**Java – String ,String Buffer & String Builder & StringTokenizer Classes**

The **StringBuffer** and **StringBuilder** classes are used when there is a necessity to make a lot of modifications to Strings of characters.

Unlike Strings objects of type StringBuffer and Stringbuilder can be modified over and over again with out leaving behind a lot of new unused objects.

The StringBuilder class was introduced as of Java 5 and the main difference between the StringBuffer and StringBuilder is that StringBuilders methods are not thread safe(not Synchronised).

It is recommended to use **StringBuilder** whenever possible because it is faster than StringBuffer. However if thread safety is necessary the best option is StringBuffer objects.

public class Test{

public static void main(String args[]){

StringBuffer sBuffer = new StringBuffer(" test");

sBuffer.append(" String Buffer");

System.ou.println(sBuffer);

}

}

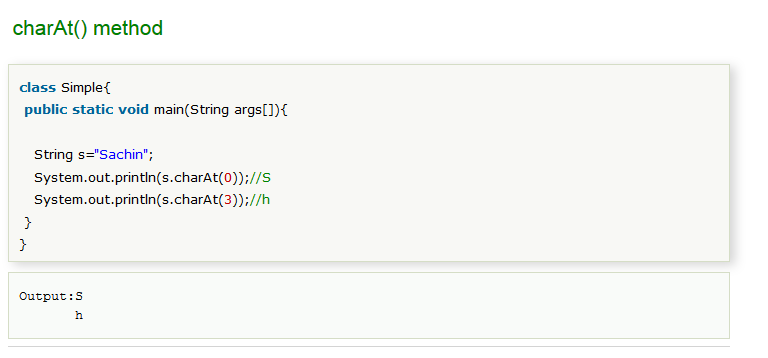
This would produce the following result:

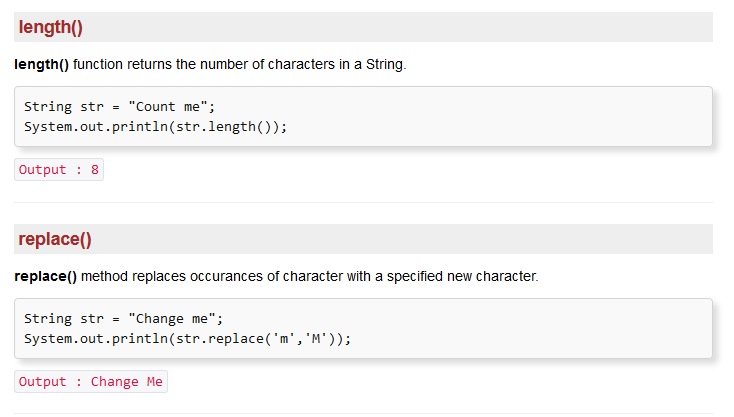
test String Buffer

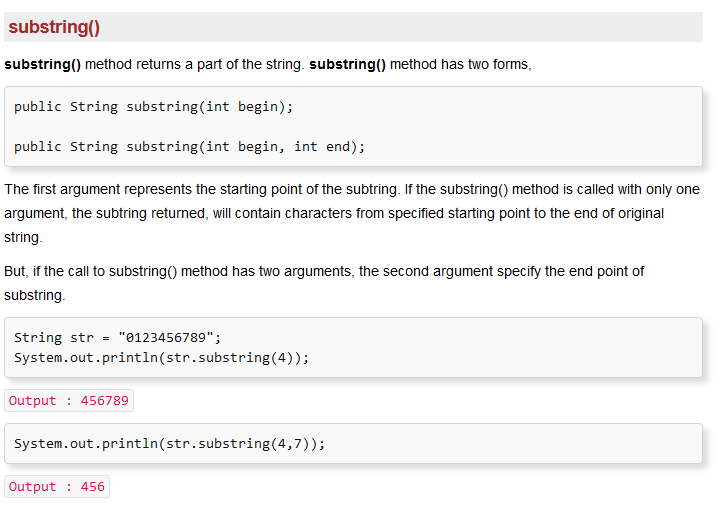
Here is the list of important methods supported by StringBuffer class:

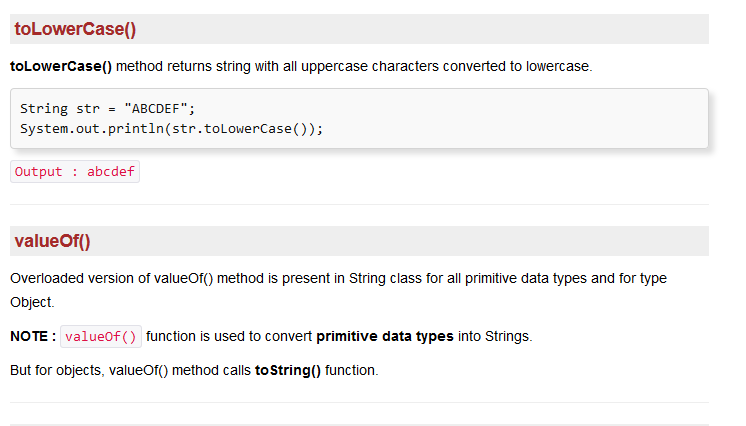
|  |  |
| --- | --- |
| **SN** | **Methods with Description** |
| 1 | [public StringBuffer append(String s)](http://www.tutorialspoint.com/java/stringbuffer_append.htm) Updates the value of the object that invoked the method. The method takes boolean, char, int, long, Strings etc. |
| 2 | [public StringBuffer reverse()](http://www.tutorialspoint.com/java/stringbuffer_reverse.htm) The method reverses the value of the StringBuffer object that invoked the method. |
| 3 | [public delete(int start, int end)](http://www.tutorialspoint.com/java/stringbuffer_delete.htm) Deletes the string starting from start index until end index. |
| 4 | [public insert(int offset, int i)](http://www.tutorialspoint.com/java/stringbuffer_insert.htm) This method inserts an string s at the position mentioned by offset. |
| 5 | [replace(int start, int end, String str)](http://www.tutorialspoint.com/java/stringbuffer_replace.htm) This method replaces the characters in a substring of this StringBuffer with characters in the specified String. |

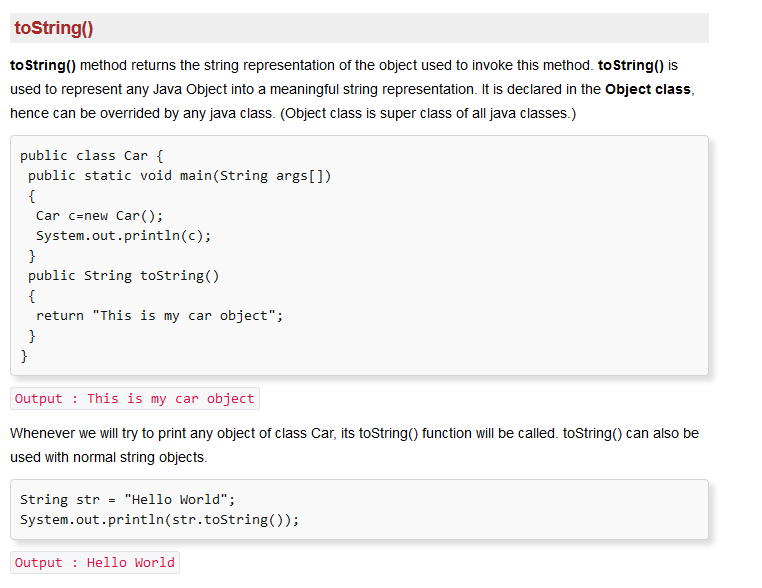


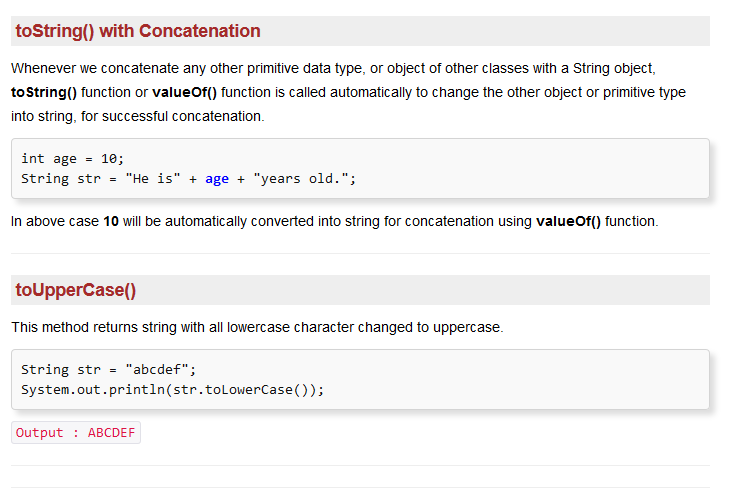




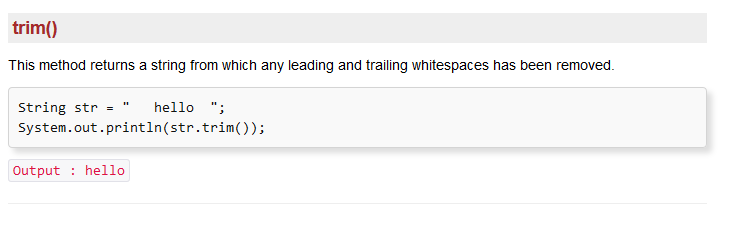




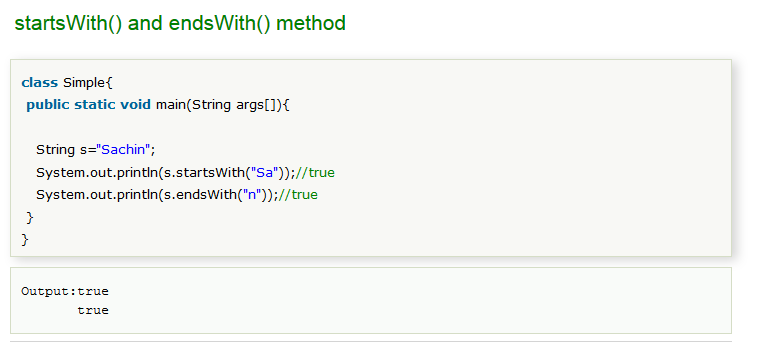


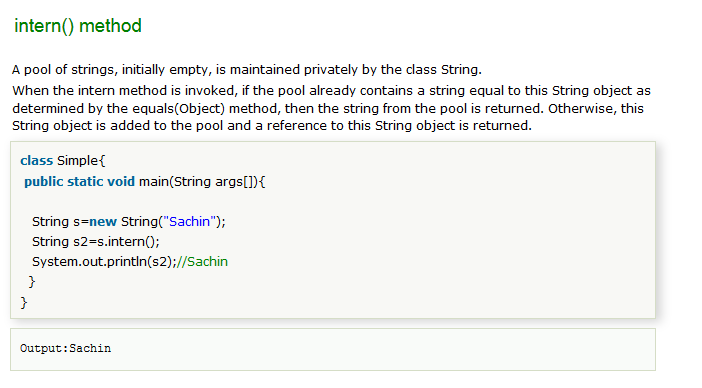


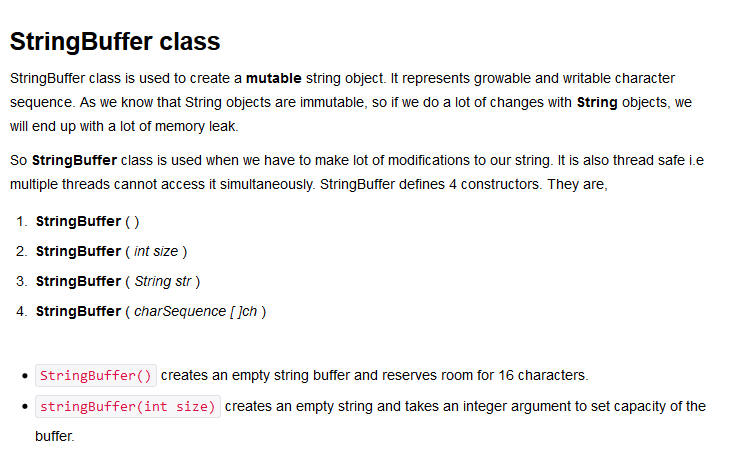


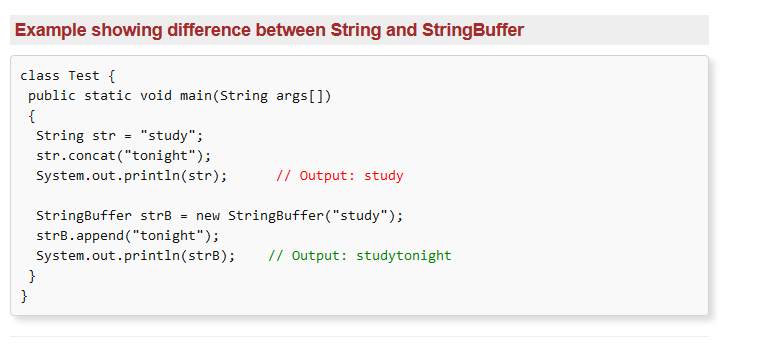


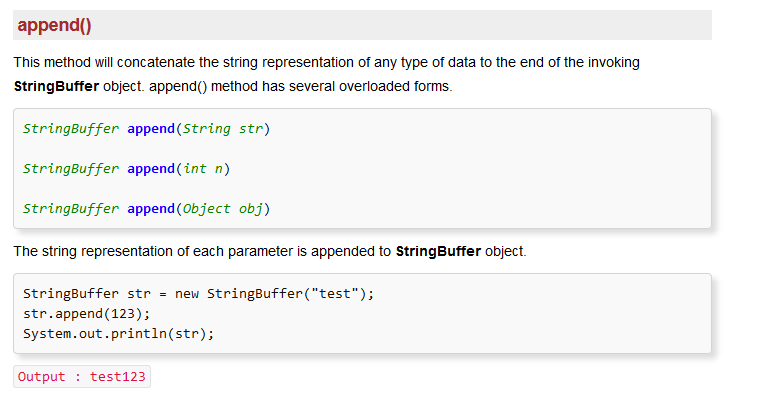


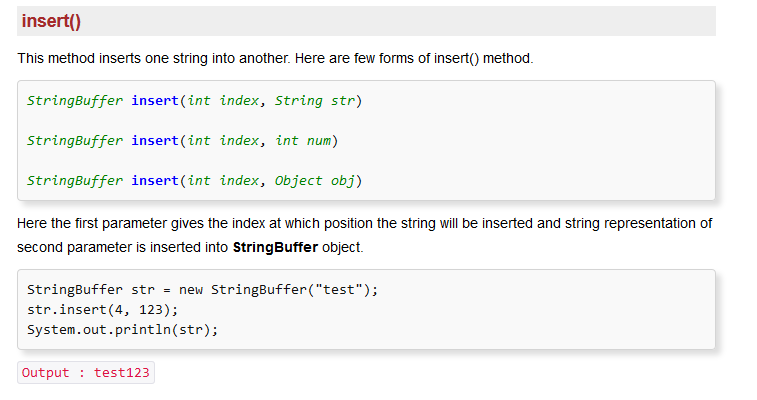




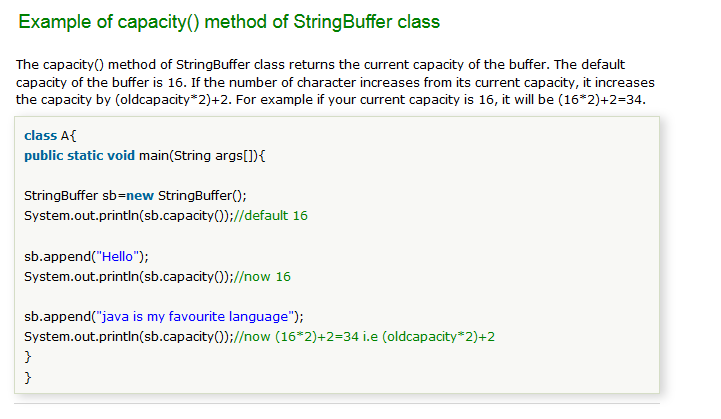


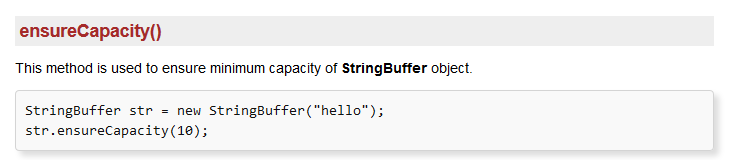


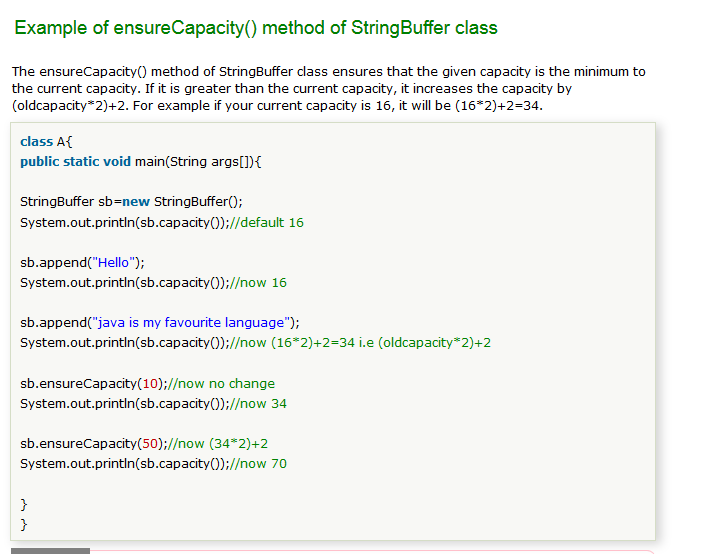


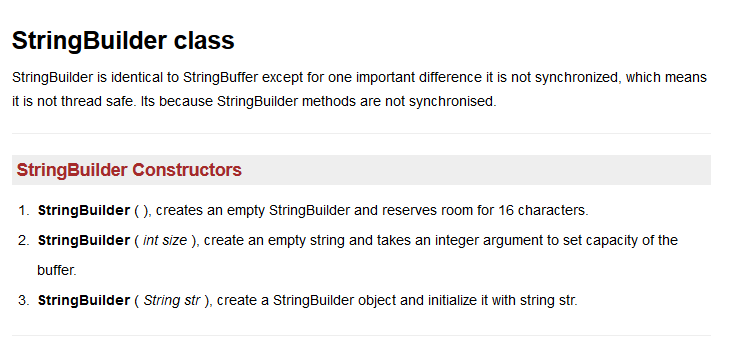


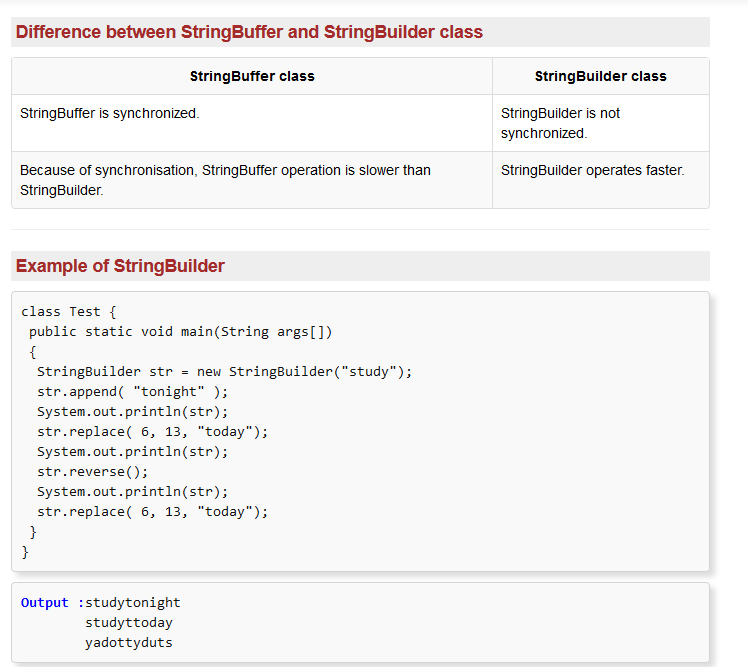












# [10 Things Every Java Programmer Should Know about String](http://javarevisited.blogspot.in/2013/07/java-string-tutorial-and-examples-beginners-programming.html)

String in Java is very special class and most frequently used class as well. There are lot many things to learn about String in Java than any other class, and having a good knowledge of different String functionalities makes you to use it properly. Given heavy use of Java String in almost any kind of project, it become even more important to know subtle detail about String. Though I have shared lot of String related article already here in **Javarevisited**, this is an effort to bring some of String feature together. In this tutorial we will see some important points about Java String, which is worth remembering. You can also refer my earlier post [10 advanced Java String questions](http://javarevisited.blogspot.com/2012/10/10-java-string-interview-question-answers-top.html) to know more about String. Though I tried to cover lot of things, there are definitely few things, which I might have missed; please let me know if you have any question or doubt on java.lang.String functionality and I will try to address them here.

**1) Strings are not null terminated in Java.**

Unlike C and C++, String in Java doesn't terminate with null character. Instead String are Object in Java and backed by character array. You can get the character array used to represent String in Java by calling toCharArray() method of java.lang.String class of JDK.

**2) Strings are immutable and final in Java**

Strings are immutable in Java it means once created you cannot modify content of String. If you modify it by using toLowerCase(), toUpperCase() or any other method, It always result in new String. Since String is final there is no way anyone can extend String or override any of String functionality. Now if you are puzzled [why String is immutable or final in Java](http://javarevisited.blogspot.com/2010/10/why-string-is-immutable-in-java.html). checkout the link.

**3) Strings are maintained in String Pool**

[Advanced Java String tutorial and example programmers ](http://3.bp.blogspot.com/-K6q0DQ1v-tw/TWu8owBtc2I/AAAAAAAAADA/oBoHDBiJ8ag/s1600/17.jpg)As I Said earlier String is special class in Java and all String literal e.g. "abc" (anything which is inside double quotes are String literal in Java) are maintained in a separate String pool, special memory location inside Java memory, more precisely inside [PermGen Space](http://javarevisited.blogspot.com/2012/01/tomcat-javalangoutofmemoryerror-permgen.html). Any time you create a new String object using String literal, JVM first checks String pool and if an object with similar content available, than it returns that and doesn't create a new object. JVM

doesn't perform String pool check if you create object using new operator.

You may face subtle issues if you are not aware of this String behaviour , here is an example

String name = "Scala"; //1st String object

String name\_1 = "Scala"; //same object referenced by name variable

String name\_2 = **new** String("Scala") //different String object

//this will return true

**if**(name==name\_1){

System.out.println("both name and name\_1 is pointing to same string object");

}

//this will return false

**if**(name==name\_2){

System.out.println("both name and name\_2 is pointing to same string object");

}

if you compare name and name\_1 using equality operator "==" it will return true because both are pointing to same object. While name==name\_2 will return false because they are pointing to different string object. It's worth remembering that [equality "==" operator compares object memory location](http://javarevisited.blogspot.sg/2012/12/difference-between-equals-method-and-equality-operator-java.html) and not characters of String. By default Java puts all string literal into string pool, but you can also put any string into pool by calling intern() method of java.lang.String class, like string created using new() operator.

**4) Use Equals methods for comparing String in Java**

String class overrides equals method and provides a content equality, which is based on characters, case and order. So if you want to compare two String object, to check whether they are same or not, always use equals() method instead of equality operator. Like in earlier example if we use [equals method](http://javarevisited.blogspot.com/2011/02/how-to-write-equals-method-in-java.html) to compare objects, they will be equal to each other because they all contains same contents. Here is example of comparing String using equals method.

String name = "Java"; //1st String object

String name\_1 = "Java"; //same object referenced by name variable

String name\_2 = **new** String("Java") //different String object

**if**(name.equals(name\_1)){

System.out.println("name and name\_1 are equal String by equals method");

}

//this will return false

**if**(name==name\_2){

System.out.println("name\_1 and name\_2 are equal String by equals method");

}

You can also check my earlier post [difference between equals() method and == operator](http://javarevisited.blogspot.com/2012/12/difference-between-equals-method-and-equality-operator-java.html) for more detail discussion on consequences of comparing two string using == operator in Java.

**5) Use indexOf() and lastIndexOf() or matches(String regex) method to search inside String**

String class in Java provides convenient method to see if a character or sub-string or a pattern exists in current String object. You can use indexOf() which will return position of character or String, if that exist in current String object or -1 if character doesn't exists in String. lastIndexOf is similar but it searches from end. String.match(String regex) is even more powerful, which allows you to search for a [regular expression pattern](http://javarevisited.blogspot.com/2012/10/regular-expression-example-in-java-to-check-String-number.html) inside String. here is examples of indexOf, lastIndexOf and matches method from java.lang.String class.

String str = "Java is best programming language";

**if**(str.indexOf("Java") != -**1**){

     System.out.println("String contains Java at index :" + str.indexOf("Java"));

}

**if**(str.matches("J.\*")){

     System.out.println("String Starts with J");

}

str ="Do you like Java ME or Java EE";

**if**(str.lastIndexOf("Java") != -**1**){

      System.out.println("String contains Java lastly at: " + str.lastIndexOf("Java"));

}

As expected indexOf will return 0 because characters in String are indexed from zero. lastIndexOf returns index of second “Java”, which starts at 23 and matches will return true because J.\* pattern is any String starting with character J followed by any character because of dot(.) and any number of time due to asterick (\*).

Remember matches() is tricky and some time non-intuitive. If you just put "Java" in matches it will return false because String is not equals to "Java" i.e. in case of plain text it behaves like equals method. See [here](http://java67.blogspot.sg/2012/09/java-string-matches-example-regular-expression.html) for more examples of String matches() method.

Apart from indexOf(), lastIndexOf() and matches(String regex) String also has methods like startsWith() and endsWidth(), which can be used to check an String if it starting or ending with certain character or String.

**6) Use SubString to get part of String in Java**

Java String provides another useful method called substring(), which can be used to get parts of String. basically you specify start and end index and substring() method returns character from that range. Index starts from 0 and goes till String.length()-1. By the way String.length() returns you number of characters in String, including white spaces like tab, space. One point which is worth remembering here is that substring is also backed up by character array, which is used by original String. This can be dangerous if original string object is very large and substring is very small, because even a small fraction can hold reference of complete array and prevents it from being garbage collected even if there is no other reference for that particular String. Read [How Substring works in Java](http://javarevisited.blogspot.com/2011/10/how-substring-in-java-works.html) for more details. Here is an example of using SubString in Java:

String str = "Java is best programming language";

//this will return part of String str from index 0 to 12

String subString = str.substring(**0**,**12**);

System.out.println("Substring: " + subString);

**7) "+" is overloaded for String concatenation**

*Java doesn't support Operator overloading* but String is special and + operator can be used to concatenate two Strings. It can even used to convert int, char, long or double to convert into String by simply concatenating with empty string "". internally + is implemented using StringBuffer prior to Java 5 and StringBuilder from Java 5 onwards. This also brings point of using StringBuffer or StringBuilder for manipulating String. Since both represent mutable object they can be used to reduce string garbage created because of temporary String. Read more about [StringBuffer vs StringBuilder](http://javarevisited.blogspot.com/2011/07/string-vs-stringbuffer-vs-stringbuilder.html) here.

**8) Use trim() to remove white spaces from String**

String in Java provides trim() method to remove white space from both end of String. If trim() removes white spaces it returns a new String otherwise it returns same String. Along with trim() String also provides replace() and replaceAll() method for replacing characters from String. replaceAll method even support regular expression. Read more about How to replace String in Java [here](http://javarevisited.blogspot.com/2011/12/java-string-replace-example-tutorial.html).

**9) Use split() for splitting String using Regular expression**

String in Java is feature rich. it has methods like split(regex) which can take any String in form of regular expression and split the String based on that. particularly useful if you dealing with comma separated file (CSV) and wanted to have individual part in a String array. There are other methods also available related to splitting String, see this [Java tutorial to split string](http://javarevisited.blogspot.com/2011/09/string-split-example-in-java-tutorial.html) for more details.

**10) Don't store sensitive data in String**

String pose security threat if used for storing sensitive data like passwords, SSN or any other sensitive information. Since String is immutable in Java there is no way you can erase contents of String and since they are kept in String pool (in case of String literal) they stay longer on Java heap ,which exposes risk of being seen by anyone who has access to Java memory, like reading from memory dump. Instead char[] should be used to store password or sensitive information. See [Why char[] is more secure than String for storing passwords in Java](http://javarevisited.blogspot.com.br/2012/03/why-character-array-is-better-than.html) for more details.  
  
  
**11) Character Encoding and String**  
Apart from all these 10 facts about String in Java, the most critical thing to know is *what encoding your String is using*. It does not make sense to have a String without knowing what encoding it uses. There is no way to interpret an String if you don't know the encoding it used. You can not assume that "plain" text is ASCII. If you have a String, in memory or stored in file, you must know what encoding it is in, or you cannot display it correctly. By default Java uses platform encoding i.e. character encoding of your server, and believe me this can cause huge trouble if you are handling Unicode data, especially if you are [converting byte array to XML String](http://javarevisited.blogspot.sg/2013/03/convert-and-print-byte-array-to-hex-string-java-example-tutorial.html). I have faced instances where our program fail to interpret Strings from European language e.g. German, French etc. because our server was not using Unicode encodings like UTF-8 or UTF-16. Thankfully, Java allows you to specify default character encoding for your application using system property file.encoding. See [here](http://javarevisited.blogspot.com/2012/01/get-set-default-character-encoding.html) to read more about character encoding in Java

That's all about String in Java. As I have said String is very special in Java, sometime even refer has God class. It has some unique feature like immutability, concatenation support, caching etc, and to become a serious Java programmer, detailed knowledge of String is quite important. Last but not the least don't forget about [character encoding](http://javarevisited.blogspot.com/2012/01/get-set-default-character-encoding.html) while converting a byte array into String in Java. Good knowledge of java.lang.String is must for good Java developers.

# [10 Java String interview Question answers - Advanced](http://javarevisited.blogspot.in/2012/10/10-java-string-interview-question-answers-top.html)

**10 Java String interview Question answers**

String interview questions in Java is one of Integral part of any [Core Java](http://javarevisited.blogspot.sg/2011/04/top-20-core-java-interview-questions.html) or [J2EE interviews](http://javarevisited.blogspot.sg/2011/09/servlet-interview-questions-answers.html). No one can deny importance of String and How much it in any Java application irrespective of whether its core Java desktop application, web application, Enterprise application or Mobile application. [String](http://javarevisited.blogspot.sg/2011/07/string-vs-stringbuffer-vs-stringbuilder.html) is one of the fundamental of Java programming language and correct understanding of String class is must for any Java programmer. What makes *String interview questions* in Java even more interesting is the special status of String in terms of features and privileges it has, like + operator is kind of overloaded to perform String concatenation despite the fact that [Java does not support operator overloading](http://javarevisited.blogspot.sg/2011/08/why-java-does-not-support-operator.html). There is a separate **String pool** to store String literal etc. In this article we will some frequently asked question on String in a Java interview which focuses on range of issues like immutability, [thread-safety](http://javarevisited.blogspot.sg/2012/01/how-to-write-thread-safe-code-in-java.html), Security etc.

## Java String interview Question

[Java String Interview Questions Answers Advanced Experienced](http://3.bp.blogspot.com/-K6q0DQ1v-tw/TWu8owBtc2I/AAAAAAAAADA/oBoHDBiJ8ag/s1600/17.jpg)Here are my list *of frequently asked question on String*, feel free to add any other interesting question which you faced on String during any [Core Java interview](http://java67.blogspot.sg/2012/09/top-10-tricky-java-interview-questions-answers.html) :

**1) What is String in Java ? Is String is data type?**

String in Java is not a primitive data type like int, long or double. String is a [class](http://javarevisited.blogspot.sg/2011/10/class-in-java-programming-general.html) or in more simple term a user defined type. This is confusing for some one who comes from C background. String is defined in java.lang package and wrappers its content in a character array. String provides [equals() method](http://javarevisited.blogspot.sg/2011/02/how-to-write-equals-method-in-java.html) to compare two String and provides various other method to operate on String like toUpperCase() to convert String into upper case, replace() to [replace String contents](http://javarevisited.blogspot.sg/2011/12/java-string-replace-example-tutorial.html), substring() to get substring, split() to [split long String](http://javarevisited.blogspot.sg/2011/09/string-split-example-in-java-tutorial.html) into multiple String.

**2) Why String is final in Java**

String is final by design in Java, some of the points which makes sense why String is final is Security, optimization and to maintain pool of String in Java. for details on each of this point see [Why String is final in Java](http://javarevisited.blogspot.sg/2010/10/why-string-is-immutable-in-java.html).

**3) What is Difference between String and StringBuffer in Java**

This is probably the most common question on String I have seen in Java interviews. Though String and Stringbuffer are two different class they are used in context of concatenating two Strings, Since String is immutable in Java every operation which changes String produces new String, which can be avoided by using Stringbuffer. See [String vs StringBuffer](http://javarevisited.blogspot.sg/2011/07/string-vs-stringbuffer-vs-stringbuilder.html)  for more details.

**4) What is difference in String on C and Java**

If you have mentioned C in your resume, then you are likely to face this *String interview question*. Well C String and Java String are completely different to each other, C String is a **null terminated** [character array](http://javarevisited.blogspot.sg/2012/02/how-to-convert-char-to-string-in-java.html) while String in Java is an Object. Also String is more feature rich in Java than C.

**5) Why char array is better than String for storing password?**

This String interview question is debatable and you might not agree with interviewer but this is also a chance to show that how deep and differently you can think of. One of the reason which people give Why you should store password in char array over String is related to immutability, since its not possible to remove erase contents of String but you can erase contents of char array. See [Why char array preferred over String for password](http://javarevisited.blogspot.sg/2012/03/why-character-array-is-better-than.html) for complete discussion.

**6) How do you compare two String in Java ?**

This is another *common String interview question* which appears on fresher level interviews. There are multiple ways to compare two String like equals() method, equalsIgnoreCase() etc, You can also see [4 ways to compare String in Java](http://javarevisited.blogspot.sg/2012/03/how-to-compare-two-string-in-java.html) for more examples. Main thing which interviewer checks is that whether candidate mentioned equality operator or not "==", comparing String with equality operator is common mistake which works in some case and doesn't work in other. next String interview question is follow-up up of this.

**7) Can we compare String using == operator? What is risk?**

As discussed in previous String question, You can compare String using equality operator but that is not suggested or advised because equality operator is used to compare primitives and [equals() method](http://javarevisited.blogspot.sg/2011/02/how-to-write-equals-method-in-java.html) should be used to compare objects. As we have seen in [pitfall of autoboxing in Java](http://javarevisited.blogspot.sg/2012/07/auto-boxing-and-unboxing-in-java-be.html) that how equality operator can cause subtle issue while comparing primitive to Object, any way String is free from that issue because it doesn't have corresponding primitive type and not participate in autoboxing. Almost all the time comparing String means comparing contents of String i.e. characters and equals() method is used to perform character based comparison. equals() return true if two String points to same object or two String has same contents while == operator returns true if two String object **points to same object** but return false if two different String object contains same contents. That explains why sometime it works and sometime it doesn't. In short [always use equals method in Java](http://javarevisited.blogspot.sg/2011/02/how-to-write-equals-method-in-java.html) to check equality of two String object.

**8) How substring method work in Java**

This is one of the [tricky Java question](http://java67.blogspot.sg/2012/09/top-10-tricky-java-interview-questions-answers.html) relate to String and until you are familiar with internals of String class, its difficult to answer. Substring shares same character array as original String which can create memory leak if original String is quite big and not required to retain in memory but unintentionally retained by substring which is very small in size and prevents large array from begin claimed during [Garbage collection in Java](http://javarevisited.blogspot.sg/2011/04/garbage-collection-in-java.html). See [How Substring works in Java](http://javarevisited.blogspot.sg/2011/10/how-substring-in-java-works.html) for more details.

**10)What is String pool in Java**

Another [tough Java question](http://java67.blogspot.sg/2012/09/top-10-tough-core-java-interview-questions-answers.html) asked in String interview. String pool is a special storage area in Java heap, mostly located on PerGen space, to store String literals like "abc". When Java program creates a new String using String literal, [JVM](http://javarevisited.blogspot.sg/2011/12/jre-jvm-jdk-jit-in-java-programming.html) checks for that String in pool and if String literal is already present in pool than same object is returned instead of creating a whole new object. String pool check is only performed when you create String as literal, if you create String using new() operator, a new String object will be created even if String with same content is available in pool.

**9) What does intern() method do in Java**

As discussed in previous String interview question, String object crated by new() operator is by default not added in String pool as opposed to String literal. intern() method allows to put an [String object](http://javarevisited.blogspot.sg/2012/08/convert-collection-to-string-in-java.html) into pool.

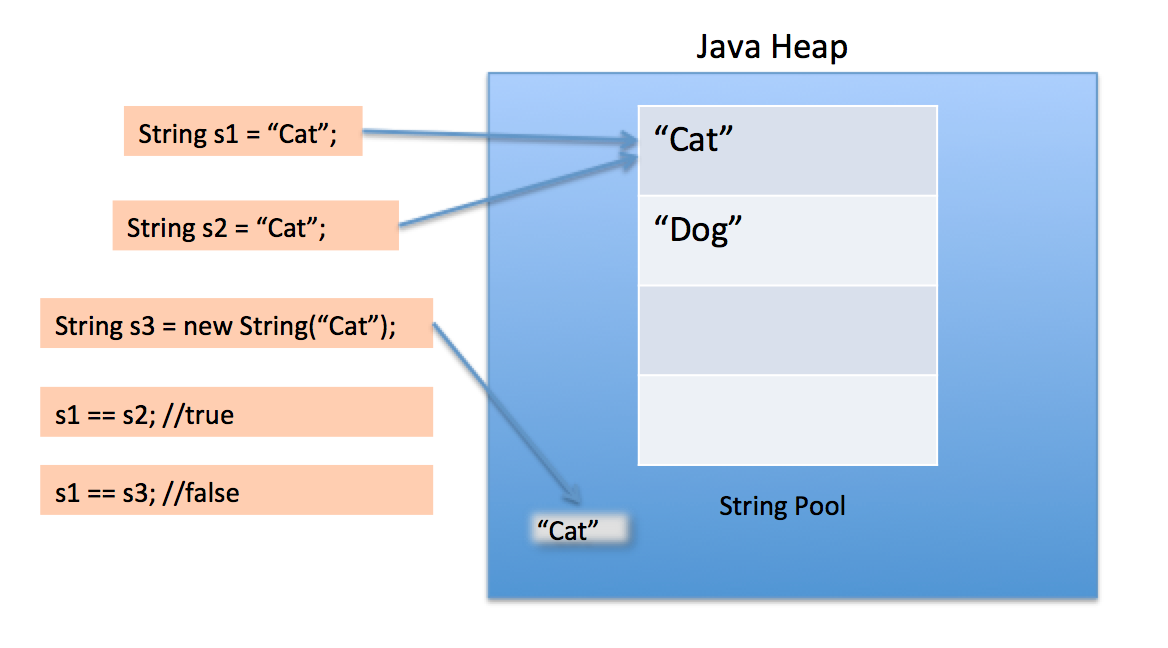
**11) Does String is thread-safe in Java**

If you are familiar with the concept of immutability and [thread-safety](http://javarevisited.blogspot.sg/2011/07/java-multi-threading-interview.html) you can easily answer this String interview question in Java. Since [String is immutable](http://javarevisited.blogspot.sg/2010/10/why-string-is-immutable-in-java.html), it is thread-safe and it can be shared between multiple thread without external synchronization.

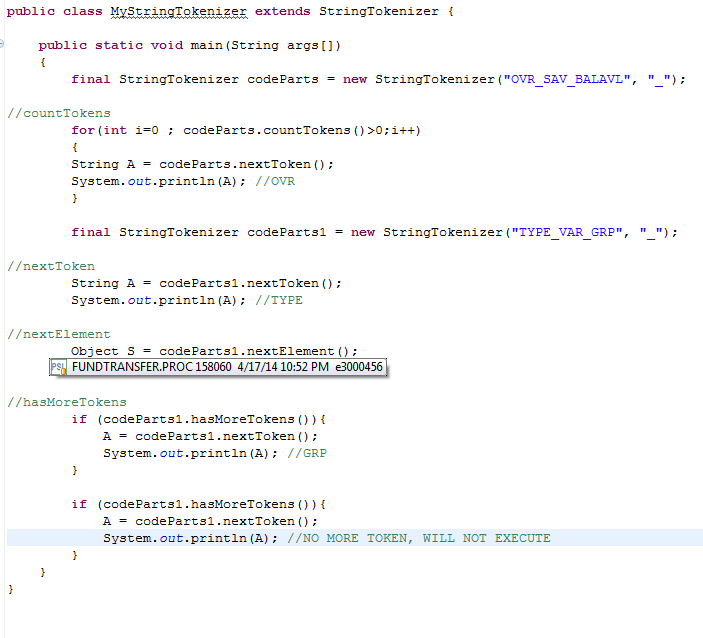
That's all on *Java String interview question*. In Summary there are lot of specifics about String which needs to be know for any one who has started Java programming and these String question will not just help to perform better on Java Interviews but also opens new door of learning about String. I didn't know many String related concepts until I come across these question which motivated to research and learn more about String in Java.

# [Why String is immutable or final in Java](http://javarevisited.blogspot.in/2010/10/why-string-is-immutable-in-java.html)

This is one of the most popular [String Interview questions in Java](http://javarevisited.blogspot.com/2012/10/10-java-string-interview-question-answers-top.html), which starts with discussion of,  What is String, How String in Java is different than String in C and C++, and then shifted towards [what is immutable object in Java](http://javarevisited.blogspot.com/2013/03/how-to-create-immutable-class-object-java-example-tutorial.html) , what are the benefits of immutable object , why do you use it and which scenarios do you use it. This is some time also asked as *"Why String is final in Java"* . Though there could be many possible answer for this question, and only designer of String class can answer this , I think below two does make sense  
  
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](http://javarevisited.blogspot.sg/2012/02/difference-between-instance-class-and.html) , 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](http://javarevisited.blogspot.sg/2012/07/read-file-line-by-line-java-example-scanner.html) 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](http://javarevisited.blogspot.com/2011/04/synchronization-in-java-synchronized.html), Immutability also makes String instance [thread-safe in Java](http://javarevisited.blogspot.sg/2012/01/how-to-write-thread-safe-code-in-java.html), means you don't need to synchronize String operation externally. Another important point to note about String is [memory leak caused by SubString](http://javarevisited.blogspot.sg/2011/10/how-substring-in-java-works.html), 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](http://javarevisited.blogspot.com/2011/02/how-hashmap-works-in-java.html).  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](http://javarevisited.blogspot.sg/2011/10/override-hashcode-in-java-example.html) 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](http://javarevisited.blogspot.sg/2012/07/when-class-loading-initialization-java-example.html), 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"  
  
  
Security and String pool being primary reason of making String immutable, I believe there could be some more very convincing reasons as well, Please post those reasons as comments and I will include those on this post. By the way, above reason holds good to answer, another [Java interview questions](http://javarevisited.blogspot.sg/2011/04/top-20-core-java-interview-questions.html) **"Why String is final in Java".** Also to be immutable you have to be final, so that your subclass doesn't break immutability.  what do you guys think ?



**StringTokenizer Class**



# What is difference between Comparator and Comparable interfaces in Java?

Posted on [January 30, 2014](http://www.somanyword.com/2014/01/what-is-difference-between-comparator-and-comparable-interfaces-in-java/) by [Manoj Sharma](http://www.somanyword.com/author/admin/)

Both Comparator and Comparable interfaces are designed for objects comparison and to help in objects collection sorting. If any class has implemented Comparator or Comparable interface, objects of that class can be sort with collection utility. But both interfaces have differences in their purpose of implementations. Let us discuss some differences in both interfaces.

**1.** Both interfaces are part of different packages. Comparator is defined in java.util package while Comparable is defined in java.lang package.

**2.** A class should implement Comparable interface by default, if this class support sorting on any default property of that class only. While a class should implement Comparator interface, if user want to sort any class by any custom property of class. For example, Lets have an example of Employee class which has data members or properties like Employee Name, Age, Salary and Department. If Employee class wants default sorting on Salary property then it must implement Comparable interface. In other hand if user of Employee class later on want to sort Employee objects on the basis of Age, then user can create a new class with implementation of Comparator interface. You can see coding example of this scenario below.

**3.** Comparable interface has a method with public int compareTo(Object o); signature, which returns positive, zero or negative integer number in corresponding situation where current object (this object) is greater than, equal or less than the given object. In case of <Comparable interface comparison happens with this or current object. While Comparator interface has a method with public int compare(Object o1, Object o2); signature, which also returns positive, zero or negative integer number in corresponding situation where first object is greater than, equal to or less than second object. In case of Comparator interface, comparison happens in two given objects, it never happens with this or current object.

**4.** It is important to understand the significance of both interfaces. If you are not the author of any class and you want to sort collection of that class, in this case you can only implement Comparator interface, not Comparable, which only an author can implement.

**5.** Comparable implemented classes can be sorted with Collections.sort(List list); utility. Where Comparator implementation is required to pass in sorting utility. For example, Collections.sort(List list, Comparator c);

Let us see coding example to see real implementation of these interfaces. This example will help you to understand it more clearly.

## Equality

Object equality is tested using the == operator, while value equality is tested using the .equals(Object) method.

For example:

String one = new String("abc");

String two = new String("abc");

String three = one;

if (one != two) System.out.println("The two objects are not the same.");

if (one.equals(two)) System.out.println("But they do contain the same value");

if (one == three) System.out.println("These two are the same, because they use the same reference.");

The output is:

The two objects are not the same.

But they do contain the same value

These two are the same, because they use the same reference.

Also, be aware that:

String abc = "abc"

and

String abc = new String("abc");

are different. For example, consider the following:

String letters = "abc";

String moreLetters = "abc";

System.out.println(letters==moreLetters);

The output is:

true

This is due to the compiler and runtime efficiency. In the compiled class file only one set of data "abc" is stored, not two. In this situation only one object is created, therefore the equality is true between these object. However, consider this:

String data = new String("123");

String moreData = new String("123");

System.out.println(data==moreData);

The output is:

false

Even though one set of data "123" is stored in the class, this is still treated differently at runtime. An explicit instantiation is used to create the String objects. Therefore, in this case, two objects have been created, so the equality is false. It is important to note that "==" is always used for object equality and does not ever refer to the values in an object. Always use .equals when checking looking for a "meaningful" comparison.

## Immutable Objects / Wrapper Class Caching

Since Java 5, wrapper class caching was introduced. The following is an examination of the cache created by an inner class, IntegerCache, located in the Integer cache. For example, the following code will create a cache:

Integer myNumber = 10

or

Integer myNumber = Integer.valueOf(10);

256 Integer objects are created in the range of -128 to 127 which are all stored in an Integer array. This caching functionality can be seen by looking at the inner class, IntegerCache, which is found in Integer:

private static class IntegerCache

{

private IntegerCache(){}

static final Integer cache[] = new Integer[-(-128) + 127 + 1];

static

{

for(int i = 0; i < cache.length; i++)

cache[i] = new Integer(i - 128);

}

}

public static Integer valueOf(int i)

{

final int offset = 128;

if (i >= -128 && i <= 127) // must cache

{

return IntegerCache.cache[i + offset];

}

return new Integer(i);

}

So when creating an object using Integer.valueOf or directly assigning a value to an Integer within the range of -128 to 127 the same object will be returned. Therefore, consider the following example:

Integer i = 100;

Integer p = 100;

if (i == p) System.out.println("i and p are the same.");

if (i != p) System.out.println("i and p are different.");

if(i.equals(p)) System.out.println("i and p contain the same value.");

The output is:

i and p are the same.

i and p contain the same value.

It is important to note that object i and p only equate to true because they are the same object, the comparison is not based on the value, it is based on object equality. If Integer i and p are outside the range of -128 or 127 the cache is not used, therefore new objects are created. When doing a comparison for value always use the “.equals” method. It is also important to note that instantiating an Integer does not create this caching. So consider the following example:

Integer i = new Integer (100);

Integer p = new Integer(100);

if(i==p) System.out.println(“i and p are the same object”);

if(i.equals(p)) System.out.println(“ i and p contain the same value”);

In this circumstance, the output is only:

i and p contain the same value

Remember that “==” is always used for object equality, it has not been overloaded for comparing unboxed values.

This behavior is documented in the [Java Language Specification](http://java.sun.com/docs/books/jls) [section 5.1.7](http://java.sun.com/docs/books/jls/third_edition/html/conversions.html#5.1.7). Quoting from there:

If the value *p* being boxed is true, false, a byte, a char in the range \u0000 to \u007f, or an int or short number between -128 and 127, then let *r1* and *r2* be the results of any two boxing conversions of *p*. It is always the case that *r1* == *r2*.

The other wrapper classes (Byte, Short, Long, Character) also contain this caching mechanism. The Byte, Short and Long all contain the same caching principle to the Integer object. The Character class caches from 0 to 127. The negative cache is not created for the Character wrapper as these values do not represent a corresponding character. There is no caching for the Float object.

BigDecimal also uses caching but uses a different mechanism. While the other objects contain a inner class to deal with caching this is not true for BigDecimal, the caching is pre-defined in a static array and only covers 11 numbers, 0 to 10:

// Cache of common small BigDecimal values.

private static final BigDecimal zeroThroughTen[] = {

new BigDecimal(BigInteger.ZERO, 0, 0),

new BigDecimal(BigInteger.ONE, 1, 0),

new BigDecimal(BigInteger.valueOf(2), 2, 0),

new BigDecimal(BigInteger.valueOf(3), 3, 0),

new BigDecimal(BigInteger.valueOf(4), 4, 0),

new BigDecimal(BigInteger.valueOf(5), 5, 0),

new BigDecimal(BigInteger.valueOf(6), 6, 0),

new BigDecimal(BigInteger.valueOf(7), 7, 0),

new BigDecimal(BigInteger.valueOf(8), 8, 0),

new BigDecimal(BigInteger.valueOf(9), 9, 0),

new BigDecimal(BigInteger.TEN, 10, 0),

};

As per Java Language Specification(JLS) the values discussed above are stored as immutable wrapper objects. This caching has been created because it is assumed these values / objects are used more frequently.

## Incrementing Values

Be careful of the post-increment operator:

int x = 5;

x = x++;

System.out.println( x );

The output is:

5

Remember that the assignment completes before the increment, hence post-increment. Using the pre-increment will update the value before the assignment. For example:

int x = 5;

x = ++x;

System.out.println( x );

The output is:

6

## Garbage Collection

Overriding "finalize()" will allow you to define you own code for what is potentially the same concept as a destructor. There are a couple of important points to remember:

* "finalize()" will only ever by called once (at most) by the Garbage Collector.
* It is never a guarantee that "finalize()" will be called i.e. that an object will be garbage collected.
* By overriding "finalize()" you can prevent an object from ever being deleted. For example, the object passes a reference of itself to another object.
* Garbage collection behaviour differs between JVMs.

## Boolean Assignment

Everyone appreciates the difference between "==" and "=" in Java. However, typos and mistakes are made, and often the compiler will catch them. However, consider the following:

boolean theTruth = false;

if (theTruth = true)

{

System.out.println("theTruth is true");

}

else

{

System.out.println("theTruth is false;");

}

The result of any assignment expression is the value of the variable following the assignment. Therefore, the above will always result in "theTruth is true". This only applies to booleans, so for example the following will not compile and would therefore be caught by the compiler:

int i = 1;

if(i=0) {}

As "i" is and integer the comparison would evaluate to (i=0) as 0 is the result of the assignment. A boolean would be expected, due the "if" statement.

## Conditions

Be on the look out for any nested "else if". Consider the following code example:

int x = 3;

if (x==5) {}

else if (x<9)

{

System.out.println("x is less than 9");

}

else if (x<6)

{

System.out.println("x is less than 6");

}

else

{

System.out.println("else");

}

Produces the output:

x is less then 9

So even though the second "else if" would equate to "true" it is never reached. This is because once an "else if" succeeds the remaining conditions will be not be processed.