String Constant pool.

→ In SCP, a Single object will be shared for all required References. Hence the main advantages of SCP are memory-utilization & performance will be improved.

→ But the problem in this approach is, As Several references pointing to the Same object by using one reference, if we are perform any change all remaining reference will be impacted.

→ To resolve these SUN people declare String objects as immutable

→ According to that once we Created a String object we can't perform any change in the existing object. if we are trying to perform any change with

So, that there is no effect on remaining references

→ Hence, "the main disadvantage of SCP is we should Compulsory maintain String objects as immutable".

(8)

StringBuffer	StringBuilder
① Every method is Synchronized	① No method is Synchronized.
② SB object is Thread Safe. Because SB object Can be accessed by only one Thread at time.	② StringBuilder is not Thread Safe Because it Can be accessed by Multiple-Threads Simultaneously.
③ relatively performance is Low	③ relatively performance is High.
④ Introduced in 1.0 Version	⑤ Introduced in 1.5 Version

String Vs StringBuffer Vs StringBuilder :-

→ If the Content ~~only~~ will not change frequently Then we should go for String

→ If Content will change frequently & ThreadSafety is required. then we should go for StringBuffer.

→ If Content will change frequently & ThreadSafety is not required. Then we should go for StringBuilder.

final vs immutable :- 300

→ If a reference variable declareded as the final Then we Can't reasign That reference variable to Some other object.

Ex:-

```
final StringBuffer sb = new StringBuffer("durga");
   sb = new StringBuffer("Software");
```

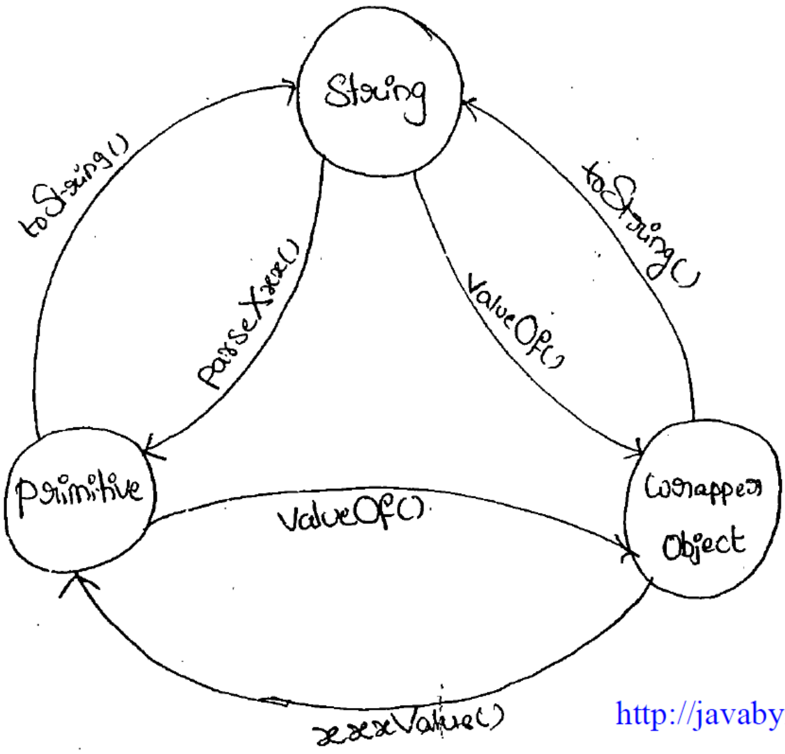
C.E:- Can't assign a value to final variable sb.

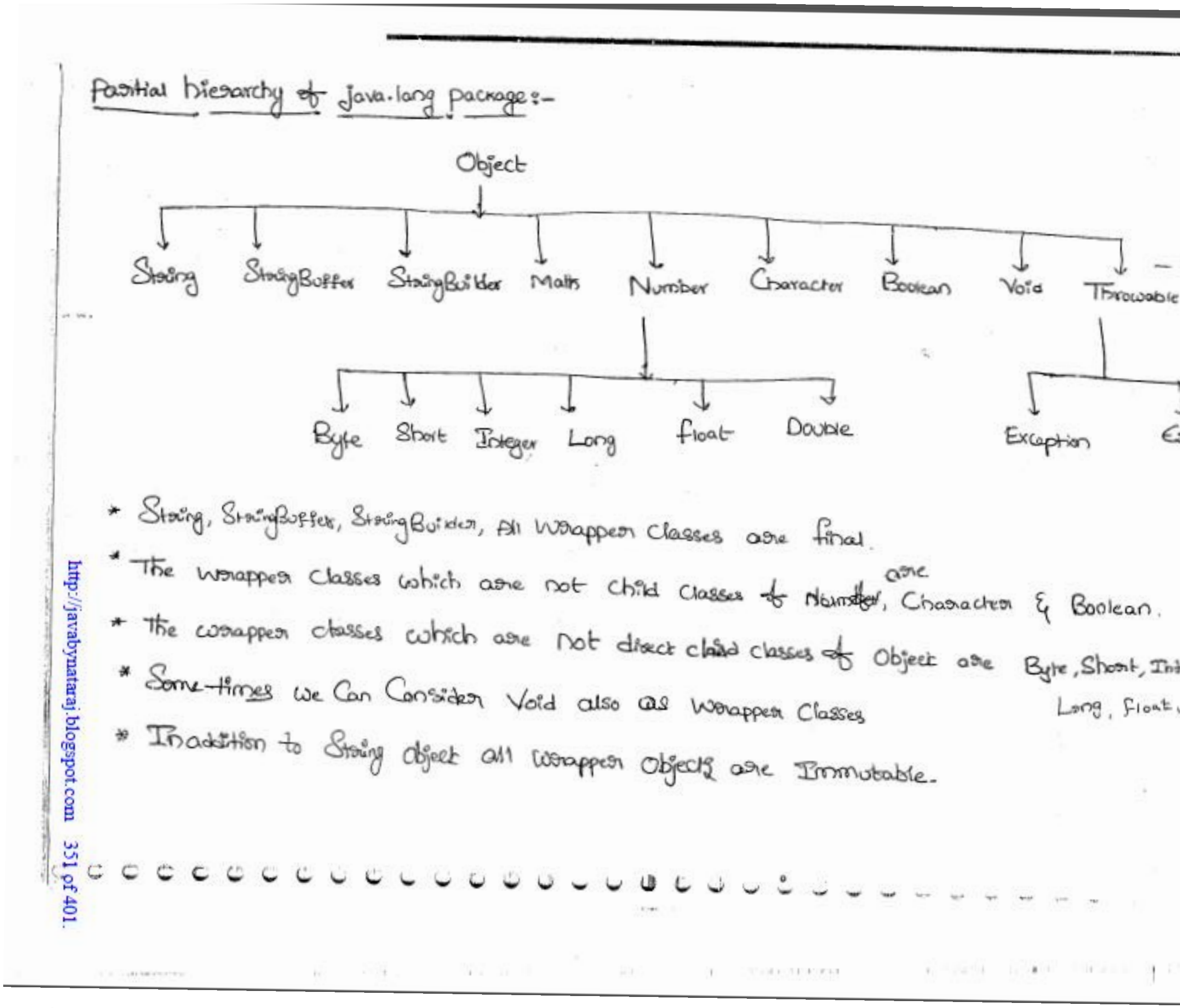
→ declaring a reference variable as final we wont get any immutability nature, in the Corresponding object we Can perform any type of change Eventhrough reference variable declared as final.

Ex:-

```
final StringBuffer sb = new StringBuffer("durga");
        Sb.append("Software");
      S.o.pln(sb);   durgasoftware
```

→ Hence final variable & Immutability both Concepts are different.


String
toString()
parseXxx()
ValueOf()
toString()
Primitive
ValueOf()
Wrapper Object
xxxValue()



Partial hierarchy of java.lang package:-

Object
String
StringBuffer
StringBuilder
Math
Number
Character
Boolean
Void
Throwable
Byte
Short
Integer
Long
float
Double
Exception

* String, StringBuffer, StringBuilder, All Wrapper Classes are final.

* The wrapper Classes which are not Child Classes of Number, are Character & Boolean.

* The wrapper classes which are not direct child classes of Object are Byte, Short, Int
Long, float,

* Some-times we Can Consider Void also as Wrapper Classes

* Inaddition to String object all wrapper Objects are Immutable.

transient Vs Static :-

→ Static variables are not part of object hence they won't participate in Serialization process. Due to this declaring a Static variable as transistent There is no impact.

transisent Vs final :-

→ final variables will be participated into Serialization directly by their values Hence declaring a final variable with transistent there is no impact.

Inheritance ex.

1. When parent and child have different name methods (No Overriding)

```
class Vehicle{
  void run(){
     System.out.println("Vehicle is running");
           }
  }

  class Bike1 extends Vehicle{
    void run1(){
   System.out.println("Bike is running safely");
   }
}

class Bike2 {
  public static void main(String args[]){
  Vehicle obj = new Bike1(); //Cases
  obj.run(); //Cases
  }
}
```

Case 1:

Vehicle obj = new Bike1();

obj.run();

Output: Vehicle is running

Case2 :

Vehicle obj = new Bike1();

obj.run1();

Output: **Compile by: javac Bike2.java**

```
122.93/Bike2.java:16: error: cannot find symbol
 obj.run1();
 ^
 symbol: method run1()
 location: variable obj of type Vehicle
1 error
```

2. When parent and child have same name methods (Overriding)

```
class Vehicle{
  void run(){
     System.out.println("Vehicle is running");
           }
  }

  class Bike1 extends Vehicle{
    void run(){
   System.out.println("Bike is running safely");
   }
}

class Bike2 {
  public static void main(String args[]){
  Vehicle obj = new Vehicle(); //Cases
  obj.run(); //cases
  }
}
```

Case1:

Vehicle obj = new Vehicle();

  obj.run();

Output: Vehicle is running

Case2:

Vehicle obj = new Bike1();

  obj.run();

Output: Bike is running safely