I. **NAME**: IMRAN

II. **Empid:**1005

III. **Dept**: Micro-Services

Today Topics I'm going to Discuss are:-

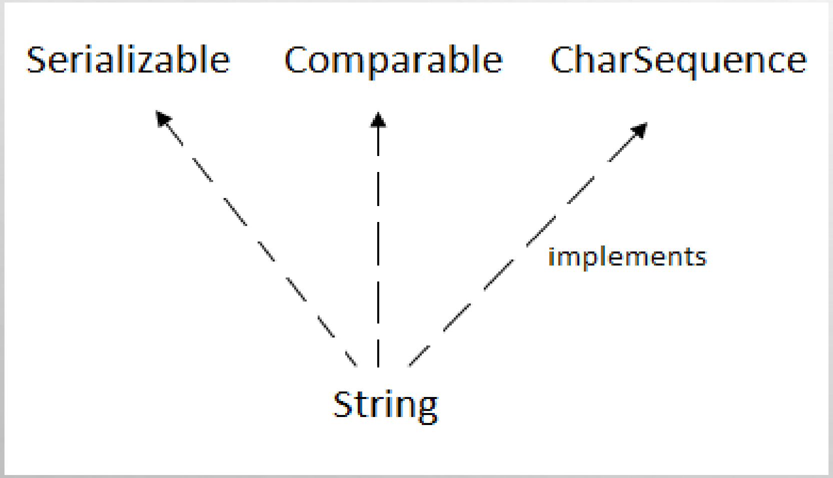
Strings, diff b/w == & equals() ,contract b/w hashcode and equals

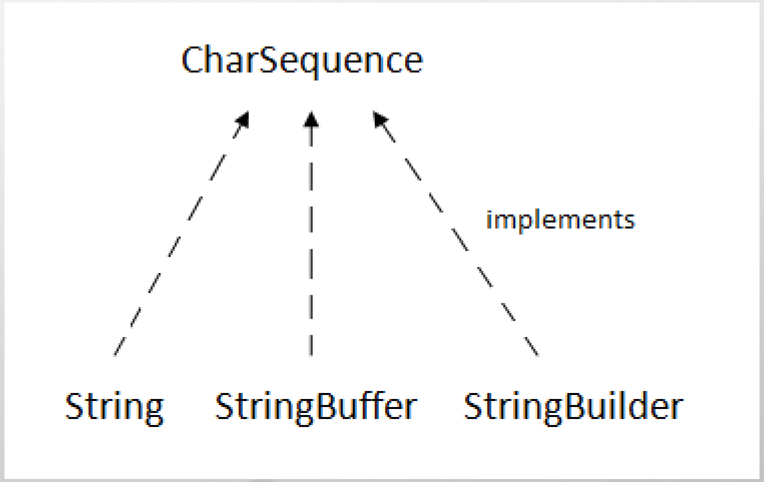
Java String: String is basically an object that represents sequence of char values.
String k=“eidiko”;

An array of characters works same as java String.For an example:

```
char  [] is={‘e’,’I’,’d’,’I’,’k’,’o’};

String s = new String(is);
```





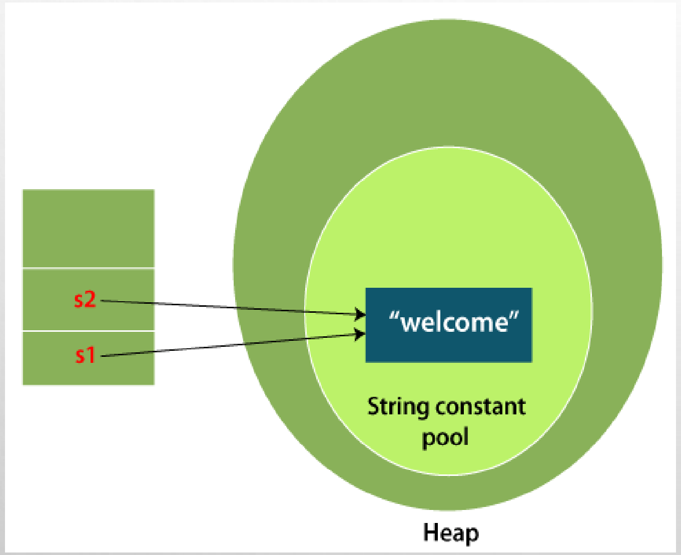
String is an object that represents a sequence of characters.

The java.lang.String class is used to create a string object.

We Create String Obj. in two ways

1.Using **String** literal
2.Using **new** keyword--→ String s=new String("welcome to eidiko"); by this two obj. will be created 1st in heap area 2nd in String Constant pool area. 2nd is refered by jvm by creating a internal ref.variable.

String s1="ismart";
String s2="ismart";



Note:By using String literal no new object will be created, if the value is present in string constant pool area.
By this we can say memory savage is happening .

Why String is Immutable or final..?
What is immutable..?
What is immutable String..?

Immutable:- immutable means that cannot be modified once it is created.

Immutable String :- immutable String means that cannot be modified once it is created. But we can change the ref.variable to the obj.

Why String Immutable or final:-**Immutable** is cause of security , synchronization and concurrency , caching and class loading.

The reason of making string **final** is not to allow others to extends it.

Reasons for making String immutable in Java:

- The String pool cannot be possible if String is not immutable in Java. A lot of heap space is saved by JRE. The same string variable can be referred to by more than one string variable in the pool.
- The String is safe for multithreading because of its immutableness. Different threads can access a single “String instance”. It removes the synchronization for thread safety because we make strings thread-safe implicitly.

```
class Sim
{
psvm(s[]a)
        {
        String s= “eidiko”;
        s1.concat(“rules”);
        Sopl(s);==➔eidiko
        }
}
```

- eidiko is referred by s

- Next VM creates another new String “eidiko rules”, but nothing refers to it, so the 2nd String is instantly lost. So we’re not able to reach it.
- Where does it goes…?

- All those exist in special area of memory called "**String constant pool**

- When compiler sees a String literal, it looks for String in the pool. If a match is found, the reference to the new literal is directed to existing String and no new String obj. is created.

- **Immutable:** In the SCPool , a String obj. is likely to have one or many ref. If several ref. point to same String without even knowing it, it would be bad if one of the ref. modified that String value. That's y String obj. are immutable.

- Now u could say, what if someone overrides the functionality of the String class? That's the reason that the **String Class is marked final** so that nobody can override

== is an Relational type operator used to check the relationship between two operands. Return type is Boolean

```
Int i=55,k=88;
Int i=55,s=55;
System.out.println(i== k);===➔false
 System.out.println(i ==s ); =➔ true
```

.equal() is a method that compares the actual content of the Strings. Return type is Boolean

```
  public class EqualsExmpl
{
Psvm(s[]args)
        {
String i1="eidiko";
String k1="EIDIKO";
String s1="eidiko";
Sopln(i1.equals(k1));==➔false
Sopln(i1.equals(s1));==➔true
        }
}
```

equals() and **hashCode() provided by the lang.object class for comparing obj.**
As we know that **Object** class is the parent class of every class, the default implementation of these two methods is already present in each class.
However, we can override these methods based on the requirement.

Equals() is a method used to compare values

<u>General Contract of equals():</u>

Reflexive: an object x must be equal to itself, which means for object,, equals(x) should return true

Symmetric: for two given objects x and y, x.equals(y) must return true if and only if equals(x) returns true.

Transitive: for any objects x, y and z, if x.equals(y) returns true and y.equals(z) returns true, then x.equals(z) should return true

Consistent: for any objects x and y, the value of x.equals(y) should change, only if the property in equals() changes.

For any object x, the *equals(null)* must return false.

hashCode() is an integer value associated with every object in java**..**

To get hashCode value we need to use hashCode() that returns integer hashCode value

The hashCode() method returns the same hash value when called on two objects, which are equal according to the equals() method. And if the objects are unequal, it usually returns different hash values.

Contract for hashCode()

If two objects are the same as per the equals(Object) method, then if we call the hashCode() method on each of the two objects, it must provide the same integer result .

For ex.

```
String a =“EIDIKO”;
String b=“EIDIKO”;
String c=“eidiko”;

If(a.equals(b))
{
Sopln(a.hashCode()+” & ”+b.hashCode());==➔  2044926797 & 2044926797
Sopln(a.hashCode()+” & ”+c.hashCode());==➔ 2044926797 & -1303369907
}
```