Object Oriented Programming Methodology

Module 6 : Exception Handling, Packages, Multithreading

Faculty In-charge:

Pragya Gupta

E-mail: pragya.g@somaiya.edu

Contents

Exception

- Exceptions in real life are rare
- are usually used to denote something unusual that does not conform to the standard rules
- In programming, exceptions are events that arise due to the occurrence of unexpected behaviour in certain statements, disrupting the normal execution of a program

Causes of Exception

- Exceptions can arise due to a number of situations. For example,
  - Trying to access the 11th element of an array when the array contains of only 10 element (*ArrayIndexOutOfBoundsException*)
  - Division by zero (*ArithmeticException*)
  - Accessing a file which is not present (*FileNotFoundException*)
  - Failure of I/O operations (*IOException*)
  - Illegal usage of null. (*NullPointerException*)

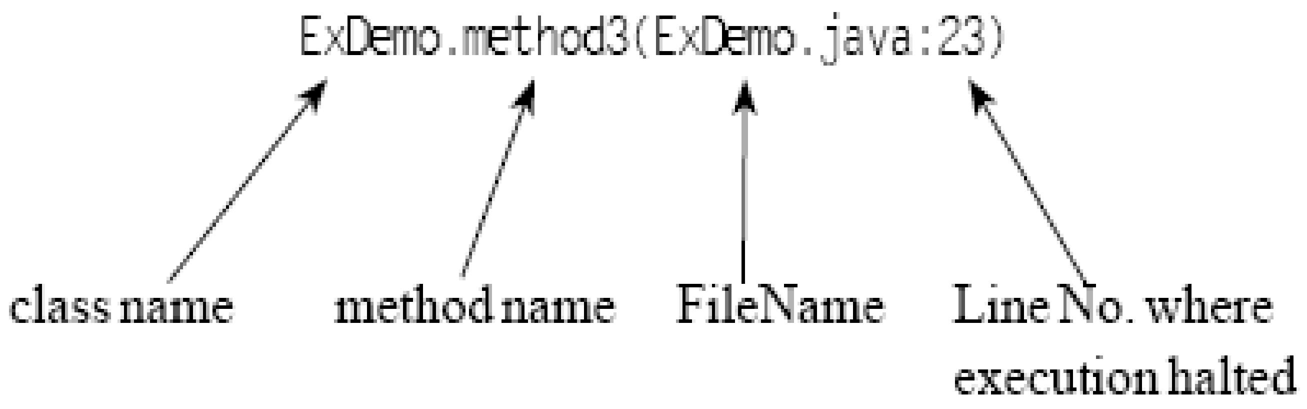
Exception classes

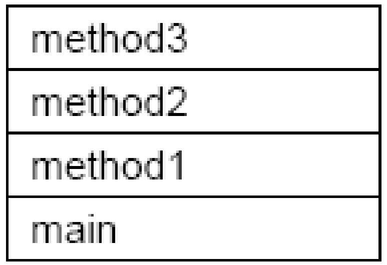
- Top class in exception hierarchy is *Throwable*
- This class has two siblings: Error and Exception
- All the classes representing exceptional conditions are subclasses of the Exception class

What happens when an execution occurs?

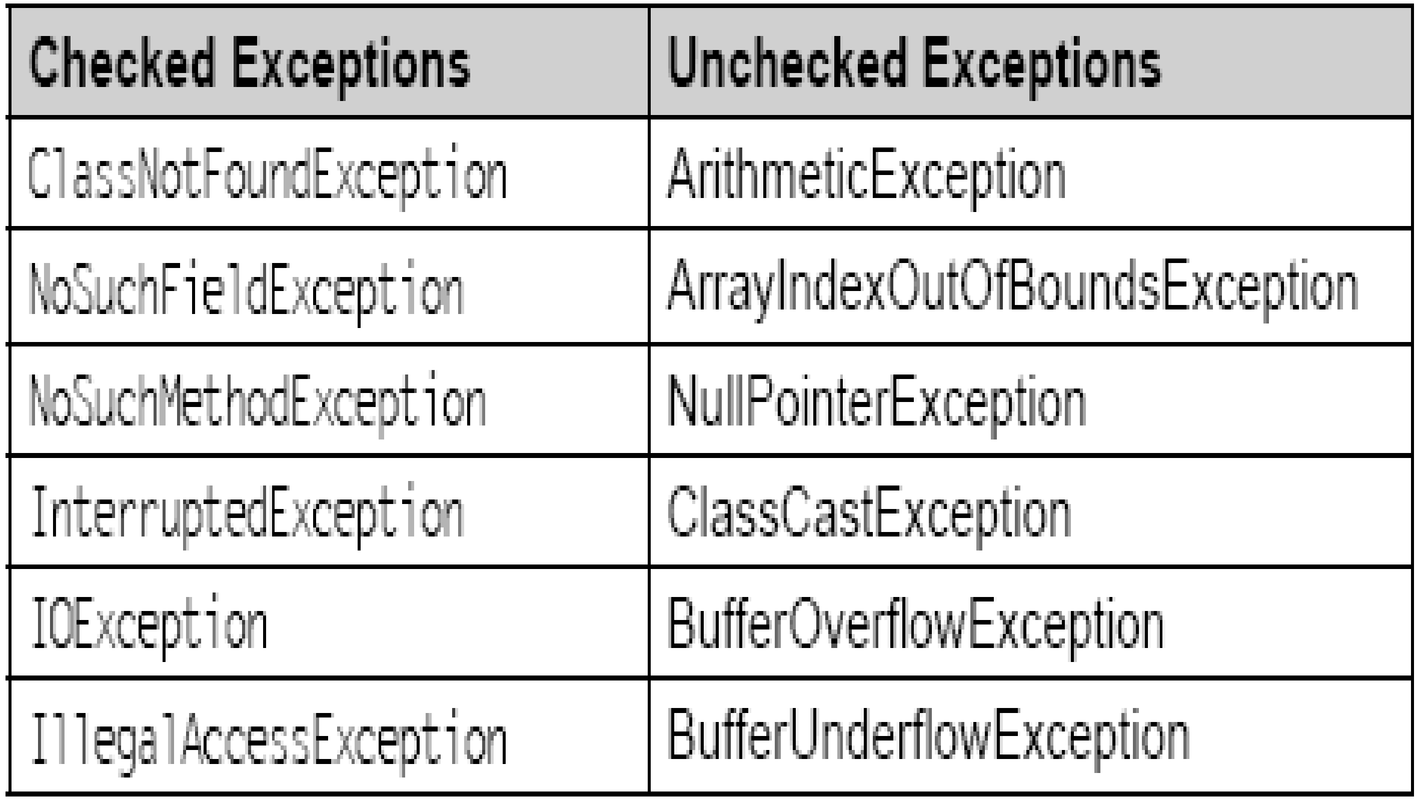
- runtime environment identifies the type of Exception and throws the object of it
- If the method does not employ any exception handling mechanism
  - the exception is passed to the caller method, and so on
- If no exception handling mechanism is employed in any of the Call Stack methods
  - the runtime environment passes the exception object to the default exception handler available with itself
  - The default handler prints the name of the exception along with an explanatory message followed by stack trace at the time the exception was thrown and the program is terminated

Stack Trace


ExDemo.method3(ExDemo.java:23)
class name
method name
FileName
Line No. where execution halted


method3
method2
method1
main

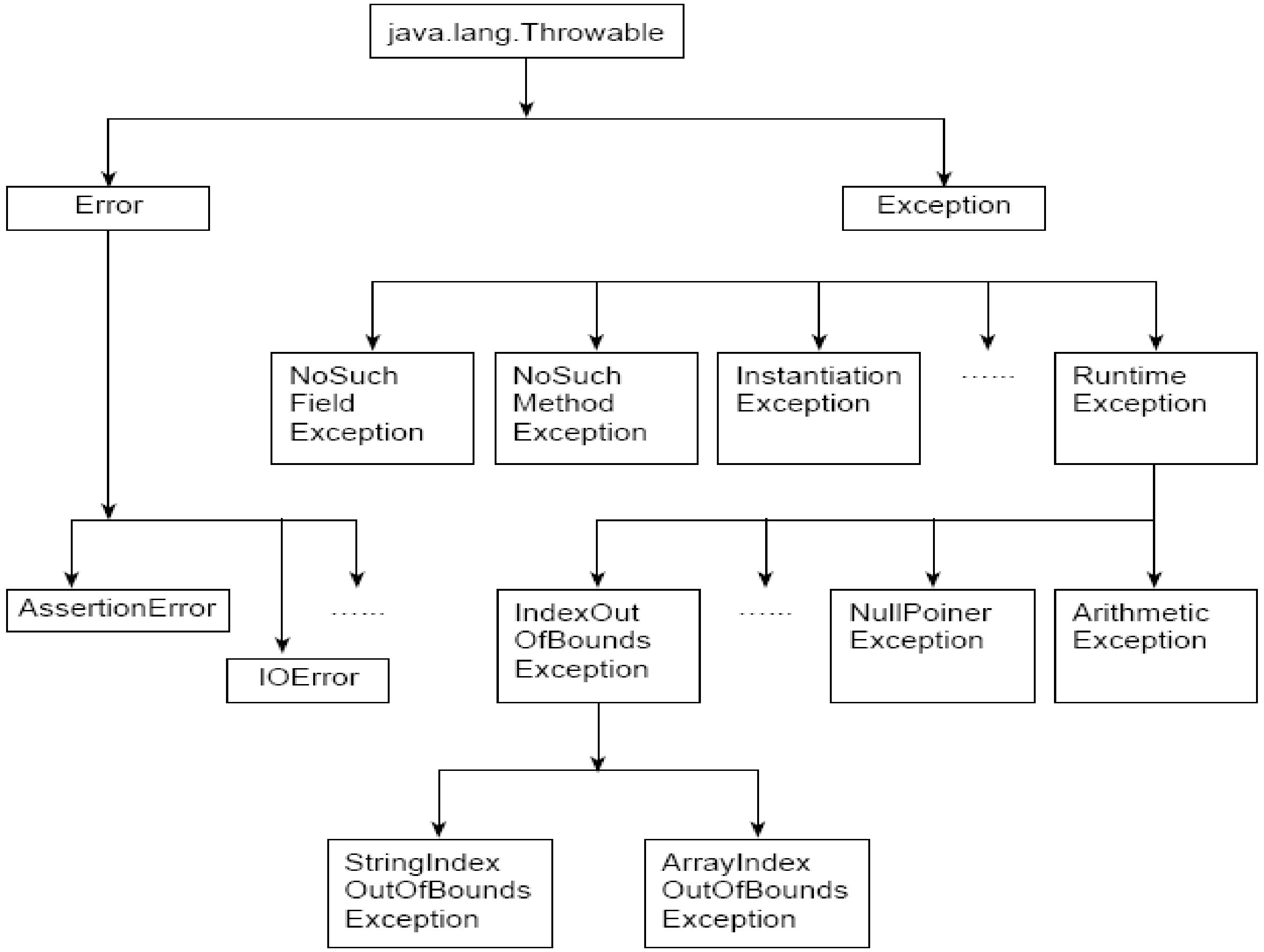
Types of Exceptions



Checked Exceptions	Unchecked Exceptions
ClassNotFoundException	ArithmeticException
NoSuchFieldException	ArrayIndexOutOfBoundsException
NoSuchMethodException	NullPointerException
InterruptedException	ClassCastException
IOException	BufferOverflowException
IllegalAccessException	BufferUnderflowException

SOMAIYA VIDYAVIHAR UNIVERSITY
K J Somaiya College of Engineering

Exception Hierarchy



Exception Handling Techniques

- try..catch
- throw
- throws
- finally

Packages

- Collection of classes and interfaces
- Provides a unique namespace for the classes
- Declaration resides at the top of a Java source file
- A package can contain the following.
    - Classes
    - Interfaces
    - Enumerated types
    - Annotations

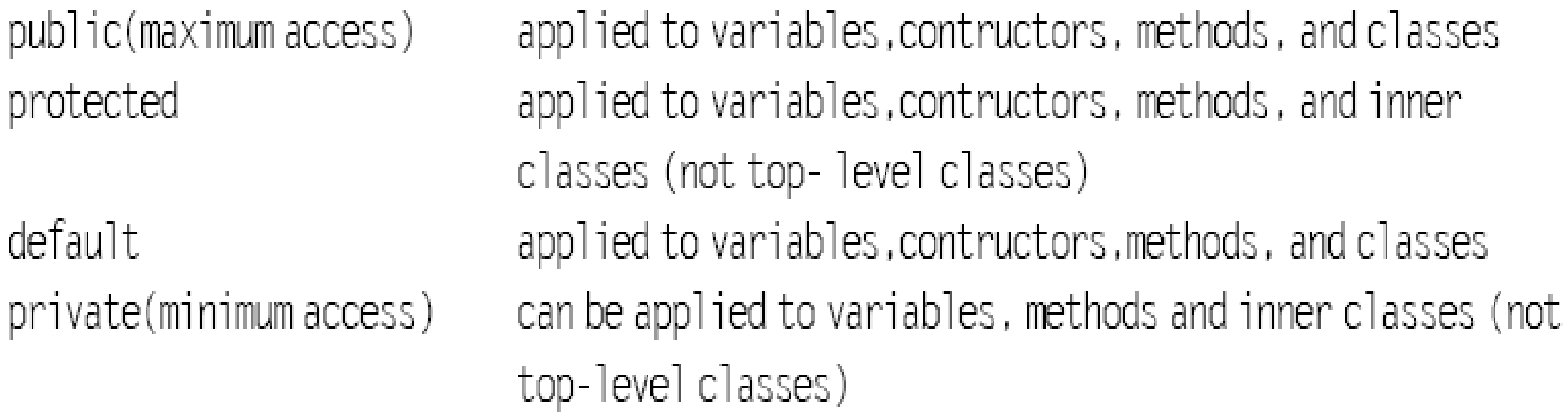
SOMAIYA
VIDYAVIHAR UNIVERSITY
K J Somaiya College of Engineering
Somaiya
TRUST

Packages (Contd.)

- Class that reside inside a package cannot be referred by their own name alone
- The package name has to precede the name of the class of which it is a part of
- All classes are part of some or the other package
- If the package keyword is not used in any class for mentioning the name of the package, then it becomes part of the default/unnamed package

Access Protection

- Four access specifiers in Java



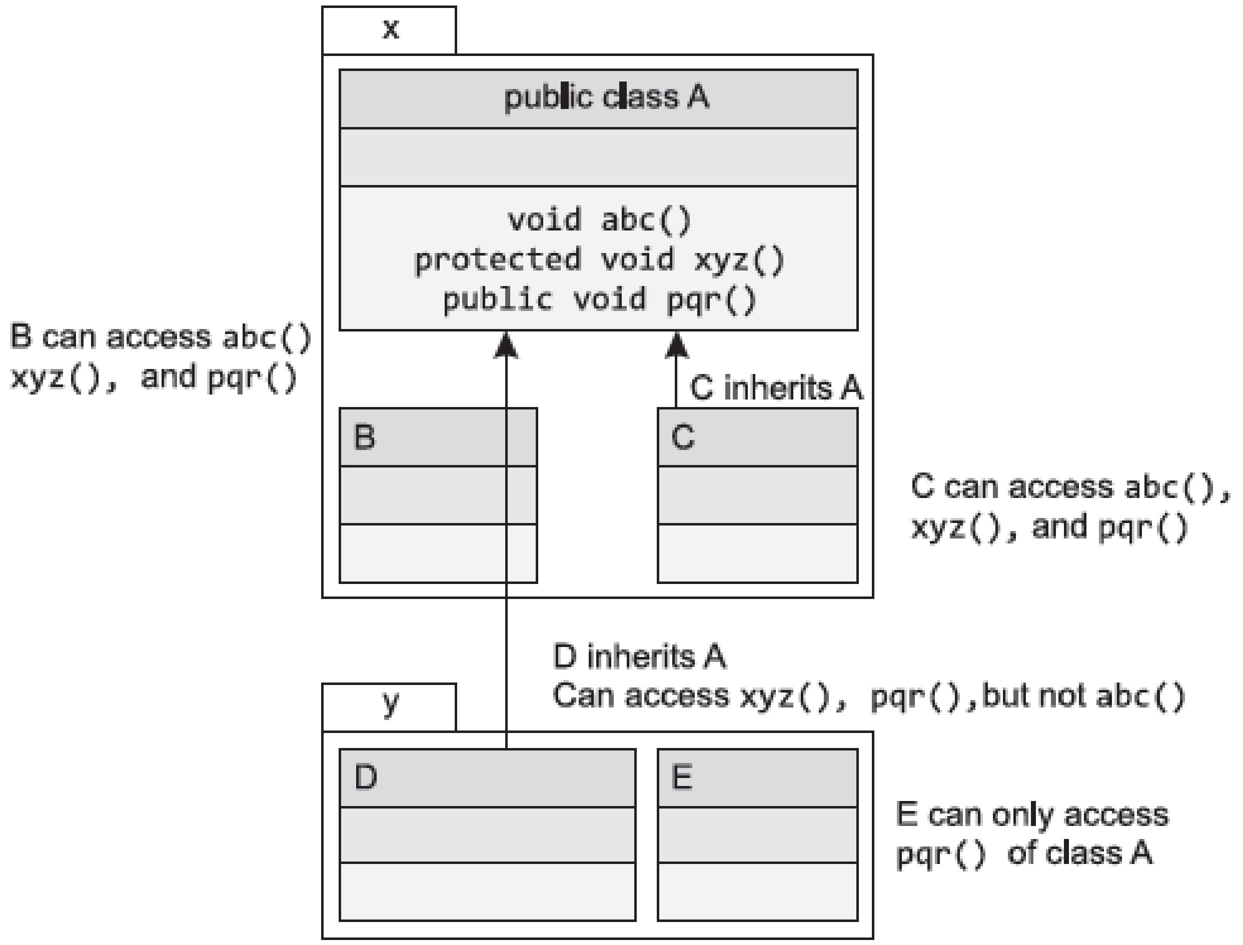
public(maximum access)	applied to variables,contructors, methods, and classes
protected	applied to variables,contructors, methods, and inner classes (not top- level classes)
default	applied to variables,contructors,methods, and classes
private(minimum access)	can be applied to variables, methods and inner classes (not top-level classes)

SOMAIYA
VIDYAVIHAR UNIVERSITY
K J Somaiya College of Engineering

Access Specifiers

- **Public:** Accessibility for all
- **Private:** Accessibility from within the class only
- **Default (blank):** Accessible only from within the package
- **Protected**: Accessibility outside the packages but only to subclasses

Access Protection


x
public class A
void abc()
protected void xyz()
public void pqr()
B can access abc() xyz(), and pqr()
C inherits A
B
C
C can access abc(), xyz(), and pqr()
D inherits A
Can access xyz(), pqr(),but not abc()
y
D
E
E can only access pqr() of class A

Thread Life Cycle

- In Java, a thread always exists in any one of the following states
- These states are:
    - New
    - Active
    - Blocked / Waiting
    - Timed Waiting
    - Terminated

Thread States

- **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
- **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**

Thread States(Contd.)

- **Runnable:**
  - A thread, that is ready to run is then moved to the runnable state
  - In the runnable state, the thread may be running or may be ready to run at any given instant of time
  - It is the duty of the thread scheduler to provide the thread time to run, i.e., moving the thread the running state
  - A program implementing multithreading acquires a fixed slice of time to each individual thread
  - Each and every thread runs for a short span of time and when that allocated time slice is over, the thread voluntarily gives up the CPU to the other thread, so that the other threads can also run for their slice of time
  - Whenever such a scenario occurs, all those threads that are willing to run, waiting for their turn to run, lie in the runnable state
  - In the runnable state, there is a queue where the threads lie

Thread States(Contd.)

- **Running**
  - When the thread gets the CPU, it moves from the runnable to the running state
  - Generally, the most common change in the state of a thread is from runnable to running and again back to runnable

Thread States(Contd.)

- **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
  - For example, a thread (let's say its name is A) may want to print some data from the printer. However, at the same time, the other thread (let's say its name is B) is using the printer to print some data. Therefore, thread A has to wait for thread B to use the printer. Thus, thread A is in the blocked state. A thread in the blocked state is unable to perform any execution and thus never consume any cycle of the Central Processing Unit (CPU). Hence, we can say that thread A remains idle until the thread scheduler reactivates thread A, which is in the waiting or blocked state

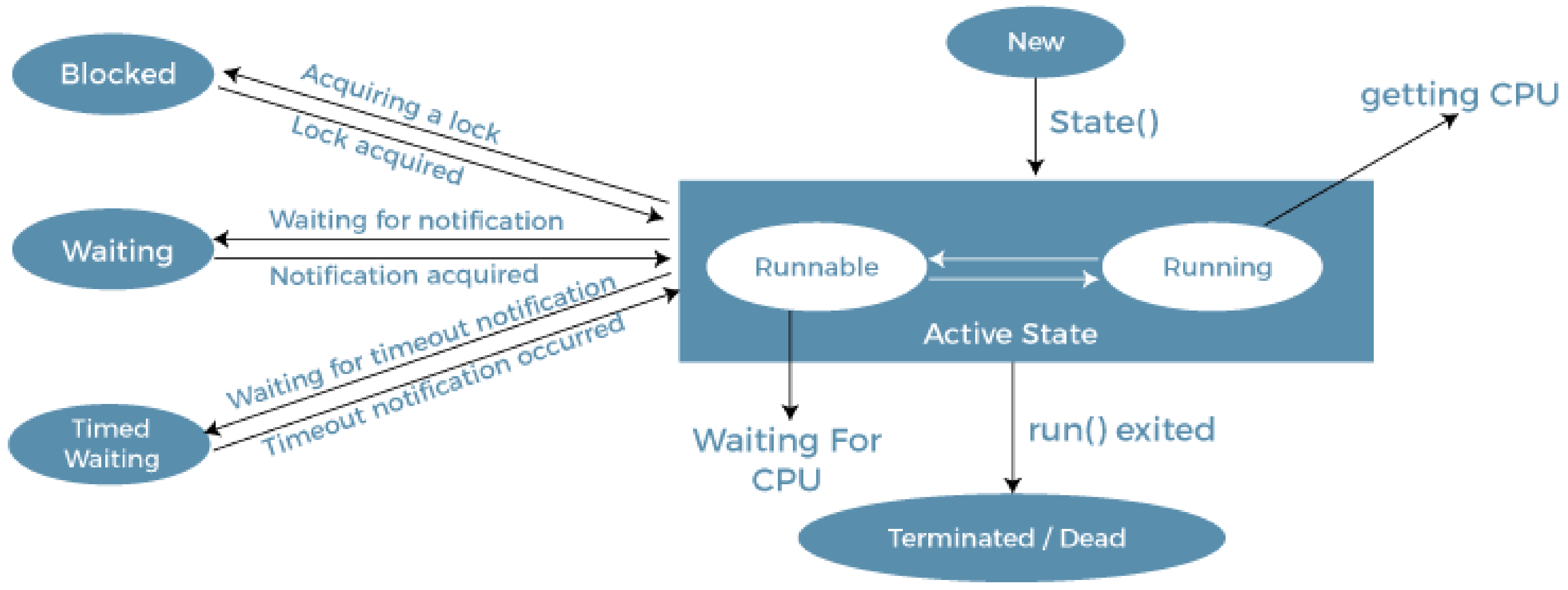
Thread States(Contd.)

- **Timed Waiting**
  - Sometimes, waiting for leads to starvation
  - For example, a thread (A) has entered the critical section of a code and is not willing to leave that critical section. In such a scenario, another thread (B) has to wait forever, which leads to starvation. To avoid such scenario, a timed waiting state is given to thread B. Thus, thread lies in the waiting state for a specific span of time, and not forever. A real example of timed waiting is when we invoke the sleep() method on a specific thread. The sleep() method puts the thread in the timed wait state. After the time runs out, the thread wakes up and start its execution from when it has left earlier.

Thread States(Contd.)

- **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

Life Cycle of a Thread



Life Cycle of a Thread

Multithreading

- A thread is a single sequential flow of control within a program
- Multitasking
    - Process Based
    - Thread Based

Process Vs Thread

- If we assume programs as processes, then process based multitasking is nothing but execution of more than one program concurrently
- While the thread based multitasking is executing a program having more than one thread, performing different tasks simultaneously
- Processes are heavyweight tasks
- Threads are light weight tasks

Thread based Multitasking: Multithreading

- Multithreading enables programs to have more than one execution paths (separate) which execute concurrently
- Each such path of execution is a thread
- Through multithreading, efficient utilization of system resources can be achieved, such as maximum utilization of CPU cycles and minimizing idle time of CPU

Thread class

- In Java, there is a class named as Thread class, which belongs to java.lang package, declared as,
- public class Thread extends Object implements Runnable
- This class encapsulates any thread of execution. Threads are created as the instance of this class, which contains run() methods in it. The functionality of the thread can only be achieved by overriding this run() method.
- public void run( ) {

```
        // statement for implementing thread
        }
```

Methods

- run() should be invoked by an object of the concerned thread
- This can be achieved by creating the thread and calling start() on that

Thread class Constructors

- Thread( )
- Thread(String threadName)
- Thread(ThreadGroup threadGroup, String threadName)

Main Thread

- Every Java program has a default thread, main thread
- When the execution of Java program starts, the JVM creates the main thread and calls the program's main() method within that thread
- Apart from this JVM also creates some invisible threads, which are important for JVM housekeeping tasks
- Programmers can always take control of the main thread

Main Thread (contd.)

- thread object can be created by using the currentThread() method, which returns a reference to the current thread
- The main thread can be controlled by this reference only

Thread creation

- By extending the Thread class
- By implementing the Runnable interface

Extending the Thread class

Steps to be undertaken for Thread creation

- Declare your own class as extending the Thread class
- Override the run( ) method, which constitutes the body of the thread
- Create the thread object and use the start( ) method to initiate the thread execution

Declaring a class

Any new class can be declared to extend the Thread class, thus inheriting all the functionalities of the Thread class.

```
class NewThread extends Thread
{
    ……………………………..
    ……………………………..
}
```

Here, we have a new type of thread, named as ‘NewThread’

Overriding the run( )method

- The run() method has been inherited by the class NewThread.
- The run() method has to be overridden by writing code required for the thread. The thread behaves as per this code segment.

```
public void run( )
    {
     //code segment providing the functionality of thread
    }
```

Starting New Thread

- Third part talks about start() method, required to create and initiate an instance of our thread class
- Following piece of code is responsible for the same

```
NewThread  thread1 = new NewThread( );
thread1.start( );
```

- First line creates an instance of class NewThread, where the object is just created. The thread is in newborn state
- Second line, which calls the start() method moves the thread in runnable state, where the java runtime will schedule the thread to run by invoking the run() method automatically

Implementing the Runnable interface

- Runnable interface is implemented by Thread class in the package java.lang
- This interface is declared as,
- public interface Runnable
- The interface needs to be implemented by any class whose instance is to be executed by a thread
- The implementing class must also override a void method named as run(), defined as a lone method in the Runnable interface as

```
public void run( )
{
…………………
}
```

Runnable interface (contd.)

- The object's run() method is called automatically whenever the thread is scheduled for execution by the thread scheduler
- The functionality of the thread depends on the code written within this run() method
- Other methods can be called from within run()
- The Thread will stop as soon as the run() exits

Runnable interface (contd.)

- Once a class that implements Runnable interface is created, an object of Thread class must be instantiated from within that class
- The constructors of Thread class helps in instantiating the object of Thread class
  - Thread(Runnable threadObj)
  - Thread(Runnable threadObj, String threadName)
  - Thread(ThreadGroup threadGroup, Runnable threadObj)
  - Thread(ThreadGroup threadGroup, Runnable threadObj, String threadName)

Runnable interface (contd.)

- Even if thread is created, it will not start executing unless the start() method of the Thread class is called
- This start() method when called, in turn calls the run()