Strings in Java

A String is a series of characters enclosed within double quotes.

'S' ← character

"SWIFT" ← string

Handling character type of data:

char ch = 'S';

Handling String type of data:

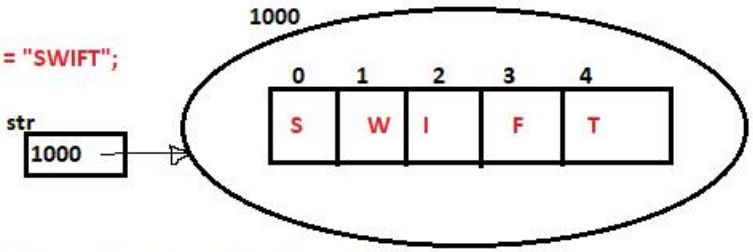
In C

char str[] = "SWIFT";

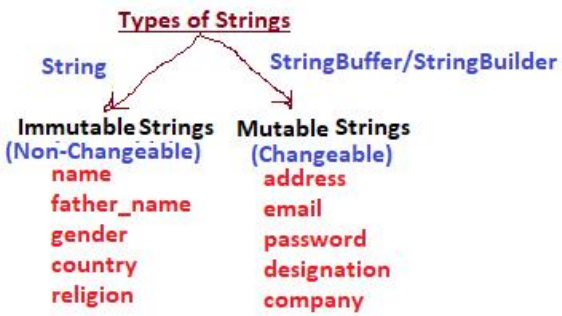
In C, a string is an array of characters terminated with a '\0' character.

In Java

String str = "SWIFT";



In Java, a string is an object containing the series of characters which are not null terminated.



IMMUTABLE STRINGS

Different ways of creating Immutable Strings:

1. String s = "ABCDE"; (String literal)
2. String s = new String("ABCDE"); (String)
3. char[] c = {'A','B','C','D','E'};
   String s = new String(c); (char[])
4. byte[] b = {65,66,67,68,69};
   String s = new String(b); (byte[])

NOTE:
There are 13 Constructors in the String class.

Different ways of Concatenating Strings:

1. concat() method
2. Concatenation operator (+)

Different ways of Comparing Strings:

1. Equality Operator (==) ---> String references are compared.

2. equals() ---> String values are compared by considering the case-sensitivity.
   equalsIgnoreCase() ---> String values are compared by ignoring the case-sensitivity.

3. compareTo() ---> String values are compared LEXICOGRAPHICALLY by considering the case-sensitivity.
   compareToIgnoreCase() ---> String values are compared LEXICOGRAPHICALLY by ignoring the case-sensitivity.

4. regionMatches() ---> Specific regions/portions/substrings are compared.

C++ ✓ Java ✓

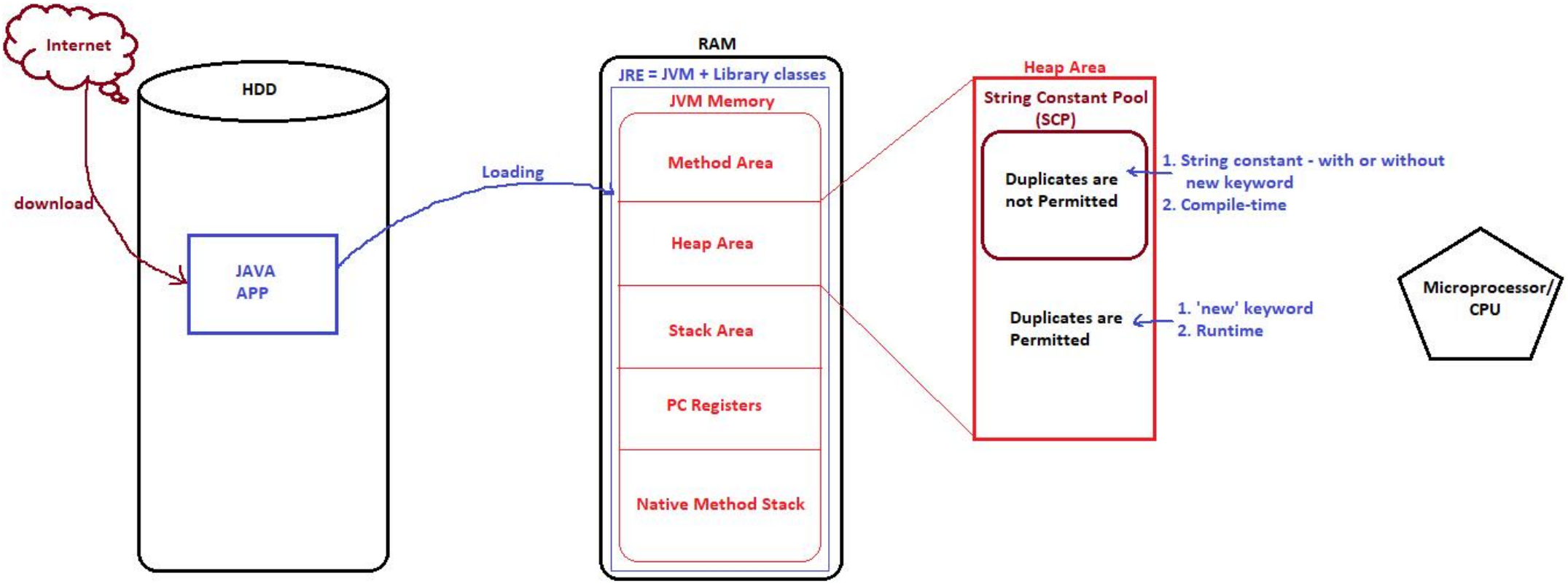
add(int a, int b)
add(float a, float b)

METHOD OVERLOADING

C++ ✓ Java ✗

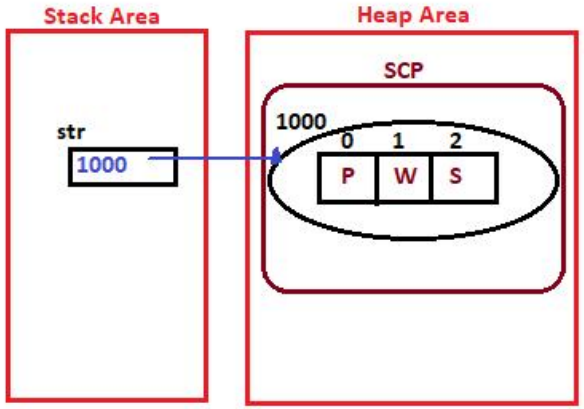
10 + 20 = 30 --> ADDITION
"PW" + "SKILLS" --> "PWSKILLS" CONCATENATION

OPERATOR OVERLOADING


Internet
HDD
download
JAVA
APP
Loading
RAM
JRE = JVM + Library classes
JVM Memory
Method Area
Heap Area
Stack Area
PC Registers
Native Method Stack
Heap Area
String Constant Pool
(SCP)
Duplicates are
not Permitted
1. String constant - with or without
new keyword
2. Compile-time
Duplicates are
Permitted
1. 'new' keyword
2. Runtime
Microprocessor/
CPU

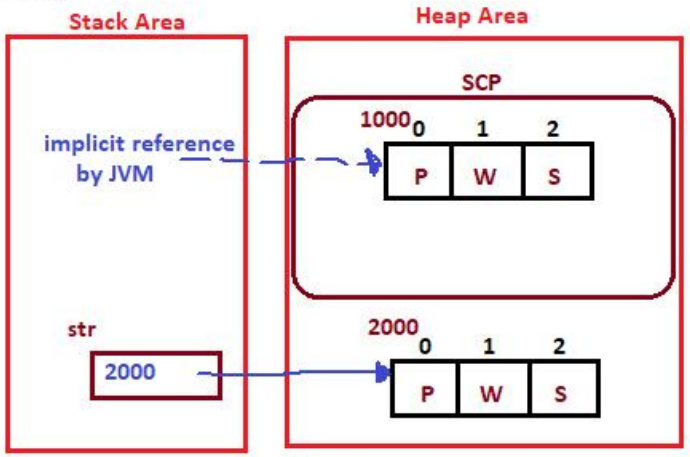
String str = "PWS";

S.o.p(str); //PWS



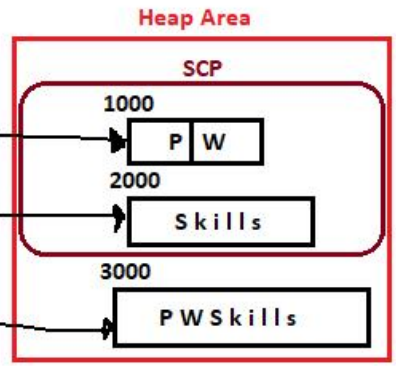
String str = new String("PWS");

S.o.p(str); //PWS



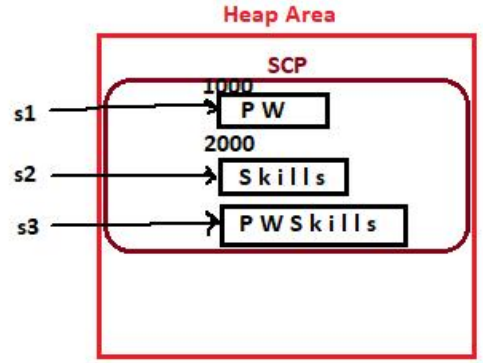
```
String s1 = "PW";
String s2 = "Skills";

String s3 = s1.concat(s2); // Method Call - Runtime Operation

S.o.p(s1); //PW
S.o.p(s2); //Skills
S.o.p(s3); //PWSkills
```

Method

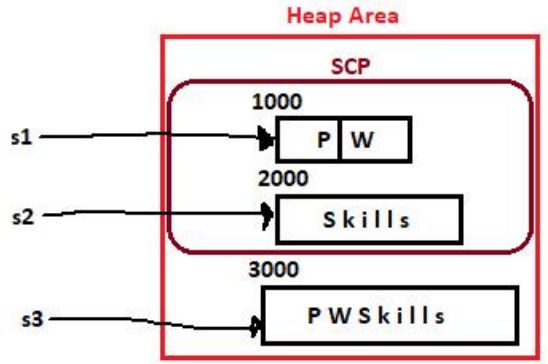



```
class String
{
   -----
   -----
String  concat()
   {
     ------
     ------
     ------
   }
}
```

```
String s1 = "PW";
String s2 = "Skills";

String s3 =  "PW" + "Skills";

Compiler → "PWSkills" ← JVM

S.o.p(s1); //PW
S.o.p(s2); //Skills
S.o.p(s3); //PWSkills
```



```
String s1 = "PW";
String s2 = "Skills";

String s3 = s1 + s2;
            Runtime

S.o.p(s1); //PW
S.o.p(s2); //Skills
S.o.p(s3); //PWSkills
```



```
String s1 = new String("Skills");
String s2 = new String("Skills");
```


```
S.o.p(s1 == s2); //false
```

Main Heap Area


```
String s1 = new String("Skills");
String s2 = new String("Skills");
```

---Memory Map----
---Same as above--

```
S.o.p(s1.equals(s2)); //true
```

Eg: Passwords

```
String s1 = new String("SKILLS");
String s2 = new String("skills");
```

--4 Objects--
--2 in SCP--
--2 in Main Heap Area--

```
S.o.p(s1.equals(s2)); //false
```

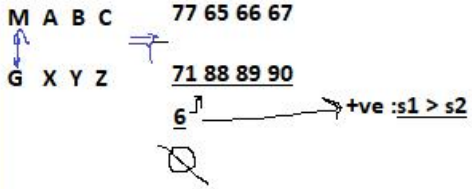
```
String s1 = new String("SKILLS");
String s2 = new String("skills");

S.o.p(s1.equalsIgnoreCase(s2)); //true
```

Eg: Email IDs

LEXICOGRAPHICALLY (Character by Character)

Arranging words in Dictionary

compareToIgnoreCase() -
Lexicographical comparison by ignoring case-sensitivity




```
String s1 = "MABC";
String s2 = "GXYZ";

int res = s1.compareTo(s2);

if(res == 0)
  S.o.p("String1 is equal to String2");
else if(res > 0)
  S.o.p("String1 is greater than String2");
else
  S.o.p("String1 is lesser than String2");
```

Output: String1 is greater than String2

```
String s1 = "AlakhPhy Pandey Sir";
String s2 = "SohanKE Pandey";
      S.o.p(s1.regionMatches(9, s2, 8, 6)); //true
```

2nd str

1st str — fromIndex in 1st str — fromIndex in 2nd str — No. of chars to be compared

String class Inbuilt Methods

```
          0 1 2 3 4 5 6  7  8 9 10 11 12 13 14
String s = "RajaRamMohanRoy";
```

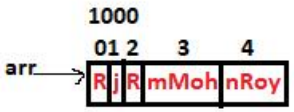
```
S.o.p(s); //RajaRamMohanRoy
//S.o.p(s[4]);  XError: Individual characters of a string cannot be directly accessed
S.o.p(s.charAt(4)); //R
S.o.p(s.indexOf('R')); //0
S.o.p(s.indexOf('Z')); //-1
S.o.p(s.indexOf('R', 3)); //4
S.o.p(s.indexOf("Ra")); //0
S.o.p(s.indexOf("Ra", 2)); //4
S.o.p(s.indexOf("KE")); //-1

S.o.p(s.lastIndexOf('R')); //12
S.o.p(s.lastIndexOf('R', 11)); //4
S.o.p(s.lastIndexOf("aR")); //3
```

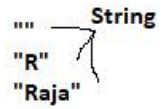
In arrays,
int[] a = {10,20,30,40,50};
S.o.p(a.length); //5 -->length is a variable

In strings,
String s = "Dhoni";
S.o.p(s.length()); //5 --> length() is a method

```
                0 1 2 3 4 5 6  7  8 9 10 11 12 13 14
String s = "RajaRamMohanRoy";
S.o.p(s.toUpperCase()); //RAJARAMMOHANROY

S.o.p(s); //RajaRamMohanRoy

S.o.p(s.toLowerCase()); //rajarammohanroy

S.o.p(s.substring(7)); //MohanRoy
S.o.p(s.substring(7, 12)); //Mohan
                  start     end
                  index     index
                  (Inclusive) (exclusive)

S.o.p(s.contains("Ram")); //true
S.o.p(s.contains("Sita")); //false

S.o.p(s.startsWith("Raja")); //true
S.o.p(s.endsWith("Roy")); //true
```

```
String s = "RajaRamMohanRoy";
String[] arr = s.split("a");

for( String elem : arr)
{
    S.o.p(elem);
}

Output:
R
j
R
mMoh
nRoy
```





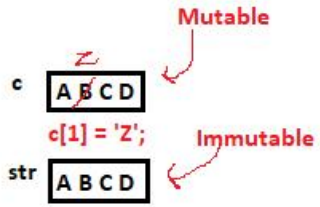

```
String s = "Think Twice Code Once";

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

for(String elem: arr)
{
  S.o.p(elem);
}

Output:
Think
Twice
Code
Once
```

```
String s = "RajaRamMohanRoy";

S.o.p(s.replace('R','M')); //MajaMamMohanMoy
S.o.p(s.replace("Ra", "Mo")); //MojaMomMohanRoy
```

```
String s = "---Raja--Ram-Mohan---Roy----";

S.o.p(s.trim()); //Raja--Ram-Mohan---Roy

S.o.p(s.strip()); //Raja--Ram-Mohan---Roy
S.o.p(s.stripLeading()); //Raja--Ram-Mohan---Roy----
S.o.p(s.stripTrailing()); //---Raja--Ram-Mohan---Roy

S.o.p(s.replace(" ","")); //RajaRamMohanRoy
```

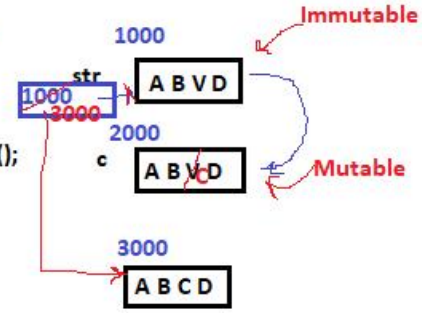
Character Array to String

```
char[] c = {'A','B','C','D'};
String str = new String(c);
```



String to Character Array

```
String str = "ABVD";

char[] c = str.toCharArray();

c[2] = 'C';

str = new String(c);
```




```
S.o.p(str); //ABCD
```

Mutable Strings

They are such strings which once initialized can be modified

Immutability

```
String str = new String("PW");
S.o.p(str);  //PW

str.concat("Skills");
S.o.p(str);  //PW
```

str --> "PW"
"PWSkills"

Mutability

```
StringBuffer sb = new StringBuffer("PW");
S.o.p(sb); //PW

sb.append("Skills");
S.o.p(sb); //PWSkills
```

sb ---> "PWSkills"

```
StringBuilder sb = new StringBuilder("PW");
S.o.p(sb); //PW

sb.append("Skills");
S.o.p(sb); //PWSkills
```

sb ---> "PWSkills"

Different ways of creating of Mutable Strings: [capacity()]

```
StringBuffer sb1 = new StringBuffer();
S.o.p(sb1.capacity()); //16

StringBuffer sb2 = new StringBuffer("Sachin");
S.o.p(sb2.capacity()); //22

StringBuffer sb3 = new StringBuffer(20);
S.o.p(sb3.capacity()); //20
```

```
StringBuffer sb = new StringBuffer();
S.o.p(sb.capacity());  //16

sb.append("Sachin");
S.o.p(sb.capacity());  //16

sb.append(" is a batsman."); //new capacity = (old capacity + 1) * 2 = (16 + 1) * 2 = 17 * 2 = 34
S.o.p(sb.capacity());  //34

sb.append(" He is also a MP.");  //(34+1)*2 = 70
S.o.p(sb.capacity());  //70
```

ensureCapacity(minCapacity)

new capacity will be largest of:

1. minCapacity
2. (oldcapacity+1)*2

ensureCapacity(25)

1. 25
2. (20+1)*2 = 42

ensureCapacity() method

This method can be used to change the capacity after the object is constructed. It ensures that the capacity is atleast equal to the specified minimum capacity.

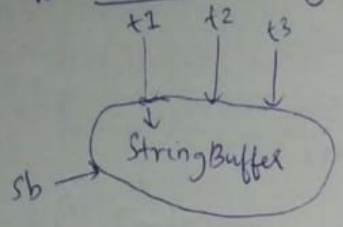
ensureCapacity(int minimumCapacity)

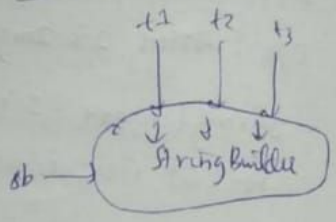
The new capacity is the largest of
1. minimumCapacity argument
2. old capacity * 2 + 2

Eg: StringBuffer sb = new StringBuffer(20);
S.o.p(sb.capacity()); //20
sb.ensureCapacity(25);
S.o.p(sb.capacity()); //42
sb.ensureCapacity(100);
S.o.p(sb.capacity()); //100
sb.ensureCapacity(200);
S.o.p(sb.capacity()); //202

StringBuilder sb = new StringBuilder(20);
S.o.p(sb.capacity()); //20
sb.ensureCapacity(25);
S.o.p(sb.capacity()); //42
sb.ensureCapacity(100);
S.o.p(sb.capacity()); //100
sb.ensureCapacity(200);
S.o.p(sb.capacity()); //202

Differences b/w StringBuffer and StringBuilder class






```
Synchronized
class StringBuffer
{
  synchronized append()
  synchronized length()
  synchronized charAt()
}
```

Java 1.0

```
class StringBuilder
{
  append()
  length()
  charAt()
}
```

Java 1.5

StringBuffer	StringBuilder
1) Commonality :-	
• Used to create mutable strings.	• Used to create mutable strings.
• Default initial capacity is 16.	• Default initial capacity is 16.
2) Differences :-	
• Introduced in Java 1.0 version	• Introduced in Java 1.5 version.
• Most of the methods present in this class are synchronized.	• No method present in this class is synchronized.
• At a time only one thread is allowed to operate on StringBuffer object and hence, it is thread-safe.	• At a time multiple threads are allowed to operate on StringBuilder object and hence it is not thread safe.
• Threads are required to wait to operate on StringBuffer object and hence relatively performance is low.	• Threads are not required to wait to operate on StringBuilder object and hence relatively performance is high.
- Not suitable for multi-threading.	• Suitable for multithreading.
• Slow in execution.	• Fast in execution.