Thread Scheduling

- In a Java application, **main** is a thread on its own
- Once multiple threads are made Runnable
  - the thread scheduler of the Java Runtime Environment (JRE) decides
  - when each thread is scheduled to run
  - based on a simple algorithm called **fixed-priority scheduling**
    - this algorithm schedules threads on the basis of their priority **relative** to other Runnable threads
- Every thread has a priority associated with it
  - An int constant ranging between 1 (MIN_PRIORITY) and 10 (MAX_PRIORITY)
  - The default value is the middle value, 5 (NORM_PRIORITY)
  - When a thread is created, it inherits its priority from the thread that created it
Thread Priorities

- The JRE chooses the **Runnable** thread with the highest priority
  - Only when this thread stops, yields, or becomes not-Runnable does a lower priority thread get its chance to run
  - If multiple threads with the same priority are ready to run, the scheduler picks one randomly, and the chosen thread will run until one of the following happens:
    1. a higher priority thread becomes Runnable
    2. this thread yields or its run() method exits
    3. the *time-slicing* of the OS decides this thread’s time allocation is over

- At any given time, the highest priority thread is running. But there is NO guarantee! The thread scheduler may some times run a lower-priority thread to avoid starvation (we will soon see what this means)

- Therefore, use priorities for efficiency, but do NOT rely on them for 100% correct algorithm
Thread State Transitions

Thread $t = \text{new Thread}();$

- New
- Runnable
- Blocked
- Dead

This is a slightly simplified picture. For details, see the **Thread.State** enumerable type:

- [http://docs.oracle.com/javase/7/docs/api/java/lang/Thread.State.html](http://docs.oracle.com/javase/7/docs/api/java/lang/Thread.State.html)
Thread t = new Thread();

- Newly constructed Thread
- Not yet start()ed
- Not yet Runnable
- Not yet known to the thread scheduler

Runnable

- t.start()
- t.sleep(…)
- sleep time over!
- t.stop(), or run() method exits
- t.stop()

Blocked

Dead
Thread \( t = \) new Thread();

- After being `start()` ed
- Is Runnable
- Can be scheduled by the thread scheduler

Multiple threads can be in this state at a given time
Thread t = new Thread();

New

Runnable

Blocked

Dead

- t.start()
- t.sleep(...)
- sleep time over!
- t.stop(), or run() method exits
- t.stop()

- Thread is made un-Runnable by the sleep() method
- In this state, the thread is waiting for a monitor lock to enter (or re-enter) a synchronized block/method
Thread State Transitions

- **run()** method exits
- **stop()** method called
  - *Deprecated method. Do NOT use.*
  - *In fact, avoid deprecated methods as a rule of thumb*
- Once in the dead state, a thread cannot be scheduled again
  - Use isAlive() to take a pulse. A thread is said to be alive if it has been started and has not yet died

**Diagram: Thread State Transitions**

- **Runnable**
  - t.sleep(...)
- **Blocked**
  - sleep time over!
- **Dead**
  - t.stop(), or run() method exits
  - t.stop()
Executing a thread: `start()`ing or `run()`ning?

```java
public class StartRunning {
    private static class Wool extends Thread {
        @Override
        public void run() {
            System.out.println("Run me for the winter.");
        }
    }

    public static void main(String... args) {
        Wool t = new Wool();
        t.run();
        System.out.println("Put a break point on me and use the debugger to see how many threads are running.");
        t.start();
        System.out.println("How about now?");
        System.out.println("That was cozy!");
    }
}
```
Thread Interference

class Counter {
    private int c = 0;
    public void increment() { c++; }
    public void decrement() { c--; }
    public int value() { return c; }
}

- Even simple unary operations like `c++` are not atomic
  - It consists of three steps:
    i. Retrieve current value of `c`
    ii. Increment the retrieved value by 1
    iii. Store the incremented value back in `c`

- An atomic transaction is one that cannot be broken down into sub-transactions
- It either runs completely, or not at all
- In this context, a transaction is simply a code execution that changes stored data
- Parts of a code that need to be atomic (for multithreading to work properly) are called critical areas of the code
Thread Interference

class Counter {
private int c = 0;
public void increment() { c++; }
public void decrement() { c--; }
public int value() { return c; }
}

- Thread interference leads to the result of thread A being lost, as it is overwritten by thread B

- Suppose this application has two threads A and B
  - A calls increment() at around the same time as B calls decrement()
  - Their actions being not atomic, the actual sequence of transactions may be the following:
    i. A: retrieve c
    ii. B: retrieve c
    iii. A: increment() with result = 1
    iv. B: decrement() with result = -1
    v. A: store result in c; c is now 1
    vi. B: store result in c; c is now -1
Memory Inconsistency

- Memory consistency errors occur when different threads have inconsistent views of what should be the same data.
  - Can be avoided by understanding the `happens-before` relationship
  - Guarantees that memory writes by one statement are visible to another statement

- Example:

  ```java
  int c = 0; // int defined and initialized
  c++;
  System.out.println(c);
  ```

  - As before, suppose A and B are two threads
  - And A increments the value: `c++`
  - Then B prints the value: `System.out.println(c)`
  - The printed value may still be zero!
    - Because there is no guarantee that A’s action `happens-before` that of B
Creating the *happens-before* relation

- Invoking a thread’s `start()` method
  - Everything before calling `start()` is within a single thread, so transactions happen in the order of the code (i.e., line by line)
  - Every statement before calling `start()` *happens before* every statement of the new thread
- Terminating a thread and causing another thread to return via `join()`
  - Then, all the statements executed by the terminated thread have a *happens-before* relation with all the statements following the successful join
- But the most important way of establishing the happens-before relation is via *synchronization*
Synchronization

- Prevents **thread interference** and **memory inconsistency**
- But can cause **thread contention**
  - multiple threads try to access the same resource at the same time, and
  - cause the JRE to execute threads more slowly (or even suspend thread execution)
- Types of thread contention are
  - Starvation
  - Livelock
Synchronization

- Synchronization is built around an internal entity known as the **intrinsic lock** or **monitor lock**
  - Enforces exclusive access to an object’s state
  - Establishes *happens-before* relations required for correct visibility

- **Every** object has an intrinsic lock associated with it
  - A thread that requires exclusive access *acquires* this lock
  - And after accomplishing its task, *releases* this lock

- There are other types of locks as well, like the **ReentrantLock**
  - See the bank example in the code examples
Synchronization

Methods can be synchronized

```java
public class SynchronizedCounter {
    private int c = 0;
    public synchronized void increment() {
        c++;
    }
    public synchronized void decrement() {
        c--;
    }
    public synchronized int value() {
        return c;
    }
}
```

- Just add the `synchronized` keyword to the method declaration
- Each invocation to a synchronized method is atomic
  - When one thread is executing a synchronized method for this, all other threads that invoke synchronized methods for the same object suspend execution until the first thread is done
- When a synchronized method exits, it establishes a `happens-before` relation with any future invocation of a synchronized method for the same object
Synchronization

- Code blocks can be synchronized
- A synchronized statement can be used to acquire a lock on any object, not just this object, when executing a block of the code in a method:

  ```java
  synchronized (expr) {
      statements;
  }
  ```

- The expression `expr` must evaluate to an object reference
  - If the object is already locked by another thread, the thread is blocked until the lock is released
  - When a lock is obtained on the object, the statements in the synchronized block are executed, and then the lock is released
Liveness

- A concurrent application's ability to execute in a timely manner is known as its liveness.
- The most common problem of liveness is **deadlock**.
- Two other types of liveness problems:
  - a. Starvation
  - b. Livelock
The Dining Philosopher’s Problem

- Five silent philosophers sit at a table around a bowl of spaghetti, and a fork is placed between each pair of adjacent philosophers.
- Each philosopher must alternately think and eat.
- A philosopher can only eat spaghetti when he has both left and right forks.
- A philosopher can grab the fork on his right or the one on his left as they become available, but can't start eating before getting both of them.
- The possibility of a **deadlock**: if all 5 philosophers pick up the left fork at the same time and wait until the right fork is available, then no progress is possible again (**starvation**).
Deadlocks

A thread T1 holds a lock on L1 and wants lock L2

A thread T2 holds a lock on L2 and wants lock L1

Deadlock Resolution

- One technique: don’t let waiting threads lock other data!
- Release all their locks before making them wait
- There are all sorts of proper algorithms for thread lock ordering (beyond the scope of this course)
Starvation

- A situation where a thread is unable to gain regular access to shared resources and is unable to make progress
- This happens when shared resources are made unavailable for long periods by greedy threads. For example:
  - Suppose an object provides a synchronized method that often takes a long time to return
  - If one thread invokes this method frequently, other threads that also need frequent synchronized access to the same object will often be blocked
Livelock

- A thread T1 acts in response to the action of another thread T2
  - T2, in turn, acts in response to another thread T3
    - T3, in turn, acts in response to T4
    - ...
- **Livelock:** threads not blocked, but too busy responding to each other
  - The responses could be cyclical $T_1 \rightarrow T_2 \rightarrow T_3 \rightarrow \ldots \rightarrow T_1$
- A and B are walking in opposite directions in a corridor, and
  - A moves to his left to let B pass while B moves to her right to let A pass
  - Seeing that they are still blocking each other, A moves to his right, while B moves to her left
  - They're still blocking each other, so it goes on ...
Executors

- There is a close connection between
  - the task being done by a new thread (the Runnable object), and
  - the thread itself (the Thread object)

- Putting them together works fine for simple applications

- But at larger scales
  - separate thread management and thread creation from the rest of the application

- Objects that encapsulate these functions are known as **executors**

- Three executor object types defined by the executor interfaces
  1. Executor
  2. ExecutorService
  3. ScheduledExecutorService
## Executor Interfaces

The `java.util.concurrent` package defines three executor interfaces:

<table>
<thead>
<tr>
<th>Executor</th>
<th>ExecutorService</th>
<th>ScheduledExecutorService</th>
</tr>
</thead>
<tbody>
<tr>
<td>A simple interface that supports launching new tasks</td>
<td>Subinterface of <code>Executor</code> that adds features to manage the lifecycle of individual tasks as well as of the executor itself</td>
<td>A subinterface of <code>ExecutorService</code> that supports future and/or periodic execution of tasks</td>
</tr>
</tbody>
</table>
A **thread pool** is a **software design pattern** for concurrent programs.
Thread Pool and Executors

- The Executors class contains several methods for creation of pre-configured thread pool instances.
- A simple example of using Executors to acquire an Executor instance backed by a single thread pool and an unbounded queue for executing tasks sequentially:

```java
Executor executor = Executors.newSingleThreadExecutor();
executor.execute(() -> System.out.println("Hello World"));
```
- The Executor and ExecutorService interfaces are used to work with different thread pool implementations in Java.
  - Usually, it is a good idea to keep your code decoupled from the actual implementation of the thread pool and use these interfaces throughout your application.