Arrays

An array is a collection of similar data types. Array is a container object that holds values of homogeneous type. It is also known as static data structure because size of an array must be specified at the time of its declaration. Array starts from **zero** index and goes to **n-1** where n is length of the array. Array can be single dimensional or multidimensional in Java. In Java Since arrays are objects in Java, we can find their size using member **length**.

Features of Array

- It is always indexed. Index begins from 0.
- It is a collection of similar data types.
- It occupies a contiguous memory location.
- It allows to access elements randomly.
- all arrays are dynamically allocated.

Single Dimensional Array

Single dimensional array use single index to store elements.

Array Declaration

Syntax :

datatype[] arrayName;

or

datatype arrayName[];

Java allows declaring array by using both declaration syntax, both are valid. The arrayName can be any valid array name and datatype can be any like: int, float, byte etc.

Example : int[ ] arr; char[ ] arr; short[ ] arr; long[ ] arr;

Initialization of Array

Initialization is a process of allocating memory to an array. At the time of initialization, the size of array to reserve memory area is specified.

Initialization Syntax : **arrayName = new datatype[size]**

new operator is used to initialize an array. The arrayName is the name of array, **new** is a keyword used to allocate memory and size is length of array.

Both declaration and initialization can be combined into a single statement.

Datatype[] arrayName = new datatype[size]

Example 1: Simple Array Declaration

```
class ArrayDemo
{
   public static void main (String[] args)
   {
        int[] arr;       // declares an Array of integers.
        arr = new int[5];  // allocating memory for 5 integers.
        arr[0] = 10;  // initialize the first elements of the array
        arr[1] = 20; // initialize the second elements of the array
        arr[2] = 30;
        arr[3] = 40;
        arr[4] = 50;
        // accessing the elements of the specified array
        for (int i = 0; i < arr.length; i++)
                System.out.println("Element at index "+i+" : "+ arr[i]);
   }
}
```

Output

```
Element at index 0 : 10
Element at index 1 : 20
Element at index 2 : 30
Element at index 3 : 40
Element at index 4 : 50
```

Multi-Dimensional Array

A multi-dimensional array is very much similar to a single dimensional array. It can have multiple rows and multiple columns unlike single dimensional array, which can have only one row index. It represents data into tabular form in which data is stored into row and columns.

Multi-Dimensional Array Declaration

datatype[ ][ ] arrayName;

Initialization of Array

datatype[ ][ ] arrayName = new int[no_of_rows][no_of_columns];

Example 2: Two dimensional Array

```
class Demo
{
                public static void main(String[] args)
                {
                     int arr[ ][ ] = {{1,2,3,4,5},{6,7,8,9,10},{11,12,13,14,15}};
                     for(int i=0;i<3;i++)
                     {
                          for (int j = 0; j < 5; j++)
                          {
                                   System.out.print(arr[i][j]+" ");
                          }
                          System.out.println();
                     }
                }
}
```

Output

```
1 2 3 4 5
6 7 8 9 10
11 12 13 14 15
```

Enhanced For Loop - Java For-each Loop

The Java **for-each** loop or enhanced for loop is an alternative approach to traverse the array or collection in Java. It is mainly used to traverse the array or collection elements. The advantage of for-each loop is that it eliminates the possibility of bugs and makes the code more readable. It is known as for-each loop because it traverses each element one by one.

The drawback of the enhanced for loop is that it cannot traverse the elements in reverse order. Also there is no option to skip any element because it does not work on an index basis. Moreover, the odd or even element traversal is not possible.

Syntax

The syntax of Java for-each loop consists of data_type with the variable followed by a colon (:), then array or collection.

```
for(data_type variable : array | collection)
{
//body of for-each loop
}
```

The Java for-each loop traverses the array or collection until the last element. For each element, it stores the element in the variable and executes the body of the for-each loop.

Example 3: for-each loop

```
class ForEachExample1
{
	public static void main(String args[])
	{
		int arr[]={12,13,14,44};  //declaring an array
		//traversing the array with for-each loop
		for(int i:arr)
		{
			System.out.println(i);
		}
	}
}
```

Output

```
12
12
14
44
```

Jagged Array

Jagged array is an array that has different numbers of columns elements. In java, a jagged array means to have a multi-dimensional array with ***uneven size of columns*** in it. Jagged array initialization is different little different. We have to set columns size for each row independently.

Example:
```
int[ ][ ] arr = new int[3][ ];
arr[0] = new int[3];
arr[1] = new int[4];
arr[2] = new int[5];
```

Example 4: jagged array

```
class Demo
{
	public static void main(String[] args)
	{
		int arr[ ][ ] = {{1,2,3},{4,5},{6,7,8,9}};
		for(int i=0;i<3;i++)
		{
			for (int j = 0; j < arr[i].length; j++)
			{
				System.out.print(arr[i][j]+" ");
			}
			System.out.println();

		}
	}
}
```

Output

```
1 2 3
4 5
6 7 8 9
```

Here, we can see number of rows are 3 and columns are different for each row. This type of array is called jagged array.

1	2	3	
4	5		
6	7	8	9

← This is the structure of array in example 4

String Handling

String is an object that represents sequence of characters. In Java, String is represented by **String** class which is located into **java.lang** package. It is probably the most commonly used class in java library. In java, every string that we create is actually an object of type String. One important thing to notice about string object is that string objects are immutable that means once a string object is created it cannot be changed.

String class -Creating a String object

String can be created in number of ways. They are

1) Using a String literal

String literal is a simple string enclosed in double quotes " ". A string literal is treated as a String object.

Example 5: creating strings using string literal

```
public class Demo
{
        public static void main(String[] args)
        {
                String s1 = "Hello Java";
                System.out.println(s1);
        }
}
```

Output

```
Hello Java
```

2) Using new Keyword

A new string object can be created by using new operator that allocates memory for the object.

Example 6: creating strings using new operator

```
public class Demo
{
        public static void main(String[] args)
        {
                String s1 = new String("Hello Java");
                System.out.println(s1);
        }
}
```

Output

```
Hello Java
```

Java String class methods

No.	Method	Description
1	char charAt(int index)	returns char value for the particular index
2	int length()	returns string length
3	String substring(int beginIndex)	returns substring for given begin index.
4	String substring(int beginIndex, int endIndex)	returns substring for given begin index and end index.
5	boolean equals(Object another)	checks the equality of string with the given object.
6	String concat(String str)	concatenates the specified string.
7	String replace(char old, char new)	replaces all occurrences of the specified char value.
8	String replace(CharSequence old, CharSequence new)	replaces all occurrences of the specified CharSequence.
9	static String equalsIgnoreCase(String another)	compares another string. It doesn't check case.
10	int indexOf(int ch)	returns the specified char value index.
11	int indexOf(int ch, int fromIndex)	returns the specified char value index starting with given index.
12	int indexOf(String substring)	returns the specified substring index.
13	int indexOf(String substring, int fromIndex)	returns the specified substring index starting with given index.
14	String toLowerCase()	returns a string in lowercase.
15	String toUpperCase()	returns a string in uppercase.
16	String trim()	removes beginning and ending spaces of this string.

Example 7: String programs demo

```
public class StrDemo1
{
        public static void main(String[] args)
        {
                String s = "Hell";
                String s1 = "Hello";
                String s2 = "Hello";
                boolean b = s1.equals(s2);    //true
                System.out.println(b);
                b =       s.equals(s1) ;   //false
                System.out.println(b);
        }
}
```

Output

```
true
false
```

Example 8: String programs demo

```
public class StrDemo2
{
        public static void main(String[] args)
        {
                String str = "studytonight";
                System.out.println(str.charAt(2));
                String str1 = "java";
                System.out.println(str1.equalsIgnoreCase("JAVA"));
                String str2="StudyTonight";
                System.out.println(str2.indexOf('u'));
                System.out.println(str2.indexOf('t', 3));
                String subString="Ton";
                System.out.println(str2.indexOf(subString));
                System.out.println(str2.indexOf(subString,7));
        }
}
```

Output

```
u
true
2
11
5
-1
```

Example 9: String programs demo

```
public class StrDemo3
{
        public static void main(String[] args)
        {
                String str = "Change me";
                System.out.println(str.replace('m','M'));
                String str1 = "ABCDEF";
                System.out.println(str1.toLowerCase());
                String str2 = "   hello  ";
                System.out.println(str2.trim());
        }
}
```

Output

```
Change Me
abcdef
hello
```

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 preallocated than are actually needed, to allow room for growth.

Method	**Description**
append(String s)	is used to append the specified string with this string.
insert(int offset, String s)	is used to insert the specified string with this string at the specified position.
replace(int startIndex, int endIndex, String str)	is used to replace the string from specified startIndex and endIndex.
delete(int startIndex, int endIndex)	is used to delete the string from specified startIndex and endIndex.

reverse()	is used to reverse the string.
charAt(int index)	is used to return the character at the specified position.
length()	is used to return the length of the string i.e. total number of characters.
substring(int beginIndex)	is used to return the substring from the specified beginIndex.
substring(int beginIndex, int endIndex)	is used to return the substring from the specified beginIndex and endIndex.

Example 10: StringBuffer demo

```
class stringManipulation
{
        public static void main(String args[])
        {
                // str is an object of class StringBuffer
                StringBuffer str = new StringBuffer("Object language");
                System.out.println("Original String : "+str);
                System.out.println("Length of String is : "+ str.length());
                for(int i = 0; i < str.length(); i++)
                {
                        int p = i + 1;
                        // accessing characters in a string
                        System.out.println("Character at position : "+p+" is :"+str.charAt(i));
                }
                String aString = new String(str.toString());
                int pos = aString.indexOf("language");
                // inserting a string in the middle
                str.insert(pos, "Oriented");
                System.out.println("Modified string : " + str);
                // Modifying character at position 6
                str.setCharAt(6, '`');
                System.out.println("String now : " + str);
                // Appending a string at the end
                str.append("improves security : ");
                System.out.println("Append string : "+str);
            }
    }
```

Output

```
Original String : Object language
Length of String is : 15
Character at position : 1 is : O
Character at position : 2 is : b
Character at position : 3 is : j
Character at position : 4 is : e
Character at position : 5 is : c
Character at position : 6 is : t
Character at position : 7 is :
Character at position : 8 is : l
Character at position : 9 is : a
Character at position : 10 is : n
Character at position : 11 is : g
Character at position : 12 is : u
Character at position : 13 is : a
Character at position : 14 is : g
Character at position : 15 is : e
Modified string : Object Orientedlanguage
String now : Object`Orientedlanguage
Append string : Object`Orientedlanguageimproves security :
```

Packages

Package is a collection of related classes. Java uses package to group related classes, interfaces and sub-packages. A package can be assumed as a folder or a directory that is used to store similar files. Packages are used for

- Preventing naming conflicts. For example there can be two classes with same name **Employee** in two packages, ***college.staff.cse.Employee*** and ***college.staff.ee.Employee***
- Making searching/locating and usage of classes, interfaces, enumerations and annotations easier
- Providing controlled access: protected and default have package level access control. A protected member is accessible by classes in the same package and its subclasses. A default member (without any access specifier) is accessible by classes in the same package only.
- Packages can be considered as data encapsulation (or data-hiding).

Package can be built-in and user-defined. Java provides rich set of built-in packages in form of API that stores related classes and sub-packages.

Java API packages

The Java API is a library of prewritten classes that are free to use, included in the Java Development Environment. The library is divided into packages and classes. Either a single class (along with its methods and attributes) or a whole package that contain all the classes that belong to the specified package can be imported.

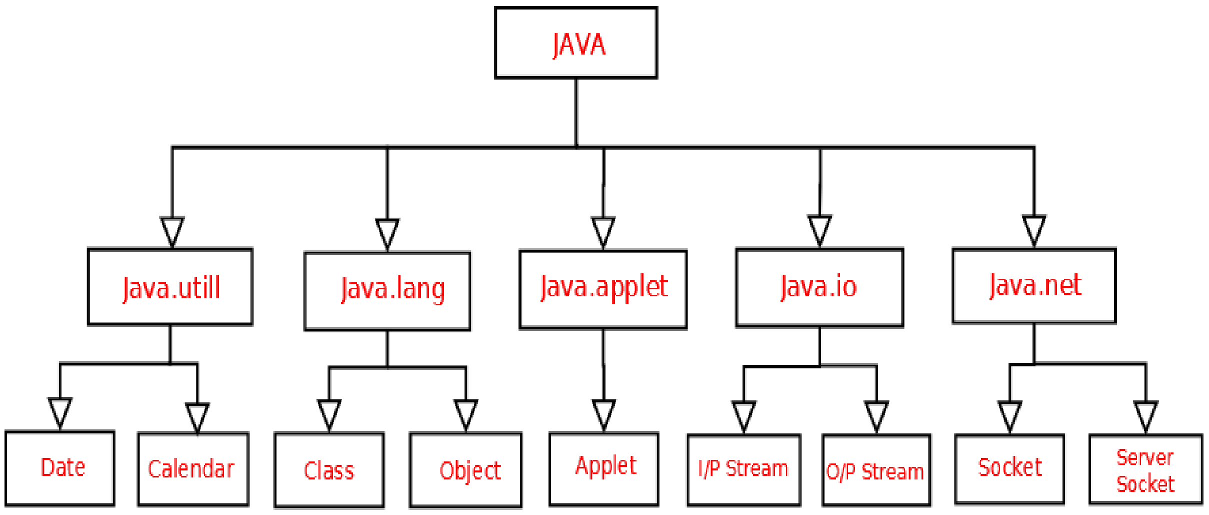

Figure 1: Hierarchy of Java API packages

Some of the commonly used built-in API packages are:

- **java.lang:** Contains language support classes. It contains classes for primitive types, strings, math functions, threads, and exceptions. This package is automatically imported.
- **java.io:** Contains classed for supporting input / output operations. It has stream classes for Input/Output.
- **java.util**: Contains utility classes which implement data structures like Linked List, Dictionary and support. It contains classes such as vectors, hash tables, dates, Calendars, etc.
- **java.applet**: Contains classes for creating implementing applets
- **java.awt**: Contain classes for implementing the components for Graphical User Interface – windows, buttons, menus, etc.

- **java.net:** Contain classes for supporting networking operations.
- **javax.swing** : package provides classes for java swing API such as JButton, JTextField, JTextArea, JRadioButton, JCheckbox, JMenu, JColorChooser etc.

User defined packages

User defined packages are Java packages created by user to categorize their project's classes and interface are known as user-defined packages.

Creating packages

Creating a package in java is quite easy, simply include a package command followed by name of the package as the first statement in java source file.

```
package mypack;
public class employee
{
   String empId;
   String name;
}
```

The above statement will create a package with name **mypack** in the project directory.

Using packages

To include java package (built-in or user defined) into a class, **import** keyword is used. It is used to access package and its classes into the java program.

Use **import** to access built-in and user-defined packages into java source file so that the class can refer to a class that is in another package by directly using its name.

There are 3 different ways to refer to any class that is present in a different package:

- without import the package
- import package with specified class
- import package with all classes

Accessing package without import keyword

If you use fully qualified name to import any class into your program, then only that particular class of the package will be accessible in your program, other classes in the same package will not be accessible. For this approach, there is no need to use the import statement. But you will have to use the fully qualified name every time you are accessing the class or the interface. This is generally used when two packages have classes with same names.

Example 11: Package

```
package pack;
public class A
{
        public void msg()
        {
                System.out.println("Hello");
        }
}

class B
{
        public static void main(String args[])
        {
                pack.A obj = new pack.A();  //using fully qualified name
                obj.msg();
        }
}
```

Output

```
Hello
```

Import the Specific Class

Package can have many classes but sometimes we want to access only specific class in our program in that case, Java allows us to specify class name along with package name. If we use import **packagename.classname** statement then only the class with name classname in the package will be available for use.

Example 12: Package

```
package pack;
public class Demo
```

```
{
        public void msg()
        {
                System.out.println("Hello");
        }
}

import pack.Demo;
class Test
{
        public static void main(String args[])
        {
                Demo obj = new Demo();
                obj.msg();
        }
}
```

Output

```
Hello
```

Import all classes of the package

If we use **packagename.*** statement, then all the classes and interfaces of this package will be accessible but the classes and interface inside the subpackages will not be available for use. The import keyword is used to make the classes of another package accessible to the current package.

Example 13: Package

```
package learnjava;
public class First
{
        public void msg()
        {
                System.out.println("Hello");
        }
}

package Java;
import learnjava.*;
class Second
{
        public static void main(String args[])
        {
                First obj = new First();
                obj.msg();
        }
}
```

<u>Output</u>

```
Hello
```

<u>Exception Handling Techniques</u>

Exception Handling is a mechanism to handle exception at runtime. **Exception is a condition that occurs during program execution and lead to program termination abnormally.** There can be several reasons that can lead to exceptions, including programmer error, hardware failures, files that need to be opened cannot be found, resource exhaustion etc.

Suppose we run a program to read data from a file and if the file is not available then the program will stop execution and terminate the program by reporting the exception message. The problem with the exception is, it terminates the program and skip rest of the execution that means if a program have 100 lines of code and at line 10 an exception occur then program will terminate immediately by skipping execution of rest 90 lines of code.

To handle this problem, we use **exception handling that avoid program termination and continue the execution by skipping exception code.**

Java exception handling provides a meaningful message to the user about the issue rather than a system generated message, which may not be understandable to a user.

A Java Exception is an object that describes the exception that occurs in a program. **When an exceptional event occurs in java, an exception is said to be thrown**. **The code that's responsible for doing something about the exception is called an exception handler.**

Java provides controls to handle exception in the program. These controls are listed below.

- **try : It is used to enclose the suspected code.**
- **catch: It acts as exception handler.**
- **finally: It is used to execute necessary code.**
- **throw: It throws the exception explicitly.**
- **throws: It informs for the possible exception.**

Types of Exceptions

In Java, exceptions broadly can be categorised into checked exception and unchecked exception based on the nature of exception.

Checked Exception: The exception that can be predicted by the JVM at the compile time. For example: File that need to be opened is not found, SQLException etc. These type of exceptions must be checked at compile time.

Unchecked Exception : Unchecked exceptions are the class that extends RuntimeException class. Unchecked exception are ignored at compile time and checked at runtime. For example : ArithmeticException, NullPointerException, Array Index out of Bound exception. Unchecked exceptions are checked at runtime.

Java Exception class Hierarchy

All exception types are subclasses of class **Throwable**, which is at the top of exception class hierarchy. 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.

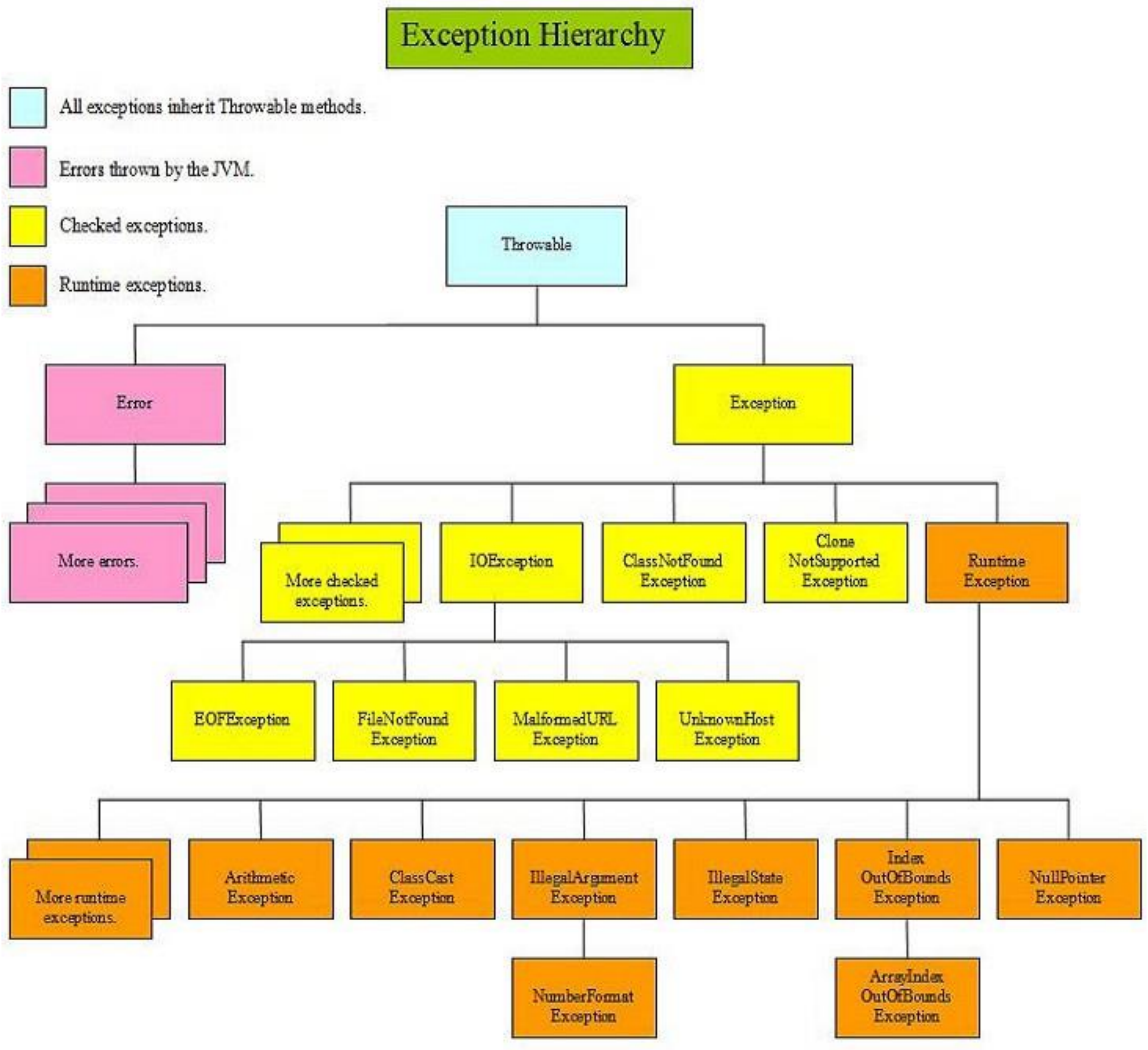

Figure 2: Exception class hierarchy

try and catch

try and catch both are Java keywords and used for exception handling. The try block is used to enclose the suspected code. Suspected code is a code that may raise an exception during program execution.

For example, if a code raise arithmetic exception due to divide by zero then we can wrap that code into the try block.

```
try
{
        int a = 10;
        int b = 0
        int c = a/b; // exception
}
```

The catch block also known as handler is used to handle the exception. It handles the exception thrown by the code enclosed into the try block. Try block must provide a catch handler or a finally block. The catch block must be used after the try block only. We can also use multiple catch block with a single try block.

```
try
{
        int a = 10;
        int b = 0
        int c = a/b; // exception
}
catch(ArithmeticException e)
{
        System.out.println(e);
}
```

To declare try catch block, a general syntax is given below.

```
try
{
  // suspected code
}
catch(ExceptionClass ec)
{
}
```

Exception handling is done by transferring the execution of a program to an appropriate exception handler (catch block) when exception occurs.

Example 14:Exception Handling

```
class Excp
{
        public static void main(String args[])
        {
                int a,b,c;
                try
                {
                        a = 0;
                        b = 10;
                        c = b/a;
                        System.out.println("This line will not be executed");
                }
                catch(ArithmeticException e)
                {
                        System.out.println("Divided by zero");
                }
                System.out.println("After exception is handled");
        }
}
```

Output

```
Divided by zero
After exception is handled
```

In the above program, an exception will thrown by this program as we are trying to divide a number by zero inside try block. The program control is transferred outside try block. Thus the line "This line will not be executed" is never parsed by the compiler. The exception thrown is handled in catch block. Once the exception is handled, the program control is continue with the next line in the program i.e after catch block. Thus the line "After exception is handled" is printed.

Multiple catch blocks

A try block can be followed by multiple catch blocks. That is any number of catch blocks can be included after a single try block. If an exception occurs in the guarded code (try block) the exception is passed to the first catch block in the list. If the exception type matches with the first catch block it gets caught, if not the exception is passed down to the next catch block. This continue until the exception is caught or falls through all catches.

To declare the multiple catch handler, we can use the following syntax.

```
try
{
  // suspected code
}
catch(Exception1 e)
{
   // handler code
}
```

```
catch(Exception2 e)
{
  // handler code
}
```

Example 15:Exception Handling- multiple catch blocks

```
public class CatchDemo2
{
	public static void main(String[] args)
	{
		try
		{
			int a[]=new int[10];
			System.out.println(a[20]);
		}
		catch(ArithmeticException e)
		{
			System.out.println("Arithmetic Exception  ");
		}
		catch(ArrayIndexOutOfBoundsException e)
		{
			System.out.println("ArrayIndexOutOfBounds Exception ");
		}
		catch(Exception e)
		{
			System.out.println(e);
		}
	}
}
```

Output

```
ArrayIndexOutOfBounds Exception
```

throw, throws and finally Keyword

throw, throws and finally are the keywords in Java that are used in exception handling. The throw keyword is used to throw an exception and throws is used to declare the list of possible exceptions with the method signature. Whereas finally block is used to execute essential code, especially to release the occupied resources.

The **throw** keyword is used to throw an exception explicitly. Only object of Throwable class or its sub classes can be thrown. Program execution stops on encountering throw statement, and the closest catch statement is checked for matching type of exception.

Syntax

```
throw ThrowableInstance
```

Example 16:Exception Handling- throwing an exception

```
class Test
{
        static void avg()
        {
                try
                {
                        throw new ArithmeticException("demo");
                }
                catch(ArithmeticException e)
                {
                        System.out.println("Exception caught");
                }
        }
        public static void main(String args[])
        {
                avg();
        }
}
```

Output

```
Exception caught
```

In the above example the avg() method throw an instance of ArithmeticException, which is successfully handled using the catch statement and thus, the program prints the output "Exception caught".

The **throws** keyword is used to declare the list of exception that a method may throw during execution of program. Any method that is capable of causing exceptions must list all the exceptions possible during its execution, so that anyone calling that method gets a prior knowledge about which exceptions are to be handled. A method can do so by using the throws keyword.

Syntax:

```
type method_name(parameter_list) throws exception_list
{
  // definition of method
}
```

Example 17:Exception Handling- throws keyword

```
class Test
{
        static void check() throws ArithmeticException
        {
                System.out.println("Inside check function");
                throw new ArithmeticException("demo");
        }
        public static void main(String args[])
        {
                try
                {
                        check();
                }
                catch(ArithmeticException e)
                {
                        System.out.println("caught" + e);
                }
        }
}
```

Output

```
Inside check function
caughtjava.lang.ArithmeticException: demo
```

A **finally** keyword is used to create a block of code that follows a try block. A finally block of code is always executed whether an exception has occurred or not. Using a finally block, programmer can run any cleanup type statements to execute, no matter what happens in the protected code. A finally block appears at the end of catch block.

Example 18:Exception Handling- finally keyword

```
class ExceptionTest
{
        public static void main(String[] args)
        {
                int a[] = new int[2];
                System.out.println("out of try");
                try
                {
                        System.out.println("Access invalid element"+ a[3]);
                /* the above statement will throw ArrayIndexOutOfBoundException */
                }
                finally
                {
                        System.out.println("finally is always executed.");
                }
        }
}
```

Output

```
Out of try
finally is always executed.
Exception in thread main java. Lang. exception array Index out of bound exception.
```

Example 19:Exception Handling- finally keyword

```
class Demo
{
        public static void main(String[] args)
        {
                int a[] = new int[2];
                try
                {
                        System.out.println("Access invalid element"+ a[3]);
                        /* the above statement will throw ArrayIndexOutOfBoundException */
                }
                catch(ArrayIndexOutOfBoundsException e)
                {
                    System.out.println("Exception caught");
                }
                finally
                {
                    System.out.println("finally is always executed.");
                }
        }
}
```

Output

```
Exception caught
finally is always executed.
```

Multithreading

Multithreading is a concept of running multiple threads simultaneously. Thread is a lightweight unit of a process that executes in multithreading environment.

A program can be divided into a number of small processes. Each small process can be addressed as a single thread (a lightweight process). You can think of a lightweight process as a virtual CPU that executes code or system calls.

Multithreaded programs contain two or more threads that can run concurrently and each thread defines a separate path of execution. This means that a single program can perform two or more tasks simultaneously. For example, one thread is writing content on a file at the same time another thread is performing spelling check.

In Java, the word thread means two different things- An instance of Thread class or a thread of execution. An instance of Thread class is just an object, like any other object in java. But a thread of execution means an individual "lightweight" process that has its own call stack. In java each thread has its own call stack.

Advantage of Multithreading

Multithreading reduces the CPU idle time that increase overall performance of the system. Since thread is lightweight process then it takes less memory and perform context switching as well that helps to share the memory and reduce time of switching between threads.

Multitasking is a process of performing multiple tasks simultaneously. Multitasking can be achieved either by using multiprocessing or multithreading. Multitasking by using multiprocessing involves multiple processes to execute multiple tasks simultaneously whereas Multithreading involves multiple threads to execute multiple tasks.

Thread has many advantages over the process to perform multitasking. Process is heavy weight, takes more memory and occupy CPU for longer time that may lead to performance issue with the system. To overcome these issue process is broken into small unit of independent sub-process. These sub-process are called threads that can perform independent task efficiently. So nowadays computer systems prefer to use thread over the process and use multithreading to perform multitasking.

Thread life cycle.

The life cycle of the thread in java is controlled by JVM. A thread in Java at any point of time exists in any one of the following states.

- Newborn
- Runnable
- Running
- Blocked/Waiting state
- Terminated/ Dead

Newborn state: When a new thread is created, it is in the newborn state. The thread has not yet started to run when thread is in this state. When a thread lies in the new state, it’s code is yet to be run and hasn’t started to execute.

Runnable State: A thread that is ready to run is moved to runnable state. In this state, a thread might actually be running or it might be ready run at any instant of time. It is the responsibility of the thread scheduler to give the thread time to run.

A multi-threaded program allocates a fixed amount of time to each individual thread. Each and every thread runs for a short while and then pauses and relinquishes the CPU to another thread, so that other threads can get a chance to run. When this happens, all such threads that are ready to run, waiting for the CPU and the currently running thread lies in runnable state. The process of assigning time to threads is known as time-slicing.

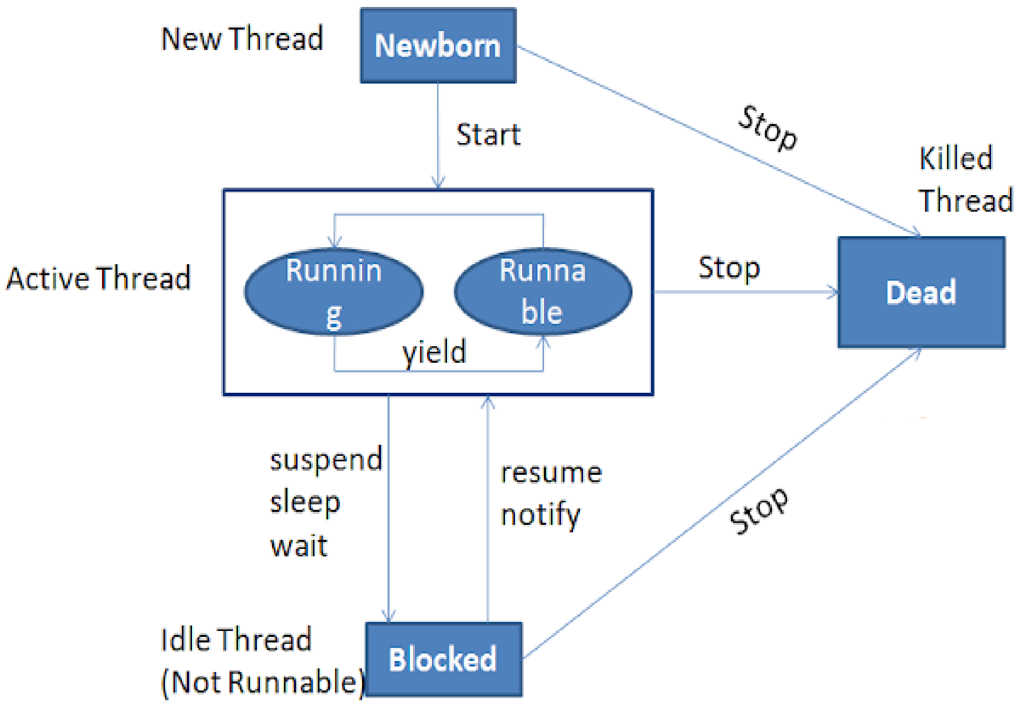

Figure 3 :State transition diagram of a thread

<u>Running state :</u> When thread is executing, it's state is changed to Running. Thread scheduler picks one of the thread from the runnable thread pool and change it's state to Running. Then CPU starts executing this thread. A thread can change state to Runnable, Dead or Blocked from running state depends on time slicing, thread completion of run() method or waiting for some resources.

A running thread can move to blocked state in the following situations.

1. It has been suspended using **suspend().** A suspended thread can be revived by using the **resume()**.

2. It has been made to sleep using **sleep(milliseconds)** method. The thread re-enters the runnable state as soon as this time period elapsed.
3. It has been told to wait until some event occurs. This is done using **wait()**. The thread can be scheduled to run again using the **notify()** method

Blocked/Waiting state: When a thread is temporarily inactive, then it's in one of the following states: **Blocked or Waiting**

For example, when a thread is waiting for I/O to complete, it lies in the blocked state. It's the responsibility of the thread scheduler to reactivate and schedule a blocked/waiting thread. A thread in this state cannot continue its execution any further until it is moved to runnable state. Any thread in these states does not consume any CPU cycle.

If a currently running thread is moved to blocked/waiting state, another thread in the runnable state is scheduled by the thread scheduler to run. It is the responsibility of thread scheduler to determine which thread to run.

A thread lies in **timed waiting state** when it calls a method with a time out parameter. A thread lies in this state until the timeout is completed or until a notification is received. For example, when a thread calls sleep or a conditional wait, it is moved to a timed waiting state.

Terminated/ Dead State: Once the thread finished executing, it's state is changed to Dead and it's considered to be not alive. That is a thread is in terminated or dead state when its run() method exits. A thread that lies in a terminated state does no longer consumes any cycles of CPU.

Creation of multithreaded program

To create a thread, Java provides a class **Thread** and an interface **Runnable** both are located into **java.lang** package.

Thread can be created either by extending Thread class or implementing Runnable interface. Both includes a run method that must be override to provide thread implementation.

It is recommended to use Runnable interface if you just want to create a thread but can use Thread class for implementation of other thread functionalities as well.

Implementing the Runnable Interface

The easiest way to create a thread is to create a class that implements the runnable interface. After implementing runnable interface, the class needs to implement the run() method.

Run Method Syntax: **public void run()**

It introduces a concurrent thread into your program. This thread will end when **run()** method terminates. The code that the thread will execute should be specified inside run() method**. run()** method can call other methods, can use other classes and declare variables just like any other normal method.

Example 20: Thread using Runnable interface

```
class MyThread implements Runnable
{
        public void run()
        {
                System.out.println("concurrent thread started running..");
        }
}

class MyThreadDemo
{
        public static void main(String args[])
        {
                MyThread mt = new MyThread();
                Thread t = new Thread(mt);
                t.start();
        }
}
```

Output

```
concurrent thread started running..
```

To call the run()method, start() method is used. On calling start(), a new stack is provided to the thread and run() method is called to introduce the new thread into the program.

Note: If you are implementing Runnable interface in your class, then you need to explicitly create a Thread class object and need to pass the Runnable interface implemented class object as a parameter in its constructor.

Extending Thread class

This is another way to create a thread by a new class that extends **Thread** class and create an instance of that class.

The extending class must override **run()** method which is the entry point of new thread.

Example 21: Thread using Thread class

```
class MyThread extends Thread
{
        public void run()
        {
                System.out.println("concurrent thread started running..");
        }
}

classMyThreadDemo
{
        public static void main(String args[])
        {
                MyThread mt = new  MyThread();
                mt.start();
        }
}
```

Output

```
concurrent thread started running..
```

In this case also, we must override the run() and then use the start() method to run the thread. Also, when you create MyThread class object, Thread class constructor will also be invoked, as it is the super class, hence MyThread class object acts as Thread class object.

Thread Priorities

Every Java thread has a priority that helps the operating system determine the order in which threads are scheduled. Java thread priorities are in the range between MIN_PRIORITY (a constant of 1) and MAX_PRIORITY (a constant of 10). By default, every thread is given priority NORM_PRIORITY (a constant of 5).

Threads with higher priority are more important to a program and should be allocated processor time before lower-priority threads. However, thread priorities cannot guarantee the order in which threads execute and are very much platform dependent.

Example 22: Thread priority- getPriority()

```
class MyThread extends Thread
{
        public void run()
        {
                System.out.println("Thread Running...");
        }
        public static void main(String[]args)
        {
                MyThread p1 = new MyThread();
                MyThread p2 = new MyThread();
                MyThread p3 = new MyThread();
                p1.start();
                System.out.println("P1 thread priority : " + p1.getPriority());
                System.out.println("P2 thread priority : " + p2.getPriority());
                System.out.println("P3 thread priority : " + p3.getPriority());

        }
}
```

Output

```
P1 thread priority : 5
Thread Running...
P2 thread priority : 5
P3 thread priority : 5
```

Example 23: Thread priority

```
class MyThread extends Thread
{
        public void run()
        {
                System.out.println("Thread Running...");
        }
        public static void main(String[]args)
        {
                MyThread p1 = new MyThread();
                p1.start();
                System.out.println("max thread priority : " + p1.MAX_PRIORITY);
                System.out.println("min thread priority : " + p1.MIN_PRIORITY);
                System.out.println("normal thread priority : " + p1.NORM_PRIORITY);

        }
}
```

Output

```
Thread Running...
max thread priority : 10
min thread priority : 1
normal thread priority : 5
```

Example 24: Thread priority-setPriority()

```
class MyThread extends Thread
{
        public void run()
        {
                System.out.println("Thread Running...");
        }
        public static void main(String[]args)
        {
                MyThread p1 = new MyThread();
                p1.start();         // Starting thread
                p1.setPriority(2);         // Setting priority
                int p = p1.getPriority();         // Getting priority
                System.out.println("thread priority : " + p);
        }
}
```

Output

```
thread priority : 2
Thread Running...
```

Thread class methods

Method	Description
setName()	to give thread a name
getName()	return thread's name
getPriority()	return thread's priority
isAlive()	checks if thread is still running or not
join()	Wait for a thread to end
run()	Entry point for a thread
sleep()	suspend thread for a specified time
start()	start a thread by calling run() method
currentThread()	Returns a reference to the currently executing thread object.
getState()	Returns the state of this thread.
setPriority(int newPriority)	Changes the priority of this thread.
yield()	A hint to the scheduler that the current thread is willing to yield its current use of a processor.

Thread Synchronization

Synchronization is a process of handling resource accessibility by multiple thread requests. The main purpose of synchronization is to avoid thread interference. At times when more than one thread try to access a shared resource, we need to ensure that resource will be used by only one thread at a time. The process by which this is achieved is called synchronization. The synchronization keyword in java creates a block of code referred to as critical section.

General Syntax:

```
synchronized (object)
{
        //statement to be synchronized
}
```

Example 25: Thread program WITHOUT SYNCHRONIZATION

```
class Table
{
        void printTable(int n)
        {//method not synchronized
```

```
                    for(int i=1;i<=5;i++)
                    {
                              System.out.println(n*i);
                              try
                              {
                                        Thread.sleep(400);
                              }
                              catch(Exception e)
                              {
                                        System.out.println(e);
                              }
                    }
          }
}

class MyThread1 extends Thread
{
          Table t;
          MyThread1(Table t)
          {
                    this.t=t;
          }
          public void run()
          {
                    t.printTable(5);
          }
}
class MyThread2 extends Thread
{
          Table t;
          MyThread2(Table t)
          {
                    this.t=t;
          }
          public void run()
          {
                    t.printTable(100);
          }
}
class TestSynchronization1
{
          public static void main(String args[])
          {
                    Table obj = new Table();//only one object
```

```
                MyThread1 t1=new MyThread1(obj);
                MyThread2 t2=new MyThread2(obj);
                t1.start();
                t2.start();
        }
}
```

Output

```
5
100
10
200
15
300
20
400
25
500
```

Example 26: Thread program WITH SYNCHRONIZATION

```
class Table
{
        synchronized void printTable(int n)
        {//method not synchronized
                for(int i=1;i<=5;i++)
                {
                        System.out.println(n*i);
                        try
                        {
                                Thread.sleep(400);
                        }
                        catch(Exception e)
                        {
                                System.out.println(e);
                        }
                }
        }
}

class MyThread1 extends Thread
{
        Table t;
        MyThread1(Table t)
        {
                this.t=t;
        }
        public void run()
        {
                t.printTable(5);
        }
}
class MyThread2 extends Thread
{
        Table t;
        MyThread2(Table t)
        {
                this.t=t;
        }
        public void run()
        {
                t.printTable(100);
        }
}
class TestSynchronization1
{
        public static void main(String args[])
        {
                Table obj = new Table();//only one object
                MyThread1 t1=new MyThread1(obj);
                MyThread2 t2=new MyThread2(obj);
                t1.start();
                t2.start();
        }
}
```

Output

```
5
10
15
20
25
100
200
300
400
500
```