Generics

**Type Safe:** By default arrays are type safe i.e. a String array can store only string objects.

**1. What is Generics?**

It is the new feature in java 5 by using this we can solves the type safety and typecasting problems in Collection Frame Work.

Type safety: Array’s are by default type safe i.e. String array can only accept string object.

* Generics not applicable on primitives.
* Generic parameters are not participating in the polymorphism. But base type is following that. i.e.

List <*String*> l = new Array List<*String*> ()

Base type Parameter type

* Generic is the concept which is applicable on compile time to provide type safe. There is no generic concept at runtime.
* Through a process called “type erasure”, the compiler does all of its verifications on your generic code and then strips the type information out of the class byte code.
* 
* **Class Map Set List Ordered Sorted**

String

* All characters in Strings are uni-code characters (16-bits).
* String objects are immutable (“anil”) but not its references (s1, s2, s3).
* When compiler encounters a string literal it checks for the identical literal in the Constant pool. If there it assigns the reference to it.
* If any String object is created at run time it just created only at heap memory but not at SCP.
* Memory leak will happen in Strings when we are using “substring()” method. But from java 1.7 onwards it’s resolved.

1. **Why String is immutable?**

S1

### Anil

S2

S3

String Constant pool

Once we assign a value, that value is never changed is called immutability

Here s1,s2,s3 are string reference and “Anil” is string object.

If we change s2 refered object value it will reflects the s1 and s3 values.

To avoid that string became immutable.

1. **Difference between new keyword and normal?**

String s=”abc”; 🡪in this case one string object and one reference variable is created in connection pool.

String s=new String(“abc”); 🡪in this case 2objects one is at main memory and another one is at constant pool and one reference variable to refer obj at main memory.

1. **What is SCP?**

* Scp is a special area of memory created by jvm. When compilers encounters a string literal it checks weather identical string is available in pool or not if availble it assings the new string reference to it. If not it creats the new string and assing.
* In this case many references refer same string if any reference change the string it will reflects the other one so string is marked as immutable.
* Objects in scp is not eligible for G.C even it has no reference. All objects in scp will destroyed only when JVM is shout down.

1. **What is StringBuffer?**

* It is mutable class having initial capacity is “**16**”.
* All methods are synchronized and thread safe by default.
* Capacity = (Initial capacity + 1)\*2.
* StringBuffer sb=new StringBuffer(“anil”);

System.out.println(sb.capacity()); 🡪20 (16+4).

* String Buffer not overriding equlas() so on calling equals() on string buffer object it will calls object class equals(). It can perform reference comparission.
* Length is final variable in Array class, but length () is a final method available on String class.

Wrapper Calsses

* All wrapper classes are immutable
* Boolean class constructor takes either ‘true’ or ‘false’ or string. The string value is ‘true’ (case insensitive). Anything else is taken as false.
* Void is also a wrapper class. The Void class is an uninstantiable placeholder class to hold a reference to the Class object representing the Java keyword void.

public class Test {

public static void main(String[] args) throws SecurityException, NoSuchMethodException {

Class c1 = Test1.class.getMethod("Testt",null).getReturnType();

System.out.println(c1 == Void.TYPE);

System.out.println(c1 == Void.class);

}

}

Serialization

Serialization has below mentioned points

* Default Protocol (implementing Serializable )
* Customizing default Protocol (overriding readObject(), writeObject())
* Creating own protocol (Externalizable)
* Serial version Id (serialVersionUID)
* Serialization in case of singleton
* Serialization with Inheritance.
* Purpose of readReslove() and writeReplace()
* Serialization is the process of saving object state into sequence of bytes and recreating live object by using this sequence of bytes.
* Serialization process won’t care about access modifiers such as ‘private’ –all not transient variables are part of object state and are eligible for serialization.

FileInputStream fis = new FileInputStream(fileName);

ObjectInputStream ois = new ObjectInputStream(fis);

Object obj = ois.readObject();

FileOutputStream fos = new FileOutputStream(fileName);

ObjectOutputStream oos = new ObjectOutputStream(fos);

oos.writeObject(obj);

* Static variables are not part of the object, so it won’t participate in this process. While de-serializing a value can be available for Static variables if the same is provided while initialization of the base class. It doesn’t mean that static variable will be serialized. It only means that static variable will be initialized with the same value, it is assigned while loading the class (Which is TATA in this case). Please pay attention to example code.
* If the variable is defined as Static and Transient both than static modifier will dominate the behaviour of variable and not Transient.
* Static and transient is different. In cases, their behaviour looks same, but not always. If you have assigned a value to a variable while loading the class then that value will be assigned to the static variable while de-serializing the class but not to transient.

class Emp implements Serializable {

public String lastName;

/\* Here i am providing the value of company name, companyCEO and address while defining these variables.\*/

static String companyName = "TATA";

transient String address = "DEL";

static transient String companyCEO = "Jayshree";

}

* From above ex, while deserialization companyName and companyCEO variables will have the data but not address, because we assigned value to these static variables at class level.
* ‘Final’ variables will part of serialization, and making final variable as ‘transient’ there is no impact. Even though it will be part of object state.

Externalizable:

* Externalizable interface in child interface of Serializable i.e. Externalizable extends Serializable. So if any class implements Externalizable interface and override it’s writeExternal() and readExternal() methods then first preference is given to these methods over default serialization mechanism provided by JVM.

|  |  |
| --- | --- |
| **SERIALIZABLE** | **EXTERNALIZABLE** |
| Serializable is a marker interface i.e. does not contain any method. | Externalizable interface contains two methods writeExternal() and readExternal() which implementing classes MUST override. |
| Serializable interface pass the responsibility of serialization to JVM and it’s default algorithm. | ExternalizaExternalizableble provides control of serialization logic to programmer – to write custom logic. |
| Mostly, default serialization is easy to implement, but has higher performance cost. | Serialization done using Externalizable, add more responsibility to programmer but often result in better performance. |
| It’s hard to analyze and modify class structure because any change may break the serialization. | It’s easier to analyze and modify class structure because of complete control over serialization logic. |
| Default serialization does not call any class constructor. | A public no-arg constructor is required while using Externalizable interface. |

Collections

## Array List:

* ArrayList is Resizable-array implementation of the java.util.[List](http://www.javamadesoeasy.com/2015/04/list-hierarchy-in-java-detailed.html) interface. As ArrayList uses array it is index based structure in java. Default size is 10.
* It will allow adding ‘null’.
* Java.util.ArrayList implements RandomAccess(Marker interface) to indicate that they support fast random access (i.e. index based access) in java.
* Random Access is a marker interface; the primary purpose of this interface is to allow generic algorithms to alter their behaviour to provide good performance when applied to either random or sequential access lists.
* The iterator returned by ArrayList/Linked list is “**Fail-Fast**”. Means any structural changes like adding, deleting during iteration will throw ConcurrentModificationException. (Enhanced for loop is also “Fail-Fast”).
* We must prefer ArrayList for when add and remove operations are less as compared to get operations

|  |  |  |
| --- | --- | --- |
| **Operation/ method** | **Worst case** | **Best case** |
| add | O (n), when array is full it needs restructuring, Operation runs in amortized constant time. | O (1), when array does not need any restructuring. |
| remove | O (n), when removal is done from between restructuring is needed. | O (1), when removal is done at last position, no restructuring is needed. |
| get | O (1), it is index based structure. So, complexity of get operation is always done in O (1). | O (1) it is index based structure. So, complexity of get operation is always done in O (1). |
| set | O (1), it is index based structure; no restructuring is needed in set operation. So, complexity of operation is always O(1) | O (1), it is index based structure, no restructuring is needed in set operation. So, complexity of operation is always O(1) |
| iterator | O(n), because iteration is done over each and every element. | O (n), because iteration is done over each and every element. |

## Linked List:

* LinkedList implemented using Double-linked. It implements Deque (*public interface Deque extends Queue*) and java.util.[List](http://www.javamadesoeasy.com/2015/04/list-hierarchy-in-java-detailed.html) interface. A linked list is a data structure consisting of a group of nodes which together represent a sequence. Under the simplest form, each node is composed of a data and a\reference (in other words, a link) to the next node in the sequence in java.
* The java.util.Deque interface is a subtype of the [java.util.Queue](http://tutorials.jenkov.com/java-collections/queue.html) interface. It represents a queue where you can insert and remove elements from both ends of the queue.
* In LinkedList, data is accessed sequentially, so for obtaining data at specific index, iteration is done on nodes sequentially in java.
* Get method iterates on nodes sequentially to get element on specified index. Hence, offering O (n) complexity. get() -> O(n)
* LinkedList maintains insertion order if all elements are added using add() or addLast()method because both method adds elements add element at end of LinkedList**.**
* List list = Collections.synchronizedList(new LinkedList(...))
* It will supports Queue operations like peek (), poll (), etc,.

|  |  |  |
| --- | --- | --- |
| **Operation/ method** | **Worst case** | **Best case** |
| add(E element) | O(1), Adds specified element to the end of LinkedList. | O (1), Adds specified element to the end of LinkedList. |
| add(int index, E element) | O(n), because iteration is done on all elements one by one to find out specified index. Current element is placed at specified index and one is added to indices of subsequent elements on right. | O(n) |
| addFirst(E element) | O(1) | O(1) |
| addLast(E element) | O(1) | O(1) |
| remove() | O (1), Method retrieves and removes the first element (head) of this list. | O(1) |
| remove(int index) | O (n), because iteration is done on all elements one by one to find out specified index. One is subtracted from indices of subsequent elements on right. | O(n) |



# Set:

* + HashSet
  + LinkedHashSet
  + TreeSet
  + EnumSet

## Hash Set:

* It used to store elements, it doesn’t allow duplicates.
* HashSet internally maintain HashMap (transient). It stores each element as a key in the map and creates and stores a static object for Object class as a value.
* It won’t maintain ‘Insertion Order’ and allow one ‘null’ value.
* Any set implementation don’t have any get() method. We have to iterate the set to get elements.
* Default capacity is 16 and load factor is 0.75, i.e. when set will be 75% filled, its capacity will be doubled.

## Linked Hash Set:

* This class extends ‘HashSet’ class. It is Hash table and linked list implementation of set interface.
* The implementation differs from HashSet in that it maintains a doubly-linked list running through all of its entries. This linked list defines the iteration ordering, which is the order in which elements were inserted into the set (insertion-order).
* Insertion order is not affected if an element is re-inserted into the set. Before inserting an element

## Tree Set:

* This class implements ‘Navigable Set’.
* The element are ordered using their natural ordering, or by a **comparator** provided at set creation time, depending on which constructor is used.
* This implementation provides guaranteed log (n) time cost for the basic operations (add, remove and contains).



## Map:

* Map interface not extended by Collection interface. Map.Entry which describe each “key-value” pair in map object.
* Two Map instances are equal if they represent the same key-value mappings.
* Should not use “mutable” object as map key because if the key object state has changed (once state changed hash code will changes) then it’s almost impossible to get the respective value associated with it.

**HashMap:**

* The HashMap class is roughly equivalent to Hashtable, except that it is unsynchronized and permits one null key and multiple null values.
* It won’t maintain the Insertion order (0n which basis it will store?) and this implementation is not synchronized**.**
* An instance of HashMap has two parameters that affect its performance: initial capacity and load factor. The capacity initial capacity is simply the capacity at the time the sshash table is created.
* The load factor is a measure of how full the hash table is allowed to get before its capacity is automatically increased. When the number of entries in the hash table exceeds the product of the load factor and the current capacity, the hash table is rehashed (that is, internal data structures are rebuilt) so that the hash table has approximately twice the number of buckets.
* The iterators returned by this class is **fail-fast**: if the map is structurally modified at any time after the iterator is created, in any way except through the iterator's own **remove()**, the iterator will throw a [ConcurrentModificationException](https://docs.oracle.com/javase/7/docs/api/java/util/ConcurrentModificationException.html" \o "class in java.util).
  + <https://howtodoinjava.com/core-java/collections/how-hashmap-works-in-java/>

**Why** [**immutable**](#_Immutable_Class) **class objects are preferred as “keys” in Map**:

**LinkedHashMap:**

* This implementation differs from HashMap in that it maintains a doubly-linked list running through all of its entries.
* It will maintain the insertion order. The insertion order is not affected if a key is re-inserted into the map.

**TreeMap**:

* A Red-Black tree based [NavigableMap](https://docs.oracle.com/javase/7/docs/api/java/util/NavigableMap.html" \o "interface in java.util) implementation. The map is sorted according to the [natural ordering](https://docs.oracle.com/javase/7/docs/api/java/lang/Comparable.html) (Comparable) of its keys, or by a [Comparator](https://docs.oracle.com/javase/7/docs/api/java/util/Comparator.html) **provided at map creation time**, depending on which constructor is used.
* This implementation provides guaranteed log(n) time cost for the containsKey, get, put and remove operations.
* All map implementation classes will decide whether two keys or equal or not based on *“****equals****()” but SortedMap will perform based on* compareTo/compare() methods.
* TreeMap is not synchronized. So multiple threads can access map concurrently and changes done by one thread will reflects rest of the threads. We can make TreeMap as synchronized like
  + SortedMap m = Collections.synchronizedSortedMap (new TreeMap());
* We will get ‘NullPointerException” if the specified key is null and this map uses natural ordering, or its comparator does not permit null keys

How to sort map based on ‘values’

How to get key based on value?

**ConcurrentHashMa**p:

* This class obeys the same functional specification as [Hashtable](https://docs.oracle.com/javase/7/docs/api/java/util/Hashtable.html" \o "class in java.util), and includes versions of methods corresponding to each method of Hashtable
* Thread will block entire HashTable object in case of synchronization. But ConcurrentHashMap synchronization will apply “**Segment**” wise. So it thread wont block entire map object.
* Concurrency level tells how many threads can access ConcurrentHashMap concurrently; default concurrency level of ConcurrentHashMap is **16**. ConcurrentHashMap is divided into different segments based on concurrency level. So different threads can access different segments concurrently in java
  + new ConcurrentHashMap() 🡪 it will create new map with concurrency level 16.
* When thread locks one segment for updating it does not block it for retrieval (done by get method) hence some other thread can read the segment (by get method)
* Iterator returned by ConcurrentHashMap is fail-safe. So any structural modification made to ConcurrentHashMap like adding or removing elements during Iteration will **not** **throw** any Exception
* Like ‘HashTable’ but unlike ‘HashMap’ this class does not allow storing null key or null value. Any attempt it will throws runtimeException (NullPointerException).
  + <http://www.javamadesoeasy.com/2015/04/concurrenthashmap-in-java.html>

**Queue:**

* PriorityQueue
* LinkedBlockingQueue
* ArrayLinkedBlockingQueue
* PriorityBlockingQueue
* DelayBlockingQueue

**PriorityQueue:**

* Queue follows **F**irst-**I**n-**F**irst-**O**ut model but sometimes we need to process the objects in the queue based on the priority. That is when Java PriorityQueue is used.
* The elements of the priority queue are ordered according to their [natural ordering](https://docs.oracle.com/javase/8/docs/api/java/lang/Comparable.html), or by a [comparator](https://docs.oracle.com/javase/8/docs/api/java/util/Comparator.html) provided at queue construction time, depending on which constructor is used.
* A priority queue does not permit null elements.
* The Iterator provided in method [iterator()](https://docs.oracle.com/javase/8/docs/api/java/util/PriorityQueue.html" \l "iterator--) is *not* guaranteed to traverse the elements of the priority queue in any particular order. If you need ordered traversal, consider using Arrays.sort(pq.toArray())
* Java Priority Queue implementation provides **O (log (n))** time for enqueing and dequeing method( (offer(), poll(), remove() and add())
* Both add() & offer() methods add new element into queue. Both will differs in case of “capacity-constrained” queue, that add() always returns true and throws an exception if it can't add the element, whereas offer() is allowed to return false if it can't add the element. However, this doesn't apply to PriorityQueue; the two functions are synonymous.
* PriorityQueue is **not thread safe**, so java provides PriorityBlockingQueue class that implements the [BlockingQueue interface](https://www.journaldev.com/1034/java-blockingqueue-example) to use in [java multithreading](https://www.journaldev.com/1079/multithreading-in-java) environment.

[**LinkedBlockingQueue**](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/LinkedBlockingQueue.html) — an optionally bounded FIFO blocking queue backed by linked nodes

[**ArrayBlockingQueue**](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ArrayBlockingQueue.html) — a bounded FIFO blocking queue backed by an array

[**PriorityBlockingQueue**](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/PriorityBlockingQueue.html) — an unbounded blocking priority queue backed by a heap

[**DelayQueue**](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/DelayQueue.html) — a time-based scheduling queue backed by a heap

Threads

* Each Thread will maintain its own stack.
* Once thread has started, it won’t start again. If you call start () again it will throw “illegal thread state exception”.
* If we override start () then run () won’t call.

**Thread vs. Runnable**

We can create a thread in two ways.

* By extending Thread class.
* By implementing Runnable interface.
* Each thread created by extending the Thread class creates a unique object for it and get associated with that object. On the other hand, each thread created by implementing a Runnable interface shares the same Runnable instance.
* Implementing Runnable makes your class more flexible. If you extend thread then the action you’re doing is always going to be in a thread. However, if you extend Runnable it doesn’t have to be. You can run it in a thread, or pass it to some kind of executor service, or just pass it around as a task within a single threaded application.
* One must extend a Thread class only if it has to override or specialise some other methods of Thread class. You must implement a Runnable interface if you only want to specialise run method only.
* Extending the Thread class introduces tight coupling in the code as the code of Thread and job of thread is contained by the same class. On the other hand, Implementing Runnable interface introduces loose coupling in the code as the code of Thread is separate from the job assigned to the thread.

**Class level vs. Object level locking:**

* Object level locking is mechanism where it will allow only one thread of a given instance (thread created from object) to execute “synchronized non-static method/block”. This should always be done to make instance level data thread safe. For ex :

private final Object lock = new Object();

synchronized (**this or lock**) {

// this block will be execute by only one thread of one instance at a time.

}

ObjectThread first = new ObjectThread ();

Thread t = new Thread (first, "first");

t.start ();

Thread t2 = new Thread (first, "second");

t2.start ();

In this case we created two threads from a single object. So at a particular time only one thread allow to execute synchronized block/method.

* Class level locking prevents multiple threads to enter in synchronized block in any of all available instances on runtime. This means if in runtime there are 100 instances of  DemoClass, then only one thread will be able to execute demoMethod () in any one of instance at a time, and all other instances will be locked for other threads. This should always be done to make static data thread safe.

private final static Object lock = new Object();

synchronized (**DemoClass.class or** lock) {

            //other thread safe code

 }

* Java Synchronization will throw NullPointerException if object used in java synchronized block is null. For example, in above code sample **if lock is initialized as null**, the synchronized (lock) will throw NullPointerException.
* According to the Java language specification you **cannot use java synchronized keyword with constructor** it’s illegal and result in compilation error.

**Volatile Keyword:**

* Volatile keyword is applicable only for variables, not for methods and classes.
* Volatile keyword will give guarantee that the value of variable will be read from “main memory” instead of cache memory/thread stack.
* When a thread writes to a volatile variable, then not just the volatile variable itself is written to main memory. Also all other variables changed by the thread before writing to the volatile variable are also flushed to main memory.
* When a thread reads a volatile variable it will also read all other variables from main memory which were flushed to main memory together with the volatile variable.
* A volatile variable should not be a final. It will gives you compile time error.

**Volatile vs. Atomic in java:**

**Happens Before in java:**

**Thread Methods:**

**sleep:**

* Sleep is a static method available in Thread class, which causes current executing thread to sleep for specified time.
* Public static native void sleep(long mills) throw InteruptedException;
* When sleep () is called on thread it goes from running to waiting state and can return to Runnable state when sleep time is up.
* Thread need not to acquire object lock before calling sleep () method i.e. sleep () method can be called from outside synchronized block.

**Wait ():**

* Wait () is an instance method available in Object class, which causes current executing thread to wait until another thread invokes notify()/notifyAll() methods on for same object.
* The current thread must own this object's monitor. The thread releases ownership of this monitor and waits until another thread notifies threads waiting on this object's monitor to wake up either through a call to the notify()/ notifyAll() method. The thread then waits until it can re-obtain ownership of the monitor and resumes execution.
  + public final void wait(long timeout) throws InterruptedException
  + public final void wait(long timeout, int nanoseconds) throws IE.
  + public final void wait() throws InterruptedException
  + wait() internally calls wait(0)

**Yield ():**

**Thread Starvation:**

* Starvation happens when thread does not enough CPU for its execution
* Starvation describes a situation where a thread is unable to gain regular access to shared resources and is unable to make progress.
* This happens when shared resources are made unavailable for long periods by "greedy" threads. For example, suppose an object provides a synchronized method that often takes a long time to return. If one thread invokes this method frequently, other threads that also need frequent synchronized access to the same object will often be blocked.
* Low priority threads get less CPU (time for execution) as compared to high priority threads. Lower priority thread may starve away waiting to get enough CPU to perform calculations.
* Thread might be waiting indefinitely for lock on object’s monitor (by calling [wait ()](http://www.javamadesoeasy.com/2015/03/wait-and-notify-methods-definition-8.html) method), because no other thread is calling [notify()/notifAll()](http://www.javamadesoeasy.com/2015/03/difference-between-notify-and-notifyall.html) method on object. In that case, Thread starves away to get CPU.

**Livelock:**

* A thread often acts in response to the action of another thread. If the other thread's action is also a response to the action of another thread, then live lock may result.
* As with deadlock, live locked threads are unable to make further progress. However, the threads are not blocked — they are simply too busy responding to each other to resume work. This is comparable to two people attempting to pass each other in a corridor: Alphonse moves to his left to let Gaston pass, while Gaston moves to his right to let Alphonse pass. Seeing that they are still blocking each other, Alphone moves to his right, while Gaston moves to his left. They're still blocking each other, so...

**ThreadLocal:**

* When you create thread that implements the Runnable interface and then start various Thread objects using the same Runnable object, all the threads share the same attributes that are defined inside the Runnable object. This essentially means that if you change any attribute in a thread, all the threads will be affected by this change and will see the modified value by first thread. To avoid this we are using ThreadLocal.
* This class provides thread-local variables. These variables differ from their normal counterparts in that each thread that accesses one (via its get or set method) has its own, independently initialized copy of the variable.
* ThreadLocal instances are typically private static fields in classes that wish to associate state with a thread (e.g., a user ID or Transaction ID).
* This class contains methods like ‘initialValue ()’, ‘get()’, ‘set()’, and ‘remove()’..

Concurrency

In previous examples, there’s a close connection (dependency) between Task and the Thread (which is create by Runnable or Thread). To avoid is we are using Executors.

**Executor Interfaces:**

1. Executor (i)
   1. execute()
2. ExecutorService (i)
   1. Future submit( Callable task)
   2. Future submit( Runnable task)
   3. Future submit(Runnable task, T result)
   4. Void shutdown()
   5. List<Runnable> shutdownNow()
   6. boolean isTerminated().
3. ScheduleExecutorService

* Callable is an interface, which contains call(). We can create a task by implementing Runnable or Callable interface.
  + call() method returns computed result or throws an exception if unable to do so.
  + A Runnable does not return a result and can neither throw a checked exception
* Future is an interface, which is used to fetch the result returned by submit(). For ex: if we observe above points b, c “Runnable task(run method)” won’t return anything because run() don’t have return type.
  + In point a, successful execution of call() we will get the output which is returned by call() of task.
  + So in point b, on successful execution of run() method, future.get() returned null.
  + In case of point c, on successful execution of submit() will return ‘result’ value(second parameter)

**Thread Pool:**

**Count Down Latch:**

* This class introduced in java5 in java.util.concurrent package.
* This class **enables a java thread to wait until other set of threads completes** their tasks. E.g. Application’s main thread wants to wait, till other service threads completed.
* CountDownLatch works by having a counter initialized with number of threads, which is decremented each time a thread complete its execution. When count reaches to zero, it means all threads have completed their execution, and thread waiting on latch resumes the execution.
* Once latch completed (count is zero) it won’t start again. It’s possible in Cyclic Barrier.
* It has a constructor like below.
  + public CountDownLatch(int count) //count is essentially the number of threads, for which latch should wait
* It contains below mentioned methods
  + **await**() - Causes the current thread to wait until the latch has counted down to zero, unless the thread is [interrupted](https://docs.oracle.com/javase/7/docs/api/java/lang/Thread.html#interrupt()).
  + **await**(long timeout, TimeUnit unit) - Causes the current thread to wait until the latch has counted down to zero, unless the thread is [interrupted](https://docs.oracle.com/javase/7/docs/api/java/lang/Thread.html#interrupt()), or the specified waiting time elapses.
  + **countDown**() - Decrements the count of the latch, releasing all waiting threads if the count reaches zero. It reduces the latch count by 1. If count is zero then all waiting threads are released.
  + **getCount**() – returns the current count.

**CyclicBarrier:**

* There might be situation where we might have to trigger event only when one or more threads completes certain operation.
* Two or more threads wait for each other to reach a common barrier point. When all threads have reached common barrier point (i.e. when all threads have called await() method)
  + All thread are released and
  + Event can be triggered as well.
* We have two types constructor
  + CyclicBarrier(int parties)
    - New cyclic barrier is crated where parties is number of thread wait for each other to reach common barrier point, when all threads have reached common barrier point, parties number of waiting threads are released.
  + CyclicBarrier(int parties, Runnable barrierAction)
    - New CyclicBarrier is created where parties number of thread wait for each other to reach common barrier point, when all threads have reached common barrier point, parties number of waiting threads are released and barrierAction (event) is triggered.
* await method has two forms
  + int **await()** throws InterruptedException, BrokenBarrierException
    - If the current thread is not the last to arrive(i.e. call await() method) then it waits until one of the following things happens -
      * The last thread to call arrive(i,.e. call await() method), or
      * Some other thread interrupts the current thread, or
      * Some other thread interrupts one of the other waiting threads, or
      * Some other thread times out while waiting for barrier, or
      * Some other thread invokes reset() method on this cyclicBarrier.
  + int await(long timeout, TimeUnit unit) throws InterruptedException, BrokenBarrierException, TimeoutException
    - If the current thread is not the last to arrive(i.e. call await() method) then it waits until one of the following things happens -
      * The last thread to call arrive(i,.e. call await() method), or
      * The specified timeout elapses, or
      * Some other thread interrupts the current thread, or
      * Some other thread interrupts one of the other waiting threads, or
      * Some other thread times out while waiting for barrier, or
    - Some other thread invokes reset() method on this cyclicBarrier

**Blocking Queue**:

* java.util.concurrent.BlockingQueue is A [Queue](https://docs.oracle.com/javase/7/docs/api/java/util/Queue.html) that additionally supports operations that wait for the queue to become non-empty when retrieving an element, and wait for space to become available in the queue when storing an element.
* Java BlockingQueue doesn’t accept null values and throw NullPointerException if you try to store null value in the queue.
* BlockingQueue implementations are designed to be used primarily for producer-consumer queues, but additionally support the [Collection](https://docs.oracle.com/javase/7/docs/api/java/util/Collection.html) interface. So, for example, it is possible to remove an arbitrary element from a queue using remove(x).
* BlockingQueue implementations are thread-safe. All queuing methods achieve their effects atomically using internal locks or other forms of concurrency control.
* Blocking queue internally uses Lock and Conditions interfaces for synchronization process.
* Below is the constructors
  + [**ArrayBlockingQueue**](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/ArrayBlockingQueue.html#ArrayBlockingQueue(int))(int capacity)
  + [**ArrayBlockingQueue**](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/ArrayBlockingQueue.html#ArrayBlockingQueue(int,%20boolean))(int capacity, boolean fair)

|  |  |  |  |
| --- | --- | --- | --- |
| *Throws exception* | *Special value* | *Blocks* | *Times out* |
| **Insert** | [add(e)](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/BlockingQueue.html#add(E)) | [offer(e)](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/BlockingQueue.html#offer(E)) | [put(e)](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/BlockingQueue.html#put(E)) | [offer(e, time, unit)](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/BlockingQueue.html#offer(E,%20long,%20java.util.concurrent.TimeUnit)) |
| **Remove** | [remove()](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/BlockingQueue.html#remove(java.lang.Object)) | [poll()](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/BlockingQueue.html#poll(long,%20java.util.concurrent.TimeUnit)) | [take()](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/BlockingQueue.html#take()) | [poll(time, unit)](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/BlockingQueue.html#poll(long,%20java.util.concurrent.TimeUnit)) |
| **Examine** | [element()](https://docs.oracle.com/javase/7/docs/api/java/util/Queue.html#element()) | [peek()](https://docs.oracle.com/javase/7/docs/api/java/util/Queue.html#peek()) | *not applicable* | *not applicable* |

**Lock & Re-entrant Lock & Monitor:**

Lock:

* Lock is an interface (java.util.concurrentlocks.Lock) and its implementation provides extensive locking operations **than Synchronized** process.
* A lock helps in controlling access to a shared resource by multiple threads. Only one thread at a time can acquire the lock and access the shared resource. If a second thread attempts to acquire the lock on shared resource when it is acquired by another thread, the second thread will wait until the lock(unlock) is released.
  + void **lock()** –
    - Acquires the lock if it is not held by another thread. And sets **lock hold** count to 1.
    - f the lock is held by another thread then the current thread waits for another thread to release lock.
  + void **unlock()** –
    - If the current thread is the holding the lock then the lock hold count is decremented by 1.
    - f the lock hold count has reached 0, then the lock is released. If the current thread is not holding the lock then IllegalMonitorStateException is thrown.
  + boolean **tryLock() –** 
    - Acquires the lock if it is not held by another thread and returns true. And sets lock hold count to 1.
    - If current thread already holds lock then method returns true. And increments lock hold count by 1.If lock is held by another thread then method return false.

Condition:

* Condition is an interface available in java.util.concurrent.locks package which is used to create multiple “wait-sets” per object.
* Condition instance are similar to using [Wait(), notify() and notifyAll()](http://www.javamadesoeasy.com/2015/03/wait-and-notify-methods-definition-8.html) methods on object.
* In case of traditional synchronization, there is only one object monitor so we can have only single wait-set per object. But, Condition instance are used with Lock instance, Condition factors out the Object monitor methods (wait, notify and notifyAll) into distinct objects to give the multiple wait-sets per object.
* A Condition instance is intrinsically bound to a lock. To obtain a Condition instance for a particular [Lock](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/locks/Lock.html) instance use its [newCondition()](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/locks/Lock.html" \l "newCondition()) method.
* Lock replaces the use of synchronized methods and blocks, & a Condition replaces the use of the Object monitor methods
  + **await**() 🡪 similar to wait()
    - current thread to wait until one of the following thing happen

signal()/signalAll() method is called, or

current thread is interrupted.

* + boolean **await(long time, TimeUnit unit) 🡪** similar to wait(long timeout)
    - current thread to wait until one of the following thing happen

signal()/signalAll() method is called, or

current thread is interrupted.

Specified time elapses.

* + Void **signal() 🡪** similar to notify()
  + void **signalAll**() 🡪 similar to notifyAll()

ReentrantLock:

* This is the most widely used implementation class of Lock interface. This class implements the Lock interface in similar way as synchronized keyword.
* We can create this obj as like (constructors)
  + new ReentrantLock();
  + new ReentrantLock(boolean fair)
    - When fair is set true, first lock is obtained by longest-waiting thread.
    - If fair is set false, any waiting thread could get lock, at discretion of implementation.

### **Lock** **vs synchronized:**

* Java Lock API provides more visibility and options for locking, unlike synchronized where a thread might end up waiting indefinitely for the lock, we can use tryLock() to make sure thread waits for specific time only.
* synchronized keyword doesn’t provide fairness whereas we can set **fairness** to true while creating ReentrantLock object so that longest waiting thread gets the lock first.
* We can create different conditions for Lock and different thread can await() for different conditions.

# Immutable Class

* Class should be ‘final’. So it can’t be extended by other classes.
  + If it is not final then it allows creating sub class. Imagine you have a function that takes an Immutable an argument. I can pass a Mutable object to that function, since Mutable extends Immutable. Inside that function the concern is that that method might assume that the object is immutable when it really isn't. For example, the method might assume that, since it thinks the object is immutable, it can be used as the key in a HashMap. I could then break that function by passing in a Mutable, waiting for it to store the object as a key, then changing the Mutableobject. Now, lookups in that HashMap will fail, because I changed the key associated with the object.
* Private member variables 🡪 so that they can’t be accessed outside of the class.
* Final member variables.
* Constructor 🡪 initializes all fields in constructor.
* If immutable class contains any member variable of ‘mutable’ class, then while initializing that reference variable in constructor by cloning. So it will create new object instead of refereeing old. For ex,
* Object of mutable class - Any change made to object of mutable class doesn't produces new object.
  + Integer, [String](http://www.javamadesoeasy.com/2015/05/string-is-immutable-in-java.html) is immutable class, any changes made to object of these classes produces new object. So return reference variable of Integer.
* [HashMap](http://javamadesoeasy.com/2015/02/hashmap-custom-implementation.html) is mutable class.
  + Any changes made to HashMap object won't produce new HashMap object. So return copy/clone of object, not reference variable of HashMap.

**Cloning Process**:

* Cloning is done for copying the object, cloning can be done using shallow or deep copy.

protected native object clone() throws CloneNotSupportedException();

* + clone() is present in Object class
  + It is protected method, can’t able to call outside of class without ‘inheritance’.
  + Clone is not a keyword in java.
* Class must implement marker interface java.lang.Cloneable. If class doesn’t implement Cloneable than calling clone method on its object will throw CloneNotSupportedException.
* While doing cloning, it won’t call “constructor” of class.
* We have two types of cloning process
  + Shallow copy:
    - In shallow copy different object will be created in clone process, but the member variables keep on referring to same object.

Id

Cloned Emp obj

Emp obj

map

* Deep Copy:
  + In deep copy, different object is created after cloning (i.e. clonedEmp is created from emp) , also member variables starts referring to different objects (i.e. name and map).
  + For ex:
    - x.clone() != x 🡪 true
    - x.clone().getClass() == x.getClass() 🡪 true

**Hash code and equals() in java**:

* hashCode() method is used to get a **unique integer for given object.** This integer is used for determining the bucket location, when this object needs to be stored in some [HashTable](https://en.wikipedia.org/wiki/Hash_table" \o "Hashtable) like data structure. By default, Object’s hashCode() method returns and integer representation of memory address where object is stored.
* equals () method, as name suggest, is used to simply verify the equality of two objects. Default implementation checks the **object references of two objects** to verify their equality.

Why hashcode() has to override when equals() did:

* Let us consider we are storing ‘Employee’ class objects into a set. Two employee objects are equal or not we are defining in it’s equals() implementation.
* To add an object into set first it will check exact bucket location based on it’s hashcode. If any two objects are having same hashcode then it will perform equality check on those objects. So in this case we have to provide implementation for hashcode in such a way like two employee object having same id should have same hashcode.
* Whenever **a.equals(b)**, then *a.hashCode()* must be same as *b.hashCode()*.

public boolean equals(Object obj);

public int hashCode();

## Ways to create Objects in java?

1. By using ‘new’ keyword.
2. Using newInstance () in class called ‘**Class’**.
   1. In this process it will create object dynamically. If class not available it will throw ‘ClassNotFoundException’.
   2. Can only invoke default constructor (zero arg).
   3. Can able to call public constructor alone.
   4. It will throw any exception thrown by the constructor, regardless of whether it is checked or unchecked.

Class c = Class.forName (“Test”);🡪 (class will load. Static block of class will called)

Test t = (Test) c.newInstance ();

1. Using newInstance () in **Constructor** class.
   1. Can able to invoke parameterized constructor
   2. Can able to call private constructor.
   3. Always wraps the thrown exception with an [InvocationTargetException](https://docs.oracle.com/javase/8/docs/api/java/lang/reflect/InvocationTargetException.html" \t "_blank).

Class c = Class.forName("com.threads.Delete");

Constructor[] constr = c.getDeclaredConstructors();

constr[0].newInstance()

1. Using cloning process.
2. De-serialization process

In short, use new if you know at compile-time what the type of the object is that you want to create. Use Class.forName().newInstance() if you don't know what type of object you'll be making.

1. **What is a class?**

* Class is template that can describes the kind of state and behaviour and supports objects of its type.

1. **What is an object?**
2. An object is an instance of class, which have its own state and access all of the behaviours defined by its class.

## What is encapsulation?

* Binding (or wrapping) code and data together into a single unit is known as encapsulation. Encapsulation uses to hide the implementation details and members behind the interface.
* Encapsulation forces the callers of code to go through methods (getter and setter methods) rather than variables.
* All the members of the class declared as private then it called as *Tightly Encapsulated class*

## What is abstraction?

* + Hiding internal details and showing functionality is known as abstraction. For example: phone call, we don't know the internal processing.
* In java, we use abstract class and interface to achieve abstraction.

1. **What is inheritance?**

* A class that can be derived from another class is called sub class or derived class and the class from which the sub class is derived is called super class. This mechanism is called inheritance.
* The idea of inheritance is simple but powerful: When you want to create a new class and there is already a class that includes some of the code that you want, you can derive your new class from the existing class. In doing this, you can reuse the fields and methods of the existing class without having to write (and debug!) them yourself.

## Why object class is super class?

* By having the Object as the super class of all Java classes, without knowing the type we can pass around objects using the Object declaration.
* Before generics were introduced, imagine the state of heterogeneous Java collections. A collection class like ArrayList allows storing any type of classes. It was made possible only by Object class hierarchy.

## What is covariant return?

* It is the java 1.5 feature. You r allow to change the return type of the overriding method as long as the new return type is subtype of the overridden method.
* Covariant return is not applicable for the primitives.

1. **What is Inheritance?**
2. Inheritance allows a class to be a sub class for a super class.
3. Inheritance is the base concept for IS-A, overloading, overriding, polymorphism.
4. **What is IS-A relationship?**

* IS-A relationship refers to inheritance or implementation.
* IS-A relationship is expressed with the keyword “extends”
* Main advantage is reusability.

1. **What is HAS-A relationship?**

* HAS-S relationship is based on usage rather than inheritance.
* HAS-A relationship is getting by using “new“keyword. It is also called as **aggregate or composition**

1. **Difference between valueOf () and parseXxx () methods?**

* Both methods takes Strings as arguments converts the primitives into the wrapper type.
* But the valueOf() returns a newly created wrapped type.
* parseXxx() returns the named primitive.

1. **Difference b/w String Buffer and String Builder?**

* All the methods in the String Buffer (capacity=16 for= (size+1)\*2) class is synchronized by default but in String Builder class not synchronized (1.5)
* StringBuffer and StringBuilder do not override the default equals() method in Object class.
* Passing a null argument to a constructor or method in both classes will cause a [NullPointerException](https://docs.oracle.com/javase/8/docs/api/java/lang/NullPointerException.html" \o "class in java.lang) to be thrown

1. **Ways to call Garbage Collector?**

There are 2 ways.

1. Using System class 🡪 System.gc(); (static method)
2. Using Runtime class 🡪 Runtime.gc(); (instance method)
   1. Runtime rt = Runtime.getRuntime();
   2. rt.gc();
3. **Java memory model?**
4. **Stack:**
   1. One stack is created per thread and it stores stack frames which again stores local variables and if a variable is a reference type then that variable refers to a memory location in heap for the actual object.
   2. If stack memory full then java will throw ***java.lang.StackOverFlowError.***
   3. This memory is thread safe, as every thread operates on his own stack.

**2. Heap:** All kinds of objects will be created in heap only.

Heap memory is again divided into 3 portions

1. **Young Generation**: Stores objects which have a short life, Young Generation itself can be divided into two categories Eden Space and Survivor Space (S1, S2).
2. **Old Generation**: Store objects which have survived many garbage collection cycles and still being referenced.
3. **PermGen**: Stores metadata about the program e.g. runtime constant pool and String constant pool belongs to permanent generation area of Heap memory. Method Area is part of space in the Perm Gen and used to store class structure (runtime constants and static variables) and code for methods and constructors.
4. **What** **is “Meta Space in java 8?**

Permanent generation has been removed in HotSpot for JDK8 because of following drawbacks.

* Fixed size at startup – difficult to tune.
* Internal Hotspot types were Java objects : Could move with full GC, opaque, not strongly typed and hard to debug, needed meta-metadata.
* Simplify full collections : Special iterators for metadata for each collector
* Want to deallocate class data concurrently and not during GC pause.

The Permanent Generation (PermGen) space has completely been removed and is kind of replaced by a new space called **Metaspace**. The consequences of the PermGen removal is that obviously the *PermSize and MaxPermSize JVM arguments are ignored* and you will never get a java.lang.OutOfMemoryError: PermGen error.

1. **What is use of Runtime class?**

* Every Java application has a single instance of class Runtime that allows the application to interface with the environment in which the application is running. The current runtime can be obtained from the getRuntime method.
* Free memory (), total Memory (), gc (), maxMemory(), get Runtime().

1. **What is coupling & coherence?**

* The degree of dependency b/w components is nothing but coupling. We can achieve coupling by using Interface.
* For every component we have to define a well defined single purpose. Such type of components are said to be followed high “cohesion”.

1. **What is use of the intern ()?**

* Intern () is used to represent the string object created in the StringConstantPool i.e. used to represent the corresponding string in SCP for the string created in heap

1. **What is a java annotation?**

Annotations provide data about a program that is not part of the program itself. They have no direct effect on the operation of the code they annotate.

Annotations have a number of uses, among them:

* **Information for the compiler** — Annotations can be used by the compiler to detect errors or suppress warnings.
* **Compiler-time and deployment-time processing** — Software tools can process annotation information to generate code, XML files, and so forth.
* **Runtime processing** — some annotations are available to be examined at runtime.
  + Override
  + Deprecated
  + Suppress Warnings(“unchecked”);

1. **What is Serialization and deserialization?**

* Serialization is nothing but saves the object along with its state.
* We use ObjectOutputStream (write Object) and ObjectInputStream (read Object) classes for this process.
* Static variables are not participating in serialization bcz. They are not part of the object state.

1. **What is object graph?**

* When we are saving an object into a file, all the objects which are reachable from that object are also saved. This set of objects are called object graph.
* If any class in object graph is not implement Serializable interface at runtime we get not serializable exception.

1. **Inheritance in Serialization?**
   * If child class is serialized and super class is not still we allowed performing serialization on sub class objects. While performing serialization jvm ignores the inherited variables which are coming from non – serializable parents.
   * While performing deserialization jvm will check is there any parent class is non Serializable or not. If not jvm will create an object for every non serializable parent and share its instance variables for the current child object.
   * **Non serializable parent class should have no argument constructor** otherwise we will get runtime error.
2. **What is externalization?**

* It is an interface that can provide custom serialization for us.
* Externalization(i) extends serialization(i)
* We provide implementation for writeExternal() and readExternal() methods.

1. **What is use of Native in java?**

It marks a method, that it will be implemented in other language not in java. It will work together with java native interface. Native is mostly used in

* Need to call a library from java that is developed in other language.
* You need to access system or hardware resources that are only reachable from the other language (typically C). Actually, many system functions that interact with real computer (disk and network IO, for instance) can only do this because they call native code.

1. [**Difference between <? super T> and <? extends T> in Java**](https://stackoverflow.com/questions/4343202/difference-between-super-t-and-extends-t-in-java)

* List<? super Integer> foo3 means
  + Foo3 = new ArrayList<Integer>();
  + Foo3 = new ArrayList<Number>();
  + Foo3 = new ArrayList<Object>();

Because foo3 will accepts all **super** classes of Integer.

* List<? extends Number> foo3 means
  + Foo3 = new ArrayList<Number>();
  + Foo3 = new ArrayList<Integer>();
  + Foo3 = new ArrayList<Double>();

1. **What is java Class loader?**
2. **Why ‘Object’ class is super class in java?**
3. Is **java pass by value or pass by reference?**
   * Java is pass by value.

# Oops

1. **What are the oops concepts?**
2. Abstraction.
3. Encapsulation
4. Inheritance
5. Polymorphism

1.5 features

* Var – args
* Generics
* Enums
* Auto boxing and un boxing
* Static imports
* Co variant return type

## Var – args:

* The method which can allow ‘n’ no. Of arguments is called var - arg method.
* These methods internally implemented using single dimension array.
* Var arg parameter must be the last parameter.
* We can’t take more than one var arg parameter.
* Var arg method gives least priority.
* Widening, boxing, var arg.
* It is illegal to box after widening but legal widening after boxing.

# 1.8 features

* Static and default methods in interface.
* Functional interfaces & Lambda Expressions
* Method References.
* Streams
* Collection Improvements

**Static & Default methods**:

* Before java 8 all methods in interfaces are by default ‘abstract’. But from java 8, interfaces are allowed to provide implementations to methods using ‘default’.
* Java interface default methods will help us in extending interfaces without having the fear of breaking implementation classes. One of the major reason for introducing default methods in interfaces is to enhance the Collections API in Java 8 to support lambda expressions.
* It’s made mandatory to provide implementation for common default methods of interfaces. For ex, if a class implementing two interfaces and both interfaces have common default method then the class should provide implementation for that method otherwise compiler will throw compile time error.

## Static methods:

* Static methods are same like default methods except that we can’t override them in the implementation classes. This feature helps us in avoiding poor implementation in implementation classes.
* Java interface static method is part of interface, we can’t use it for implementation class objects.

**Functional Interfaces:**

* An interface with exactly one abstract method is known as Functional Interface. A new annotation @FunctionalInterface has been introduced to mark an interface as Functional Interface and it’s any optional to mark.
* A new package java.util.function with bunch of functional interfaces is added to provide target types for lambda expressions and method references. For ex: Consumer, Supplier, Function and Predicate
* The major benefit of java 8 functional interfaces is that we can use **lambda expressions** to instantiate them and avoid using bulky anonymous class implementation.