Enlighters Academy

Multi-threading

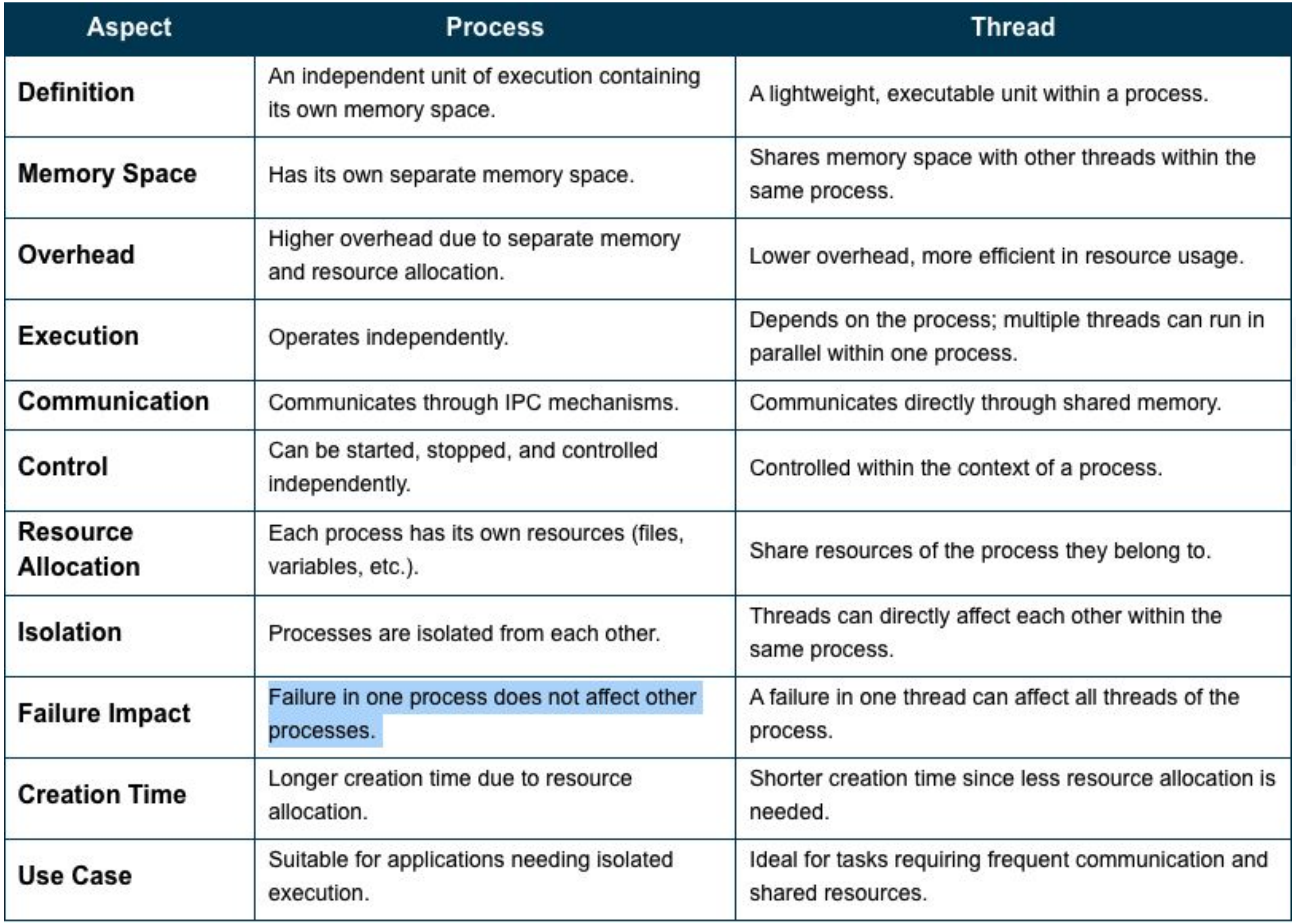
Why?

- Responsiveness
  - Resource intensive logics can be run as separate threads thus unblocking other executions
- Resource Sharing
  - threads share common code, data, and other resources, which allows multiple tasks to be performed simultaneously. Network and disk IO is often a lot slower than CPU's and memory IO
- Scalability
  - Utilization of multitasking architectures
- Independence
  - Threads are independent so it doesn't affect other threads if exception occur in a single thread

Concepts

- Process
  - Process is program under execution
  - Process related info is stored in PCB (Process Control block) for each process
- Threads
  - Thread is the segment of a process
  - Threads within the same process run in shared memory space

Process vs Thread



Aspect	Process	Thread
Definition	An independent unit of execution containing its own memory space.	A lightweight, executable unit within a process.
Memory Space	Has its own separate memory space.	Shares memory space with other threads within the same process.
Overhead	Higher overhead due to separate memory and resource allocation.	Lower overhead, more efficient in resource usage.
Execution	Operates independently.	Depends on the process; multiple threads can run in parallel within one process.
Communication	Communicates through IPC mechanisms.	Communicates directly through shared memory.
Control	Can be started, stopped, and controlled independently.	Controlled within the context of a process.
Resource Allocation	Each process has its own resources (files, variables, etc.).	Share resources of the process they belong to.
Isolation	Processes are isolated from each other.	Threads can directly affect each other within the same process.
Failure Impact	Failure in one process does not affect other processes.	A failure in one thread can affect all threads of the process.
Creation Time	Longer creation time due to resource allocation.	Shorter creation time since less resource allocation is needed.
Use Case	Suitable for applications needing isolated execution.	Ideal for tasks requiring frequent communication and shared resources.

Time to think

Which is Faster to Create: a Process or a Thread?

Creating a thread is generally faster than creating a process, as threads require less resource allocation.

When Should I Use a Process Over a Thread?

Processes are more suitable for tasks that require isolated execution environments, while threads are better for tasks that require frequent communication and shared resources.

Can a Single Process Have Multiple Threads?

Yes, a single process can have multiple threads, which can run in parallel, sharing the process's resources but maintaining separate execution sequences.

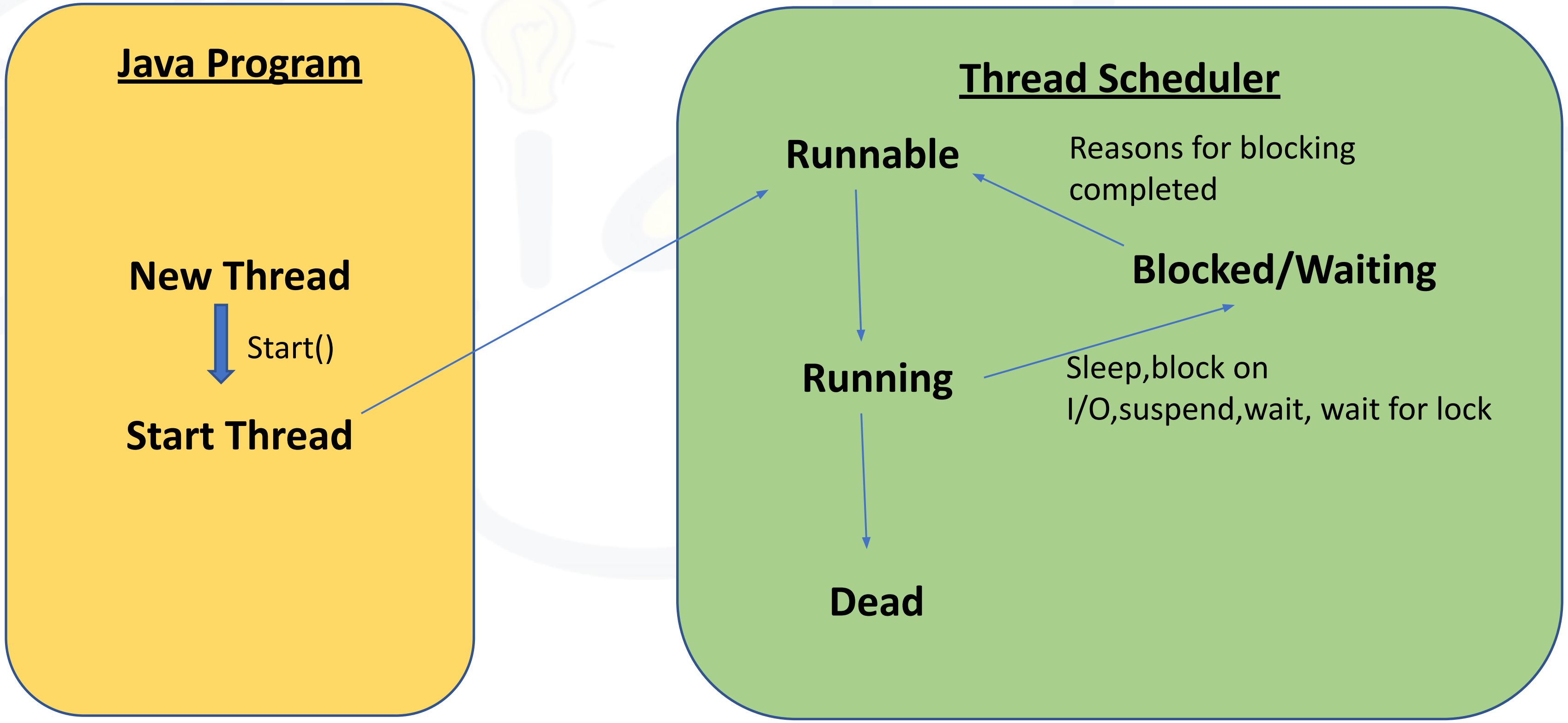
What is the Overhead Difference Between a Process and a Thread?

Processes have a higher overhead due to separate memory and resource allocation, while threads have lower overhead, making them more resource-efficient.

Threads in Java

- A thread is an execution path in a program
- Every program has at least one thread
- Each thread has its own stack, priority, and virtual set of registers
- Threads compete for processor time
- Scheduled by the operating system
- The scheduler maintains a list of ready threads, and decides which thread to run
- Threads are sometimes called *lightweight processes*. Both processes and threads provide an execution environment, but creating a new thread requires fewer resources than creating a new process.

Life cycle



Java Program

New Thread

Start()

Start Thread

Thread Scheduler

Runnable

Reasons for blocking completed

Blocked/Waiting

Running

Sleep,block on I/O,suspend,wait, wait for lock

Dead

Creating/starting a thread option-1

- Define a subclass of Thread
- Override its run() method
- Create an instance of the class
- Call its start() method
- Scheduler then calls its run() method
- run() is the entry point for the new thread. As soon as run() returns, the thread dies and cannot be restarted

Creating/starting a thread option-2

- **Implement the Runnable interface**
- Override its run() method
- Create an instance of the class
- **Create an instance of Thread class and pass above object as the parameter**
- Call its start() method
- Scheduler then calls its run() method

Creating/starting a thread option-3

- Create an instance of Thread class and define a anonymous inner class of Runnable interface
- Override its run() method
- call its start() method
- Scheduler then calls its run() method

Sleep

- Can cause a current thread to pause execution using Thread.sleep(<time in milli seconds>) or Thread.sleep(<time in milli seconds>, <time in nanoseconds)
- These sleep times are not guaranteed to be precise, because they are limited by the facilities provided by the underlying OS
- This method throws an InterruptedException when another thread interrupts the current thread while sleep is active

Join & isAlive

- The join method allows one thread to wait for the completion of the thread to which join is called
- Overloads of join allows to specify a waiting period
- Like sleep, join responds to an interruption by exiting with an InterruptedException

- final boolean isAlive() - returns true if the thread object is still not exited

Demon Threads & Thread priorities

- Daemon threads are usually used for background supporting tasks
- If normal threads are not running and remaining threads are daemon threads then the interpreter exits
- Demon thread is set by command <thread instance>. setDaemon(true)
- Daemon threads executes as a low priority thread
- Thread priorities are set by command <thread instance>.setPriority(Thread. MAX_PRIORITY/Thread. MIN_PRIORITY/Thread. NORM_PRIORITY)

Synchronization

- Synchronization in java is the capability *to control the access of multiple threads to any shared resource*.
- There are two types of thread synchronization mutual exclusive and inter-thread communication.
- Mutual Exclusive
  - Synchronized method
  - Synchronized block
  - static synchronization
- Cooperation (Inter-thread communication in java)

Synchronization: Locks

- Every object has a lock called a “monitor.”
- This monitor can be used to restrict access to Objects, because only one thread can own an object’s monitor at any one time.
- If a thread has ownership of an object monitor, then other threads attempting to do so are blocked until it is released

Synchronized Method

- If you declare any method after the access modifier with keyword “synchronized”, it is known as synchronized method.
- Calling an synchronized method acquires the lock on the object for modification

```
public class Counter {
        public synchronized void countNumbers(int limit) {
        //code block
        }
}
```

Synchronized Block

- Synchronized block can be used in two scenarios:
  1. When you want to synchronize acquiring the lock of a specific object
  2. You want to synchronize only a particular logic in a method

```
public class Counter {
        public void countNumbers(int limit) {
        //optional code block
        synchronized( <object>)
        }
}
```

Static Synchronization

- Making a static method as synchronized, the lock will be on the class and not on object
- If you have both non-static and static synchronized methods, a thread having a Object lock by calling non-static method will not block another thread calling the static methods as that lock is on the class level

```
public class Counter {
        public static synchronized void countNumbers(int limit) {
        //code block
        }
}
```

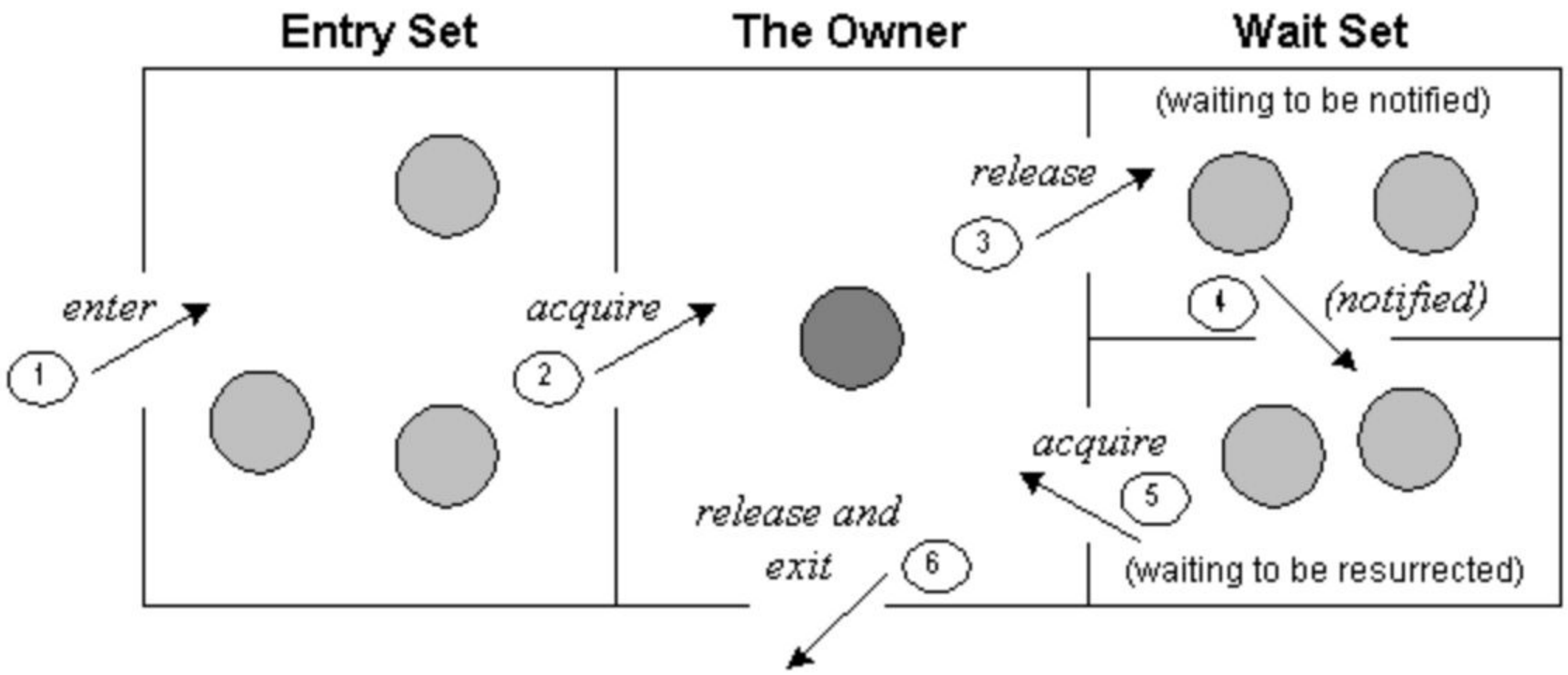
Dead Lock

- Multithreading with synchronization can cause Java Dead Lock if not coded properly
- Deadlock can occur in a situation when a thread is waiting for an object lock, that is acquired by another thread and second thread is waiting for an object lock that is acquired by first thread.
- Since, both threads are waiting for each other to release the lock, the condition is called deadlock

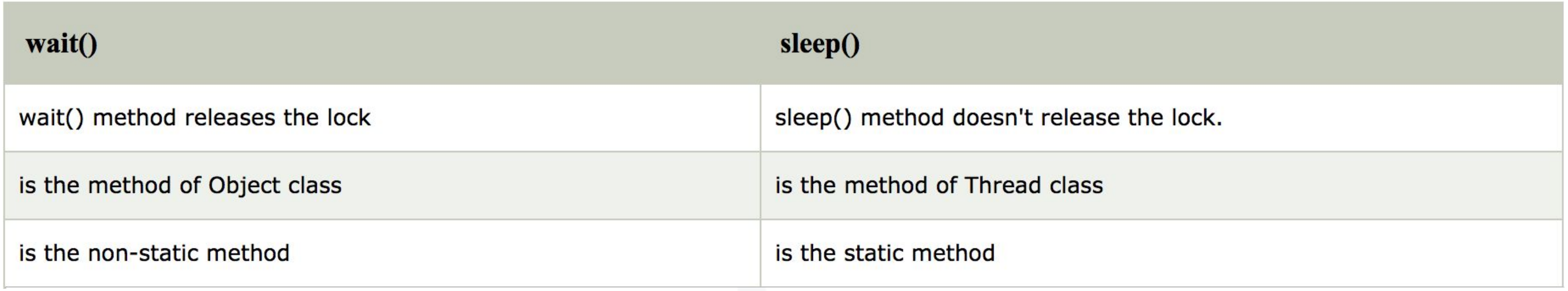
Inter-thread communication

- Allowing synchronized threads to communicate with each other
- a mechanism in which a thread is paused running in its critical section and another thread is allowed to enter (or lock) in the same critical section to be executed
- implemented by following methods of **Object class**
  - **wait() :** *public final void wait() throws InterruptedException*
    public final void wait(long timeout) throws InterruptedException
  - **notify() :** *public final void notify()*
  - **notifyAll():** *public final void notifyAll()*

Inter-thread communication



Difference between wait & sleep?



wait()	sleep()
wait() method releases the lock	sleep() method doesn't release the lock.
is the method of Object class	is the method of Thread class
is the non-static method	is the static method

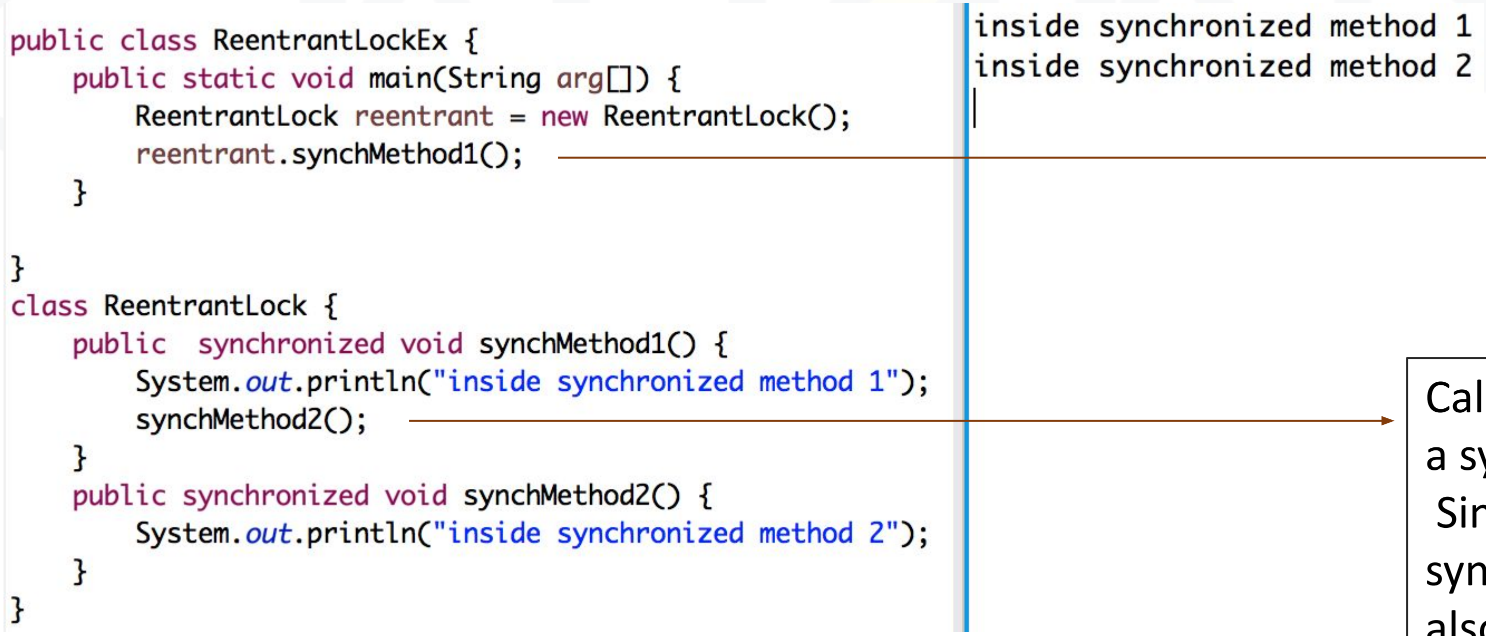
Interrupting a Thread

- If any thread is in sleeping or waiting state (i.e. sleep() or wait() is invoked), calling the interrupt() method on the thread, breaks out the sleeping or waiting state throwing InterruptedException

- Three methods of Thread class for Interruption:
  - **public void interrupt()**
  - **public boolean isInterrupted()**

Reentrant Monitor

- Java monitors are reentrant means java thread can reuse the same monitor/lock for different synchronized methods if another synchronized method is called from a synchronized method itself



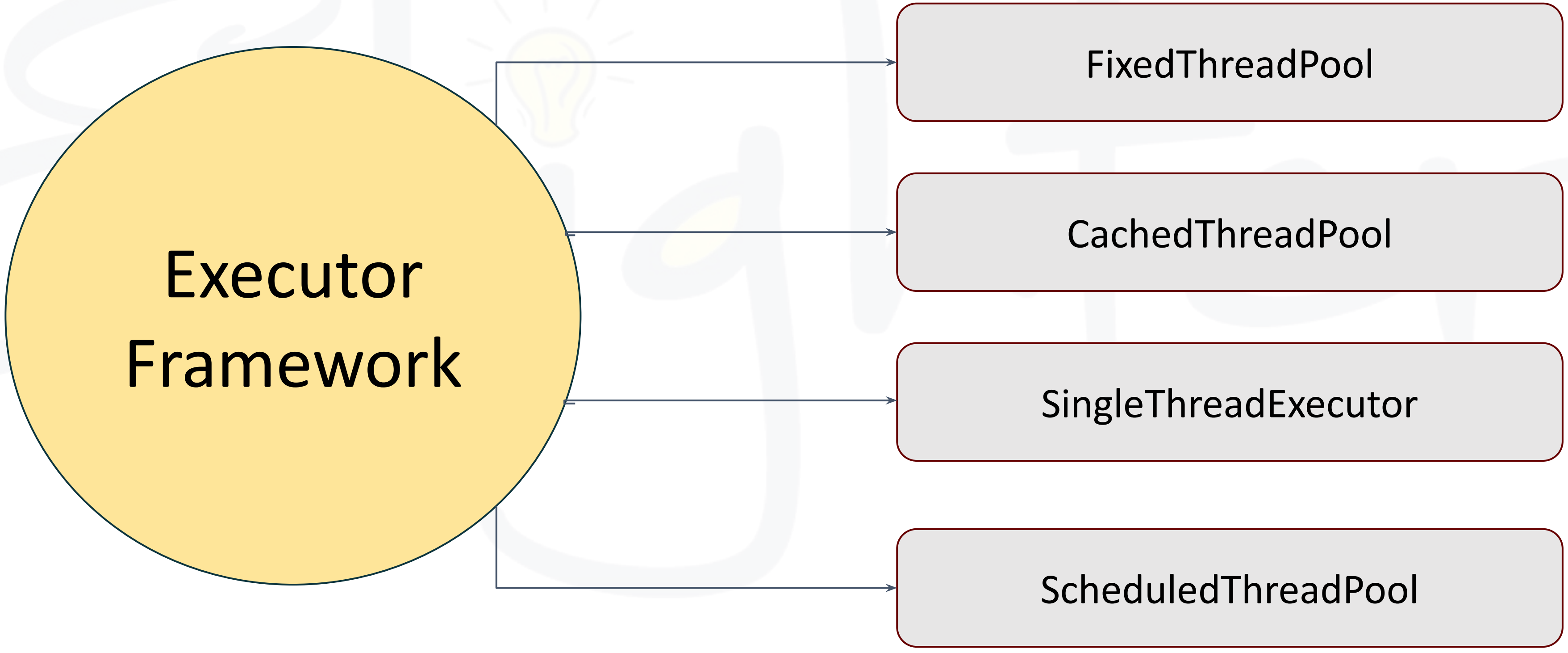
```
public class ReentrantLockEx {
    public static void main(String arg[]) {
        ReentrantLock reentrant = new ReentrantLock();
        reentrant.synchMethod1();
    }

}
class ReentrantLock {
    public  synchronized void synchMethod1() {
        System.out.println("inside synchronized method 1");
        synchMethod2();
    }
    public synchronized void synchMethod2() {
        System.out.println("inside synchronized method 2");
    }
}
```

```
inside synchronized method 1
inside synchronized method 2
```

Call to a synchronized method and hence acquiring lock of reentrant object

Call to another synchronized method from a synchronized method.
 Since Java locks are reentrant, synchronized method syncMethod2() is also been executed.

Executor Services



FixedThreadPool

- Thread pool with fixed number of threads
- If all threads in pool are busy, subsequent task are queued until a thread becomes available

Ex: Web Application where we want to limit the HTTP request always to a max value.

Pros	Cons
Guarantees a fixed number of threads	Resource wastage
Prevents resource exhaustion	Increased response time

CachedThreadPool

- Thread pool which creates new threads as needed
- Reuses the previously constructed threads if they are still available
- If any thread remains ideal for certain amount of time (60 sec), it will be terminated and removed from pool

Pros	Cons
Efficient for bursty workloads	Consistent high load can cause issue
Terminating idle threads	No explicit Thread Management

SingleThreadExecutor

- Max pool size is 1
- If we want processing sequentially instead of concurrently
- Allows us to submit multiple tasks still but executed one at a time

ScheduledThreadPool

- Schedule commands to run after a specified delay or to execute periodically
- ScheduleAtFixedRate
- ScheduleWithFixedDelay

Alerts !!

- Errors arising from incorrect thread synchronization can be very hard to detect, reproduce and fix
- Idle threads can eat up memory to keep its local stack.

Summary…

- Why multi threading?
- What is a thread in Java and its lifecycle?
- Different means of creating a Java thread?
- What are the different in build Thread methods?
- Synchronization and its different types?
- What is an Inter-thread communication?
- Difference between User Threads and Demon threads?