Arraylist, Vectors and Strings

ArrayList

- In Java, we use the ArrayList class to implement the functionality of resizable-arrays. It implements the List interface of the collections framework

- It allows us to create resizable arrays. Unlike arrays, arraylists can automatically adjust their capacity when we add or remove elements from them. Hence, arraylists are also known as dynamic arrays.

- Import the java.util.ArrayList package
- To create arraylists in Java
- ArrayList<Type> arrayList= new ArrayList<>();
- Type indicates the type of an arraylist. For example, Integer or String.

Basic Operations on ArrayList

- Add elements: To add a single element to the arraylist, we use the add() method of the ArrayList class.
- Access elements: To access an element from the arraylist, we use the get() method of the ArrayList class.
- Change elements: To change elements of the arraylist, we use the set() method of the ArrayList class.
- Remove elements: To remove an element from the arraylist, we can use the remove() method of the ArrayList class.

Adding element to an arraylist

```
import java.util.ArrayList;

class Languages{
  public static void main(String[] args){
    // create ArrayList
    ArrayList<String> languages = new ArrayList<>();

    // add() method without the index parameter
    languages.add("Java");
    languages.add("C");
    languages.add("Python");
    System.out.println("ArrayList: " + languages);
  }
}
```

```
ArrayList: [Java, C, Python]
```

Access ArrayList elements

```
String s0 = languages.get(0);
String s1 = languages.get(1);
String s2 = languages.get(2);
System.out.println("Element at index 0: " + s0);
System.out.println("Element at index 1: " + s1);
System.out.println("Element at index 2: " + s2);
```

```
Element at index 0: Java
Element at index 1: C
Element at index 2: Python
```

Change ArrayList elements

```
languages.set(1, "C++");
languages.set(2, "JavaScript");
System.out.println("Modified ArrayList: " + languages);
```

```
ArrayList: [Java, C, Python]
Modified ArrayList: [Java, C++, JavaScript]
```

Remove ArrayList Elements

```
String str = languages.remove(2);
System.out.println("Updated ArrayList: " + languages);
System.out.println("Removed Element: " + str);
```

```
ArrayList: [Java, C, Python]
Updated ArrayList: [Java, C]
Removed Element: Python
```

Vectors

- The Vector class implements a growable array of objects.
- Like an array, it contains components that can be accessed using an integer index.
- However, the size of a Vector can grow or shrink as needed to accommodate adding and removing items after the Vector has been created.
- The Vector class is an implementation of the List interface that allows us to create resizable-arrays similar to the ArrayList class.

- Syntax to create vectors in Java:

  Vector<Type> vector = new Vector<>();

  Type indicates the type of a linked list. For example: Integer , String.
- ArrayList is not synchronized whereas Vector is synchronized. So, ArrayList is faster than Vector.
- ArrayList prefers the iterator interface to traverse the components whereas Vector prefers Enumeration or Iterator interface to traverse the elements.

- Some common methods when working with vectors:

  - Add Elements to Vector.

  - add(element) - adds an element to vectors
  - add(index, element) - adds an element to the specified position
  - addAll(vector) - adds all elements of a vector to another vector

```java
import java.util.Vector;

class V1 {
    public static void main(String[] args) {
        Vector<String> birds = new Vector<>();

        // Using the add() method
        birds.add("Sparrow");
        birds.add("Eagle");

        // Using index number
        birds.add(2, "Owl");
        System.out.println("Vector: " + birds);

        // Using addAll()
        Vector<String> animals = new Vector<>();
        animals.add("Lion");

        animals.addAll(birds);
        System.out.println("New Vector: " + animals);
    }
}
```

```
Vector: [Sparrow, Eagle, Owl]
New Vector: [Lion, Sparrow, Eagle, Owl]
```

Access vector Elements

- get(index) - returns an element specified by the index
- iterator() - returns an iterator object to sequentially access vector elements

```
import java.util.Iterator;
import java.util.Vector;

class V2{
    public static void main(String[] args) {
        Vector<String> animals = new Vector<>();
        animals.add("Dog");
        animals.add("Horse");
        animals.add("Cat");

        // Using get()
        String element = animals.get(2);
        System.out.println("Element at index 2: " + element);

        // Using iterator()
        Iterator<String> iterate = animals.iterator();
        System.out.print("Vector: ");
        while(iterate.hasNext()) {
            System.out.print(iterate.next());
            System.out.print(", ");
        }
    }
}
```

```
Element at index 2: Cat
Vector: Dog, Horse, Cat,
```

Remove Vector Elements

- remove(index) - removes an element from specified position
- removeAll() - removes all the elements
- clear() - removes all elements. It is more efficient than removeAll()

```
import java.util.Vector;

class V3{
    public static void main(String[] args) {
        Vector<String> fruits = new Vector<>();
        fruits.add("Apple");
        fruits.add("Banana");
        fruits.add("Cherry");

        System.out.println("Initial Vector: " + fruits);

        // Using remove()
        String element = fruits.remove(1);
        System.out.println("Removed Element: " + element);  // Banana
        System.out.println("New Vector: " + fruits);        // [Apple, Cherry]

        // Using clear()
        fruits.clear();
        System.out.println("Vector after clear(): " + fruits);  // []
    }
}
```

```
Initial Vector: [Apple, Banana, Cherry]
Removed Element: Banana
New Vector: [Apple, Cherry]
Vector after clear(): []
```

Vector capacity() Method in Java

The **Java.util.Vector.capacity()** method in Java is used to get the capacity of the Vector or the length of the array present in the Vector.

Syntax of Vector capacity() method

Vector.capacity()

```
// Java code to demonstrate
// capacity() Method
import java.util.*;

public class GFG
{
   public static void main(String args[])
   {
      // Creating an empty Vector
      Vector<String> v = new Vector<String>();

      // Add elements in Vector
      v.add("Welcome");
      v.add("To");
      v.add("GFG");

      // Displaying the Vector
      System.out.println("Elements of Vector : " + v);

      // Displaying the capacity of Vector
      System.out.println("The capacity is : " + v.capacity());
   }
}
```

String and String Functions in Java

Java String

- In Java, string is basically **an object** that represents sequence of char values. An array of characters works same as Java string.

For example:

```
char[] ch={'j','a','v','a'};
String s=new String(ch);
```

is same as:

```
String s="java";
```

- The Java String is immutable which means it cannot be changed.
- Whenever we change any string, a new instance is created.
- For mutable strings, you can use StringBuffer class.

How to create a string object?

- There are two ways to create String object:
  1. By string literal
  2. By new keyword

1) String Literal

- Java String literal is created by using double quotes.
- For Example:

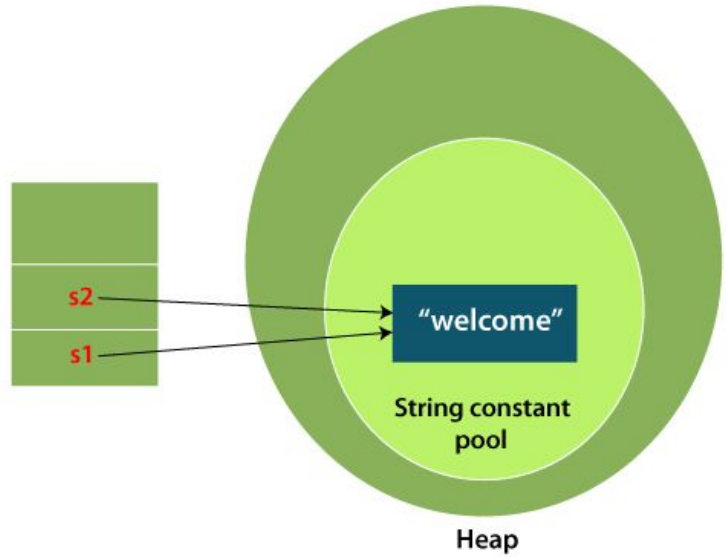
  String s="welcome";

Each time you create a string literal, the JVM checks the "string constant pool" first. If the string already exists in the pool, **a reference to the pooled instance is returned**. If the string doesn't exist in the pool, a new string instance is created and placed in the pool.

For example:

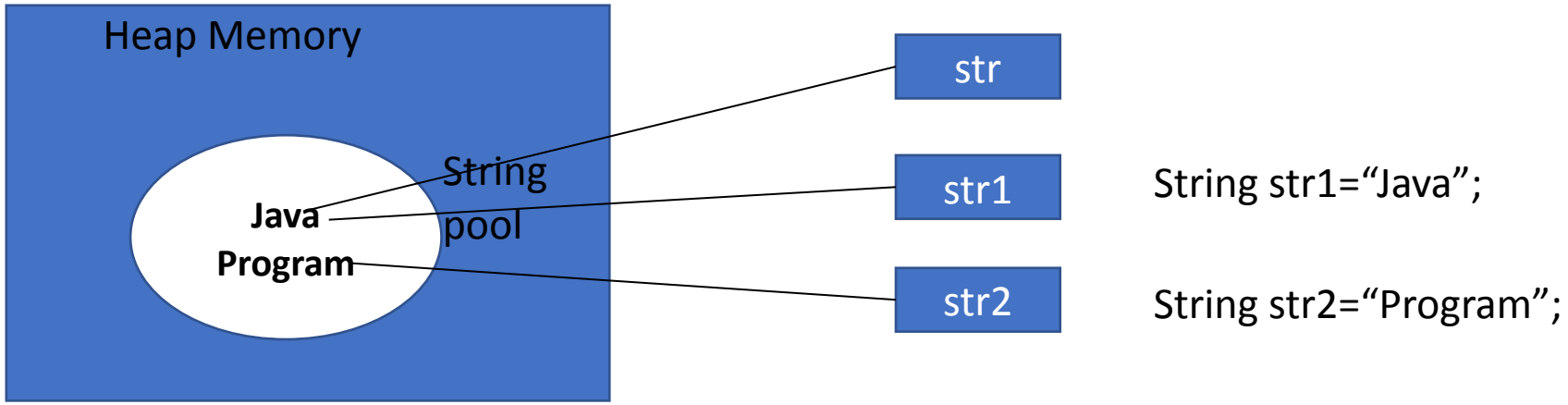
String s1="Welcome";
String s2="Welcome";//It doesn't create a new instance

String objects are stored in a special memory area known as the "string constant pool".



Creating String

- String str="Java"; // Java is Literal & str is reference variable



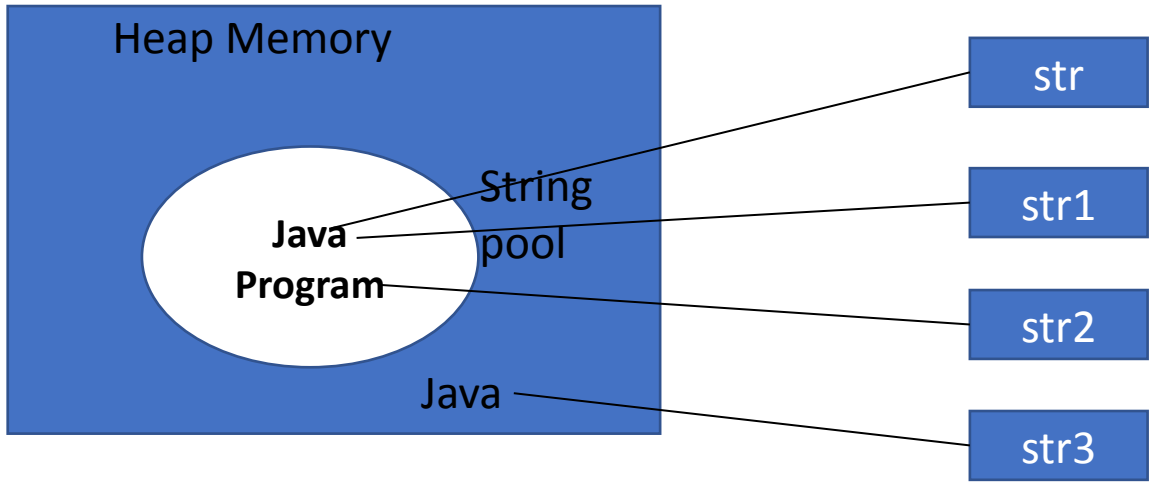
2) By new keyword

String s=**new** String("Welcome”);

- In such case, **JVM** will create a new string object in normal (non-pool) heap memory, and the literal "Welcome" will be placed in the string constant pool.
- The variable s will refer to the object in a heap (non-pool).

Using new keyword

- String object created with “new Keyword it always create a new object in heap memory
- String str3=new String(“Java”);



Java String class methods

The java.lang.String class provides many useful methods to perform operations on sequence of char values.

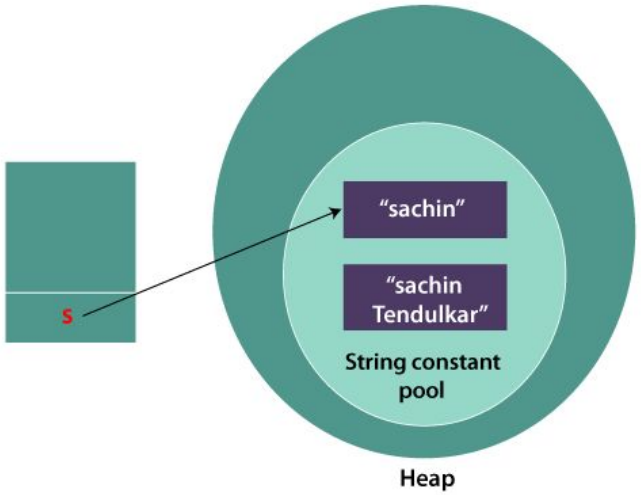
Method call	Meaning
S2=s1.toLowerCase()	Convert string s1 to lowercase
S2=s1.toUpperCase()	Convert string s1 to uppercase
S2=s1.repalce('x', 'y')	Replace occurrence x with y
S2=s1.trim()	Remove whitespaces at the beginning and end of the string s1
S1.equals(s2)	If s1 equals to s2 return true
S1.equalsIgnoreCase(s2)	If s1==s2 then return true with irrespective of case of charecters
S1.length()	Give length of s1
S1.CharAt(n)	Give nth character of s1 string
S1.compareTo(s2)	If s1<s2 –ve no If s1>s2 +ve no If s1==s2 then 0
S1.concat(s2)	Concatenate s1 and s2
S1.substring(n)	Give substring staring from nth character

- **String objects are immutable**.
  Immutable simply means unmodifiable or unchangeable.
  Once String object is created its data or state can't be changed but a new String object is created.

E.g.

```
class Testimmutablestring{
 public static void main(String args[]){
   String s="Sachin";
   s.concat(" Tendulkar");//concat() method appends the string at the end
   System.out.println(s);//will print Sachin because strings are immutable
 }
}
```

O/P: Sachin



Java String compare

- Strings in Java can be compared on the basis of content and reference.
- It is used in **authentication** (by equals() method), **sorting** (by compareTo() method), **reference matching** (by == operator) etc.

- There are three ways to compare String in Java:

1. By Using == Operator
2. By Using equals() Method
3. By compareTo() Method

By Using == Operator

```
String str1="Hello";
String str2=“Hello";
if(str1==str2)
   System.out.println("str1==str2");
else
   System.out.println("str1!=str2");
String str3= new String("Hello");
String str4=new String("Hello");
if(str3==str4)
   System.out.println("str3==str4");
else
   System.out.println("str3!=str4");
```

By Using equals() Method

```
if(str3.equals(str4)) // compares the contents of the strings
    System.out.println("str3 is equal to str4");
else
    System.out.println("str3 is not equal to str4");

String str5=“Welcome”;
If (str3.compareTo(str5)=0)
```

CompareTo()

```
public class CompareToExample{
public static void main(String args[]){
String s1="hello";
String s2="hello";
String s3="meklo";
String s4="hemlo";
String s5="flag";
System.out.println(s1.compareTo(s2));//0 because both are equal
System.out.println(s1.compareTo(s3));//-5 because "h" is 5 times lower than "
m"
System.out.println(s1.compareTo(s4));//-1 because "l" is 1 times lower than "m
"
System.out.println(s1.compareTo(s5));//2 because "h" is 2 times greater than
"f"
}}
```

Length() function

```
String str="John,Jennie,Jim,Jack,Joe";
int len=str.length();
System.out.println(" string length is:" +len);
```

toUpperCase() function

```
String str="John,Jennie,Jim,Jack,Joe";
str.toUpperCase();
System.out.println("uppercase string is: "+ str);
String s1=str.toUpperCase();
System.out.println("uppercase string is: "+ s1);
```

contains() function

```
String str="John,Jennie,Jim,Jack,Joe";
if(str.contains("Jim"))
    System.out.println("Jim is in the string");
String s2=str.substring(5);
System.out.println("Substring is: "+ s2);
String s3=str.substring(6,10);
System.out.println("Substring is: "+ s3);
```

replace() function

```
String str="John,Jennie,Jim,Jack,Joe";
String s4=str.replace('J','k');
System.out.println("new string is: "+ s4);
```

isEmpty(),trim() and concat()

```
String email="abc@example.com";
      if(!email.isEmpty())
      System.out.println("email is not empty:");
      String s5="    welcome    ";
      System.out.println(s5);
      String s6=s5.trim();
      System.out.println(s6);
            s6.concat(email);
         System.out.println(s6);
        String s7=s6.concat(email);
         System.out.println(s7);
         s6=s6.concat(email);
         System.out.println(s6);
```

StringBuffer

- **StringBuffer** is a peer class of **String** that provides much of the functionality of strings. String represents fixed-length, immutable character sequences while StringBuffer represents growable and writable character sequences.
- **StringBuffer** may have characters and substrings inserted in the middle or appended to the end. It will automatically grow to make room for such additions and often has more characters preallocated than are actually needed, to allow room for growth.
- **StringBuffer Constructors**
- **StringBuffer( ):** It reserves room for 16 characters without reallocation.

StringBuffer

```
StringBuffer s=new StringBuffer();
```

StringBuffer(String str): It accepts a **String** argument that sets the initial contents of the StringBuffer object and reserves room for 16 more characters without reallocation.

```
StringBuffer s=new StringBuffer(“JavaProgramming");
```

StringBuffer class

```
import java.io. * ;
class StrBuff {
  public static void main(String[] args) {
    StringBuffer str = new StringBuffer("JavaProgramming");
    int len = str.length();
    int cap= str.capacity();
    System.out.println("Length : " + len);
    System.out.println(“Capacity: “ +cap);
  }
}
```

append()

```
    str.append("Notepad++");
    System.out.println(str);
```

insert()

```
str.insert(15, " in ");
   System.out.println(str);
  }
```

reverse()

```
str.reverse();
System.out.println(str);
```

delete()

```
str.delete(0,4);
System.out.println(str);
```

deleteCharAt(n);

```
str.deleteCharAt(6);
System.out.println(str);
```

```
replace(start_index, end_index, string);
s.replace(10, 17, "Tutorial");
System.out.println(str);
```