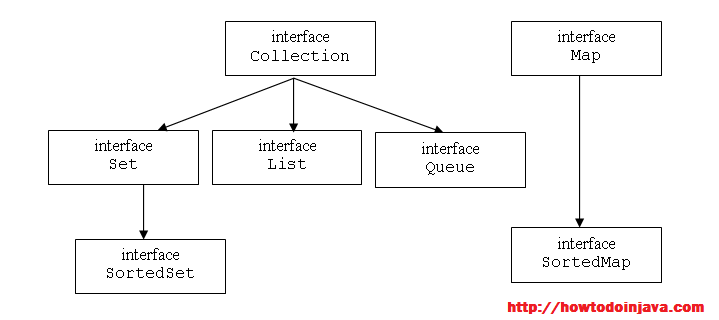
##### **Java Collection**

##### Explain Collection’s hierarchy?



Java Collection Hierarchy

As shown in above image, collection framework has one interface at top i.e. **Collection**. It is **extended by Set, List and Queue interfaces**. Then there are loads of other classes in these 3 branches which we will learn in following questions.

Remember the signature of Collection interface. It will help you in many question.

**publicinterfaceCollection extendsIterable {**

**}**

Framework also consist of Map interface, which is part of collection framework. but it does not extend Collection interface. We will see the reason in 4th question in this question bank.

##### **Why Map interface does not extend Collection interface?**

A good answer to this interview question is “**because they are incompatible**“. Collection has a method add(Object o). Map can not have such method because it need key-value pair. There are other reasons also such as Map supports keySet, valueSet etc. Collection classes does not have such views.

Due to such big differences, Collection interface was not used in Map interface, and it was build in separate hierarchy.

##### **How to convert an array of String to arraylist?**

This is more of a programmatic question which is seen at beginner level. The intent is to check the knowledge of applicant in Collection utility classes. For now, lets learn that there are two utility classes in Collection framework which are mostly seen in interviews i.e. **Collections and Arrays**.

Collections class provides some static functions to perform specific operations on collection types. And Arrays provide utility functions to be performed on array types.

**List wordList = Arrays.asList(words);**

**How to reverse the list?**

This question is just like above to test your knowledge of **Collections** utility class. Use it **reverse()** method to reverse the list.

**Collections.reverse(list);**

### ****Set interface related****

##### **How HashSet store elements?**

You must know that HashMap store key-value pairs, with one condition i.e. keys will be unique. HashSet uses Map’s this feature to ensure uniqueness of elements. In HashSet class, a map declaration is as below:

privatetransientHashMap<E,Object> map;

 //This is added as value for each key

privatestaticfinalObject PRESENT = newObject();

So **when you store a element in HashSet, it stores the element as key in map and “PRESENT” object as value**. (See declaration above).

publicbooleanadd(E e) {

returnmap.put(e, PRESENT)==null;

}

##### **Can a null element added to a TreeSet or HashSet?**

As you see, There is no null check in add() method in previous question. And HashMap also allows one null key, so **one “null” is allowed in HashSet**.

TreeSet uses the same concept as HashSet for internal logic, but uses NavigableMap for storing the elements.

privatetransientNavigableMap<E,Object> m;

 // Dummy value to associate with an Object in the backing Map

**privatestaticfinalObject PRESENT = newObject();**

NavigableMap is subtype of SortedMap which does not allow null keys. So essentially,**TreeSet also does not support null keys**. It will throw NullPointerException if you try to add null element in TreeSet.

##### **What are IdentityHashMap and WeakHashMap?**

**IdentityHashMap** is similar to HashMap except that **it uses reference equality when comparing elements**. IdentityHashMap class is not a widely used Map implementation. While this class implements the Map interface, it intentionally violates Map’s general contract, which mandates the use of the equals() method when comparing objects. IdentityHashMap is designed for use only in the rare cases wherein reference-equality semantics are required.

**WeakHashMap** is an implementation of the Map interface **that stores only weak references to its keys**. Storing only weak references allows a key-value pair to be garbage collected when its key is no longer referenced outside of the WeakHashMap. This class is intended primarily for use with key objects whose equals methods test for object identity using the == operator. Once such a key is discarded it can never be recreated, so it is impossible to do a look-up of that key in a WeakHashMap at some later time and be surprised that its entry has been removed.

##### **Explain ConcurrentHashMap? How it works?**

Taking from java docs:

**A hash table supporting full concurrency of retrievals and adjustable expected concurrency for updates**. This class obeys the same functional specification as Hashtable, and includes versions of methods corresponding to each method of Hashtable. However, even though all operations are thread-safe, retrieval operations do not entail locking, and there is not any support for locking the entire table in a way that prevents all access. This class is fully interoperable with Hashtable in programs that rely on its thread safety but not on its synchronization details.

Read more about how [**concurrent hashmap works and related interview questions**](http://howtodoinjava.com/2013/06/14/popular-hashmap-and-concurrenthashmap-interview-questions/).

##### **How hashmap works?**

The **most important question** which is most likely to be seen in every level of job interviews. You must be very clear on this topic., not only because it is most asked question but also it will open up your mind in further questions related to collection APIs.

Answer to this question is very large and you should read it my post: [**How HashMap works?**](http://howtodoinjava.com/2012/10/09/how-hashmap-works-in-java/) For now, lets remember that HashMap works **on principle of Hashing**. A map by definition is : “An object that maps keys to values”. To store such structure, **it uses an inner class Entry**:

staticclassEntry implementsMap.Entry

{

finalK key;

V value;

Entry next;

finalinthash;

...//More code goes here

}

Here key and value variables are used to store key-value pairs. Whole entry object is stored in an array.

**transientEntry[] table;**

The index of array is calculated on basis on hashcode of Key object. Read more of linked topic.

##### **When to use HashMap or TreeMap?**

HashMap is well known class and all of us know that. So, I will leave this part by saying that it is used to store key-value pairs and allows to perform many operations on such collection of pairs.

TreeMap is special form of HashMap. **It maintains the ordering of keys** which is missing in HashMap class. This ordering is **by default “natural ordering”**. The default ordering can be override by providing an instance of Comparator class, whose compare method will be used to maintain ordering of keys.

Please note that **all keys inserted into the map must implement the Comparable interface** (this is necessary to decide the ordering). Furthermore, all such keys must be mutually comparable: k1.compareTo(k2) must not throw a ClassCastException for any keys k1 and k2 in the map. If the user attempts to put a key into the map that violates this constraint (for example, the user attempts to put a string key into a map whose keys are integers), the put(Object key, Object value) call will throw a ClassCastException.

##### **Difference between HashMap and HashTable?**

There are several differences between HashMap and Hashtable in Java:

* Hashtable is synchronized, whereas HashMap is not.
* Hashtable does not allow null keys or values. HashMap allows one null key and any number of null values.
* The third significant difference between HashMap vs Hashtable is that Iterator in the HashMap is a fail-fast iterator while the enumerator for the Hashtable is not.

##### **Difference between Vector and ArrayList?**

Lets note down the differences:

* All the methods of Vector is synchronized. But, the methods of ArrayList is not synchronized.
* Vector is a Legacy class added in first release of JDK. ArrayList was part of JDK 1.2, when collection framework was introduced in java.
* By default, Vector doubles the size of its array when it is re-sized internally. But, ArrayList increases by half of its size when it is re-sized.

##### **Difference between Iterator and Enumeration?**

Iterators differ from enumerations in three ways:

* Iterators allow the caller to remove elements from the underlying collection during the iteration with its remove() method. You can not add/remove elements from a collection when using enumerator.
* Enumeration is available in legacy classesi.e Vector/Stack etc. whereas Iterator is available in all modern collection classes.
* Another minor difference is that Iterator has improved method names e.g. Enumeration.hasMoreElement() has become Iterator.hasNext(), Enumeration.nextElement() has become Iterator.next() etc.

##### **Difference between Iterator and ListIterator?**

There are three Differences are there:

* We can use Iterator to traverse Set and List and also Map type of Objects. But List Iterator can be used to traverse for List type Objects, but not for Set type of Objects.
* By using Iterator we can retrieve the elements from Collection Object in forward direction only whereas List Iterator, which allows you to traverse in either directions using hasPrevious() and previous() methods.
* ListIterator allows you modify the list using add() remove() methods. Using Iterator you can not add, only remove the elements.

##### **Difference between TreeSet and SortedSet?**

SortedSet is an interface which TreeSet implements. That’ it !!

##### **Difference between ArrayList and LinkedList?**

* LinkedList store elements within a doubly-linked list data structure. ArrayList store elements within a dynamically resizing array.
* LinkedList allows for constant-time insertions or removals, but only sequential access of elements. In other words, you can walk the list forwards or backwards, but grabbing an element in the middle takes time proportional to the size of the list. ArrayLists, on the other hand, allow random access, so you can grab any element in constant time. But adding or removing from anywhere but the end requires shifting all the latter elements over, either to make an opening or fill the gap.
* LinkedList has more memory overhead than ArrayList because in ArrayList each index only holds actual object (data) but in case of LinkedList each node holds both data and address of next and previous node.

##### **How to make a collection read only?**

Use following methods:

* Collections.unmodifiableList(list);
* Collections.unmodifiableSet(set);
* Collections.unmodifiableMap(map);

These methods takes collection parameter and return a new read-only collection with same elements as in original collection.

##### **How to make a collection thread safe?**

Use below methods:

* Collections.synchronizedList(list);
* Collections.synchronizedSet(set);
* Collections.synchronizedMap(map);

Above methods take collection as parameter and return same type of collection which are synchronized and thread safe.

##### **Why there is not method like Iterator.add() to add elements to the collection?**

The sole purpose of an Iterator is to enumerate through a collection. All collections contain the add() method to serve your purpose. There would be no point in adding to an Iterator because the **collection may or may not be ordered**. And **add() method can not have same implementation for ordered and unordered collections**.

##### **What are different ways to iterate over a list?**

You can iterate over a list using following ways:

* Iterator loop
* For loop
* For loop (Advance)
* While loop

##### **What do you understand by iterator fail-fast property?**

**Fail-fast Iterators fail as soon as they realized that structure of Collection has been changed since iteration has begun**. Structural changes means adding, removing or updating any element from collection while one thread is Iterating over that collection.

Fail-fast behavior is implemented by keeping a modification count and if iteration thread realizes the change in modification count it throws ConcurrentModificationException.

##### **What is difference between fail-fast and fail-safe?**

You have understood fail-fast in previous question. **Fail-safe iterators** are just opposite to fail-fast. **They never fail if you modify the underlying collection on which they are iterating**, because they work on clone of Collection instead of original collection and that’s why they are called as fail-safe iterator.

Iterator of CopyOnWriteArrayList is an example of fail-safe Iterator also iterator written by ConcurrentHashMapkeySet is also fail-safe iterator and never throwConcurrentModificationException.

##### **How to avoid ConcurrentModificationException while iterating a collection?**

You should first try to **find another alternative iterator which are fail-safe**. For example if you are using List and you can use ListIterator. If it is legacy collection, you can use enumeration.

If above options are not possible then you can use one of three changes:

* If you are using JDK1.5 or higher then you can use ConcurrentHashMap and CopyOnWriteArrayList classes. It is the recommended approach.
* You can convert the list to an array and then iterate on the array.
* You can lock the list while iterating by putting it in a synchronized block.

Please note that last two approaches will cause a performance hit.

##### **What is UnsupportedOperationException?**

This exception is thrown **on invoked methods which are not supported by actual collection type**.

For example, if you make a read-only list list using “Collections.unmodifiableList(list)” and then call add() or remove() method, what should happen. It should clearly throw UnsupportedOperationException.

##### **Which collection classes provide random access of it’s elements?**

ArrayList, HashMap, TreeMap, Hashtable classes provide random access to it’s elements.

##### 

##### **What is Comparable and Comparator interface?**

In java.all collection which have feature of automatic sorting, uses compare methods to ensure the correct sorting of elements. For example classes which use sorting are TreeSet, TreeMap etc.

**To sort the data elements a class needs to implement Comparator or Comparable interface**. That’s why all Wrapper classes like Integer,Double and String class implements Comparable interface.

**Comparable helps in preserving default natural sorting, whereas Comparator helps in sorting the elements in some special required sorting pattern.**The instance of comparator if passed usually as collection’s constructor argument in supporting collections.

**Java String**

### What is String in Java? String is a data type?

String is a Class in java and defined in java.lang package. It’s not a primitive data type like int and long. String class represents character Strings. String is used in almost all the Java applications and there are some interesting facts we should know about String. String in immutable and final in Java and JVM uses String Pool to store all the String objects.  
Some other interesting things about String is the way we can instantiate a String object using double quotes and overloading of “+” operator for concatenation.

### What are different ways to create String Object?

We can create String object using new operator like any normal java class or we can use double quotes to create a String object. There are several constructors available in String class to get String from char array, byte array, StringBuffer and StringBuilder.

|  |  |
| --- | --- |
| 1  2 | String str = newString("abc");  String str1 = "abc"; |

When we create a String using double quotes, JVM looks in the String pool to find if any other String is stored with same value. If found, it just returns the reference to that String object else it creates a new String object with given value and stores it in the String pool.  
When we use new operator, JVM creates the String object but don’t store it into the String Pool. We can use intern() method to store the String object into String pool or return the reference if there is already a String with equal value present in the pool.

### Write a method to check if input String is Palindrome?

A String is said to be Palindrome if it’s value is same when reversed. For example “aba” is a Palindrome String.  
String class doesn’t provide any method to reverse the String but StringBuffer and StringBuilder class has reverse method that we can use to check if String is palindrome or not.

privatestaticbooleanisPalindrome(String str) {

    if(str == null)

        returnfalse;

    StringBuilderstrBuilder = newStringBuilder(str);

    strBuilder.reverse();

    returnstrBuilder.toString().equals(str);

}

Sometimes interviewer asks not to use any other class to check this, in that case we can compare characters in the String from both ends to find out if it’s palindrome or not.

privatestaticbooleanisPalindromeString(String str) {

    if(str == null)

        returnfalse;

    intlength = str.length();

    System.out.println(length / 2);

    for(inti = 0; i < length / 2; i++) {

         if(str.charAt(i) != str.charAt(length - i - 1))

            returnfalse;

    }

    returntrue;

}

### Write a method that will remove given character from the String?

We can use replaceAll method to replace all the occurance of a String with another String. The important point to note is that it accepts String as argument, so we will use Character class to create String and use it to replace all the characters with empty String.

|  |  |
| --- | --- |
| 1  2  3  4  5 | privatestaticString removeChar(String str, charc) {      if(str == null)          returnnull;      returnstr.replaceAll(Character.toString(c), "");  } |

### How can we make String upper case or lower case?

We can use String class toUpperCase and toLowerCase methods to get the String in all upper case or lower case. These methods have a variant that accepts Locale argument and use that locale rules to convert String to upper or lower case.

### What is String subSequence method?

Java 1.4 introduced CharSequence interface and String implements this interface, this is the only reason for the implementation of subSequence method in String class. Internally it invokes the String substring method.

### How to compare two Strings in java program?

Java String implements Comparable interface and it has two variants of compareTo() methods.

compareTo(String anotherString) method compares the String object with the String argument passed lexicographically. If String object precedes the argument passed, it returns negative integer and if String object follows the argument String passed, it returns positive integer. It returns zero when both the String have same value, in this case equals(String str) method will also return true.

compareToIgnoreCase(String str): This method is similar to the first one, except that it ignores the case. It uses String CASE\_INSENSITIVE\_ORDER Comparator for case insensitive comparison. If the value is zero then equalsIgnoreCase(String str) will also return true.

### How to convert String to char and vice versa?

This is a tricky question because String is a sequence of characters, so we can’t convert it to a single character. We can use usecharAt method to get the character at given index or we can use toCharArray() method to convert String to character array.

### How to convert String to byte array and vice versa?

We can use String getBytes() method to convert String to byte array and we can use String constructor new String(byte[] arr) to convert byte array to String.

### Can we use String in switch case?

This is a tricky question used to check your knowledge of current Java developments. Java 7 extended the capability of switch case to use Strings also, earlier java versions doesn’t support this.  
If you are implementing conditional flow for Strings, you can use if-else conditions and you can use switch case if you are using Java 7 or higher versions.

### 

### Write a program to print all permutations of String?

This is a tricky question and we need to use recursion to find all the permutations of a String, for example “AAB” permutations will be “AAB”, “ABA” and “BAA”.  
We also need to use Set to make sure there are no duplicate values.

.

### Difference between String, StringBuffer and StringBuilder?

String is immutable and final in java, so whenever we do String manipulation, it creates a new String. String manipulations are resource consuming, so java provides two utility classes for String manipulations – StringBuffer and StringBuilder.  
StringBuffer and StringBuilder are mutable classes. StringBuffer operations are thread-safe and synchronized where StringBuilder operations are not thread-safe. So when multiple threads are working on same String, we should use StringBuffer but in single threaded environment we should use StringBuilder.  
StringBuilder performance is fast than StringBuffer because of no overhead of synchronization.

Check this post for extensive details about [String vs StringBuffer vs StringBuilder](http://www.journaldev.com/538/java-interview-question-string-vs-stringbuffer-vs-stringbuilder).  
Read this post for benchmarking of [StringBuffer vs StringBuilder](http://www.journaldev.com/137/stringbuffer-vs-stringbuilder-benchmarking).

### Why String is immutable or final in Java

There are several benefits of String because it’s immutable and final.

* String Pool is possible because String is immutable in java.
* It increases security because any hacker can’t change its value and it’s used for storing sensitive information such as database username, password etc.
* Since String is immutable, it’s safe to use in multi-threading and we don’t need any synchronization.
* Strings are used in java classloader and immutability provides security that correct class is getting loaded by Classloader.

### Why Char array is preferred over String for storing password?

String is immutable in java and stored in String pool. Once it’s created it stays in the pool until unless garbage collected, so even though we are done with password it’s available in memory for longer duration and there is no way to avoid it. It’s a security risk because anyone having access to memory dump can find the password as clear text.  
If we use char array to store password, we can set it to blank once we are done with it. So we can control for how long it’s available in memory that avoids the security threat with String.

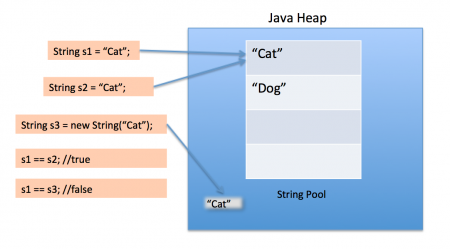
### What is String Pool?

As the name suggests, String Pool is a pool of Strings stored in Java heap memory. We know that String is special class in java and we can create String object using new operator as well as providing values in double quotes.

# What is Java String Pool?

As the name suggests, **String Pool** is a pool of Strings stored in Java heap memory. We know that String is special class in java and we can create String object using new operator as well as providing values in double quotes.

Here is a diagram which clearly explains how String Pool is maintained in java heap space and what happens when we use different ways to create Strings.

[](http://cdn.journaldev.com/wp-content/uploads/2012/11/String-Pool-Java1.png)

String Pool is possible only because [String is immutable in Java](http://www.journaldev.com/802/why-string-is-immutable-or-final-in-java" \o "Why String is immutable or final in Java) and it’s implementation of [String interning](http://en.wikipedia.org/wiki/String_interning) concept. String pool is also example of [Flyweight design pattern](http://www.journaldev.com/1562/flyweight-pattern-in-java-example-tutorial" \o "Flyweight Pattern in Java – Example Tutorial).

String pool helps in saving a lot of space for Java Runtime although it takes more time to create the String.

When we use double quotes to create a String, it first looks for String with same value in the String pool, if found it just returns the reference else it creates a new String in the pool and then returns the reference.

However using *new* operator, we force String class to create a new String object and then we can use intern() method to put it into the pool or refer to other String object from pool having same value.

Here is the java program for the String Pool image:

Output of the above program is:

packagecom.journaldev.util;

### What does String intern() method do?

When the intern method is invoked, if the pool already contains a string equal to this String object as determined by the equals(Object) method, then the string from the pool is returned. Otherwise, this String object is added to the pool and a reference to this String object is returned.  
This method always return a String that has the same contents as this string, but is guaranteed to be from a pool of unique strings.

### Does String is thread-safe in Java?

Strings are immutable, so we can’t change it’s value in program. Hence it’s thread-safe and can be safely used in multi-threaded environment.  
Check this post for [Thread Safety in Java](http://www.journaldev.com/1061/java-synchronization-and-thread-safety-tutorial-with-examples).

### Why String is popular HashMap key in Java?

Since String is immutable, its hashcode is cached at the time of creation and it doesn’t need to be calculated again. This makes it a great candidate for key in a Map and it’s processing is fast than other HashMap key objects. This is why String is mostly used Object as HashMap keys.

**Memory Management**

### How would you improve performance of a Java application?

* Pool valuable system resources like threads, database connections, socket connections etc. Emphasize on reuse of threads from a pool of threads. Creating new threads and discarding them after use can adversely affect performance. Also consider using multi-threading in your single-threaded applications where possible to enhance performance. Optimize the pool sizes based on system and application specifications and requirements. Having too many threads in a pool also can result in performance and scalability problems due to consumption of memory stacks and CPU context switching (i.e. switching between threads as opposed to doing real computation.).
* Minimize network overheads by retrieving several related items simultaneously in one remote invocation if possible. Remote method invocations involve a network round-trip, marshaling and unmarshaling of parameters, which can cause huge performance problems if the remote interface is poorly designed.

Most applications need to retrieve data from and save/update data into one or more databases. Database calls are remote calls over the network. In general data should be lazily loaded (i.e. load only when required as opposed to pre-loading from the database with a view that it can be used later) from a database to conserve memory but there are use cases (i.e. need to make several database calls) where eagerly loading data and caching can improve performance by minimizing network trips to the database. Data can be eagerly loaded with a help of SQL scripts with complex joins or stored procedures and cached using third party frameworks or building your own framework.

### How would you refresh your cache?

You could say that one of the two following strategies can be used:

* Timed cache strategy where the cache can be replenished periodically (i.e. every 30 minutes, every hour etc). This is a simple strategy applicable when it is acceptable to show dirty data at times and also the data in the database does not change very frequently.
* Dirty check strategy where your application is the only one which can mutate (i.e. modify) the data in the database. You can set a "isDirty" flag to true when the data is modified in the database through your application and consequently your cache can be refreshed based on the "isDirty" flag.

### How would you detect and minimize memory leaks in Java?

In Java, memory leaks are caused by poor program design where object references are long lived and the garbage collector is unable to reclaim those objects.

#### Detecting memory leaks:

* Use tools like JProbe, OptimizeItetc to detect memory leaks.
* Use operating system process monitors like task manager on NT systems, ps, vmstat, iostat, netstatetc on UNIX systems.
* Write your own utility class with the help of totalMemory() and freeMemory() methods in the Java Runtime class. Place these calls in your code strategically for pre and post memory recording where you suspect to be causing memory leaks. An even better approach than a utility class is using dynamic proxies or Aspect Oriented Programming (AOP) for pre and post memory recording where you have the control of activating memory measurement only when needed.

#### Minimizing memory leaks:

In Java, typically memory leak occurs when an object of a longer lifecycle has a reference to objects of a short life cycle. This prevents the objects with short life cycle being garbage collected. The developer must remember to remove the references to the short-lived objects from the long-lived objects. Objects with the same life cycle do not cause any issues because the garbage collector is smart enough to deal with the circular references

* Design applications with an object’s life cycle in mind, instead of relying on the clever features of the JVM. Letting go of the object’s reference in one’s own class as soon as possible can mitigate memory problems. Example: myRef = null;
* Unreachable collection objects can magnify a memory leak problem. In Java it is easy to let go of an entire collection by setting the root of the collection to null. The garbage collector will reclaim all the objects (unless some objects are needed elsewhere).
* Use weak references if you are the only one using it. The WeakHashMap is a combination of HashMap and WeakReference. This class can be used for programming problems where you need to have a HashMap of information, but you would like that information to be garbage collected if you are the only one referencing it.
* Free native system resources like AWT frame, files, JNI etc when finished with them. Example: Frame, Dialog, and Graphics classes require that the method dispose() be called on them when they are no longer used, to free up the system resources they reserve.

**Enum**

Java enums are a special Java type used to define collections of constants. An enum type is a special kind of Java class. It can contain constants, methods etc.

**Enum Example**

Here is a simple enum definition:

publicenum Level {

HIGH,

MEDIUM,

LOW

}

You can refer to the constants in the above enum like this:

Level level = Level.HIGH;

Notice how the level variable is of the type Level which is the enum type from the example above. The level variable can take one of the Levelenum constants as value (HIGH, MEDIUM or LOW). In this case level is set to HIGH.

**Enums in if Statements**

Being constants, you will often have to compare a variable pointing to an enum constant against the possible constants in the enum type. Here is how that could look:

Level level = ... //assign some Level constant to it

if( level == Level.HIGH) {

} else if( level == Level.MEDIUM) {

} else if( level == Level.LOW) {

}

This code compares the level variable against each of the possible enum constants in the Levelenum.

If one of the enum values occur more often than the others, checking for that value in the first if-statement will result in better performance, as less comparison on average are executed. This is not a big difference though, unless the comparisons are executed a lot.

**Enums in switch Statements**

If your enum types contain a lot constants and you need to check a variable against the values as shown in the previous section, using a switch statement might be a good idea.

You can use enums in switch statements like this:

Level level = ... //assign some Level constant to it

switch (level) {

case HIGH : ...; break;

case MEDIUM : ...; break;

case LOW : ...; break;

}

Replace the ... with the code to execute if the level variable matches the given Level constant value. The code could be a simple Java operation, a method call etc.

**Enum Iteration**

You can obtain an array of all the possible values of an enum type by calling its static values() method. All enum types get a static values() method automatically by the Java compiler. Here is an example:

for (Level level : Level.values()) {

System.out.println(level);

}

Running this code would print out all the enum values. Here is the output:

HIGH

MEDIUM

LOW

Notice how the names of the constants themselves are printed out.

**Enum Fields**

You can add fields to an enum. Thus, each constant value gets these fields. The field values must be supplied to the constructor of the enum when defining the constants. Here is an example:

publicenum Level {

HIGH (3), //calls constructor with value 3

MEDIUM(2), //calls constructor with value 2

LOW (1) //calls constructor with value 1

; // semicolon needed when fields / methods follow

private final intlevelCode;

public Level(intlevelCode) {

this.levelCode = levelCode;

}

}

**Enum Methods**

You can add methods to an enum too. Here is an example:

publicenum Level {

HIGH (3), //calls constructor with value 3

MEDIUM(2), //calls constructor with value 2

LOW (1) //calls constructor with value 1

; // semicolon needed when fields / methods follow

private final intlevelCode;

Level(intlevelCode) {

this.levelCode = levelCode;

}

**publicintgetLevelCode() {**

**returnthis.levelCode;**

**}**

}

You call en enum method via a reference to one of the constant values. Here is an example:

Level level = Level.HIGH;

System.out.println(level.getLevelCode());

This code would print out the value 3 which is the value of the levelCode field for the constant HIGH.

You are not restricted to simple getter and setter methods. You can also create methods that make calculations based on the field values of the enum constant. If your fields are not declared final you can even modify the values of the fields (although that may not be so good an idea, considering that the enums are supposed to be constants).

**Enum Miscellaneous Details**

Enums extend the java.lang.Enum class implicitly, so your enum types cannot extend another class.

when an enum contain fields and methods, the definition of fields and methods must always come *after* the list of constants in the enum. Additionally, the list of constants must be terminated by a semicolon;

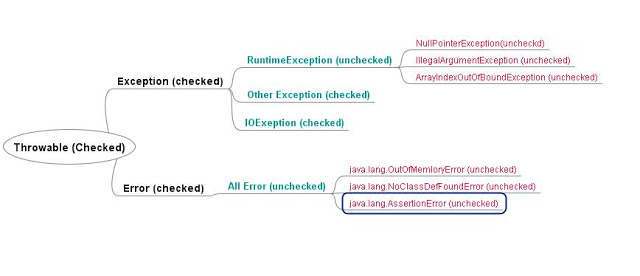
## Difference between Checked vs Unchecked Exception in Java

### What is Checked Exception in Java?

**Checked Exception in Java** is all those Exception which requires being catches and handled during [compile time](http://javarevisited.blogspot.com/2012/03/what-is-static-and-dynamic-binding-in.html). If Compiler doesn’t see try or catch block handling a Checked Exception, it throws Compilation error. Now Which Exception is checked Exception and Why Checked Exception are introduced in first place? All the Exception which are direct sub Class of Exception but not inherit RuntimeExceptionare Checked Exception.

While doing File Programming in C++ I found that most of the time programmer forgets to close file descriptors , which often result in locking of file on OS level. Since Java is introduced after C++, designers of Java thought to ensure such mistakes are not allowed and resources opened are closed properly. To ensure this they introduced Checked Exception. If you see most of [File IO related operation](http://javarevisited.blogspot.com/2011/12/read-and-write-text-file-java.html) comes under IOException which is checked one. Though is a special scenario related to Checked Exception but you can generalize this as, where Java sees an opportunity of failure more, they ensure that programmer provide recovery strategy or at least handle those scenario gracefully.

Since a picture is worth 1000 words I have put together Exception hierarchy in mind map which clearly says which Exceptions are checked and which Exceptions are unchecked.

[](http://4.bp.blogspot.com/-ocSXMYac3Eo/T9IGCdsbHQI/AAAAAAAAAYM/EjDyZa9ykik/s1600/checked+exception+vs+unchecked+exception+in+java.JPG)

### When to use Checked Exception in Java

Knowing **Checked Exception** is not that useful until you know how to use Checked Exception in Java. Java has often been criticized for its Checked Exception strategy, arguments given are that checked Exception adds lot of boiler plate code and makes whole class or function unreadable. Somewhat I agree with this and java also recognize this by introducing [improved Exception handling mechanism in Java7](http://javarevisited.blogspot.com/2011/07/jdk7-multi-cache-block-example-tutorial.html) but Checked Exception does have its real purpose. Following are some scenarios where I would prefer to use Checked Exception to ensure that [Code is Robust and stable](http://javarevisited.blogspot.com/2011/09/how-to-write-production-quality-code.html):

1) All Operation where chances of failure is more e.g. IO Operation, Database Access or Networking operation can be handled with Checked Exception.

2) When you know what to do (i.e. you have alternative) when an Exception occurs, may be as part of Business Process.

3) **Checked Exception** is a reminder by compiler to programmer to handle failure scenario.

### Example of checked Exception in Java API Following are some Examples of Checked Exception in Java library:

IOException

[SQLException](http://javarevisited.blogspot.com/2012/01/javasqlsqlexception-invalid-column.html)

DataAccessException

[ClassNotFoundException](http://javarevisited.blogspot.com/2011/08/classnotfoundexception-in-java-example.html)

InvocationTargetException

### What is Unchecked Exception in Java?

**Unchecked Exception in Java** is those Exceptions whose handling is not verified during Compile time. Unchecked Exceptions mostly arise due to programming errors like accessing method of a null object, accessing element outside an array bonding or invoking method with illegal arguments. In Java, Unchecked Exception is direct sub Class of RuntimeException. *What is major benefit of Unchecked Exception* is that it doesn't reduce code readability and keeps the client code clean.

### When to use UnCheckedException in Java

A good strategy of **Exception handling in Java** is wrapping a checked Exception into **UnCheckedException**. Since most of [Database operation](http://javarevisited.blogspot.com/2011/11/database-transaction-tutorial-example.html) throws SQLException but it’s not good to let SQLException propagate from your DAO layer to up higher on business layer and client code provide exception handling you can handle SQLException in DAO layer and you can wrap the cause in a RuntimeException to propagate through client code. Also as I said earlier unchecked exceptions are mostly programming errors and to catch them is real hard until you do a load test with all possible input and scenario.

### Difference between Checked and Unchecked Exception in Java

Now we have enough information to differentiate Checked Exception with Unchecked Exception:

1) Checked Exception is required to be handled by compile time while Unchecked Exception doesn't.

2) Checked Exception is direct sub-Class of Exception while Unchecked Exception are of RuntimeException.

3) CheckedException represent scenario with higher failure rate while UnCheckedException are mostly programming mistakes.

### Example of unchecked Exception in Java API

Here are few **examples of Unchecked Exception in Java** library:

NullPointerException

ArrayIndexOutOfBound

IllegalArgumentException

IllegalStateException

## What are annotations?

Java annotations are used to provide meta data for your Java code.

**What is transient?**

Transient variables are not serialized during Serialization process and initialize with default value during deserialization

**Why java is platform independent?**

Java code can be run on multiple platforms e.g.Windows,Linux,SunSolaris,Mac/OS etc. Java code is compiled by the compiler and converted into bytecode.Thisbytecode is a platform independent code because it can be run on multiple platforms i.e. Write Once and Run Anywhere(WORA).

### Can constructor perform other tasks instead of initialization?

Yes, like object creation, starting a thread, calling method etc. You can perform any operation in the constructor as you perform in the method.

**Is-A relation**

Inheritance represents the **IS-A relationship**.Programmer is the subclass and Employee is the superclass. Relationship between two classes is **Programmer IS-A Employee**.It means that Programmer is a type of Employee.

**HAS-A relation**

**Aggregation** represents HAS-A relationship

Consider a situation, Employee object contains many informations such as id, name, emailId etc. It contains one more object named address, which contains its own informations such as city, state, country, zipcode etc. as given below.

1. **class** Employee{
2. **int** id;
3. String name;
4. Address address;//Address is a class
5. ...
6. }

class Employee{

int id;

String name;

Address address;//Address is a class

...

}

In such case, Employee has an entity reference address, so relationship is Employee HAS-A address.

### Can we override static method?

|  |
| --- |
| No, static method cannot be overridden. It can be proved by runtime polymorphism so we will learn it later. |

### Why we cannot override static method?

because static method is bound with class whereas instance method is bound with object. Static belongs to class area and instance belongs to heap area.

**Super**keyword

1. super is used to refer immediate parent class instance variable.

System.out.println(super.speed);//will print speed of Vehicle now

1. super() is used to invoke immediate parent class constructor.

super();//will invoke parent class constructor

1. super is used to invoke immediate parent class method.

super.message();//will invoke parent class message() method

**final keyword**

### 1) final variable

If you make any variable as final, you cannot change the value of final variable(It will be constant).

final int speedlimit=90;

### 2) final method

If you make any method as final, you cannot override it.

### 3) final class

If you make any class as final, you cannot extend it.

### Run Time Polymorphism

Class Car{

Int price = 100;

Public void **printName**(){

Syso(“print name from **CAR** class”);

}

Public void **printPrice**(){

Syso(“price is ”+**price**);

}

}

Class Maruthi extends Car{

Int price 200;

Public void **printName**(){

Syso(“print name from **Maruthi** class”);

}

Public void **printPrice**(){

Syso(“price is ”+**price**);

}

Public void **printMaruthi**(){

Syso(“Maruthi”);

}

}

Class test{

Public static void main(String str[]){

Car obj = new Maruthi();

Obj.printName();//print name from **Maruthi** class

Obj.printPrice();//price is **200**

Syso(obj.price);//**100**

Obj.printMaruthi();//**Comp Error**

}

}

## Singleton

public class Singleton {

  private static Singleton singleInstance;

  private Singleton() {}

  public static Singleton getSingleInstance() {

    if (singleInstance == null) {

      synchronized (Singleton.class) {

        if (singleInstance == null) {

          singleInstance = new Singleton();

        }

      }

    }

    returnsingleInstance;

  }

**What is runtime polymorphism or dynamic method dispatch?**

In Java, runtime polymorphism or dynamic method dispatch is a process in which a call to an overridden method is resolved at runtime rather than at compile-time. In this process, an overridden method is called through the reference variable of a superclass. The determination of the method to be called is based on the object being referred to by the reference variable.

**What is Dynamic Binding?**

Binding refers to the linking of a procedure call to the code to be executed in response to the call. Dynamic binding (also known as late binding) means that the code associated with a given procedure call is not known until the time of the call at run-time. It is associated with polymorphism and inheritance.

**Can there be an abstract class with no abstract methods in it?**

Yes, there can be an abstract class without abstract methods.

**How are this() and super() used with constructors?**

* Constructors use *this* to refer to another constructor in the same class with a different parameter list.
* Constructors use *super* to invoke the superclass's constructor. If a constructor uses *super*, it must use it in the first line; otherwise, the compiler will complain.

**What are the differences between Class Methods and Instance Methods?**

|  |  |
| --- | --- |
| **Class Methods** | **Instance Methods** |
| Class methods are methods which are declared as static. The method can be called without creating an instance of the class | Instance methods on the other hand require an instance of the class to exist before they can be called, so an instance of a class needs to be created by using the new keyword. Instance methods operate on specific instances of classes. |
| Class methods can only operate on class members and not on instance members as class methods are unaware of instance members. | Instance methods of the class can also not be called from within a class method unless they are being called on an instance of that class. |
| Class methods are methods which are declared as static. The method can be called without creating an  instance of the class. | Instance methods are not declared as static. |

**What are the differences between the Comparable and Comparator interfaces ?**

|  |  |
| --- | --- |
| **Comparable** | **Comparato** |
| It uses the *compareTo()* method.  *intobjectOne.compareTo(objectTwo).* | t uses the *compare()* method.  *int compare(ObjOne, ObjTwo)* |
| It is necessary to modify the class whose instance is going to be sorted. | A separate class can be created in order to sort the instances. |
| Only one sort sequence can be created. | Many sort sequences can be created. |
| It is frequently used by the API classes. | It used by third-party classes to sort instances. |

**How to create a immutable object in Java? Count all benefits?**

An immutable class is one whose state can not be changed once created. Here, state of object essentially means the values stored in instance variable in class whether they are primitive types or reference types.

To make a class immutable, below steps needs to be followed:

1. Don’t provide “setter” methods or methods that modify fields or objects referred to by fields. Setter methods are meant to change the state of object and this is what we want to prevent here.
2. Make all fields final and private. Fields declared private will not be accessible outside the class and making them final will ensure the even accidentally you can not change them.
3. Don’t allow subclasses to override methods. The simplest way to do this is to declare the class as final. Final classes in java can not be overridden.
4. Always remember that your instance variables will be either mutable or immutable. Identify them and return new objects with copied content for all mutable objects (object references). Immutable variables (primitive types) can be returned safely without extra effort.

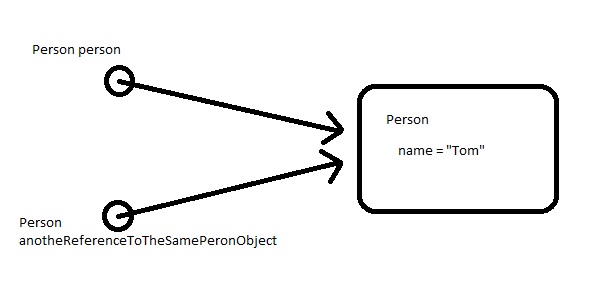
Also, you should memorize following benefits of immutable class. You might need them during interview. Immutable classes --

* are simple to construct, test, and use
* are automatically thread-safe and have no synchronization issues
* do not need a copy constructor
* do not need an implementation of clone
* allow hashCode to use lazy initialization, and to cache its return value
* do not need to be copied defensively when used as a field
* make good Map keys and Set elements (these objects must not change state while in the collection)
* have their class invariant established once upon construction, and it never needs to be checked again
* always have “failure atomicity” (a term used by Joshua Bloch) : if an immutable object throws an exception, it’s never left in an undesirable or indeterminate state.

**Is Java Pass by Reference or Pass by Value?**

The Java Spec says that ***everything in Java is pass-by-value***. There is no such thing as “pass-by-reference” in Java. These terms are associated with method calling and passing variables as method parameters. Well, primitive types are always pass by value without any confusion. But, the concept should be understood in context of method parameter of complex types.

In java, when we pass a reference of complex types as any method parameters, always the memory address is copied to new reference variable bit by bit. See in below picture:

[](http://howtodoinjava.files.wordpress.com/2013/03/pass-by-value.jpg)In above example, address bits of first instance are copied to another reference variable, thus resulting both references to point a single memory location where actual object is stored. Remember, making another reference to null will not make first reference also null. But, changing state from either reference variable have impact seen in other reference also.

**What is the use of the finally block? Is finally block in Java guaranteed to be called? When finally block is NOT called?**

The finally block always executes when the try block exits. This ensures that the finally block is executed even if an unexpected exception occurs. But finally is useful for more than just exception handling — it allows having cleanup code accidentally bypassed by a return, continue, or break. Putting cleanup code in a finally block is always a good practice, even when no exceptions are anticipated.

If the JVM exits while the try or catch code is being executed, then the finally block may not execute. Likewise, if the thread executing the try or catch code is interrupted or killed, the finally block may not execute even though the application as a whole continues.

**Why there are two Date classes; one in java.util package and another in java.sql?**

A java.util.Date represents date and time of day, a java.sql.Date only represents a date. The complement of java.sql.Date is java.sql.Time, which only represents a time of day.  
The java.sql.Date is a subclass (an extension) of java.util.Date. So, what changed in java.sql.Date:

-- toString() generates a different string representation: yyyy-mm-dd  
-- a static valueOf(String) methods to create a Date from a String with above representation  
-- the getters and setter for hours, minutes and seconds are deprecated

The java.sql.Date class is used with JDBC and it was intended to not have a time part, that is, hours, minutes, seconds, and milliseconds should be zero… but this is not enforced by the class.

**Explain marker interfaces?**

The marker interface pattern is a design pattern in computer science, used with languages that **provide run-time type information about objects**. It provides **a means to associate metadata with a class where the language does not have explicit support for such metadata.** In java, it is used as interfaces with no method specified.

A good example of use of marker interface in java is [Serializable](http://howtodoinjava.com/2012/11/21/a-mini-guide-for-implementing-serializable-interface-in-java/) interface. A class implements this interface to indicate that its non-transient data members can be written to a byte steam or file system.

A *major problem* with marker interfaces is that an interface defines a contract for implementing classes, and that contract is inherited by all subclasses. This means that you cannot “un-implement” a marker. In the example given, if you create a subclass that you do not want to serialize (perhaps because it depends on transient state), you must resort to explicitly throwing NotSerializableException.

**Why main() in java is declared as public static void?**

*Why public*? main method is public so that it can be accessible everywhere and to every object which may desire to use it for launching the application. Here, i am not saying that JDK/JRE had similar reasons because java.exe or javaw.exe (for windows) use Java Native Interface (JNI) calls to invoke method, so they can have invoked it either way irrespective of any access modifier.

*Why static*? Lets suppose we do not have main method as static. Now, to invoke any method you need an instance of it. Right? Java can have overloaded constructors, we all know. Now, which one should be used and from where the parameters for overloaded constructors will come.

*Why void*? Then there is no use of returning any value to JVM, who actually invokes this method. The only thing application would like to communicate to invoking process is: normal or abnormal termination. This is already possible using  System.exit(int). A non-zero value means abnormal termination otherwise everything was fine.

**What is the difference between creating String as new() and literal?**

When we create string with new() it’s created in heap and also added into string pool, while String created using literal are created in String pool only which exists in Perm area of heap.

**How does substring () inside String works?**

String in java are like any other programming language, a sequence of characters. This is more like a utility class to work on that char sequence. This char sequence is maintained in following variable:

/\*\* The value is used for character storage. \*/  
**private final char value[];**

To access this array in different scenarios, following variables are used:

/\*\* The offset is the first index of the storage that is used. \*/  
**private final int offset;**

/\*\* The count is the number of characters in the String. \*/  
**private final int count;**

Whenever we create a substring from any existing string instance, substring() method only set’s the new values of offset and count variables. The internal char array is unchanged. This is a possible source of memory leak if substring() method is used without care.

**Difference between interfaces and abstract classes?**

This is very common question if you are appearing interview for junior level programmer. Well, most noticeable differences are as below:

* Variables declared in a Java interface is by default final. An  abstract class may contain non-final variables.
* Java interface are implicitly abstract and cannot have implementations. A Java abstract class can have instance methods that implements a default behavior.
* Members of a Java interface are public by default. A Java abstract class can have the usual flavors of class members like private, protected.
* Java interface should be implemented using keyword “implements”; A Java abstract class should be extended using keyword “extends”.
* A Java class can implement multiple interfaces but it can extend only one abstract class.
* Interface is absolutely abstract and cannot be instantiated; A Java abstract class also cannot be instantiated, but can be invoked if a main() exists.
* Abstract class are slightly faster than interface because interface involves a search before calling any overridden method in Java. This is not a significant difference in most of cases but if you are writing a time critical application than you may not want to leave any stone unturned.

**When do you override hashCode() and equals()?**

hashCode() and equals() methods have been defined in Object class which is parent class for java objects. For this reason, all java objects inherit a default implementation of these methods.

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 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 simply check the object references of two objects to verify their equality.

Note that it is generally necessary to override the hashCode method whenever this method is overridden, so as to maintain the general contract for the hashCode method, which states that equal objects must have equal hash codes.

equals()  must define an equality relation (it must be reflexive, symmetric, and transitive). In addition, it must be consistent (if the objects are not modified, then it must keep returning the same value). Furthermore, o.equals(null) must always return false.

hashCode()  must also be consistent (if the object is not modified in terms of equals(), it must keep returning the same value).

The relation between the two methods is:

Whenever a.equals(b), then a.hashCode() must be same as b.hashCode().

**Why finalize() method should be avoided?**

We all know the basic statement that finalize() method is called by garbage collector thread before reclaiming the memory allocated to the object. See [this program](http://howtodoinjava.com/2012/10/31/why-not-to-use-finalize-method-in-java/) which prove that finalize() invocation is not guaranteed at all. Other reasons can be:

1) finalize() methods do not work in chaining like constructors. It means like when you call a constructor then constructors of all super classes will be invokes implicitly. But, in case of finalize methods, this is not followed. Super class’s finalize() should be called explicitly.

2) Any Exception thrown by finalize method is ignored by GC thread and it will not be propagated further, in fact it will not be logged in your log files. So bad, isn’t it?

3) Also, There is some performance penalty when finalize() in included in your class. In Effective java (2nd edition ) Joshua bloch says,

“Oh, and one more thing: there is a severe performance penalty for using finalizers. On my machine, the time to create and destroy a simple object is about 5.6 ns.  
Adding a finalizer increases the time to 2,400 ns. In other words, it is about 430 times slower to create and destroy objects with finalizers.”

**Why HashMap should not be used in multithreaded environment? Can it cause infinite loop as well?**

We know that HashMap is non-synchronized collection where as its synchronized counter-part is HashTable. So, when you are accessing the collection in multithreaded environment and all threads are accessing a single instance of collection, then its safer to use HashTable for various obvious reasons e.g. to avoid dirty reads and to maintain data consistency. In worst case, this mutithreaded environment can result in infinite loop as well.

Yes, it is true. HashMap.get() can cause an infinite loop. Lets see how??

If you look at the source code HashMap.get(Object key) method, it looks like this:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | publicObject get(Object key) {      Object k = maskNull(key);      inthash = hash(k);      inti = indexFor(hash, table.length);      Entry e = table[i];      while(true) {          if(e == null)              returne;          if(e.hash == hash &amp;&amp; eq(k, e.key))              returne.value;          e = e.next;      }  } |

while(true) can always be a victim of infinite loop at runtime in multithreaded environment, IF, somehow e.next can point to itself. This will result in infinite loop. But, how e.next will point to itself (i.e. e).

This can happen in void transfer(Entry[] newTable) method, which is invoked at time the HashMap resizing is done.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | do{      Entry next = e.next;      inti = indexFor(e.hash, newCapacity);      e.next = newTable[i];      newTable[i] = e;      e = next;  } while(e != null); |

This piece of code is prone to produce above condition, if resizing happen and at the same time other threads tried to modify the map instance.

Only way to avoid this scenario is to use synchronization in code, or better, used synchronized collection.

**Explain abstraction and encapsulation? How are they related?**

In simple words: “***Abstraction****captures only those details about an object that are relevant to the current  perspective.*”  
In object-oriented programming theory, abstraction involves the facility to define objects that represent abstract “actors” that can perform work, report on and change their state, and “communicate” with other objects in the system.

Abstraction in any programming language works in many ways. It can be seen from creating subroutines to defining interfaces for making low level language calls. Some abstractions try to limit the breadth of concepts a programmer needs, by completely hiding the abstractions they in turn are built on, e.g. design patterns.

Typically abstraction can be seen in two ways:

**Data abstraction** is the way to create complex data types and exposing only meaningful operations to interact with data type, where as hiding all the implementation details from outside works. **Control abstraction** is the process of identifying all such statements and expose them as a unit of work. We normally use this feature when we create a function to perform any work.

**Wrapping data and methods within classes in combination with implementation hiding (through access control) is often called encapsulation.** The result is a data type with characteristics and behaviors. Encapsulation essentially has both i.e. information hiding and implementation hiding.

“***Whatever changes, encapsulate it***“. It has been quoted as a famous design principle. For that matter in any class, changes can happen in data in runtime and changes in implementation can happen in future releases. So, encapsulation applies to both i.e. data as well as implementation.

SO, they can relate like following :

-- Abstraction is more about ‘What‘ a class can do. [Idea]  
-- Encapsulation is more about ‘How‘ to achieve that functionality. [Implementation]

**Difference between interfaces and abstract classes?**

Basic differences can be counted as follows:

* An interface cannot implement any methods, whereas an abstract class can
* A class can implement many interfaces but can have only one superclass (abstract or not)
* An interface is not part of the class hierarchy. Unrelated classes can implement the same interface

You should remember that : “When you can fully describe the concept in terms of “***what it does***” without needing to specify any of “***how it does***“, then you should use an interface.  If you need to include some implementation details, then you will need to represent your concept in an abstract class.”

Also, if i talk differently : Are there many classes that can be “*grouped together*” and described by one noun? If so, have an abstract class by the name of this noun, and inherit the classes from it. For example Cat and Dog can both inherit from abstract class Animal, and this abstract base class will implement a method void Breathe() which all animals will thus do in exactly the same fashion.

What kinds of verbs can be applied to my class, that might in general also be applied to others? Create an interface for each of these verbs. For example, All animals can be fed, so I will create an interface called IFeedable and have Animal implement that. Only Dog and Horse are nice enough though to implement ILikeable, but some are not.

As said by someone: the main difference is where you want your implementation. By creating an interface, you can move your implementation to any class that implements your interface. By creating an abstract class, you can share implementation for all derived classes in one central place, and avoid lots of bad things like code duplication.

**How StringBuffer save the memory?**

A String is implemented as an immutable object; that is, when you initially decide to put something into a String object, the JVM allocates a fixed-width array of exactly the size of your initial value. This is then treated as a constant inside the JVM, which allows for very significant performance savings in the case where the String’s value is not changed. However, if you decide to change the String’s contents in any way, what the JVM then essentially does is copy the contents of the original String into a temporary space, make your changes, then save those changes into a whole new memory array. Thus, making changes to a String’s value after initialization is a fairly expensive operation.

StringBuffer, on the other hand, is implemented as a dynamically -- growable array inside the JVM, which means that any change operation can occur on the existing memory location, with new memory allocated only as-needed. However, there is no opportunity for the JVM to make optimizations around the StringBuffer, since its contents are assumed to be changeable at any instance.

**Why wait and notify is declared in Object class instead of Thread?**

Wait , notify , notifyAll methods are only required when you want your threads to access a shared resource and a shared resource could be any java object which is on the heap. So, these methods are defined on the core Object class so that each object has control of allowing Threads to wait on it’s monitor. Java doesn’t have any special object which is used for sharing a common resource. No such data structure is defined.So, onus is given on the Object class to be able to become shared resource providing it will helper methods like wait(),notify() and notifyAll().Java is based on Hoare’s monitors idea. In Java all object has a monitor. Threads waits on monitors so, to perform a wait, we need 2 parameters:

-- a Thread  
-- a monitor (any object)

In the Java design, the thread can not be specified, it is always the current thread running the code. However, we can specify the monitor (which is the object we call wait on). This is a good design, because if we could make any other thread to wait on a desired monitor, this would lead to an “intrusion”, posing difficulties on designing /programming concurrent programs. Remember that in Java all operations that are intrusive in another thread’s execution are deprecated (e.g. stop()).

**Write Java program to create deadlock in Java and fix it ?**

In java, a deadlock is a situation where minimum two threads are holding lock on some different resource, and both are waiting for other resource to complete its task. And, none is able to leave the lock on resource it is holding.

To create a deadlock situation, and to know the solution : read full post “[Writing a deadlock and resolving in java](http://howtodoinjava.com/2012/10/16/writing-a-deadlock-and-resolving-in-java/)“.

**What happens if your Serializable class contains a member which is not serializable? How do you fix it?**

In this case, **NotSerializableException** will be thrown at runtime. To fix this issue, a very simple solution is to mark such fields transient. It means these fields will not be serialized. If you want to save the state of these fields as well then you should consider reference variables which already implements serializable interface.

You also might need to use readResolve() and writeResolve() methods. Lets summarize this:

* First, make your non-serialisable field transient.
* In writeObject, first call defaultWriteObject on the stream to store all the non-transient fields, then call other methods to serialise the individual properties of your non-serialisable object.
* In readObject, first call defaultReadObject on the stream to read back all the non-transient fields, then call other methods (corresponding to the ones you added to writeObject) to deserialise your non-serialisable object.

Also, i will highly recommend to read [**full guide on serialization in java**](http://howtodoinjava.com/2012/11/21/a-mini-guide-for-implementing-serializable-interface-in-java/).

**Explain transient and volatile keywords in java?**

“*The****transient****keyword in Java is used to indicate that a field should not be serialized.*” According to language specification: Variables may be marked transient to indicate that they are not part of the persistent state of an object. For example, you may have fields that are derived from other fields, and should only be done so programmatically, rather than having the state be persisted via serialization.

For example, in class BankPayment.java fields like principal and rate can be serialized while interest can be calculated any time even after de-serialization.

If we recall, each thread in java has its own local memory space as well and it does all read/write operations in its local memory. Once all operations are done, it write back the modified state of variable in main memory from where all threads access this variable. Normally, this is the default flow inside JVM. But, the volatile modifier tells the JVM that a thread accessing the variable must always reconcile its own private copy of the variable with the master copy in memory. It means every time thread want to read the state of variable, it must flush its local memory state and update the variable from main memory.

**Volatile** is most useful in lock-free algorithms. You mark the variable holding shared data as volatile when you are not using locking to access that variable and you want changes made by one thread to be visible in another, or you want to create a “happens-after” relation to ensure that computation is not re-ordered, again, to ensure changes become visible at the appropriate time.

The volatile should be used to safely publish immutable objects in a multi-threaded Environment. Declaring a field like public volatile ImmutableObject foo secures that all threads always see the currently available instance reference.

**Difference between Iterator and ListIterator?**

We can use Iterator to traverse a Set or a List or a Map. But ListIterator can only be used to traverse a List only. Other differences can be listed as below.

You can

1. iterate backwards.
2. obtain the index at any point.
3. add a new value at any point.
4. set a new value at that point.

**Deep copy and shallow copy?**

A clone is an exact copy of the original. In java, it essentially means the ability to create an object with similar state as the original object. The clone() method provides this functionality.

Shallow copies duplicate as little as possible.  By default, java cloning is shallow copy or ‘field by field copy’ i.e. as the Object class does not have idea about the structure of class on which clone() method will be invoked. So, JVM when called for cloning, do following things:

1) If the class has only primitive data type members then a completely new copy of the object will be created and the reference to the new object copy will be returned.

2) If the class contains members of any class type then only the object references to those members are copied and hence the member references in both the original object as well as the cloned object refer to the same object.

Deep copies duplicate everything. A deep copy of a collection is two collections with all of the elements in the original collection duplicated. Here, we want a clone which is independent of original and making changes in clone should not affect original.

*Deep cloning requires satisfaction of following rules.*

1. No need to separately copy primitives.
2. All the member classes in original class should support cloning and in clone method of original class in context should call super.clone() on all member classes.
3. If any member class does not support cloning then in clone method, one must create a new instance of that member class and copy all its attributes one by one to new member class object. This new member class object will be set in cloned object.

[Read more about cloning here](http://howtodoinjava.com/2012/11/08/a-guide-to-object-cloning-in-java/).

**What is synchronization? Object level locking and class level locking?**

***Synchronization***refers to multi-threading. A synchronized block of code can only be executed by one thread at a time. Java supports multiple threads to be executed. This may cause two or more threads to access the same fields or objects. Synchronization is a process which keeps all concurrent threads in execution to be in synch. Synchronization avoids memory consistence errors caused due to inconsistent view of shared memory. When a method is declared as synchronized; the thread holds the monitor for that method’s object If another thread is executing the synchronized method, your thread is blocked until that thread releases the monitor.

Synchronization in java is achieved using synchronized keyword. You can use synchronized keyword in your class on defined methods or blocks. Keyword can not be used with variables or attributes in class definition.

***Object level locking*** is mechanism when you want to synchronize a non-static method or non-static code block such that only one thread will be able to execute the code block on given instance of the class. This should always be done to make instance level data thread safe.

***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.

[Read more about synchronization here.](http://howtodoinjava.com/2013/03/08/thread-synchronization-object-level-locking-and-class-level-locking/)

**Difference between sleep() and wait()?**

sleep() is a method which is used to hold the process for few seconds or the time you wanted but in case of wait() method thread goes in waiting state and it won’t come back automatically until we call the notify() or notifyAll().

The major difference is that wait() releases the lock or monitor while sleep() doesn’t releases any lock or monitor while waiting. Wait is used for inter-thread communication while sleep is used to introduce pause on execution, generally.

Thread.sleep() sends the current thread into the “Not Runnable” state for some amount of time. The thread keeps the monitors it has aquired — i.e. if the thread is currently in a synchronized block or method no other thread can enter this block or method. If another thread calls t.interrupt() it will wake up the sleeping thread. Note that sleep is a static method, which means that it always affects the current thread (the one that is executing the sleep method). A common mistake is to call t.sleep() where t is a different thread; even then, it is the current thread that will sleep, not the t thread.

object.wait() sends the current thread into the “Not Runnable” state, like sleep(), but with a twist. Wait is called on a object, not a thread; we call this object the “lock object.” Before lock.wait() is called, the current thread must synchronize on the lock object; wait() then releases this lock, and adds the thread to the “wait list” associated with the lock. Later, another thread can synchronize on the same lock object and call lock.notify(). This wakes up the original, waiting thread. Basically, wait()/notify() is like sleep()/interrupt(), only the active thread does not need a direct pointer to the sleeping thread, but only to the shared lock object.

[Read the difference in detail here.](http://howtodoinjava.com/2013/03/08/difference-between-sleep-and-wait/)

**Can you assign null to this reference variable?**

NO. You can’t. In java, left hand side of an assignment statement must be a variable. ‘this’ is a special keyword which represent the current instance always. This is not any variable.

Similarly, null can not be assigned to ‘super’ or any such keyword for that matter.

**What if the difference between && and &??**

& is bitwise and && is logical.

* & evaluates both sides of the operation.
* && evaluates the left side of the operation, if it’s true, it continues and evaluates the right side.

[Read here for deep understanding.](http://en.wikipedia.org/wiki/Bitwise_operation)

**How to override equals and hashCode() methods?**

hashCode() and equals() methods have been defined in Object class which is parent class for java objects. For this reason, all java objects inherit a default implementation of these methods.

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 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 simply check the object references of two objects to verify their equality.

Below are the important points to keep remember while overriding these functions.

1. Always use same attributes of an object to generate hashCode() and equals() both. As in our case, we have used employee id.
2. equals() must be consistent (if the objects are not modified, then it must keep returning the same value).
3. Whenever a.equals(b), then a.hashCode() must be same as b.hashCode().
4. If you override one, then you should override the other.

[Read more interesting facts and how to guide here.](http://howtodoinjava.com/2012/10/09/working-with-hashcode-and-equals-methods-in-java/)

**What is garbage collection? Can we enforce it?**

Garbage collection is an automatic memory management feature in many modern programming languages, such as Java and languages in the .NET framework. Languages that use garbage collection are often interpreted or run within a virtual machine like the JVM. In each case, the environment that runs the code is also responsible for garbage collection. A GC has two goals: any unused memory should be freed, and no memory should be freed unless the program will not use it anymore.

Can you force garbage collection?? Nope, System.gc() is as close as you can get. Your best option is to call System.gc() which simply is a hint to the garbage collector that you want it to do a collection. There is no way to force and immediate collection though as the garbage collector is non-deterministic. Also, under the documentation for OutOfMemoryError it declares that it will not be thrown unless the VM has failed to reclaim memory following a full garbage collection. So if you keep allocating memory until you get the error, you will have already forced a full garbage collection.

[Read more about garbage collection here.](http://howtodoinjava.com/2012/11/11/revisiting-memory-management-and-garbage-collection-mechanisms-in-java/)

**What is native keyword? Explain in detail?**

The native keyword is applied to a method to indicate that the method is implemented in native code using JNI. It marks a method, that it will be implemented in other languages, not in Java.

Native methods were used in the past to write performance critical sections but with Java getting faster this is now less common. Native methods are currently needed when

* You need to call a library from Java that is written 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.

The downsides of using native code libraries are also significant:

1. JNI / JNA have a tendency to destabilize the JVM, especially if you try to do something complicated. If your native code gets native code memory management wrong, there’s a chance that you will crash the JVM. If your native code is non-reentrant and gets called from more than one Java thread, bad things will happen … sporadically. And so on.
2. Java with native code is harder to debug than pure Java or pure C/C++.
3. Native code can introduce significant platform dependencies / issues for an otherwise platform independent Java app.
4. Native code requires a separate build framework, and that may have platform / portability issues as well.

**What is serialization? Explain the catches?**

In computer science, in the context of data storage and transmission, serialization is the process of translating data structures or object state into a format that can be stored  and “resurrected” later in the same or another computer environment.  When the resulting series of bits is reread according to the serialization format, it can be used to create a semantically identical clone of the original object.

Java provides automatic serialization which requires that the object be marked by implementing the java.io.Serializable interface. Implementing the interface marks the class as “okay to serialize,” and Java then handles serialization internally. There are no serialization methods defined on the Serializable interface, but a serializable class can optionally define methods with certain special names and signatures that if defined, will be called as part of the serialization/deserialization process.

Once an object is serialized, changes in its class break the de-serialization process. To identify the future changes in your class which will be compatible and others which will prove incompatible, please read the[**full guide here**](http://howtodoinjava.com/2012/11/21/a-mini-guide-for-implementing-serializable-interface-in-java/). In short, I am listing down here:

**Incompatible changes**

* Deleting fields
* Moving classes up or down the hierarchy
* Changing a non-static field to static or a non-transient field to transient
* Changing the declared type of a primitive field
* Changing the writeObject or readObject method so that it no longer writes or reads the default field data
* Changing a class from Serializable to Externalizable or vice-versa
* Changing a class from a non-enum type to an enum type or vice versa
* Removing either Serializable or Externalizable
* Adding the writeReplace or readResolve method to a class

**Compatible changes**

* Adding fields
* Adding/ Removing classes
* Adding writeObject/readObject methods [defaultReadObject or defaultWriteObject should be called first]
* Removing writeObject/readObject methods
* Adding java.io.Serializable
* Changing the access to a field
* Changing a field from static to non-static or transient to non transient

Implementing Comparable interface and Testing our code

Comparable interface provides one method [*compareTo(T o)*](http://docs.oracle.com/javase/6/docs/api/java/lang/Comparable.html#compareTo%28T%29) to implement in any class so that two instances of that class can be compared. Signature of method is:

|  |  |
| --- | --- |
| 1 | publicintcompareTo(T o); |

*Here, out of two instances to compare, one is instance itself on which method will be invoked, and other is passed as parameter o.*

Lets see how our Employee class will look after implementing Comparable interface.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36 | packagecorejava.compare;    publicclassEmployee implementsComparable<Employee> {      privateintid = -1;      privateString firstName = null;      privateString lastName = null;      privateintage = -1;        publicEmployee(intid, String fName, String lName, intage) {          this.id = id;          this.firstName = fName;          this.lastName = lName;          this.age = age;      }        @Override      publicintcompareTo(Employee o) {          returnthis.id - o.id;      }        @Override      publicString toString() {          return"Employee : "+ id + " - "+ firstName + " - "+ lastName          + " - "+ age + "n";      }        publicintgetId() {          returnid;      }        publicvoidsetId(intid) {          this.id = id;      }        // Other accessor methods  } |

Default way to sort a list of employees, in our case, is by their id. Whatever, your default sorting order is, use in compare() method.  
In implemented compare() method, we have simply returned the difference in employee ids of two instances. Two equal employee ids will return zero, indicating same object.

Lets test our compare() method.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27 | packagecorejava.compare;    importjava.util.ArrayList;  importjava.util.Collections;  importjava.util.List;    publicclassTestSorting {      publicstaticvoidmain(String[] args) {          Employee e1 = newEmployee(1, "aTestName", "dLastName", 34);          Employee e2 = newEmployee(2, "nTestName", "pLastName", 30);          Employee e3 = newEmployee(3, "kTestName", "sLastName", 31);          Employee e4 = newEmployee(4, "dTestName", "zLastName", 25);            List<Employee> employees = newArrayList<Employee>();          employees.add(e2);          employees.add(e3);          employees.add(e1);          employees.add(e4);            // UnSorted List          System.out.println(employees);            Collections.sort(employees);          // Default Sorting by employee id          System.out.println(employees);      }  } |

In above program, first print statement prints an unsorted list of employees and in second print statement, employees are sorted by their employee id.

Comparator

So, now we can sort a collection of employees by their id. Now lets consider a case where we want to sort employees list based on some user input which is essentially sorting field i.e. sometime he want to sort by first name, sometimes by even age also.

This can be achieved by [jquery](http://jquery.com/) plugins easily, but what if browser has disables the java script. You will have to sort the list on server side only to not break the application functionality.

Here comes the Comparators to rescue you. A comparator can be used to sort a collection of instances on some particular basis. **To sort of different fields, we need multiple comparators.**

Adding different Comparator interface implementations

As discussed in last section, we need to have multiple implementations for different sorting cases. Lets write them one by one:

**First name sorter**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | packagecorejava.compare;    importjava.util.Comparator;    publicclassFirstNameSorter implementsComparator<Employee>{    @Override  publicintcompare(Employee o1, Employee o2) {  returno1.getFirstName().compareTo(o2.getFirstName());  }  } |

**Last name sorter**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | packagecorejava.compare;    importjava.util.Comparator;    publicclassLastNameSorter implementsComparator<Employee> {        @Override      publicintcompare(Employee o1, Employee o2) {          returno1.getLastName().compareTo(o2.getLastName());      }    } |

**Age sorter**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | packagecorejava.compare;    importjava.util.Comparator;    publicclassAgeSorter implementsComparator<Employee> {      @Override      publicintcompare(Employee o1, Employee o2) {          returno1.getAge() - o2.getAge();      }  } |

Testing our code

So theoretically, we should be able to sort of any field at our wish with minimum code. Lets see if we really are:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57 | packagecorejava.compare;    importjava.util.ArrayList;  importjava.util.Collections;  importjava.util.List;    publicclassTestSorting {      publicstaticvoidmain(String[] args) {          Employee e1 = newEmployee(1, "aTestName", "dLastName", 34);          Employee e2 = newEmployee(2, "nTestName", "pLastName", 30);          Employee e3 = newEmployee(3, "kTestName", "sLastName", 31);          Employee e4 = newEmployee(4, "dTestName", "zLastName", 25);            List<Employee> employees = newArrayList<Employee>();          employees.add(e2);          employees.add(e3);          employees.add(e1);          employees.add(e4);            // UnSorted List          System.out.println(employees);            Collections.sort(employees);          // Default Sorting by employee id          System.out.println(employees);            Collections.sort(employees, newFirstNameSorter());          // Sorted by firstName          System.out.println(employees);            Collections.sort(employees, newLastNameSorter());          // Sorted by lastName          System.out.println(employees);            Collections.sort(employees, newAgeSorter());          // Sorted by age          System.out.println(employees);      }  }    Output:    Employee : 2- nTestName - pLastName - 30  , Employee : 3- kTestName - sLastName - 31  , Employee : 1- aTestName - dLastName - 34  , Employee : 4- dTestName - zLastName - 25  ]  [Employee : 1- aTestName - dLastName - 34  , Employee : 2- nTestName - pLastName - 30  , Employee : 3- kTestName - sLastName - 31  , Employee : 4- dTestName - zLastName - 25  ]  [Employee : 1- aTestName - dLastName - 34  , Employee : 4- dTestName - zLastName - 25  , Employee : 3- kTestName - sLastName - 31  , Employee : 2- nTestName - pLastName - 30  ]  [Employee : 1- aTestName - dLastName - 34  , Employee : 2- nTestName - pLastName - 30  , Employee : 3- kTestName - sLastName - 31  , Employee : 4- dTestName - zLastName - 25  ]  [Employee : 4- dTestName - zLastName - 25  , Employee : 2- nTestName - pLastName - 30  , Employee : 3- kTestName - sLastName - 31  , Employee : 1- aTestName - dLastName - 34  ] |

Above class’s output shows clearly, now we are able to sort the collection of employees on any field using appropriate comparator implementation.

???????????????/

##### **How to convert an array of String to arraylist?**

Arrays.asList(strArray);

##### **How to reverse the list?**

Collections.reverse(list);

##### **Can a null element added to a TreeSet or HashSet?**

Yes in hashset

No in Treeset (will throw nullPointerExc)

##### **Difference between Set and List?**

The most noticeable differences are :

* Set is unordered collection where List is ordered collection based on zero based index.
* List allow duplicate elements but Set does not allow duplicates.
* List does not prevent inserting null elements (as many you like), but Set will allow only one null element.

##### **Difference between HashMap and HashTable?**

There are several differences between HashMap and Hashtable in Java:

* Hashtable is synchronized, whereas HashMap is not.
* Hashtable does not allow null keys or values. HashMap allows one null key and any number of null values.
* The third significant difference between HashMap vs Hashtable is that Iterator in the HashMap is a fail-fast iterator while the enumerator for the Hashtable is not.

##### **Difference between Vector and ArrayList?**

Lets note down the differences:

* All the methods of Vector is synchronized. But, the methods of ArrayList is not synchronized.
* Vector is a Legacy class added in first release of JDK. ArrayList was part of JDK 1.2, when collection framework was introduced in java.
* By default, Vector doubles the size of its array when it is re-sized internally. But, ArrayList increases by half of its size when it is re-sized.

##### **Difference between HashMap and HashSet?**

HashMap is collection of key-value pairs whereas HashSet is un-ordered collection of unique elements. That’s it. No need to describe further

##### **How to make a collection read only?**

Use following methods:

* Collections.unmodifiableList(list);
* Collections.unmodifiableSet(set);
* Collections.unmodifiableMap(map);

These methods takes collection parameter and return a new read-only collection with same elements as in original collection.

##### **How to make a collection thread safe?**

Use below methods:

* Collections.synchronizedList(list);
* Collections.synchronizedSet(set);
* Collections.synchronizedMap(map);

Above methods take collection as parameter and return same type of collection which are synchronized and thread safe.

##### **What do you understand by iterator fail-fast property?**

**Fail-fast Iterators fail as soon as they realized that structure of Collection has been changed since iteration has begun**. Structural changes means adding, removing or updating any element from collection while one thread is Iterating over that collection.

Fail-fast behavior is implemented by keeping a modification count and if iteration thread realizes the change in modification count it throws ConcurrentModificationException.

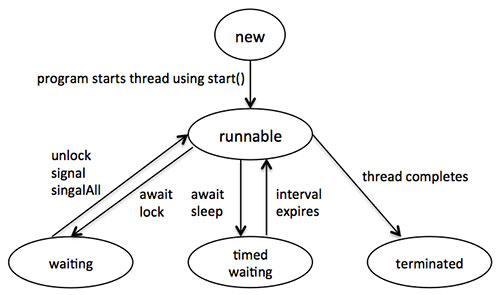
##### **What is difference between fail-fast and fail-safe?**

You have understood fail-fast in previous question. **Fail-safe iterators** are just opposite to fail-fast. **They never fail if you modify the underlying collection on which they are iterating**, because they work on clone of Collection instead of original collection and that’s why they are called as fail-safe iterator.

Iterator of CopyOnWriteArrayList is an example of fail-safe Iterator also iterator written by ConcurrentHashMapkeySet is also fail-safe iterator and never throwConcurrentModificationException.

## Life Cycle of a Thread:

A thread goes through various stages in its life cycle. For example, a thread is born, started, runs, and then dies. Following diagram shows complete life cycle of a thread.



Above-mentioned stages are explained here:

* **New:** A new thread begins its life cycle in the new state. It remains in this state until the program starts the thread. It is also referred to as a born thread.
* **Runnable:** After a newly born thread is started, the thread becomes runnable. A thread in this state is considered to be executing its task.
* **Waiting:** Sometimes, a thread transitions to the waiting state while the thread waits for another thread to perform a task.A thread transitions back to the runnable state only when another thread signals the waiting thread to continue executing.
* **Timed waiting:** A runnable thread can enter the timed waiting state for a specified interval of time. A thread in this state transitions back to the runnable state when that time interval expires or when the event it is waiting for occurs.
* **Terminated:**A runnable thread enters the terminated state when it completes its task or otherwise terminates.

## Thread Priorities:

Every Java thread has a priority that helps the operating system determine the order in which threads are scheduled.

Java thread priorities are in the range between MIN\_PRIORITY (a constant of 1) and MAX\_PRIORITY (a constant of 10). By default, every thread is given priority NORM\_PRIORITY (a constant of 5).

Threads with higher priority are more important to a program and should be allocated processor time before lower-priority threads. However, thread priorities cannot guarantee the order in which threads execute and very much platform dependentant.

## Create Thread by Implementing Runnable Interface:

If your class is intended to be executed as a thread then you can achieve this by implementing**Runnable** interface. You will need to follow three basic steps:

### STEP 1:

As a first step you need to implement a run() method provided by **Runnable** interface. This method provides entry point for the thread and you will put you complete business logic inside this method. Following is simple syntax of run() method:

publicvoid run()

### STEP 2:

At second step you will instantiate a **Thread** object using the following constructor:

Thread(RunnablethreadObj,StringthreadName);

Where, *threadObj* is an instance of a class that implements the **Runnable** interface and **threadName** is the name given to the new thread.

### STEP 3

Once Thread object is created, you can start it by calling **start( )** method, which executes a call to run( ) method. Following is simple syntax of start() method:

void start();

## Example:

Here is an example that creates a new thread and starts it running:

classRunnableDemoimplementsRunnable{

privateThread t;

privateStringthreadName;

RunnableDemo(String name){

threadName= name;

System.out.println("Creating "+threadName);

}

publicvoid run(){

System.out.println("Running "+threadName);

try{

for(inti=4;i>0;i--){

System.out.println("Thread: "+threadName+", "+i);

// Let the thread sleep for a while.

Thread.sleep(50);

}

}catch(InterruptedException e){

System.out.println("Thread "+threadName+" interrupted.");

}

System.out.println("Thread "+threadName+" exiting.");

}

publicvoid start ()

{

System.out.println("Starting "+threadName);

if(t ==null)

{

t =newThread(this,threadName);

t.start();

}

}

}

publicclassTestThread{

publicstaticvoid main(Stringargs[]){

RunnableDemo R1 =newRunnableDemo("Thread-1");

R1.start();

RunnableDemo R2 =newRunnableDemo("Thread-2");

R2.start();

}

}

This would produce the following result:

Creating Thread-1

Starting Thread-1

Creating Thread-2

Starting Thread-2

Running Thread-1

Thread: Thread-1, 4

Running Thread-2

Thread: Thread-2, 4

Thread: Thread-1, 3

Thread: Thread-2, 3

Thread: Thread-1, 2

Thread: Thread-2, 2

Thread: Thread-1, 1

Thread: Thread-2, 1

Thread Thread-1 exiting.

Thread Thread-2 exiting.

## Create Thread by Extending Thread Class:

The second way to create a thread is to create a new class that extends **Thread** class using the following two simple steps. This approach provides more flexibility in handling multiple threads created using available methods in Thread class.

### STEP 1

You will need to override **run( )** method available in Thread class. This method provides entry point for the thread and you will put you complete business logic inside this method. Following is simple syntax of run() method:

publicvoid run()

### STEP 2

Once Thread object is created, you can start it by calling **start( )** method, which executes a call to run( ) method. Following is simple syntax of start() method:

void start();

## Example:

Here is the preceding program rewritten to extend Thread:

classThreadDemoextendsThread{

privateThread t;

privateStringthreadName;

ThreadDemo(String name){

threadName= name;

System.out.println("Creating "+threadName);

}

publicvoid run(){

System.out.println("Running "+threadName);

try{

for(inti=4;i>0;i--){

System.out.println("Thread: "+threadName+", "+i);

// Let the thread sleep for a while.

Thread.sleep(50);

}

}catch(InterruptedException e){

System.out.println("Thread "+threadName+" interrupted.");

}

System.out.println("Thread "+threadName+" exiting.");

}

publicvoid start ()

{

System.out.println("Starting "+threadName);

if(t ==null)

{

t =newThread(this,threadName);

t.start();

}

}

}

publicclassTestThread{

publicstaticvoid main(Stringargs[]){

ThreadDemo T1 =newThreadDemo("Thread-1");

T1.start();

ThreadDemo T2 =newThreadDemo("Thread-2");

T2.start();

}

}

This would produce the following result:

Creating Thread-1

Starting Thread-1

Creating Thread-2

Starting Thread-2

Running Thread-1

Thread: Thread-1, 4

Running Thread-2

Thread: Thread-2, 4

Thread: Thread-1, 3

Thread: Thread-2, 3

Thread: Thread-1, 2

Thread: Thread-2, 2

Thread: Thread-1, 1

Thread: Thread-2, 1

Thread Thread-1 exiting.

Thread Thread-2 exiting.

## Thread Methods:

Following is the list of important methods available in the Thread class.

|  |  |
| --- | --- |
| **SN** | **Methods with Description** |
| 1 | **public void start()** Starts the thread in a separate path of execution, then invokes the run() method on this Thread object. |
| 2 | **public void run()** If this Thread object was instantiated using a separate Runnable target, the run() method is invoked on that Runnable object. |
| 3 | **public final void setName(String name)** Changes the name of the Thread object. There is also a getName() method for retrieving the name. |
| 4 | **public final void setPriority(int priority)** Sets the priority of this Thread object. The possible values are between 1 and 10. |
| 5 | **public final void setDaemon(boolean on)** A parameter of true denotes this Thread as a daemon thread. |
| 6 | **public final void join(long millisec)** The current thread invokes this method on a second thread, causing the current thread to block until the second thread terminates or the specified number of milliseconds passes. |
| 7 | **public void interrupt()** Interrupts this thread, causing it to continue execution if it was blocked for any reason. |
| 8 | **public final booleanisAlive()** Returns true if the thread is alive, which is any time after the thread has been started but before it runs to completion. |