UNIT-III: **15 hrs**

Exception: Introduction, Types, Exception Handling Techniques-try, catch, multiple catch, User-Defined Exception.

Multithreading: Introduction, Main Thread and Creation of New Threads –By Inheriting the Thread Class, Thread Lifecycle, Thread Priority.

Input/Output: Introduction, java.io Package, Reading and Writing Data- Reading/Writing Console User Input, Scanner Class, Reading/Writing Buffered Byte Stream Classes-Buffered Input Stream Class, Buffered Output Stream Class.

Types of errors:

Errors are the mistakes which make the program go wrong.

13.2 Types of Errors

Errors may broadly be classified into two categories:

- Compile-time errors
- Run-time errors

Compile-Time Errors

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 that we fix all the errors before we can successfully compile and run the program.

```
/* This program contains an error */
class Error1
{
        public static void main(String args[])
        {
            System.out.println("Hello Java!") // Missing;
        }
}
```

The Java compiler does a nice job of telling us where the errors are in the program. For example, if we have missed the semicolon at the end of print statement in Program 13.1, the following message will be displayed in the screen:

```
Error1.java :7: ';' expected
System.out.println ("Hello Java!")
^
1 error
```

We can now go to the appropriate line, correct the error, and recompile the program. Sometimes, a

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.

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

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
- And may more

Program 13.2 ***Illustration of run-time errors***

```
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);
      }
}
```

When Java run-time tries to execute a division by zero, it generates an error condition, which causes the program to stop after displaying an appropriate message.

```
java.lang.ArithmeticException: / by zero
     at Error2.main(Error2.java:10)
```

Exceptions or Pre-defined exceptions:

13.3 Exceptions

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 (i.e. informs us that an error has occurred).

If the exception object is not caught and handled properly, the interpreter will display an error message as shown in the output of Program 13.2 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*.

Types of exceptions

There are two types of exceptions in Java:

1)Checked exceptions

2)Unchecked exceptions

Checked exceptions

All exceptions other than Runtime Exceptions are known as Checked exceptions as the compiler checks them during compilation to see whether the programmer has handled them or not. If these exceptions are not handled/declared in the program, you will get

compilation error. For example, SQLException, IOException, ClassNotFoundException etc.

Unchecked Exceptions

Runtime Exceptions are also known as Unchecked Exceptions. These exceptions are not checked at compile-time so compiler does not check whether the programmer has handled them or not but it's the responsibility of the programmer to handle these exceptions and provide a safe exit. For example, ArithmeticException, NullPointerException, ArrayIndexOutOfBoundsException etc.

Error handling performs the following tasks:

1. Find the problem (***Hit*** the exception).
2. Inform that an error has occurred (***Throw*** the exception)
3. Receive the error information (***Catch*** the exception)
4. Take corrective actions (***Handle*** the exception)

The error handling code basically consists of two segments, one to detect errors and to throw exceptions and the other to catch exceptions and to take appropriate actions.

Table 13.1 Common Java Exceptions

Exception Type	*Cause of Exception*
ArithmeticException	Caused by math errors such as division by zero
ArrayIndexOutOfBoundsException	Caused by bad array indexes
ArrayStoreException	Caused when a program tries to store the wrong type of data in an array
FileNotFoundException	Caused by an attempt to access a nonexistent file
IOException	Caused by general I/O failures, such as inability to read from a file
NullPointerException	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

Syntax of Exception handling code(try catch in java)

```
try
{
     //statements that may cause an exception
}
catch (exception(type) e(object))
{
     //error handling code
```

```
}
```

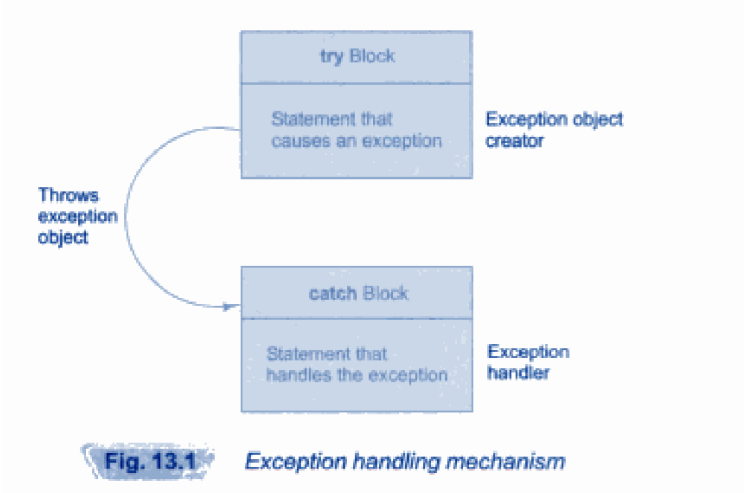


Fig. 13.1 *Exception handling mechanism*

Java uses a keyword **try** to preface a block of code that is likely to cause an error condition and "throw" an exception. A catch block defined by the keyword **catch** "catches" the exception "thrown" by thee try block and handles it appropriately. The catch block is added immediately after the try block.

Try block

The try block contains set of statements where an exception can occur. A try block is always followed by a catch block, which handles the exception that occurs in associated try block. A try block must be followed by catch blocks or finally block or both.
While writing a program, if you think that certain statements in a program can throw a exception, enclosed them in try block and handle that exception

Catch block

A catch block is where you handle the exceptions, this block must follow the try block. A single try block can have several catch blocks associated with it. You can catch different exceptions in different catch blocks. When an exception occurs in try block, the corresponding catch block that handles that particular exception executes. For example if an arithmetic exception occurs in try block then the statements enclosed in catch block for arithmetic exception executes.

Eg for try…catch block..(ArithmeticException or divideby zero exception)

```
Class DemoException
{
public static void main(String args[])
{
int a,b,c;
try
{
a=Integer.parseInt(args[0]);
b=Integer.parseInt(args[1]);
```

```
c=a/b;
System.out.println("c="+c));
}
Catch(ArithmeticException e)
{
System.out.println("cannot divide a number by 0");
}
System.out.println("out of try-catch- blocks"));
}
}
```

o/p:

```
>javac  DemoException.java
>java DemoException 5  5
C=1
out of try-catch blocks
>javac DemoException.java
>java DemoException 5  0
```

Cannot divide a number by 0

out of try-catch blocks

Multiple catch statements in Java

- A single try block can have any number of catch blocks.

- A generic catch block can handle all the exceptions. Whether it is ArrayIndexOutOfBoundsException or ArithmeticException or NullPointerException or any other type of exception, this handles all of them.

```
catch(Exception e){
  //This catch block catches all the exceptions
}
```

    - In generic exception handler you can display a message but you are not sure for which type of exception it may trigger so it will display the same message for all the exceptions and user may not be able to understand which exception occurred. Thats the reason you should place is at the end of all the specific exception catch blocks
    - If no exception occurs in try block then the catch blocks are completely ignored.

- Corresponding catch blocks execute for that specific type of exception:
  catch(ArithmeticException e) is a catch block that can hanlde ArithmeticException
  catch(NullPointerException e) is a catch block that can handle NullPointerException

Example of Multiple catch blocks

```
class ExceptionDemo
{
public static void main(String args[])
{
int a[]=new int[4],b,c;
for(int i=0;i<3;i++)
a[i]=Integer.parseInt(args[i]);
b=Integer.parseInt(args[0]);
try
{
for(int i=0;i<3;i++)
{
c=a[i]/b;
System.out.println(c);
}
}
catch(ArithmeticException e)
{
System.out.println(e);
}
catch(NumberFormatException e)
{
System.out.println(e);
}
catch(ArrayIndexOutOfBoundsException e)
{
System.out.println("IndexOutOfBoundsException");
}
catch(Exception e)
{
System.out.println(e);
}
System.out.println("demo exception");
```

```
}
}
(Or)
//exception program; try,catch block; multiple exceptions;
command line arguments
class DemoException
{
public static void main(String args[])
{
int a,b,c;
int d[]={10,20,30};
a=Integer.parseInt(args[0]);
b=Integer.parseInt(args[1]);
int i=Integer.parseInt(args[2]);
try
{
c=a/b;
System.out.println("division:"+c);
System.out.println(d[i]);
}
catch(ArithmeticException e)
{
System.out.println("b cannot be zero"+e);
}
catch(ArrayIndexOutOfBoundsException e)
{
System.out.println("trying to access an element out of the
boundary");
}
catch(Exception e)
{
System.out.println(e);
}
c=a+b;
System.out.println("add:"+c);
c=a-b;
System.out.println("sub:"+c);
c=a*b;
System.out.println("mult:"+c);
}
```

```
}
```

Finally statement

A **finally block** contains all the crucial statements that must be executed whether exception occurs or not. The statements present in this block will always execute regardless of whether exception occurs in try block or not such as closing a connection, stream etc.

Syntax of Finally block

```
try {
    //Statements that may cause an exception
}
catch {
   //Handling exception
}
finally {
   //Statements to be executed
}
```

Example for finally block:

```
Class DemoException
{
public static void main(String args[])
{
int a,b,c;
try
{
a=Integer.parseInt(args[0]);
b=Integer.parseInt(args[1]);
c=a/b;
System.out.println("c="+c));
}
Catch(ArithmeticException e)
{
System.out.println("cannot divide a number by 0");
}
Finally
{
System.out.println("This is finally block");
}
System.out.println("out of try-catch-finally blocks"));
}
}
```

o/p:

>javac DemoException.java

>java DemoException 5 5

C=1

This is finally block

Out of try-catch-finally blocks

>javac DemoException.java

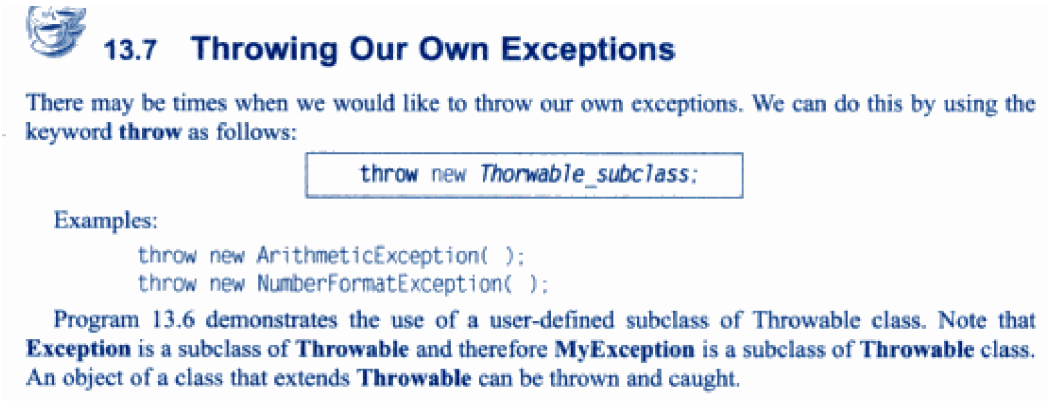
>java DemoException 5 0

Cannot divide a number by 0

This is finally block

Out of try-catch-finally blocks

Throwing an exception(how to throw user defined exceptions):



13.7 Throwing Our Own Exceptions

There may be times when we would like to throw our own exceptions. We can do this by using the keyword **throw** as follows:

```
throw new Thorwable_subclass;
```

Examples:

```
throw new ArithmeticException( );
throw new NumberFormatException( );
```

Program 13.6 demonstrates the use of a user-defined subclass of Throwable class. Note that **Exception** is a subclass of **Throwable** and therefore **MyException** is a subclass of **Throwable** class. An object of a class that extends **Throwable** can be thrown and caught.

```
class CheckMarks extends Exception
{
CheckMarks()
{
System.out.println("marks out of bound");
}
}
class MException
{
public static void main(String args[])
{
int m1=Integer.parseInt(args[0]);
int m2=Integer.parseInt(args[1]);
int m3=Integer.parseInt(args[2]);
try
{
```

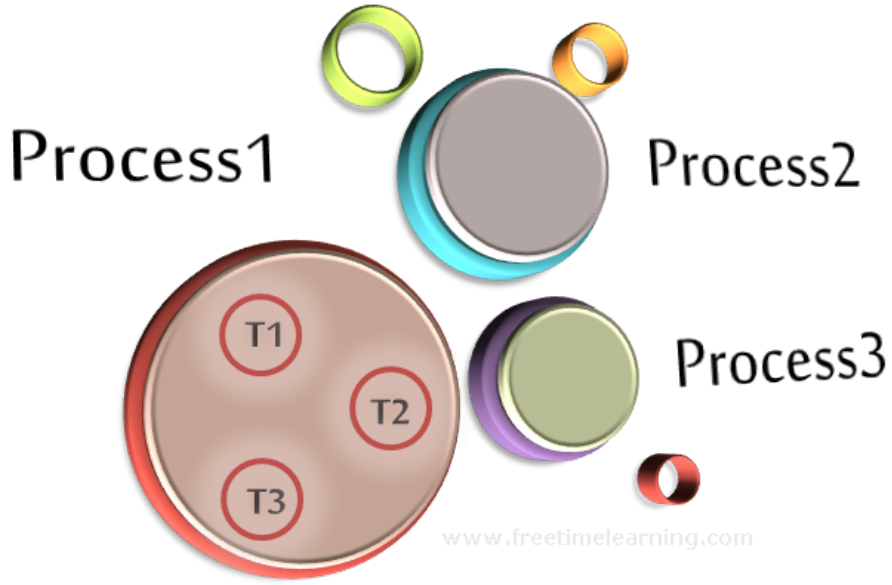
```
if(m1>100 || m2>100 || m3>100)
{
throw new CheckMarks();
}
int s=m1+m2+m3;
System.out.println("total="+s);
}
catch(Exception e)
{
System.out.println(e);
}
}
}
```

Multi Threading:

1) The earlier days the computers memory is occupied only one program after completion of one program it is possible to execute another program is called uni programming.

2) Whenever one program execution is completed then only second program execution will be started such type of execution is called co operative execution, this execution we are having lot of disadvantages.

a. Most of the times memory will be wasted.
b. CPU utilization will be reduced because only program allow executing at a time.
c. The program queue is developed on the basis co operative execution



multiprogramming
1) Multiprogramming means executing the more than one program at a time.

2) All these programs are controlled by the CPU scheduler.

3) CPU scheduler will allocate a particular time period for each and every program.

4) Executing several programs simultaneously is called multiprogramming.

5) In multiprogramming a program can be entered in different states.
a. Ready state.
b. Running state.
c. Waiting state.

6) Multiprogramming mainly focuses on the number of programs.
Advantages of multiprogramming:-
1. CPU utilization will be increased.
2. Execution speed will be increased and response time will be decreased.
3. CPU resources are not wasted.

Create Thread
1) Thread is nothing but separate path of sequential execution.

2) The independent execution technical name is called thread.

3) Whenever different parts of the program executed simultaneously that each and every part is called thread.

4) The thread is light weight process because whenever we are creating thread it is not occupying the separate memory it uses the same memory. Whenever the memory is shared means it is not consuming more memory.

5) Executing more than one thread a time is called multithreading.

//Single threaded program

```
class Test
{
public static void main(String[] args)
{
System.out.println("Hello World!");
System.out.println("hi freetimelearn");
}
}
```

Output :
Hello World!
hi freetimelearn

//Thread Information

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

```
import java.util.*;
class Test
{
public static void main(String[] args)
{
Thread t=Thread.currentThread();
System.out.println("currrent thread information is : "+t);
System.out.println("currrent thread priority is : "+t.getPriority());
System.out.println("currrent thread name is : "+t.getName());
}
}
```

Output :

currrent thread information is : Thread[main,5,main] currrent thread priority is : 5 currrent thread name is : main

The main important application areas of the multithreading are

1. Developing video games

2. Implementing multimedia graphics.

3. Developing animations

There are two different ways to create a thread is available

1) Create class that extending standard java.lang.Thread Class
2) Create class that Implementing java.lang.Runnable interface

First approach to create thread extending Thread class:-

Step 1:-

Creates a class that is extend by Thread classes and override the run() method

```
class MyThread extends Thread
{
public void run()
{
System.out.println("business logic of the thread");
System.out.println("body of the thread");
}
};
```

Step 2:-

Create a Thread object

```
MyThread t=new MyThread();
```

Step 3:-

Starts the execution of a thread.

```
t.start();
```

//Thread program

```
class MyThread extends Thread
```

```
{
public void run()
{
System.out.println("freetimelearn");
System.out.println("body of the thread");
}
};
class ThreadDemo
{
public static void main(String[] args)
{
MyThread t=new MyThread();
t.start();
}
}
```

Output :

freetimelearn
body of the thread

Note :-

1) Whenever we are calling t.start() method the JVM search for the start() in the MyThread class but the start() method is not present in the MyThread class so JVM goes to parent class called Thread class and search for the start() method.
2) In the Thread class start() method is available hence JVM is executing start() method.
3) Whenever the thread class start() that start() is responsible person to call run() method.
4) Finally the run() automatically executed whenever we are calling start() method.
5) Whenever we are giving a chance to the Thread class start() method then only a new thread will be created.

Life Cycle of Thread in Java | Thread State

Life Cycle of Thread in Java is basically state transitions of a thread that starts from its birth and ends on its death.

When an instance of a thread is created and is executed by calling start() method of *Thread class*, the thread goes into runnable state.

When sleep() or wait() method is called by Thread class, the thread enters into non-runnable state.

From non-runnable state, thread comes back to runnable state and continues execution of statements.

When the thread comes out of run() method, it dies. These state transitions of a thread are called **Thread life cycle in Java**.

To work with threads in a program, it is important to identify thread state. So. let's understand how to identify thread states in Java thread life cycle.

Thread States in Java

A thread is a path of execution in a program that enters any one of the following five states during its life cycle. The five states are as follows:

1. New

2. Runnable

3. Running

4. Blocked (Non-runnable state)

5. Dead

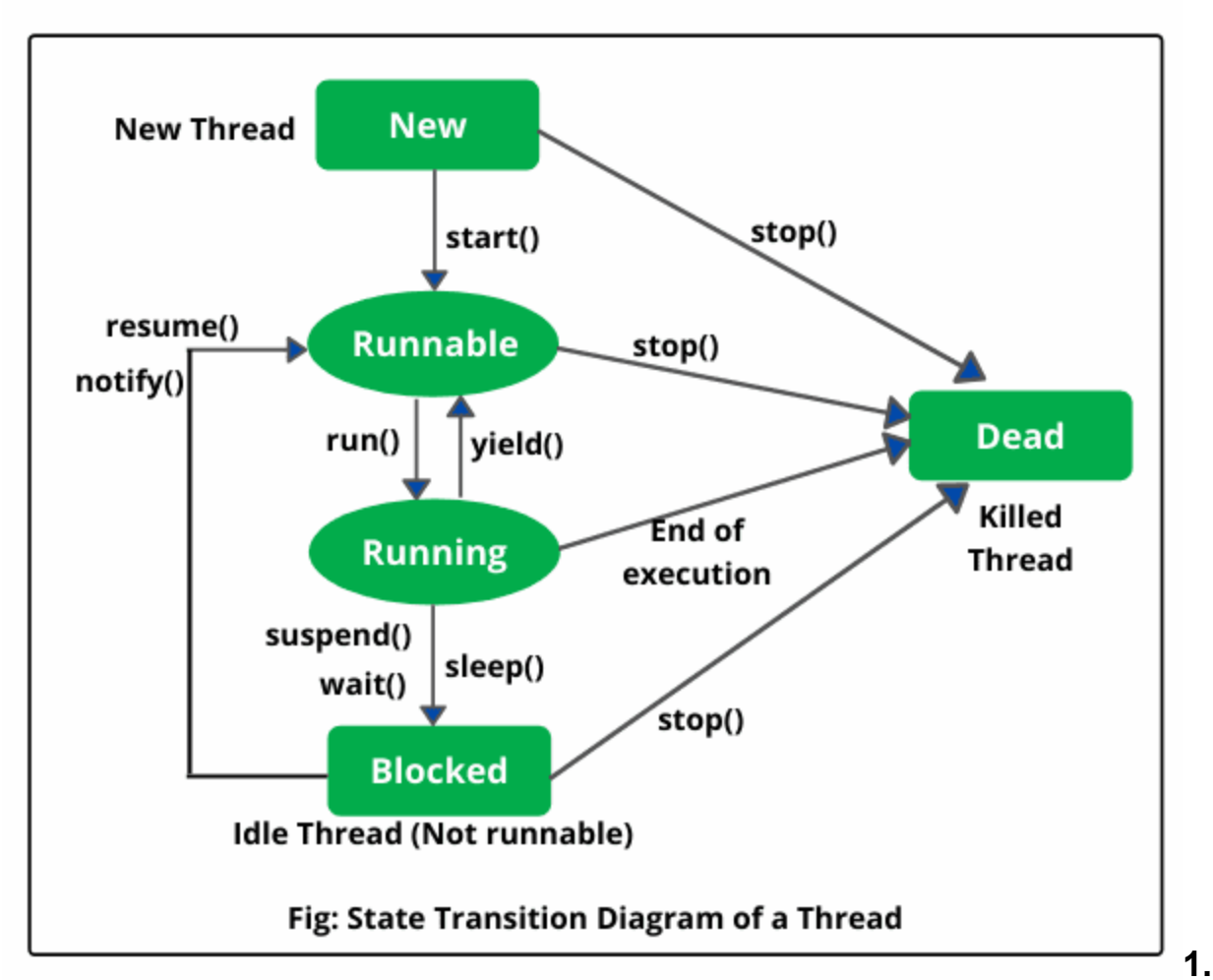
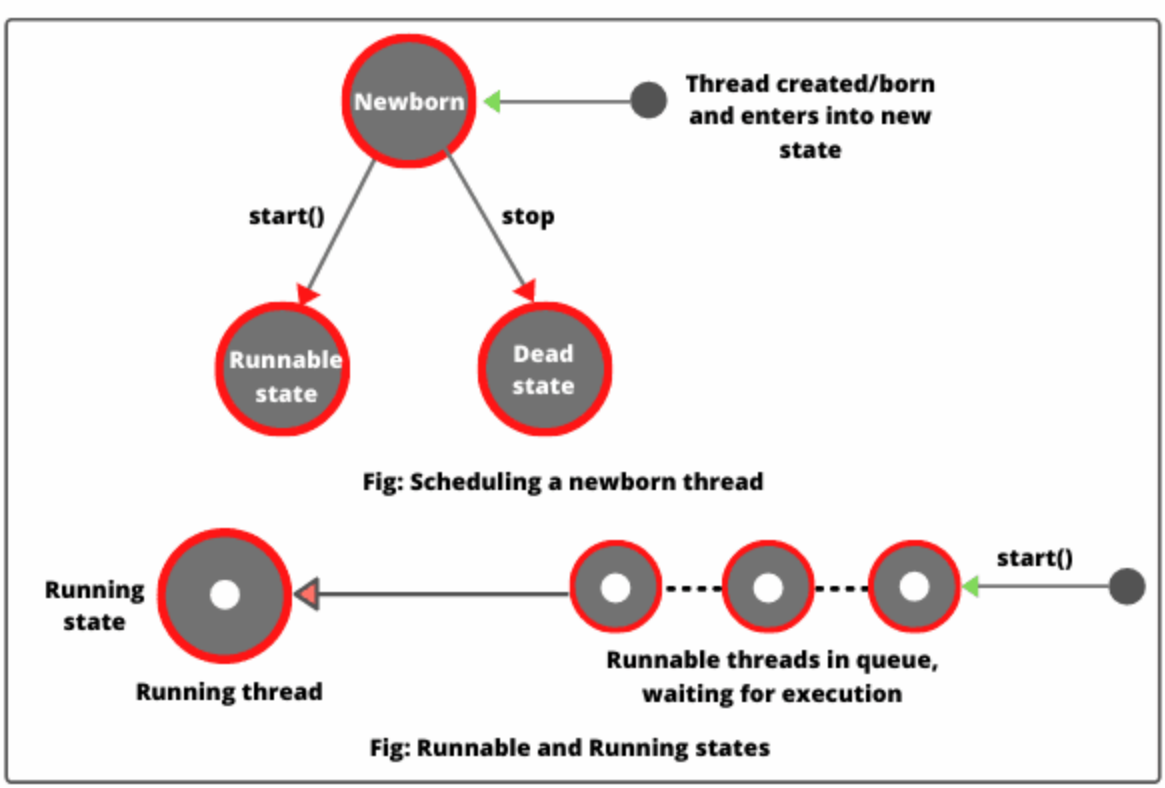


Fig: State Transition Diagram of a Thread

1.

New (Newborn State): When we create a thread object using Thread class, thread is born and is known to be in Newborn state. That is, when a thread is born, it enters into new state but the start() method has not been called yet on

the instance.



In other words, Thread object exists but it cannot execute any statement because it is not an execution of thread. Only start() method can be called on a new thread; otherwise, an **IllegalThreadStateException** will be thrown.

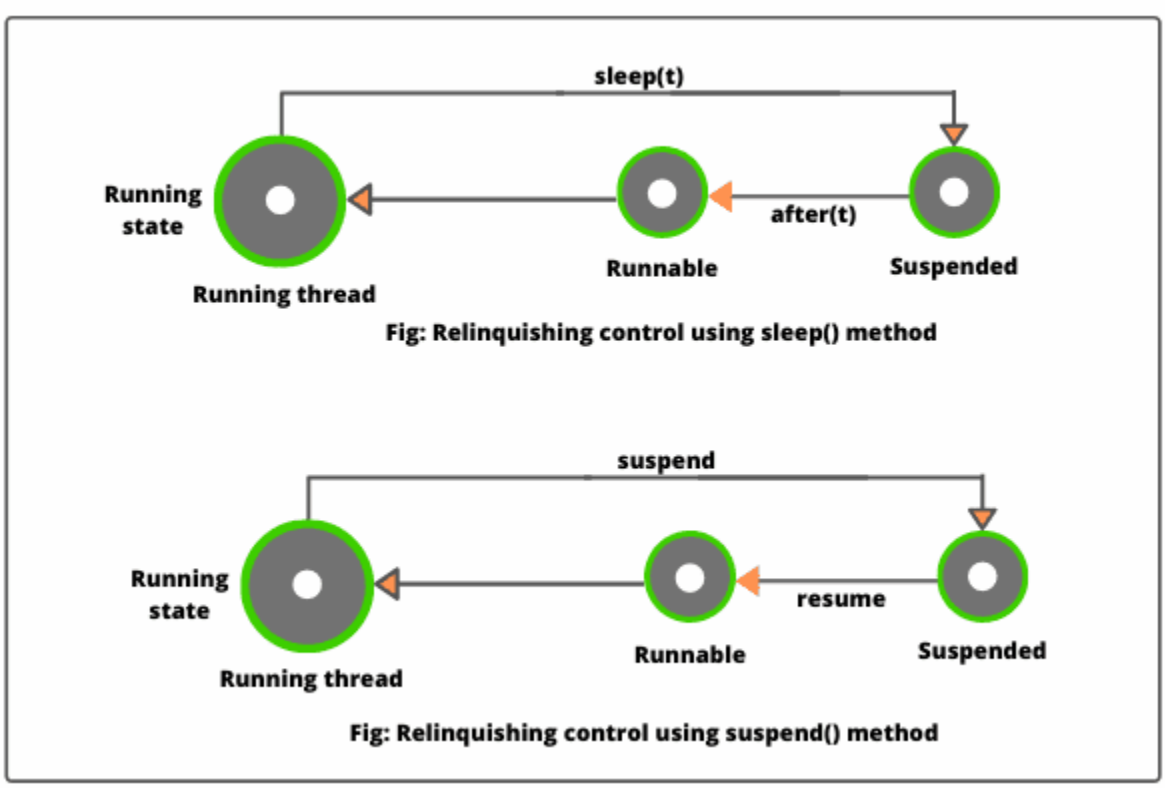
2. Runnable state: Runnable state means a thread is ready for execution. When the start() method is called on a new thread, thread enters into a runnable state.

In runnable state, thread is ready for execution and is waiting for availability of the processor (CPU time). That is, thread has joined queue (line) of threads that are waiting for execution.
If all threads have equal priority, CPU allocates time slots for thread execution on the basis of first-come, first-serve manner. The process of allocating time to threads is known as **time slicing**. A thread can come into runnable state from running, waiting, or new states.

3. Running state: Running means Processor (CPU) has allocated time slot to thread for its execution. When thread scheduler selects a thread from the runnable state for execution, it goes into running state. Look at the above figure.

In running state, processor gives its time to the thread for execution and executes its run method. This is the state where thread performs its actual functions. A thread can come into running state only from runnable state.

A running thread may give up its control in any one of the following situations and can enter into the blocked state.



a) When sleep() method is invoked on a thread to sleep for specified time period, the thread is out of queue during this time period. The thread again reenters into the runnable state as soon as this time period is elapsed.

b) When a thread is suspended using suspend() method for some time in order to satisfy some conditions. A suspended thread can be revived by using resume() method.

c) When wait() method is called on a thread to wait for some time. The thread in wait state can be run again using notify() or notifyAll() method.

4. Blocked state: A thread is considered to be in the blocked state when it is suspended, sleeping, or waiting for some time in order to satisfy some condition.

5. Dead state: A thread dies or moves into dead state automatically when its run() method completes the execution of statements. That is, a thread is terminated or dead when a thread comes out of run() method. A thread can also be dead when the stop() method is called.

During the life cycle of thread in Java, a thread moves from one state to another state in a variety of ways. This is because in multithreading environment, when multiple threads are executing, only one thread can use CPU at a time.

All other threads live in some other states, either waiting for their turn on CPU or waiting for satisfying some conditions. Therefore, a thread is always in any of the five states.

Example program: copy from running notes

Thread Priorities

1. Every Thread in java has some property. It may be default priority provided be the JVM or customized priority provided by the programmer.

2. The valid range of thread priorities is 1 10. Where one is lowest priority and 10 is highest priority.

3. The default priority of main thread is 5. The priority of child thread is inherited from the parent.

4. Thread defines the following constants to represent some standard priorities.

5. Thread Scheduler will use priorities while allocating processor the thread which is having highest priority will get chance first and the thread which is having low priority.

6. If two threads having the same priority then we can't expect exact execution order it depends upon Thread Scheduler.

7. The thread which is having low priority has to wait until completion of high priority threads.

8. Three constant values for the thread priority.
a. MIN_PRIORITY = 1
b. NORM_PRIORITY = 5
c. MAX_PRIORITY = 10
Thread class defines the following methods to get and set priority of a Thread.
a. Public final int getPriority()
b. Public final void setPriority(int priority)
Here priority indicates a number which is in the allowed range of 1 - 10. Otherwise we will get Runtime exception saying IllegalArgumentException.
Thread Priorities program

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

```
class frist extends Thread
{
public void run()
{
System.out.println(" frist thread stated");
for (int i=0;i<10 ;i++ )
{
System.out.println(i);
}
System.out.println("frist thread ended");
}
};
class secound extends Thread
{
public void run()
{
System.out.println("Secound thread ");
for (int i=0;i<10 ;i++ )
{
System.out.println(i);
}
System.out.println("Secound thread ended");
}
};
class threed extends Thread
{
public void run()
{
System.out.println("3th thread");
for (int i=0;i<10 ;i++ )
{
System.out.println(i);
}
System.out.println("3th thread ended");
}
};
class Test
{
public static void main(String[] durga)
{
frist thread1=new frist();
secound thread2=new secound();
threed thread3=new threed();
thread1.setPriority(Thread.MAX_PRIORITY);
thread2.setPriority(Thread.MIN_PRIORITY);
thread3.setPriority(thread2.getPriority()+1);
System.out.println("frist Thread");
thread1.start();
System.out.println("secound Thread");
thread2.start();
```

```
System.out.println("3th Thread");
thread3.start();
}
};
```

Output :

```
frist Thread
secound Thread
3th Thread
Secound thread
0
1
2
3
4
5
6
7
8
9
Secound thread ended
frist thread stated
0
1
2
3
4
5
6
7
8
9
frist thread ended
3th thread
0
1
2
3
4
5
6
7
8
9
3th thread ended
```

Java Console Class

The Java Console class is be used to get input from console. It provides methods to read texts and passwords.

If you read password using Console class, it will not be displayed to the user.

The java.io.Console class is attached with system console internally. The Console class is introduced since 1.5.

Let's see a simple example to read text from console.

```
String text=System.console().readLine();

System.out.println("Text is: "+text);
```

Java Console class declaration

Let's see the declaration for Java.io.Console class:

```
public final class Console extends Object implements Flushable
```

Java Console class methods

Method	Description
Reader reader()	It is used to retrieve the reader object associated with the console
String readLine()	It is used to read a single line of text from the console.
String readLine(String fmt, Object... args)	It provides a formatted prompt then reads the single line of text from the console.
char[] readPassword()	It is used to read password that is not being displayed on the console.
char[] readPassword(String fmt, Object... args)	It provides a formatted prompt then reads the password that is not being displayed on the console.
Console printf(String format, Object... args)	It is used to write a string to the console output stream.
PrintWriter writer()	It is used to retrieve the PrintWriter object associated with the console.
void flush()	It is used to flushes the console.

How to get the object of Console

System class provides a static method console() that returns the singleton instance of Console class.

```
public static Console console(){}
```

Let's see the code to get the instance of Console class.

```
Console c=System.console();
```

Java Console Example

```
import java.io.Console;
class ReadStringTest{
public static void main(String args[]){
Console c=System.console();
System.out.println("Enter your name: ");
String n=c.readLine();
System.out.println("Welcome "+n);
System.out.println("Enter password: ");
char[] ch=c.readPassword();
String pass=String.valueOf(ch);//converting char array into string
System.out.println("Password is: "+pass);
}}
```

Output:

```
Enter your name: Nakul Jain
Welcome Nakul Jain
```

Enter password:

Password is: 123

Java Scanner

Scanner class in Java is found in the java.util package. Java provides various ways to read input from the keyboard, the java.util.Scanner class is one of them.

The Java Scanner class breaks the input into tokens using a delimiter which is whitespace by default. It provides many methods to read and parse various primitive values.

The Java Scanner class is widely used to parse text for strings and primitive types using a regular expression. It is the simplest way to get input in Java. By the help of Scanner in Java, we can get input from the user in primitive types such as int, long, double, byte, float, short, etc.

The Java Scanner class extends Object class and implements Iterator and Closeable interfaces.

The Java Scanner class provides nextXXX() methods to return the type of value such as nextInt(), nextByte(), nextShort(), next(), nextLine(), nextDouble(), nextFloat(), nextBoolean(), etc. To get a single character from the scanner, you can call next().charAt(0) method which returns a single character.

How to get Java Scanner

To get the instance of Java Scanner which reads input from the user, we need to pass the input stream (System.in) in the constructor of Scanner class. For Example:

```
Scanner in = new Scanner(System.in);
```

To get the instance of Java Scanner which parses the strings, we need to pass the strings in the constructor of Scanner class. For Example:

How to get Java Scanner

To get the instance of Java Scanner which reads input from the user, we need to pass the input stream (System.in) in the constructor of Scanner class. For Example:

```
Scanner in = new Scanner(System.in);
```

To get the instance of Java Scanner which parses the strings, we need to pass the strings in the constructor of Scanner class. For Example:

Java Scanner Class Methods

The following are the list of Scanner methods:

SN	Modifier	Method	Description

	& Type		
1)	void	close()	It is used to close this scanner.
4)	String	findInLine()	It is used to find the next occurrence of a pattern constructed from the specified string, ignoring delimiters.
5)	string	findWithinHorizon()	It is used to find the next occurrence of a pattern constructed from the specified string, ignoring delimiters.
6)	boolean	hasNext()	It returns true if this scanner has another token in its input.
23)	boolean	nextBoolean()	It scans the next token of the input into a boolean value and returns that value.
24)	byte	nextByte()	It scans the next token of the input as a byte.
25)	double	nextDouble()	It scans the next token of the input as a double.
26)	float	nextFloat()	It scans the next token of the input as a float.
27)	int	nextInt()	It scans the next token of the input as an Int.
28)	String	nextLine()	It is used to get the input string that was skipped of the Scanner object.

29)	long	nextLong()	It scans the next token of the input as a long

Example 1

Let's see a simple example of Java Scanner where we are getting a single input from the user. Here, we are asking for a string through in.nextLine() method.

```
import java.util.*;

public class ScannerExample {

public static void main(String args[]){
```

```
        Scanner in = new Scanner(System.in);
        System.out.print("Enter your name: ");
        String name = in.nextLine();
        System.out.println("Name is: " + name);
        in.close();
        }
}
```

Output:

```
Enter your name: sonoo jaiswal
Name is: sonoo jaiswal
```

Example 2

```
import java.util.*;
public class ScannerClassExample1 {
    public static void main(String args[]){
        String s = "Hello, This is JavaTpoint.";
        //Create scanner Object and pass string in it
        Scanner scan = new Scanner(s);
        //Check if the scanner has a token
```

```
        System.out.println("Boolean Result: " + scan.hasNext());
        //Print the string
        System.out.println("String: " +scan.nextLine());
        scan.close();
        System.out.println("--------Enter Your Details-------- ");
        Scanner in = new Scanner(System.in);
        System.out.print("Enter your name: ");
        String name = in.next();
        System.out.println("Name: " + name);
        System.out.print("Enter your age: ");
        int i = in.nextInt();
        System.out.println("Age: " + i);
        System.out.print("Enter your salary: ");
        double d = in.nextDouble();
        System.out.println("Salary: " + d);
        in.close();
        }
}
```

Output:

```
Boolean Result: true
String: Hello, This is JavaTpoint.
-------Enter Your Details---------
Enter your name: Abhishek
Name: Abhishek
```

```
Enter your age: 23

Age: 23

Enter your salary: 25000

Salary: 25000.0
```

Class BufferedInputStream
java.lang.Object

java.io.InputStream

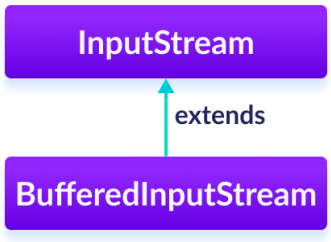
java.io.FilterInputStream

java.io.BufferedInputStream

Java BufferedInputStream Class

The `BufferedInputStream` class of the `java.io` package is used with other input streams to read the data (in bytes) more efficiently.

It extends the `InputStream` abstract class.



Working of BufferedInputStream

The `BufferedInputStream` maintains an internal **buffer of 8192 bytes**.

During the read operation in `BufferedInputStream`, a chunk of bytes is read from the disk and stored in the internal buffer. And from the internal buffer bytes are read individually.

Hence, the number of communication to the disk is reduced. This is why reading bytes is faster using the `BufferedInputStream`.

Create a BufferedInputStream

In order to create a `BufferedInputStream`, we must import the `java.io.BufferedInputStream` package first. Once we import the package here is how we can create the input stream.

```
// Creates a FileInputStream

FileInputStream file = new FileInputStream(String path);


// Creates a BufferedInputStream

BufferedInputStream buffer = new BufferInputStream(file);
```

In the above example, we have created a `BufferdInputStream` named `buffer` with the `FileInputStream` named `file`.

Here, the internal buffer has the default size of 8192 bytes. However, we can specify the size of the internal buffer as well.

```
// Creates a BufferedInputStream with specified size internal buffer

BufferedInputStream buffer = new BufferInputStream(file, int size);
```

The `buffer` will help to read bytes from the files more quickly.

Let's try to read the file using `BufferedInputStream`.

```
import java.io.BufferedInputStream;
import java.io.FileInputStream;

class Main {
  public static void main(String[] args) {
    try {

      // Creates a FileInputStream
      FileInputStream file = new FileInputStream("input.txt");
```

```
        // Creates a BufferedInputStream
        BufferedInputStream input = new BufferedInputStream(file);
// Returns the available number of bytes
      System.out.println("Available bytes at the beginning: " +
buffer.available());

        // Reads first byte from file
        int i = input .read();

        while (i != -1) {
          System.out.print((char) i);

          // Reads next byte from the file
          i = input.read();
        }
        input.close();
      }

      catch (Exception e) {
        e.getStackTrace();
      }
  }
}
```

Output

```
Available bytes at the beginning: 39

This is a line of text inside the file.
```

In the above example, we have created a buffered input stream named *`buffer`* along with `FileInputStream`. The input stream is linked with the file **input.txt**.

```
FileInputStream file = new FileInputStream("input.txt");
BufferedInputStream buffer = new BufferedInputStream(file);
```

Here, we have used the `read()` method to read an array of bytes from the internal buffer of the buffered reader.

Java BufferedInputStream Class

Java BufferedInputStream class is used to read information from stream. It internally uses buffer mechanism to make the performance fast.

The important points about BufferedInputStream are:

When the bytes from the stream are skipped or read, the internal buffer automatically refilled from the contained input stream, many bytes at a time.

When a BufferedInputStream is created, an internal buffer array is created.

Java BufferedInputStream class declaration

Let's see the declaration for Java.io.BufferedInputStream class:

```
public class BufferedInputStream extends FilterInputStream
```

Java BufferedInputStream class constructors

Constructor	Description
BufferedInputStream(InputStream IS)	It creates the BufferedInputStream and saves it argument, the input stream IS, for later use.

BufferedInputStream(InputStream IS, int size)	It creates the BufferedInputStream with a specified buffer size and saves it argument, the input stream IS, for later use.

Java BufferedInputStream class methods

Method	Description
int available()	It returns an estimate number of bytes that can be read from the input stream without blocking by the next invocation method for the input stream.
int read()	It read the next byte of data from the input stream.
int read(byte[] b, int off, int ln)	It read the bytes from the specified byte-input stream into a specified byte array, starting with the given offset.

void close()	It closes the input stream and releases any of the system resources associated with the stream.
void reset()	It repositions the stream at a position the mark method was last called on this input stream.

Example of Java BufferedInputStream

Let's see the simple example to read data of file using BufferedInputStream:

```
  import java.io.*;
public class BufferedInputStreamExample{
 public static void main(String args[]){
  try{
    FileInputStream fin=new FileInputStream("D:\\testout.txt");

    BufferedInputStream bin=new BufferedInputStream(fin);
    int i;
    while((i=bin.read())!=-1){
     System.out.print((char)i);
    }
    bin.close();
    fin.close();
  }catch(Exception e){System.out.println(e);}
 }   }
```

Here, we are assuming that you have following data in **"testout.txt"** file:

Hello students

Output:

Hello students