|  |
| --- |
| **Immutable Class Interview Questions**  **Q:**  **Difference between String and StringBuffer/StringBuilder in Java**  Well, the most important difference between String and StringBuffer/StringBuilder in java is that String object is immutable whereas StringBuffer/StringBuilder objects are mutable.  By immutable, we mean that the value stored in the String object cannot be changed. Then the next question that comes to our mind is “If String is immutable then how am I able to change the contents of the object whenever I wish to?” . Well, to be precise it’s not the same String object that reflects the changes you do. Internally a new String object is created to do the changes.  So suppose you declare a String object:  String myString = “Hello”;  Next, you want to append “Guest” to the same String. What do you do?  myString = myString + ” Guest”;  When you print the contents of myString the output will be “Hello Guest”. Although we made use of the same object(myString), internally a new object was created in the process. So, if you were to do some string operation involving an append or trim or some other method call to modify your string object, you would really be creating those many new objects of class String.  Now isn’t that a performance issue?  Yes, it definitely is.  Then how do you make your string operations efficient?  By using StringBuffer or StringBuilder.  How would that help?  Well, since StringBuffer/StringBuilder objects are mutable, we can make changes to the value stored in the object. What this effectively means is that string operations such as append would be more efficient if performed using StringBuffer/StringBuilder objects than String objects.  Finally, what’s the difference between StringBuffer and StringBuilder?  StringBuffer and StringBuilder have the same methods with one difference and that’s of synchronization. StringBuffer is synchronized (which means it is thread safe and hence you can use it when you implement threads for your methods) whereas StringBuilder is not synchronized (which implies it isn’t thread safe).  So, if you aren’t going to use threading then use the StringBuilder class as it’ll be more efficient than StringBuffer due to the absence of synchronization.  [In case you do not know – Here’s how you use StringBuilder](http://techtamasha.com/?p=30)  [A simple Example to demonstrate that String object is Immutable](http://techtamasha.com/?p=31)  In case you still have any doubts regarding String or StringBuilder then do leave a comment. I’ll be more than eager to help you out.  **Note: StringBuilder was introduced in Java 1.5 (so if you happen to use versions 1.4 or below you’ll have to use StringBuffer)** |
| Q.) [Why String is immutable or final in Java](http://javarevisited.blogspot.in/2010/10/why-string-is-immutable-in-java.html) Ans) 1) Imagine StringPool facility without making string immutable , its not possible at all because in case of string pool one string object/literal e.g. "Test" has referenced by many reference variables , so if any one of them change the value others will be automatically gets affected i.e. lets say  String A = "Test"  String B = "Test"  Now String B called "Test".toUpperCase() which change the same object into "TEST" , so A will also be "TEST" which is not desirable.  2)String has been widely used as parameter for many Java classes e.g. for opening network connection, you can pass hostname and port number as string , you can pass database URL as string for opening database connection, you can open any file in Java by passing name of file as argument to File I/O classes.  In case, if String is not immutable, this would lead serious security threat , I mean some one can access to any file for which he has authorization, and then can change the file name either deliberately or accidentally and gain access of those file. Because of immutability, you don't need to worry about those kind of threats. This reason also gel with, Why String is final in Java, by making java.lang.String final, Java designer ensured that no one overrides any behavior of String class.  3)Since String is immutable it can safely shared between many threads ,which is very important for multithreaded programming and to avoid any synchronization issues in Java, Immutability also makes String instance thread-safe in Java, means you don't need to synchronize String operation externally. Another important point to note about String is memory leak caused by SubString, which is not a thread related issues but something to be aware of.  4) Another reason of Why String is immutable in Java is to allow String to cache its hashcode , being immutable String in Java caches its hashcode, and do not calculate every time we call hashcode method of String, which makes it very fast as hashmap key to be used in hashmap in Java. This one is also suggested by Jaroslav Sedlacek in comments below. In short because String is immutable, no one can change its contents once created which guarantees hashCode of String to be same on multiple invocation.  5) Another good reason of Why String is immutable in Java suggested by Dan Bergh Johnsson on comments is: The absolutely most important reason that String is immutable is that it is used by the class loading mechanism, and thus have profound and fundamental security aspects. Had String been mutable, a request to load "java.io.Writer" could have been changed to load "mil.vogoon.DiskErasingWriter".  **Q1) What is an immutable class?**  **Ans)** Immutable class is a class which once created; its contents cannot be changed. Immutable objects are the objects whose state cannot be changed once constructed. e.g. String class |
| **Q2) How to create an immutable class?**  **Ans)** To create an immutable class following steps should be followed:   1. Create a final class. 2. Set the values of properties using constructor only. 3. Make the properties of the class final and private 4. Do not provide any setters for these properties. 5. If the instance fields include references to mutable objects, don't allow those objects to be changed:    1. Don't provide methods that modify the mutable objects.    2. Don't share references to the mutable objects. Never store references to external, mutable objects passed to the constructor; if necessary, create copies, and store references to the copies. Similarly, create copies of your internal mutable objects when necessary to avoid returning the originals in your methods.   E.g. **public** **final** **class** FinalPersonClass {  **private** **final** String name;        **private** **final** **int** age;               **public** FinalPersonClass(**final** String name, **final** **int** age) {              **super**();              **this**.name = name;              **this**.age = age;        }        **public** **int** getAge() {              **return** age;        }        **public** String getName() {              **return** name;        }         } |
| **Q3) Immutable objects are automatically thread-safe –true/false?**  **Ans)** True. Since the state of the immutable objects can not be changed once they are created they are automatically synchronized/thread-safe. |
| **Q4) Which classes in java are immutable?**  **Ans)** All wrapper classes in java.lang are immutable –  String, Integer, Boolean, Character, Byte, Short, Long, Float, Double, BigDecimal, BigInteger |
| **Q5) What are the advantages of immutability?**  **Ans)** The advantages are:  1) Immutable objects are automatically thread-safe, the overhead caused due to use of synchronisation is avoided. 2) Once created the state of the immutable object cannot be changed so there is no possibility of them getting into an inconsistent state. 3) The references to the immutable objects can be easily shared or cached without having to copy or clone them as there state cannot be changed ever after construction.   4) The best use of the immutable objects is as the keys of a map. |

**Immutable Object Example:**

**package** immutable;

**final** **class** ImmutableOblectTraingle {

**private** **final** **int** edge1;

**private** **final** **int** edge2;

**private** **final** **int** edge3;

**private** **final** StringBuffer sb;

**public** ImmutableOblectTraingle(**int** edge1, **int** edge2, **int** edge3) {

**super**();

**this**.edge1 = edge1;

**this**.edge2 = edge2;

**this**.edge3 = edge3;

sb=**new** StringBuffer("String buffer is a mutable object..");

System.*out*.println("Immutable object instantiated..");

}

@Override

**public** String toString() {

**return** "ImmutableOblectTraingle [edge1=" + edge1 + ", edge2=" + edge2

+ ", edge3=" + edge3 + ", sb=" + sb + "]";

}

**public** **int** getEdge1() {

**return** edge1;

}

**public** **int** getEdge2() {

**return** edge2;

}

**public** **int** getEdge3() {

**return** edge3;

}

/\*\*

\* Returns a mutable object - likely bad style.

\*

\* The caller gets a direct reference to the internal field. This is usually dangerous,

\* since the StringBuffer object state can be changed both by this class and its caller.

\* That is, this class is no longer in complete control of sb.

\*/

**public** StringBuffer getSb() {

**return** sb;

}

/\*\*

\* Returns a mutable object - good style.

\*

\* Returns a defensive copy of the field.

\* The caller of this method can do anything they want with the

\* returned StringBuffer object, without affecting the internals of this

\* class in any way. Why? Because they do not have a reference to

\* actual sb(StringBuffer). Rather, they are playing with a second StringBuffer that initially has the

\* same data as sb.

\*/

/\*public StringBuffer getSb() {

return new StringBuffer(sb);

}\*/

}

**public** **class** ImmutableObjectTest {

**public** **static** **void** main(String[] args) {

ImmutableOblectTraingle immutable=**new** ImmutableOblectTraingle(1,2, 3);

immutable.getSb().append("Oh Yes, StringBuffer is modified.");

System.*out*.println(immutable.toString());

System.*out*.println(immutable.getSb());

}

}

**Output when original mutable object is returned>>>**

Immutable object instantiated..

ImmutableOblectTraingle [edge1=1, edge2=2, edge3=3, sb=String buffer is a mutable object..Oh Yes, StringBuffer is modified.]

String buffer is a mutable object..Oh Yes, StringBuffer is modified.

**final** **class** ImmutableOblectTraingle {

**private** **final** **int** edge1;

**private** **final** **int** edge2;

**private** **final** **int** edge3;

**private** **final** StringBuffer sb;

**public** ImmutableOblectTraingle(**int** edge1, **int** edge2, **int** edge3) {

**super**();

**this**.edge1 = edge1;

**this**.edge2 = edge2;

**this**.edge3 = edge3;

sb=**new** StringBuffer("String buffer is a mutable object..");

System.*out*.println("Immutable object instantiated..");

}

@Override

**public** String toString() {

**return** "ImmutableOblectTraingle [edge1=" + edge1 + ", edge2=" + edge2

+ ", edge3=" + edge3 + ", sb=" + sb + "]";

}

**public** **int** getEdge1() {

**return** edge1;

}

**public** **int** getEdge2() {

**return** edge2;

}

**public** **int** getEdge3() {

**return** edge3;

}

/\*\*

\* Returns a mutable object - likely bad style.

\* The caller gets a direct reference to the internal field. This is usually dangerous, since the StringBuffer object state can be changed both by this class and its caller.

\* That is, this class is no longer in complete control of sb.

\*/

/\*public StringBuffer getSb() {

return sb;

}\*/

/\*\*

\* Returns a mutable object - good style.

\*

\* Returns a defensive copy of the field.

\* The caller of this method can do anything they want with the

\* returned StringBuffer object, without affecting the internals of this

\* class in any way. Why? Because they do not have a reference to

\* actual sb(StringBuffer). Rather, they are playing with a second StringBuffer that initially has the

\* same data as sb.

\*/

**public** StringBuffer getSb() {

**return** **new** StringBuffer(sb);

}

}

**public** **class** ImmutableObjectTest {

**public** **static** **void** main(String[] args) {

ImmutableOblectTraingle immutable=**new** ImmutableOblectTraingle(1,2, 3);

immutable.getSb().append("Oh Yes, StringBuffer is modified.");

System.*out*.println(immutable.toString());

System.*out*.println(immutable.getSb());

}

}

**Output when original mutable object is not returned>>>**

Immutable object instantiated..

ImmutableOblectTraingle [edge1=1, edge2=2, edge3=3, sb=String buffer is a mutable object..]

String buffer is a mutable object..

**Q: Can we extend immutable classes (final classes)?**

**Ans:** No.

/\*

//Error: The type SubClass cannot subclass the final class ImmutableOblectTraingle

class SubClass extends ImmutableOblectTraingle{

}\*/