Exception Handling & Multithreading

Course: CSL304

Prof. Juhi Ganwani

CONTENT

Error vs Exception

- **Error:** An Error indicates serious problem that a reasonable application should not try to catch.
- **Exception:** Exception indicates conditions that a reasonable application might try to catch.
- When a problem encounters and unexpected termination or fault, it is called an exception
- When we try and divide by ‘0’ we terminate abnormally.
- Exception handling gives us another opportunity to recover from the abnormality.
- Sometimes we might encounter situations from which we cannot recover like ‘Out of memory’. These are considered as errors.

Exception	Error
1) Exceptions can be recovered	1) Errors cannot be recovered.
2) Exceptions can be classified in to two types: a) Checked Exception b) Unchecked Exception	2) There is no such classification for errors. Errors are always unchecked.
3) In case of checked Exceptions compiler will have knowledge of checked exceptions and force to keep try catch blocks.	3) In case of Errors compiler won't have knowledge of errors.

Exception

- An exception is an unwanted or unexpected event, which occurs during the execution of a program i.e. at run time, that disrupts the normal flow of the program’s instructions.
- Exception indicates conditions that a reasonable application might try to catch.
- Subclass defined in java.lang package

Types of exception

1. **Checked exception**

- Checked by the compiler at compile time
- If some code within a method throws a checked exception, then the method must either handle the exception or it must specify the exception using *throws* keyword.
- Programmer should take care of (handle) these exceptions.
- Example:if you use **FileReader** class in your program to read data from a file, if the file specified in its constructor doesn't exist, then a *FileNotFoundException* occurs, and the compiler prompts the programmer to handle the exception.

Example

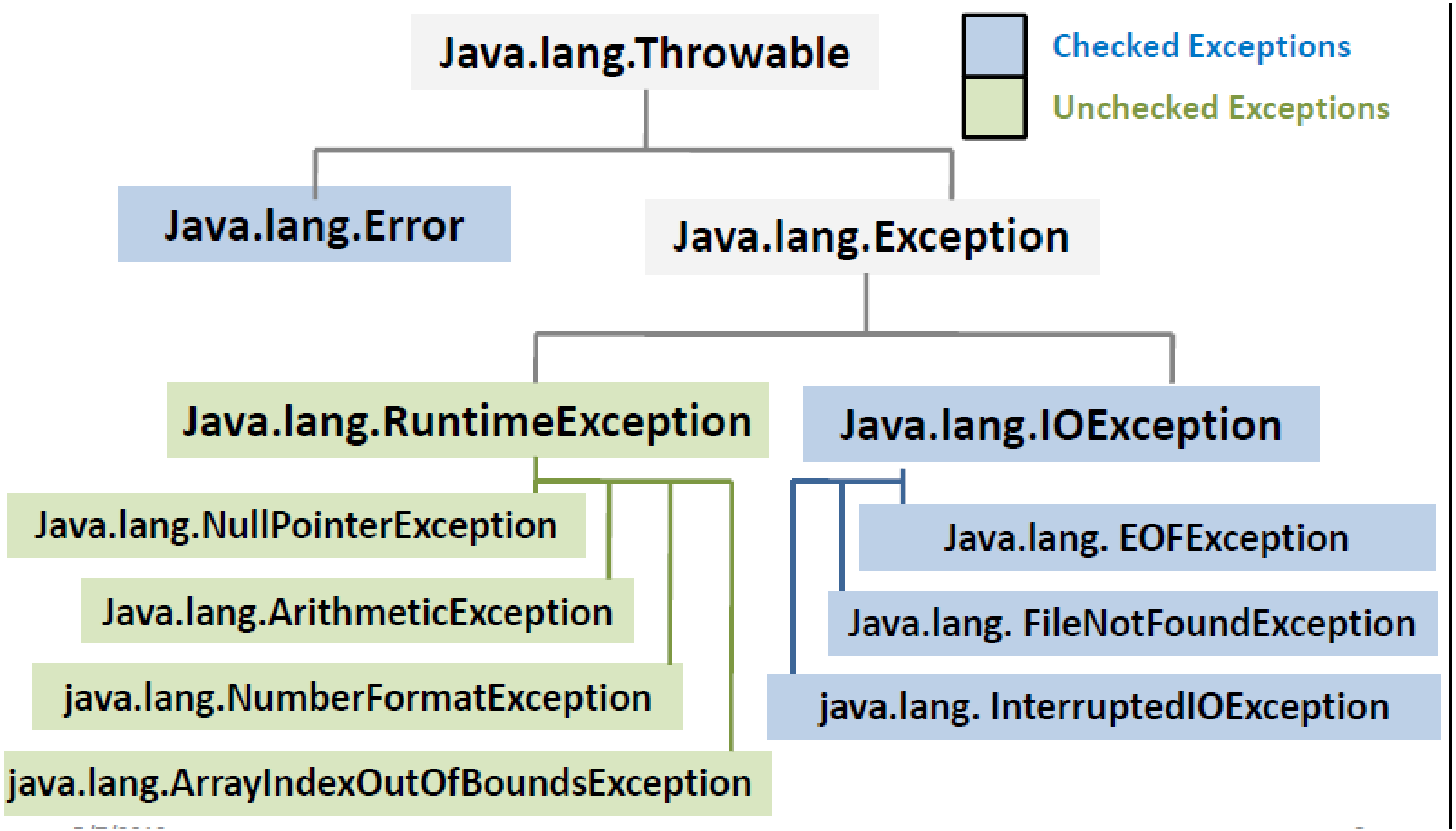
```
import java.io.*;
class FileNotFound {
   public static void main(String args[])
   {
          FileInputStream fis = null;
          fis = new FileInputStream("D:/myfile.txt");
   }
}
```

Types of exception

2. **Unchecked Exceptions**

- Exceptions that are not checked at compiled time
- Consider the following Java program. It compiles fine, but it throws *ArithmeticException* when run.

```
class DivisionException {
   public static void main(String args[]) {
      int x = 0;
      int y = 10;
      int z = y/x;
  }
}
```


Java.lang.Throwable
Checked Exceptions
Unchecked Exceptions
Java.lang.Error
Java.lang.Exception
Java.lang.RuntimeException
Java.lang.IOException
Java.lang.NullPointerException
Java.lang.ArithmeticException
java.lang.NumberFormatException
java.lang.ArrayIndexOutOfBoundsException
Java.lang. EOFException
Java.lang. FileNotFoundException
java.lang. InterruptedIOException

Exception handling

- It is a mechanism to handle runtime errors such as FileNotFoundException, ArithmeticException, ArrayIndexOutOfBoundsException,etc
- Core advantage of exception handling is to maintain the normal flow of the application

```
statement 1;
statement 2;
statement 3; //exception occurs
statement 4;
statement 5;
statement 6;
```

Statement 4,5 & 6 won’t be executed if exception occurs at 3.

try….catch

- **try** statement allows you to define a block of code to be tested for errors while it is being executed.
- **catch** statement allows you to define a block of code to be executed, if an error occurs in the try block.
- try and catch keywords come in pairs

```
try {
  //  Block of code to try
}
catch(Exception e) {
  //  Block of code to handle errors
}
```

Rules of try..catch

- Catch block is added immediately after the try block
- Try block can have multiple catch blocks
- Try block can have multiple statements which can generate an exception
- If any one statement generates an exception, the remaining statement in the block are skipped & execution jumps to the catch block
- Every try block should have atleast one catch block
- Nested try block is also possible

Examples

ArrayIndexOutOfBoundsException

- Generally array is of fixed size & elements are accessed using indices.
- Say suppose an array, a is defined with size 10, we can access array with indices 0 to 9 i.e. a[0] to a[9].
- Whenever you used an –ve value or, the value greater than or equal to the size of the array, then the **ArrayIndexOutOfBoundsException** is thrown.
- These types of exception can be handled using **ArrayIndexOutOfBoundsException** class.

Program without handling exception

```
public class ArrayException
{
    public static void main(String[] args)
    {
        int ar[] = {1, 2, 3, 4, 5};
        for (int i=0; i<=ar.length; i++)
          System.out.println(ar[i]);
        System.out.println(“Loop terminated”);
    }
}
```

Program handling the exception

```
public class ArrayExceptionHandling
{
    public static void main(String[] args)
    {
        int ar[] = {1, 2, 3, 4, 5};
        try{
        for (int i=0; i<=ar.length; i++)
          System.out.println(ar[i]);}
         catch(ArrayIndexOutOfBoundsException e){
                  System.out.println("Exception thrown: "+e);}
         System.out.println("Out of the block"); }
}
```

Arithmetic Exception

- One would come across **java.lang.ArithmeticException: / by zero** which occurs when an attempt is made to **divide two numbers** and the number in the **denominator is zero**

Example

```
class ArithmeticMain
{
            public static void main(String args[])
            {
            int a[]={5,10};
            int b=5;
            try{
            int x=a[1]/(a[0]-b);
            }
            catch(ArithmeticException e){

            System.out.println("Division by zero");}
catch(ArrayInsexOutOfBoundsException e)
{
System.out.println("Array index is out of
bound");
}
int y=a[1]/a[0];
System.out.println("y="+y);
}
}
```

NumberFormatException

- NumberFormatException can be thrown by many methods/constructors in the classes of java.lang package.
- For instance, public static int parseInt(String s) throws NumberFormatException when:
  - String s is null or length of s is zero.
  - If the String s contains non-numeric characters.
  - Value of String s doesn’t represent an Integer.

Example

```
public class NumberFormatExceptionTest
{
   public static void main(String[] args){
      int x = Integer.parseInt("30k");
      System.out.println(x);
   }
}
```

Handling NumberFormatException

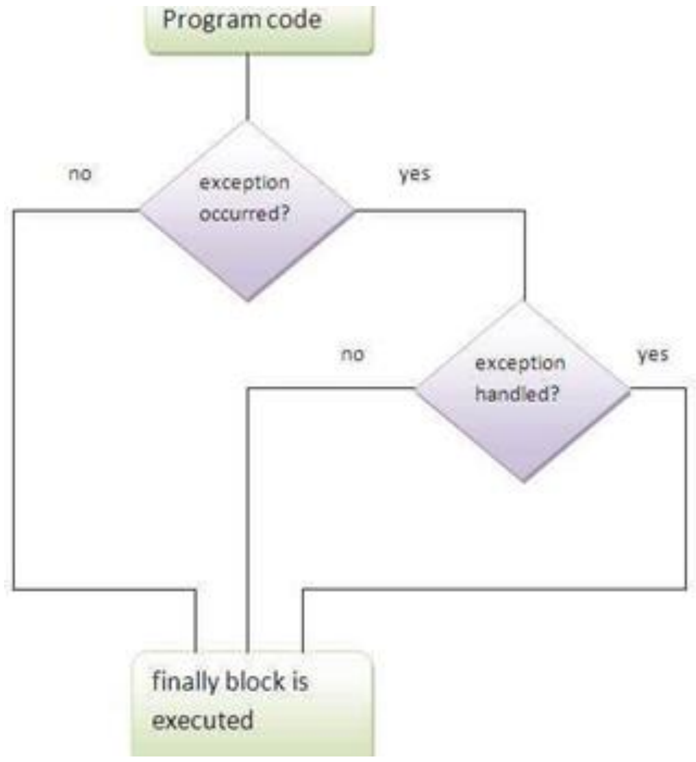
```
public class NumberFormatExceptionHandling
{
   public static void main(String[] args){
     try {
      int x = Integer.parseInt("30k");
      System.out.println(x);
     }
     catch(NumberFormatException e){
              System.out.println("Exception handled");
      }
   }
}
```

Nested try catch

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

        System.out.println("Going to divide");
        int b=45/0;}

        catch(ArithmeticException e){

        System.out.println(e);}
        System.out.println("Outer try catch");
}
catch(Exception e){

        System.out.println("Handled");}
System.out.println("Normal flow");
        }
}
```

finally statement

- finally block follows a try or a catch block
- It always executes irrespective of occurrence of an exception
- Finally block in java can be used to put "cleanup" code such as closing a file, closing connection etc.



Example

```
public class TestFinallyBlock
{
  public static void main(String args[]){
  try{
   int data=25/0;
   System.out.println(data);
  }
  catch(ArithmeticException e){
     System.out.println(e);}
  finally{
  System.out.println("finally block is always executed");}
  System.out.println("rest of the code...");  }
}
```

throws keyword

- It is used to declare an exception that may occur during the method execution.
- It gives an information to the programmer that there may occur an exception.
- It is mainly used to handle the checked exceptions.

Syntax:

```
return-type method_name()throws exception_classname
{
        //method code
}
```

throw keyword

- It is used to explicitly throw an exception.
- We can throw either checked or unchecked exception in java by throw keyword
- It is mainly used to throw custom exception.

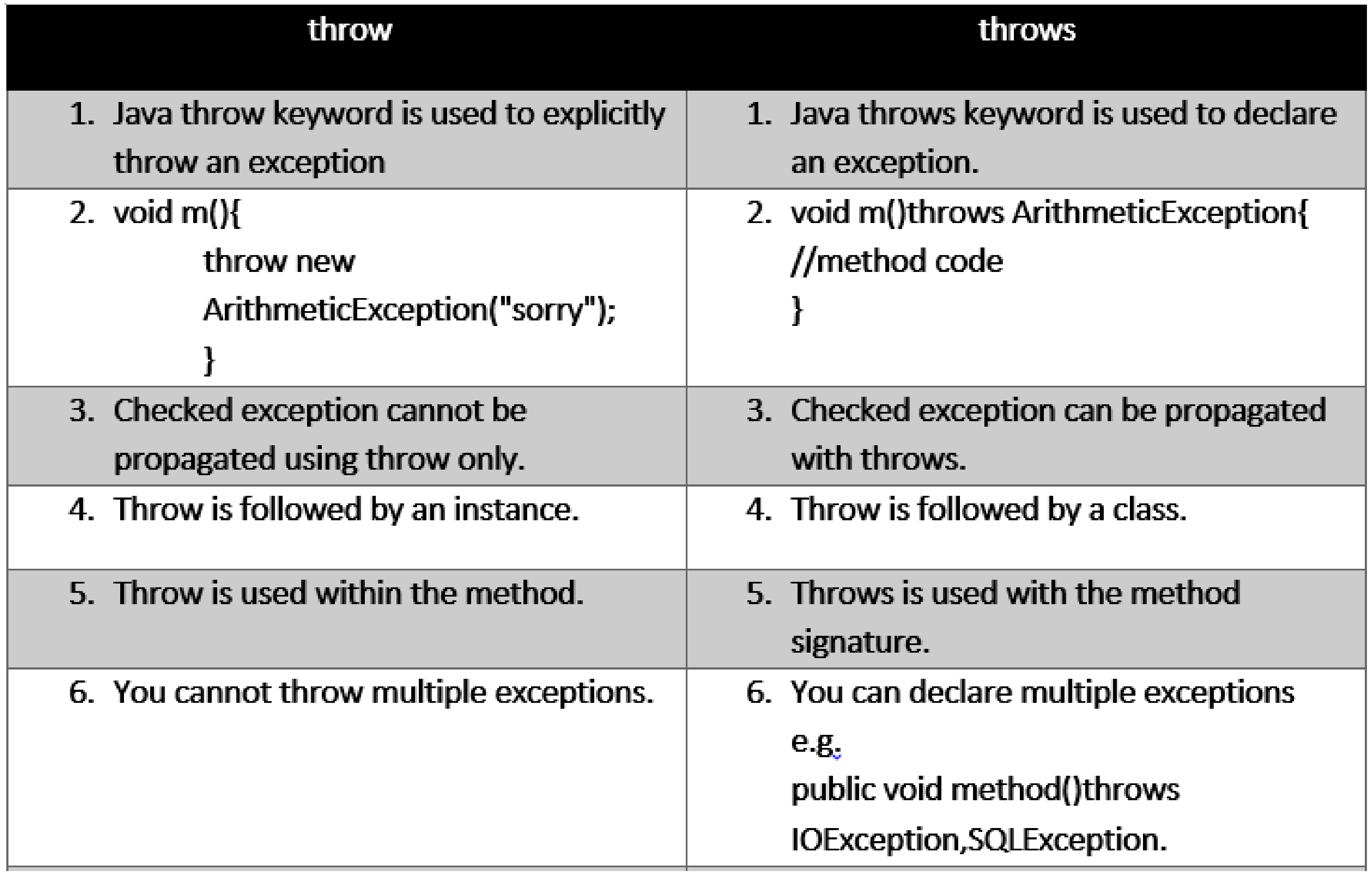
Syntax:

```
        throw new exception_classname(message);
```

Example

```
public class TestThrow
{
        static void validate(int age)
        {
                if(age<18)
                        throw new ArithmeticException("not valid");
                else
                        System.out.println("Welcome to vote");
        }
        public static void main(String args[])
        {
                validate(13);
                System.out.println("rest of the code..");
        }
}
```

throw vs throws



throw	throws
1. Java throw keyword is used to explicitly throw an exception	1. Java throws keyword is used to declare an exception.
2. void m(){ throw new ArithmeticException("sorry"); }	2. void m()throws ArithmeticException{ //method code }
3. Checked exception cannot be propagated using throw only.	3. Checked exception can be propagated with throws.
4. Throw is followed by an instance.	4. Throw is followed by a class.
5. Throw is used within the method.	5. Throws is used with the method signature.
6. You cannot throw multiple exceptions.	6. You can declare multiple exceptions e.g. public void method()throws IOException,SQLException.

Creating own exceptions

- Java provides us facility to create our own exceptions which are basically derived classes of Exception.
- These are called as Java custom exceptions or user-defined exceptions.

Example

```
class InvalidAgeException extends Exception
{
InvalidAgeException(String s){
super(s);}
}
class TestCustomException
{
static void validate(int age)throws InvalidAgeException{
if(age<18)
        throw new InvalidAgeException("not valid");
else
        System.out.println("Welcome to vote");}
```

```
public static void main(String args[])
{
try
{validate(13);}
catch(Exception e)
{
System.out.println("Exception occured "+e);}
System.out.println("rest of the code");
        }
}
```

Multithreading

Introduction

- A thread is a lightweight sub-process, the smallest unit of processing.
- Multithreading in Java is a process of executing multiple threads simultaneously.
- It is used to achieve multitasking.
- Multiprocessing and multithreading, both are used to achieve multitasking
- Multithreading is more preferred as threads use a shared memory area & context switching between threads takes lesser time than process.

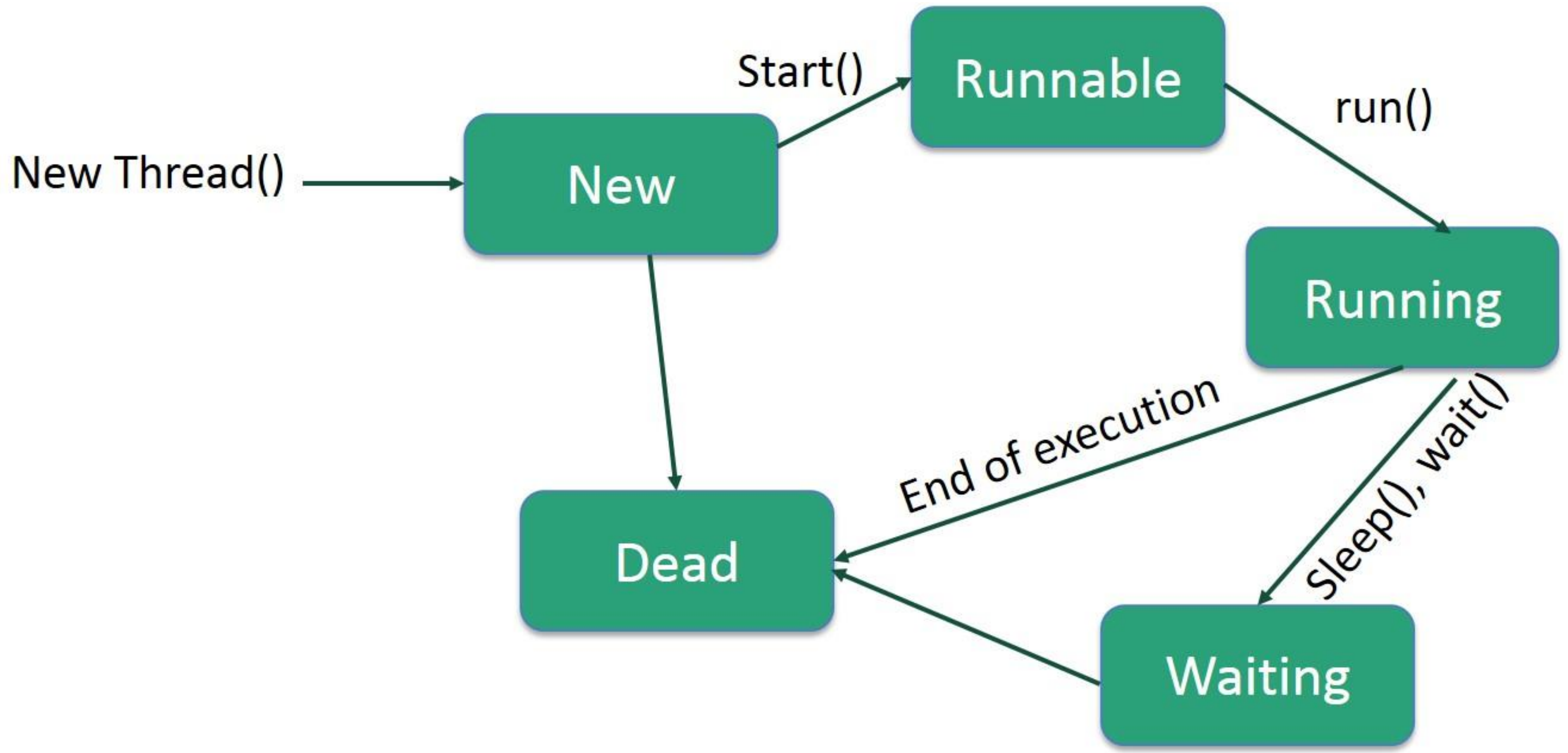
Advantage of using threads

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.

Life cycle of thread



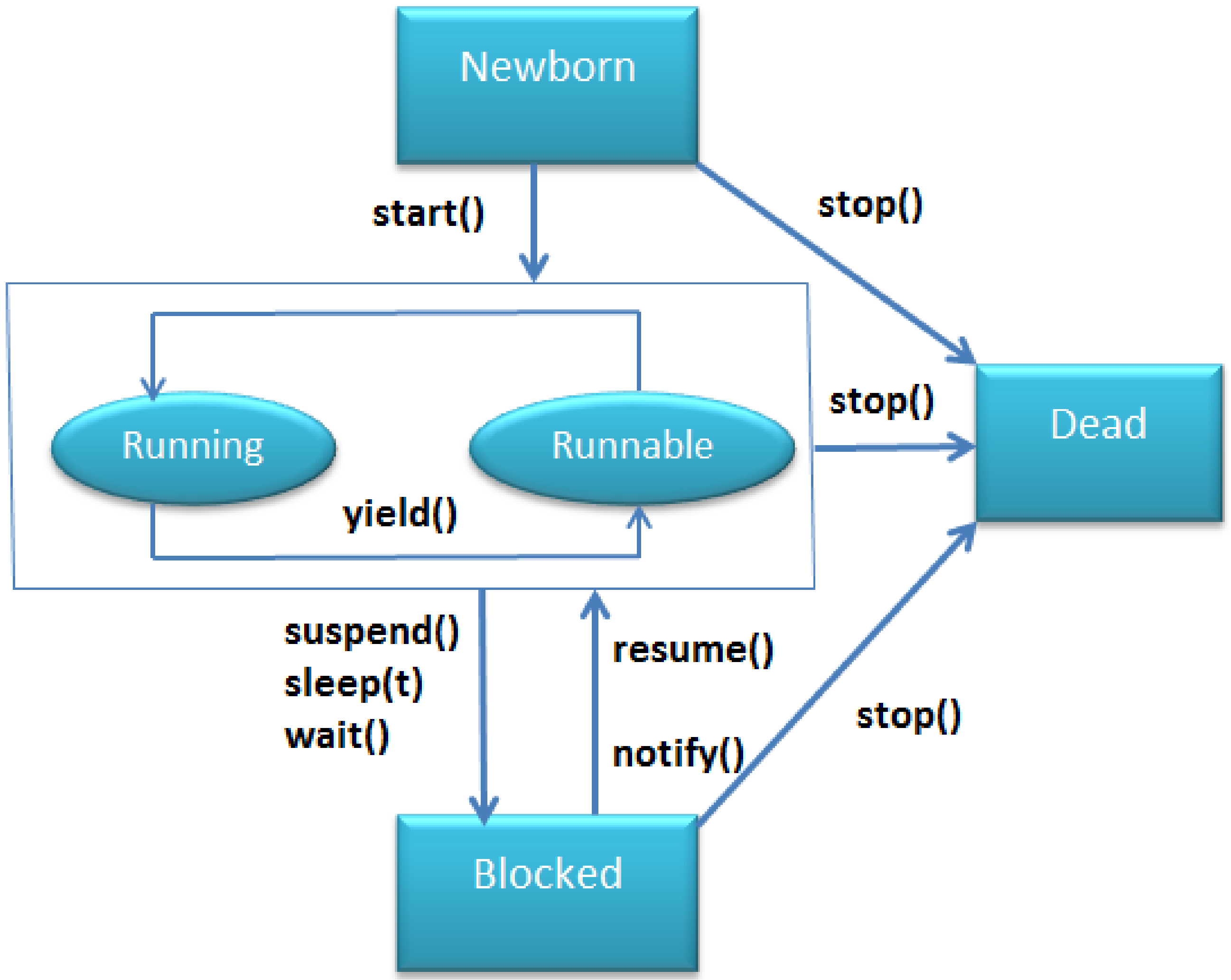


Fig: Life Cycle of Thread

Newborn state:

- Thread object is created using start() method.

Runnable state:

- It means that the thread is ready for execution and it is waiting for the availability of the processor.
- If all threads have equal priority, then they are given time slots for execution FCFS manner (**First Come First Serve**).
- Then the thread that relinquishes control joins the queue at the end and again waits for its turn.
- This process of assigning time to threads that are known as **Time-Slicing**. At this State, we can do by using **yield()** method.

Running state:

- It means that the processor has given its time to the thread for its execution. The thread runs until it relinquishes control on its own or it is preempted by a higher priority thread.
- When a running thread may relinquish its control, then some following situations occur:

i. It may be suspended using **suspend()** method. A suspended thread can be revived by using **resume()** method.

ii. It has been made to sleep. So, we can put to sleep a thread for a specified time period by using **sleep(time)** method, where time is in milliseconds. It means that the thread is out of the queue during this time period.

iii. It has been told to wait until some event occurs. We can do by using **wait()** method. Then the thread can be scheduled to run again using **notify()** method.

Blocked state:

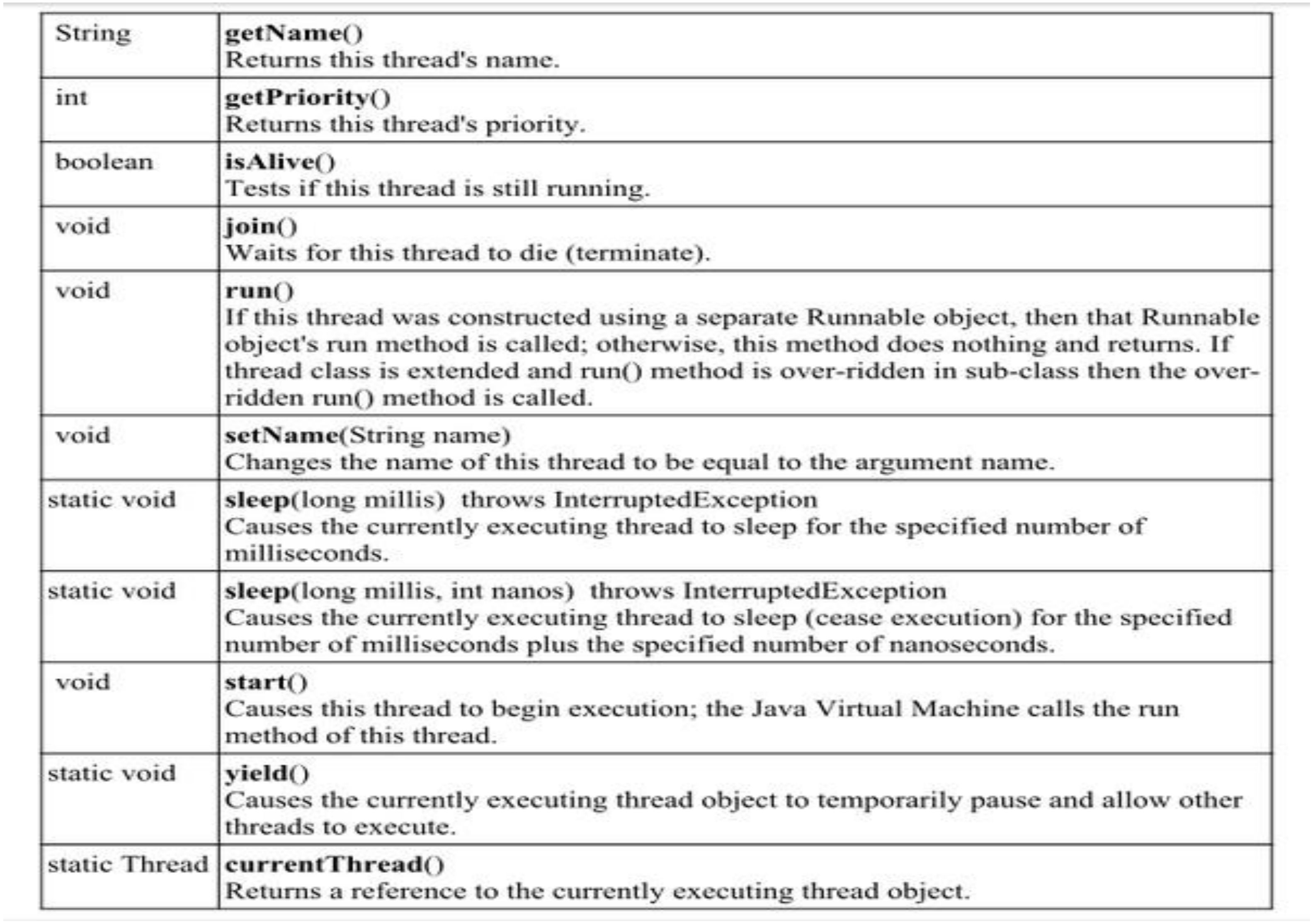
- A thread is said to be blocked when it is prevented from entering into the runnable state and subsequently the running state. This happens when the thread is suspended, sleeping or waiting to satisfy certain requirements. A blocked thread is “**not runnable**” but not dead.

Dead State:

- A running thread ends its life when it has completed executing its **run()** method. It is a natural death. We can kill the thread by using **stop()** method.

Note: stop(), resume() and suspend() method is deprecated.

Thread class methods



String	**getName**() Returns this thread's name.
int	**getPriority**() Returns this thread's priority.
boolean	**isAlive**() Tests if this thread is still running.
void	**join**() Waits for this thread to die (terminate).
void	**run**() If this thread was constructed using a separate Runnable object, then that Runnable object's run method is called; otherwise, this method does nothing and returns. If thread class is extended and run() method is over-ridden in sub-class then the over-ridden run() method is called.
void	**setName**(String name) Changes the name of this thread to be equal to the argument name.
static void	**sleep**(long millis) throws InterruptedException Causes the currently executing thread to sleep for the specified number of milliseconds.
static void	**sleep**(long millis, int nanos) throws InterruptedException Causes the currently executing thread to sleep (cease execution) for the specified number of milliseconds plus the specified number of nanoseconds.
void	**start**() Causes this thread to begin execution; the Java Virtual Machine calls the run method of this thread.
static void	**yield**() Causes the currently executing thread object to temporarily pause and allow other threads to execute.
static Thread	**currentThread**() Returns a reference to the currently executing thread object.

Creating a thread

- There are two ways to create a thread:

1. It can be created by extending the Thread class & overriding its run() method.
2. Another way to create a thread is to implement the Runnable interface & implement a run() method.

Creating a thread by extending a thread class

```
class MyThread extends Thread
{
public void run()
{
for(int i=0;i<10;i++)
System.out.println("child thread");
}
}
```

```
class MultiThreadUsingClass
{
public static void main(String args[])
{
MyThread t=new MyThread();
t.start();
for(int i=0;i<10;i++)
System.out.println("main thread");
}
}
```

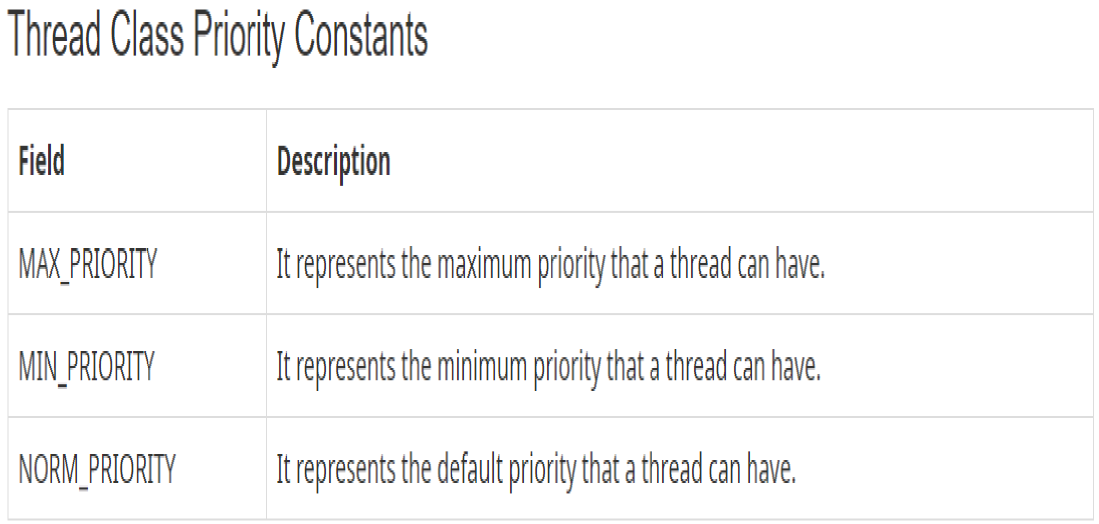
Creating a thread by implementing Runnable interface

```
class MyRunnable implements Runnable
{
public void run()
{
for(int i=0;i<10;i++)
System.out.println("child thread");
}
}
```

```
class MultiThreadUsingInterface
{
public static void main(String args[])
{
MyRunnable r=new MyRunnable();
Thread t=new Thread(r);
t.start();
for(int i=0;i<10;i++)
        System.out.println("main thread");
}
}
```

Thread class priority constants

- Scheduler assigns processor to a thread based on priority of thread. Whenever we create a thread in Java, it always has some priority assigned to it.
- Priority can either be given by JVM while creating the thread or it can be given by programmer explicitly.
- Accepted value of priority for a thread is in range of 1 to 10.
- There are 3 static variables defined in Thread class for priority.
- MAX_PRIORITY: 10, MIN_PRIORITY: 1, NORM_PRIORITY: 5



Thread Class Priority Constants

Field	Description
MAX_PRIORITY	It represents the maximum priority that a thread can have.
MIN_PRIORITY	It represents the minimum priority that a thread can have.
NORM_PRIORITY	It represents the default priority that a thread can have.

Example

```
class TestMultiPriority extends Thread{
 public void run(){
   System.out.println("running thread name is:"+Thread.currentThread().getName());
   System.out.println("running thread priority is:"+Thread.currentThread().getPriority());    }
 public static void main(String args[]){
  TestMultiPriority m1=new TestMultiPriority();
  TestMultiPriority m2=new TestMultiPriority();
  TestMultiPriority m3=new TestMultiPriority();
  m1.setPriority(Thread.MIN_PRIORITY);
  m2.setPriority(Thread.MAX_PRIORITY);
  m1.start();
  m2.start();
  m3.start(); }  }
```

Multiple threads Example

```
class RunnableDemo implements Runnable {
  private Thread t;
  private String threadName;

  RunnableDemo( String name) {
    threadName = name;
    System.out.println("Creating " +  threadName );
  }

  public void run() {
    System.out.println("Running " +  threadName );
    try {
      for(int i = 4; i > 0; i--) {
        System.out.println("Thread: " + threadName + ", " + i);
        // Let the thread sleep for a while.
        Thread.sleep(50);
      }
    } catch (InterruptedException e) {
      System.out.println("Thread " +  threadName + " interrupted.");
    }
    System.out.println("Thread " +  threadName + " exiting.");
  }

  public void start () {
    System.out.println("Starting " +  threadName );
    if (t == null) {
      t = new Thread (this, threadName);
      t.start ();
    }
  }
}

public class TestMultipleThread {

  public static void main(String args[]) {
    RunnableDemo r1 = new RunnableDemo( "Thread-1");
    r1.start();

    RunnableDemo r2 = new RunnableDemo( "Thread-2");
   r2.start();
  }
}
```