Practice Set 2

Q.Why String is an immutable class?

- String pool is possible only because String is immutable in Java. This way Java Runtime saves a lot of heap space because different String variables can refer to the same String variable in the pool. If String would not have been immutable, then String interning would not have been possible because if any variable would have changed the value, it would have been reflected in the other variables too.
- If String is not immutable then it would cause a severe security threat to the application. For example, database username, password are passed as String to get database connection and in socket programming host and port details passed as String. Since String is immutable, its value can't be changed otherwise any hacker could change the referenced value to cause security issues in the application.
- Since String is immutable, it is safe for multithreading. A single String instance can be shared across different threads. This avoids the use of synchronization for thread safety. Strings are implicitly thread-safe.
- Strings are used in java classloader and immutability provides security that correct class is getting loaded by Classloader. For example, think of an instance where you are trying to load java.sql.Connection class but the referenced value is changed to myhacked.Connection class that can do unwanted things to your database.
- 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 the key in a Map and its processing is faster than other HashMap key objects. This is why String is the most widely used as HashMap keys.
- String Pool is possible only because String is immutable in Java and its implementation of String interning concept. String pool is also example of Flyweight design pattern.

Q.Explain working of: a. CopyOnWriteArrayList

CopyOnWriteArrayList: CopyOnWriteArrayList class is introduced in JDK 1.5, which implements List interface. It is enhanced version of ArrayList in which all modifications (add, set, remove, etc) are implemented by making a fresh copy.

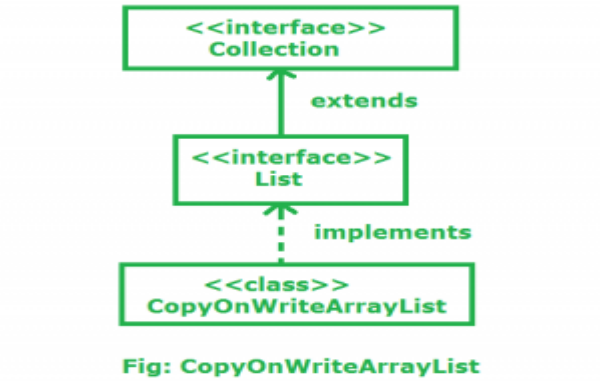


Fig: CopyOnWriteArrayList

- As the name indicates, CopyOnWriteArrayList creates a Cloned copy of underlying ArrayList, for every update operation at certain point both will be synchronized automatically ,which is taken care by JVM. Therefore there is no effect for threads which are performing read operation.
- It is costly to use because for every update operation a cloned copy will be created. Hence CopyOnWriteArrayList is the best choice if our frequent operation is read operation.
- It extends object and implements Serializable, Cloneable, Iterable<E>, Collection<E>, List<E> and RandomAccess
- The underlined data structure is grow-able array.
- It is thread-safe version of ArrayList.
- Insertion is preserved, duplicates are allowed and heterogeneous Objects are allowed.
- The main important point about CopyOnWriteArrayList is Iterator of CopyOnWriteArrayList can not perform remove operation otherwise we get Run-time exception saying UnsupportedOperationException.

b. CopyOnWriteArraySet

- CopyOnWriteArraySet is a Set that uses an internal CopyOnWriteArrayList for all of its operations. It is introduced in JDK 1.5, we can say that it is thread-safe version of Set.

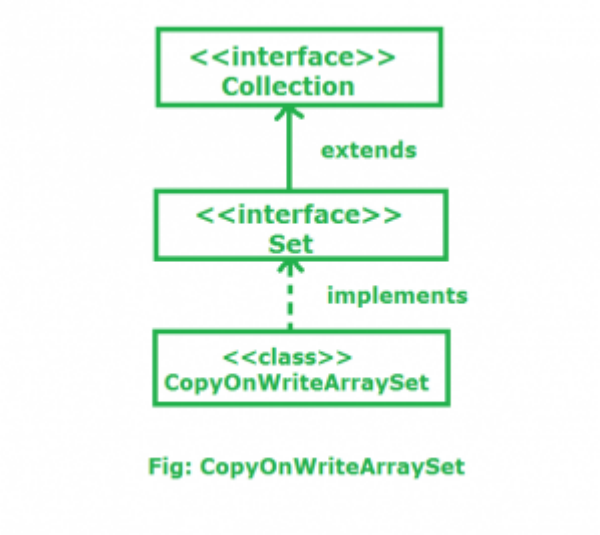


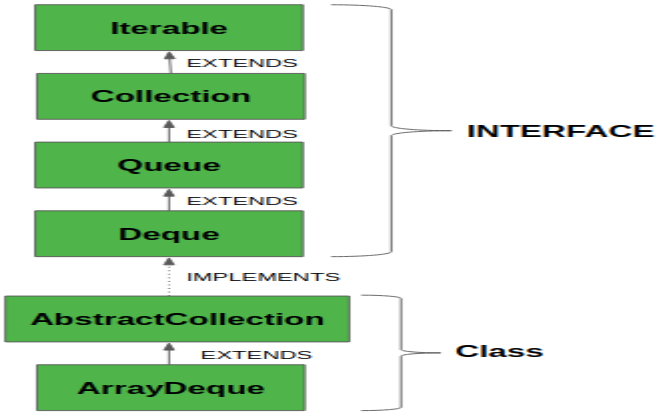
Fig: CopyOnWriteArraySet

- The internal implementation of CopyOnWriteArraySet is CopyOnWriteArrayList only.
- Multiple Threads are able to perform update operation simultaneously but for every update operation a separate cloned copy is created. As for every update a new cloned copy will be created which is costly. Hence if multiple update operation are required then it is not recommended to use CopyOnWriteArraySet.
- While one thread iterating the Set, other threads can perform updation, here we wont get any runtime exception like ConcurrentModificationException.
- Iterator of CopyOnWriteArraySet class can perform only read only and wont perform deletion, otherwise we will get Run-time exception UnsupportedOperationException.

c. Deque

- Deque or Double Ended Queue is a generalized version of Queue data structure that allows insert and delete at both ends.

- The java.util.Deque interface is a subtype of the java.util.Queue interface. The Deque is related to the double-ended queue that supports addition or removal of elements from either end of the data structure, it can be used as a queue (first-in-first-out/FIFO) or as a stack (last-in-first-out/LIFO). These are faster than Stack and LinkedList.
- This is the hierarchy of Deque interface in Java:



- Few important features of Deque are:

- It provides the support of resizable array and helps in restriction-free capacity, so to grow the array according to the usage.
- Array deques prohibit the use of Null elements and do not accept any such elements.
- Any concurrent access by multiple threads is not supported.
- In the absence of external synchronization, Deque is not thread-safe.

Summarizing the methods, we get:

	First Element or Head		Last Element or Tail	
Operations	**Throws exception**	**Special Value**	**Throws exception**	**Special Value**
Insert	addFirst(element)	offerFirst(element)	addLast(element)	offerLast(element)
Remove	removeFirst()	pollFirst()	removeLast()	pollLast()
Examine	getFirst()	peekFirst()	getLast()	peekLast()

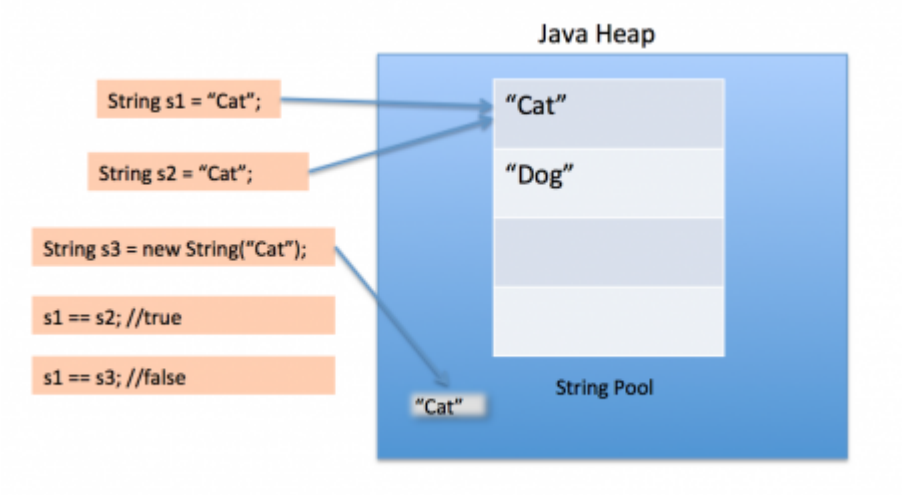
d. IdentityHashMap

- IdentityHashMap implements Map, Serializable and Clonable interfaces and extends AbstractMap class.
- This class is not a general-purpose 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.
- This class is used when the user requires the objects to be compared via reference.
- IdentityHashMap vs HashMap
- IdentityHashMap uses equality operator "==" for comparing keys and values while HashMap uses equals method for comparing keys and values inside Map.

- Since IdentityHashMap doesn't use equals() its comparatively faster than HashMap for object with expensive equals().
- IdentityHashMap doesn't require keys to be immutable as it is not relied on equals().

Q.Explain the concept of String pool. What is flyweight design pattern?

- As the name suggests, String Pool in java is a pool of Strings stored in Java Heap Memory. We know that String is a special class in java and we can create String objects using a new operator as well as providing values in double-quotes.
- String Pool in Java
- Here is a diagram that clearly explains how String Pool is maintained in java heap space and what happens when we use different ways to create Strings.



- String Pool is possible only because String is immutable in Java and its implementation of String interning concept. String pool is also example of Flyweight design pattern.
- 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 the 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 in heap space. We can use intern() method to put it into the pool or refer to another String object from the string pool having the same value.

Here is the java program for the String Pool image:

```
public class StringPool {
   public static void main(String[] args) {
      String s1 = "Cat";
      String s2 = "Cat";
      String s3 = new String("Cat");
      System.out.println("s1 == s2 :"+(s1==s2));
      System.out.println("s1 == s3 :"+(s1==s3));
```

```
  }
}
```
Output of the above program is:

s1 == s2 :true

s1 == s3 :false

How many strings are getting created in the below statement?

- String str = new String("Cat");
- In the above statement, either 1 or 2 string will be created. If there is already a string literal “Cat” in the pool, then only one string “str” will be created in the pool. If there is no string literal “Cat” in the pool, then it will be first created in the pool and then in the heap space, so a total of 2 string objects will be created.

Flyweight Design Pattern

- Flyweight pattern is one of the structural design patterns as this pattern provides ways to decrease object count thus improving application required objects structure. Flyweight pattern is used when we need to create a large number of similar objects (say 105). One important feature of flyweight objects is that they are immutable. This means that they cannot be modified once they have been constructed.
- Why do we care for number of objects in our program?
- Less number of objects reduces the memory usage, and it manages to keep us away from errors related to memory like java.lang.OutOfMemoryError.
- Although creating an object in Java is really fast, we can still reduce the execution time of our program by sharing objects.
- In Flyweight pattern we use a HashMap that stores reference to the object which have already been created, every object is associated with a key. Now when a client wants to create an object, he simply has to pass a key associated with it and if the object has already been created we simply get the reference to that object else it creates a new object and then returns it reference to the client.
- Intrinsic and Extrinsic States
- To understand Intrinsic and Extrinsic state, let us consider an example.
- Suppose in a text editor when we enter a character, an object of Character class is created, the attributes of the Character class are {name, font, size}. We do not need to create an object every time client enters a character since letter ‘B’ is no different from another ‘B’ . If client again types a ‘B’ we simply return the object which we have already created before. Now all these are intrinsic states (name, font, size), since they can be shared among the different objects as they are similar to each other.
- Now we add to more attributes to the Character class, they are row and column. They specify the position of a character in the document. Now these attributes will not be similar even for same characters, since no two characters will have the same position in a document, these states are termed as extrinsic states, and they can’t be shared among objects.
- Implementation : We implement the creation of Terrorists and Counter Terrorists In the game of Counter Strike. So we have 2 classes one for Terrorist(T) and other for Counter Terrorist(CT). Whenever a player asks for a weapon we assign him the asked

weapon. In the mission, terrorist's task is to plant a bomb while the counter terrorists have to diffuse the bomb.

- Why to use Flyweight Design Pattern in this example? Here we use the Fly Weight design pattern, since here we need to reduce the object count for players. Now we have n number of players playing CS 1.6, if we do not follow the Fly Weight Design Pattern then we will have to create n number of objects, one for each player. But now we will only have to create 2 objects one for terrorists and other for counter terrorists, we will reuse then again and again whenever required.
- Intrinsic State : Here 'task' is an intrinsic state for both types of players, since this is always same for T's/CT's. We can have some other states like their color or any other properties which are similar for all the Terrorists/Counter Terrorists in their respective Terrorists/Counter Terrorists class.
- Extrinsic State : Weapon is an extrinsic state since each player can carry any weapon of his/her choice. Weapon need to be passed as a parameter by the client itself.
- Class Diagram :

Q. Explain the following type of iterators a. Fail-fast
b. Fail-safe

Which Collections use the above types of iterators?

- **Concurrent Modification:** Concurrent Modification in programming means to modify an object concurrently when another task is already running over it. For example, in Java to modify a collection when another thread is iterating over it. Some Iterator implementations (including those of all the general purpose collection implementations provided by the JRE) may choose to throw ConcurrentModificationException if this behavior is detected.
- Iterators in java are used to iterate over the Collection objects.Fail-Fast iterators immediately throw ConcurrentModificationException if there is structural modification of the collection. Structural modification means adding, removing or updating any element from collection while a thread is iterating over that collection. Iterator on ArrayList, HashMap classes are some examples of fail-fast Iterator.
- Fail-Safe iterators don't throw any exceptions if a collection is structurally modified while iterating over it. This is because, they operate on the clone of the collection, not on the original collection and that's why they are called fail-safe iterators. Iterator on CopyOnWriteArrayList, ConcurrentHashMap classes are examples of fail-safe Iterator.
- Important points of fail-fast iterators :
- These iterators throw ConcurrentModificationException if a collection is modified while iterating over it.
- They use original collection to traverse over the elements of the collection.
- These iterators don't require extra memory.
- Ex : Iterators returned by ArrayList, Vector, HashMap.

- Note 1(from java-docs): The fail-fast behavior of an iterator cannot be guaranteed as it is, generally speaking, impossible to make any hard guarantees in the presence of unsynchronized concurrent modification. Fail-fast iterators throw ConcurrentModificationException on a best-effort basis. Therefore, it would be wrong to write a program that depended on this exception for its correctness: the fail-fast behavior of iterators should be used only to detect bugs.
- Note 2 : If you remove an element via Iterator remove() method, exception will not be thrown. However, in case of removing via a particular collection remove() method, ConcurrentModificationException will be thrown.
- Fail Safe Iterator
- First of all, there is no term as fail-safe given in many places as Java SE specifications does not use this term. I am using this term to demonstrate the difference between Fail Fast and Non-Fail Fast Iterator. These iterators make a copy of the internal collection (object array) and iterates over the copied collection. Any structural modification done to the iterator affects the copied collection, not original collection. So, original collection remains structurally unchanged.
- Fail-safe iterators allow modifications of a collection while iterating over it.
- These iterators don't throw any Exception if a collection is modified while iterating over it.
- They use copy of original collection to traverse over the elements of the collection.
- These iterators require extra memory for cloning of collection. Ex : ConcurrentHashMap, CopyOnWriteArrayList
- Also, those collections which don't use fail-fast concept may not necessarily create clone/snapshot of it in memory to avoid ConcurrentModificationException. For example, in case of ConcurrentHashMap, it does not operate on a separate copy although it is not fail-fast. Instead, it has semantics that is described by the official specification as weakly consistent(memory consistency properties in Java).
- Note(from java-docs) : The iterators returned by ConcurrentHashMap is weakly consistent. This means that this iterator can tolerate concurrent modification, traverses elements as they existed when iterator was constructed and may (but not guaranteed to) reflect modifications to the collection after the construction of the iterator.

- Difference between Fail Fast Iterator and Fail Safe Iterator
- The major difference is fail-safe iterator doesn't throw any Exception, contrary to fail-fast Iterator.This is because they work on a clone of Collection instead of the original collection and that's why they are called as the fail-safe iterator.
- How Fail Fast Iterator works ?
- To know whether the collection is structurally modified or not, fail-fast iterators use an internal flag called modCount which is updated each time a collection is modified.Fail-fast iterators checks the modCount flag whenever it gets the next value (i.e. using next() method), and if it finds that the modCount has been modified after this iterator has been created, it throws ConcurrentModificationException.