High-level concurrency

Introduction

- Introduced in Java 5
- Out of the box representation of high-level concurrency concepts:
  - Locks
  - Executors
  - Concurrent collections
  - Atomic variables
  - Synchronizers

Locks

- Similar to implicit locks used with the synchronized keyword.
- Biggest advantage over implicit locks is the ability to back out of an attempt to acquire a lock through the **tryLock()** and **lockInterruptibly()** mehods.

Example: Locks

Executors

java.util.concurrent defines three interfaces:

- **Executor**, a simple interface that supports launching new tasks.
- **ExecutorService**, a subinterface of Executor, which adds features that help manage the lifecycle, both of the individual tasks and of the executor itself.
- **ScheduledExecutorService**, a subinterface of ExecutorService, supports periodic execution of tasks.

Executor interface

- Provides a single method **execute(Runnable)** which is a drop-in replacement for the common thread-creation idiom.

  Instead of:

  ```
  (new Thread(someRunnable)).start();
  ```

  you can use:

  ```
  anExecutor.execute(someRunnable);
  ```

- What actually happens in execute() depends on the concrete Executor implementation. E.g. the Runnable could be started immediately or it could be placed in a queue to be started later on.

ExecutorService interface

- Extends the Executor interface.
- Additionally provides methods that help manage the lifecycle, both of the submitted tasks and of the executor itself.
- Offers the more versatile **submit()** method that accepts Runnable much like execute() but also **Callable** which is similar to Runnable but allows a task to return a value.
- The submit() method returns an instance of the Future class which can be used to get the return value of the Callable task as well as control its lifecycle.

ScheduledExecutorService interface

- Extends the ExecutorService interface.
- Additionally provides methods to execute Runnable and Callable tasks after a specific delay or repeatedly at defined intervals.

Thread pools

- Thread creation is an expensive operation.
- Executors typically use thread pools rather than creating a new thread each time a task is executed or submitted.
- Examples:
  - `Executors.newSingleThreadExecutor()`
  - `Executors.newFixedThreadPool()`
  - `Executors.newCachedThreadPool()`
  - `Executors.newScheduledThreadPool()`
  - …

Example: Executors

Synchronizers

- A synchronizer is any object that coordinates the control flow of threads based on its state.
- Built-in synchronizers in Java (since version 5):
  - Latches
  - Barriers
  - Semaphores

Latches

- A latch is a synchronizer that can delay the progress of threads until it reaches its terminal state.
- A latch acts as a gate: until the latch reaches the terminal state the gate is closed and no thread can pass, and in the terminal state the gate opens, allowing all threads to pass.

Example: CountdownLatch

Barriers

- Barriers are similar to latches in that they block a group of threads until some event has occurred. The key difference is that with a barrier, all the threads must come together at a barrier point at the same time in order to proceed.
- Latches are for waiting for events; barriers are for waiting for other threads.
- Example: "Everyone meet at Record at 22:00; once you get there, stay there until everyone shows up, and then we'll figure out what we're doing next."

Example: CyclicBarrier

Semaphores

- Semaphores are used to control the number of activities that can access a certain resource or perform a given action at the same time.
- A Semaphore manages a set of virtual permits; the initial number of permits is passed to the Semaphore constructor. Activities can acquire permits (as long as some remain) and release permits when they are done with them. If no permit is available, acquire blocks until one is (or until interrupted or the operation times out). The release method returns a permit to the semaphore.
- Semaphores can be used to implement resource pools or to impose a bound on a collection.

Example: Semaphore

Exercises

Exercise: MaxResult

Modify the CountDownLatch example to execute "complex calculations" rather than "complex operations". Collect the results from ten different operations and print out the largest result.