Strings

- In java, string is basically an object that represents sequence of char values. The easiest way to represent a sequence of characters in Java is by using a character array :

```
char  charArray[] = new char[4];
char charArray[0] = 'J';
char charArray[0] = 'a';
char charArray[0] = 'v';
char charArray[0] = 'a';
```

- In java strings are class object and implemented using two classes, namely **String** and **StringBuffer**. Java string is an instantiated object of String class.
- Java String, as compared to C string are more reliable and predictable. A java string is not a character array and is not NULL terminated.
- There are two ways to create String object:
  - By string literal
  - By new keyword
- **String Literal :**
- Java String literal is created by using double quotes.
- For Example:

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

- **By new keyword :**
- String may be declared and created as follows :

```
String stringName;
stringName = new String("string");
   OR
String s=new String("string");
```

- **Example:**

```
String firstName;
firstName=new String("SYBCA");
```

- This two statements may be combined as followed:

```
String firstName=new String("SYBCA");
```

String Arrays

- We can also create and use array that contain strings. The statement

```
String itemArray[ ] = new String[3];
```

- Will create an itemArray of size 3 to hold three string constant. We can assign the strings to the itemArray element by element using three different statement or by using a for loop.

String Methods

- The String class defines a number of methods that allow us to accomplish a variety of string manipulation task.
- The java.lang.String class provides many useful methods to perform operations on strings.

1. **charAt():** it returns the character at given position.

- **Syntax:**

```
Str.charAt(int i)
```

- Here str is string and i specifies the position in integer. Position start from 0 index.
- **Example:**

```
String str= new String("sybca");
Str.charAt(3) // it returns c
```

2. **concat( ):** Concatenates specified string to the end of this string.

- **Syntax:**

```
Str.concat(str1);
```

- Here, str and str1 are strings.

```
String s1 = ”sy”;
String s2 = ”bca”;
String output = s1.concat(s2); // returns “sybca”
```

3. **equals( ):** Compares this string to the specified object.

- **Syntax:**

```
Str.equals(str1);
```

- Here, str and str1 are strings.

```
Boolean out = "bca".equals("bca"); // returns true
Boolean out = "BCA".equals("bca"); // returns false
```

4. **indexOf ():** Returns the index within the string of the first occurrence of the specified string, starting at the specified index.

- **Syntax:**

  Str.indexOf(str1, int i);

- Here, str is a string and str1 is specified sub string and i is starting index.

```
String s = "Learn Share Learn";
int output = s.indexOf("ea",3);// returns 13
```

- Here, indexOf method return index position from specified starting index 3 and find substring "ea" in specified string "Learn share learn"

5. **isEmpty() :** This method checks whether a String is empty or not. This method returns true if the given string is empty, else it returns false.

- **Syntax:**

  Str.isEmpty();

- Here, str is a string.

```
String s1 = "Learn Share Learn";
String s2=" ";
S1.isEmpty();  // returns false
S2.isEmpty();  // returns true
```

6. **join() :** method returns a string joined with a given delimiter. In the String join() method, the delimiter is copied for each element.

- **Syntax:**

  String join(CharSequence delimiter, CharSequence... elements)

- Here, delimiter : char value to be added with each element.
- elements : char value to be attached with delimiter
- It returns join string with delimiter.

```
String joinString1=String.join("-","welcome","to","Vapi");
System.out.println(joinString1);// returns welcome-to-Vapi
```

7. **lastIndexOf( ):** Returns the index within the string of the last occurrence of the specified string.

- **Syntax:**

  Str.lastIndexOf(str1);

- Here, str is a string and str1 is sub string.

```
String s = ”Learn Share Learn”;
int output = s.lastIndexOf("a"); // returns 14
```

8. length() : it returns the number of character in the string.

- **Syntax:**

  Str.length()

- Here str is a string variable or object.
- Example:

```
String str= new String("sybca");
Str.length() // it returns 5
```

9. split() : method breaks a given string around matches of the given regular expression. After splitting against the given regular expression, this method returns a char array.

- **Syntax:**

  String [ ] split (String regex, int limit)

- **Parameters:**

  regex – a delimiting regular expression

  Limit – the resulting threshold

  Returns: An array of strings is computed by splitting the given string.

  Throws: PatternSyntaxException – if the provided regular expression’s syntax is invalid.

- The limit parameter can have 3 values:

  limit > 0 – If this is the case, then the pattern will be applied at most limit-1 times, the resulting array’s length will not be more than n, and the resulting array’s last entry will contain all input beyond the last matched pattern.

  limit < 0 – In this case, the pattern will be applied as many times as possible, and the resulting array can be of any size.

limit = 0 – In this case, the pattern will be applied as many times as possible, the resulting array can be of any size, and trailing empty strings will be discarded.

10. substring(): it return the substring from the given index to end.

- **Syntax:**

  Str.substring(int i);

- Here str is string and i specifies the index value in integer.
- Example:

```
String str= new String("sybca");
Str.substring(2) // it returns bca
```

11. substring(int i, int j) : returns the substring from i to j-1 index.

- **Syntax:**
- Str.substring(int i,int j);
- Here i specifies the starting index in integer and j represent number of character from the starting index.
- Example:

```
String str= new String("sybca");
Str.substring(2,3) // it returns bca
```

12. trim(): Returns the copy of the String, by removing whitespaces at both ends. It does not affect whitespaces in the middle.

```
String word1 = " Learn Share Learn ";
String word2 = word1.trim(); // returns "Learn Share Learn"
```

Extra Functions:

1. **equalsIgnoreCase (String anotherString):** Compares string to another string, ignoring case considerations.

```
Boolean out= "bca".equalsIgnoreCase("bca"); // returns true
Boolean out = "bca".equalsIgnoreCase("BCA"); // returns true
```

2. **compareTo(String anotherString):** Compares two string lexicographically.

```
int out = s1.compareTo(s2);  // where s1 ans s2 are
                             // strings to be compared
```

```
This returns difference s1-s2. If :
out < 0  // s1 comes before s2
out = 0  // s1 and s2 are equal.
out > 0   // s1 comes after s2.
```

3. **compareToIgnoreCase(String anotherString):** Compares two string lexicographically, ignoring case considerations.

```
int out = s1.compareToIgnoreCase(s2);
// where s1 ans s2 are strings to be compared

This returns difference s1-s2. If :
out < 0  // s1 comes before s2
out = 0   // s1 and s2 are equal.
out > 0   // s1 comes after s2.
```

4. **toLowerCase():** Converts all the characters in the String to lower case.

```
String word1 = "HeLLo";
String word3 = word1.toLowerCase(); // returns "hello"
```

5. **toUpperCase():** Converts all the characters in the String to upper case.

```
String word1 = "HeLLo";
String word2 = word1.toUpperCase(); // returns "HELLO"
```

6. **replace (char oldChar, char newChar):** Returns new string by replacing all occurrences of *oldChar* with *newChar.*

```
String s1 = "bba and bca";
String s2 = s1.replace('b' ,'c'); // returns "cca and cca"
```

StringBuffer Class

- **StringBuffer** is a peer class of **String** that provides much of the functionality of strings.
- String represents **fixed-length, immutable** character sequences while StringBuffer represents growable and writable character sequences.
- **StringBuffer** may have characters and substrings inserted in the middle or appended to the end. It will automatically grow to make room for such additions and often has more characters reallocated than are actually needed, to allow room for growth.

StringBuffer Constructors

- **StringBuffer( ):** It reserves room for 16 characters without reallocation.

  StringBuffer str1=new StringBuffer();

  StringBuffer s=new StringBuffer(int size);

- **StringBuffer(int size) :** It accepts an integer argument that explicitly sets the size of the buffer.

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

  StringBuffer s=new StringBuffer("ROFEL BCA");

- **length( ) and capacity( ):** The length of a StringBuffer can be found by the length( ) method, while the total allocated capacity can be found by the capacity( ) method.

```
import java.io.*;

class Prog
{
      public static void main(String[] args)
      {
         StringBuffer s = new StringBuffer("SYBCA vapi");
         int p = s.length();
         int q = s.capacity();
         System.out.println("Length of string is : " + p);
         System.out.println("Capacity of string is : " + q);
      }
}
```

- **append( ):** It is used to add text at the end of the existence text. Here are a few of its forms:

  StringBuffer append(String str)

  StringBuffer append(int num)

```
import java.io.*;
class Prog
```

```
{
	public static void main(String[] args)
	{
		StringBuffer s = new StringBuffer("Hello");
		s.append("Everyone");
		System.out.println(s); // returns HelloEveryone
		s.append(1);
		System.out.println(s); // returns HelloEveryone1
	}
}
```

- **insert( ):** It is used to insert text at the specified index position.
- These are a few of its forms:

  StringBuffer insert(int index, String str)

  StringBuffer insert(int index, char ch)

- Here, *index* specifies the index at which point the string will be inserted into the invoking StringBuffer object.

```
import java.io.*;
class Prog
{
	public static void main(String[] args)
	{
		StringBuffer s = new StringBuffer("Helloone");
		s.insert(4, "every");
		System.out.println(s); // returns Helloeveryone

		s.insert(0, 5);
		System.out.println(s); // returns 5Helloeveryone

		s.insert(3, true);
		System.out.println(s); // returns 5Hetruelloeveryone

		s.insert(5, 41.3d);
		System.out.println(s);//returns 5Hetru41.3delloeveryone

		char geeks_arr[] = { 'p', 'a', 'w', 'a', 'n' };

		// insert character array at offset 9
		s.insert(2, geeks_arr);
		System.out.println(s);//returns 5Hpawanetruelloeveryone
	}
}
```

- **reverse( ):** It can reverse the characters within a StringBuffer object using reverse( ).This method returns the reversed object on which it was called.

```
import java.io.*;
class Prog
{
	public static void main(String[] args)
	{
		StringBuffer s = new StringBuffer("Helloeveryone");
		s.reverse();
		System.out.println(s); // returns enoyreveolleH
	}
}
```

- **delete( ) and deleteCharAt( ):** It can delete characters within a StringBuffer by using the methods delete( ) and deleteCharAt( ).
- The delete( ) method deletes a sequence of characters from the invoking object.
- Here, start Index specifies the index of the first character to remove, and end Index specifies an index one past the last character to remove. Thus, the substring deleted runs from start Index to endIndex–1. The resulting StringBuffer object is returned.
- **The deleteCharAt( )** method deletes the character at the index specified by *loc.* It returns the resulting StringBuffer object.These methods are shown here:

StringBuffer delete(int startIndex, int endIndex)

StringBuffer deleteCharAt(int loc)

```
import java.io.*;
class Prog
{
	public static void main(String[] args)
	{
		StringBuffer s = new StringBuffer("HelloEveryone");
		s.delete(0, 5);
		System.out.println(s); // returns Everyone
		s.deleteCharAt(7);
		System.out.println(s); // returns Everyon
	}
}
```

- **replace( ):** It can replace one set of characters with another set inside a StringBuffer object by calling replace( ). The substring being replaced is specified by the indexes start Index and endIndex. Thus, the substring at start Index through endIndex–1 is replaced. The replacement string is passed in str.The resulting StringBuffer object is returned.Its signature is shown here:

  StringBuffer replace(int startIndex, int endIndex, String str)

```
import java.io.*;

class GFG
{
      public static void main(String[] args)
      {
            StringBuffer s = new StringBuffer("HelloEveryone");
            s.replace(5, 8, "all");
            System.out.println(s); // returns Helloallone
      }
}
```

What is error? Explain types of error.

- An Error indicates serious problem that a reasonable application should not try to catch.
- An error may produce an incorrect output or may terminate the execution of the program or even may cause the system to crash.

Types:

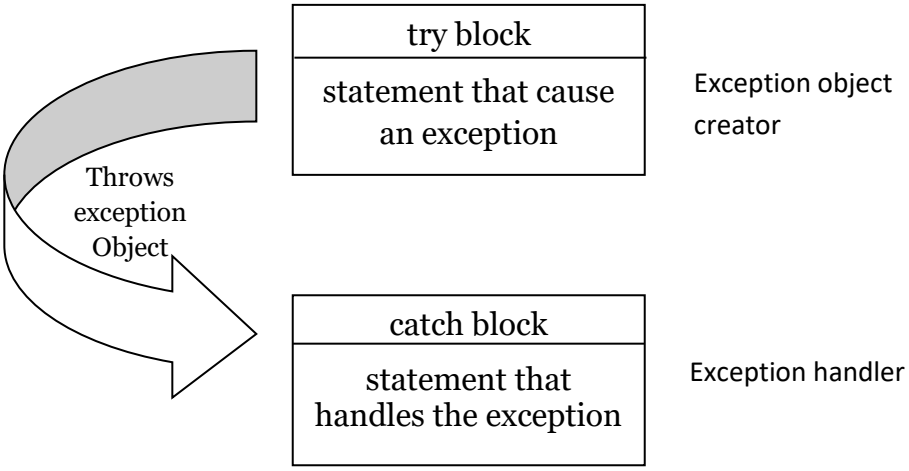
1. Compile time errors
2. Run time errors

- **Compile-time errors :** all syntax error will be detected and displayed by the java compiler and therefore these errors are known as compile time errors. Most of the compile-time errors are due to typing mistakes. The most common errors are:
  - Missing semicolons.
  - Missing brackets in classes and methods.
  - Misspelling of identifiers and keywords.
  - Missing double quotes in strings.
  - Use of undeclared variables etc.

- **Run-Time Errors:** Sometimes, a program may compile successfully creating the .class file but may not run properly. Such programs may produce wrong due to wrong logic or may terminate due to errors such as stack overflow.
- Most common run-time errors are:
  - Dividing an integer by zero.
  - Accessing an element that is out of the bounds of an array.
  - Trying to store a value into an array of an incompatible class or type.
  - Accessing a character that is out of bounds of a string etc.

Exception Handling in Java.

- An exception is a condition that is caused by a run-time error in the program. When the Java interpreter encounters an error such as dividing an integer by zero, it creates an exception object and throws it.
- If the exception object is not caught and handled properly, the interpreter will display an error message and will terminate the program.
- If we want the program to continue with the execution of the remaining code, then we should try to catch the exception object thrown by the error condition and then display an appropriate message for taking corrective actions. This task is known as exception handling.
- The purpose of exception handling mechanism is to provide a means to detect and report an "exceptional circumstance" so that appropriate action can be taken.
- Exceptions in Java can be categorized into two types:
- **Checked exception:** These exceptions are extended from the java.lang.Exception class.
- **Unchecked exception:** These exceptions are not essentially handled in the program code, instead the JVM handles such exceptions. Unchecked exceptions are extended from the java.lang.RuntimeException class.
- For exception handling we have to follow following task:
    - Find the problem (Hit the exception)
    - Inform that an error has occurred (Throw the exception)
    - Receive the error information (Catch the exception)
    - Take corrective actions (Handle the execution)
- The basic concept of exception handling are throwing an exception and catching it.



- Java uses a keyword try to preface a block of code that is likely to cause an error condition and "throw" an exception.
- A catch block defined by the keyword catch "catches" the exception "thrown" by the try block and handles it appropriately.
- The catch block immediately after the try block.
- The following example illustrate the use of simple try and catch statement.

```
----------
----------
----------
try
{
      Statement;    // generates an exception
}
Catch (Exception type e)
{
      Statement;    // process the exception
}
----------
----------
```

- The try block can have one or more statements that could generate an exception. If anyone statement generates an exception, the remaining statements in the block are skipped and execution jumps to the catch block that is placed next to the try block.
- The catch block too can have one or more statements that are necessary to process the exception.
- Try statement should be followed by at least one catch statement, otherwise compilation error will occur.

- Example :

```
class error1
{
  public static void main(String args[])
  {
      int a=10;
      int b=5;
      int c=5;
      int x,y;
      try
      {
        x=a/(b-c);
      }
      catch(ArithmeticException e)
      {
        System.out.println("Divided by zero");
      }
      y=a/(b+c);
      System.out.println("y="+y);
  }
}
```

- Following are some common exceptions

Exception Type	**Cause of Exception**
ArithmeticException	Cause by math error such as division by zero
ArrayIndexOutOfBoundException	Cause by bad array indexes
ArrayStoreException	Caused when a program tries to store the wrong type of data in an array
FileNotFoundException	Caused by an attempt to access a nonexistent file
IOException	Caused by general I/O failure, such as inability to read from a file
NullPointerException	Caused by referencing a null pointer
NumberFormatException	Caused when a conversion between strings and number fails
OutOfMemoryException	Caused when there's not enough memory to allocate a new object
SecurityException	Caused when an applet tries to perform an action not allowed by the browser's security setting
StackOverFlowException	Caused when the system runs out of stack space
StringIndexOutOfBoundsException	Caused when a program attempts to access a nonexistent character position in a string

Multiple Catch Statement

- It is possible to have more than one catch statement in the catch block as illustrated below:

```
try
{
      Statement;   // generates an exception
}
Catch (Exception _type1 e)
{
      Statement;   // process the exception type 1
}
Catch (Exception _type2 e)
{
      Statement;   // process the exception type 2
}
.
.
.
Catch (Exception _typeN e)
{
      Statement;   // process the exception type N
}
----------
----------
```

- When an exception in a try block is generated, the java treats the multiple catch statements like cases in a switch statement. The first statement whose parameter matches with the exception object will be executed and the remaining statement will be skipped.

```
public class MultipleCatchBlock1
{
    public static void main(String[] args)
    {
        try
        {
             int a[]=new int[5];
             a[5]=30/0;
        }
        catch(ArithmeticException e)
        {
            System.out.println("Arithmetic Exception occurs");
        }
        catch(ArrayIndexOutOfBoundsException e)
        {
             System.out.println("Array Index Exception occurs");
        }
        catch(Exception e)
        {
             System.out.println("Parent Exception occurs");
        }
              System.out.println("rest of the code");
    }
}
```

Java finally block

- Java finally block is a block that is used to execute important code such as closing connection, stream etc.
- Java finally block is always executed whether exception is handled or not.
- Java finally block follows try or catch block.
- If you don't handle exception, before terminating the program, JVM executes finally block(if any).
- Finally block in java can be used to put "cleanup" code such as closing a file, closing connection etc.

```
public class TestFinallyBlock2
{
  public static void main(String args[])
  {
     try
     {
         int data=25/0;
         System.out.println(data);
      }
      catch(ArithmeticException e)
      {
        System.out.println(e);
      }
      finally
      {
        System.out.println("finally block is always executed");
      }
       System.out.println("rest of the code...");
  }
}
```

Throw and Throws

- **Throw**
- The throw keyword in Java is used to explicitly throw an exception from a method or any block of code.
- The throw keyword is mainly used to throw custom exceptions.
- **Syntax:**

```
throw Instance
```

- **Example:**

```
throw new ArithmeticException("/ by zero");
```

- But this exception i.e, Instance must be of type Throwable or a subclass of Throwable. For
- Example Exception is a sub-class of Throwable and user defined exceptions typically extend Exception class.
- Data types such as int, char, floats or non-throwable classes cannot be used as exceptions.
- The flow of execution of the program stops immediately after the throw statement is executed and the nearest enclosing try block is checked to see if it has a catch statement that matches the type of exception.
- If it finds a match, controlled is transferred to that statement otherwise next enclosing try block is checked and so on. If no matching catch is found then the default exception handler will halt the program.
- **Example:**

```
class PROG
{
      public static void main(String[] args)
      {
            try
            {
                  // double x=3/0;
                  throw new ArithmeticException();
            }
            catch (ArithmeticException e)
            {
                  System.out.println(e);
            }
      }
}
```

- **Example 2:**

```
class ExpThrow
{
      static void checkNum(int n)
      {
        if(n<1)
        {
              throw new ArithmeticException("Number is negative");
        }
        else
        {
              System.out.println("Square of "+n+" is"+(n*n));
        }
       }
}
```

```
public static void main(String args[])
{
        Try
        {
              checkNum(-1);
        }
        catch(ArithmeticException e)
        {
              System.out.println(e);
        }
        System.out.println("Rest of Code");
}
```

- **Throws**
- throws is a keyword in Java which is used in the signature of method to indicate that this method might throw one of the listed type exceptions. The caller to these methods has to handle the exception using a try-catch block.
- **Syntax:**

```
type method_name(parameters) throws exception_list
```

- *exception_list* is a comma separated list of all the exceptions which a method might throw.
- We can use throws keyword to delegate the responsibility of exception handling to the caller (It may be a method or JVM) then caller method is responsible to handle that exception.
- **Example:**

```
public class TestThrows
{
    int divideNum(int m, int n) throws ArithmeticException
    {
      int div = m / n;
      return div;
    }
    public static void main(String[] args)
    {
         TestThrows obj = new TestThrows();
         try
         {
             System.out.println(obj.divideNum(45, 0));
         }
         catch (ArithmeticException e)
         {
             System.out.println("\nNumber cannot be divided by 0");
         }
         System.out.println("Rest of the code..");
    }
}
```

User Defined Exception

- Java provides us facility to create our own exceptions which are basically derived classes of Exception. For example MyException in below code extends the Exception class.
- We pass the string to the constructor of the super class- Exception which is obtained using "getMessage()" function on the object created.

```
// A Class that represents user-defined exception
class MyException extends Exception
{
      public MyException(String s)
      {
            // Call constructor of parent Exception
            super(s);
      }
}

// A Class that uses above MyException
public class Main
{
      // Driver Program
      public static void main(String args[])
      {
            try
            {
                  // Throw an object of user defined exception
                throw new MyException("User Defined error occur");
            }
            catch (MyException ex)
            {
                  System.out.println("Caught");

                  // Print the message from MyException object
                  System.out.println(ex.getMessage());
            }
      }
}
```

- Here, constructor of MyException requires a string as its argument. The string is passed to parent class Exception's constructor using super(). The constructor of Exception class can also be called without a parameter and call to super is not mandatory.

Example 2:

```
// A Class that represents use-defined exception

      class MyException extends Exception
      {

      }

// A Class that uses above MyException
      public class setText
      {
            // Driver Program
            public static void main(String args[])
            {
                  try {
                        // Throw an object of user defined exception
                        throw new MyException();
                  }
                  catch (MyException ex)
                  {
                        System.out.println("Caught");
                        System.out.println(ex.getMessage());
                  }
            }
      }
```