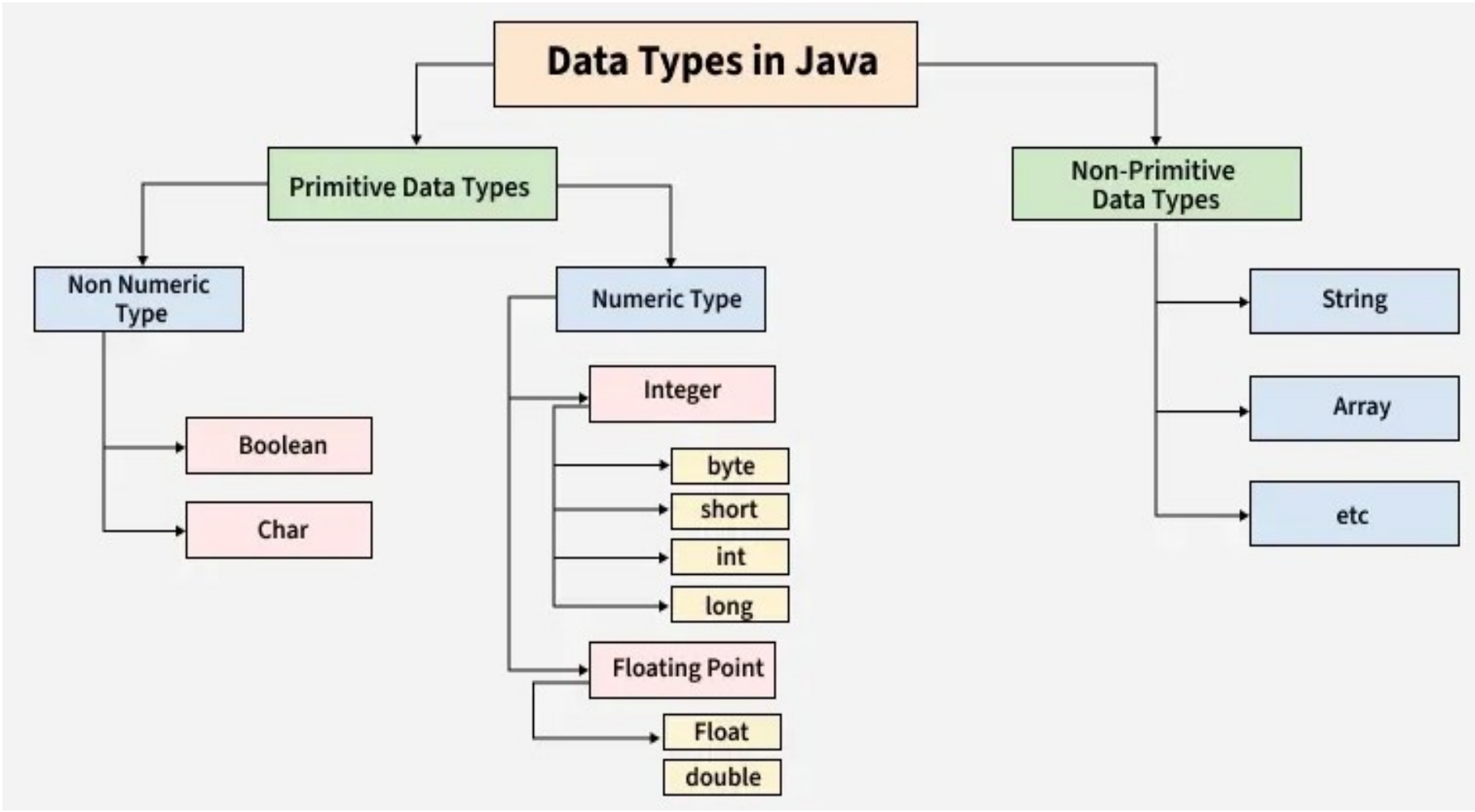
Introduction to Java

Learning Outcomes

- Understand basic terminologies and concepts of Java
  - Primitive Data types and Wrapper Types
  - Casting and Autoboxing
  - Arrays, Strings, Vector class

Data Types in Java



Primitive Data types

- **Primitive data types** are the building blocks of data manipulation.
- These are the **basic data** types.
- **Eight** primitive data types **– boolean** data type, **char** data type, **byte** data type, **short** data type, **int** data type, **long** data type, **float** data type, **double** data type

Primitive Data types

- **boolean Data Type**
  - **boolean** data type represents a **single bit** of information with **two** possible states: **true** or **false**.
  - **Size** is typically **1 byte (8 bits)**
  - Used to **store** the **result of logical expressions or conditions**.
  - **boolean** a = **false**;
  - **boolean** b = **true**;

Primitive Data types

- **byte Data Type**
  - Represents a **8-bits signed two's complement integer**.
  - It has a range of values from **-128** to **127**.
  - Its default value is **0**.
  - **Size** is **1 byte (8 bits)**
  - Used when working with **raw binary data** or when **memory conservation** is a concern, as it occupies **less memory**
  - **byte** a = 10;
  - **byte** b = -20;

Primitive Data types

- **short Data Type**
  - Represents a **16-bits signed two-complement integer**.
  - Its range of values is **-32,768** to **32,767**.
  - Its default value is **0**.
  - **Size** is **2 bytes (16 bits)**
  - Used when **memory conservation is a concern**, but **more precision than byte** is required
  - **short** a = 10000;
  - **short** b = -5000;

Primitive Data types

- **int Data Type**
  - Represents a **32-bits signed two's complement integer**.
  - It has a range of values from **-2,147,483,648** to **2,147,483,647**.
  - Its default value is **0**.
  - **Size** is **4 bytes (32 bits)**
  - Used to store **whole numbers** without decimal points
  - **int** myInt = 54;
  - In **Java SE 8** and later versions, we can use the int data type to represent an **unsigned 32-bit integer**. It has a value in the range [0, 2^{32}-1].

Primitive Data types

- **long Data Type**
  - Represents a **64-bits signed two's complement integer**.
  - It has a **wider range of values** than int, ranging from **-9,223,372,036,854,775,808** to **9,223,372,036,854,775,807**.
  - Its default value is **0.0L** or **0.0l**.
  - **Size** is **8 bytes (64 bits)**
  - Used **when a larger range of integer values** is needed.
  - **long** a = 5000000L; **long** b = -6000000L;
  - In **Java SE 8** and later versions, we can use the long data type to represent an **unsigned 64-bit long number**. It has a value in the range [0, 2^{64}-1].

Primitive Data types

- **float Data Type**
  - Represents **single-precision 32-bits IEEE 754 floating-point numbers**.
  - Its default value is **0.0f** or **0.0F**.
  - **Size** is **4 bytes (32 bits)**
  - Useful for applications where a **higher range of values is needed and precision is not critical**.
  - **float** f = 234.5f;

Primitive Data types

- **double Data Type**
  - Represents **double-precision 64-bits IEEE 754 floating-point numbers**.
  - Its default value is **0.0d**.
  - **Size** is **8 bytes (64 bits)**
  - It provides a **wider range of values and greater precision**
  - Suitable for applications that require high precision, such as financial calculations, scientific computations, and graphics programming
  - **double** num = 75.658d;

Primitive Data types

- **char Data Type**
  - Represents a **single 16-bits Unicode character**.
  - It can store **any character from the Unicode character set**
  - **Size is 2 bytes (16 bits)**
  - Used to represent **characters**, such as letters, digits, and symbols.
  - It can also be used to perform **arithmetic operations**, as the Unicode values of characters can be treated as integers.
  - **char** a = 65, b = 66, c = 67; **char** d = 'A';

Non-Primitive Data types

- Are also known as **reference data types**.
- It is used to **store complex objects** rather than simple values.
- **Reference data types** store references or memory addresses that point to the location of the object in memory.
- Eg: **String**, **Array**, **Class**, **Interface**, etc.

Exercise

- **Java uses the Unicode system, not the ASCII code system**
- **Ex-1:** Explore Unicode system
- **Ex-2:** List out basic differences between Unicode system and ASCII code system

Casting in Java

- **Type casting** is a **technique** that is used either by the **compiler** or a **programmer** to **convert one data type to another in Java**.
- **Type casting** is also known as **type conversion**.
- For example, **converting int to double**, **double to int**, **short to int**, etc.
- There are **two types of type casting** allowed in Java programming:
  - **Widening type casting**
  - **Narrowing type casting**

Casting in Java

- **Widening Type Casting**
  - **Widening type casting** is also known as i**mplicit type casting** in which a **smaller type is converted into a larger type**, it is done by the **compiler automatically**.
  - The **hierarchy of widening type casting** in Java:

    byte>short>char>int>long>float>double
  - **int num1 = 5004;**
  - **double num2 = 2.5;**
  - **double sum = num1 + num2;**

Casting in Java

- **Narrowing Type Casting**
  - **Narrowing type casting** is also known as **explicit type casting** or **explicit type conversion** which is done by the **programmer manually**.
  - In the **narrowing type casting,** a **larger type can be converted into a smaller type**.

Casting in Java

```
public class Tester
{
  public static void main(String[] args)
 {
      int num = 5004;
      double doubleNum = (double) num;
      System.out.println("The value of " + num + " after converting
      to the double is " + doubleNum);
      int convertedInt = (int) doubleNum;
      System.out.println("The value of " + doubleNum + " after
      converting to the int again is " + convertedInt);
 }
}
```

Output

The value of 5004 after converting to the double is 5004.0
The value of 5004.0 after converting to the int again is 5004

Wrapper Classes in Java

- A **Wrapper class** in Java is **one whose object wraps** or **contains primitive data types**.
- When we create an **object** in a wrapper class, it contains a **field**, and in this field, we can **store primitive data types**.
- In other words, we can **wrap a primitive value into a wrapper class object**.

Wrapper Classes in Java

Primitive Data Type	Wrapper Class
char	Character
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double
boolean	Boolean

Wrapper Classes in Java

- **Need of Wrapper Classes**
  - They **convert primitive data types into objects**.
  - **Objects** are needed if we wish to **modify the arguments passed into a method** (because primitive types are passed by value).
  - The classes in **java.util package** handle **only objects** and hence wrapper classes help in this case.
  - Data structures in the **Collection framework**, such as **ArrayList** and **Vector**, **store only objects** (reference types) and not primitive types.
  - An object is needed to support **synchronization** in multithreading.

Wrapper Classes in Java

- **Need of Wrapper Classes**
  - When working with **Collection objects**, such as **ArrayList**, where **primitive types cannot be used** (the **list can only store objects**):
  - **ArrayList<int> myNumbers = new ArrayList<int>();**
  - // Invalid
  - **ArrayList<Integer> myNumbers = new ArrayList<Integer>();**
  - // Valid

Wrapper Classes in Java

```
public class Main {
  public static void main(String[] args)
{
    Integer myInt = 5;
    Double myDouble = 5.99;
    Character myChar = 'A';
    System.out.println(myInt);
    System.out.println(myDouble);
    System.out.println(myChar);
  }
}
```

```
public class Main {
  public static void main(String[] args)
{
    Integer myInt = 5;
    Double myDouble = 5.99;
    Character myChar = 'A';
System.out.println(myInt.intValue());
System.out.println(myDouble.doubleValue());
System.out.println(myChar.charValue());
  }
}
```

- Since we are now working with **objects**, we can **use certain methods** to **get information** about the specific object.
- For example, the **following methods** are used to **get the value** associated with the corresponding **wrapper object**:
  - ***intValue(), byteValue(), shortValue(), longValue(), floatValue(), doubleValue(), charValue(), booleanValue()***.

Autoboxing and Unboxing in Java

- **Autoboxing** refers to the conversion of a **primitive value** into an **object** of the corresponding wrapper class.
  - For example, **converting int to Integer class**
- **Unboxing** refers to converting an **object** of a wrapper type to its corresponding **primitive value**.
  - For example, **conversion of Integer to int.**

Autoboxing and Unboxing in Java

```
public class Testclass {
    public static void main(String[] args)
    {
        // Creating an Integer Object with custom value say it be 25, Autoboxing
        Integer i = 25;

        // Unboxing the Object
        int i1 = i;

        // Print statements
        System.out.println("Value of i:" + i);
        System.out.println("Value of i1: " + i1);

         char cc ='a';
        // Autoboxing of character
        Character gfg = cc;

        // Unboxing of Character
        char ch = gfg;

        // Print statements
        System.out.println("Value of cc: " + cc);
        System.out.println("Value of ch: " + ch);
        System.out.println(" Value of gfg: " + gfg);
    }
}
```

Autoboxing and Unboxing in Java

```
public class Testclass
{
    public static void main(String[] args)
    {
        Character ch = 'a'; //autoboxing

        // unboxing - Character object (ch) to primitive conversion
        char a = ch;

        ArrayList<Integer> arrayList = new ArrayList<Integer>();
        arrayList.add(24); //autoboxing

        // unboxing because get method returns an Integer object
        int num = arrayList.get(0);

        // printing the values from primitive data types
        System.out.println(num);
    }
}
```

Variables and Keywords

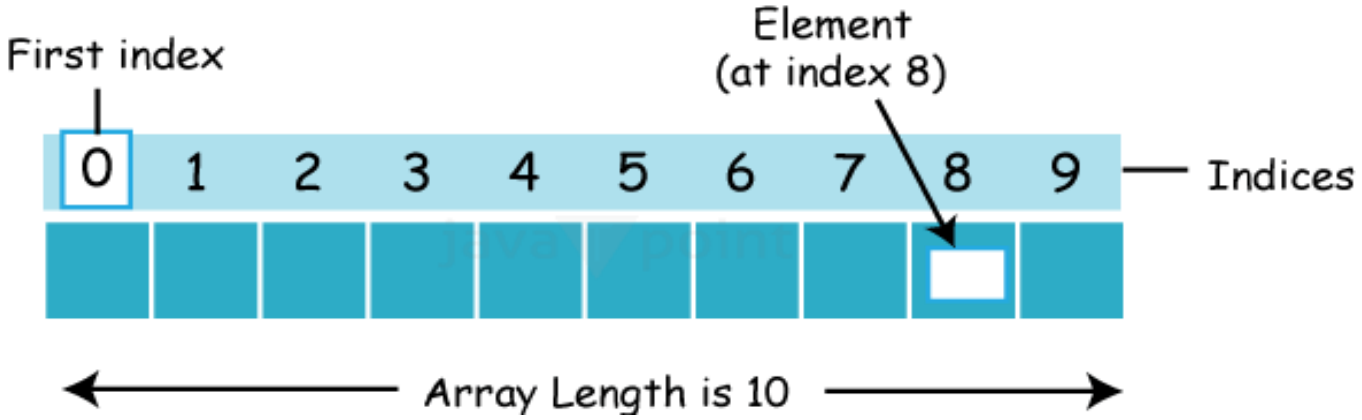
- **Variable** is a **name of the memory location**
  - Its **value** can be **changed**.
  - **int data=50;** //Here **data** is variable
  - There are three types of variables in Java: **local** variable, **instance** variable, **static** variable
- **Keywords** are also known as **reserved words**.
  - These are **predefined** words by Java so they **cannot** be **used** as a **variable** or **object name** or **class name**.
  - Eg: int, char, boolean; if, else, do, while, case, switch, for, public, final, etc.

Exercise

- There are **three types** of **variables** in Java: **local** variable, **instance** variable, **static** variable
- **Ex-1:** Explore all the three types of variables.
- **Ex-2:** List out different usecase scenarios of these variable
    - types
- **Ex-3:** Explore Variable Naming Convention in Java

Arrays

- In Java, **array** is an **object** which contains **elements of a similar data type**.
- The **elements** of an array are **stored in a contiguous memory location**.
- It is a **data structure** where we **store similar elements**.
- We can **store only a fixed set of elements** in a Java array.
- In Java, **an array can hold objects or primitive values**.



Arrays

- **Advantages:**
  - **Code Optimization:** It makes the code optimized, we can retrieve or sort the data efficiently.
  - **Random access:** We can get any data located at an **index** position.
- **Disadvantages:**
  - **Size Limit:** Arrays have a **fixed size** and **do not grow dynamically** at runtime.
- **Types** of Array in java:
  - **Single** Dimensional Array
  - **Multi-**Dimensional Array

Arrays – Single Dimensional Array

- A **single-dimensional array** in Java is a **linear collection of elements of the same data type**.
- It is **declared** and **instantiated** using the following syntax:
  - **dataType[] arr;** (or)
  - **dataType []arr;** (or)
  - **dataType arr[];**
- **Instantiation** of an Array in Java
  - **arr = new datatype[size]**;
- In **single line**,
  - **int a[]={33,3,4,5};** //declaration, instantiation and initialization

Arrays – Single Dimensional Array

```
public class Main{
 public static void main(String args[]){
    //declaration and instantiation of an array
    int a[]=new int[5];
    a[0]=10;//initialization
    a[1]=20;
    a[2]=70;
    a[3]=40;
    a[4]=50;
    //traversing array
    for(int i=0;i<a.length;i++){//length is the property of array
          System.out.println(a[i]);
    }
 }
}
```

Arrays - Multi-Dimensional Array

- A **multi-dimensional** array in Java is **an array of arrays** where **each element can be an array itself**.
- It is useful for **storing data in row and column format**.
- **Declaration:**
  - **dataType[][] arrayRefVar;** (or)
  - **dataType [][]arrayRefVar;** (or)
  - **dataType arrayRefVar[][];** (or)
  - **dataType []arrayRefVar[];**
- **Instantiation :**
  - **int[][] arr=new int[3][3];** //3 row and 3 column

Arrays - Multi-Dimensional Array

```
public class Main
{
    public static void main(String args[])
    {
            int arr[][] = {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}}; // 3x3 matrix

            // Printing the 2D array

            for (int i = 0; i < 3; i++)
              {
                for (int j = 0; j < 3; j++)
                 {
                    System.out.print(arr[i][j] + " ");
                 }
                System.out.println();
            }
     }
 }
```

Vector

- **Vector** is like the **dynamic array** which can **grow or shrink its size**.
- We can store n-number of elements in it as **there is no size limit**.
- It is a part of **Java Collection framework** since Java 1.2.
- It is found in the **java.util** package and implements the **List interface**, so we can use **all the methods of List interface** here.

Vector

```
import java.util.*;
public class Main
{
    public static void main(String args[]) {

        //Create a vector
        Vector<String> vec = new Vector<String>();

        //Adding elements using add() method of List
        vec.add("Tiger");
        vec.add("Lion");
        vec.add("Dog");
        vec.add("Elephant");

        //Adding elements using addElement() method of Vector
        vec.addElement("Rat");
        vec.addElement("Cat");
        vec.addElement("Deer");
        System.out.println("Elements are: "+vec);
    }
}
```

Elements are: [Tiger, Lion, Dog, Elephant, Rat, Cat, Deer]

String

- In Java, **string** is basically **an object** that represents **sequence of char values / charaters**.
- An **array of characters works** as a **string** in Java.
- For example:
  - **char[] ch={'o','o','p','c','l','a','s','s'};**
  - **String s=new String(ch);** is same as:
  - **String s="oopclass";**
- Package: **java.lang.String**

String

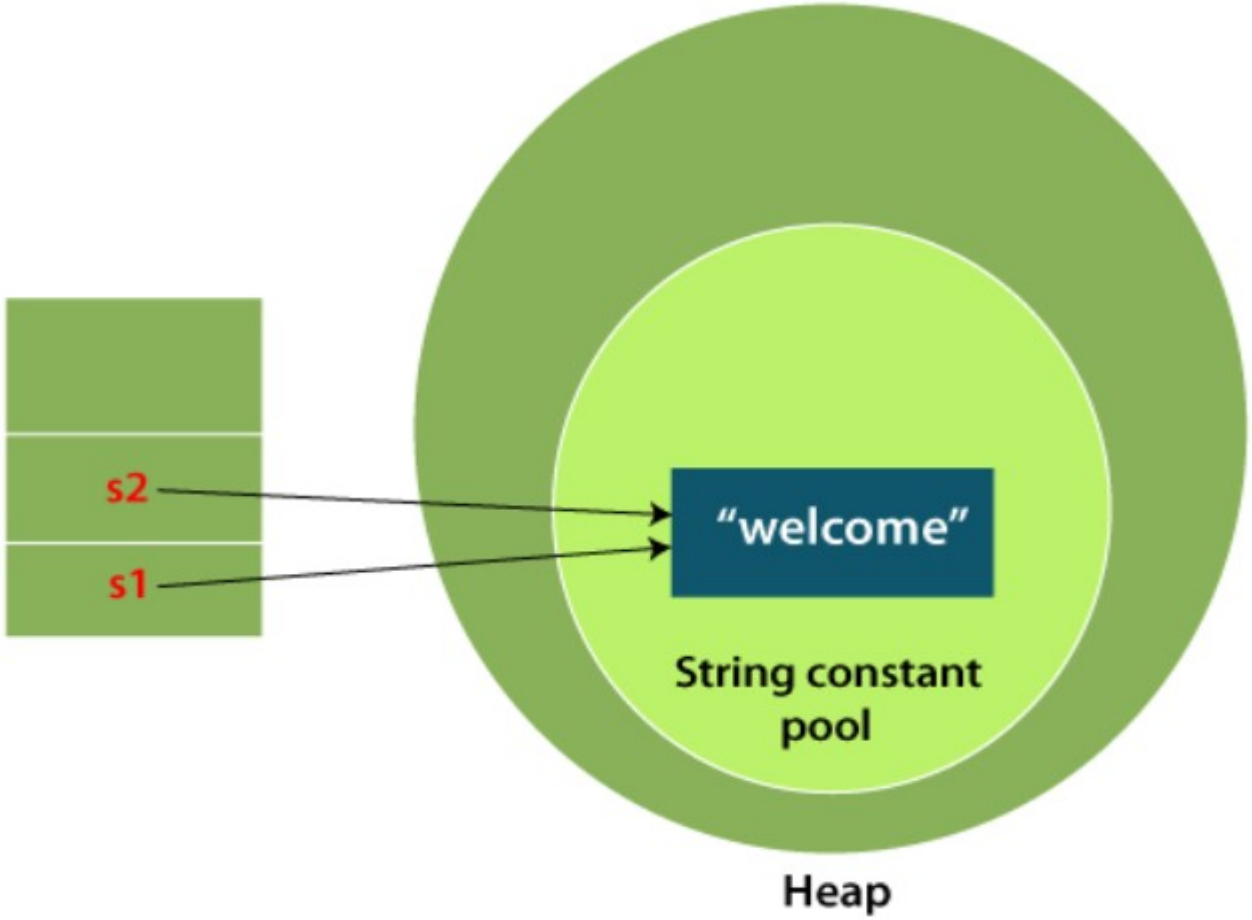
- Java **String** class provides a lot of **methods** to perform **operations on strings** such as ***compare(), concat(), equals(), split(), length(), replace(), compareTo(), intern(), substring()*** etc.
- The Java **String** is **immutable** which means **it cannot be changed**.
- Whenever **we change any string**, **a new instance** is created.
- For **mutable** strings, you can use **StringBuffer** and **StringBuilder** classes.
- There are **two ways to create String object**:
  - 1) By **string literal**, 2) By **new keyword**

String - String Literal

- Java **String literal** is created by using **double quotes**.
- For Example: **String s="welcome";**
- **String objects** are **stored** in a **special memory area** known as the "**string constant pool**".
- 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.

String - String Literal

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



String - String Literal

- In the above example, **only one object** will be created.
- **Firstly**, JVM will **not find any string object** with the value "Welcome" in string constant pool that is why it **will create a new object**.
- **After that** it **will find the string** with the value "Welcome" in the pool, it **will not create a new object** but **will return the reference to the same instance**.
- Why Java uses the concept of **String literal**?
  - To make Java **more memory efficient** (because no new objects are created if it exists already in the string constant pool).

String - String Literal

```
//Java Program to create string using string literal
public class Main
{
 public static void main(String[] args)
{
    String s1="Welcome";
    String s2="Welcome";  //It doesn't create a new instance

    System.out.println(s1+" "+s2);
 }
}
```

Output: Welcome Welcome

String – new Keyword

- **String s=new String("Welcome");**
- //creates **two objects** and **one reference** variable
- 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).

String – new Keyword

```
//Java Program to create string using new keyword
 public class Main
 {
    public static void main(String[] args)
     {
       String s1=new String("Welcome");
       String s2=new String("Welcome");

       System.out.println(s1+" "+s2);
    }
 }
```

Output: Welcome Welcome

String – new Keyword

```
public class StringMemoryAllotment {
    public static void main(String[] args) {
        // String literals - stored in the string pool
        String str1 = "Java";
        String str2 = "Java";

        // Checking if str1 and str2 point to the same object
        System.out.println("str1 == str2: " + (str1 == str2)); // true, because both refer to the same string literal in the pool

        // Strings created with 'new' - stored in heap memory outside the string pool
        String str3 = new String("Java");
        String str4 = new String("Java");

        // Checking if str3 and str4 point to the same object
        System.out.println("str3 == str4: " + (str3 == str4)); // false, because 'new' creates a new object each time
    }
}
```

Output:
str1 == str2: true
str3 == str4: false

How many objects are created here?

Exercise

- You have studied **Arrays**, **Vector**, and **String**
- **Ex-1:** Explore various methods used in the **Vector** class.
- **Ex-2:** Explore various methods used in the **String** class.
- **Ex-3:** Work-out various sample Java programs that involve **Array** / **Vector / String**.