Java

Thread

Multitasking

- Multitasking allows several activities to occur concurrently on the computer
- Levels of multitasking:
  - **Process-based multitasking**
    - Allows programs (processes) to run concurrently
  - **Thread-base multitasking (multithreading)**
    - Allows parts of the same process (threads) to run concurrently

Multithreading

- Advantages of multithreading over process-based multitasking
  - Threads share the same address space
  - Context switching between threads is usually inexpensive
  - Communication between thread is usually inexpensive
- Java supports ***thread-based multitasking*** and provides high-level facilities for ***multithreaded programming***

Main Thread

- When a Java program starts up, one thread begins running immediately
- This is called the ***main thread*** of the program
- It is the thread from which the child threads will be spawned
- Often, it must be the last thread to finish execution

Main Thread

```
3   public class MainThread {
4       public static void main(String[] args) {
5           Thread t = Thread.currentThread();
6           System.out.println("Current thread: " + t);
7           // change the name of the thread
8           t.setName("My Thread");
9           System.out.println("After name change: " + t);
10          try
11          {
12              for(int n = 5; n > 0; n--)
13              {
14                  System.out.println(n);
15                  Thread.sleep(1000);
16              }
17          }catch (InterruptedException e)
18          {
19              System.out.println("Main thread interrupted");
20          }
21      }
22  }
```

How to create Thread

1. By extending the ***Thread*** class
2. By implementing ***Runnable*** Interface

- *Extending Thread*
  - Need to override the public void run() method
- *Implementing Runnable*
  - Need to implement the public void run() method
- Which one is better ?

Extending Thread

```
3   class NewThread2 extends Thread
4   {
5       NewThread2() {
6           super("Extends Thread");
7           start();
8       }
9       // This is the entry point for the thread.
10      public void run() {
11          try {
12              for(int i = 5; i > 0; i--) {
13                  System.out.println("Child Thread: " + i);
14                  Thread.sleep(500);
15              }
16          } catch (InterruptedException e) {
17              System.out.println("Child interrupted.");
18          }
19          System.out.println("Exiting child thread.");
20      }
21  }
22
23  public class ExtendsThread {
24      public static void main(String[] args) {
25          new NewThread2();
26      }
27  }
```

Implementing Runnable

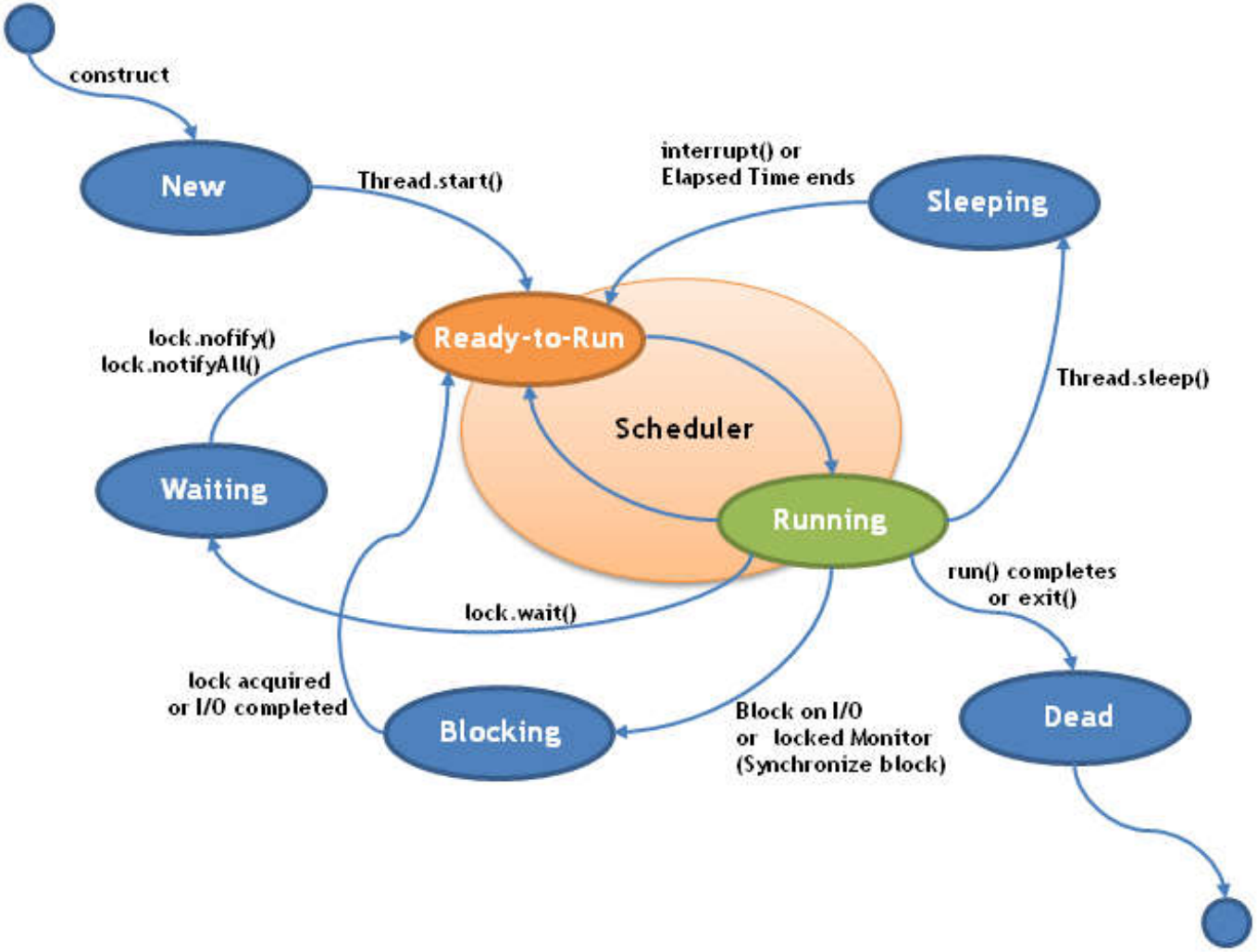
```
3   class NewThread1 implements Runnable
4   {
5       Thread t;
6       NewThread1() {
7           t = new Thread(this, "Runnable Thread");
8           t.start();
9       }
10      // This is the entry point for the thread.
11      public void run() {
12          try {
13              for(int i = 5; i > 0; i--) {
14                  System.out.println("Child Thread: " + i);
15                  Thread.sleep(500);
16              }
17          } catch (InterruptedException e) {
18              System.out.println("Child interrupted.");
19          }
20          System.out.println("Exiting child thread.");
21      }
22  }
23
24  public class RunnableThread {
25      public static void main(String[] args) {
26          new NewThread1();
27      }
28  }
```

Multiple Threads

- It is possible to create more than one thread inside the main
- In multiple threads, often you will want the main thread to finish last. This is accomplished by
  - using a large delay in the main thread
  - using the **join()** method
- Whether a thread has finished or not can be known using **isAlive()** method
- ***Example****: MultipleThreads.java, JoinAliveThreads.java*

Thread States

Source: https://avaldes.com/java-thread-states-life-cycle-of-java-threads/



Thread Pool

- Thread Pools are useful when you need to limit the number of threads running in your application
  - Performance overhead starting a new thread
  - Each thread is also allocated some memory for its stack
- Instead of starting a new thread for every task to execute concurrently, the task can be passed to a thread pool
  - As soon as the pool has any idle threads the task is assigned to one of them and executed

Thread Pool

- Thread pools are often used in multi threaded servers
  - Each connection arriving at the server via the network is wrapped as a task and passed on to a thread pool
  - The threads in the thread pool will process the requests on the connections concurrently
- Java provides Thread Pool implementation with ***java.util.concurrent.ExecutorService***

ExecutorService

```java
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

class MyRunnable implements Runnable {
    public void run() {
        System.out.println("Running task");
        for (int j = 5; j > 0; j--) {
            System.out.println(j);
        }
    }
}

public class ExecutorServiceTest {
    public static void main(String[] args) throws Exception{
        ExecutorService executorService = Executors.newFixedThreadPool( nThreads: 10);
        for (int i = 0; i < 20; i++) {
            executorService.execute(new MyRunnable());
        }
        executorService.shutdown();
    }
}
```

Callable and Future

- Runnable cannot return a result to the caller
- ***java.util.concurrent.Callable*** object allows to return values after completion
- Callable task returns a Future object to return result
- The result can be obtained using get() that remains blocked until the result is computed
- Check completion by isDone(), cancel by cancel()
- ***Example****: CallableFutures.java*

Synchronization

- When two or more threads need access to a **shared resource**, they need some way to ensure that the resource will be used by only one thread at a time
- The process by which this is achieved is called **synchronization**
- Key to synchronization is the concept of the **monitor**
- A monitor is an object that is used as a mutually exclusive lock
  - Only one thread can own a monitor at a given time

Synchronization

- When a thread acquires a lock, it is said to have entered the monitor
- All other threads attempting to enter the locked monitor will be suspended until the first thread exits the monitor
- These other threads are said to be waiting for the monitor

Synchronization

- Three ways to achieve synchronization.
- Synchronized method

  synchronized void call(String msg) { }
- Synchronized block

  public void run() {

  synchronized(target) { target.call(msg); } }
- Lock (java.util.concurrent package)
- ***Example****: SynchronizedBlock.java, SynchronizedMethod.java, SynchronizationLock.java*

Inter Thread Communication

- One way is to use polling
  - a loop that is used to check some condition repeatedly
  - Once the condition is true, appropriate action is taken
- Java includes an elegant inter thread communication mechanism via the **wait()**, **notify()** and **notifyAll()** methods
- These methods are implemented as final methods in Object, so all classes have them
- All three methods can be called only from within a synchronized method

Inter Thread Communication

- ***wait()***
  - tells the calling thread to give up the monitor and go to sleep until some other thread enters the same monitor and calls notify()
- ***notify()***
  - wakes up the first thread that called wait() on same object
- ***notifyAll()***
  - wakes up all the threads that called wait() on same object. The highest priority thread will run first
- ***Example****: IncorrectPC.java, CorrectPC.java, PCBlockingQueue.java*

Suspend, Resume and Stop

- Suspend
  - ***Thread t; t.suspend();***
- Resume
  - ***Thread t; t.resume();***
- Stop
  - ***Thread t; t.stop();***
  - Cannot be resumed later
- suspend and stop can sometimes cause serious system failures
- ***Example****: SuspendResume.java*