JAVA BASICS

By
Sudha Agarwal

INDEX

- Exception Handling
- Throw/ Throws Keyword
- Finally Block
- User Defined Exceptions
- Strings
- StringBuffer

EXCEPTION HANDLING

❖An exception (or exceptional event) is a problem that arises during the execution of a program.

❖ When an Exception occurs the normal flow of the program is disrupted and the program/Application terminates abnormally, which is not recommended, therefore, these exceptions are to be handled.

EXCEPTION HANDLING

❖An exception can occur for many different reasons. Following are some scenarios where an exception occurs.

❖A user has entered an invalid data.

❖A file that needs to be opened cannot be found.

❖A network connection has been lost in the middle of communications or the JVM has run out of memory.

❖Some of these exceptions are caused by user error, others by programmer error, and others by physical resources that have failed in some manner.

EXCEPTION HANDLING

- Based on these, we have three categories of Exceptions:
  - **Checked exceptions** – A checked exception is an exception that occurs at the compile time, these are also called as compile time exceptions.
  - **Unchecked exceptions** – An unchecked exception is an exception that occurs at the time of execution. These are also called as Runtime Exceptions. These include programming bugs, such as logic errors or improper use of an API.
  - **Errors** – These are not exceptions at all, but problems that arise beyond the control of the user or the programmer. Errors are typically ignored in your code because you can rarely do anything about an error. For example, if a stack overflow occurs, an error will arise. They are also ignored at the time of compilation.

EXCEPTION HIERARCHY

❖All exception classes are subtypes of the **java.lang.Exception** class.

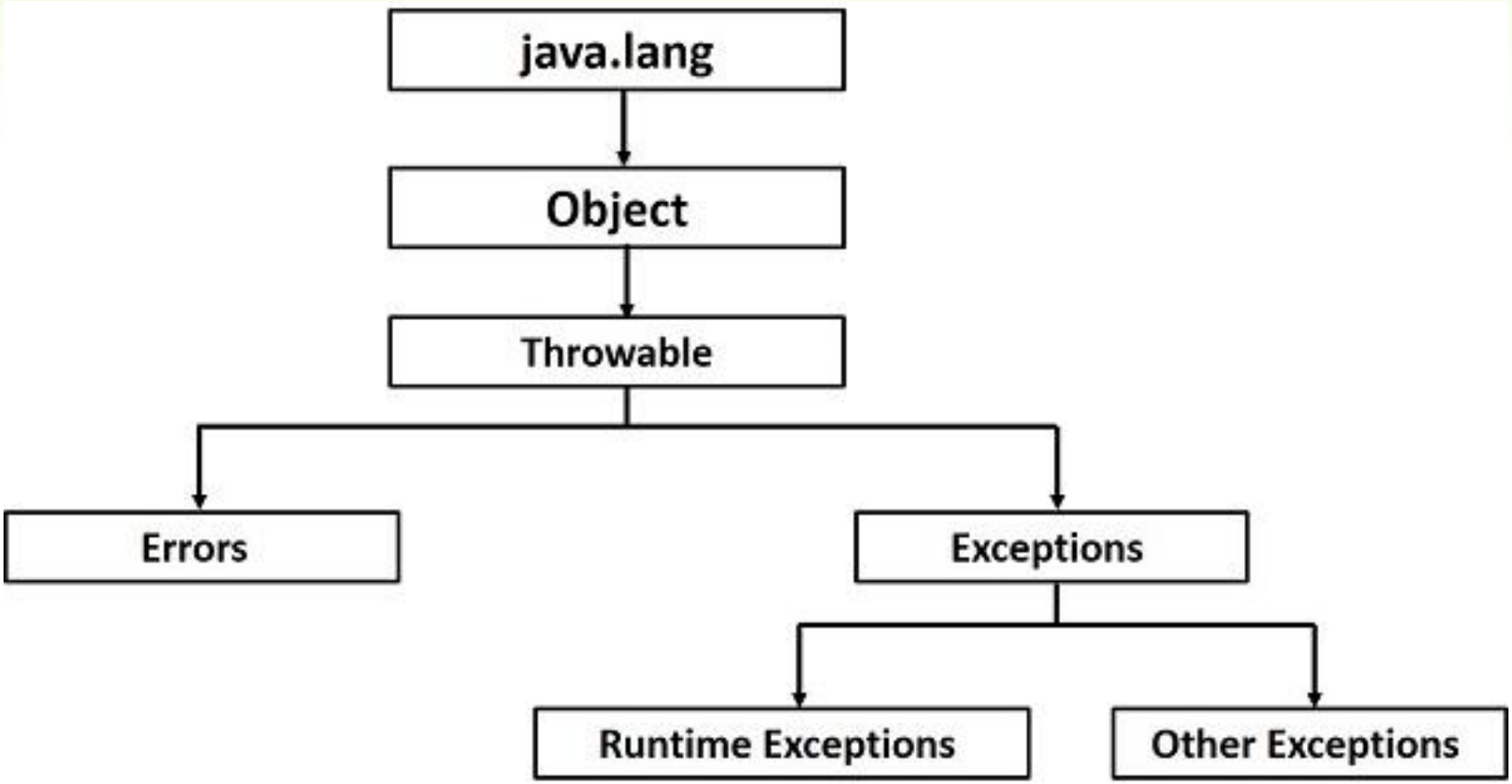
❖The exception class is a subclass of the **Throwable** class.

❖Other than the exception class there is another subclass called **Error** which is derived from the Throwable class.

❖Errors are abnormal conditions that happen in case of severe failures, these are not handled by the Java programs. Errors are generated to indicate errors generated by the runtime environment. Example: JVM is out of memory. Normally, programs cannot recover from errors.

EXCEPTION HIERARCHY

❖The Exception class has two main subclasses: **IOException** class and **RuntimeException** Class.



EXCEPTION METHODS

❖**public String getMessage() –** Returns a detailed message about the exception that has occurred. This message is initialized in the Throwable constructor.

❖**public String toString()** – Returns the name of the class concatenated with the result of getMessage().

❖**public void printStackTrace()** – Prints the result of toString() along with the stack trace to System.err, the error output stream.

CATCHING EXCEPTIONS

❖A method catches an exception using a combination of the **try and catch keywords**.

❖A try/catch block is placed around the code that might generate an exception.

❖Code within a try/catch block is referred to as protected code, and the syntax for using try/catch looks like

```
try {
   // Protected code
catch(ExceptionName e1) {
   // Catch block
}
```

CATCHING EXCEPTIONS

❖The code which is prone to exceptions is placed in the try block. When an exception occurs, that exception occurred is handled by catch block associated with it.

❖ Every try block should be immediately followed either by a catch block or finally block.

❖A catch statement involves declaring the type of exception you are trying to catch. If an exception occurs in protected code, the catch block (or blocks) that follows the try is checked.

CATCHING MULTIPLE TYPE EXCEPTIONS

❖Since Java 7, you can handle more than one exception using a single catch block, this feature simplifies the code.

```
catch (IOException|FileNotFoundException ex) {
   logger.log(ex);
```

CATCHING EXCEPTIONS

❖The code which is prone to exceptions is placed in the try block. When an exception occurs, that exception occurred is handled by catch block associated with it.

❖ Every try block should be immediately followed either by a catch block or finally block.

❖A catch statement involves declaring the type of exception you are trying to catch. If an exception occurs in protected code, the catch block (or blocks) that follows the try is checked.

THROW/THROWS KEYWORD

❖If a method does not handle a **checked exception**, the method must declare it using the **throws** keyword. The throws keyword appears at the end of a method's signature.

❖You can throw an exception, either a newly instantiated one or an exception that you just caught, by using the **throw** keyword.

```
public class className {

   public void deposit(double amount) throws
RemoteException {
      // Method implementation
      throw new RemoteException();
   }
   // Remainder of class definition
}
```

TRY WITH RESOURCES

❖**try-with-resources,** also referred as **automatic resource management,** is a new exception handling mechanism that was introduced in Java 7, which automatically closes the resources used within the try catch block.

❖To use this statement, you simply need to declare the required resources within the parenthesis, and the created resource will be closed automatically at the end of the block.

```
try(FileReader fr = new FileReader("file path")) {
   // use the resource
   }catch() {
      // body of catch
   }
}
```

The finally block

❖The finally block follows a try block or a catch block. A finally block of code always executes, irrespective of occurrence of an Exception.

❖Using a finally block allows you to run any cleanup-type statements that you want to execute, no matter what happens in the protected code.

```
try {
   // Protected code
}catch(ExceptionType1 e1) {
   // Catch block
}catch(ExceptionType2 e2) {
   // Catch block
}finally {
   // The finally block always executes.
}
```

Nested try catch blocks

❖In Java, try-catch blocks can be nested. i.e one try block can contain another try-catch block. The syntax for nesting try blocks is,

```
try     //Outer try block
{

    try    //Inner try block
    {
        //Some Statements
    }
    catch (Exception ex)    //Inner catch block
    {

    }
}
catch(Exception ex)     //Outer catch block
{

}
```

Nested try catch blocks

❖Nested try blocks are useful when different statements of try block throw different types of exceptions.

❖If the exception thrown by the inner try block can not be caught by it’s catch block, then this exception is propagated to outer try blocks. Any one of the outer catch block should handle this exception otherwise program will terminate abruptly.

Return Value

❖If finally block returns a value then try and catch blocks may or may not return a value.

❖If finally block does not return a value then both try and catch blocks must return a value.

❖If try-catch-finally blocks are returning a value according to above rules, then you should not keep any statements after finally block. Because they become unreachable and in Java, Unreachable code gives compile time error.

❖finally block overrides any return values from try and catch blocks.

❖finally block will be always executed even though try and catch blocks are returning the control.

Throwing an Exception

❖Throwable class is super class for all types of errors and exceptions.

❖An object to this Throwable class or it's sub classes can be created in two ways.

❖First one is using an argument of catch block. In this way, Throwable object or object to it's sub classes is implicitly created and thrown by java run time system.

❖Second one is using **new operator**. In this way, Throwable object or object to it's sub classes is explicitly created and thrown by the code.

METHOD OVERRIDING WITH THROWS

❖Rules need to follow when overriding a method with throws clause.

❖If super class method is not throwing any exceptions, then it can be overridden with any unchecked type of exceptions, but can not be overridden with checked type of exceptions.

❖If a super class method is throwing unchecked exception, then it can be overridden in the sub class with same exception or any other unchecked exceptions but can not be overridden with checked exceptions.

❖If super class method is throwing checked type of exception, then it can be overridden with same exception or with it's sub class exceptions i.e you can decrease the scope of the exception, but can not be overridden with it's super class exceptions i.e you can not increase the scope of the exception.

USER DEFINED EXCEPTIONS

❖We can define our own exception classes as per our requirements. These exceptions are called **user defined exceptions** in java OR **Customized exceptions**.

❖User defined exceptions must extend any one of the classes in the hierarchy of exceptions.

❖If the user defined class is extending Exception class, then checked exception is created.

❖If the user defined Exception class is extending RunTimeException Class, then unchecked exception is created.

STRINGS

❖String represents sequence of characters enclosed within the double quotes. “abc”, “JAVA”, “123”, “A” are some examples of strings.

❖ In many languages, strings are treated as character arrays.

❖But In java, strings are treated as objects.

❖String class is encapsulated under **java.lang** package.

❖To create and manipulate the strings, Java provides three classes.

1) java.lang.String
2) java.lang.StringBuffer
3) java.lang.StringBuilder (From JDK 1.5)

STRINGS

❖ All these three classes are members **of java.lang** package and they are final classes. That means you can't create subclasses to these three classes.

❖**java.lang.String** objects are **immutable** in java. That is, once you create String objects, you can't modify them. Whenever you try to modify the existing String object, a new String object is created with modifications. Existing object is not at all altered.

❖Where as **java.lang.StringBuffer** and **java.lang.StringBuilder** objects are mutable. That means, you can perform modifications to existing objects.

STRINGS

❖Only **String** and **StringBuffer** objects are **thread safe**. StringBuilder objects are not thread safe. So whenever you want immutable and thread safe string objects, use java.lang.String class and whenever you want mutable as well as thread safe string objects then use java.lang.StringBuffer class.

❖In all three classes, **toString()** method is overridden. Whenever you use reference variables of these three types, they will return contents of the objects not physical address of the objects.

❖**hashCode()** and **equals()** methods are overridden only in **java.lang.String** class but not in java.lang.StringBuffer and java.lang.StringBuilder classes.

STRINGS

❖There is no **reverse()** and **delete()** methods in String class. But, StringBuffer and StringBuilder have reverse() and delete() methods.

❖ In case of String class, you can create the objects without new operator. But in case of StringBuffer and StringBuilder class, you have to use new operator to create the objects.

STRING CONSTRUCTOR

❖If you want to create an empty string object, then use no-arg constructor of String class.

```
String s = new String();
```

❖Below constructor takes character array as an argument.

```
char[] chars = {'J', 'A', 'V', 'A'};      //Character
Array
String s = new String(chars);
```

❖Below constructor takes string as an argument.

```
String s = new String("JAVA");
```

STRING CONSTRUCTOR

❖This constructor takes **StringBuffer** type as an argument.

```
StringBuffer strBuff = new StringBuffer("abc");
String s = new String(strBuff);
```

```
StringBuilder strBldr = new StringBuilder("abc");
String s = new String(strBldr);
```

STRING LITERALS

❖In Java, all string literals like “java”, “abc”, “123” are treated as objects of java.lang.String class. That means, all methods of String class are also applicable to string literals.

❖You can also create the objects of String class without using new operator. This can be done by assigning a string literal to reference variable of type java.lang.String class.

```
String s1 = "abc";

String s2 = "abc"+"def";

String s3 = "123"+"A"+"B";
```

How Strings are stored

❖JVM divides the allocated memory to a Java program into two parts. one is **Stack** and another one is **heap**.

❖Stack is used for execution purpose and heap is used for storage purpose.

❖In that heap memory, JVM allocates some memory specially meant for string literals. This part of the heap memory is called **String Constant Pool**.

❖Each time you create a String literal, the JVM checks the string pool first. If the string literal already exists in the pool, a reference to the pool instance is returned. If string does not exist in the pool, a new string object is created, and is placed in the pool.

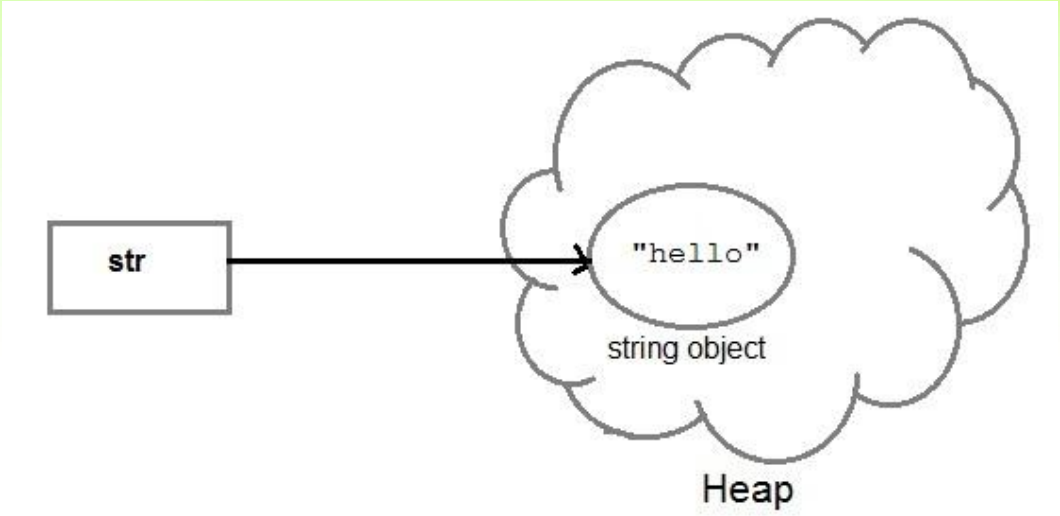
How Strings are Stored

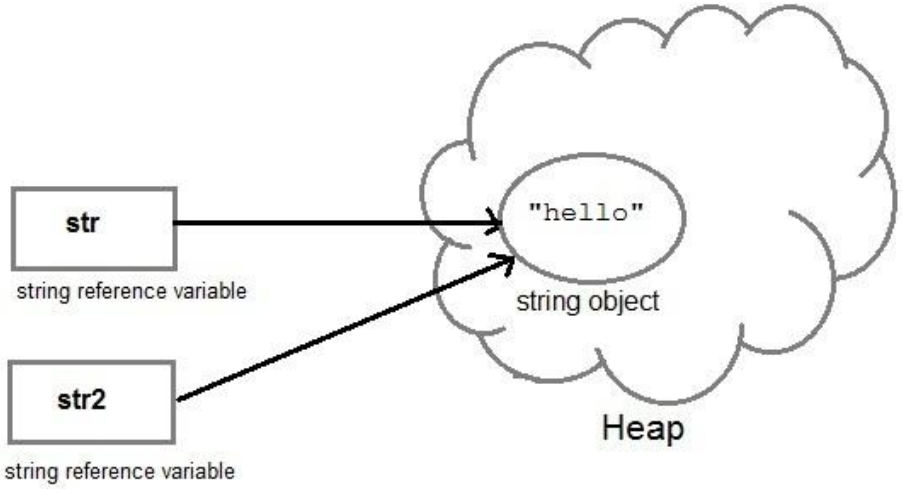
❖Whenever you create a string object using **string literal**, that object is stored in the **string constant pool** and whenever you create a string object using **new** keyword, such object is stored in the **heap memory**.

❖pool space is allocated to an object depending upon it’s content. There will be no two objects in the pool having the same content.

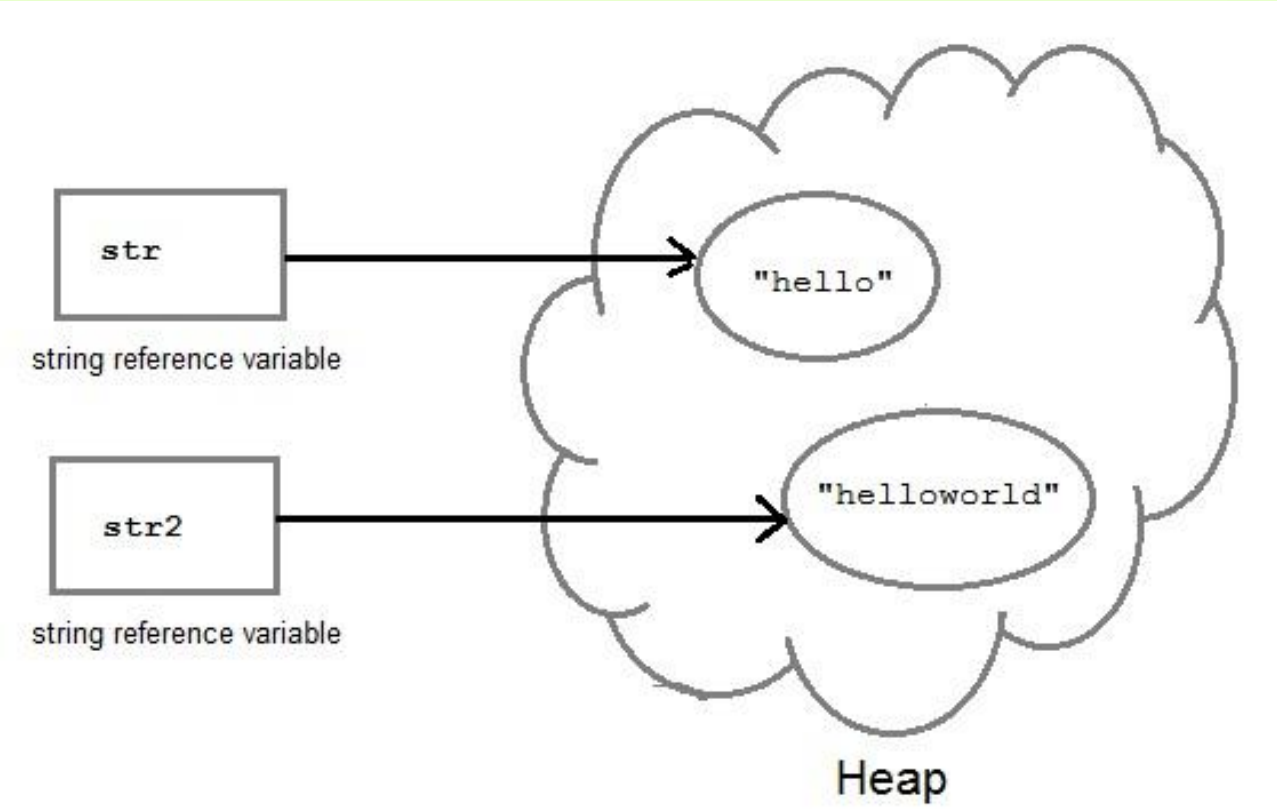
❖But, when you create string objects using new keyword, a new object is created whether the content is same or not.

HOW STRINGS ARE STORED





How Strings are stored



== VS. EQUALS

- **“==”** operator compares the two objects on their physical address. That means if two references are pointing to same object in the memory, then comparing those two references using “==” operator will return true.

- For example, if s1 and s2 are two references pointing to same object in the memory, then invoking s1 == s2 will return true.

- This type of comparison is called **“Shallow Comparison”.**

== VS. EQUALS

❖In **java.lang.String** class, **equals()** method is overridden to provide the comparison of two string objects based on their contents.

❖That means, any two string objects having same content will be equal according to equals() method.

❖For example, if s1 and s2 are two string objects having the same content, then invoking s1.equals(s2) will return true.

STRING CLASS FUNCTION

❖**charAt()** – function returns the character located at the specified index.

❖**equalsIgnoreCase()** – determines the equality of two Strings, ignoring thier case.

❖**length()** – returns the number of characters in a String.

❖**replace()** – replaces occurances of character with a specified new character.

❖**substring()** – returns a part of the string.

❖**toLowerCase()**

❖**toUpperCase()**

STRING CLASS FUNCTION

❖**indexOf()** – returns the index of first occurrence of a substring or a character. **indexOf()** method has four forms:

❖**int indexOf(String str):** It returns the index within this string of the first occurrence of the specified substring.

❖**int indexOf(int ch, int fromIndex):** It returns the index within this string of the first occurrence of the specified character, starting the search at the specified index.

❖**int indexOf(int ch):** It returns the index within this string of the first occurrence of the specified character.

❖**int indexOf(String str, int fromIndex):** It returns the index within this string of the first occurrence of the specified substring, starting at the specified index.

STRINGBUFFER

❖**StringBuffer** class is used to create a mutable string object i.e its state can be changed after it is created.

❖It represents growable and writable character sequence.

❖As we know that String objects are immutable, so if we do a lot of changes with String objects, we will end up with a lot of memory leak.

❖So **StringBuffer** class is used when we have to make lot of modifications to our string. It is also **thread safe** i.e multiple threads cannot access it simultaneously.

STRINGBUFFER

- StringBuffer defines 4 constructors. They are,

- StringBuffer ( )
- StringBuffer (int size)
- StringBuffer (String str)
- StringBuffer (charSequence [ ]ch)

- **StringBuffer()** creates an empty string buffer and reserves room for 16 characters.

- **stringBuffer(int size)** creates an empty string and takes an integer argument to set capacity of the buffer.

STRINGBUFFER METHODS

❖**append()** – This method will concatenate the string representation of any type of data to the end of the invoking StringBuffer object.

```
StringBuffer append(String str)

StringBuffer append(int n)

StringBuffer append(Object obj)
```

❖**insert()** – This method inserts one string into another. Here are few forms of insert() method.

```
StringBuffer insert(int index, String str)

StringBuffer insert(int index, int num)

StringBuffer insert(int index, Object obj)
```

STRINGBUFFER METHODS

❖**reverse()** – This method reverses the characters within a StringBuffer object.

```
StringBuffer str = new StringBuffer("Hello");
str.reverse();
System.out.println(str);
```

❖**replace()** – This method replaces the string from specified start index to the end index.

```
StringBuffer str = new StringBuffer("Hello World");
str.replace( 6, 11, "java");
System.out.println(str);
```

LOOPS

❖**While loop –** Repeats a statement or group of statements while a given condition is true. It tests the condition before executing the loop body.

```
while(Boolean_expression) {
   // Statements
}
```

❖**For loop** – is a repetition control structure that allows you to efficiently write a loop that needs to be executed a specific number of times.

```
for(initialization; Boolean_expression; update) {
   // Statements
}
```

LAB4

- Q1. Create an abstract class 'Parent' with a method 'message'.

- It has two subclasses each having a method with the same name 'message' that prints "This is first subclass" and "This is second subclass" respectively.

- Call the methods 'message' by creating an object for each subclass.

LAB4

❖Q2. We have to calculate the percentage of marks obtained in three subjects (each out of 100) by student A and in four subjects (each out of 100) by student B.

❖Create an abstract class 'Marks' with an abstract method 'getPercentage'. It is inherited by two other classes 'A' and 'B' each having a method with the same name which returns the percentage of the students.

❖The constructor of student A takes the marks in three subjects as its parameters and the marks in four subjects as its parameters for student B. Create an object for each of the two classes and print the percentage of marks for both the students.

LAB4

❖Q2. We have to calculate the percentage of marks obtained in three subjects (each out of 100) by student A and in four subjects (each out of 100) by student B.

❖Create an abstract class 'Marks' with an abstract method 'getPercentage'. It is inherited by two other classes 'A' and 'B' each having a method with the same name which returns the percentage of the students.

❖The constructor of student A takes the marks in three subjects as its parameters and the marks in four subjects as its parameters for student B. Create an object for each of the two classes and print the percentage of marks for both the students.

LAB4

❖Q3. Create an abstract class 'Animals' with two abstract methods 'cats' and 'dogs'.

❖Now create a class 'Cats' with a method 'cats' which prints "Cats meow" and a class 'Dogs' with a method 'dogs' which prints "Dogs bark", both inheriting the class 'Animals'.

❖Now create an object for each of the subclasses and call their respective methods.