

Unit:3 (Exception handling ,Threading)

Exception Handling(try , catch , throw , throws , finally)

Creating user defined exception class

Multi-Threading

Thread and it's lifecycle

Thread class and it's methods

Synchronization in multiple Thread

Deamon Thread and Non-Deamon Thread

Exception Handling in Java

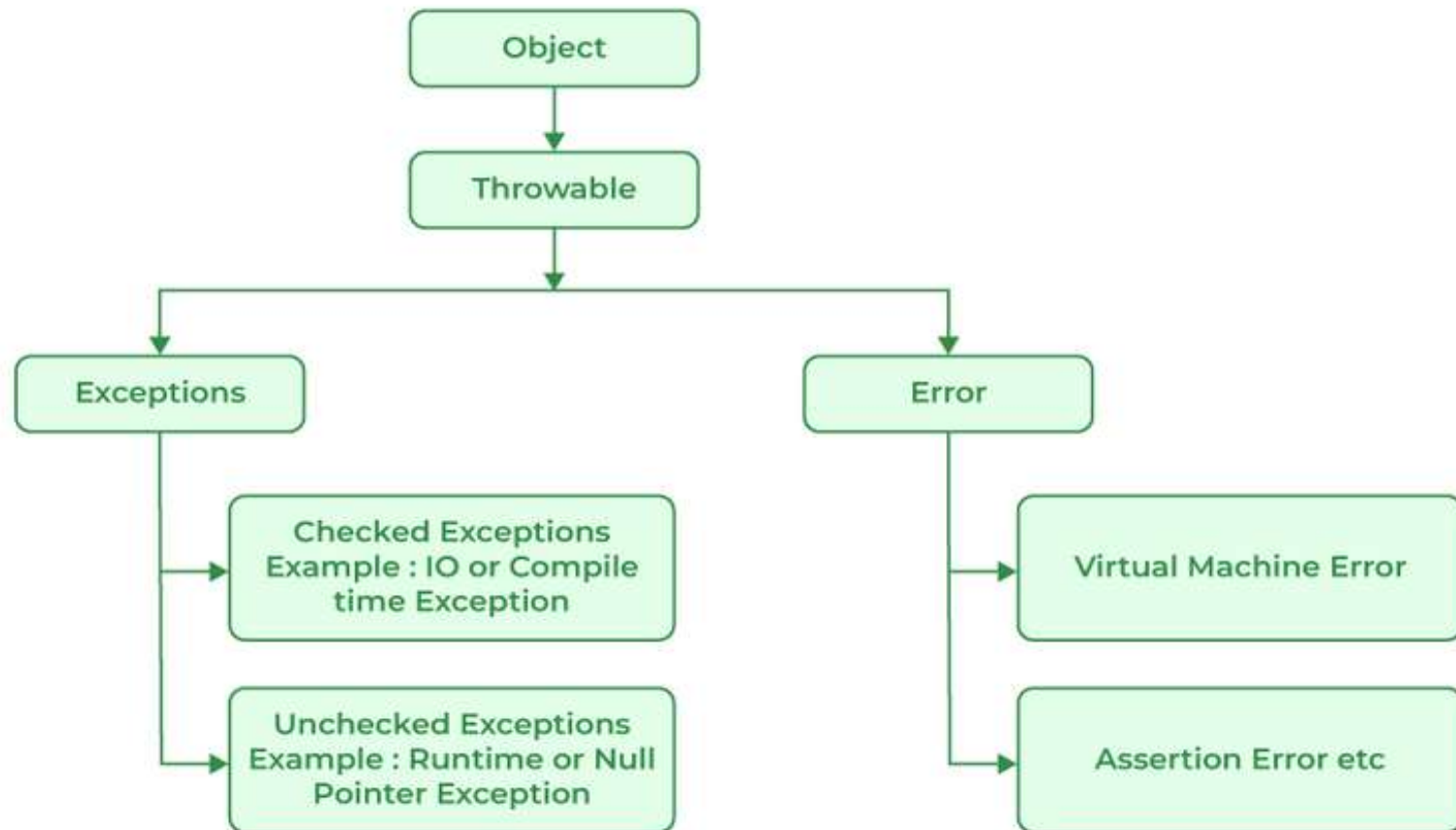
- Dictionary Meaning: Exception is an abnormal condition.
- The Exception Handling in Java is one of the powerful mechanism to handle the runtime errors so that the normal flow of the application can be maintained. It is an object which is thrown at runtime.
- It handle runtime errors such as `ClassNotFoundException`, `IOException`, `SQLException`, `RemoteException`, etc.
- The core advantage of exception handling is to maintain the normal flow of the application

Difference between Exception and Error

S.No	Errors	Exceptions
1.	Errors primarily arise due to the lack of system resources.	Exceptions may occur during both runtime and compile time.
2.	Recovery from an error is typically not possible.	Recovery from an exception is possible.
3.	All errors in Java are unchecked.	Exceptions in Java can be either checked or unchecked.
4.	The system executing the program is responsible for errors.	The program's code is responsible for exceptions.
5.	Errors are defined in the <code>java.lang.Error</code> package.	Exceptions are defined in the <code>java.lang.Exception</code> package.

Hierarchy of Java Exception classes

The `java.lang.Throwable` class is the root class of Java Exception hierarchy inherited by two subclasses: `Exception` and `Error`. The hierarchy of Java Exception classes is given below:




Types of Java Exceptions

- There are mainly two types of exceptions: checked and unchecked. An error is considered as the unchecked exception. However, according to Oracle, there are three types of exceptions namely:
 1. Checked Exception
 2. Unchecked Exception
 3. Error

1) Checked Exception

- The classes that directly inherit the Throwable class except RuntimeException and Error are known as checked exceptions. For example, IOException, SQLException, etc. Checked exceptions are checked at compile-time.

2) Unchecked Exception

- The classes that inherit the RuntimeException are known as unchecked exceptions. For example, ArithmeticException, NullPointerException, ArrayIndexOutOfBoundsException, etc. Unchecked exceptions are not checked at compile-time, but they are checked at runtime. 

3) Error

- Error is irrecoverable. Some example of errors are OutOfMemoryError, VirtualMachineError, AssertionError etc.

Java Exception Keywords

1) try

- Java **try** block is used to enclose the code that might throw an exception. It must be used within the method.
- If an exception occurs at the particular statement in the try block, the rest of the block code will not execute. So, it is recommended not to keep the code in try block that will not throw an exception.
- Java try block must be followed by either catch or finally block.
- Syntax:

```
try {  
    //code that may throw an exception }  
catch(Exception_class_Name ref) {}
```

2) catch

- Java catch block is used to handle the Exception by declaring the type of exception within the parameter. The declared exception must be the parent class exception (i.e., Exception) or the generated exception type. However, the good approach is to declare the generated type of exception.
- The catch block must be used after the try block only. You can use multiple catch block with a single try block.

Java Exception Handling Example

```
class ExceptionExample
{
    public static void main(String args[])
    {
        try
        {
            int a=90,b=0;
            int c=a/b;
            System.out.println(c);
        }
        catch(ArithmeticException e)
        {
            System.out.println(e);
        }
    }
}
```

```
    }
    //rest code of the program
    System.out.println("rest of the
code...");
}
}
```

Output:

```
java.lang.ArithmeticException: /
by zero
rest of the code..
```

Java Multi-catch block

- A try block can be followed by one or more catch blocks. Each catch block must contain a different exception handler. So, if you have to perform different tasks at the occurrence of different exceptions, use java multi-catch block.
- Points to remember
 1. At a time only one exception occurs and at a time only one catch block is executed.
 2. All catch blocks must be ordered from most specific to most general, i.e. catch for `ArithmeticException` must come before catch for `Exception`.

Example of java multi-catch block

```
public class MultipleCatchBlock
{
    public static void main(String[] args)
    {
        try {
            int a[]=new int[5];
            a[5]=30/0;
            System.out.println(a[10]);
        }
        catch(ArithmeticException e)
        {
            System.out.println("Arithmetic Exception occurs");
        }
        catch(ArrayIndexOutOfBoundsException e)
```

```
{
    System.out.println("ArrayIndexOut
OfBounds Exception occurs");
}
    catch(Exception e)
    {
        System.out.println("Parent Exception occurs");
    }
    System.out.println("multiple-catch
block");
}
}
```

output:

Arithmetic Exception occurs
multiple-catch block

Java Nested try block

- In Java, using a try block inside another try block is permitted. It is called as nested try block. Every statement that we enter a statement in try block, context of that exception is pushed onto the stack.
- For example, the inner try block can be used to handle `ArrayIndexOutOfBoundsException` while the outer try block can handle the `ArithmeticException` (division by zero).
- Why use nested try block
- Sometimes a situation may arise where a part of a block may cause one error and the entire block itself may cause another error. In such cases, exception handlers have to be nested.

Example Nested try block

```
class except {  
    public static void main(String  
        args[])  
    {  
        try {  
            try {  
                String name=null;  
                System.out.print(name.length());  
            }  
            catch (ArithmeticException e)  
            {  
                System.out.println("Arithmetic
```

```
exception");  
            } }  
            catch (NullPointerException c)  
            {  
                System.out.println(c);  
                System.out.println(" outer-try  
block ");  
            } } }
```

output:

```
java.lang.NullPointerException  
outer-try block
```

3) finally

- Java finally block is always executed whether an exception is handled or not. Therefore, it contains all the necessary statements that need to be printed regardless of the exception occurs or not.
- Why use Java finally block?
- finally block in Java can be used to put "**cleanup**" code such as closing a file, closing connection, etc.
- The important statements to be printed can be placed in the finally block.
- Rule: For each try block there can be zero or more catch blocks, but only one finally block.
- Note: If you don't handle the exception, before terminating the program, JVM executes finally block (if any).

Example

```
public class TestFinallyBlock
{
    public static void main(String args[]) {
        try {
            System.out.println("Inside the try
            block");
            //below code throws divide by zero

            int data=25/0;
            System.out.println(data);
        }

        //cannot handle Arithmetic type
        catch(NullPointerException e)
```

```
{
    System.out.println(e); }
//executes regardless of exception
occured or not

finally
{
    System.out.println("finally block is
    always executed");
}
System.out.println("rest of the cod
e...");
} }
```

4) throw

- The Java throw keyword is used to throw an exception explicitly.
- We specify the **exception** object which is to be thrown. The Exception has some message with it that provides the error description. These exceptions may be related to user inputs, server, etc.
- We can throw either checked or unchecked exceptions in Java by throw keyword. It is mainly used to throw a custom exception.
- Syntax: **throw new exception_class("error message");**
- eg: **throw new IOException("sorry device error");**


```

class TestThrows {
    //function to check if person is eligible
    //to vote or not
    public static void validate(int age)
    {
        if(age<18) {
            //throw Arithmetic exception if not
            //eligible to vote
            throw new
                ArithmeticException("Person is not
                    eligible to vote");
        }
        else
        {
            System.out.println("Person is
                eligible to vote!!");
        }
    }
    public static void main(String args[])

```

```

{    //calling the function
    validate(13);
    System.out.println("rest of the
        code...");
} }

```

output:

```

Exception in thread "main"
java.lang.ArithmeticException: Person
is not eligible to vote
    at
    Gautam.validate(TestThrows.java:76)
    at
    Gautam.main(TestThrows.java:85)

```

5) throws

- The **Java throws keyword** is used to declare an exception. It gives an information to the programmer that there may occur an exception. So, it is better for the programmer to provide the exception handling code so that the normal flow of the program can be maintained.
- Syntax of Java throws
return_type method_name() **throws** exception_class_name
{ //method code
}
- It provides information to the caller of the method about the exception

```

class TestThrows
{
    public static void main(String args[]) throws InterruptedException
    {
        for(int i=0;i<10;i++)
        {
            try
            {
                Thread.sleep(1000);
                System.out.print(i);
            }
            catch(InterruptedException e)
            {
                System.out.println(e);
            }
        }
    }
}

```

throw

Throw keyword is used to throw an exception object explicitly.

```
Void square()  
{ throw new AE(); }
```

Throw keyword always present inside **method body**.

We can throw only one exception at a time.

```
Throw new AE
```

Throw is followed by an instance.

throws

Throws keyword is used to declared an exception as well as by the caller.

```
Void method() throws AE  
{ // block code }
```

Throws keyword always used with **method signature**.

We can handle multiple exception using throws keyword.

```
method() AE,NPE,IO { }
```

Th**rows is** followed by class.

```
class InvalidAgeException extends  
Exception
```

```
{  
    InvalidAgeException(String msg)  
    {  
        System.out.println(msg);  
    }  
}
```

```
Class UserException
```

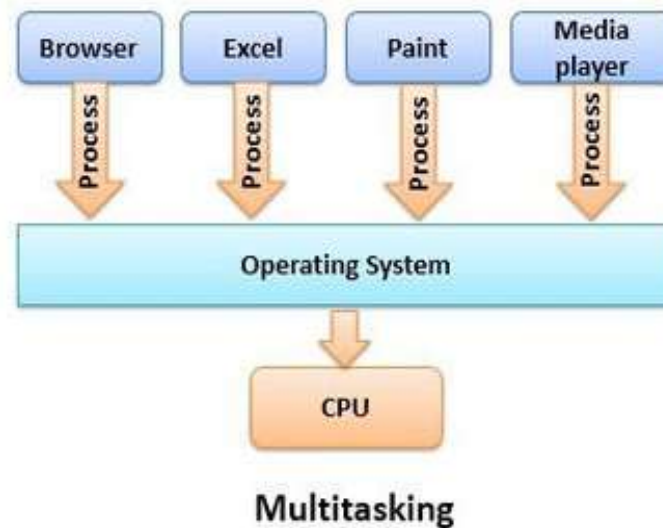
```
{  
    public static void main(String args[])  
throws InvalidAgeException  
    {  
        validateage(12);  
    }  
    public static void validateage(int age)  
throws InvalidAgeException
```

```
{  
    if(age<18)  
    {  
        throw new  
InvalidAgeException("notvalid");  
    }  
    else  
    {  
        System.out.println("valid age for  
vote");  
    }  
}}
```

Output:
Not valid

Multitasking

- Multitasking is a process of executing multiple tasks simultaneously. We use multitasking to utilize the CPU. Multitasking can be achieved in two ways:
 1. Process-based Multitasking (Multiprocessing)
 2. Thread-based Multitasking (Multithreading)

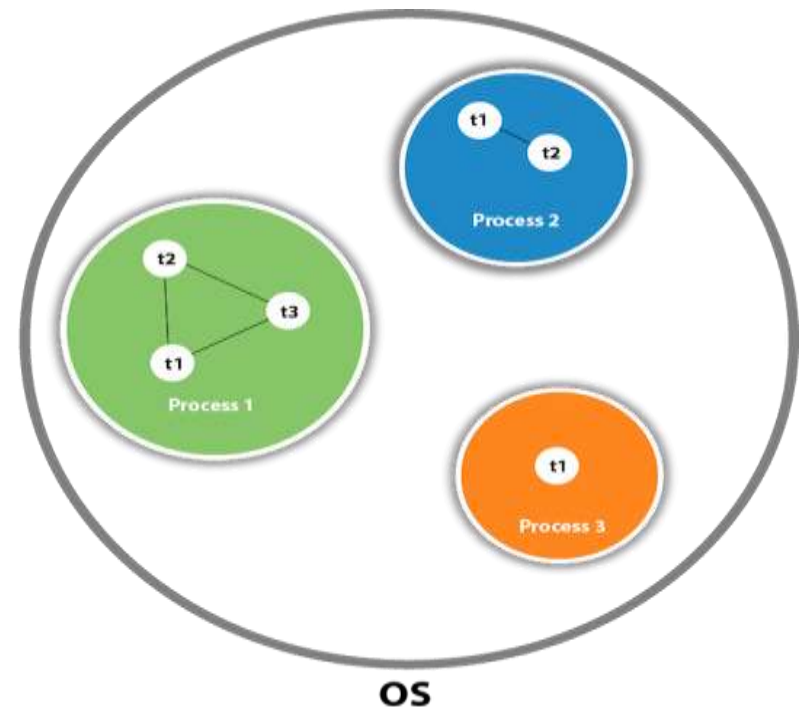


1) Process-based Multitasking (Multiprocessing)

- Each process has an address in memory. In other words, each process allocates a separate memory area.
- A process is heavyweight.
- Cost of communication between the process is high.
- Switching from one process to another requires some time for saving and loading registers, memory maps, updating lists, etc.

2) Thread-based Multitasking (Multithreading)

- Threads share the same address space.
- A thread is lightweight.
- Cost of communication between the thread is low.



Multithreading in Java

- Multiprocessing and multithreading, both are used to achieve multitasking.
- However, we use multithreading than multiprocessing because threads use a shared memory area. They don't allocate separate memory area so saves memory, and context-switching between the threads takes less time than process.
- Java Multithreading is mostly used in games, animation, etc.
- Advantages of Java Multithreading
 - 1) It doesn't block the user because threads are independent and you can perform multiple operations at the same time.
 - 2) You can perform many operations together, so it saves time.
 - 3) Threads are independent, so it doesn't affect other threads if an exception occurs in a single thread.

What is Thread in java

- Multithreading is a Java feature that allows concurrent execution of two or more parts of a program for maximum utilization of CPU. Each part of such program is called a thread.
- A thread is a lightweight sub process, the smallest unit of processing. It is a separate path of execution.
- Threads are independent. If there occurs exception in one thread, it doesn't affect other threads. It uses a shared memory area.
- There can be multiple processes inside the OS, and one process can have multiple threads.
- Threads can be created by using two mechanisms :
 1. Extending the Thread class
 2. Implementing the Runnable Interface

Thread creation by extending the Thread class

- We create a class that extends the **java.lang.Thread** class. This class overrides the run() method available in the Thread class. A thread begins its life inside run() method. We create an object of our new class and call start() method to start the execution of a thread. Start() invokes the run() method on the Thread object.
- **Commonly used Constructors of Thread class:**
 - ❖ Thread()
 - ❖ Thread(String name)
 - ❖ Thread(Runnable r)
 - ❖ Thread(Runnable r, String name)

```

class ThreadDemo extends Thread
{
    public void run()
    { try
      {
        for(int i=1;i<=3;i++)
        {
          System.out.println("ThreadDemo");
          Thread.sleep(1000);
        }
      }
    }
    catch(InterruptedException i)
    {}
}
}

class moto
{
    public static void main(String args[])
    throws InterruptedException

```

```

{
    ThreadDemo t=new ThreadDemo();
    t.start();
    for(int i=1;i<=3;i++)
    {
        System.out.println("moto-class");
        Thread.sleep(1000);
    }
}
}
}

```

output:

```

ThreadDemo
moto-class
ThreadDemo
ThreadDemo
moto-class
moto-class

```

Runnable interface

- The Runnable interface should be implemented by any class whose instances are intended to be executed by a thread. Runnable interface have only one method named run().
- **public void run():** is used to perform action for a thread.

Starting a thread:

- The **start() method** of Thread class is used to start a newly created thread. It performs the following tasks:
- A new thread starts(with new callstack).
- The thread moves from New state to the Runnable state.
- When the thread gets a chance to execute, its target run() method will run.

Thread creation Implementing the Runnable Interface

class ThreadDemo implements Runnable

```
{  
    public void run()  
    {  
        for(int i=1;i<=3;i++)  
        {  
            System.out.println("ThreadDemo");  
        }  
    }  
}
```

class moto

```
{  
    public static void main(String[] args)  
    {  
        ThreadDemo obj=new ThreadDemo();  
    }  
}
```

```
Thread t = new Thread(obj);  
// Using constructor Thread(Runnable r)  
t.start();  
for(int i=1;i<=3;i++)  
{  
    System.out.println("moto");  
}}
```

output:

```
moto  
ThreadDemo  
ThreadDemo  
moto  
ThreadDemo  
moto
```

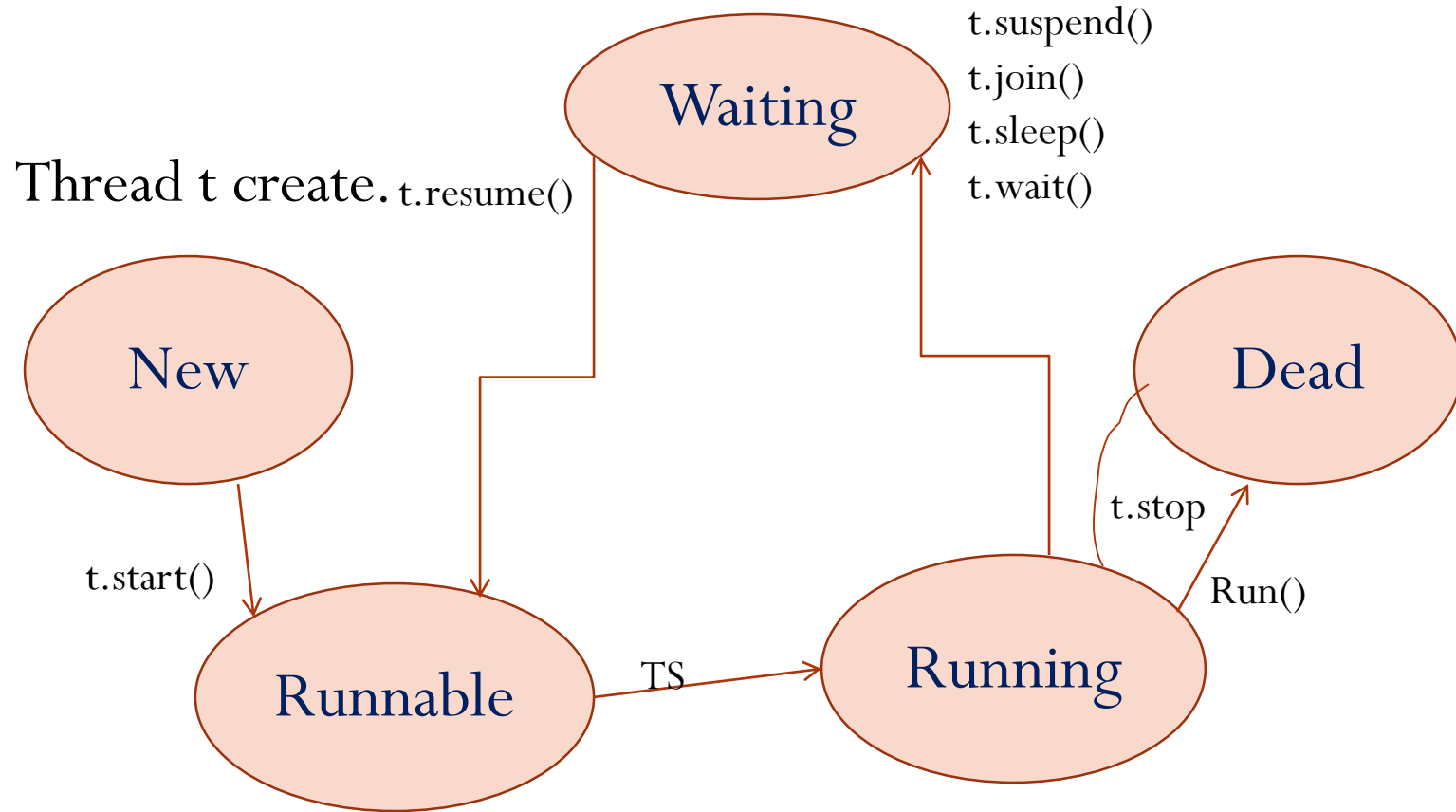
Thread Scheduler in Java

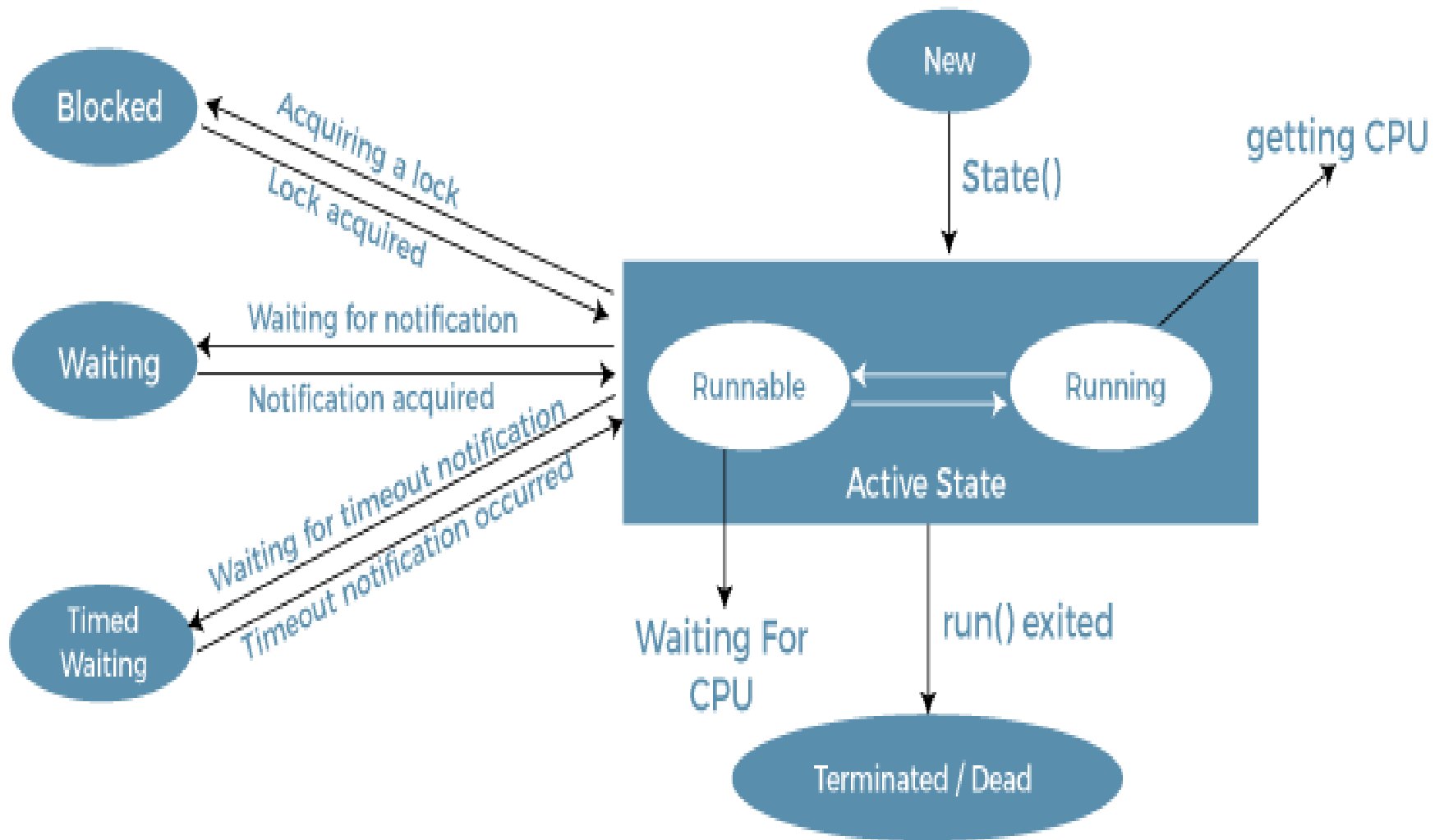
- A component of Java that decides which thread to run or execute and which thread to wait is called a **thread scheduler in Java**.
- In Java, a thread is only chosen by a thread scheduler if it is in the runnable state.
- However, if there is more than one thread in the runnable state, it is up to the thread scheduler to pick one of the threads and ignore the other ones.
- There are two factors for scheduling a thread i.e. **Priority** and **Time of arrival**.
- Thread Scheduler Algorithms
 - First Come First Serve Scheduling
 - Short job first
 - Round Robin

Life cycle of a Thread (Thread States)

- In Java, a thread always exists in any one of the following states. These states are:
 1. New state(Born)
 2. Runnable state (Ready)
 3. Running state (Execution)
 4. Waiting state (Blocked)
 5. Dead state (Exit)
- **New:** Whenever a new thread is created, it is always in the new state. For a thread in the new state, the code has not been run yet and thus has not begun its execution.

Thread Lifecycle





Life Cycle of a Thread

- **Active:** When a thread invokes the start() method, it moves from the new state to the active state. The active state contains two states within it: one is runnable, and the other is running.
- **Runnable:** A thread, that is ready to run is then moved to the runnable state. It is the duty of the thread scheduler to provide the thread time to run.
- **Running:** When the thread gets the CPU, it moves from the runnable to the running state.
- **Blocked or Waiting:** Whenever a thread is inactive for a span of time (not permanently) then, either the thread is in the blocked state or is in the waiting state.
- If there are a lot of threads in the waiting or blocked state, then it is the duty of the thread scheduler to determine which thread to choose and which one to reject, and the chosen thread is then given the opportunity to run.

- **Timed Waiting:** Sometimes, waiting for leads to starvation.
- **Terminated:** A thread reaches the termination state because of the following reasons:
 - When a thread has finished its job, then it exists or terminates normally.
 - **Abnormal termination:** It occurs when some unusual events such as an unhandled exception or segmentation fault.
 - A terminated thread means the thread is no more in the system. In other words, the thread is dead, and there is no way one can respawn (active after kill) the dead thread.

Thread class method

- What is Thread Priority ?
- In java it is possible to assign the priority of thread.
- To set these priority java thread class has provided two predefined methods.
 1. `setPriority(int newPriority)` -Changes the priority of this thread
 2. `getPriority()`- Returns this thread's priority.
- The thread class has also provided three pre-defined final static variable and its value lies between 1 to 10.
- `Thread.MIN_PRIORITY`----- 1
- `Thread.NORM_PRIORITY`----- 5
- `Thread.MAX_PRIORITY`-----10

Modifier and Type	Method	Description
void	Start()	It is used to start the execution of the thread.
void	run()	It is used to do an action for a thread.
Static void()	Sleep()	It sleeps a thread for the specified amount of time.
Static Thread	currentThread()	It returns a reference to the currently executing thread object
String	getName()	It returns the name of the thread.
void	setName()	It changes the name of the thread
long	getId()	It returns the id of the thread.
boolean	isAlive()	It tests if the thread is alive.
void	Suspend()	It is used to suspend the thread
void	Resume()	It is used to resume the suspended thread.

```
import java.lang.*;
public class Demo extends Thread
{
    public void run()
    {
        System.out.println(Thread.currentThread().getName());
        System.out.println(Thread.currentThread().getPriority());
    }
    public static void main(String
        args[])
    {
```

```
        Demo th1 = new Demo();
        Demo th2 = new Demo();
        th1.setName("Thread1");
        th2.setName("Thread2");

        th1.setPriority(6);
        th2.setPriority(9);
        th1.start();
        System.out.println(th1.isAlive());
        th2.start();
        System.out.println(Thread.currentThread().getId());
    } }
```

Synchronization

- Synchronization is a technique through which we can control multiple thread or among no. Of thread only one thread will enter inside the synchronized area.

Problem with multithreading:

- Multithreading is very good whenever we want to complete our task as soon as possible but in some situation it may provides some wrong result or some corrupted data, this situation occur whenever an same resource is accessible by multiple thread at the same time.
- Synchronized is broadly classified into two categories:
 1. Method level Synchronization
 2. Block level Synchronization

```

class ThreadDemo implements Runnable
{
    int available=2,passenger;
    ThreadDemo(int passenger)
    {
        this.passenger=passenger;
        //passenger=1
    }
    public synchronized void run()
    { String name=Thread.currentThread().
        getName();
        if(available>= passenger)    //1>=1
        {
            System.out.println(name + "reserved");
            available=available- passenger;    //1-1=0
        }
        else
        {
            System.out.println("not reserved!");
        }
    }
}

```


```

}}
class moto
{
    public static void main(String[] args)
    {
        ThreadDemo obj=new ThreadDemo(1);
        Thread t1=new Thread(obj);
        Thread t2=new Thread(obj);
        Thread t3=new Thread(obj);

        t1.setName("Thread-1");
        t2.setName("Thread-2");
        t3.setName("Thread-3");
        t1.start();
        t2.start();
        t3.start();
    }
}

```


What is method level Synchronization ?

- In method level Synchronization the entire method get Synchronized so, only one thread will enter inside the Synchronized area and remaining all the thread will wait at method level.
- Note: Every object have a lock in java and this lock can be given to only one thread at all the time.
- Ex:  `Table obj=new Table();`
 `Thread t1=new Thread();`
 `Thread t1=new Thread();`

```

class Table
{
    synchronized void printTable(int n)
    {   for(int i=1;i<=5;i++)
        {
            System.out.print(n*i);
        } } }
class MyThread1 extends Thread
{   Table t;
    MyThread1(Table t) {
        this.t=t;
    }
    public void run() {
        t.printTable(5);
    } }
class MyThread2 extends Thread
{   Table t;
    MyThread2(Table t)

```

```

    {
        this.t=t;
    }
    public void run() {
        t.printTable(4);
    } }
public class ThreadDemo
{
    public static void main(String args[])
    {
        Table obj = new Table();    // only
        one object
        MyThread1 t1=new MyThread1(obj);
        MyThread2 t2=new MyThread2(obj);
        t1.start();
        t2.start();
    } }

```

Output: 5 10 15 20 25 4 8 12 16 20

What is block level Synchronization ?

- In block level synchronization the entire method is not get synchronized only the part of the method get synchronized, we have to enclosed those few lines of the code put inside synchronized block.
- Ex:

```
public void show()  
{  
    Lines of code  
    synchronized(this)  
    {  
        Block of code  
    }  
}
```

Points to Remember

- Scope of synchronized block is smaller than the method.
- The system performance may degrade because of the slower working of synchronized keyword.
- Java synchronized block is more efficient than Java synchronized method.

```

class msg
{
    public void show(String name) {
        synchronized(this) {
            for(int i=1;i<=3;i++)
            {
                System.out.println("hello" + " "
                    +name);
            }
        }
    }
}

class ourThread extends Thread
{
    msg m;
    String name;
    ourThread(msg m, String name)
    {
        this.m=m;
        this.name=name;
    }
}

```

```

        public void run()
        {
            m.show(name);
        }
    }

class Demo2
{
    public static void main(String[] args) {
        msg m=new msg();
        ourThread t1=new ourThread(m,
            "ram");
        ourThread t2=new ourThread(m,
            "sita");
        t1.start();
        t2.start();
    }
}

```

- **What are Non-Daemon threads?**

- Non-daemon threads are user threads designed to do specific, complex tasks. These are high priority threads that run in the foreground. They are often referred to as “normal threads” that we see and use in day to day coding. Also, any thread created from the Main thread is a non-daemon/user thread. Non daemon / user threads are created usually by the Java application.

- **What are Daemon Threads?**

- These are low priority threads that run in the background, which provide services to user threads running in the same Java application. Daemon threads are mostly created by JVM to perform background tasks and they run automatically e.g. garbage collection(gc), finalizer etc...