Introduction to Threads

Introduction to Parallel Programming

Why do we need parallel programming?

- Multiple tasks at once
- Processes
- Self-contained execution environment
- Private run-time resources and own memory space

What is a thread?

- Lightweight subprocess (fewer resources)
- Exists within a process (every process has at least one)
- Share process's resources

Creating Thread

- Extending Thread class:

```
public class MyThread extends Thread {

  @Override
  public void run() {
    System.out.println("Done in another thread.");
  }
}
```

Creating Thread(2)

- Implementing Runnable :

```
public class MyThread implements Runnable {

    @Override
    public void run() {
      System.out.println("Done in another
thread.");
    }
}
```

Starting and Stopping Thread

- Create an object and call **start()**.
- Beware of calling run() method. That will not start a new Thread !

```
// when extending thread
Thread thread = new MyThread();

// when implementing runnable
Thread thread2 = new Thread(new MyThread());

thread.start();
thread2.start();
```

Starting and Stopping Thread(2)

- You may use anonymous classes and lambdas as well:

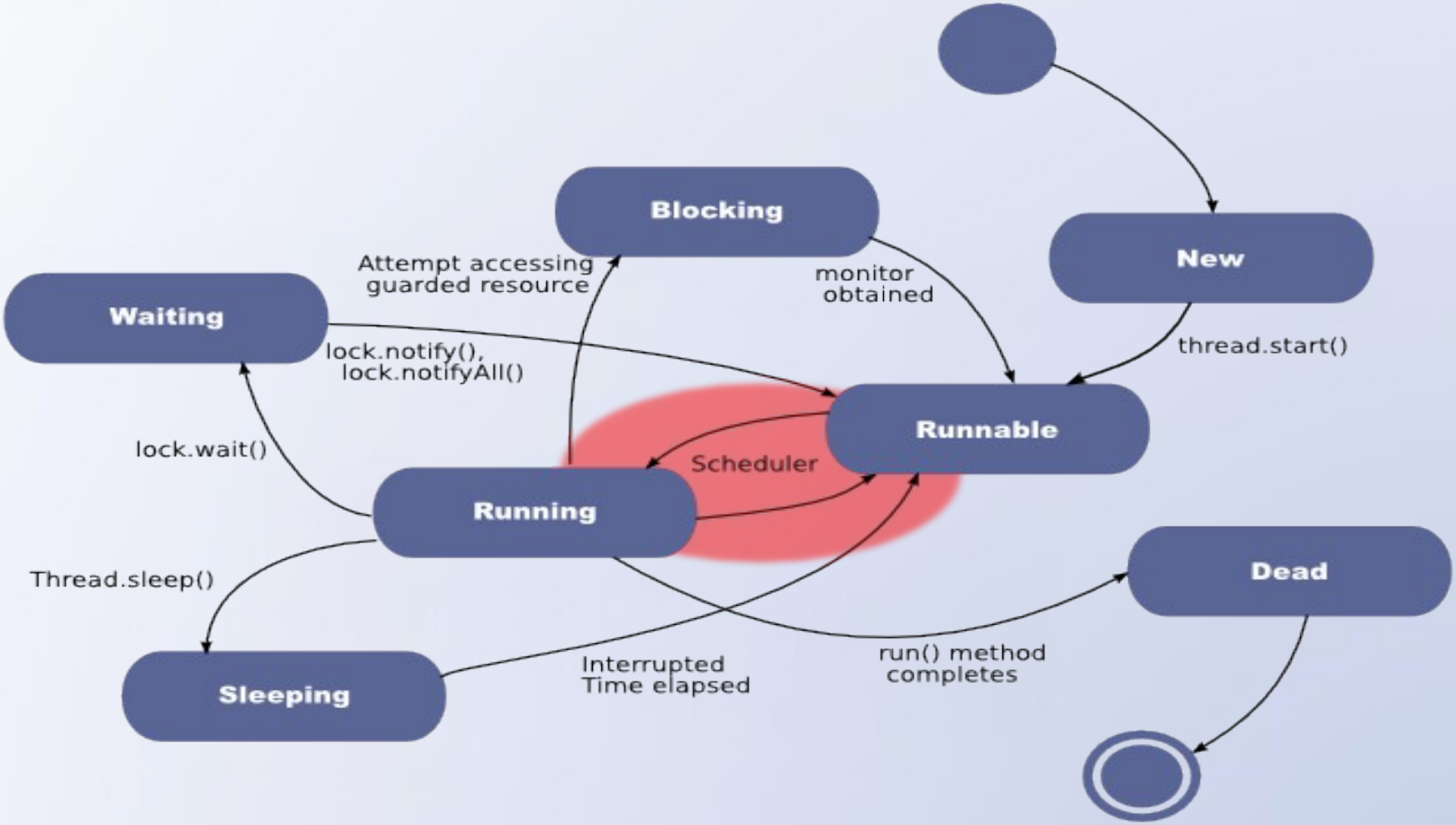
```
//using anonymous class
new Thread(new Runnable() {

   @Override
   public void run() {
     System.out.println("Do stuff");
   }
}).start();

//using lambda
new Thread(()->System.out.println("Do smth in another thread.")).start();
```

DEMO: Starting and Stopping Threads

Thread Scheduler

- Either the JVM or the underlying platform's operating system decides how to share the processor resource among threads.
- This task is known as **thread scheduling**.
- That portion of the JVM or operating system that performs thread scheduling is a **thread scheduler**.

Thread States (2)



More About Threads

- Every main() method starts a thread.
  - All our previous programs were single-threaded programs
- Every thread has its own call-stack.
- Every thread can be debugged as a separate program.

Thread Sleep

- Sleeping thread for a specified time
- Static method!
- Causes ONLY THE CURRENT thread to sleep

```
for (int i = 1; i <= 10; i++) {
  Thread.sleep(200);
  System.out.println("Counting... " + i);
}
```

Thread Interrupt

```
MyRunnable runnable = new MyRunnable();
Thread t1 = new Thread(runnable, "Thread 1");
Thread t2 = new Thread(runnable, "Thread 2");
Thread t3 = new Thread(runnable, "Thread 3");
Thread t4 = new Thread(runnable, "Thread 4");
Thread t5 = new Thread(runnable, "Thread 5");

t1.start();
t2.start();
t3.start();
t4.start();
t5.start();
Thread.sleep(2700);
t1.interrupt();
```

Thread Join

- Current thread waits the joined thread. Let's look at simple thread class:

```
public class MyRunnable implements Runnable {

 @Override
 public void run() {
  // getting current thread's data
  System.out.println(Thread.currentThread().getName());
 }
}
```

Thread Join(2)

```
MyRunnable runnable = new MyRunnable();

//specifying thread name as well
Thread t1 = new Thread(runnable, "Thread 1");
Thread t2 = new Thread(runnable, "Thread 2");
Thread t3 = new Thread(runnable, "Thread 3");

// starting 3 threads
t1.start();
t2.start();
t3.start();

// main thread waiting for thread 1 to finish
t1.join();
// main thread waiting for thread 2 to finish
t2.join();
// main thread waiting for thread 3 to finish
t3.join();

System.out.println("Main thread : all threads are dead.");
```

Thread Priority

- Each thread have a **priority**. Priorities are represented by a number between 1 and 10.
- In most cases, thread scheduler schedules the threads according to their priority (known as **preemptive scheduling**).
- But it is not guaranteed because it depends on JVM specification and which scheduling it'll choose.

Thread Priority(2)

3 constants defiend in Thread class:

public static int MIN_PRIORITY
public static int NORM_PRIORITY
public static int MAX_PRIORITY

Default priority of a thread is 5 (NORM_PRIORITY).

The value of MIN_PRIORITY is 1 and the value of MAX_PRIORITY is 10.

Thread Priority(3)

```
class Demo4 extends Thread {
public void run() {
System.out.println("running thread with name : "
+ Thread.currentThread().getName());

System.out.println("running thread with priority : "
+ Thread.currentThread().getPriority());
}

public static void main(String args[]) {
Demo4 m1 = new Demo4();
Demo4 m2 = new Demo4();

m1.setPriority(Thread.MIN_PRIORITY);
m2.setPriority(Thread.MAX_PRIORITY);
m1.start();
m2.start();
}
}
```

Daemon Thread

- Daemon thread in java is a service provider thread that provides services to the user thread.
- Its life depens on the mercy of user threads i.e. when all the user threads dies, JVM terminates this thread automatically.
- There are many java daemon threads running automatically e.g. gc, finalizer etc.

Daemon Thread(2)

```
public class Demo4 extends Thread {
public void run() {
if (Thread.currentThread().isDaemon()) {// checking for daemon thread
System.out.println("daemon thread work");
} else {
System.out.println("user thread work");
}
}

public static void main(String[] args) {
Demo4 t1 = new Demo4();// creating thread
Demo4 t2 = new Demo4();
Demo4 t3 = new Demo4();

t1.setDaemon(true);// now t1 is daemon thread

t1.start();// starting threads
t2.start();
t3.start();
}
}
```

Thread Groups

Java provides a convenient way to group multiple threads in a single object of class ThreadGroup.

In such way, we can interrupt group of threads by a single method call.

Contructors:

ThreadGroup(String name)
creates a thread group with given name.

ThreadGroup(ThreadGroup parent, String name)
creates a thread group with given parent group and name.

Thread Groups(2)

int activeCount() returns no. of threads running in current group.

void destroy() destroys this thread group and all its sub groups.

String getName() returns the name of this group.

ThreadGroup getParent() returns the parent of this group.

void interrupt() interrupts all threads of this group.

void list() prints information of this group to standard console.

DEMO: Thread Groups

Thread Pools

If you've several long running tasks that you want to load in parralel and then wait for the completion of all the tasks, it's getting little bit harder.

If you want to get the return value of all the tasks it becomes really difficult to keep a good code.

Java has a solution - **Executors**. This simple class allows you to create **thread pools** and **thread factories**.

Type of Thread Pools

Single Thread Pool : A thread pool with only one thread. So all the submitted task will be executed sequentially.

Cached Thread Pool : A thread pool that create as many threads it needs to execute the task in parralel. The old available threads will be reused for the new tasks. If a thread is not used during 60 seconds, it will be terminated and removed from the pool.

Fixed Thread Pool : A thread pool with a fixed number of threads. If a thread is not available for the task, the task is put in queue waiting for an other task to ends.

Scheduled Thread Pool : A thread pool made to schedule future task.

Single Thread Scheduled Pool : A thread pool with only one thread to schedule future task.

Working with Thread Pools

Creating a **Callable** and submitting **:**

```
private static class HiTask implements Callable<String> {
    public String call() {
        return "Hello world";
    }
}


for(int i = 0; i < 10; i++){
   pool.submit(new HiTask());
}

// don't forget to add this
pool.shutdown();
```

Getting Results

Getting a **Future** :

```
List<Future<String>> futures = new ArrayList<Future<String>>(10);

for(int i = 0; i < 10; i++){
   futures.add(pool.submit(new HiTask()));
}

for(Future<String> future : futures){
   // Waits for the computation to complete
   // and then retrieves its result
   String result = future.get();

   // work with result
}
```

Getting Results(2)

Working with **CompletionService :**

```
ExecutorService threadPool = Executors.newFixedThreadPool(4);

CompletionService<String> pool = new
ExecutorCompletionService<String>(threadPool);

for(int i = 0; i < 10; i++){
   pool.submit(new HiTask());
}

for(int i = 0; i < 10; i++){
   String result = pool.take().get();

   //Compute the result
}

threadPool.shutdown();
```

DEMO: Thread Pools