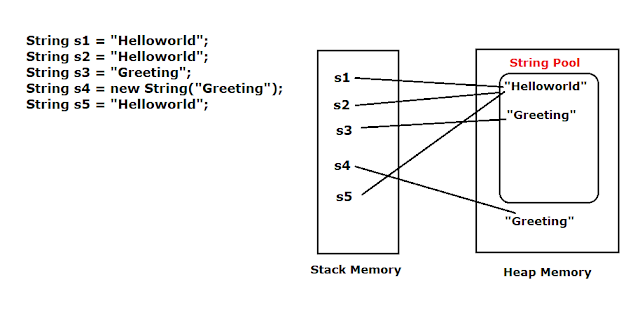
**What is the *String Constant Pool***

String objects are stored in a special memory area known as *String Constant Pool*.

****

* In the above example, only one object will be created. Firstly JVM will not find any string object with the value *"Helloworld"* in string constant pool so it will create a new object. After that it will find the string with the value *"Helloworld"* in the pool, it will not create the new object but will return the reference to the same instance.
* String s1 = "Helloworld";
* String s2 = "Helloworld";

String s5 = "Helloworld";

* The "Greeting" object is created by using the new keyword, this "Greeting" object stored in the heap memory.

String s4 = new String("Greeting");

**2.2. String Interning**

As we know String is *immutable*in Java, the JVM can optimize the amount of memory allocated for them by storing only one copy of each literal String in the pool. This process is called *interning*.

When we create a String variable and assign a value to it, the JVM searches the pool for a String of equal value.

If found, the Java compiler will simply return a reference to its memory address, without allocating additional memory.

If not found, it’ll be added to the pool (interned) and its reference will be returned.

Let’s write a small test to verify this:

private static void stringPool(){

String s1 = "Helloworld";

String s2 = "Helloworld";

String s3 = "Greeting";

String s4 = new String("Greeting");

String s5 = "Helloworld";

// == operator used to check equality of reference variables.

if(s1 == s2){

System.out.println("Helloworld");

}

// s3 and s4 are not equal.

// s3 refer to string pool

// s4 refer to heap memory

if(s3 == s4){

System.out.println("equals");

}else{

System.out.println("not equals");

}

}

**2.3. String object using the new keyword**

What happens if we create String object using *new* keyword and where this object will get a store.

When we create a String via the *new* operator, the Java compiler will create a new object and store it in the heap space reserved for the JVM.

Every String created like this will point to a different memory region with its own address.

Example:

String s3 = "Helloworld";

String s4 = new String("Greeting");

// s3 and s4 are not equal.

// s3 refer to string pool

// s4 refer to heap memory

if(s3 == s4){

System.out.println("equals");

}else{

System.out.println("not equals");

}

**2.4. How to manually intern the String?**

We can manually intern a String in the Java String Pool by calling the *intern()* method on the object we want to intern.

Manually interning the String will store its reference in the pool, and the JVM will return this reference when needed.

String s3 = "Helloworld";

String s4 = new String("Helloworld");

s4 = s4.intern();

// s3 and s4 are not equal.

// s3 refer to string pool

// s4 refer to heap memory

if(s3 == s4){

System.out.println("equals");

}else{

System.out.println("not equals");

}

**2.5. Garbage Collection**

Before *Java 7*, the JVM placed the Java String Pool in the *PermGen*space, which has a fixed size — it can’t be expanded at runtime and is not eligible for garbage collection.

The risk of interning Strings in the *PermGen*(instead of the Heap) is that we can get an *OutOfMemory*error from the JVM if we intern too many Strings.

From Java 7 onwards, the Java String Pool is stored in the Heap space, which is garbage collected by the JVM. The advantage of this approach is the reduced risk of OutOfMemory error because unreferenced Strings will be removed from the pool, thereby releasing memory.

**3. Key Points why String is immutable in Java**

1. *String pool* is possible only because String is *immutable*in Java, this way Java Runtime saves a lot of java heap space because different String variables can refer to a same String variable in the pool. This String Pool process we have seen in this post.
2. As String is immutable it can safely share between many threads which are very important for multi-threaded programming and to avoid any synchronization issues in Java, Immutability also makes String instance thread-safe in Java, means you don't need to synchronize String operation externally.
3. Strings are used in java classloader and immutability provides security that correct class is getting loaded by Classloader.
4. Since String is immutable, its hashcode is cached at the time of creation and it doesn’t need to be calculated again. This makes it a great candidate for a key in a *Map* and it’s processing is fast than other *HashMap* key objects. This is why String is mostly used Object as *HashMap* keys.

Read this post to know how to write your own **immutable class**.

**4. Conclusion**

In this guide, we have seen how the JVM and the Java compiler optimize memory allocations for String objects via the Java String Pool. We learned why the string is immutable or final in java.