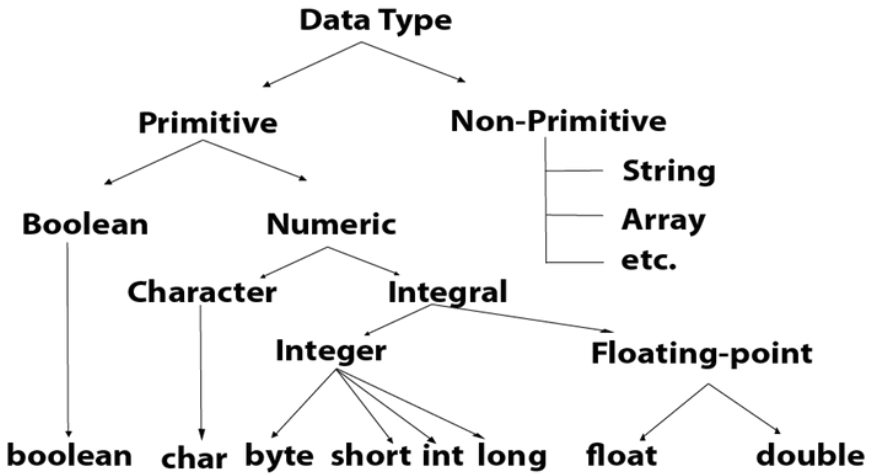
Non-Primitive Data Types or Reference Data Types I



Non-Primitive Data Types or Reference Data Types II

✓ The Reference Data Types will contain a memory address of variable value because the reference types won't store the variable value directly in memory.

✓ They are *arrays, strings, objects*.

Arrays in Java:

An array is a group of like-typed variables that are referred to by a common name. Arrays in Java work differently than they do in C/C++. Following are some important points about Java arrays.

- In Java all arrays are dynamically allocated.
- Since arrays are objects in Java, we can find their length using the object property length. This is different from C/C++ where we find length using sizeof.

Non-Primitive Data Types or Reference Data Types III

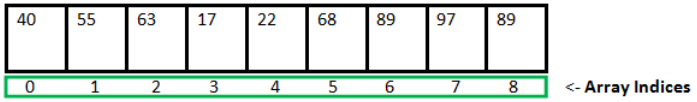
- A Java array variable can also be declared like other variables with [] after the data type.
- The variables in the array are ordered and each have an index beginning from 0.
- Java array can be also be used as a static field, a local variable or a method parameter.
- The size of an array must be specified by an int value and not long or short.
- The direct superclass of an array type is Object.
- Every array type implements the interfaces Cloneable and java.io.Serializable.

Non-Primitive Data Types or Reference Data Types IV

✓ Array can contain primitives (int, char, etc.) as well as object (or non-primitive) references of a class depending on the definition of the array.

✓ In case of primitive data types, the actual values are stored in contiguous memory locations.

✓ In case of objects of a class, the actual objects are stored in heap segment.



Array Length = 9
First Index = 0
Last Index = 8

Creating, Initializing, and Accessing an Array

One-Dimensional Arrays :

The general form of a one-dimensional array declaration is:

```
type var-name[];
OR
type[] var-name;
```

✓ An array declaration has two components: the type and the name. *type* declares the element type of the array.

✓ The element type determines the data type of each element that comprises the array.

✓ Like an array of integers, we can also create an array of other primitive data types like char, float, double, etc. or user-defined data types (objects of a class).

✓ Thus, the element type for the array determines what type of data the array will hold.

Non-Primitive Data Types or Reference Data Types VI

```
1  // both are valid declarations
2  int intArray[];
3  or int[] intArray;
4
5  byte byteArray[];
6  short shortsArray[];
7  boolean booleanArray[];
8  long longArray[];
9  float floatArray[];
10 double doubleArray[];
11 char charArray[];
12
13
14 Object[] ao,// array of Object
15 Collection[] ca; // array of Collection of unknown type
```

Non-Primitive Data Types or Reference Data Types VII

✓ Although the first declaration above establishes the fact that intArray is an array variable, no actual array exists.

✓ It merely tells the compiler that this variable (intArray) will hold an array of the integer type.

✓ To link intArray with an actual, physical array of integers, you must allocate one using new and assign it to intArray.

Instantiating an Array in Java:

✓ When an array is declared, only a reference of array is created.

✓ Obtaining an array is a two-step process. First, you must declare a variable of the desired array type. Second, you must allocate the memory that will hold the array, using new, and assign it to the array variable. Thus, in Java all arrays are **dynamically allocated**.

Non-Primitive Data Types or Reference Data Types VIII

✓ To actually create or give memory to array, you create an array like this:The general form of new as it applies to one-dimensional arrays appears as follows:

```
1 //Instantiating an Array in Java
2 var-name = new type [size];
3 //Here, type specifies the type of data being allocated, size specifies the number of elements in the array,
      and var-name is the name of array variable that is linked to the array. That is, to use new to allocate an array,
      you must specify the type and number of elements to allocate.
4 ---------------------------
5 int[] anArray;
6 anArray = new int[10];
7 OR
8
9 int[] intArray = new int[20]; // combining both statements in one
```

✓ The elements in the array allocated by new will automatically be initialized to $zero$ (for numeric types), $false$ (for boolean), or $null$ (for reference types).

Below are the default assigned values.

Non-Primitive Data Types or Reference Data Types IX

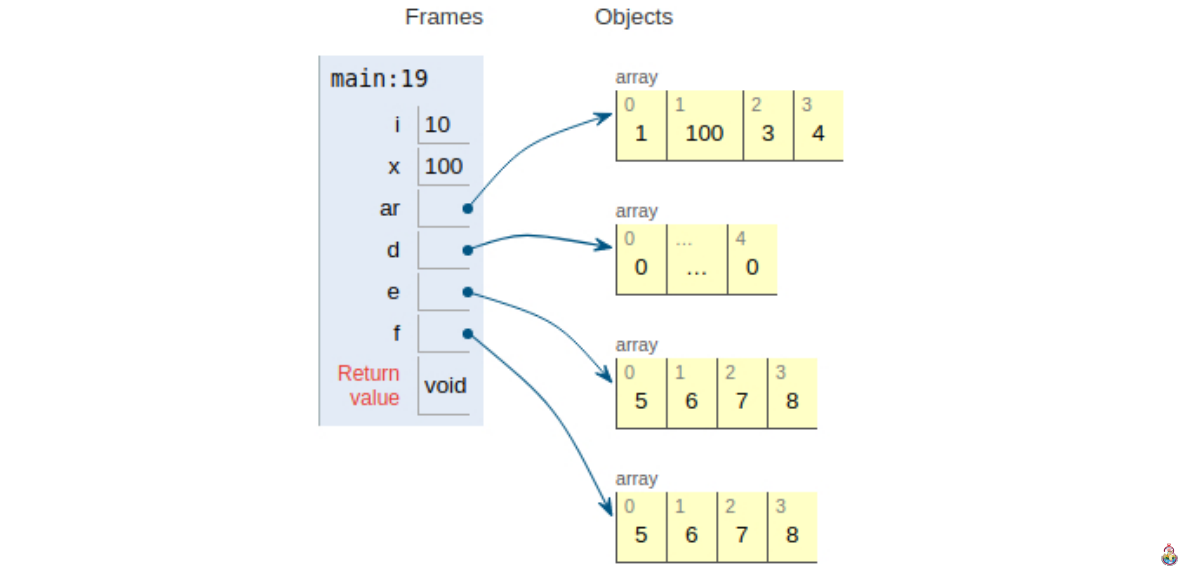
```
boolean : false
int : 0
double : 0.0
String : null
User Defined Type : null
```

```
import java.io.*;
public class GFG
{
      public static void main(String args[]) throws IOException
      {
            int i, x;
            i=10;
            x=i;
            x=100;
            int ar[] =
            {
                  1, 2, 3, 4
            }
            ;
            int[] d=ar;
            d[1]=100;
            d =new int[5];
```

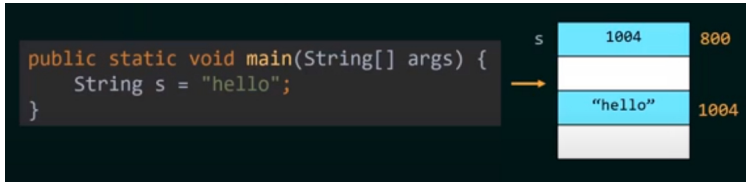
Non-Primitive Data Types or Reference Data Types X

```
18              int[] e=
19              {
20                    5,6,7,8
21              }
22              ;
23              int f[]=
24              {
25                    5,6,7,8
26              }
27              ; //even values are same but pointing different object
28
29        }
30  }
```

Non-Primitive Data Types or Reference Data Types XI



String I



✓ The String class represents character strings.

✓ All string literals in Java programs, such as ”abc”, are implemented as instances of this class.

✓ Strings in Java are Objects that are backed internally by a char array.

✓ Since arrays are immutable(cannot grow), Strings are immutable as well. Whenever a change to a String is made, an entirely new String is created.

✓ Strings are constant; their values cannot be changed after they are created.

String II

✓ String buffers support mutable strings. Because String objects are immutable they can be shared. For example:

```
String str = "abc";
is equivalent to:
char data[] =
{
      'a', 'b', 'c'
}
;

String str = new String(data);
```

String III

Here are some more examples of how strings can be used:

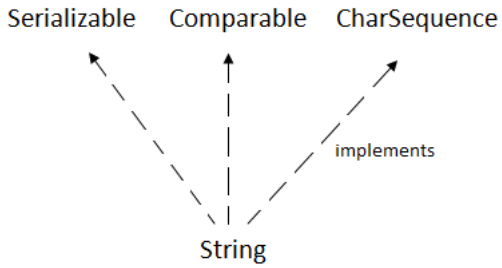
```
System.out.println("abc");
String cde = "cde";
System.out.println("abc" + cde);
String c = "abc".substring(2,3);
String d = cde.substring(1, 2);

char[] ch=
{
      'j','a','v','a','t','p','o','i','n','t'
}
;
String s=new String(ch);

or
String s="javatpoint";
```

String IV

✓ 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.lang.String class implements Serializable, Comparable and CharSequence interfaces.

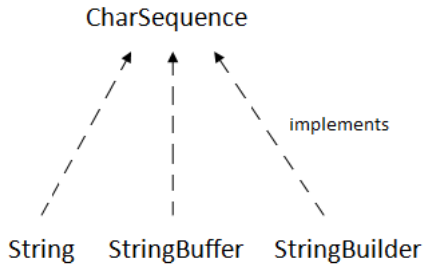
String V


Serializable
Comparable
CharSequence
implements
String

String VI

CharSequence Interface:

✓ The CharSequence interface is used to represent the sequence of characters. String, StringBuffer and StringBuilder classes implement it. It means, we can create strings in java by using these three classes.



String VII

✓ 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.

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:

```
1 String s="welcome";
```

String VIII

✓ Each time you create a string literal, the JVM checks the ”string constant pool” first. This allows JVM to optimize the initialization of String literal.

✓ 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.

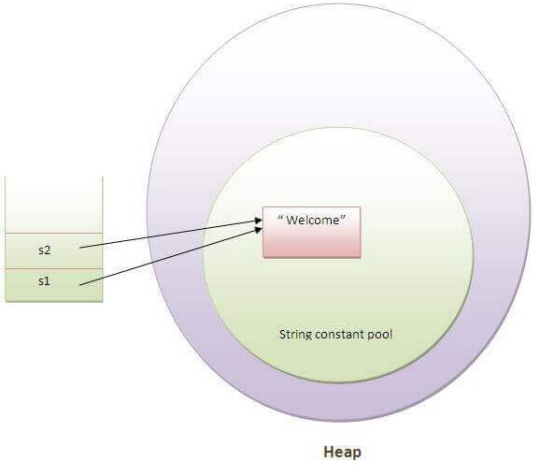
For example:

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

String IX

```
1  public class JavaSample
2  {
3        public static void main(String[] args)
4        {
5              String s1="Welcome";
6              String s2="Welcome"; //It doesn't create a new instance
7              System.out.println(Integer.toHexString(s1.hashCode()));
8              System.out.println(Integer.toHexString(s2.hashCode()));
9        }
10
11 }
```

String X



String XI

✓ 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.

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

✓ To make Java more memory efficient (because no new objects are created if it exists already in the string constant pool).

2. By new keyword:

```
1 String s=new String("Welcome"); //creates two objects and one reference variable
```

String XII

✓ 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).

Java String Example:

```
1  public class StringExample
2  {
3        public static void main(String args[])
4        {
5              String s1="java"; //creating string by java string literal
6              char ch[]=
7              {
8                    's','t','r','i','n','g','s'
9              }
10             ;
11             String s2=new String(ch); //converting char array to string
12             String s3=new String("example"); //creating java string by new keyword
13             System.out.println(s1);
14             System.out.println(s2);
15             System.out.println(s3);
16       }
17
18 }
```

String XIV

✓ The class String includes methods for examining individual characters of the sequence, for comparing strings, for searching strings, for extracting substrings, and for creating a copy of a string with all characters translated to uppercase or to lowercase. Case mapping is based on the Unicode Standard version specified by the Character class.

✓ The Java language provides special support for the string concatenation operator (+), and for conversion of other objects to strings.

✓ String concatenation is implemented through the StringBuilder(or StringBuffer) class and its append method.

✓ String conversions are implemented through the method toString, defined by Object and inherited by all classes in Java. For additional information on string concatenation and conversion, see Gosling, Joy, and Steele, The Java Language Specification.

String XV

✓ Unless otherwise noted, passing a null argument to a constructor or method in this class will cause a NullPointerException to be thrown.

✓ A String represents a string in the UTF-16 format in which supplementary characters are represented by surrogate pairs.

✓ Index values refer to char code units, so a supplementary character uses two positions in a String.

✓ The String class provides methods for dealing with Unicode code points (i.e., characters), in addition to those for dealing with Unicode code units (i.e., char values).

Java String compare:

✓ We can compare string in java 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 equals() method
2. By = = operator
3. By compareTo() method

1. String compare by equals() method
   The String equals() method compares the original content of the string. It compares values of string for equality. String class provides two methods:

String XVII

- public boolean equals(Object another) compares this string to the specified object.
- public boolean equalsIgnoreCase(String another) compares this String to another string, ignoring case.

```
1  public class Teststringcomparison1
2  {
3      public static void main(String args[])
4      {
5          String s1="Sachin";
6          String s2="Sachin";
7          String s3=new String("Sachin");
8          String s4="Saurav";
9          System.out.println(s1.equals(s2)); //true
10         System.out.println(s1.equals(s3)); //true
11         System.out.println(s1.equals(s4)); //false
12     }
13 }
```

String XVIII

```
1  class Teststringcomparison2
2  {
3        public static void main(String args[])
4        {
5              String s1="Sachin";
6              String s2="SACHIN";
7
8              System.out.println(s1.equals(s2)); //false
9              System.out.println(s1.equalsIgnoreCase(s2)); //true
10       }
11 }
```

2. String compare by == operator

   The = = operator compares references not values.

String XIX

```
1  class Teststringcomparison3
2  {
3        public static void main(String args[])
4        {
5              String s1="Sachin";
6              String s2="Sachin";
7              String s3=new String("Sachin");
8              System.out.println(s1==s2); //true (because both refer to same instance)
9              System.out.println(s1==s3); //false(because s3 refers to instance created in nonpool)
10       }
11 }
```

3. String compare by compareTo() method:
   The String compareTo() method compares values lexicographically and returns an integer value that describes if first string is less than, equal to or greater than second string.
   Suppose s1 and s2 are two string variables. If:

String XX

- s1 == s2 :0
- s1 ¿ s2 :positive value
- s1 ¡ s2 :negative value

```
1  class Teststringcomparison4
2  {
3        public static void main(String args[])
4        {
5              String s1="Sachin";
6              String s2="Sachin";
7              String s3="Ratan";
8              System.out.println(s1.compareTo(s2)); //0
9              System.out.println(s1.compareTo(s3)); //1(because s1>s3)
10             System.out.println(s3.compareTo(s1)); //-1(because s3 < s1 )
11       }
12 }
```

StringBuilder:

✓ The StringBuilder in Java represents a mutable sequence of characters.

✓ Since the String Class in Java creates an immutable sequence of characters, the StringBuilder class provides an alternate to String Class, as it creates a mutable sequence of characters.

Syntax:

```
1 StringBuilder str = new StringBuilder();
2 str.append("GFG");
```

String XXII

```
1  public class JavaSample
2  {
3        public static void main(String[] args)
4        {
5              StringBuilder str = new StringBuilder();
6              System.out.println(Integer.toHexString(str.hashCode()));
7              str.append("GFG");
8              System.out.println(str);
9              System.out.println(Integer.toHexString(str.hashCode()));
10             System.out.println(str);
11             str.append(" MCA");
12             System.out.println(Integer.toHexString(str.hashCode()));
13             System.out.println(str);
14       }
15 }
```

Immutable String in Java

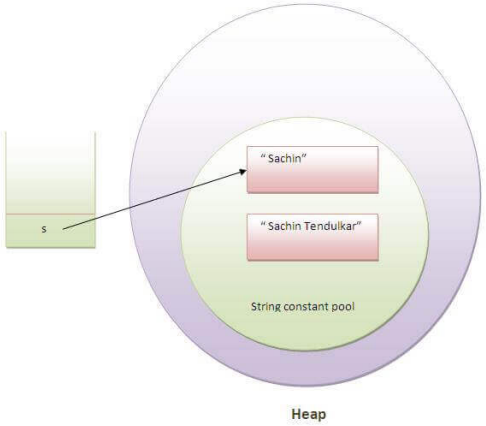
In java, string objects are immutable. Immutable simply means unmodifiable or unchangeable.

Once string object is created its data or state can't be changed but a new string object is created.

Let's try to understand the immutability concept by the example given below:

```
1  class Testimmutablestring
2  {
3        public static void main(String args[])
4        {
5              String s="Sachin";
6              s.concat(" Tendulkar"); //concat() method appends the string at the end
7              System.out.println(s); //will print Sachin because strings are immutable objects
8        }
9  }
```

String XXIV



String XXV

✓ But if we explicitely assign it to the reference variable, it will refer to ”Sachin Tendulkar” object.For example:

```
class Testimmutablestring1
{
      public static void main(String args[])
      {
            String s="Sachin";
            s=s.concat(" Tendulkar");
            System.out.println(s);
      }
}
```

✓ Because java uses the concept of string literal.Suppose there are 5 reference variables,all referes to one object ”sachin”.If one reference variable changes the value of the object, it will be affected to all the reference variables. That is why string objects are immutable in java.

String Concatenation in Java:

✓ In java, string concatenation forms a new string that is the combination of multiple strings. There are two ways to concat string in java:

1. By + (string concatenation) operator
2. By concat() method

1. By + (string concatenation) operator

```
1  class TestStringConcatenation1
2  {
3        public static void main(String args[])
4        {
5              String s="Sachin"+" Tendulkar";
6              System.out.println(s); //Sachin Tendulkar
7        }
8  }
```

String XXVII

String s=(new StringBuilder()).append(”Sachin”).append(” Tendulkar).toString();

```
1  class TestStringConcatenation2
2  {
3        public static void main(String args[])
4        {
5              String s=50+30+"Sachin"+40+40;
6              System.out.println(s); //80Sachin4040
7        }
8  }
```

❷ String Concatenation by concat() method:

String XXVIII

```
1  //public String concat(String another)
2  class TestStringConcatenation3
3  {
4        public static void main(String args[])
5        {
6              String s1="Sachin ";
7              String s2="Tendulkar";
8              String s3=s1.concat(s2);
9              System.out.println(s3); //Sachin Tendulkar
10       }
11 }
```

Java StringBuffer class:

✓ Java StringBuffer class is used to create mutable (modifiable) string. The StringBuffer class in java is same as String class except it is mutable i.e. it can be changed.

Important Constructors of StringBuffer class.

Constructor	Description
StringBuffer()	creates an empty string buffer with the initial capacity of 16.
StringBuffer(String str)	creates a string buffer with the specified string.
StringBuffer(int capacity)	creates an empty string buffer with the specified capacity as length.

Important methods of StringBuffer class:

StringBuffer append() method

✓ The append() method concatenates the given argument with this string.

```
class StringBufferExample
{
      public static void main(String args[])
      {
            StringBuffer sb=new StringBuffer("Hello ");
            sb.append("Java"); //now original string is changed
            System.out.println(sb); //prints Hello Java
      }
}
```

StringBuffer insert() method:

✓ The insert() method inserts the given string with this string at the given position.

```
1  class StringBufferExample2
2  {
3        public static void main(String args[])
4        {
5              StringBuffer sb=new StringBuffer("Hello ");
6              sb.insert(1,"Java"); //now original string is changed
7              System.out.println(sb); //prints HJavaello
8        }
9  }
```

StringBuffer replace() method

✓ The replace() method replaces the given string from the specified beginIndex and endIndex.

```
1  class StringBufferExample3
2  {
3        public static void main(String args[])
4        {
5              StringBuffer sb=new StringBuffer("Hello");
6              sb.replace(1,3,"Java");
7              System.out.println(sb); //prints HJavalo
8        }
9  }
```

StringBuffer delete() method

✓ The delete() method of StringBuffer class deletes the string from the specified beginIndex to endIndex.

```
class StringBufferExample4
{
      public static void main(String args[])
      {
            StringBuffer sb=new StringBuffer("Hello");
            sb.delete(1,3);
            System.out.println(sb); //prints Hlo
      }
}
```

StringBuffer reverse() method

✓ The reverse() method of StringBuilder class reverses the current string.

```
class StringBufferExample5
{
      public static void main(String args[])
      {
            StringBuffer sb=new StringBuffer("Hello");
            sb.reverse();
            System.out.println(sb); //prints olleH
      }
}
```

StringBuffer capacity() method

✓ The capacity() method of StringBuffer class returns the current capacity of the buffer. The default capacity of the buffer is 16. If the number of character increases from its current capacity, it increases the capacity by (oldcapacity*2)+2. For example if your current capacity is 16, it will be (16*2)+2=34.

```
1  class StringBufferExample6
2  {
3        public static void main(String args[])
4        {
5              StringBuffer sb=new StringBuffer();
6              System.out.println(sb.capacity()); //default 16
7              sb.append("Hello");
8              System.out.println(sb.capacity()); //now 16
9              sb.append("java is my favourite language");
10             System.out.println(sb.capacity()); //now (16*2)+2=34 i.e (oldcapacity*2)+2
11       }
12 }
```

StringBuffer ensureCapacity() method

✓ The ensureCapacity() method of StringBuffer class ensures that the given capacity is the minimum to the current capacity. If it is greater than the current capacity, it increases the capacity by (oldcapacity*2)+2. For example if your current capacity is 16, it will be (16*2)+2=34.

```
1  class StringBufferExample7
2  {
3        public static void main(String args[])
4        {
5              StringBuffer sb=new StringBuffer();
6              System.out.println(sb.capacity()); //default 16
7              sb.append("Hello");
8              System.out.println(sb.capacity()); //now 16
9              sb.append("java is my favourite language");
10             System.out.println(sb.capacity()); //now (16*2)+2=34 i.e (oldcapacity*2)+2
11             sb.ensureCapacity(10); //now no change
12             System.out.println(sb.capacity()); //now 34
13             sb.ensureCapacity(50); //now (34*2)+2
14             System.out.println(sb.capacity()); //now 70
```

```
15          }
16  }
```

Difference between String and StringBuffer:

No.	String	StringBuffer
1)	String class is immutable.	StringBuffer class is mutable.
2)	String is slow and consumes more memory when you concat too many strings because every time it creates new instance.	StringBuffer is fast and consumes less memory when you cancat strings.
3)	String class overrides the equals() method of Object class. So you can compare the contents of two strings by equals() method.	StringBuffer class doesn't override the equals() method of Object class.

String XXXVIII

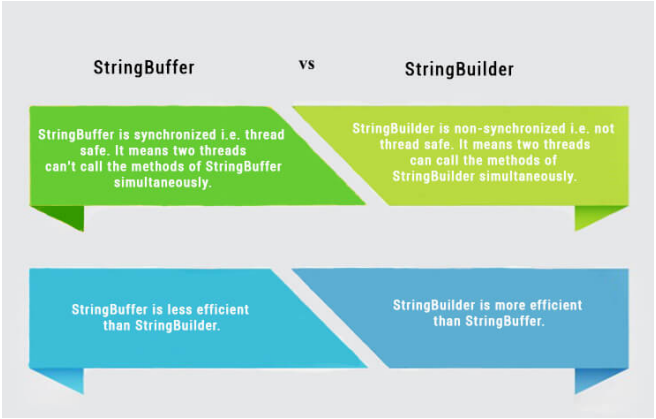
```
1  //Performance Test of String and StringBuffer
2  public class ConcatTest
3  {
4        public static String concatWithString()
5        {
6              String t = "Java";
7              for (int i=0; i<10000; i++)
8              {
9                    t = t + "Tpoint";
10             }
11             return t;
12       }
13       public static String concatWithStringBuffer()
14       {
15             StringBuffer sb = new StringBuffer("Java");
16             for (int i=0; i<10000; i++)
17             {
18                   sb.append("Tpoint");
19             }
20             return sb.toString();
21       }
22       public static void main(String[] args)
23       {
```

String XXXIX

```
24              long startTime = System.currentTimeMillis();
25              concatWithString();
26              System.out.println("Time taken by Concating with String: "+(System.currentTimeMillis()-startTime)+"ms");
27              startTime = System.currentTimeMillis();
28              concatWithStringBuffer();
29              System.out.println("Time taken by Concating with StringBuffer: "+(System.currentTimeMillis()-startTime)+"ms");
30      }
31  }
```

String XL

Difference between StringBuffer and StringBuilder:



Performance Test of StringBuffer and StringBuilder:

```
1  //Java Program to demonstrate the performance of StringBuffer and StringBuilder classes.
2  public class ConcatTest
3  {
4        public static void main(String[] args)
5        {
6              long startTime = System.currentTimeMillis();
7              StringBuffer sb = new StringBuffer("Java");
8              for (int i=0; i<10000; i++)
9              {
10                   sb.append("Tpoint");
11             }
12             System.out.println("Time taken by StringBuffer: " + (System.currentTimeMillis() - startTime) + "ms");
13             startTime = System.currentTimeMillis();
14             StringBuilder sb2 = new StringBuilder("Java");
15             for (int i=0; i<10000; i++)
16             {
17                   sb2.append("Tpoint");
18             }
19             System.out.println("Time taken by StringBuilder: " + (System.currentTimeMillis() - startTime) + "ms");
20       }
21 }
```

How to read primitive data types in Java I

Java Scanner Class

✓ Java Scanner class allows the user to take input from the console.

✓ It belongs to java.util package. It is used to read the input of primitive types like int, double, long, short, float, and byte. It is the easiest way to read input in Java program.

Syntax

```
1 Scanner sc=new Scanner(System.in);
```

✓ The above statement creates a constructor of the Scanner class having System.in as an argument. It means it is going to read from the standard input stream of the program. The java.util package should be import while using Scanner class.

✓ It also converts the Bytes (from the input stream) into characters using the platform's default charset.

Methods of Java Scanner Class:

How to read primitive data types in Java II

Method	Description
int nextInt()	It is used to scan the next token of the input as an integer.
float nextFloat()	It is used to scan the next token of the input as a float.
double nextDouble()	It is used to scan the next token of the input as a double.
byte nextByte()	It is used to scan the next token of the input as a byte.
String nextLine()	Advances this scanner past the current line.
boolean nextBoolean()	It is used to scan the next token of the input into a boolean value.
long nextLong()	It is used to scan the next token of the input as a long.
short nextShort()	It is used to scan the next token of the input as a Short.
BigInteger nextBigInteger()	It is used to scan the next token of the input as a BigInteger.
BigDecimal nextBigDecimal()	It is used to scan the next token of the input as a BigDecimal.

How to read primitive data types in Java III

Example of integer Input from user

```
1  import java.util.*;
2  class UserInputDemo
3  {
4        public static void main(String[] args)
5        {
6              Scanner sc= new Scanner(System.in); //System.in is a standard input stream
7              System.out.print("Enter first number- ");
8              int a= sc.nextInt();
9              System.out.print("Enter second number- ");
10             int b= sc.nextInt();
11             System.out.print("Enter third number- ");
12             int c= sc.nextInt();
13             int d=a+b+c;
14             System.out.println("Total= " +d);
15       }
16 }
```

✓ Example of String Input from user

How to read primitive data types in Java IV

```
1  import java.util.*;
2  class UserInputDemo1
3  {
4        public static void main(String[] args)
5        {
6              Scanner sc= new Scanner(System.in); //System.in is a standard input stream
7              System.out.print("Enter a string: ");
8              String str= sc.nextLine();        //reads string
9              System.out.print("You have entered: "+str);
10       }
11 }
```

✓ To read an element of an array uses these methods in a for loop:

How to read primitive data types in Java V

```
1  import java.util.Arrays;
2  import java.util.Scanner;
3
4  public class ReadingWithScanner
5  {
6        public static void main(String args[])
7        {
8              Scanner s = new Scanner(System.in);
9              System.out.println("Enter the length of the array:");
10             int length = s.nextInt();
11             int [] myArray = new int[length];
12             System.out.println("Enter the elements of the array:");
13
14             for(int i=0; i<length; i++ )
15             {
16                   myArray[i] = s.nextInt();
17             }
18
19             System.out.println(Arrays.toString(myArray));
20       }
21 }
```