Unit 3- String Handling and Exception Handling

Strings:

* Strings are objects that represent sequence of characters.
* Creating a String object is like creating a string that cannot be changed
* Strings are otherwise called as immutable strings
* The String class is defined in java.lang. Also they are ‘final’ which means it cannot be subclassed.

String Constructors:

1) String s=new String() // Creates an instance with no chars

2) String(char chars[])

(e.g.)
```
char chars[]={‘a’,’b’,’c’};
String s=new String(chars);
```

3) String(char chars[], int startindex, int numchars)

(e.g.)
```
char chars[]={‘a’,’b’,’c’,’d’,’e’,’f’};
String s=new String(chars, 2,3);
```

4) String(String strobj)

(e.g.)
```
class makestr
{
        public static void main(String args[])
        {
                char c[]={‘j’,’a’,’v’,’a’};
                String s1=new String(c);
                String s2=new String(s1);

                System.out.println(s1);
                System.out.println(s2);
}       }
```

5) Instead of creating an String instance explicitly, use string literal.
For each literal, java automatically constructs a String object.

```
String s=“abc”   // String literal
```

* String Objects are stored in a special memory area known as string constant pool.

* No new objects are created if it exists already in string constant pool.

Operations on Strings:

1) **String Length:** Returns the number of characters in a given string

Syntax: int length()

(e.g.)

```
String s=new String(“java”);
int m=s.length();

String s1=“abcde”;
System.out.println(“abcde”.length());
```

2) **String Concatenation:** use ‘+’ operator

3) **charAt():** Extract a single character from a String

Syntax: char charAt(int where)

(e.g.)

```
char ch;
ch=“abc”.charAt(1);
```

4) equals() and equalIgnoreCase():

* compares two strings for equality

Syntax: boolean equals(Object str)

// Here, str is the String object being compared with the invoking
// String object. It returns true if the strings contain the same characters
// in the same order, and false otherwise.

(e.g.)

```
String s1="Hello";
String s2="Hello";
String s3="Hi";

System.out.println(s1.equals(s2));
System.out.println(s1.equals(s3));
```

indexOf(): Searches for the first occurrence of a character or substring

Search for the first occurrence of a character:	int indexOf(char ch)
Search for the last occurrence of a character:	int lastIndexOf(char ch)
Search for the first occurrence of a substring:	int indexOf(String str)
Search for the last occurrence of a substring:	int lastindexOf(String str)

Specify the starting point for the search

```
int indexOf(char ch, int startindex)
int lastIndexOf(char ch, int startindex)

int indexOf(String str, int startindex)
int lastIndexOf(String str, int startindex)
```

Substring(): Extract a substring

syntax: String substring(int startindex)
String substring(int startindex, int endindex)

concat(): Concatenates two strings
syntax: String concat(String str)

replace(): Replaces all occurrences of one character in the invoking string with another character.
syntax: String replace(char original, char replacement)

trim(): Removes all the leading and trailing whitespaces
syntax: String trim()

toLowerCase() and toUpperCase():

syntax: String toLowerCase()
String toUpperCase()

compareTo():

syntax: int compareTo(String str)
int compareToIgnoreCase(String str)

// Here, str is the String being compared with the invoking String.
// The result of the comparison is returned and is interpreted as

Less than zero	-	The invoking string is less than str
Greater than zero	-	The invoking string is greater than str
zero	-	The two strings are equal

getChars():

* To extract more than one character at a time

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

(e.g.)

```
String s="This is a demo";
int start=5;
int end=8;

char buf[]=new  char[end-start];
s.getChars(start,end,buf,0);
System.out.println(buf);
```

toCharArray():

* Converting all the characters in a String object into a character array

```
char[] toCharArray()
```

<u>startsWith() and endsWith():</u>

* Determines whether a given String begins or ends with a specified string.

```
boolean startsWith(String str)
boolean endsWith(String str)
boolean startsWith(String str, int startindex)
```

<u>StringBuffer class:</u>

* The java.lang.StringBuffer class is mutable sequence of characters.

* StringBuffer creates strings of flexible length that can be modified in terms of both length and content. Represents growable and writeable character sequences.

* Insert characters and substrings in the middle of string or append another string to the end.

<u>StringBuffer Constructors:</u>

StringBuffer() : Reserves room for 16 characters without reallocation
StringBuffer(int size): size of buffer specified explicitly
StringBuffer(String str)
StringBuffer(CharSequence chars)

StringBuffer Methods:

1) public StringBuffer append(String s)
2) public StringBuffer reverse()
3) public delete(int start, int end)   public deleteCharAt(int loc)
4) public insert(int offset, String s)   public insert(int offset, char ch)
5) StringBuffer replace(int start, int end, String str)
6) int capacity()
7) char charAt(int index)
8) int indexOf(String str)
9) int indexOf(String str, int fromindex)
10) int lastIndexOf(String str)
11) int lastIndexOf(String str, int fromIndex)
12) int length()
13) Void setLength(int len)
14) void setCharAt(int index, char ch)
15) String substring(int start)
16) String substring(int start, int end)
17) Void getChars(int sourceStart, int sourceEnd, char target[], int targetStart)

Exception:

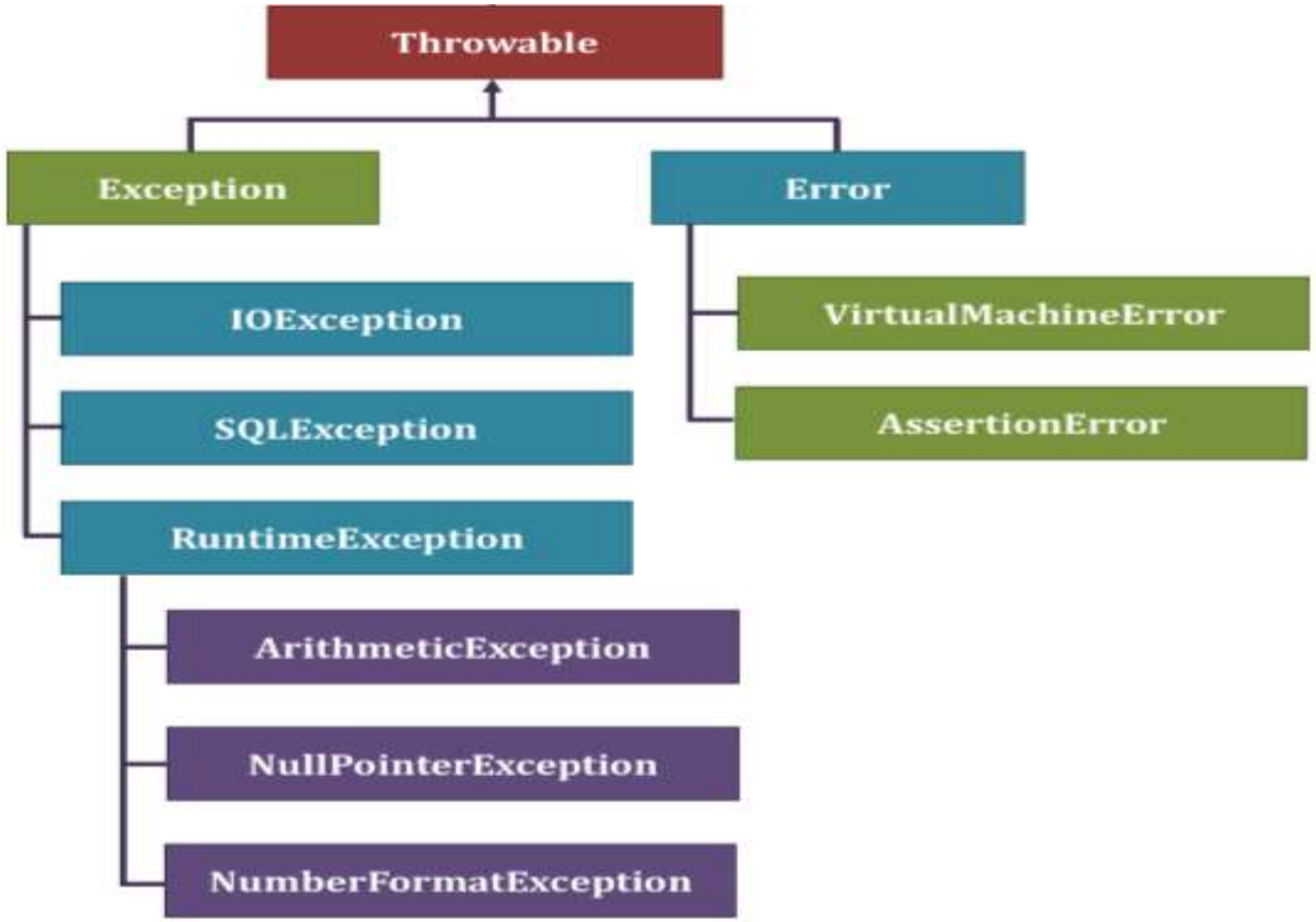
* An exception is a condition that is caused by run-time error in a program.

* When the Java interpreter encounters an error, 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.

* 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 the appropriate message for taking corrective actions.

The Exception Hierarchy



Exception	Description
ArithmeticException	Arithmetic error, such as divide-by-zero.
ArrayIndexOutOfBoundsException	Array index is out-of-bounds.
ArrayStoreException	Assignment to an array element of an incompatible type.
ClassCastException	Invalid cast.
IllegalArgumentException	Illegal argument used to invoke a method.
IllegalMonitorStateException	Illegal monitor operation, such as waiting on an unlocked thread.
IllegalStateException	Environment or application is in incorrect state.
IllegalThreadStateException	Requested operation not compatible with current thread state.
IndexOutOfBoundsException	Some type of index is out-of-bounds.
NegativeArraySizeException	Array created with a negative size.
NullPointerException	Invalid use of a null reference.
NumberFormatException	Invalid conversion of a string to a numeric format.
SecurityException	Attempt to violate security.
StringIndexOutOfBounds	Attempt to index outside the bounds of a string.
UnsupportedOperationException	An unsupported operation was encountered.

Exception Handling:

* An exception is an event that occurs during the execution of a program, which disrupts the normal flow of the program's instructions.

* Java provides exception handling mechanism which can be used to trap this exception and run programs smoothly after catching the exception.

* When an error occurs within a method, the method creates an object and hands it off to the runtime system. (exception object)

* Creating an exception object and handing it to the runtime system is called throwing an exception.

Tasks of Exception Handling:

* Find the exception (Hit the exception)
* Inform that error has occurred (Throw the exception)
* Receive the error information (Catch the exception)
* Take corrective results (Handle the exception)

Common Java exceptions:

* ArithmeticException
* ArrayIndexOutOfBoundsException
* FileNotFoundException
* IOException
* NullPointerException
* OutOfMemoryException
* NumberFormatException

Using try and catch:

* To handle a run-time error, enclose the code that has to be monitored inside a try block.

* Immediately following the try block, include a catch clause that specifies the exception type that has to be catched.

* Once the catch statement has executed, program control continues with the next line in the program following the entire try/catch mechanism.

* A try and catch statement form a unit. The scope of the catch clause is restricted to those statements specified by the immediately preceding try statement.

* A catch statement cannot catch an exception thrown by another try statement.

(e.g)

```
try
        {
                int num;
                num=Integer.parseInt("ABC123");
                System.out.println("Control won't reach here");
        }
        catch(NumberFormatException e)
        {
                System.out.println("Exception thrown");
                System.out.println(e.getMessage()); // display internally generated msg
                e.printStackTrace(); // indicates types and exception occurred and
                                     // place and line number.
        }
        System.out.println("out of try-catch block");
```

(e.g)

```
int a=10,b=5,c=5,x,y;
        try
        {
                x=a/(b-c);
        }
        catch(ArithmeticException e)
        {
                System.out.println(“Division by zero”);
        }
        y= a/(b+c);
        System.out.println(“y=“+y);
```

Multiple catch statements:

* More than one exception could be raised by a single piece of code.

* Two or more catch clauses can be specified, each catching a different type of exception.

* When an exception is thrown, each catch statement is inspected in order, and the first one whose type matches that of the exception is executed.

* After one catch statement executes, the others are bypassed and execution continues after the try/catch block.

* When using multiple catch statements, it is important to remember that exception sub-classes must come before any of their subclasses.

* A subclass would never be reached if it comes after its super class. In Java, unreachable code is an error.

(e.g)

```
try
        {
                int a=args.length;
                System.out.println(“a=“+a);
                int b=42/a;
                int c[]={1};
                c[42]=99;
        }
        catch(ArithmeticException e)
        {
                System.out.println(“Divide by zero:”+e);
        }
        catch(ArrayIndexOutOfBoundsException e)
        {
                System.out.println(“Array Index:”+e);
        }
         System.out.println(“After try/catch blocks”);
```

(e.g)

```
int a[]={5,10};
int b=5;

try
{
        int x=a[2]/b-a[1];
}
catch(ArithmeticException e)
{
        System.out.println(“Division by zero”);
}
catch(ArrayIndexOutOfBoundsException e)
{
        System.out.println(“Array Index error”);
}
catch(ArrayStoreException e)
{
        System.out.println(“Wrong data type”);
}
int y=a[1]/a[0];
System.out.println(“y=“+y);
```

Throw statement:

* An exception can be thrown explicitly using throw keyword.

* The flow of execution stops immediately after the throw statement; any subsequent statements are not executed.

* The nearest enclosing try block is inspected to see if it has a catch statement that matches the type of exception. If it does find a match, control is transferred to that statement. If not, the next enclosing try statement is inspected, and so on. If no matching catch is found, then the default exception handler halts the program and prints the stack trace.

```
throw throwableinstance;
```

* There are two ways of obtaining a Throwable object:
    -- using a parameter in a catch clause
    -- Creating with the new operator.

(e.g.)

```
class throwdemo
{       static void demoproc()
        {
                try
                {
                        throw new NullPointerException(“demo”);
                }
                catch(NullPointerException e)
                {
                        System.out.println(“Caught inside demoproc”);
                        throw e;
        }       }
        public static void main(String[] args)
        {       try
                {
                        demoproc();
                }
                catch(NullPointerException e)
                {
                        System.out.println(“Recaught:”+e);
        }       }
```

* ‘new’ is used to construct an instance of NullPointerException.

* Many of Java’s built in run-time exceptions have atleast two constructors
  - --One with no parameter.
  - -- string parameter

throws clause:

* If a method is capable of raising an exception that it does not handle, it must specify that the exception has to be handled by the calling method.

* lists the types of exceptions that a method might throw.

Finally:

* finally creates a block of code that will be executed after a try/catch block has completed and before the code following the try/catch block.

* The finally block will execute whether or not the exception is thrown.

* The finally clause is optional.

* Each try statement requires at least one catch or finally clause.

(e.g.)
```
class finallydemo
    {   static void procA
        {       try     {
                System.out.println(“inside procA”);
                throw new Exception(“demo”);
                }
                finally{
                        System.out.println(“procA finally”);
                }
    }   }
```

Creating a custom exception:

* Exception classes can be created by extending the Exception class.

* The extended class contains constructors, data members and methods like any other class.

* The throw and throws keywords are used while implementing the user-defined exceptions.

Creating a custom exception:

```
class MyException extends Exception
        {
                private int details;
                MyException(int a)
                {
                        detail=a;
                }
                public String toString()
                {               return “MyException[“+detail+”];        }
        }
        class exceptiondemo
        {       static void compute(int a) throws MyException
                {
                        if (a>20) throw new MyException(a);
                }
 public static void main(String[] args) {
    int k=30;
    try{
      compute(k);
    }
    catch (MyException my){
      System.out.println(my.toString());
    }
  }}
```