**Quick Reference to**

**Java Immutable**

Index

[1. Java Memory Model (Just for knowledge) 3](#_Toc468997506)

[Memory Management in Java – Young Generation 4](#_Toc468997507)

[Memory Management in Java – Young Generation 4](#_Toc468997508)

[Java Memory Model – Permanent Generation 4](#_Toc468997509)

[i) Method Area 5](#_Toc468997510)

[ii) Memory Pool 5](#_Toc468997511)

[iii) Runtime Constant Pool 5](#_Toc468997512)

[iv) Java Stack Memory 5](#_Toc468997513)

[2. String pool Change in Java 7 and what is String interning? 6](#_Toc468997514)

[3. Memory Model Change in Java 8 7](#_Toc468997515)

[4. Difference between String literal vs intern()? 8](#_Toc468997516)

[5. What is the difference between new String() and a String literal? 9](#_Toc468997517)

[6. Immutable Concept 10](#_Toc468997518)

[What is immutable class in Java 11](#_Toc468997519)

[How to write immutable class in Java 11](#_Toc468997520)

[Immutable Class Example in Java 12](#_Toc468997521)

[Benefits of Immutable Classes in Java 13](#_Toc468997522)

[7. Immutable objects are automatically thread-safe –true/false? 14](#_Toc468997523)

[8. Which classes in java are immutable? 14](#_Toc468997524)

[9. What are the advantages of immutability? (Why String is Immutable?) 14](#_Toc468997525)

[10. Difference between Mutable objects and Immutable objects (with example) 15](#_Toc468997526)

[11. String is Special !! 15](#_Toc468997527)

[12. StringBuffer & StringBuilder 18](#_Toc468997528)

[13. Important Methods of StringBuilder 18](#_Toc468997529)

[14. StringTokenizer 19](#_Toc468997530)

[15. How to insert a string at particular location in String? 20](#_Toc468997531)

[16. When String literal is GC? 21](#_Toc468997532)

[17. String memory allocation using + operator 21](#_Toc468997533)

[18. String concatenation: concat() vs + operator 22](#_Toc468997534)

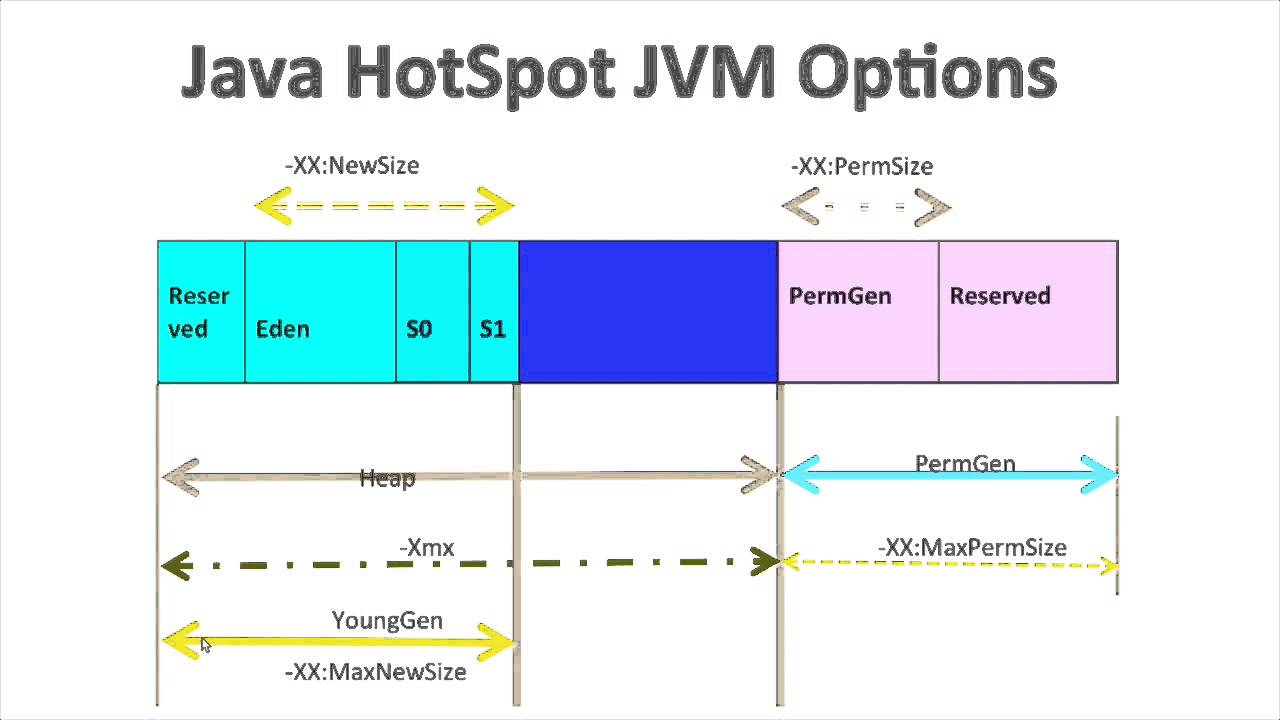
[19. Output for following program. 23](#_Toc468997535)

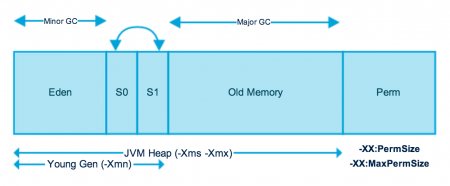
[20. Java String Immutability storage when String object is changed 25](#_Toc468997536)

[References 27](#_Toc468997537)

# Java Memory Model (Just for knowledge)

* (<http://www.journaldev.com/2856/java-jvm-memory-model-memory-management-in-java>)

[](https://www.google.com/url?sa=i&rct=j&q=&esrc=s&source=images&cd=&cad=rja&uact=8&ved=0ahUKEwiGv__Q9t_QAhXrxFQKHQ7FAJIQjRwIBw&url=https://www.youtube.com/watch?v%3D0vbnkZ7-25s&bvm=bv.139782543,d.cGw&psig=AFQjCNHZJ6-TeGeDxJoRgXKBTNnR7JoaMg&ust=1481126041068895)(Should read the information from above link for more detail but for interview this much would be fine)

[](http://cdn.journaldev.com/wp-content/uploads/2014/05/Java-Memory-Model.png)

As you can see in the above image, JVM memory is divided into separate parts. At broad level, JVM Heap memory is physically divided into two parts – **Young Generation** and **Old Generation.**

## Memory Management in Java – Young Generation

Young generation is the place where all the new objects are created. When young generation is filled, garbage collection is performed. This garbage collection is called **Minor GC**. Young Generation is divided into three parts – **Eden Memory** and two **Survivor Memory** spaces.

Important Points about Young Generation Spaces:

* Most of the newly created objects are located in the Eden memory space.
* When Eden space is filled with objects, Minor GC is performed and all the survivor objects are moved to one of the survivor spaces.
* Minor GC also checks the survivor objects and move them to the other survivor space. So at a time, one of the survivor space is always empty.
* Objects that are survived after many cycles of GC, are moved to the Old generation memory space. Usually it’s done by setting a threshold for the age of the young generation objects before they

## Memory Management in Java – Young Generation

Young generation is the place where all the new objects are created. When young generation is filled, garbage collection is performed. This garbage collection is called **Minor GC**. Young Generation is divided into three parts – **Eden Memory** and two **Survivor Memory** spaces.

Important Points about Young Generation Spaces:

* Most of the newly created objects are located in the Eden memory space.
* When Eden space is filled with objects, Minor GC is performed and all the survivor objects are moved to one of the survivor spaces.
* Minor GC also checks the survivor objects and move them to the other survivor space. So at a time, one of the survivor space is always empty.
* Objects that are survived after many cycles of GC, are moved to the Old generation memory space. Usually it’s done by setting a threshold for the age of the young generation objects before they

## Java Memory Model – Permanent Generation

Permanent Generation or “Perm Gen” contains the application metadata required by the JVM to describe the classes and methods used in the application. Note that Perm Gen is not part of Java Heap memory.

Perm Gen is populated by JVM at runtime based on the classes used by the application. Perm Gen also contains Java SE library classes and methods. Perm Gen objects are garbage collected in a full garbage collection.

### Method Area

Method Area is part of space in the Perm Gen and used to store class structure (runtime constants and static variables) and code for methods and constructors.

### Memory Pool

Memory Pools are created by JVM memory managers to create a pool of immutable objects, if implementation supports it. String Pool is a good example of this kind of memory pool. Memory Pool can belong to Heap or Perm Gen, depending on the JVM memory manager implementation.

### Runtime Constant Pool

Runtime constant pool is per-class runtime representation of constant pool in a class. It contains class runtime constants and static methods. Runtime constant pool is the part of method area.

### Java Stack Memory

Java Stack memory is used for execution of a thread. They contain method specific values that are short-lived and references to other objects in the heap that are getting referred from the method. You should read [Difference between Stack and Heap Memory](http://www.journaldev.com/4098/java-heap-space-vs-stack-memory). Stack memory is always referenced in LIFO (Last-In-First-Out) order. Whenever a method is invoked, a new block is created in the stack memory for the method to hold local primitive values and reference to other objects in the method. As soon as method ends, the block becomes unused and become available for next method.  
Stack memory size is very less compared to Heap memory.

**Memory Management in Java – Java Heap Memory Switches**

Java provides a lot of memory switches that we can use to set the memory sizes and their ratios. Some of the commonly used memory switches are:

|  |  |
| --- | --- |
| VM Switch | VM Switch Description |
| -Xms | For setting the initial heap size when JVM starts |
| -Xmx | For setting the maximum heap size. |
| -Xmn | For setting the size of the Young Generation, rest of the space goes for Old Generation. |
| -XX:PermGen | For setting the initial size of the Permanent Generation memory |
| -XX:MaxPermGen | For setting the maximum size of Perm Gen |
| -XX:SurvivorRatio | For providing ratio of Eden space and Survivor Space, for example if Young Generation size is 10m and VM switch is -XX:SurvivorRatio=2 then 5m will be reserved for Eden Space and 2.5m each for both the Survivor spaces. The default value is 8. |
| -XX:NewRatio | For providing ratio of old/new generation sizes. The default value is 2. |

Most of the times, above options are sufficient, but if you want to check out other options too then please check [JVM Options Official Page](http://www.oracle.com/technetwork/java/javase/tech/vmoptions-jsp-140102.html).

# String pool Change in Java 7 and what is String interning?

* (<http://stackoverflow.com/questions/10578984/what-is-string-interning>)

In Java, String class has a public method intern() that returns a canonical representation for the string object. Java’s String class privately maintains a pool of strings, where String literals (e.g. String a=”test”) are automatically interned.

When the intern() method is invoked on a String object it looks the string contained by this String object in the pool, if the string is found there 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.

--From ([http://docs.oracle.com/javase/7/docs/api/java/lang/String.html#intern()](http://docs.oracle.com/javase/7/docs/api/java/lang/String.html#intern%28%29))

More on memory constraints of using intern()

On one hand, it is true that you can remove String duplicates by internalizing them. The problem is that the internalized strings go to the Permanent Generation, which is an area of the JVM that is reserved for non-user objects, like Classes, Methods and other internal JVM objects. The size of this area is limited, and is usually much smaller than the heap. Calling intern() on a String has the effect of moving it out from the heap into the permanent generation, and you risk running out of PermGen space.

--From (<http://www.codeinstructions.com/2009/01/busting-javalangstringintern-myths.html>)

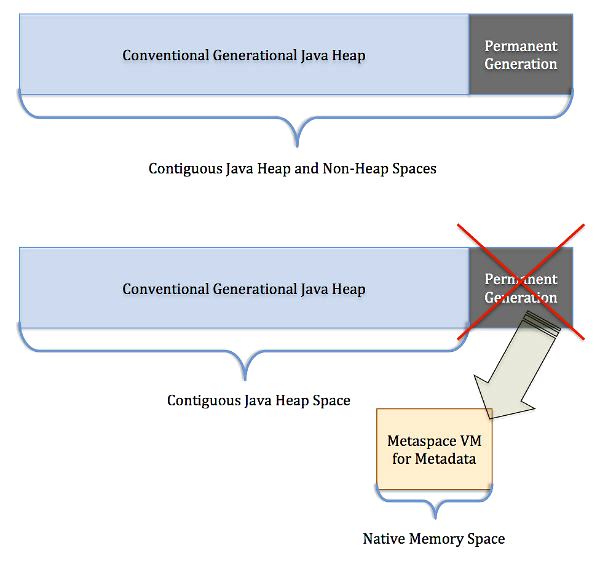
**From JDK 7 (I mean in HotSpot), something has changed.**

In JDK 7, interned strings are no longer allocated in the permanent generation of the Java heap, but are instead allocated in the main part of the Java heap (known as the young and old generations), along with the other objects created by the application. This change will result in more data residing in the main Java heap, and less data in the permanent generation, and thus may require heap sizes to be adjusted. Most applications will see only relatively small differences in heap usage due to this change, but larger applications that load many classes or make heavy use of the String.intern() method will see more significant differences.

-- From [Java SE 7 Features and Enhancements](http://www.oracle.com/technetwork/java/javase/jdk7-relnotes-418459.html)

# Memory Model Change in Java 8

* (<https://dzone.com/articles/java-8-permgen-metaspace>)

[](https://www.google.com/url?sa=i&rct=j&q=&esrc=s&source=images&cd=&cad=rja&uact=8&ved=0ahUKEwjHsJiB3OTQAhUL3mMKHb6QDb8QjRwIBw&url=https://www.infoq.com/articles/Java-PERMGEN-Removed&psig=AFQjCNFUW7Wk3R78LmKg4vd8TWmr5fK8Xw&ust=1481290621681741)

In summary the changes in Java 8 Memory Model:

**PermGen space situation**

* This memory space is completely removed.
* The PermSize and MaxPermSize JVM arguments are ignored and a warning is issued if present at start-up.

**Metaspace memory allocation model**

* Most allocations for the class metadata are now allocated out of native memory.
* By default class metadata allocation is limited by the amount of available native memory (capacity will of course depend if you use a 32-bit JVM vs. 64-bit along with OS virtual memory availability).
* A new flag is available (MaxMetaspaceSize), allowing you to limit the amount of native memory used for class metadata. If you don’t specify this flag, the Metaspace will dynamically re-size depending of the application demand at runtime.

**Metaspace garbage collection**

* Garbage collection of the dead classes and classloaders is triggered once the class metadata usage reaches the “MaxMetaspaceSize”.

# Difference between String literal vs intern()?

**(**Are the below two pieces of code the same?

String foo = "foo";

String foo = new String("foo").intern();

**)**

* The first one i.e.

String foo = "foo";

in this line, we are creating a String using String literals. That means the string is automatically saved in String Constant pool.

In the 2nd one , i.e. -

String foo = new String("foo").intern();

Here we are creating a String using new String() & then manually saving it to the String constant pool. If we didn't have mentiones intern() , it would not have saved in the String constant pool.

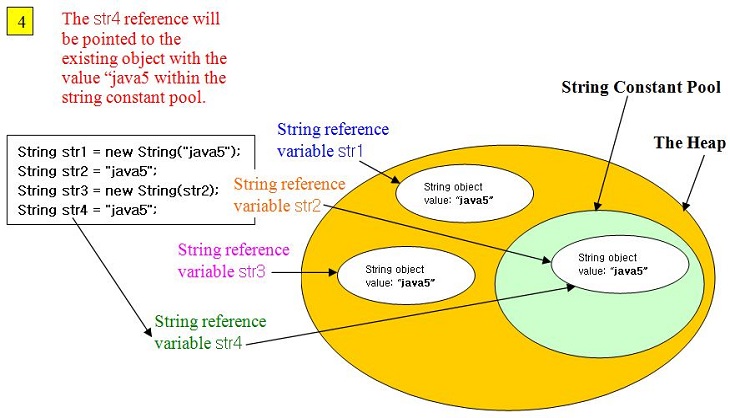
When the intern method is invoked, if the pool already contains a string equal to this String object as determined by the equals(Object) method, then the string from the pool is returned. Otherwise, this String object is added to the pool and a reference to this String object is returned.

All literal strings and string-valued constant expressions are interned.

So the end result is the same: A variable referencing the interned string "foo".

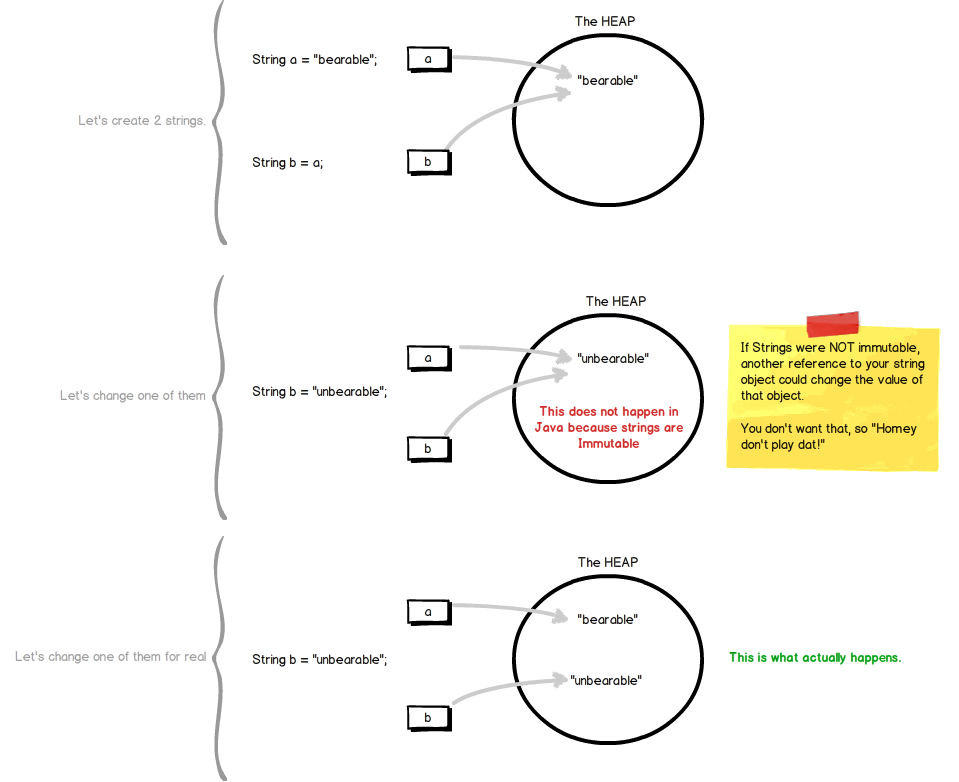
# What is the difference between new String() and a String literal?

* Image below is self-explanatory which indicates that String created with new keyword would be creating new object in Heap, while assigning String literal will return the same String from the pool if exists or it will create a new one. Please check the comparisons given in image also note that the memory model is on or after JDK 7 (String Constant Pool inside the Heap Space).



# Immutable Concept

* (<http://javarevisited.blogspot.com/2013/03/how-to-create-immutable-class-object-java-example-tutorial.html>)



Writing or creating immutable classes in Java is becoming popular day by day, because of concurrency and multithreading advantage provided by immutable objects. Immutable objects offers several benefits over conventional mutable object, especially while creating concurrent Java application. Immutable object not only guarantees safe publication of object’s state, but also can be shared among other threads without any external [synchronization](http://javarevisited.blogspot.com/2011/04/synchronization-in-java-synchronized.html). In fact JDK itself contains several immutable classes like [String](http://javarevisited.blogspot.com/2012/03/how-to-compare-two-string-in-java.html), [Integer](http://javarevisited.blogspot.com/2011/08/convert-string-to-integer-to-string.html) and other wrapper classes. For those, who doesn’t know what is immutable class or object, Immutable objects are those, whose state can not be changed once created e.g.java.lang.String, once created can not be modified e.g. trim, uppercase, lowercase. All modification in String result in new object, see [why String is immutable in Java](http://javarevisited.blogspot.com/2010/10/why-string-is-immutable-in-java.html) for more details. In this Java programming tutorial, we will learn, how to write immutable class in Java or how to make a class immutable. By the way making a class immutable is not difficult on code level, but its the decision to make, which class mutable or immutable which makes difference. I also suggest reading, Java Concurrency in Practice to learn more about concurrency benefit offered by Immutable object.

## What is immutable class in Java

As said earlier, Immutable classes are those class, whose [object](http://javarevisited.blogspot.com/2012/12/what-is-object-in-java-or-oops-example.html) can not be modified once created, it means any modification on immutable object will result in another immutable object. best example to understand immutable and mutable objects are, [String and StringBuffer](http://javarevisited.blogspot.com/2011/07/string-vs-stringbuffer-vs-stringbuilder.html). Since String is immutable class, any change on existing string object will result in another string e.g. replacing a character into String, [creating substring from String](http://javarevisited.blogspot.in/2011/10/how-substring-in-java-works.html), all result in a new objects. While in case of mutable object like StringBuffer, any modification is done on object itself and no new objects are created. Sometimes this immutability of String can also cause security hole, and that the reason [why password should be stored on char array instead of String](http://javarevisited.blogspot.com/2012/03/why-character-array-is-better-than.html).

## How to write immutable class in Java

Despite of few disadvantages, Immutable object still offers several benefits in multi-threaded programming and it’s a great choice to achieve [thread safety](http://javarevisited.blogspot.com/2012/12/how-to-create-thread-safe-singleton-in-java-example.html) in Java code. here are few rules, which helps to make a class immutable in Java :

1. State of immutable object cannot be modified after construction, any modification should result in new immutable object.

2. All fields of Immutable class should be final.

3. Object must be properly constructed i.e. object reference must not leak during construction process.

4. Object should be final in order to restrict sub-class for altering immutability of parent class.

By the way, you can still create immutable object by violating few rules, like String has its [hashcode](http://javarevisited.blogspot.com/2011/10/override-hashcode-in-java-example.html) in non final field, but its always guaranteed to be same. No matter how many times you calculate it, because it’s calculated from final fields, which is guaranteed to be same. This required a deep knowledge of Java memory model, and can create subtle [race conditions](http://javarevisited.blogspot.com/2012/02/what-is-race-condition-in.html) if not addressed properly. In next section we will see simple example of writing immutable class in Java. By the way, if your Immutable class has lots of optional and mandatory fields, then you can also use [Builder design pattern](http://javarevisited.blogspot.com/2012/06/builder-design-pattern-in-java-example.html) to make a class Immutable in Java.

## Immutable Class Example in Java

Here is complete code example of writing immutable class in Java. We have followed simplest approach and all rules for making a class immutable, including it [making class final](http://javarevisited.blogspot.com/2011/12/final-variable-method-class-java.html) to avoid putting immutability at risk due to [Inheritance](http://javarevisited.blogspot.com/2012/10/what-is-inheritance-in-java-and-oops-programming.html) and [Polymorphism](http://javarevisited.blogspot.com/2011/08/what-is-polymorphism-in-java-example.html).

public final class Contacts {

private final String name;

private final String mobile;

public Contacts(String name, String mobile) {

this.name = name;

this.mobile = mobile;

}

public String getName(){

return name;

}

public String getMobile(){

return mobile;

}

}

This Java class is immutable, because its state can not be changed once created. You can see that all of it’s fields are final. This is one of the most simple way of creating immutable class in Java, where all fields of class also remains immutable like String in above case. Some time you may need to write immutable class which includes mutable classes like[java.util.Date](http://javarevisited.blogspot.com/2012/04/difference-between-javautildate-and.html), **despite storing Date into final field it can be modified** **internally,** if internal date is returned to the client. In order to preserve immutability in such cases, its advised to **return copy of original object**, which is also one of the [Java best practice](http://javarevisited.blogspot.co.uk/2012/08/top-10-jdbc-best-practices-for-java.html). here is another example of making a class immutable in Java, which includes mutable member variable.

public final class ImmutableReminder{

private final Date remindingDate;

public ImmutableReminder (Date remindingDate) {

if(remindingDate.getTime() < System.currentTimeMillis()){

throw new IllegalArgumentException("Can not set reminder” +

“ for past time: " + remindingDate);

}

this.remindingDate = new Date(remindingDate.getTime());

}

public Date getRemindingDate() {

return (Date) remindingDate.clone();

}

}

In above example of creating immutable class, [Date](http://javarevisited.blogspot.com/2011/09/convert-date-to-string-simpledateformat.html) is a mutable object. If getRemindingDate() returns actual Date object than despite remindingDate being final variable, internals of Date can be modified by client code. By returning clone() or copy of remindingDate, we avoid that danger and preserves immutability of class.

## Benefits of Immutable Classes in Java

As I said earlier Immutable classes offers several benefits, here are few to mention:

1) Immutable objects are by default [thread safe](http://javarevisited.blogspot.com/2012/01/how-to-write-thread-safe-code-in-java.html), can be shared without synchronization in concurrent environment.

2) Immutable object simplifies development, because it Is easier to share between multiple threads without external synchronization.

3) Immutable object boost performance of Java application by reducing [synchronization](http://java67.blogspot.com/2013/01/difference-between-synchronized-block-vs-method-java-example.html) in code.

4) Another important benefit of Immutable objects is **reusability**, you can cache Immutable object and reuse them, much like String literals and Integers. You can use [static factory methods](http://javarevisited.blogspot.it/2011/12/factory-design-pattern-java-example.html) to provide methods like valueOf(), which can return an existing Immutable object from cache, instead of creating a new one.

Apart from above advantages, immutable object has disadvantage of creating garbage as well. Since immutable object can not be reused and they are just a use and throw. String being a prime example, which can create lot of garbage and can potentially slow down application due to [heavy garbage collection](http://javarevisited.blogspot.com/2011/04/garbage-collection-in-java.html), but again that's extreme case and if used properly Immutable object adds lot of value.

That's all on **how to write immutable class in Java**. we have seen rules of writing immutable classes, benefits offered by immutable objects and how we can create immutable class in Java which involves mutable fields. Don’t forget to read more about concurrency benefit offered by Immutable object in one of the best Java book recommended to Java programmers, Concurrency Practice in Java.

# Immutable objects are automatically thread-safe –true/false?

* True. Since the state of the immutable objects can not be changed once they are created they are automatically synchronized/thread-safe.

# Which classes in java are immutable?

* All wrapper classes in java.lang are immutable –

String, Integer, Boolean, Character, Byte, Short, Long, Float, Double, BigDecimal, BigInteger

# What are the advantages of immutability? (Why String is Immutable?)

* Below is the list of advantages/reasons of/for being String Immutable,
* Immutable objects are automatically thread-safe, the overhead caused due to use of synchronisation is avoided.
* Once created the state of the immutable object can not be changed so there is no possibility of them getting into an inconsistent state.
* 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.
* The best use of the immutable objects is as the keys of a map.
* String is widely used as parameter for many java classes, e.g. network connection, opening files, etc. Were String not immutable, a connection or file would be changed and lead to serious security threat

# Difference between Mutable objects and Immutable objects (with example)

* (<http://stackoverflow.com/questions/4658453/difference-between-mutable-objects-and-immutable-objects>)

They are not different from the point of view of JVM. Immutable objects don't have methods that can change the instance variables. And the instance variables are private; therefore you can't change it after you create it. A famous example would be String. You don't have methods like setString, or setCharAt. And s1 = s1 + "w" will create a new string, with the original one abandoned. That's my understanding.

Immutable objects are simply objects whose state (the object's data) cannot change after construction. Examples of immutable objects from the JDK include String and Integer.

For example:(Point is mutable and string immutable)

Point myPoint = new Point( 0, 0 ); System.out.println( myPoint ); myPoint.setLocation( 1.0, 0.0 ); System.out.println( myPoint ); String myString = new String( "old String" ); System.out.println( myString ); myString.replaceAll( "old", "new" ); System.out.println( myString );

The output is:

java.awt.Point[0.0, 0.0] java.awt.Point[1.0, 0.0] old String old String

# String is Special !!

* (<https://www3.ntu.edu.sg/home/ehchua/programming/java/J3d_String.html>)

A Java String contains an immutable sequence of Unicode characters. Unlike C/C++, where string is simply an array of char, A Java String is an object of the class java.lang.

Java String is, however, special. Unlike an ordinary class:

* String is associated with string literal in the form of double-quoted texts such as "Hello, world!". You can assign a string literal directly into a String variable, instead of calling the constructor to create a String instance.
* The '+' operator is overloaded to concatenate two String operands. '+' does not work on any other objects such as Point and Circle.
* String is *immutable*. That is, its content cannot be modified once it is created. For example, the method toUpperCase() constructs and returns a new String instead of modifying the its existing content.

The commonly-used method in the String class are summarized below. Refer to the JDK API for java.lang.String a complete listing.

// Length

int length() // returns the length of the String

boolean isEmpty() // same as *thisString*.length == 0

// Comparison

boolean equals(String another) // CANNOT use '==' or '!=' to compare two Strings in Java

boolean equalsIgnoreCase(String another)

int compareTo(String another) // return 0 if this string is the same as another;

// <0 if lexicographically less than another; or >0

int compareToIgnoreCase(String another)

boolean startsWith(String another)

boolean startsWith(String another, int fromIndex) // search begins at fromIndex

boolean endsWith(String another)

// Searching & Indexing

int indexOf(String search)

int indexOf(String search, int fromIndex)

int indexOf(int character)

int indexOf(int character, int fromIndex) // search forward starting at fromIndex

int lastIndexOf(String search)

int lastIndexOf(String search, int fromIndex) // search backward starting at fromIndex

int lastIndexOf(int character)

int lastIndexOf(int character, int fromIndex)

// Extracting a char or part of the String (substring)

char charAt(int index) // index from 0 to String's length - 1

String substring(int fromIndex)

String substring(int fromIndex, int endIndex) // exclude endIndex

// Creating a new String or char[] from the original (Strings are immutable!)

String toLowerCase()

String toUpperCase()

String trim() // create a new String removing white spaces from front and back

String replace(char oldChar, char newChar) // create a new String with oldChar replaced by newChar

String concat(String another) // same as *thisString* + another

char[] toCharArray() // create a char[] from this string

void getChars(int srcBegin, int srcEnd, char[] dst, int dstBegin) // copy into dst char[]

// Static methods for converting primitives to String

static String ValueOf(*type arg*) // *type* can be primitives or char[]

// Static method resulted in a formatted String using format specifiers

static String format(String formattingString, Object... args) // same as printf()

// Regular Expression (JDK 1.4)

boolean matches(String regexe)

String replaceAll(String regexe, String replacement)

String replaceAll(String regexe, String replacement)

String[] split(String regexe) // Split the String using regexe as delimiter,

// return a String array

String[] split(String regexe, int count) // for count times only

# StringBuffer & StringBuilder

* As explained earlier, Strings are immutable because String literals with same content share the same storage in the string common pool. Modifying the content of one String directly may cause adverse side-effects to other Strings sharing the same storage.

JDK provides two classes to support mutable strings: StringBuffer and StringBuilder (in core package java.lang) . A StringBuffer or StringBuilder object is just like any ordinary object, which are stored in the heap and not shared, and therefore, can be modified without causing adverse side-effect to other objects.

StringBuilder class was introduced in JDK 1.5. It is the same as StringBuffer class, except that StringBuilder is not synchronized for multi-thread operations. However, for single-thread program, StringBuilder, without the synchronization overhead, is more efficient.

Example:

StringBuilder sb = new StringBuilder(some\_appropriate\_size);

sb.append("select id1, id2 ");

sb.append(" from xyz");

return sb.toString();

# Important Methods of StringBuilder

* (<http://www.javatpoint.com/StringBuilder-class>)

|  |  |
| --- | --- |
| **Method** | **Description** |
| public StringBuilder append(String s) | is used to append the specified string with this string. The append() method is overloaded like append(char), append(boolean), append(int), append(float), append(double) etc. |
| public StringBuilder insert(int offset, String s) | is used to insert the specified string with this string at the specified position. The insert() method is overloaded like insert(int, char), insert(int, boolean), insert(int, int), insert(int, float), insert(int, double) etc. |
| Public StringBuilder replace(int startIndex, int endIndex, String str) | is used to replace the string from specified startIndex and endIndex. |
| public StringBuilder delete(int startIndex, int endIndex) | is used to delete the string from specified startIndex and endIndex. |
| public StringBuilder reverse() | is used to reverse the string. |
| public int capacity() | is used to return the current capacity. |
| public void ensureCapacity(int minimumCapacity) | is used to ensure the capacity at least equal to the given minimum. |
| public char charAt(int index) | is used to return the character at the specified position. |
| public int length() | is used to return the length of the string i.e. total number of characters. |
| public String substring(int beginIndex) | is used to return the substring from the specified beginIndex. |
| public String substring(int beginIndex, int endIndex) | is used to return the substring from the specified beginIndex and endIndex |

# StringTokenizer

* Very often, you need to break a line of texts into tokens delimited by white spaces. The java.util.StringTokenizer class supports this.

For example, the following program reverses the words in a String.

// Reverse the words in a String using StringTokenizer

import java.util.StringTokenizer;

public class StringTokenizerTest {

public static void main(String[] args) {

String str = "Monday Tuesday Wednesday Thursday Friday Saturday Sunday";

String strReverse;

StringBuilder sb = new StringBuilder();

StringTokenizer st = new StringTokenizer(str);

while (st.hasMoreTokens()) {

sb.insert(0, st.nextToken());

if (st.hasMoreTokens()) {

sb.insert(0, " ");

}

}

strReverse = sb.toString();

System.out.println(strReverse);

}

The JDK documentation stated that "StringTokenizer is a legacy class that is retained for compatibility reasons although its use is discouraged in new code. It is recommended that anyone seeking this functionality use the split() method of String or the java.util.regex package instead."

# How to insert a string at particular location in String?

* Following example illustrates how to do the same,

**public** **class** Main {

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

StringBuilder sb = **new** StringBuilder();

sb.append(**"java2s.com"**);

sb.append(123);// j av a 2s . c o m

sb.append(1.23);

sb.append(true);

sb.insert(4,**"JAVA2S.COM"**);

System.out.println(sb.toString());

}

}

**The output:**

http://java2s.com/Tutorials/JavaImage/myResult/C/COMBINE_APPEND_AND_INSERT_METHODS__DE68474AE04D04354094.PNG

# When String literal is GC?

* (<http://stackoverflow.com/questions/18406703/when-will-a-string-be-garbage-collected-in-java>)

First, it is String literals that get automatically interned / added to the String pool. Strings that are created by an application are not interned ... unless your application explicitly calls String.intern().

Second, in fact the rules for garbage collecting objects in the String pool are the same as for other Strings / other objects. The strings will be garbage collected if they ever become unreachable.

# String memory allocation using + operator

**(**String a = "abc";  
String b = "xyz";  
String result = a + b;

I was wondering if "result" string is a String constant allocated memory in string pool or a new object created on heap.)

* **(**[**http://stackoverflow.com/questions/11970785/string-memory-allocation-using-operator**](http://stackoverflow.com/questions/11970785/string-memory-allocation-using-operator)**)**

An important note:

String a = "abc";

String b = "xyz";

String result = a + b;

is the same as

// creates a number of objects.

String result = new StringBuilder().append(a).append(b).toString();

but

final String a = "abc";

final String b = "xyz";

String result = a + b;

is the same as (compiler optimizations)

String result = "abcxyz"; // creates no new objects

# String concatenation: concat() vs + operator

**(**I'm curious and wasn't sure, so I thought I'd ask:

Assuming String a and b.

a+=b

a=a.concat(b)

**)**

* **(**[**http://stackoverflow.com/questions/47605/string-concatenation-concat-vs-operator**](http://stackoverflow.com/questions/47605/string-concatenation-concat-vs-operator)**)**

**Under the hood for concat method (a=a.concat(b)):**

public String concat(String s) {

int i = s.length();

if (i == 0) {

return this;

} else {

char ac[] = new char[count + i];

getChars(0, count, ac, 0);

s.getChars(0, i, ac, count);

return new String(0, count + i, ac);

}

}

**While a+=b will be equal to:**

a = new StringBuilder().append(a).append(b).toString();

**Note:** Both the + and concat will store the object in heap memory not in constant pool as they are returning new object of the String except it is optimized by the compiler as shown in next question’s answer.

**Differences:**

* + If a is null then concat will throw NullPointerExcetion while a+=b will threat a as “null”” value.

e.g. String a=null, b=”test”; then a+=b; will be “nulltest”

* + concat() method accepts only String values while + operator will silently convert the argument to a string.

# Output for following program.

* **(**[**http://stackoverflow.com/questions/3029244/are-strings-created-with-concatenation-stored-in-the-string-pool**](http://stackoverflow.com/questions/3029244/are-strings-created-with-concatenation-stored-in-the-string-pool)**)**

**(**[**http://stackoverflow.com/questions/11970785/string-memory-allocation-using-operator**](http://stackoverflow.com/questions/11970785/string-memory-allocation-using-operator)**)**

public class InternTest1 {

public static void main(String[] args) {

String hello = "Hello", lo = "lo";

System.out.println(hello == "Hello");

System.out.println(Other.hello == hello);

//System.out.print(other.Other.hello == hello);

System.out.println(hello == ("Hel" + "lo"));

System.out.println(hello == ("Hel" + lo));

System.out.println(hello == ("Hel" + lo).intern());

String str1 = "abc";

String str2 = new String("def");

String str3 = str1.concat(str2);

String str4 = str2.concat("HI");

System.out.println(str3=="abcdef");

System.out.println(str4=="defHI");

System.out.println(str3="abcdef");

}

}

class Other {

static String hello = "Hello";

}

**Output:**

true

true

true

false

true

false

false

abcdef

# Java String Immutability storage when String object is changed

(I understood that if a String is initialized with a literal then it is allotted a space in String Pool and if initialized with the new Keyword it create a String's object. But I am confused with a case which is written below.

My question is what if a String is created with the new keyword and then it value is updated with a literal?

E.g.

String s = new String("Value1"); -- Creates a new object in heap space

then what if write the next statement as below.

s = "value2";

So my question is,

1 Will it create a String literal in a String Pool or it will update the value of that object?

2 If it creates a new literal in String Pool what will be happened to the currently existed object? Will it be destroyed or it will be there until the garbage collector is called.

**This is a small string if the string is say of the thousands of characters then I am just worried about the space it uses. So my key question is for the space.**

**Will it immediately free the space from the heap after assigning the literal?**

**Can anyone explain what what value goes where from the first statement to the second and what will happened to the memory area (heap and String Pool).**

)

* (<http://stackoverflow.com/questions/32564322/java-string-immutability-storage-when-string-object-is-changed>)

The **value is not updated** when running

s = "value2";

In Java, except for the *primitive types*, all other variables are *references* to objects. This means that only s is pointing to a new value.

**Immutability** guarantees that the state of an object **cannot change after construction**. In other words, there are no means to modify the content of any String object in Java. If you for instance state s = s+"a"; you have creates a *new* string, that somehow stores the new text.

1. **Practice Problem**

**(**Given A = 1, B = 2, C = 3, D = 4 .. Z = 26, write a java standalone program to encode the word “TELEPHONE” into numbers.)

* Following is the solution for given problem,

package examples.problems;

package examples.problems;

public class TelephoneProblem {

public static void main(String[] args) {

String s = "TELEPHONE";

char[] arrChar = s.toCharArray();

int[] val = new int[s.length()];

int i = 0;

for (char c : arrChar) {

val[i] = (c - 'A') + 1;

i++;

}

for (int z : val) {

System.out.print(z);

}

System.out.println("");

System.out.println("The value of 'C' - 'A' is " + ('C'-'A'));

}

}

/\*

Output

20512516815145

The value of 'C' - 'A' is 2

\*/

# References

* **More to Study**
  + **intern() behaving differently in Java 6 and Java 7**

(<http://stackoverflow.com/questions/7065337/intern-behaving-differently-in-java-6-and-java-7/>)

It is not answered properly but one can check for the reference and try to solve if it is possible.

* + <https://www3.ntu.edu.sg/home/ehchua/programming/java/J3d_String.html>
  + [Java Memory Model From a Programmer's Point-of-View](https://dzone.com/articles/java-memory-model-programer%E2%80%99s)