Java-managing Errors and Exceptions:

Errors are wrongs that can make a program go wrong. Errors may be logical or may be typing mistakes. An error may produce an incorrect output or may terminate the execution of the program abruptly or even may cause the system to crash. It is therefore important to detect and manage properly all the possible error conditions in the programs so that the program will not terminate or crash during execution.

Types of Errors

1. Compile time errors.
2. Run time errors.

Compile Time Errors: Compile-time errors highlight a problem with the Java syntax of a program as it's being compiled (e.g., a misspelled variable name, an incorrect class definition, etc..). A program with a compile-time error cannot be compiled successfully until the error is corrected. All syntax errors will be detected and displayed by the java compiler and therefore these errors are known as **compile time errors**. Whenever the compiler displays an error, it will not create the **.class** file. It is therefore necessary to clear the errors before successfully compile and run the program.
For example:

```
class MyClass
{
 public static void main(String args[])
{
System.out.println("Hello java")                    // missing ;
}
```

Java compiler displays errors messages on the screen

Error1.java : 3 : ';' expected
System.out.println ("Hello Java")
^
1 error

When error occurs, go to the appropriate line, correct the error, and recompile the program. Most of the compile time errors are due to typing mistakes. Typographical errors are hard to find. We may have to check the code word by word, or even character by character.

The most common problems are:

- Missing semicolons.
- Missing (or mismatch of) brackets in classes and methods.
- Misspelling of identifiers and keywords.
- Missing double quotes in strings.
- Use of undeclared variables.
- Incompatible types in assignments / initialization.
- Bad references to objects.
- Use of = in place of == operator and so on.

Other errors we may encounter are related to directory paths such as
Javac: command not found i.e. we have not set the path correctly.

Run-Time Errors: Sometimes a program may compile successfully creating the .class file but may not run properly. Such programs may produce wrong results 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.
- Trying to cast an instance of a class to one of its subclasses.
- Passing a parameter that is not in a valid range or value for a method.
- Trying to illegally change the state of a thread.
- Attempting to use a negative size for an array.
- Using a null object reference as a legitimate object reference to access a method or a variable.
- Converting invalid string to a number.
- Accessing a character that is out of bounds of a string.

When such errors are encountered, java typically generates an error message and aborts the program.

Example:

```
class Error2
{ public static void main(String args[])
{ int a=10;
  int b=5;
  int c=5;
  int x=a/(b-c);    //Division by zero
  System.out.println("x="+x);
  int y=a/(b+c);
  System.out.println("y="+y);
}}
```

The above program is syntactically correct and therefore does not cause any problem during compilation. However while executing, it displays the following message and stops without executing further statements.

java.lang.ArithmaticException: / by zero
at Error2.main(Error2.java:8)

Exceptions

An exception is a problem that arises during the execution of a program. An exception can occur for many different reasons, including the following:

- A user has entered 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. 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**. Java exception handling is used to handle error conditions in a program systematically by taking the necessary action.

The mechanism performs the following tasks

- 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 exception)

The error handling code contains two parts:

1. To detect errors and to throw exceptions.
2. To catch exceptions and to take appropriate actions.

Common Java Exceptions are:

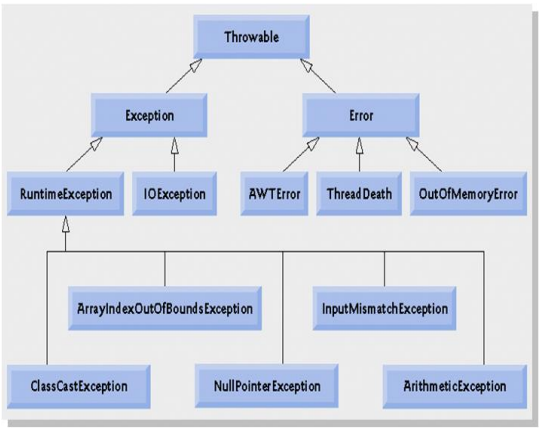
Exception type	Cause of Exception
ArithmaticException	Arithmetic error such as division by zero.
ArrayIndexOutOfBoundsException	Array index is out of bound.
ArrayStoreException	Caused when a program tries to store the wrong type of data in an array.
FileNotFoundException	Caused by an attempt to access a nonexistent file.
IOException	Caused by general I/O failures, such as inability to read from a file.
NullPointertException	Caused by referencing a null object.
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.

Exception in java can be categorized into two types:

1. Checked Exceptions.
2. Unchecked Exceptions.

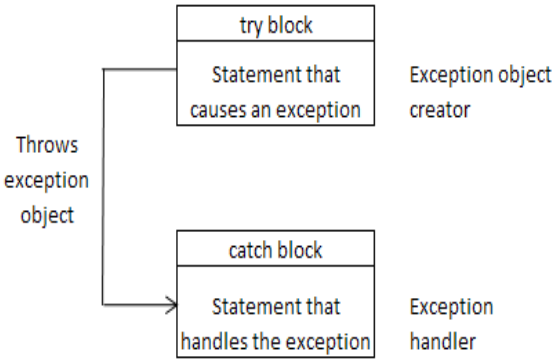
Checked exceptions: These are inherited from the core Java class Exception. They represent exceptions that are frequently considered **"non fatal"** to program execution. These exceptions are explicitly handled in the code itself with the help of **try catch** blocks. Checked exceptions are extended from the java.lang.Exception class. Checked exceptions include errors such as "array index out of bounds", "file not found", "number format conversion" and "Arithmetic exception".

Unchecked Exceptions: 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. Unchecked exceptions represent error conditions that are considered **"fatal"** to program execution. You do not have to do anything with an unchecked exception. Your program will terminate with an appropriate error message. Unchecked exceptions include errors such as "null pointer".



Inheritance hierarchy for class Throwable.

Exception handling Mechanism:



The basic idea of exception handling mechanism are throwing an exception and catching it.

Java exception handling is managed via five keywords: try, catch, throw, throws, finally.

- **Try:** The Program statements that you want to monitor for exceptions are contained within try block. If an exception occurs within the try block, it is thrown.
- **Catch:** Your code can catch this exception (using catch) and then throw it in some rational manner.
- **Throw:** System generated exceptions are automatically thrown by java runtime system. To manually throw an exception, use the keyword throw.
- **Throws:** Any exception that is thrown out of a method must be specified as such by a "throws" clause.
- **Finally:** Any codes that must be executed before a method returns are put in finally block. A finally block is always executed.

Try and Catch statement:

try

- It is a block which contains statement which we want to check for execution.
- If statement in try block generates exception then JVM creates object of exception and throw it.
- It is reserved keyword in java.

catch

- This block is always used after the try block which is used to handle the exception.
- It is a reserved keyword in java.
- This block catches the object thrown by the try block.

Syntax:

```
try
{
Statements;      //generates an exception
}
catch (Exception-type e)
{
Statements;      //processes the exception
}
```

Example of exception handling mechanism:

```
public class Exceptions
{
public static void main(String args[])
{
try
{
System.out.println("Try Block before the error.");
System.out.println(1/0);
System.out.println("Try Block after the error."); //this line will not print
}
catch(java.lang.ArithmeticException e)
{
System.out.println("Catch Block");
System.out.println("A Stack Trace of the Error:");
e.printStackTrace();
//e.getMessage(); //This one is useable when we write our own exceptions
System.out.println("The operation is not possible.");
}
finally
{
System.out.println("Finally Block");
}
}
}
```

Output:

```
Try Block before the error.
Catch Block
A Stack Trace of the Error:
java.lang.ArithmeticException: / by zero
at Exceptions.main(Exceptions.java:5)
The operation is not possible.
Finally Block
```

Multiple Catch Statements:

It is possible to have more than one catch statement in the catch block.

Syntax:

```
….
….
try
{
Statements;  //generates an exception
}
catch (Exception-type1 e)
{
Statements;  //processes exception type 1
}
catch (Exception-type2 e)
{
Statements;  //processes exception type 2
}
…..
…..
catch (Exception-typeN e)
{
Statements; //processes exception type N
}
….
```

Example of Multiple Catch Statement:

```
public class MultipleCatchDemo
{
public static void main(String args[])
{
try
{
int num1 = 24;
int num2 = 0;
System.out.println("\nIn try block trying to devide…");
int result = num1 / num2;
```

```
System.out.println("\n\nResult of division: " + result);
}
catch(ArithmeticException e1)
{
System.out.println(e1);
}
catch(Exception e)
{
System.out.println("\n\nException: " + e);
}
finally
{
System.out.println("\n\nIn the finally block…");
}
System.out.println("\n\n");
}
}
```

Output:

```
In try block trying to devide…
java.lang.ArithmeticException: / by zero
In the finally block…
```

Throw:

If you want to throw an exception explicitly then you need to use the throw clause. All the system-defined exceptions are thrown automatically, but the user-defined exceptions must be thrown explicitly using the throw clause.

Syntax: throw throwableInstance;

- If user want to create Exception and throw, **throw** keyword is used.
- Throw is reserved keyword.

The ThrowableInstance must be an object of type Throwable or subclass of Throwable. Primitive types such as int or char as well as non-Throwable classes such as String and Object, cannot be used as exceptions. There are two ways to obtain a Throwable object; using parameter in a catch clause or creating one with the new operator.

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 the exception. If it does find a match, control is transferred to that statement. If not then 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.

Example of Throw keyword:

```
import java.lang.*;
public class TestThrow
{
public static void main(String args[]) throws Exception
{
```

```
int num1=20, num2=10;
if(num1>num2)    //check condition.
throw new Exception("Number1 is greater");
else
System.out.println("Number2 is greter");
}
}
```

Output:

Exception in thread "main" java.Exception: Number1 is greater
At TestThrow.main (TestThrow.java:8)

Throws:

When there is no appropriate catch block to handle the (checked) exception that was thrown by an object, the compiler does not compile the program. To overcome this, java allows the programmer to redirect exceptions that have been raised up the call stack, by using the keyword **throws**. Thus, an exception thrown by a method can be handled either in the method itself or passed to a different method in the call stack.

- When a method generate an exception but does not handle it used throws exception.
- This behavior must be specifying to callers of the method so that caller can guard themselves against that exception.
- Throws is reserved Keyword.

Syntax: type method-name(parameter-list) throws exception-list

You do this by including a throws clause in the methods declaration. A throws clause lists the types of exceptions that a method might throw. This is necessary for all exceptions, except those of type error or RuntimeException, or any of their subclasses. All other exceptions that a method can throw must be declared in the throws clause. If they are not a compile time error will result.

Example of throws keyword:

```
class TestThrows
{
static void throwone() throws IllegalAccessException
{
System.out.println("Inside throwone");
throw new IllegalAccessException("demo");
}
public static void main(String args[])
{
try
{
throwone();
}
catch(IllegalAccessException e)
{
System.out.println("Caught "+e);
}
}
```

Output:

Inside throwone
Caught java.lang.IllegalAccessException: demo

}

Difference between throw and throws keyword:

Throw	Throws
Throw is followed by an instance.	Throws is followed by class.
Whenever we want to force an exception then we use **throw** keyword.	When we know that a particular exception may be thrown or to pass a possible exception then we use **throws** keyword.
You cannot throw multiple exceptions.	You can declare multiple exceptions e.g. public void method()throws IOException, SQLException.
"Throw" is used to handle user-defined exception.	JVM handles the exceptions which are specified by "throws".
It can also pass a custom message to your exception handling module.	Point to note here is that the Java compiler very well knows about the exceptions thrown by some methods so it insists us to handle them.
Throw keyword can also be used to pass a custom message to the exception handling module i.e. the message which we want to be printed.	We can also use **throws clause** on the surrounding method instead of **try and catch exception handler.**
Throw is used to through exception system explicitly.	Throws is used for to throws exception means throws ioexception and servletexception and etc. Java throws keyword is used to declare an exception.
Throw is used to actually throw the exception.	Whereas *throws* is declarative for the method. They are not interchangeable.
It is used to generate an exception.	It is used to forward an exception.

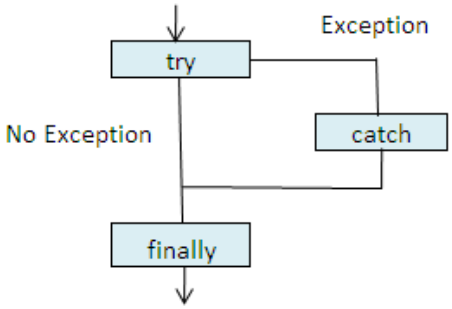
Finally Statement:

Finally block can be used to handle any exception generated within a try block. **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 an exception is thrown. If an exception is thrown, the **finally** block will execute even if no **catch** statement matches the exception. Any time a method is about to return to the caller from inside a **try/catch** block, via an uncaught exception or an explicit return statement, the **finally** clause is also executed just before the method returns. Generally the finally block is used to close the files that have been opened, connections to the databases that have been opened, sockets in the networks that have been opened or for releasing any system resources. The **finally** clause is optional. However, each **try** statement requires at least one **catch** or a **finally** clause. Finally statement can be used to handle an exception that is not caught by any of the previous catch statements. When finally block is defined, this is guaranteed to execute, regardless of whether or not in exception is thrown

Syntax:

try
{……..}

finally
{……}
Or
try
{ ……}
catch(…..)
{…….}
catch(…..)
{…….}
finally
{……}



- If try is successful or if try unsuccessful then the catch is executed and the finally block is executed.
- The finally block is optional and if required can be substituted.

Example of finally keyword:

```
public class TestFinally
{
public static void main(String args[])
{
System.out.println("Program Execution Starts here\n");
int a,b,c;
try
{
a=Integer.parseInt(args[0]);
b=Integer.parseInt(args[1]);
c=a/b;
System.out.println(a+"/"+b+"="+c);
}
catch(Exception e)
{
System.out.println(e);
}
finally
{
System.out.println("Finally blocks always get executed");
}
```

```
System.out.println("Program Execution completes here");
}
}
```

Output:

```
Javac TestFinally.java
Java TestFinally 20 10
Program Execution Starts here
20/10=2
Finally blocks always get executed
Program Execution completes here
```

Throwing User defined Exception:

User-defined exceptions are useful to handle business application specific error conditions. For example, when performing a withdrawal from a bank account, it is required to validate the minimum balance in that account. To handle this requirement, user-defined exception may be created. When creating user-defined exceptions, the name of the exception type should not be reserved exception type name. User-defined exceptions are provided by an application provider or a java API provider. Generally, all the exceptions are sub-classes of the class Throwable.

Syntax: throw new Throwable_subclass;

Program using user defined exception:

```
import java.lang.Exception;
class MyException extends Exception
{
MyException(String message)
{
super(message);
}
}

class TestMyException
{
public static void main(String args[])
{
int age=30;
try
{
if(age>0)
throw new MyException("Age is Positive");
}

catch(MyException e)
{
System.out.println("Caught my Exception");
System.out.println(e.getMessage());
}
```

```
finally
{
System.out.println("I am always here");
}
}
}
```

Output:

```
Caught my Exception
Age is Positive
I am always here
```

Chained Exception

Whenever in a program the first exception causes an another exception, that is termed as **Chained Exception**. Exception chaining (also known as "nesting exception") is a technique for handling the exception, which occur one after another i.e. most of the time is given by an application to response to an exception by throwing another exception. Typically the second exception is caused by the first exception. Therefore chained exceptions help the programmer to know when one exception causes another.

The methods and constructors in Throwable that support chained exceptions are:

- Throwable getCause()
- Throwable initCause(Throwable)
- Throwable(String, Throwable)
- Throwable(Throwable)

Methods of Throwable class

Method	Description
toString()	Returns the exception followed by a message string (if one exit)
getMessage()	Returns the message string of the Throwable object.
printStackTrace()	Returns the full name of the exception class and some additional information apart from the information of first two method.
getCause()	Returns the exception that caused the occurrence of current exception.
initCause()	Returns the current exception due to the Throwable constructors and the Throwable argument to initCause.

Syntax:

```
try
{
...
}
catch (IOException e)
{
throw new SampleException("Other IOException", e);
}
```

Example:

```
import java.io.*;
import java.util.*;
```

```
class MyException extends Exception
{
MyException(String msg)
{
super(msg);
}
}

public class ChainExcep
{
public static void main(String args[])throws MyException, IOException
{
try
{
int rs=10/0;
}
catch(Exception e)
{
System.err.println(e.getMessage());
System.err.println(e.getCause());
throw new MyException("Chained ArithmeticException");
}
}
}
```

Output:

```
/ by zero
null
Exception in thread “main” MyException” Chained ArithmeticException
At ChainExcep.main(ChainExcep.java:23)
```

Introduction of Multithreading

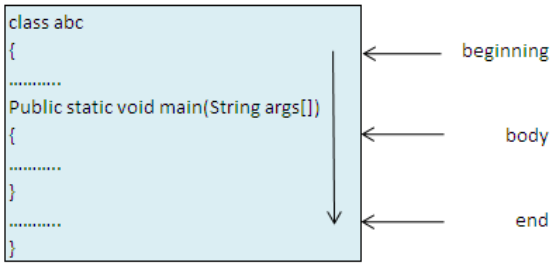
Java provides built-in support for *multithreaded programming*. A multithreaded program contains two or more parts that can run concurrently. Each part of such a program is called a thread, and each thread defines a separate path of execution. A multithreading is a specialized form of multitasking. Multithreading enables you to write very efficient programs that make maximum use of the CPU, because idle time can be kept to a minimum.

Java Multithreaded Programming

- **Multitasking** allow to execute more than one tasks at the same time, a task being a program. In multitasking only one CPU is involved but it can switches from one program to another program so quickly that's why it gives the appearance of executing all of the programs at the same time. Multitasking allow processes (i.e. programs) to run concurrently on the program. For Example running the spreadsheet program and you are working with word processor also. Multitasking is running heavyweight processes by a single Operating System.

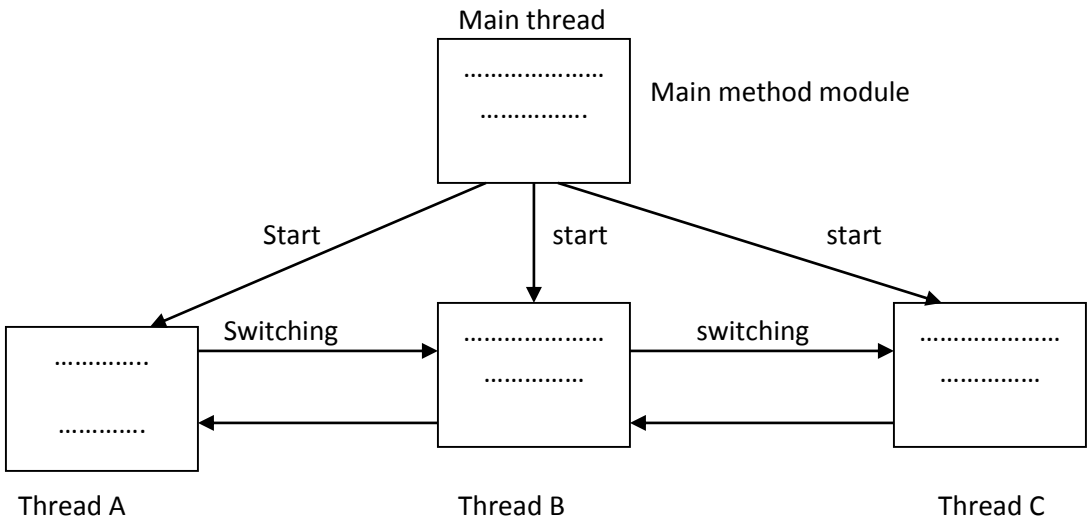
- **Multithreading** is a technique that allows a program or a process to execute many tasks concurrently (at the same time and parallel). It allows a process to run its tasks in parallel mode on a single processor system.
- In the multithreading concept, several multiple lightweight processes are run in a single process/task or program by a single processor. For Example, when you use a **word processor** you perform a many different tasks such as **printing, spell checking** and so on. Multithreaded software treats each process as a separate program.
- Multithreading is something similar to dividing a task into subtasks and assigning them to different people for execution independently and simultaneously.
- In Java, the Java Virtual Machine **(JVM)** allows an application to have multiple threads of execution running concurrently. It allows a program to be more responsible to the user. When a program contains multiple threads then the CPU can switch between the two threads to execute them at the same time.

A **thread** is similar to a program that has a single flow of control. It has a beginning, a body, end, and executes commands sequentially. All main programs in our examples can call **single threaded** programs. Every program will have at least one thread.



Single Threaded Program

- A multithreaded program contains two or more parts that can run concurrently. Each part of such a program is called a thread, and each thread defines a separate path of execution. A multithreading is a specialized form of multitasking. Multithreading enables you to write very efficient programs that make maximum use of the CPU, because idle time can be kept to a minimum. In the figure below, java program with four threads, one main thread and three others. The main thread is designed to create and start the other threads A, B and C. Once initiated by the main thread, the thread A, B and C run concurrently and share the resources jointly. The ability of a language to support multithreads is referred to as ***concurrency***. Threads in java are sub programs of a main application program and share the same memory space, they are known as **lightweight threads** or **lightweight processes**. Threads' running in parallel does not really mean that they are actually running at the same time. All the threads are running on a single processor, the flow of execution is shared between the threads. The java interpreter handles the switching of control between the threads in such a way that it appears they are running concurrently.



Multithreaded Program

Types of Multithreading

- Multithreading is a specialized form of multitasking.
- There are two types of multitasking: **Thread-based** and **Process-based.**

Process based Multitasking:

A process is, in essence, a program that is executing. Thus process-based multitasking is the feature that allows your computer to run two or more threads concurrently. Example: Process-based multitasking enables you to run the Java compiler at the same time that you are using a text editor. A program is the smallest unit of code that can be dispatched by the scheduler.

Thread based multitasking:

In thread-based multitasking environment, the thread is the smallest unit of dispatchable code. This means that a single program can perform two or more tasks simultaneously. Example: A Text Editor can format text at the same time it is printing, as long as these two actions are performed by two separate threads.

Thread based multiprocessing

- Require less overhead.
- Threads are light weight.
- They share the same address space.
- Interprocess communication is inexpensive.

Difference between multithreading and multitasking.

Multithreading	Multitasking
Multithreading is a programming concept in which a program or a process is divided into two or more subprograms or threads that are executed at the same time in parallel.	Multitasking is an operating system concept in which multiple tasks are performed simultaneously.
It supports execution of multiple parts of a single program simultaneously.	It supports execution of multiple programs simultaneously.
The processors have to switch between different parts or threads of a program.	The processor has to switch between different programs or processes.
It is highly efficient.	It is less efficient.
A thread is the smallest unit in multithreading.	A program or process is the smallest unit in a multitasking.
It helps in developing efficient programs.	It helps in developing efficient operating system.
Light weight process and can run in a same address space so context switch or Intercommunication between processes is less expensive.	Heavyweight process and runs in a different address space so context switch or intercommunication between processes is much expensive.
Multithreading requires less overhead than multitasking.	Multitasking requires more overhead.
Multithreading is a ability of program or an OS process to manage its use by more than one user at a time and to even manage multiple request by the same user without having to have multiple copies of the program running in the computer.	The operating system is able to keep track of where you are in there task and go from one to the other without losing information.

Java Thread:

Thread is a smallest unit of executable code or a single task is also called as thread. Each thread has its own local variables, program counter and lifetime. A thread is similar to program has a single flow of control. It has a beginning, a body and an end, and executes commands sequentially. The java run time system depends on thread. The benefits of java multithreading are that, one thread can pause stopping without stopping other part of program. The time of the CPU can be utilized by thread when one thread reads data from the network or waits for something.

A thread can be in various stages of execution.

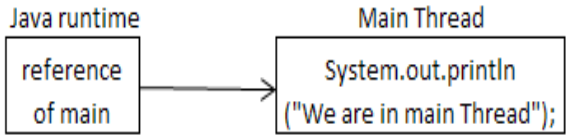
1. A thread can be running.
2. A thread can be ready to run as soon as it gets CPU time.
3. A running thread can be suspended.
4. A thread can be blocked.
5. A suspended thread can be resumed.
6. At any time thread can be terminated.

Main Thread:

When any standalone application is running, it firstly execute the **main()** method runs in a one thread, called the main thread. If no other threads are created by the main thread, then program terminates when the main() method complete its execution. The main thread creates some other threads called

child threads. The main() method execution can finish, but the program will keep running until the all threads have complete its execution. The main is the first thread created automatically when java program starts. This main thread can be controlled through a thread object. It obtains reference to it by simply calling the method currentThread.
Syntax: static Thread currentThread;
The above method returns the reference of the currentThread which method is static so it has to be called by the name of the class as: Thread.currentThread
Example:

```
class ThreadTest
{
public static void main(String args[])
{
System.out.println(“We are in main Thread”);
}
}
```



Representation of main thread

When this program is executed it creates the main thread. Whose reference will be with the java run time system? As you can imagine very properly that everything written inside the main() i.e. public static void main(String args[]) will be executing inside the main thread.
In order to control main thread we must have reference of it. As various objects of various classes are created in main and they are controlled from main itself by their reference variables. So to get the reference of main which is currentthread, we will call the above (currentThread()) method to get its reference to control it. As the main is the one thread, after getting its reference, we have to store it in an reference variable of Thread class type as explained below:

```
class MainThread
{
public static void main(String args[])
{
Thread t=Thread.currentThread();
System.out.println(“Current Thread is=”+t);
}
}
```

Output: Current Thread=Thread[main, 5, main]

In above program the reference to the main thread is obtained by calling the currentThread() method and then this reference is stored in the reference variable ‘t’ which is of type Thread. The program then displays the current thread’s information. Notice that in output it is displayed some information related to the thread, (here main) as [main,5,main]. Here it displays it in order as name of thread, its priority and group of thread. By default in java the name of mainthread is main, it’s priority as 5, which is always default value of priority of any thread and last main is the name of the group to which this thread belongs.

Advantages of multithreaded Programming:

- Large programs can be divided into threads and execute them in parallel.
- Reduces the computation time.
- Improves performance of an application.
- Threads share the same address space so it saves the memory.
- Threads shares resources of the same process, so it becomes economical.
- Context switching between threads is usually less expensive than between processes.
- Cost of communication between threads is relatively low.
- Enables to do multiple things at the same time.
- Speed of program is improved because the program is divided into threads and threads are executed concurrently.

Disadvantages of Multithreading

- Code writing, debugging, managing concurrency, testing, porting existing code is difficult in multithreading.
- Programmers need to remove static variables and replace any code that is not thread-safe to introduce threading into a previously non threaded application.
- Programmer may face the problem of race conditions or deadlock.
- Thread consumes more processor time.
- Operating system spends more time in managing and switching between the threads.

Creating Threads:

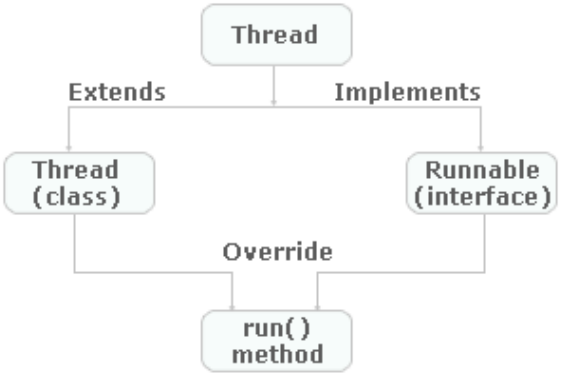
In Java, an object of the Thread class can represent a thread. When a thread is created, it must be permanently bound to an object with a run() method. When the thread is started, it will invoke the object's run() method. The run() method is the heart and soul of any thread. The run() method makes up the entire body of thread and is the only method in which the threads behavior can be implemented. A run() appears as follows:

```
public  void run()
{
……
…… (Statements for implementing thread)
……
}
```

The run() method is invoked by an object of concerned thread. This can be achieved by creating a thread and initiating it with the help of another thread method called start().

There are two ways to create thread in java are:

- **By Extending the Thread class (java.lang.Thread):** User specified thread class is created by extending the class **Thread** and overriding its **run()** method.
- **Implement the Runnable interface (java.lang.Runnable):** Define a class that implements **Runnable** interface. The Runnable interface has only one method, **run(),** that is to be defined in the method with the code to be executed by the thread.



Extending the Thread Class

For creating a thread a class have to extend the Thread Class i.e. java.lang.Thread.

Creating thread by using extending the thread class:

1. Declare the class as extending the Thread class:

```
Class MyThread extends Thread
{ ……..}
```

2. Implement the run() method that is responsible for executing the sequence of code that the thread will execute:

```
public void run()
{ ……..}
```

3. Create a thread object and call the start() method to initiate the thread execution

```
MyThread ThreadA= new MyThread();
ThreadA.start();            //invokes run() method
```

Program demonstrates a single thread creation extending the "Thread" Class:

```
class MyThread extends Thread
{
 String s=null;
 MyThread(String s1)
 {
 s=s1;
 start();
 }
 public void run()
 {
 System.out.println(s);
 }
}

public class RunThread
{
public static void main(String args[]){
MyThread m1=new MyThread("Thread started....");
}
```

```
}
```
Output: Thread started....

Program of creating thread using the Thread class

```
import java.lang.*;
class A extends Thread
{
public void run()
{
for(int i=1;i<=5;i++)
{
System.out.println("\t From Thread A="+i);
}
System.out.println("Exit from Thread A");
}
}

class B extends Thread
{
public void run()
{
for(int j=1;j<=5;j++)
{
System.out.println("\t From Thread B="+j);
}
System.out.println("Exit from Thread B");
}
}

class C extends Thread
{
public void run()
{
for(int k=1;k<=5;k++)
{
System.out.println("\t From Thread C="+k);
}
System.out.println("Exit from Thread C");
}
}

class ThreadTest
{
public static void main(String args[])
{
new A().start();
```

```
new B().start();
new C().start();
}
}
```

Implementing the Runnable Interface

The easiest way to create a thread is to create a class that implements the **Runnable** interface. **Runnable** abstracts a unit of executable code. You can construct a thread on any object that implements **Runnable**. To implement **Runnable**, a class need only implement a single method called **run( )**, which is declared like this: public void run( ). Inside **run( )**, you will define the code that constitutes the new thread. After you create a class that implements **Runnable**, you will instantiate an object of type **Thread** from within that class. After the new thread is created, it will not start running until you call its **start( )** method, which is declared within **Thread**. In essence, **start( )** executes a call to **run( )**. The **start()** method is shown here: void start( )

The procedure for creating threads by implementing the Runnable Interface is as follows:

- Declare the class as implementing the Runnable interface.
- Implement the run() method.
- Create a thread by defining an object that is instantiated from this "runnable" class as the target of the thread.
- Call the thread's start() method to run the thread.

Program for creating thread using Runnable interface:

```
class Threadrun implements Runnable
{
public void run()
{
System.out.println("Implementing the runnable interface");
}
public static void main(String args[])
{
Threadrun tr=new Threadrun();
Thread t=new Thread(tr);
t.start();
}
}
```

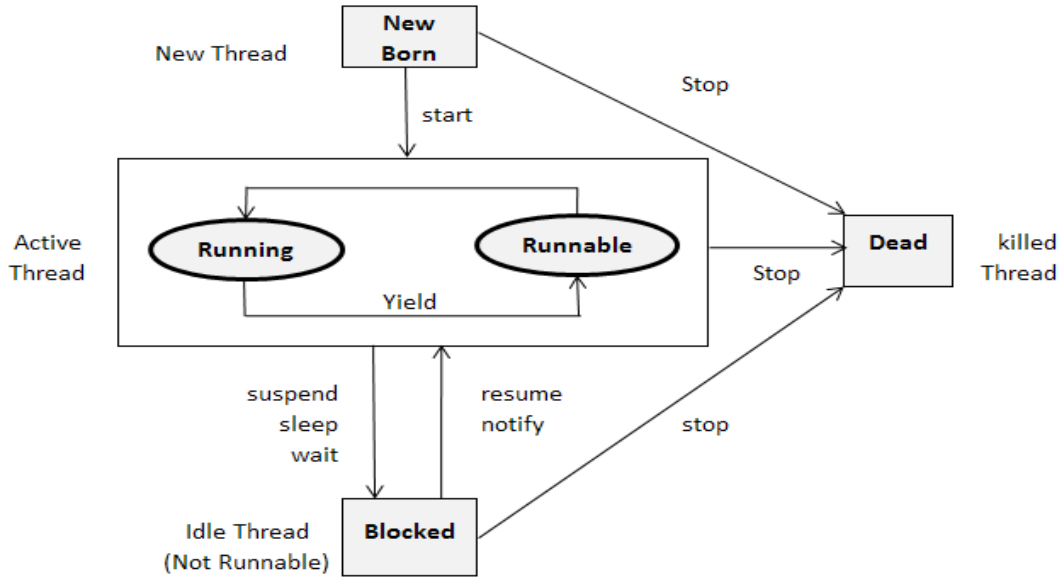
Output: Implementing the runnable interface

Life cycle of a Thread

During the life time of a thread, it enters into many states. The thread has following states:

1. Newborn state
2. Runnable state
3. Running state
4. Blocked state
5. Dead state

A thread is always in one of these five states. It can move from one state to another via a variety of ways as shown in figure.



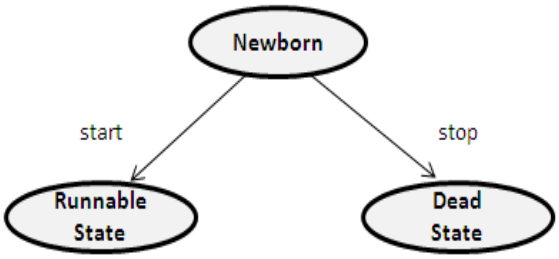
Life Cycle of Thread

Newborn State:

In Newborn the thread is newly created and is not yet running. At this stage we can do one of the following things with thread:

- Schedule it for running using start() method
- kill it using stop() method

If scheduled, it moves to the runnable state. If we attempt to use any other method at this stage, an exception will be thrown.



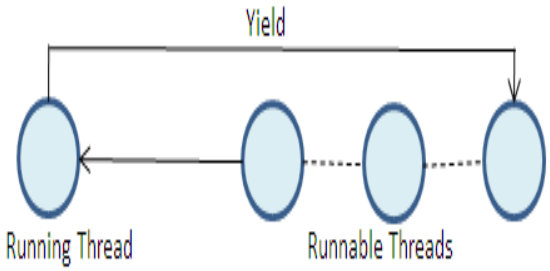
Scheduling a newborn thread

Runnable state:

Runnable state means that the thread is ready for execution but is not being executed currently. On this state a thread is waiting for a turn on the processor. The thread has joined the queue of thread that is waiting for execution. If all threads have equal priority, then they are given time slots for execution in round robin fashion i.e. first come, first serve manner.

The thread that relinquishes control joins the queue at the end and again waits for it turn. This process of assigning time to threads is known as time-slicing.

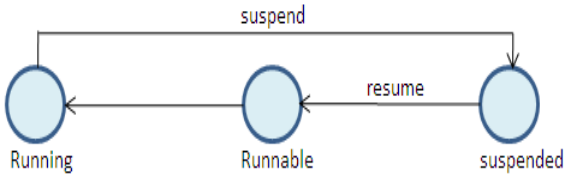
However, if we want a thread to relinquish control to another thread to equal priority before its turn comes, we can do so by using the **yield()** method.



Relinquishing control using yield() method

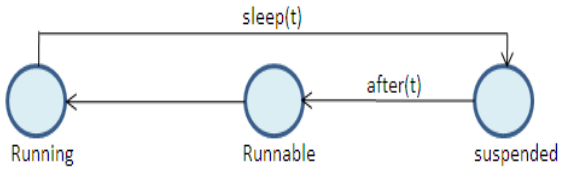
Running State: A thread is in running state that means the thread is currently executing. Processor has given its time to the thread for its execution. At a time only one thread can be in this stage, all other active thread will be in Runnable stage waiting for their execution time. The thread runs until it relinquishes control on its own or it is preempted by a higher priority thread. A running thread may relinquish its control in one of the following situations:

1) It has been suspended using **suspend()** method. A suspended thread can be revived by using the **resume()** method. This approach is useful when we want to suspend a thread for some time due to certain reason, but do not want to kill it.



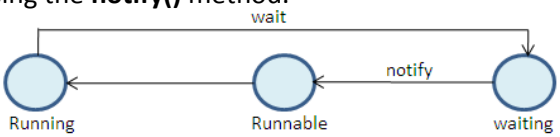
(1) Relinquishing control using suspend() method

2) it has been made to sleep. We can put a thread to sleep for a specific time period using the method **sleep(time)** where time is in milliseconds. This means that the thread is out of the queue during this time period. The thread re-enters the runnable state as soon as this time period is elapsed.



(2) Relinquishing control using sleep() method

3) It has been told to wait until some event occurs. This is done using the **wait()** method. The thread can be scheduled to run again using the **notify()** method.



(3) Relinquishing control using wait() method

Blocking a Thread:

A thread can enter the blocked state (temporarily suspended or blocked) when one of the following five conditions occurs.

- When **sleep()** is called. //blocked for a specified time.
- When **suspend()** is called. // blocked until further orders.
- When **wait()** is called. // blocked until certain condition occurs.
- The thread calls an operation, (for example, during input/output, a thread will not return until the I/O operations completes).
- The thread is waiting for a monitor.

These methods cause the thread to go into the block (or non-runnable) state. A thread must move out of a blocked state into the runnable (or running) state using the opposite of whatever phenomenon put it into the blocked state.

- If a thread has been put to **sleep(),** the specified timeout period must expire.
- If a thread has called **wait(),** then some other thread using the resource for which the first thread is waiting must call **notify()** or **notifyAll().**
- If a thread is waiting for the completion of an input or output operation, then the operation must finish.

Dead State:

The running thread can be stop by calling **stop()** method. A running thread ends it life when it has completed executing its **run()** method. It is a natural death. However we can kill it by sending the stop message to it at any state thus causing a premature death to it. A thread can be killed as soon as it is born or while it is running, or even when it is in "not runnable" (blocked) condition.

Thread Methods:

Thread class methods can be used to control the behavior of the thread. Some Important Methods defined in **java.lang.Thread** are shown in the table:

Method	Return type	Description
currentThread( )	Thread	Returns an object reference to the thread in which it is invoked.
getName( )	String	Retrieve the name of the thread object or instance.
start( )	void	Start the thread by calling its run method.
run( )	void	This method is the entry point to execute thread, like the main method for applications.
sleep( )	void	Suspends a thread for a specified amount of time (in milliseconds).
isAlive( )	boolean	This method is used to determine the thread is running or not.
activeCount( )	int	This method returns the number of active threads in a particular thread group and all its subgroups.
interrupt( )	void	The method interrupts the threads on which it is invoked.
yield( )	void	By invoking this method the current thread pause its execution temporarily and allow other threads to execute.
join( )	void	This method and **join(long millisec)** Throws InterruptedException. These two methods are invoked on a thread. These are not returned until either the thread has completed or it is timed out respectively.

Thread Priority

In java, each thread is assigned a priority, and according to their priority it is scheduled for running. The threads of the same priority are given equal treatment by the java scheduler and therefore they share the processor on a first come, first serve basis. In java to set the priority of a thread using the setPriority() method.

Syntax: ThreadName.setPriority(intNumber);

Constant	Description
MIN_PRIORITY	The minimum priority of any thread (an int value of 1)
NORM_PRIORITY	The normal priority of any thread (an int value of 5)
MAX_PRIORITY	The maximum priority of any thread. (an int value of 10)

The intNumber is an integer value to which the threads priority is set. The intNumber may assume one of these constants or any value between 1 and 10. Default setting is NORM_PRIORITY

The methods that are used to set the priority of thread shown as:

Method	Description
setPriority()	This method is used to set the priority of thread.
getPriority()	This method is used to get the priority of thread.

When a Java thread is created, it inherits its priority from the thread that created it. At any given time, when multiple threads are ready to be executed, the runtime system chooses the runnable thread with the highest priority for execution. In Java runtime system, **preemptive scheduling** algorithm is applied. If at the execution time a thread with a higher priority and all other threads are runnable then the runtime system chooses the new **higher priority** thread for execution. On the other hand, if two threads of the same priority are waiting to be executed by the CPU then the **round-robin** algorithm is applied in which the scheduler chooses one of them to run according to their round of **time-slice**.

Thread Scheduler:

In the implementation of threading, scheduler usually applies one of the following two strategies:

- **Preemptive scheduling?** If the new thread has a higher priority than current running thread leaves the runnable state and higher priority thread enter to the runnable state.
- **Time-Sliced (Round-Robin) Scheduling?** A running thread is allowed to be execute for the fixed time, after completion the time, current thread indicates to another thread to enter it in the runnable state.

Synchronization:

When two or more threads need access to a shared resource, they need some way to ensure that the resource will be used by only one thread at a time. Thread uses their own data and methods provided inside their run() methods. When threads try to use their data and methods outside themselves will create a serious problem. For example, one thread may try to read a record from a file while another is still writing to the same file. Depending on the situation, we may get strange results. Java enables us to overcome this problem using a technique known as **synchronization**.

The method that will read information from a file and the method that will update the same file may be declared as synchronized. Synchronization is the capability to control the access of multiple threads to share resources.

Example: synchronized void update()
{……
//code here is synchronized
……
}

When we declare a method synchronized, java creates a **"monitor"** and hands it over to the thread that calls the method first time. As long as the thread holds the monitor, no other thread can enter the synchronized section of code. A monitor is like a key and the thread that holds the key can only open the lock. It is also possible to mark a block of code as synchronized as below:

synchronized (lock-object)
{……..}

Whenever a thread has completed its work of using synchronized method (or block of code), it will hand over the monitor to the next thread that is ready to use the same resource.

Example of synchronisation:

```
import java.lang.*;
class MyThread extends Thread
{
static String message[]={"java","is","truly","object","oriented"};
public MyThread(String s)
{
super(s);
}

public void run()
{
Sync.display(getName(),message);
}

void waiting()
{
try
{ sleep(1000); }
catch(InterruptedException e)
{ System.out.println("Error"); }
}
}

class Sync
{
public static synchronized void display(String name, String list[])
{
for(int i=0;i<list.length;i++)
{
MyThread thread=(MyThread)Thread.currentThread();
thread.waiting();
System.out.println(name+list[i]);
}
```

```
}
}

class ThreadSync
{
public static void main(String args[])
{
MyThread thread1=new MyThread("Thread 1: ");
MyThread thread2=new MyThread("Thread 2: ");
thread1.start();
thread2.start();
}
}
```

Output:

```
Thread 1: java
Thread 1: is
Thread 1: truly
Thread 1: object
Thread 1: oriented
Thread 2: java
Thread 2: is
Thread 2: truly
Thread 2: object
Thread 2: oriented
```

Why do we need Synchronization in Java?

If your code is executing in multi-threaded environment you need synchronization for objects which are shared among multiple threads to avoid any corruption of state or any kind of unexpected behavior. Synchronization in Java will only be needed if shared object is mutable. If your shared object is read only or immutable object you don't need synchronization despite running multiple threads. Same is true with what threads are doing with object if all the threads are only reading value then you don't require synchronization in java. JVM guarantees that *Java synchronized code will only be executed by one thread at a time*.

Inter-thread Communication

What if the producer could communicate to the consumer once it has finished producing the required data? The consumer would then not use up CPU cycles just to check whether the producer has done its job. This communication between threads is called inter-thread communication. Inter-thread communication is achieved using four methods:

Method	Description
Wait()	The wait() method tells the current thread to give up the monitor and sleep until another thread calls the notify() method. You can use the wait() method to synchronize threads.
Notify()	The notify() method wakes up a single thread that is waiting for the current monitor of the object. If multiple threads are waiting, one of them is chosen arbitrarily.
notifyAll()	The method notifyAll() is used to wake up all the threads that are waiting for the monitor of the object.
Yield()	The yield() method causes the runtime system to put the current thread to sleep and execute the next thread in the queue.

Example program of inter-thread communication

```
class Q
{
int n;
boolean valueSet = false;
synchronized int get()
{
if(!valueSet)                    //if no job is available
try
{
wait();                          // wait till notified of job opening
}
catch(InterruptedException e)
{
System.out.println("InterruptedException caught");
}
System.out.println("Got: " + n);  //secured the job
valueSet = false;                //no more jobs available
notify();
return n;
}

synchronized void put(int n)
{
if(valueSet)              //if job has not been taken by a candidate
try
{
wait();                   //wait for the existing job to be taken up
}
catch(InterruptedException e)
{
System.out.println("InterruptedException caught");
}
this.n = n;
valueSet = true;
System.out.println("Put: " + n);
notify();          // notify job opening
}
}

class Producer implements Runnable
{
Q q;
Producer(Q q)
{
this.q = q;
new Thread(this, "Producer").start(); // create a thread that produces job opening
}

public void run()
{
```

```
int i = 0;
while(true)
{
q.put(i++);                    //add another job opening
}
}
}

class Consumer implements Runnable
{
Q q;
Consumer(Q q)
{
this.q = q;
new Thread(this, "Consumer").start();// create a thread that consumer job opening
}
public void run()
{
while(true)
{
q.get(); // go for the job
}
}
}

public class intertest
{
public static void main(String args[])
{
Q q = new Q();
new Producer(q);
new Consumer(q);
System.out.println("Press Control-C to stop.");
}
}
```

Inside get( ), wait( ) is called. This causes its execution to suspend until the Producer notifies you that some data is ready. When this happens, execution inside get( ) resumes. After the data has been obtained, get( ) calls notify( ). This tells Producer that it is okay to put more data in the queue. Inside put( ), wait( ) suspends execution until the Consumer has removed the item from the queue. When execution resumes, the next item of data is put in the queue, and notify( ) is called. This tells the Consumer that it should now remove it.

Output:

```
Put: 101
Got: 101
Put: 102
Got: 102
Put: 103
Got: 103
Put: 104
Got: 104
Put: 105
```

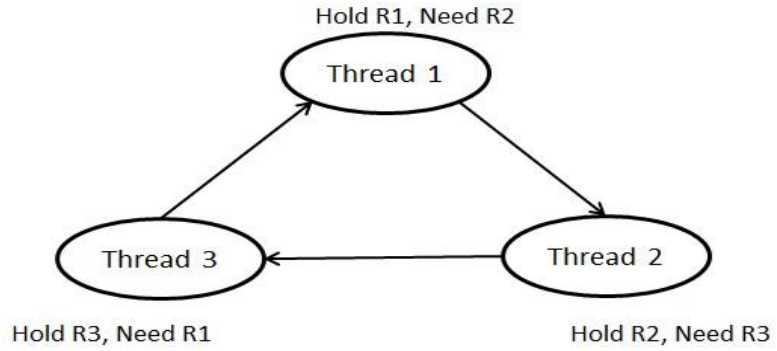
Got: 105

Difference between wait and sleep

Wait()	Sleep()
wait() method releases the lock.	sleep() method doesn't release the lock.
wait() is the method of Object class.	Sleep() is the method of Thread class.
Wait() is the non-static method.	Sleep() is the static method.
should be notified by notify() or notifyAll() methods.	After the specified amount of time, sleep is completed.
Wait() is generally used on condition.	Sleep() method is simply used to put your thread on sleep.
Wait() method called from synchronized block.	No such requirement.

Deadlock

In applications where multiple threads are competing for accessing multiple resources, there is a possible of condition known as a deadlock. Deadlock is a situation of complete lock, when no thread can complete its execution because lack of resources. In fig.



Thread-1 is holding a resource R1, and need another resource R2 to finish execution, but R2 is locked by Thread-2, which need R3, which in turn is locked by Thread-3. And Thread-3 needs R1, which is locked by Thread-1. Hence none of them can finish and are stuck in a deadlock.

Example program of deadlock

```
public class DeadlockExample
{
public static void main(String[] args)
{
final String resource1 = "Printer";
final String resource2 = "Scanner";
// t1 tries to lock resource1 then resource2
Thread t1 = new Thread()
{
public void run()
{
synchronized (resource1)
{
System.out.println("Thread 1: locked resource 1");

try
{
Thread.sleep(100);
```

```
}
catch (Exception e) {}

synchronized (resource2)
{
System.out.println("Thread 1: locked resource 2");
}
}
}
};

// t2 tries to lock resource2 then resource1
Thread t2 = new Thread()
{
public void run()
{
synchronized (resource2)
{
System.out.println("Thread 2: locked resource 2");

try
{
Thread.sleep(100);
}
catch (Exception e) {}

synchronized (resource1)
{
System.out.println("Thread 2: locked resource 1");
}
}
}
};

t1.start();
t2.start();
}
}
```

Output:

```
Thread 1: locked resource 1
Thread 2: locked resource 2
```

Solved Programs

1. Program to throw user defined exception by accepting a number from user and throw an exception if the number is not positive number.

```
import java.lang.*;
import java.io.*;
class PException extends Exception
{
PException(String msg)
{
super(msg);
}
}

class posex
{
public static void main(String args[])
{
BufferedReader bf=new BufferedReader(new InputStreamReader(System.in));
int n;
try
{
System.out.println("Enter any Number");
n=Integer.parseInt(bf.readLine());
if(n>0)
{
System.out.println(“You are Entered Positive Number, please entered Negative number for
exception ");
}
else
{
throw new PException("Entered Number is Negative Number");
}
}
catch(PException e)
{System.out.println(e);
}
catch(IOException e)
{
System.out.println(e);}
}
}
```

Output:

```
Enter any Number 3
You are Entered Positive Number, please entered Negative number for exception
Enter any Number -3
PException: Entered Number is Negative Number
```

2. Program to accept a password from the user and throw 'authentication Failure' exception if the password is incorrect.

```
import java.io.*;
import java.lang.*;
class MyException extends Exception
{
MyException(String msg)
{
super(msg);
}
}

class passwordau
{
public static void main(String args[])
{
DataInputStream d=new DataInputStream(System.in);
String s1=new String();
String s2=new String();
try
{
System.out.println("Enter to set the password");
s1=d.readLine();
System.out.println("Re-Enter the password");
s2=d.readLine();

if(s1.equals(s2))
{
System.out.println("Password Validated");
}
else
{
throw new MyException("Authentication Failure");
}
}

catch(MyException e)
{
System.out.println(e);
}
catch(IOException e)
{
System.out.println(e);
}
}
}
```

Output:

```
Enter to set the password abc
Re-Enter the password abc
```

Password Validated
Enter to set the password abc
Re-Enter the password xyz
MyException: Authentication Failure

3. Program to accept a number from the user and throw an exception if the number is not an even number.

```
import java.io.*;
import java.lang.*;
class MyException extends Exception
{
MyException(String msg)
{
super(msg);
}
}

class oddeven
{
public static void main(String args[])
{
DataInputStream d=new DataInputStream(System.in);
int n;
try
{
System.out.println("Enter a Number");
n=Integer.parseInt(d.readLine());
if(n%2==0)
{
throw new MyException("Number is EVEN");
}
else
{
throw new MyException("Number is ODD");
}
}

catch(MyException e)
{
System.out.println(e);
}

catch(IOException e)
{
System.out.println(e);
}
}
}
```

Output:

Enter a Number 2

MyException: Number is EVEN
Enter a Number 3
MyException: Number is ODD

4. Program to accept a string from the user and throw an exception if the string is not containing character 'a'.

```
import java.io.*;
import java.lang.*;
class MyException extends Exception
{
MyException(String msg)
{
super(msg);
}
}
class StringB
{
public static void main(String args[])
{
DataInputStream d=new DataInputStream(System.in);
String s;
try
{
int len;
char ch;
System.out.println("Enter the String");
s=d.readLine();
if(s.indexOf("a")!=-1)
{
System.out.println("Your String contains 'a'");
}
else
{
throw new MyException("Your String does not contains 'a'");
}
}

catch(MyException e)
{
System.out.println(e);
}
catch(IOException e)
{
System.out.println(e);
}
}
}
```

Output:

Enter the String java
Your String contains 'a'

Enter the String ops
MyException: Your String does not contains 'a'

5. Write a program to throw a user defined exception "String Mismatch Exception" if two strings are not equal. (ignore case).

```
import java.io.*;
class MyException extends Exception
{
MyException(String msg)
{
super(msg);
}
}
class ExceptionTest
{
public static void main(String args[])
{
BufferedReader br=new BufferedReader (new InputStreamReader(System.in));
String s1,s2;
try
{
System.out.println("Enter String one and String two ");
s1=br.readLine();
s2=br.readLine();
if(s1.equalsIgnoreCase(s2))                    // any similar method which give correct result
{
System.out.println("String Matched");
}
else
{
 throw new MyException("String Mismatch Exception");
}
}
catch(MyException e)
{
System.out.println(e);
}
catch(IOException e)
{
System.out.println(e);
}
}
}
```

6. Write a program to throw a user defined exception as 'Invalid Age', if age entered by the user is less than eighteen. Also mention any two common java exceptions and their cause.

```
import java.lang.Exception;
import java.io.*;
class myException extends Exception
```

```
{
myException(String msg)
{
super(msg);
}
}
class agetest
{
public static void main(String args[])
{
BufferedReader br=new BufferedReader(new InputStreamReader(System.in));
try
{
System.out.println("enter the age : ");
int n=Integer.parseInt(br.readLine());
if(n < 18 )
throw new myException("Invalid Age");
else
System.out.println("Valid age");
}
catch(myException e)
{
System.out.println(e.getMessage());
}
catch(IOException ie)
{}
}
}
```

7. Program to create two threads such that one thread print even numbers and another thread print odd numbers between 1 to 10.

```
import java.lang.*;
class even extends Thread
{
public void run()
{
try
{
for(int i=0;i<=10;i=i+2)
{
System.out.println("\tEven thread="+i);
Thread.sleep(300);
}
}
catch(InterruptedException e)
{}
}
}

class odd extends Thread
{
```

```
public void run()
{
try
{
for(int i=1;i<=10;i=i+2)
{
System.out.println("\todd thread="+i);
Thread.sleep(300);
} }
catch(InterruptedException e)
{}
}
}

class evenodd
{
public static void main(String args[])
{
new even().start();
new odd().start();
System.out.println("Exit from main");
}
}
```

Output:

```
Even thread=0
Exit from main
odd thread=1
Even thread=2
odd thread=3
Even thread=4
odd thread=5
Even thread=6
odd thread=7
Even thread=8
odd thread=9
Even thread=10
```

8. Write a program to create a thread to print all odd no. from 1 to 20 with a delay of 100 no. after each other.

```
public class Threads
{
public static void main(String[] args)
{
Thread th = new Thread();
System.out.println("Even Numbers 100 mseconds : ");
Try
{
for(int i = 0;i <= 20;i=i+2)
{
```

```
System.out.println(i);
th.sleep(100);
}
}
catch(InterruptedException e)
{
System.out.println("Thread interrupted!");
e.printStackTrace();
}
}
}
```

Output:

```
Even Numbers 100 mseconds :
0
2
4
6
8
10
12
14
16
18
20
```

9. Implement a program having two threads such that one thread prints prime numbers from 1 to 10 and other thread prints non prime numbers from 1 to 10 (use thread class) each thread has a delay of 500 milliseconds after printing one number.

```
class prime extends Thread
{
public void run()
{
int i=0,j=0;
for(i=2;i<11;i++)
{
for(j=2;j<i;j++)
{       if(i%j==0)
j=i+1;
}
if(i==j)
System.out.println(i+" is a prime number");
try
{       sleep(500);
}
catch(Exception e)
{       System.out.println("General Error"+e);
}
}
}
```

```
}

class non_prime extends Thread
{
public void run()
{
int i=0,j=0;
for(i=2;i<11;i++)
{
for(j=2;j<i;j++)
{
if(i%j==0)
j=i+1;
}
if(i!=j)
System.out.println(i+" is not a prime number");
try
{
sleep(500);
}
catch(Exception e)
{
System.out.println("General Error"+e);
}
}
}
}

class primethread
{
public static void main(String args[])
{
prime p=new prime();
non_prime n=new non_prime();
System.out.println("Start main");
System.out.println("1 is universal constatnt");
p.start();
n.start();
}
}
```

Output:

```
Start main
1 is universal constatnt
2 is a prime number
3 is a prime number
4 is not a prime number
5 is a prime number
6 is not a prime number
7 is a prime number
8 is not a prime number
```

9 is not a prime number
10 is not a prime number

10. Write a program to create two threads, one to print numbers original order and other in reverse order from 1 to 50.

```
import java.lang.*;
class even extends Thread
{
public void run()
{
try
{
for(int i=1;i<=10;i++)
{
System.out.println("\tFirst thread="+i);
Thread.sleep(300);
}
}
catch(InterruptedException e)
{}
}
}

class odd extends Thread
{
public void run()
{
try
{
for(int i=10;i>=0;i--)
{
System.out.println("\tSecond thread="+i);
Thread.sleep(300);
}
}
catch(InterruptedException e)
{}
}
}

class evenodd
{
public static void main(String args[])
{
new even().start();
new odd().start();
System.out.println("Exit from main");
}
}
```

Output:

```
First thread=1
Exit from main
Second thread=10
Second thread=9
First thread=2
Second thread=8
First thread=3
Second thread=7
First thread=4
Second thread=6
First thread=5
Second thread=5
First thread=6
Second thread=4
First thread=7
Second thread=3
First thread=8
First thread=9
Second thread=2
First thread=10
Second thread=1
Second thread=0
```

11. Program to define a class Sharedata having a member count and two synchronize member functions getDate() that initialize the value of count and putData() displays value of count. Define two threads and call synchronized methods getData() and putData() respectively.

```
class shareData
{
int count,d;
synchronized void getdata(int c)
{
try
{
Thread.sleep(2000);
}
catch(Exception e)
{
System.out.println("I/O Error");
}
d=c;
}

synchronized void putdata()
{
try
{
Thread.sleep(500);
}
catch(Exception e)
```

```
{
System.out.println("I/O Error");
}
System.out.println("Value of count= "+d);
}
}

class t1 implements Runnable
{
int count;
shareData x;
t1(shareData y, int c)
{
x=y;
count=c;
Thread t3=new Thread(this);
t3.start();
}

public void run()
{
x.getdata(count);
}
}

class t2 implements Runnable
{
Thread t4;
shareData x;
t2(shareData y)
{
x=y;
t4=new Thread(this);
t4.start();
}
public void run()
{
x.putdata();
}
}

class syncmethod
{
public static void main(String args[])
{
shareData s=new shareData();
t1 s1=new t1(s,10);
t2 s2=new t2(s);
}  }
```

Output:

Value of count= 10