**STRING HANDLING**

A string is a sequence of characters. In many languages, a string is treated a character array, but in Java, a string is treated as an Object. Even String constants are actually String objects.

For example:

**System.out.println(“This is a string”);**

In the above statement **“This is a String”** is a String constant, is also an object.

The String class has several constructors and more than 40 methods for manipulating strings.

**What is String Handling?**

Performing different operations on string is called string handling.

**In how many ways we can store String data?**

We can store in 4 ways:

1. Using char[] array
2. Using String class object
3. Using StringBuffer class object
4. Using StringBuilder class object

**Program demonstrating to store data in char[]**

class charArrayDemo

{

public static void main(String[] args)

{

char[] arr = { 'A' , 'S' , 'H' , 'U' };

for ( int i = 0; i < arr.length; i++ )

{

System.out.println(arr[i]);

}

}

}

**Why String class is given when char array is available?**

Due to some limitations in char[] array.

1. Char[] array has size limitation i.e. it is fixed in size.
2. Char[] array is of homogeneous nature.
3. In – built functions not available in char[] array.

Internal memory location on data structure of String object is char[]. So we can say that, String class is given to store data in char[] and further modifying the data.

**String object creation and memory structure**

String s1 = “abc”;

1010

2020

a

b

c

S1 0 1 2

1010 2020

String object char[] object

String object contains char[] object to store string characters.

In **java.lang** package, we have below three classes to store and perform different operations on string of characters.

1. String
2. StringBuffer
3. StringBuilder

All these three classes are siblings and they are subclasses of **CharSequence** interface. This interface was introduced in Java 5.

Below diagram shows the inheritance relation of these three classes:

**CharSequence**

**String**

**StringBuilder**

**StringBuffer**

**Object**

**Note:**

* All these three classes are **final** classes
* These three classes are also subclasses of **java.io.Serializable** interface.
* String class is a subclass of **Comparable** interface, but StringBuffer and StringBuilder are not.

**The String class**

The **String** class represents character strings. Any data that is placed in “ ” is treated by compiler and JVM as an instance of java.lang.String.

Strings are constant. Their values cannot be changed in the same memory after they are created. So **String** is defined as it is a **thread safe** **immutable sequence of characters.**

**Program to illustrate immutable nature of String**

class ImmutableDemo

{

public static void main(String[] args)

{

String s1 = "abc";

System.out.println("s1 hashcode: " + s1.hashCode());

s1 = "bcd";

System.out.println("s1 hashcode: " + s1.hashCode());

}

}

In the above example, we will observe that hashcode are changed. This proves that the value cannot be changed in the same memory after they are created.

class ImmutableDemo

{

public static void main(String[] args)

{

String s1 = "abc";

s1.concat("d"); // Garbage Collected

String s2 = s1.concat("e");

System.out.println("s1:" + s1); // abc

System.out.println("s2:" + s2); // abce

}

}

Memory structure of above program

2020

a

b

c

String s1 = “abc”;

1010

s1

1010 2020

3030

a

b

c

d

s1.concat(“d”);

4040 3030

a

b

c

e

5050

6060

String s2 = s1.concat(“e”);

s2 6060 5050

**What are different ways to construct String class object?**

There are two ways to construct String object:

1. By assigning String literal
2. By using new keyword and constructor.

**Example**

1. String s1 = “abc”; // Assigning String literal
2. String s2 = new String(“bbc”); // using new keyword and constructor

**Note:** Every string literal is a java.lang.String object. It means when JVM come across a string literal in the class, it creates an object reference with the string literal data and replaces this literal with object reference.

**// program to demonstrate that every string literal is java.lang.String object**

class StringImmutableTest

{

static void m1(String s)

{

System.out.println("s : " + s.hashCode());

}

public static void main(String[] args)

{

m1("abc");

System.out.println("abc : " + "abc".hashCode());

}

}

In the above example, we can see that in both the cases the hashcode is same.

In the above example, in m1() method call, first the string object is created for the argument and its reference is sent to the m1() definition.

**Memory structure for the above program**

2020

a

b

c

1010

**2**

m1( “abc” )

1

3

1010 2020

void m1(String s) 4

{} s

**What are the differences in creating String object with literal and constructor?**

There are 2 differences:

1. No. of object creation
2. String Pooling

**No. of object creation**

In String s1 = “a”, only one object is created.

In String s1 = new String(“b”), two object are created.

1. Argument String object if it is not yet created.
2. Current String object by copying argument String object character.

**String Pooling**

By default, String literal created in String Constant Pooled Area.

String constructor creates object in String Heap Area.

**Memory arrangement to demonstrate the differences in String literal and constructor**

String s1 = “a”;

String s2 = new String(“b”);

1010

2020

String Constant Pooled Area

1010

3030

String Heap Area

3030

Copying string object from pooled area

S1

S2

Here, String literal **s1** creates in String pooled area, whereas, String constructor creates in String heap area.

We can clearly observe that in case of String literal only one object is created which is in String pooled area.

But in case of String constructor two objects are created. First in String pooled area and Second in String heap area.

The internal concept behind object creation is that, compiler will first search for that String object in String pooled area.

1. If that String object is available in String pooled area, then the String literal will get associated to that object, whereas, in case of String constructor, compiler will first create the String object in String heap area and then the contents of the String pooled area object will get copied to the memory reference created in String heap area for the String constructor.
2. If that String object is not available in String pooled area, then compiler will first create the String object in the String pooled area and then associates the String literal, whereas, in case of String constructor, compiler will first create the String object in String heap area. Then it creates the String object in String Pooled area and copies the contents from String pooled area to the memory reference created in String heap area for the String constructor.

Thus, in case of String literal, object is created only once (in String pooled area), whereas, in case of String constructor, object created for twice (in String heap area as well as String pooled area).

1010

2020

String Constant Pooled Area

1010

3030

String Heap Area

3030

4040

4040

1010

String s1 = “a”;

String s2 = new String(“b”);

String s3 = new String(“a”);

S1

Copying string object from pooled area

S2

S3

**What is String pooling?**

String pooling means grouping of string constant objects.

A small collection object which is created to store **string constant object** references is called String Constant Pooled Area. (**In String s = “a”, “a” is the string constant object**)

When String is created using constructor, then String object reference will not get stored in String Constant Pooled Area. (In above memory structure 3030 and 4040 is not stored in SCPA)

So, this String pooling concept is applicable only for String literal object, not for String object created using new keyword and constructor.

**Why String is immutable?**

If we create String objects by assigning same string literal, only one object is created and all references variables are initialized with that same String object reference. Then all referenced variables are pointing to same String object. This is the reason, why String object becomes immutable. It means String data modification is not stored in same memory.

class StringImmutableTest

{

public static void main(String args[])

{

String s1 = "abc";

String s2 = "abc";

String s3 = "bbc";

System.out.println("s1 : " + s1.hashCode());

System.out.println("s2 : " + s2.hashCode());

System.out.println("s3 : " + s3.hashCode());

}

}

In the above example, hashcode of s1 and s2 are same because both are pointing to the same String object, whereas, hashcode of s3 is different.

**//program to demonstrate string pooling with JVM architecture**

class StringImmutableTest

{

public static void main(String args[])

{

String s1 = "a";

String s2 = "a";

String s3 = new String("a");

String s4 = new String("a");

System.out.println("s1 == s2: " + (s1 == s2));

System.out.println("s1.equals(s2) : " + s1.equals(s2);

System.out.println("s3 == s4: " + (s3 == s4));

System.out.println("s3.equals(s4) : " + s3.equals(s4));

}

}

Output

True

True

False

True

**JVM architecture of above program**

S2

S3

S4

S1

1010

1010

String Constant Pooled Area

1010

2020

String Heap Area

2020

3030

3030

1010

**Why String is created as immutable?**

1. To have the original value and modified value separately.
2. Same String data used in multiple places in project. All pointing to same object.

**Why this String pooling?**

To improve performance by saving the memory.

**When we should we use new keyword and constructor to create String object?**

We should use this approach for creating new String object by copying character from already existing string, not for creating very first String object, because it creates un – necessary two objects.

**Identify the number of objects created:**

String s1 = "a"; 🡪 1

String s2 = "a"; 🡪 0

String s3 = s1; 🡪 0

String s4 = new String("a"); 🡪 1

String s5 = new String("b"); 🡪 2

String s6 = "ab"; 🡪 1

String s7 = "a" + "b"; 🡪 0

String s8 = s1 + "b"; 🡪 1

String s9 = "a"; 🡪 0

String s10 = s1.concat("b"); 🡪 1

String s11 = s10.concat(""); 🡪 0

**//program to prove the above object creation**

class ObjectCreation

{

public static void main(String[] args)

{

String s1 = "a";

System.out.println("s1: " + System.identityHashCode(s1));

String s2 = "a";

System.out.println("s2: " + System.identityHashCode(s2));

String s3 = s1;

System.out.println("s3: " + System.identityHashCode(s3));

String s4 = new String("a");

System.out.println("s4: " + System.identityHashCode(s4));

String s5 = new String("b");

System.out.println("s5: " + System.identityHashCode(s5));

String s6 = "ab";

System.out.println("s6: " + System.identityHashCode(s6));

String s7 = "a" + "b";

System.out.println("s7: " + System.identityHashCode(s7));

String s8 = s1 + "b";

System.out.println("s8: " + System.identityHashCode(s8));

String s9 = "a";

System.out.println("s9: " + System.identityHashCode(s9));

String s10 = s1.concat("b");

System.out.println("s10: " + System.identityHashCode(s10));

String s11 = s10.concat("");

System.out.println("s11: " + System.identityHashCode(s11));

}

}

identityHashCode is used to referenced hashcode.

**Situations when new object is not created**

1. When string literal have same value
2. Current data object not modified
3. When operation on data results in already created data. (E.g.: “ab” / “a” + “b”)

**Situations when new object is created**

1. Using new keyword
2. Current data object modified
3. Expression have at least one variable (E.g.: “ab” / s1 + “b”)

**Is SCPA eligible for garbage collection?**

No, because it is not under programmer control.

**Is String literal object eligible for garbage collection?**

No

**// program to demonstrate SCPA not eligible for garbage collection**

class GCTest

{

public static void main(String[] args)

{

String s1 = "a";

String s2 = new String(s1);

System.out.println("s1: " + System.identityHashCode(s1));

System.out.println("s2: " + System.identityHashCode(s2));

s1 = null;

s2 = null;

String s3 = "a";

System.out.println("s3: " + System.identityHashCode(s3));

}

}

**Memory structure of above program before assigning null**

S1 SCPA

1010

2020

1010

a

S2

a

2020

**Memory structure of above program after assigning null**

S1 SCPA

1010

2020

1010

a

S2

2020

S3

1010

From the above memory structure it is cleared that SCPA is not eligible for garbage collection, because after assigning null in **s1** when we are assigning the same value to **s3 (as s1)** then the same memory reference is assigned to **s3** which was being assigned to **s1**.

**Is string reference variable is immutable?**

No [ String s1 = “a”, Here **s1** is string reference variable ]

**Can a string reference variable be making as immutable?**

Yes, by declaring as final.

class ImmutableDemo

{

public static void main(String[] args)

{

String s1 = "abc";

s1 = "a";

}

}

It will be compiled successfully

class ImmutableDemo

{

public static void main(String[] args)

{

final String s1 = "abc";

s1 = "a";

}

}

CE: cannot assign a value to final variable s1

**// find out how many objects are created and how many are eligible for GC**

class StringLiteralGC

{

public static void main(String[] args)

{

String s1 = "a";

String s2 = "a";

String s3 = "b";

String s4 = new String("c");

String s5 = new String("c");

String s6 = new String("C");

String s7 = s4;

s1 = null;

s3 = null;

s4 = null;

s5 = null;

}

}

**Solution**

Objects created: 5

Garbage Collected: 1

**JVM Architecture**

**Java Stack Area Heap Area**

5050

4040

7070

1010

1010

2020

4040

5050

7070

4040

S1

S2

S3

S4

S5

S6

S7

6060

3030

2020

1010

**String Constant Pooled Area**

Main thread

GC

**Explanation of the garbage collection in above program**

S5

The string literal **s1, s2** and **s3** are not eligible for garbage collection. So only **s4, s5, s6** and **s7** can be garbage collected. But GC will be applicable only on those objects which are dereferenced from the memory (**by assigning null)**. In the above program only **s4** and **s5** are dereferenced. But for GC there is one more rule is that, the dereferenced object memory should not be used by any other object. In case of **s4** memory reference, it is also used by **s7**, so this memory is not eligible for GC. In case of **s5** memory reference, it is not used by any other object, so this memory is eligible for GC.

**Note:**

1. An object which does not allow modification on its data and if it allows modification, but result is stored in different new object, not in current object memory, is called **immutable object**.
2. A **mutable object** allows modifications to be stored in current object memory itself.
3. The pre – defined immutable objects are String and all wrapper classes.
4. Wrapper classes do not allow modifications.
5. StringBuffer and StringBuilder are mutable objects.

**Can we develop custom immutable object?**

Yes, we can.

1. Define a class by declaring its entire variable as private.
2. Define a parameterized constructor for initializing object variable at least once.
3. Do not define setter method, so that data cannot be modified. If we want to allow modification, define setter method but store result in new object.
4. Define getter method to read or access values.

**// developing immutable object that does not allow modifications**

class Sample

{

private int x;

public Sample(int x)

{

this.x = x;

}

public int getX()

{

return x;

}

}

class CustomImmutable1

{

public static void main(String[] args)

{

Sample s = new Sample(15);

System.out.println(s.getX());

//s.x = 20; //CE: x has private access in Sample

}

}

**// developing immutable object that allows modifications but result stored in new object**

class Sample

{

private int x;

public Sample()

{}

public Sample(int x)

{

this.x = x;

}

public Sample setX(int x)

{

Sample s = new Sample();

s.x = x;

return s;

}

public int getX()

{

return x;

}

}

class CustomImmutable2

{

public static void main(String[] args)

{

Sample s1 = new Sample(10);

System.out.println("s1: " + s1.getX());

Sample s2 = s1.setX(15);

System.out.println("After modification");

System.out.println("s1: " + s1.getX());

System.out.println("s2: " + s2.getX());

}

}

**String Constructors**

A String class has several constructors. But among them, 8 are mostly used. They are:

1. public String()
2. public String(String s)
3. public String(StringBuffer sb)
4. public String(StringBuilder sb)
5. public String(char[] ch)
6. public String(char[] ch , int offset , int count)
7. public String(byte[] b)
8. public String(byte[] b , int offset , int count)

**String()**

Creates empty string object, not null String object.

String s = new String();

**String(String value)**

Creates String object with given String characters. It performs String copy operation.

String s1 = “a”;

String s2 = new String(s1);

System.out.println(s2); // a

**String(StringBuffer sb)**

Creates new String with the given StringBuffer object data. Performs String copy operation from StringBuffer object to String object.

**String(StringBuilder sb)**

Creates new String object with the given StringBuilder object content. Performs String copy operation from StringBuilder object to String object.

**String(char ch[])**

Creates String object with the given char array values. Performs String copy operation from char[] object to String object.

char[] ch = { ‘a’, ‘b’ , ‘c’ , ‘d’ , ‘e’ , ‘f’ };

String s = new String(ch);

System.out.println(s); // abcdef

**String(char[] ch , int offset , int count)**

Creates new String with the given count number of characters from the given offset in the char[] object. Here offset is the starting index from which characters must be copied.

char[] ch = { ‘a’, ‘b’ , ‘c’ , ‘d’ , ‘e’ , ‘f’ };

String s = new String(ch, 2 , 3);

System.out.println(s); // cde

**String(byte[] b)**

Creates new String object by copying the given byte[] numbers by converting them into their ASCII characters.

byte[] b = { 97 , 98 , 99 };

String s = new String(b);

System.out.println(s); // abc

**String(byte[] b , int offset , int count)**

Creates new String object with the given count number of bytes from the given offset in the byte[]. All bytes are stored in their ASCII characters form.

byte[] b = { 97 , 98 , 99 };

String s = new String(b , 1 , 1);

System.out.println(s); // b

**// program to demonstrate the above String constructors**

class StringConstructors

{

public static void main(String[] args)

{

String s1 = new String();

System.out.println(s1);

String s2 = "abc";

String s3 = new String(s2);

String s4 = new String("bbc");

String s5 = s4;

System.out.println("s2: " + s2);

System.out.println("s3: " + s3);

System.out.println("s4: " + s4);

char ch[] = { 'a' , 'b' , 'c' , 'd' , 'e' , 'f' };

String s6 = new String(ch);

System.out.println("s6: " + s6);

String s7 = new String(ch , 2 , 3);

System.out.println("s7: " + s7);

byte[] b = { 97 , 98 , 99 };

String s8 = new String(b);

System.out.println(s8);

String s9 = new String(b , 1 , 1);

System.out.println("s9: " + s9);

}

}

**We have 3 important rules on String constructors.**

**Rule 1**

For char[] and byte[] parameter constructor, the given offset and count argument value must be within the range [0 to Stringlength – 1], else it leads to **RE:java.lang.StringIndexOutOfBoundsException.**

byte[] b = { 97 , 98 , 99 };

String s = new String(b , 1 , 4); // **RE: StringIndexOutOfBoundsException: 4**

**// We donot have 4 bytes in the array**

**Rule 2**

We cannot pass null as argument directly to constructor. It leads to **CE: ambiguous error**, because it will matched with all parameters of String class constructors.

String s = new String(null); // **CE: ambiguous error**

**Rule 3**

We cannot create String object with null, it leads to **RE: java.lang.NullPointerException**.

String s1 = null;

String s2 = new String(s2); // **RE: java.lang.NullPointerException**

String s3 = new String( (StringBuffer)null); // **RE: java.lang.NullPointerException**

**STRING OPERATIONS**

1. **Checking String is empty or not**

The **isEmpty()** function is used to check whether the String is empty or not.

**public boolean isEmpty()**

class StringOperations

{

public static void main(String[] args)

{

String s1 = "Java Programming Langauge";

System.out.println(s1.isEmpty()); // false

String s2 = “ “;

Sytem.out.println(s2.isEmpty()); // false

String s3 = “”;

System.out.println(s3.isEmpty()); // true

}

}

1. **Find length of the string**

The **length()** function is used to find the length of the string.

**public int length()**

class StringOperations

{

public static void main(String[] args)

{

String s1 = "Java Programming Langauge";

System.out.println(s1.length()); // 25

String s2 = “ “;

System.out.println(s1.length()); // 1

String s3 = “”;

System.out.println(s3.length()); // 0

}

}

1. **Printing String objects**

The **toString()** method is used to print the string objects. This is called implicitly.

**public String toString()**

class StringOperations

{

public static void main(String[] args)

{

String s = "Java programming Language";

System.out.println(s.toString()); Java programming Language

System.out.println(s); Java programming Language

}

}

1. **Comparing two strings**

Two strings can be compared using **equals()** and **==** operator.

**equals()** method checks whether both strings have same content or not whereas == operator checks whether both strings referred to the same object or not. **equals()** and == are case sensitive.

**public boolean equals(Object o)**

class StringOperations

{

public static void main(String[] args)

{

String s1 = "Hello";

String s2 = new String(s1);

String s3 = "Hello";

System.out.println(s1.equals(s2)); // true

System.out.println(s1 == s2); // false

System.out.println(s1.equals(s3)); // true

System.out.println(s1 == s3); // true

}

}

1. **Comparing two strings ignoring case**

The **equalsIgnoreCase()** method is used to compare two strings ignoring case.

**public boolean equalsIgnoreCase(String s)**

class StringOperations

{

public static void main(String[] args)

{

String s1 = "Hello";

String s2 = "HEllo";

System.out.println(s1.equalsIgnoreCase(s2));

System.out.println(s1 == s2);

}

}

1. **Comparing two strings lexicographically**

This means after comparison method should return difference between ASCII values of String content. If both Strings will be equal, then output will be 0. This is case sensitive method.

**public int compareTo(String s)**

class StringOperations

{

public static void main(String[] args)

{

String s1 = "Hello";

String s2 = "HEllo";

String s3 = “Hello”;

System.out.println(s1.compareTo(s2)); // 32

System.out.println(s1.compareTo(s3)); // 0

}

}

class StringOperations

{

public static void main(String[] args)

{

String s1 = "abcdef";

String s2 = "abc";

System.out.println(s1.compareTo(s2)); // 3

System.out.println(s2.compareTo(s1); // -3

}

}

In the above program, both strings are differ in length, so compareTo() method will compare the difference between their length not the ASCII value.

1. **Comparing two strings lexicographically ignoring case**

The **compareToIgnoreCase()** method is used to compare to comparing strings lexicographically ignoring case.

**public int compareToIgnoreCase(String s)**

class StringOperations

{

public static void main(String[] args)

{

String s1 = "Hello";

String s2 = "HEllo";

System.out.println(s1.compareToIgnoreCase(s2)); // 0

}

}

1. **Checking the String starts with**

The **startsWith()** method is used to check whether the String is starts with particular String or not. This is a case sensitive method.

**public boolean startsWith(String s)**

class StringOperations

{

public static void main(String[] args)

{

String s1 = "Java Programming Language";

System.out.println(s1.startsWith("Java")); // true

System.out.println(s1.startsWith("java")); // false

}

}

1. **Checking the String ends with**

The **endsWith()** method is used to check whether the String ends with the particular String or not. This is a case sensitive method.

**public boolean endsWith(String s)**

class StringOperations

{

public static void main(String[] args)

{

String s1 = "Java Programming Language";

System.out.println(s1.endsWith("Language")); // true

System.out.println(s1.endsWith("language")); // false

}

}

1. **Reading character at given index**

The **charAt()** method is used to retrieve character at given index number. String index starts from zero, because its internal object is char array.

**public char charAt(int index)**

class StringOperations

{

public static void main(String[] args)

{

String s1 = "Java Programming Language";

System.out.println(s1.charAt(5)); // P

System.out.println(s1.charAt(26)); // SIOBE

}

}

If we pass index out of range, then JVM throws **StringIndexOutOfBoundsException**.

1. **Finding index of the given character**

The **indexOf()** method returns first occurrence of a given character or String from index 0.

**public int indexOf(char ch)**

**public int indexOf(String str)**

class StringOperations

{

public static void main(String[] args)

{

String s1 = "Java Programming Language";

System.out.println(s1.indexOf('o')); // 7

System.out.println(s1.indexOf("Language")); // 17

}

}

Another form of **indexOf()** method which returns first occurrence of given character or String by searching it from given index, not from index 0.

**public int indexOf(char ch, int fromIndex)**

**public int indexOf(String str, int fromIndex)**

class StringOperations

{

public static void main(String[] args)

{

String s1 = "Java Programming Language";

System.out.println(s1.indexOf('a')); // 1

System.out.println(s1.indexOf('a', 7)); // 10

}

}

The **lastIndexOf()** method returns last occurrence of the given character or String from the last index.

**public int lastIndexOf(char ch)**

**public int lastIndexOf(String str)**

class StringOperations

{

public static void main(String[] args)

{

String s1 = "Java Programming Language";

System.out.println(s1.lastIndexOf('a')); // 22

System.out.println(s1.lastIndexOf("Programming")); // 5

}

}

Another form of **lastIndexOf()** method returns the occurrence of given character or String from the given index, not from the last index.

**public int lastIndexOf(char ch, int fromIndex)**

**public int lastIndexOf(String str, int fromIndex)**

class StringOperations

{

public static void main(String[] args)

{

String s1 = "Java Programming Language";

System.out.println(s1.lastIndexOf('a')); // 22

System.out.println(s1.lastIndexOf('a', 17)); // 10

}

}

**Note:** If the given character or String is not present in the current String, then above functions return -1, not false or error.

class StringOperations

{

public static void main(String[] args)

{

String s1 = "Java Programming Language";

System.out.println(s1.indexOf('A')); // -1

System.out.println(s1.indexOf('A', 17)); // -1

System.out.println(s1.lastIndexOf('A')); // -1

System.out.println(s1.lastIndexOf('A',22)); // -1

}

}

Until version 1.4, to check String available or not **indexOf()** method is being used.

1. **Checking String availability**

In version 1.5, to check String availability **contains()** method is introduced, which returns true or false.

**public boolean contains(CharSequence s)**

We cannot pass character in the form of ‘ ’ (single quotes) in this method, because its parameter is CharSequence. So we must pass the character in the form of “ ” (double quotes).

class StringOperations

{

public static void main(String[] args)

{

String s1 = "Java Programming Language";

System.out.println(s1.contains(‘a’)); // false

System.out.println(s1.contains("a")); // true

System.out.println(s1.contains("A")); // false

}

}

1. **Retrieving substring from the String**

The **substring()** method is used to retrieve substring from the given String.

**public String substring(int fromIndex)**

class StringOperations

{

public static void main(String[] args)

{

String s1 = "Java Programming Language";

System.out.println(s1.substring(7)); // ogramming Language

}

}

Another form of **substring()** method returns the substring within the range of index.

**public String substring(int fromIndex, int toIndex)**

class StringOperations

{

public static void main(String[] args)

{

String s1 = "Java Programming Language";

System.out.println(s1.substring(7 , 10)); // ogr

}

}

**Note:** if (fromIndex == toIndex), then this method returns empty String.

class StringOperations

{

public static void main(String[] args)

{

String s1 = "Java Programming Language";

System.out.println(s1.substring(7 , 7)); // (empty)

}

}

class StringOperations

{

public static void main(String[] args)

{

String s1 = "Java Programming Language";

System.out.println(s1.substring(s1.indexOf('g') ,

s1.lastIndexOf('a' , 21))); // gramming L

}

}

**Rule**

1. The **toIndex** must always be greater than **fromIndex**, otherwise JVM throws **StringIndexOutOfBoundsException**.
2. The **toIndex** and **fromIndex** must not be a negative number, otherwise JVM throws **StringIndexOutOfBoundsException**.
3. The character at **toIndex** is not included in the resultant String.
4. The minimum value of **fromIndex** is 0 (zero) and the maximum value of **toIndex** is the length of the String.
5. **String objects concatenation**

Adding given string at the end of the current string and storing the result in the new String is called concatenation.

The **concat()** method is used to concatenate two strings.

**public String concat(String s)**

class StringOperations

{

public static void main(String[] args)

{

String s1 = "Java Programming";

String s2 = s1.concat(" Langauge");

System.out.println(s1); // Java Programming

System.out.println(s2); // Java Programming Langauge

}

}

Another way to perform String concatenation is + operator.

The binary operator + is overloaded in the sense that the operation performed is determined by the type of operands. When one of the operands is a String object, a string conversion is performed on the other operand, implicitly converted it into string representation, before the string concatenation is performed.

Non – String operands are converted as follows:

1. For an operand of a primitive data type, its value is first converted to a reference value using the object creation expression.
2. For all reference value operands, a string representation is constructed by calling the toString() method on the referred object.

**Example to illustrate both points**

String s = 100 + “ percent”; // “100 percent”

The integer literal 100 is implicitly converted to String “100” before concatenation as:

1. Creating an object of wrapper class Integer, which boxes the integer 100.  
   **new Integer(100);**
2. Creating a string from this object by using the toString() method.  
   **new Integer(100).toString();**

The string concatenation operator + is left associative and the result of concatenation is always a new String object.

class StringOperations

{

public static void main(String[] args)

{

String s1 = "a";

String s2 = s1 + "b";

System.out.println(s1); // a

System.out.println(s2); // ab

}

}

If we have to concat only two strings, then we must use **concat()** method.

If we have to concat more than two strings in a single line, it is recommended to use + operator.

class StringOperations

{

public static void main(String[] args)

{

String s1 = "Java";

String s2 = " Programming";

String s3 = " Langauge";

String s4 = s1 + s2 + s3;

String s5 = s1.concat(s2).concat(s3);

System.out.println(s4); // Java Programming Language

System.out.println(s5); // Java Programming Language

}

}

In the above program, we can observe that concatenation using + operator is more readable than using **concat()** method.

**s1.concat(s2).concat(s3)** **is known as method call chaining.**

**String expression calculation**

1. If an expression has only String literals, that expression is calculated by compiler and places result as literal. So, JVM creates that String object in String Pooled Area.
2. If an expression has a variable, compiler does not calculate it, it is calculated by JVM and JVM creates every String literal as object SCPA and resultant String in heap area.

String s1 = "ab"; 🡪 1

String s2 = "bb"; 🡪 1

String s3 = s1 + s2; 🡪 1

String s4 = "ab" + "x" + "y"; 🡪 1

String s5 = s1 + "x" + "y"; 🡪 4

String s6 = "x"; 🡪 0

String s7 = "abxy"; 🡪 0

**Memory structure of the above statements**

S1

S3

3030

3030

2020

S2

1010

1010

2020

4040

5050

6060

String Constant Pooled Area

7070

S4

4040

S5

8080

S6

5050

8080

4040

S7

In a String method call, if String object is not changed, then the current String object reference is returned. But, in case of + operator, it always creates and returns new String object irrespective of the object is changed or not.

class StringCatTest

{

public static void main(String[] args)

{

String s1 = "Ashu";

String s2 = s1.concat("");

String s3 = s1+ "";

System.out.println(s1 == s2); // true

System.out.println(s1 == s3); // false

}

}

1. **Changing all String characters to uppercase**

The **toUpperCase()** method is used to change the characters of the String to uppercase.

**public String toUpperCase()**

class StringOperations

{

public static void main(String[] args)

{

String s1 = "ashu";

String s2 = s1.toUpperCase();

System.out.println(s1); // ashu

System.out.println(s2); // ASHU

System.out.println(s1 == s2); // false

}

}

If String contains only numbers and special characters, then the above function will not effect to the String. The String remains unchanged.

class StringOperations

{

public static void main(String[] args)

{

String s1 = "12%$^#89";

String s2 = s1.toUpperCase();

System.out.println(s1); // 12%$^#89

System.out.println(s2); // 12%$^#89

System.out.println(s1 == s2); // true

}

}

class StringOperations

{

public static void main(String[] args)

{

String s1 = "ashu";

System.out.println(s1.toUpperCase()); // ASHU

System.out.println(s1); // ashu

System.out.println(s1 = s1.toUpperCase()); // ASHU

System.out.println(s1); // ASHU

}

}

1. **Changing all String characters to uppercase**

The **toLowerCase()** method is used to change the characters of the String to uppercase.

**public String toLowerCase()**

class StringOperations

{

public static void main(String[] args)

{

String s1 = "ASHU";

String s2 = s1.toLowerCase();

System.out.println(s1); // ASHU

System.out.println(s2); // ashu

System.out.println(s1 == s2); // false

}

}

If String contains only numbers and special characters, then the above function will not effect to the String. The String remains unchanged.

class StringOperations

{

public static void main(String[] args)

{

String s1 = "12%$^#89";

String s2 = s1.toLowerCase();

System.out.println(s1); // 12%$^#89

System.out.println(s2); // 12%$^#89

System.out.println(s1 == s2); // true

}

}

class StringOperations

{

public static void main(String[] args)

{

String s1 = "ASHU";

System.out.println(s1.toLowerCase()); // ashu

System.out.println(s1); // ASHU

System.out.println(s1 = s1.toLowerCase()); // ashu

System.out.println(s1); // ashu

}

}

1. **Replace old character to new character**

The **replace()** method is used to replace old character to new character.

**public String replace(char oldchar, char newchar)**

class StringOperations

{

public static void main(String[] args)

{

String s1 = "abc bbc cba";

String s2 = s1.replace('a' , 'b');

System.out.println(s2); // bbc bbc cbb

}

}

Another form of **replace()** method which accepts **CharSequence** arguments.

**public String replace(CharSequence oldString, CharSequence newString)**

class StringOperations

{

public static void main(String[] args)

{

String s1 = "abc bbc cba";

String s2 = s1.replace("ab" , "ba");

System.out.println(s2); // bac bbc cba

}

}

1. **Replace all String with new String**

The **replaceAll()** method is used to replace old String with new String.

**public String replaceAll(String oldStr, String newStr)**

class StringOperations

{

public static void main(String[] args)

{

String s1 = "abc bbc cba";

String s2 = s1.replaceAll("b" , "a");

System.out.println(s2); // aac aac caa

}

}

Another form of **replace()** method which replace only first occurrence given String with the new String.

class StringOperations

{

public static void main(String[] args)

{

String s1 = "abc bbc cba";

String s2 = s1.replaceFirst("b" , "a");

System.out.println(s2); // aac bbc cba

}

}

1. **Removing leading and trailing spaces**

The **trim()** method is used to remove leading and trailing (begin and end) spaces.

**public String trim()**

class StringOperations

{

public static void main(String[] args)

{

String s1 = " ashu akash singh ";

System.out.println(s1 + "= " + s1.length());

String s2 = s1.trim();

System.out.println(s2 + "= " + s2.length());

}

}

Output

ashu akash singh = 27

ashu akash singh= 21

From the above output we can easily observe that, **trim()** method removes only leading and trailing (begin and end) spaces. It does not remove middle spaces.

**How can we remove middle spaces with leading and trailing spaces?**

class StringOperations

{

public static void main(String[] args)

{

String s1 = " ashu akash singh ";

String s2 = s1.replace(" ","");

System.out.println(s1 + "= " + s1.length());

System.out.println(s2 + "= " + s2.length());

}

}

Output

ashu akash singh = 27

ashuakashsingh= 14

1. **Converting String into character array**

The **getChars()** method is used to extract more than one character at a time.

**void getChars(int sourceStart, int sourceEnd, char target[], int targetStart)**

Here, **sourceStart** specifies the index of the beginning of the substring. **sourceEnd** specifies the index of the end of the desired substring. That is, substring contains the characters from the **sourceStart** to **sourceEnd** (remember character at **sourceEnd** will not be included). The array will receive the characters is specified by **target**[]. The **targetStart** specifies the index within **target[]** at which substring will be copied.

class getCharsDemo

{

public static void main(String[] args)

{

String s1 = "Master in Computer Applications";

int start = 10;

int end = 15;

char[] s = new char[end - start];

s1.getChars(start , end , s , 0);

System.out.println(s); // Compu

}

}

From the above program and output, it is cleared that **sourceEnd** index character is not be included. In this program, the value of **targetStart** is 0. It defines that from 0th position in **s[]**, the substring will start copied. If we change its **(targetStart)** value, then JVM will throw **ArrayIndexOutOfBoundsException**. It is so because, now **s[]** have only 4 index left to store substring, and the length of substring is 5.

If we want to convert all the characters in a String object into character array, then we should use **toCharArray()** method. It returns array of characters for the entire String.

**char[] toCharArray()**

class ToCharArrayDemo

{

public static void main(String[] args)

{

String s1= "Master in Computer Applications";

char[] ch = s1.toCharArray();

for (int i = 0; i < s1.length(); i++)

System.out.print(ch[i]);

System.out.println();

}

}

1. **Converting character array into Strings**

To convert character array to String **valueOf()**method or **String(char [])** constructoris used.

class ToStringDemo1

{

public static void main(String[] args)

{

char[] ch1 = {'a' , 's' , 'h' , 'u'};

String s1 = String.valueOf(ch1);

System.out.println(s1); // ashu

char[] ch2 = {'a' , 'k' , 'a' , 's' , 'h'};

String s2 = new String(ch2);

System.out.println(s2); // akash

}

}

1. **Converting any data type into String type**

The **valueOf()** method is used to convert any data type into String data type. The String class also defines a set of static overloaded **valueOf()** methods to convert objects and primitive values in Strings.

**public static String valueOf(xxxType)**

Here, xxx means any Java data – type.

It is a overloaded method has a forms with parameter int, long, float, double, char, char[], boolean and Object.

**Note: there is no valueOf() methods that allows a byte or short as parameters.**

class ValueOfDemo

{

public static void main(String[] args)

{

Object obj = "Object";

String s1 = String.valueOf(obj);

System.out.println(s1);

char[] ch = {'C' , 'h' , 'a' , 'r' , ' ' , 'A' , 'r' , 'r' , 'a' , 'y'};

String s2 = String.valueOf(ch);

System.out.println(s2);

boolean b = true;

String s3 = String.valueOf(b);

System.out.println(s3);

char c = 'A';

String s4 = String.valueOf(c);

System.out.println(s4);

int i = 10;

String s5 = String.valueOf(i);

System.out.println(s5);

long l = 20L;

String s6 = String.valueOf(l);

System.out.println(s6);

float f = 3.5f;

String s7 = String.valueOf(f);

System.out.println(s7);

double d = 5.8;

String s8 = String.valueOf(d);

System.out.println(s8);

}

}

1. **Split the String into tokens**

To decompose the invoking String into parts and returns an array that contains the result, **split()** function is used.

**public String[] split(String s)**

class SplitDemo

{

public static void main(String[] args)

{

String s1 = "Master in Computer Applications";

String[] arr = s1.split(" ");

for (int i =0; i < arr.length; i++)

{

System.out.println(arr[i]);

}

}

}

Output

Master

In

Computer

Applications

Another form of **split()** method is used to split the strings into array, which takes **limit** as a parameter.

**public Sting[] split(String s, int max)**

The number of pieces is specified by max. If max is negative, then the invoking is fully decomposed. If max contains positive value, the last entry in the returned array contains the remainder of the invoking String. If max is 0, then the invoking String is fully decomposed.

class SplitDemo

{

public static void main(String[] args)

{

String s1 = "Master in Computer Applications";

String[] arr = s1.split(" " , 2);

for (int i =0; i < arr.length; i++)

{

System.out.println(arr[i]);

}

}

}

Output

Master

in Computer Applications

Here, in the above output, the value of max is 2, which means the String is decomposed in 2 parts as we can see the above output.

**Write a program to read first name and last name of the customer and print his full name on the console with the “Hi”.**

import java.util.Scanner;

class StringProgs

{

public static void main(String[] args)

{

Scanner scn = new Scanner(System.in);

System.out.print("Enter first name:");

String s1 = scn.nextLine();

System.out.print("Enter last name:");

String s2 = scn.nextLine();

System.out.println("Hi " + s1 + " " + s2);

}

}

**Write a program to validate email.**

import java.util.Scanner;

class EmailValidator

{

public static void main(String[] args)

{

Scanner scn = new Scanner(System.in);

while (true)

{

System.out.print("Enter email: ");

String s = scn.nextLine();

int fromIndex = s.indexOf( '@' );

if ( fromIndex != -1 && s.indexOf ( '.' , fromIndex ) != -1 )

{

System.out.println("Activation link is sent to ur alternate email id");

break;

}

else

{

System.out.println("Email is not valid");

}

}

}

}

**Write a program to read String from keyboard. Display it on console with every word first character in uppercase of the entered String.**

import java.util.Scanner;

class FirstAlphabetUpper

{

public static void main(String[] args)

{

Scanner scn = new Scanner(System.in);

System.out.print("Enter the String:");

String s = scn.nextLine();

String[] sarray = s.split(" ");

int length = sarray.length;

for (int i = 0; i < length; i++)

{

char ch = sarray[i].charAt(0);

String s1 = String.valueOf(ch);

String s2 = s1.toUpperCase();

String s3 = sarray[i].replaceFirst(s1, s2);

System.out.print(s3 + " ");

}

System.out.println();

}

}

**Write a program to find given string represents a file or not, if it is a file, find out it is a text file or not.**

import java.util.Scanner;

class FileCheck

{

public static void main(String[] args)

{

Scanner scn = new Scanner(System.in);

System.out.print("Enter file name:");

String filename = scn.nextLine();

if (filename.contains("."))

{

if (filename.contains(".txt"))

System.out.println("Text file");

else

System.out.println("Not a text file");

}

else

System.out.println("Not a valid file name");

}

}

**Write a program of Password Validator**

import java.util.Scanner;

import java.util.regex.\*;

class PasswordCheck

{

static boolean uppercase(String s)

{

Pattern pat = Pattern.compile("[A-Z]");

Matcher mat = pat.matcher(s);

if (mat.find())

return true;

System.out.println("Password must contain atleast 1 uppercase letter");

return false;

}

static boolean lowercase(String s)

{

Pattern pat = Pattern.compile("[a-z]");

Matcher mat = pat.matcher(s);

if (mat.find())

return true;

System.out.println("Password must contain atleast 1 uppercase letter");

return false;

}

static boolean specialchars(String s)

{

Pattern pat = Pattern.compile("[@#$%^&\*]");

Matcher mat = pat.matcher(s);

if (mat.find())

return true;

System.out.println("Password must contain atleast 1 special symbol");

return false;

}

public static void main(String[] args)

{

Scanner scn = new Scanner(System.in);

System.out.print("Enter password:");

String password = scn.nextLine();

if (password.length() >= 8)

{

if (uppercase(password))

if (lowercase(password))

if (specialchars(password))

System.out.println("Password is Valid");

}

else

System.out.println("Password must atleast 8 characters long");

}

}

**Write a program of password validator without using Regular Expression**

import java.util.Scanner;

import java.util.regex.\*;

class PasswordCheck

{

static boolean uppercase(String s)

{

char ch[] = s.toCharArray();

for (int i = 0; i < s.length(); i++)

{

if (Character.isUpperCase(ch[i]))

return true;

}

System.out.println("Password must contain atleast 1 uppercase letter");

return false;

}

static boolean lowercase(String s)

{

char ch[] = s.toCharArray();

for (int i = 0; i < s.length(); i++)

{

if (Character.isLowerCase(ch[i]))

return true;

}

System.out.println("Password must contain atleast 1 lowercase letter");

return false;

}

static boolean specialchars(String s)

{

Pattern pat = Pattern.compile("[@#$%^&\*]");

Matcher mat = pat.matcher(s);

if (mat.find())

return true;

System.out.println("Password must contain atleast 1 special symbol");

return false;

}

public static void main(String[] args)

{

Scanner scn = new Scanner(System.in);

System.out.print("Enter password:");

String password = scn.nextLine();

if (password.length() >= 8)

{

if (uppercase(password))

if (lowercase(password))

if (specialchars(password))

System.out.println("Password is Valid");

}

else

System.out.println("Password must atleast 8 characters long");

}

}

**Write a program to find the length and reverse a String without using pre – defined function**

import java.util.Scanner;

class StringReverse

{

static int len, count;

static int ulength(String s)

{

try

{

for ( int i = 0; 0 <= i; i++)

{

s.charAt(i);

count++;

}

}

catch(StringIndexOutOfBoundsException e)

{}

return count;

}

static String ureverse(String s)

{

String rev = "";

for ( int i = len - 1; i >= 0; i--)

{

rev = rev + s.charAt(i);

}

return rev;

}

public static void main(String[] args)

{

Scanner scn = new Scanner(System.in);

System.out.print("Enter the string:");

String s1 = scn.nextLine();

String s2;

len = StringReverse.ulength(s1);

System.out.println("Length is:" + len);

s2 = StringReverse.ureverse(s1);

System.out.println(s2);

}

}

**Write a program to check whether the entered String is palindrome or not.**

import java.util.Scanner;

class PalindromeCheck

{

static boolean isPalindrome(String str)

{

int low = 0;

int high = str.length() - 1;

while (low < high)

{

if (str.charAt(low) != str.charAt(high))

return false;

low++;

high--;

}

return true;

}

public static void main(String[] args)

{

Scanner scn = new Scanner(System.in);

System.out.print("Enter the string: ");

String str = scn.nextLine();

if (isPalindrome(str))

System.out.println("String is palindrome");

else

System.out.println("String is not palindrome");

}

}

**The StringBuffer class**

It is a thread – safe, mutable sequence of characters. A string buffer is like a string, but can be modified in the same memory location.

**Thread – safe** means certain operation on the StringBuffer are synchronized, so that when used by multiple threads, these operation are performed in orderly way.

**Mutable** means the character sequence in a StringBuffer can be changed and the capacity of the String can also be changed dynamically.

**How StringBuffer object can be created?**

StringBuffer object can be created only in one way