**Q. How to Create an immutable class in Java?**

**Ans.**  To create an immutable class in Java, you have to do the following steps.

1. Declare the **class as final so it can’t be extended**.
2. Make **all fields private** so that direct access is not allowed.
3. **Don’t provide setter methods** for variables.
4. **Make all mutable fields final** so that its value can be assigned only once.
5. **Initialize all the fields via a constructor performing deep copy.**
6. **Perform cloning of objects in the getter methods** to return a copy rather than returning the actual object reference.

**FinalClassExample.java**

**package com.journaldev.java;**

**import java.util.HashMap;**

**import java.util.Iterator;**

**public final class FinalClassExample {**

private final int id;

private final String name;

private final HashMap<String,String> testMap;

public int getId() {

return id;

}

public String getName() {

return name;

}

/\*\*

\* Accessor function for mutable objects

\*/

public HashMap<String, String> getTestMap() {

//return testMap;

return (HashMap<String, String>) testMap.clone();

}

/\*\*

\* Constructor performing Deep Copy

\* @param i

\* @param n

\* @param hm

\*/

public FinalClassExample(int i, String n, HashMap<String,String> hm){

System.out.println("Performing Deep Copy for Object initialization");

this.id=i;

this.name=n;

HashMap<String,String> tempMap=new HashMap<String,String>();

String key;

Iterator<String> it = hm.keySet().iterator();

while(it.hasNext()){

key=it.next();

tempMap.put(key, hm.get(key));

}

this.testMap=tempMap;

}

/\*\*

\* Constructor performing Shallow Copy

\* @param i

\* @param n

\* @param hm

\*/

/\*\*

public FinalClassExample(int i, String n, HashMap<String,String> hm){

System.out.println("Performing Shallow Copy for Object initialization");

this.id=i;

this.name=n;

this.testMap=hm;

}

\*/

/\*\*

\* To test the consequences of Shallow Copy and how to avoid it with Deep Copy for creating immutable classes

\* @param args

\*/

public static void main(String[] args) {

HashMap<String, String> h1 = new HashMap<String,String>();

h1.put("1", "first");

h1.put("2", "second");

String s = "original";

int i=10;

FinalClassExample ce = new FinalClassExample(i,s,h1);

//Lets see whether its copy by field or reference

System.out.println(s==ce.getName());

System.out.println(h1 == ce.getTestMap());

//print the ce values

System.out.println("ce id:"+ce.getId());

System.out.println("ce name:"+ce.getName());

System.out.println("ce testMap:"+ce.getTestMap());

//change the local variable values

i=20;

s="modified";

h1.put("3", "third");

//print the values again

System.out.println("ce id after local variable change:"+ce.getId());

System.out.println("ce name after local variable change:"+ce.getName());

System.out.println("ce testMap after local variable change:"+ce.getTestMap());

HashMap<String, String> hmTest = ce.getTestMap();

hmTest.put("4", "new");

System.out.println("ce testMap after changing variable from accessor methods:"+ce.getTestMap());

}

}

Output of the above example program is:

Performing Deep Copy for Object initialization

true

false

ce id:10

ce name:original

ce testMap:{2=second, 1=first}

ce id after local variable change:10

ce name after local variable change:original

ce testMap after local variable change:{2=second, 1=first}

ce testMap after changing variable from accessor methods:{2=second, 1=first}

Why Deep Copy is important for immutability?

Let’s comment the constructor providing deep copy and uncomment the constructor providing a shallow copy.

Also, uncomment the return statement in the getTestMap() method that returns the actual object reference.

Run the program after all the changes are done. It will produce the following output.

Performing Shallow Copy for Object initialization

true

true

ce id:10

ce name:original

ce testMap:{2=second, 1=first}

ce id after local variable change:10

ce name after local variable change:original

ce testMap after local variable change:{3=third, 2=second, 1=first}

ce testMap after changing variable from accessor methods:{3=third, 2=second, 1=first, 4=new}

As you can see from the output, HashMap values got changed because of shallow copy in the constructor.

It’s happening because of the direct reference to the original object in the getter function.

That’s all for an immutable class in Java. We also learned the importance of deep copy for immutable classes.

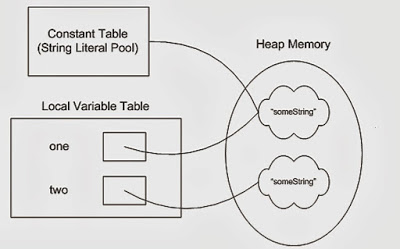
**Q) What is String in Java? Is String is data type?**

**Ans**. String in Java is not a primitive data type like int, long or double. The string is a class or in more simple term a user defined type. This is confusing for someone who comes from C background. String is defined in java.lang package and wrappers its content in a character array. String provides equals() method to compare two String and provides various other methods to operate on String like toUpperCase() to convert String into upper case, replace() to replace String contents, substring() to get substring, split() to split long String into multiple String.  
  
**Q. Why String is Immutable or Final in Java**

**Ans**. The string is Immutable in Java because **String objects are cached in the String pool**. **Since cached String literals are shared between multiple clients there is always a risk, where one client's action would affect all another client.** For example, if one client changes the value of the String "Test" to "TEST", all other clients will also see that value as explained in the first example. Since caching of String objects was important from performance reason this risk was avoided by making String class Immutable. At the same time, String was made final so that no one can compromise invariant of String class like Immutability, Caching, hashcode calculation, etc by extending and overriding behaviors. Another reason why String class is immutable could die **due to HashMap**.

Since Strings are very popular as the HashMap key, it's important for them to be immutable so that they can retrieve the value object which was stored in HashMap. Since HashMap works in the principle of hashing, which requires the same has value to function properly. Mutable String would produce two different hashcodes at the time of insertion and retrieval if contents of String was modified after insertion, potentially losing the value object in the map.

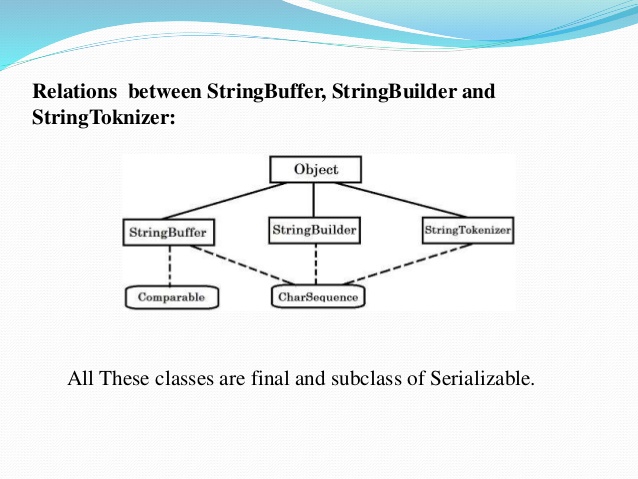
**1**) Imagine **String pool facility** without making string immutable , its not possible at all because in case of string pool one string object/literal e.g. "Test" has referenced by many reference variables, so if any one of them change the value others will be automatically gets affected i.e. lets 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. Here is a nice diagram which shows how String literals are created in heap memory and String literal pool.



**2**) String has been **widely used as parameter** for many Java classes e.g. for opening network connection, we can pass hostname and port number as string, we can pass **database URL** as a string for opening database connection, we can open any file in Java by passing the name of the file as argument to File I/O classes.  
In case, **if String is not immutable**, **this would lead serious security threat**, I mean someone can access to any file for which he has authorization, and then can change the file name either deliberately or accidentally and gain access to that file. Because of immutability, we don't need to worry about that kind of threats. This reason also gels 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 share 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 we don't need to synchronize String operation externally. Another important point to note about String is the 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. . In short because String is immutable, no one can change its contents once created which guarantees hashCode of String to be same on multiple invocations.  
**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"  
**Security and String pool being primary reason of making String immutable,**

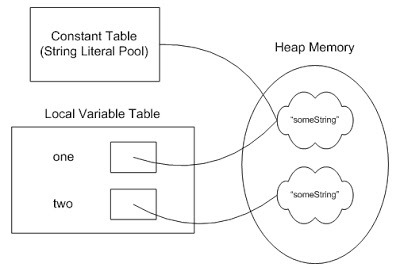
**Q. Difference between String, StringBuffer, and StringBuilder**

**Ans**. We are not aware that String is immutable and final in Java and every modification in String creates a new String object. For example, when we get the substring, we get a new String, when we convert uppercase String to lowercase, a new String is created. Even when we remove space by calling the trim() method, a new String is returned. So, now the big question is how do we manipulate String in Java without creating String garbage? StringBuilder and StringBuffer are the answer to this question. StringBuffer is old class but StringBuilder is newly added in Java 5 along with major improvements in Enum, Generics, varargs methods and Autoboxing in Java.

properties of String Class in Java  
**1**) The string is immutable in Java:  String is by design immutable in Java we can check this post for a reason. Immutability offers a lot of benefit to the String class e.g. his hashcode value can be cached which makes it a faster hashmap key and one of the reasons why String is a popular key in HashMap. Because String is final it can be safely shared between multiple threads  without any extra synchronization.  
  
  
2) When we represent a string in double quotes like **"abcd"** they are referred as String literal and **String literals are created in String pools**. When we compare two String literals using equality operator "==" it returns true because they are the actually same instance of String. Anyway comparing an object with equality operator is bad practice in Java and we should always use the equals method to check equality.  
**3**) The "+" operator is overloaded for String and used to concatenated two string. Internally "+" operation is implemented using either StringBuffer or StringBuilder.   
**4**) **Strings are backed up by character** **Array** and **represented in UTF-16 format.** By the way, this behaviour can cause a memory leak in String because same character array is shared between source String and SubString which can prevent source String from being garbage collected.   
**5**) **String class overrides equals() and hashcode() method** and two Strings are considered to be equal if they contain exactly same character in same order and in same case. If we want to ignore case comparison of two strings consider using equalsIgnoreCase() method. Another worth noting point is that equals method must be consistent with compareTo() method for String because SortedSet and SortedMap e.g. TreeMap use the compareTo method to compare String in Java.  
7) toString() method provides String representation of any object and it's declared in Object class and it's recommended for other class to implement this and provide String representation.  
9) In Java, we can create String from char array, byte array, another string, from StringBuffer or from StringBuilder. Java String class provides a constructor for all of these.  
10) Even though all StringBuffer, StringBuilder, and String are from same type hierarchy i.e. they extend from CharSequence interface, we cannot cast StringBuilder to StirngBuffer or StringBuilder to String in Java. It will throw java.lang.ClasscastException, if we tried to cast even StringBuffer to String in Java.  
and String in Java:  


**Problem with String in Java**

One of its **biggest strength** **Immutability is also biggest problem** of Java String if not used correctly. Many times we create a String and then perform a lot of operation on them e.g. converting a string into uppercase, lowercase , getting substring out of it , concatenating with other string etc. Since String is an immutable class every time a new String is created and the older one is discarded which creates lots of temporary garbage in the heap  
If String is created using String literal they remain in the String pool. To resolve this problem Java provides us, two Classes, StringBuffer and StringBuilder. String Buffer is an older class but StringBuilder is relatively new and added in JDK 5.

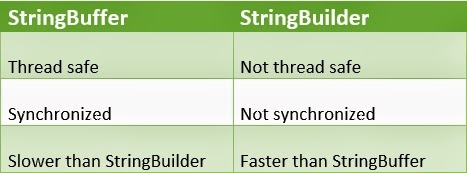


Differences between String and StringBuffer in Java

The main difference between String and StringBuffer is **String is immutable** while **StringBuffer is mutable** means we can modify a StringBuffer object once we created it without creating any new object. This mutable property makes StringBuffer an ideal choice for dealing with Strings in Java.  
We can convert a StringBuffer into String by its toString() method.

Difference between StringBuilder and StringBuffer in Java

**StringBuffer** is very good with mutable String but it has **one disadvantage all its public methods are synchronized**which makes it thread-safe but same time **slow**. In JDK 5 they provided a similar class called **StringBuilder** in Java which is a copy of StringBuffer but **without synchronization**. Try to use StringBuilder whenever possible it **performs better in** most of the cases than StringBuffer class.  
We can also use "+" for concatenating two string because "+" operation is internally implemented using either StringBuffer or StringBuilder in Java.



In summary here is list of difference between StringBuffer, String, and StringBuilder in Java :

1) The String object is immutable in Java but StringBuffer and StringBuilder are mutable objects.

2) StringBuffer is synchronized while StringBuilder is not which makes StringBuilder faster than StringBuffer.

3) Concatenation operator "+" is internally implemented using either StringBuffer or StringBuilder.

4) Use String if we require immutability, use StringBuffer in java if we need mutable + thread-safety and use StringBuilder in Java if we require mutable + without thread-safety.

**Q. Why character array is better than String for Storing password in Java**

**Ans**. some reasons why should we use char[] for storing password than String.  
**1**) **Since Strings are immutable in Java** if we store the **password as plain text it will be available in memory until the Garbage collector** clears it and since String are used in String pool for reusability there is a pretty high chance that it will remain in memory for long duration, which poses a security threat.   
Since anyone who has access to memory dump can find the password in clear text and that's another reason **we should always use an encrypted password than plain text**. Since Strings are immutable there are no way contents of Strings can be changed because any change will produce new String, while if we char[] we can still set all his elements as blank or zero. So Storing the password in character array clearly mitigates security risk of stealing passwords.

2) Java itself recommends using getPassword() method of JPasswordField which returns a char[] and deprecated getText() method which returns password in clear text stating security reason. Its good to follow advice from Java team and adhering to a standard rather than going against it.  
3) With String there is always a risk of printing plain text in log file or console but if use Array we won't print contents of array instead its memory location get printed. though not a real reason but still make sense.

String strPassword="**Unknown**";  
char[] charPassword= new char**[]{'U','n','k','w','o','n'};**  
System.out.println("String password: " + strPassword);  
System.out.println("Character password: " + charPassword);  
  
String password: Unknown  
Character password: [C@110b053

**Q. How to Compare two String in Java - String Comparison Example**

**Ans**.

Here are four examples of comparing String in Java

1) String comparison using equals method

2) String comparison using equalsIgnoreCase method

2) String comparison using compareTo method

4) String comparison using compareToIgnoreCase method

Compare two String using equals method in Java

equals()method compare two Strings for content equality. So if two string contains the same letters, in the same order and in some case they will be equals by equals() method. equals() method is defined in Object class and String class overrides that for character-based comparison. Check 5 tips to override equals()method in Java for more details on equals and its implementation. For example for two Strings "Sony" and "Sony", equals() will return true but for "Sony" and "SONY" it will return false.

Compare String using the equalsIgnoreCase method in Java

equalsIgnoreCase is more liberal than equals and compares two strings ignoring their case. So if two String contains the same characters and in the same order despite there case e.g. lower case, upper case, Capital case, or Camel case they will be equal by equalsIgnoreCase.  
For example "sony" and "SONY" two Strings will be the same by equalsIgnoreCase and it will return true but "Sony" and "Samsung" will not be the same and it will return false because they don't contain same characters.

Comparing String using compareTo

compareTo is actual comparison method unlike equals()and equalsIgnoreCase() and tell us whether two Strings are lexicographically equal, precedes or follows each other. if we want to sort Strings lexicographically compareTo() method is used.   
This is also called the natural order of String.   returns zero if two Strings are same, less than zero if calling string comes before argument string and greater than zero if calling string comes later than argument string as shown in the example below. 

Compare String using compareToIgnoreCase

Similar to compareTo() method with ignoring case like equalsIgnoreCase() and return the same values as returned by compareTo during String comparison.

Don't use "==" for String comparison

Many Java programmer makes the mistake of using "==" for string comparison. "==" just check if two reference variables are pointing two same objects in Java heap and since String is immutable in Java and maintained in String pool two String literal refer same String object which gives the sense that "==" can be used to compare string which is incorrect. always use equals() method for equality check and compare method for actual string comparison.

Another way of comparing String is by writing a custom Comparator in Java. write our comparison logic in compare() method and then we can use that logic to compare two strings.

public class StringComparisonExample {  
    public static void main(String args[]) {  
        String tv = "Bravia";  
        String television = "Bravia";  
        // String compare example using equals  
        if (tv.**equals**(television)) {  
            System.out.println("Both tv and television contains same letters and equal by equals method of String");  
        }  
        // String compare example in java using compareTo  
        if (tv.**compareTo**(television) == 0) {  
            System.out.println("Both tv and television are equal using compareTo method of String");  
        }  
        television = "BRAVIA";  
        // Java String comparison example using equalsIgnoreCase  
        if (tv.equalsIgnoreCase(television)) {  
            System.**out**.println("tv and television are equal by equalsIgnoreCase method of String");  
        }  
        // String comparison example in java using CompareToIgnoreCase  
        if (tv.compareToIgnoreCase(television) == 0) {  
            System.out.println("tv and television are same by compareToIgnoreCase of String");  
        }  
        String sony = "Sony";  
        String samsung = "Samsung";  
        // lexicographical comparison of String in Java with ComapreTo  
        if (sony.compareTo(samsung) > 0) {  
            System.out.println("Sony comes after Samsung in lexicographical order");  
        } else if (sony.compareTo(samsung) < 0) {  
            System.out.println("Sony comes before Samsung in lexicographical order");  
        }  
    }  
  
}  
  
Output:  
Both tv and television contains same letters and equal by equals method of String  
Both tv and television are equal using compareTo method of String  
tv and television are equal by equalsIgnoreCase method of String  
tv and television are the same as compareToIgnoreCase of String  
Sony comes after Samsung in lexicographical order

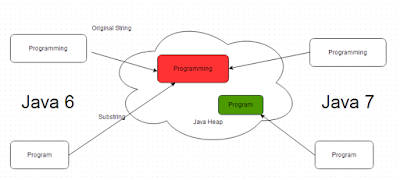
These were some common examples of comparing string in Java. Avoid the common mistake of using equality operator “==” for comparing String instead of always use equals() method.

**Q.) Can we compare String using == operator? What is the risk?**

**Ans**. while == operator returns true if two String object points to the same object but return false if two different String object contains same contents.

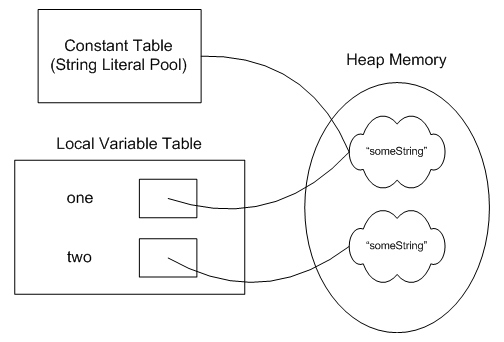
**Q) How does substring method work in Java?**

**Ans**. Substring shares same character array as original String which **can create a 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/2011/10/how-substring-in-java-works.html)

**Q.) What is String pool in Java?**

**Ans**. **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 checks for that String in the pool and if String literal is already present in the pool than the same object is returned instead of creating a whole new object. **String pool check is only performed when we create String as literal**, **if we create String using new() operator, a new String object will be created even if String with the same content is available in the pool.**



**Q.) What does the intern() method do in Java?**

**Ans**. **String object created by new() operator** is **by default not added in String pool** as opposed to String literal. The **intern method allows putting a String object into a pool**.

**Q.) Is string thread-safe in Java?**

**Ans. Yes Since String is immutable, it is thread-safe** and it can be shared between multiple threads without external synchronization.

**Q. How SubString method works in Java - Memory Leak Fixed in JDK 1.7**

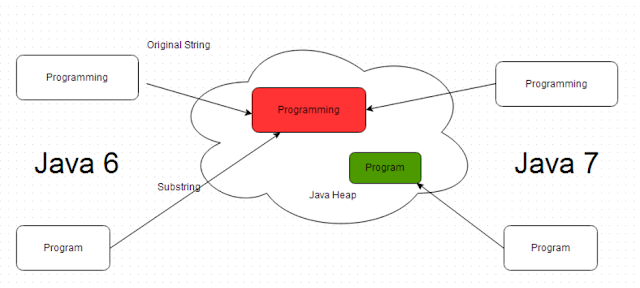
**Ans**.  java.lang.String class has also grown into some change in Java 1.7 version and offset and count variable which is used to track positions are removed from String. This may save some bytes with each String instance, but not sharing original array makes substring perform linearly, as compared to constant time previously. Anyway, it's worth to remove any string related memory leak in Java. Having said that, if we have not yet upgraded our Server to Java 7 and still working on Java 1.6 updates, this is one thing, which is worth knowing.

One version of the substring method takes just beginIndex, and returns part of String started from beginIndex till end, while other takes two parameters, beginIndex and endIndex, and returns part  of String starting from beginIndex to endIndex-1. He also stressed that every time we call  substring() method in Java,  it will return a new String because String is immutable in Java.

Next question was, what will happen if beginIndex is equal to length in substring(int beginIndex), no it won't throw IndexOutOfBoundException instead it will return empty String. Same is the case when beginIndex and endIndex is equal, in case of second method. It will only throw StringIndexBoundException when beginIndex is negative, larger than endIndex or larger than length of String.

If we look substring method inside String class, we will figure out that it calls String (int offset, int count, char value []) constructor to create new String object. What is interesting here is, value[], which is the same character array used to represent original string. So what's wrong with this?

In case If we have still not figured it out, If the original string is very long, and has array of size 1GB, no matter how small a substring is, it will hold 1GB array.  This will also stop original string to be garbage collected, in case if doesn't have any live reference. **This is a clear case of memory leak in Java**, where memory is retained even if it's not required. That's how substring method creates memory leak.



How SubString in Java works

Obviously next question from interviewer would be,  how do we deal with this problem? Though we can not go, and change Java substring method, we can still make some work around, in case we are creating substring of significant longer String. Simple solution is to trim the string, and keep size of character array according to length of substring. Luckily java.lang.String has constructor to do this, as shown in below example.

// comma separated stock symbols from NYSE

String listOfStockSymbolsOnNYSE = getStockSymbolsForNYSE();

//calling String(string) constructor

String apple = new String(

listOfStockSymbolsOnNYSE.substring(appleStartIndex, appleEndIndex)

);

If we look code on java.lang.String class, we will see that this constructor trim the array, if it’s bigger than String itself.

public String(String original) {

if (originalValue.length > size) {

// The array representing the String is bigger than the new

// String itself. Perhaps this constructor is being called

// in order to trim the baggage, so make a copy of the array.

int off = original.offset;

v = Arrays.copyOfRange(originalValue, off, off+size);

} else {

// The array representing the String is the same

// size as the String, so no point in making a copy.

v = originalValue;

}

...

}

**Another way to solve this problem is to call intern() method on substring**, which will than fetch an existing string from pool or add it if necessary. Since the String in the pool is a real string it only take space as much it requires. It’s also worth noting that sub-strings are not internalized, when we call intern() method on original String.  
The most developer successfully answers first three questions, which is related to usage of substring, but they get stuck on last two, How substring creates memory leak or How substring works. It's not completely there fault, because what we know is that every time substring() returns new String which is not exactly true, since it’s backed by same character array.

**Q. Java Tips and Best practices to avoid NullPointerException in Java Applications.**

**Ans**.

Following these Java tips also minimize the number of !=null check, which litter a lot of Java code.

**1**) Call equals() and equalsIgnoreCase() method on known String literal rather unknown object

Always call equals() method on known String which is not null. Since equals() method is symmetric, calling a.equals(b) is same as calling b.equals(a), and that’s why many programmer don’t pay attention on object a and b. One side effect of this call can result in NullPointerException, if caller is null.

Object unknownObject = null;

//wrong way - may cause NullPointerException

**if(unknownObject.equals("knownObject")){**

   System.err.println("This may result in NullPointerException if unknownObject is null");

}

//right way - avoid NullPointerException even if unknownObject is null

**if("knownObject".equals(unknownObject)){**

    System.err.println("better coding avoided NullPointerException");

}

2**) Prefer valueOf() over toString()** where both return same result

Since calling toString() on null object throws NullPointerException, if we can get same value by calling valueOf() then prefer that, as passing null to  valueOf() returns "null", specially in case of wrapper classes  like Integer, Float, Double or BigDecimal.

BigDecimal bd = getPrice();

System.out.println(**String.valueOf(bd)); //doesn’t throw NPE**

System.out.println(**bd.toString()); //throws "Exception in thread "main"** java.lang.NullPointerException"

3) **Using null safe methods and libraries**

There are lot of open source library out there, which does the heavy lifting of checking null for we. One of the most common one is StringUtils from Apache commons. We can use StringUtils.isBlank(), isNumeric(), isWhiteSpace() and other utility methods without worrying of  NullPointerException.

//StringUtils methods are null safe, they don't throw NullPointerException

System.out.println(StringUtils**.isEmpty**(null));

System.out.println(StringUtils.isBlank(null));

System.out.println(StringUtils.isNumeric(null));

System.out.println(StringUtils.isAllUpperCase(null));

Output:

true

true

false

false

**4) Avoid returning null from a method, instead, return an empty collection or an empty array**. By returning empty collection or empty array we make sure that basic calls like size(), length() don't fail with NullPointerException. Collections class provides convenient empty List, Set and Map as Collections.**EMPTY**\_LIST, Collections.EMPTY\_SET and Collections.**EMPTY**\_MAP which can be used accordingly.   
  
Here is a code example

public List getOrders(Customer customer){

**List result = Collections.EMPTY\_LIST;**

**return result;**

}

Similarly, we can use Collections.EMPTY\_SET and Collections.EMPTY\_MAP instead of returning null.

5)  **Use of annotation @NotNull and @Nullable**

While writing method we can define contracts about nullability, by declaring whether a method is null safe or not, by using annotations like @NotNull and @Nullable. Modern days compiler, IDE or tool can read this annotation and assist we to put a missing null check, or may inform we about an unnecessary null check, which is cluttering our code. IntelliJ IDE and findbugs already supports such annotation.   
These annotations are also part of JSR 305, but even in the absence of any tool or IDE support, this  annotation itself work as documentation. By looking @NotNull and @Nullable, programmer can himself decide whether to check for null or not.   
6)  **Avoid unnecessary autoboxing and unboxing in our code**

Despite of other disadvantages like creating temporary object, autoboxing are also prone to NullPointerException, if the wrapper class object is null. For example,  following code will fail with NullPointerException if person doesn't have phone number and instead return null.

Person ram = new Person("ram");

int phone = ram.getPhone();

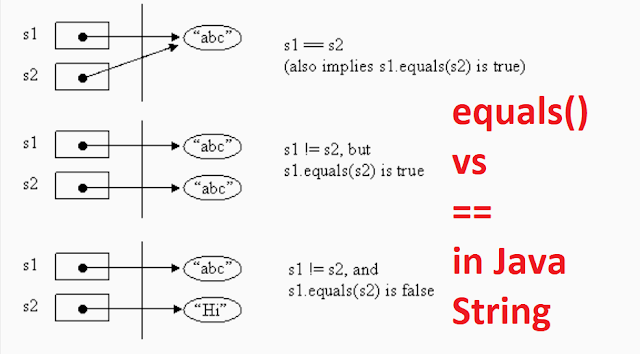
Not just equality but < , > can also throw NullPointerException if used along autoboxing and unboxing.

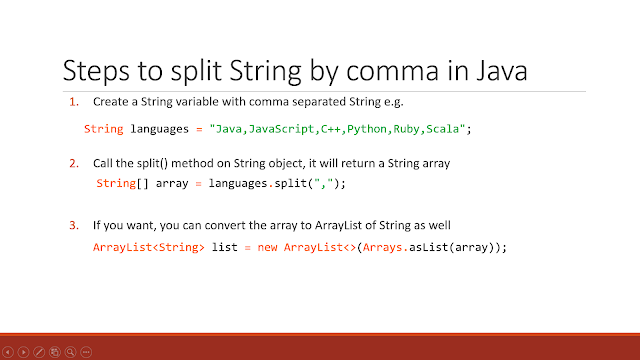
7) **Follow Contract and define reasonable default value**

One of the best way to avoid NullPointerException in Java is as simple as defining contracts and following them. Most of the NullPointerException occurs because Object is created with incomplete information or all required dependency is not provided. If we don't allow to create incomplete object and gracefully deny any such request we can prevent lots of NullPointerException down the road.   
Similarly if  Object is allowed to be created, than we should work with reasonable default value. for example an Employee object can not be created without id and name, but can have an optional phone number.   
Now if Employee doesn't have phone number than instead of returning null, return default value like zero, but that choice has to be carefully taken sometime checking for null is easy rather than calling an invalid number.   
**8**)  If we are using database for storing our domain object such as Customers, Orders etc than we should define our null-ability constraints on database itself. Since database can acquire data from multiple sources, having null-ability check in DB will ensure data integrity. Maintaining null constraints on database will also help in reducing null check in Java code. While loading objects from database we will be sure, which field can be null and which field is not null, this will minimize unnecessary != null check in code.

9) Use Null Object Pattern

This is another way of avoiding NullPointerExcpetion in Java. If a method returns an object, on which caller, perform some operations e.g. Collection.iterator() method returns Iterator, on which caller performs traversal. Suppose if a caller doesn’t have any Iterator, it can return Null object instead of null.   
The Null object is a special object, which has different meaning in different context, for example, here an empty Iterator, calling hasNext() on which returns false, can be a null object.   
Similarly in case of method, which returns Container or Collection types, empty object should be used instead of returning null.





**Q. Useful Methods from String class**1) indexOf  -Returns index of a character from String  
2) toCharArray -Converts String to a character array, actually, return the copy of internal character array from a String object.  
3) valueOf- Used to convert int, long, float, the double and other data type to String in Java. See this post to learn more about how to convert String to an integer in Java.  
4) charAt -Returns the character at given index  
5) compareTo -Compares this String to given String. it returns a positive integer if this String comes after given String in lexicographic order, a negative integer if this string comes before and zero if both strings is equal.

6) concat -Concatenates the specified string to the end of this string.  
7) contains -Returns true if and only if this string contains the specified sequence of char values. See here for an example of this method.  
8) startsWith -Checks if this string starts with the specified suffix.  
9) endsWith -Tests if this string ends with the specified suffix.  
10) equals -Compares this string to the specified object.  
11) equalsIgnoreCase -Compares this string to given string and ignore case. see here for an example  
**12) format** -Returns a formatted string using the specified format string and arguments. See here for an example of this method.  
**13) length** -Returns the number of character in a given String. See here for an example of this method.  
**14) lastIndexOf** -Returns the index of given character from the end.  
**15) matches** -Tells whether or not this string matches the given regular expression. See here for an example of this method.  
**16) replace** -Returns a string resulting from replacing all occurrences of given character with new character. See here for an example of this method.  
**17) replaceAll** -Replaces each substring of this string that matches the given regular expression with the given replacement.  
**18) split** -Used to break String into small pieces or substring based upon given delimiter. See here for an example of this method.  
**19) substring** -Returns a substring of given range from this String.  See here for an example of this method.  
**20) intern** -Used to put a given String into the pool, also return the reference of String object from the pool if present already.  See here for an example of this method.  
**21) trim** -Used to remove whitespace from both ends of String.  See here for an example of this method.  
**22) toUpperCase and toLowerCase**Used to change the case of this String. Since String is Immutable, a new String is returned.