Multithreading

Lecture 2

Topic Outline

- Concepts of tasks, threads and multithreading.
- Thread pools and executors.
- Thread synchronization.
- Thread cooperation.

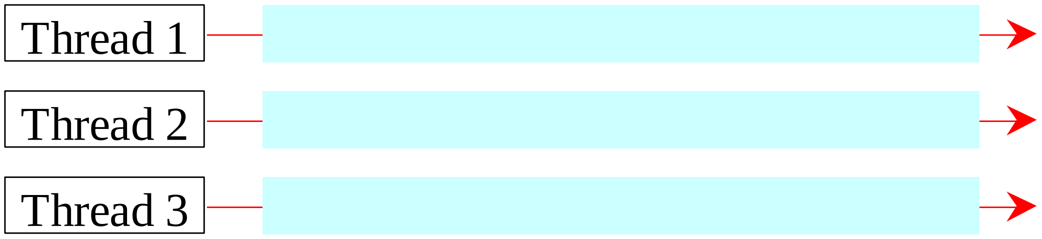
Multithreading

- Multithreading
  - powerful feature of Java

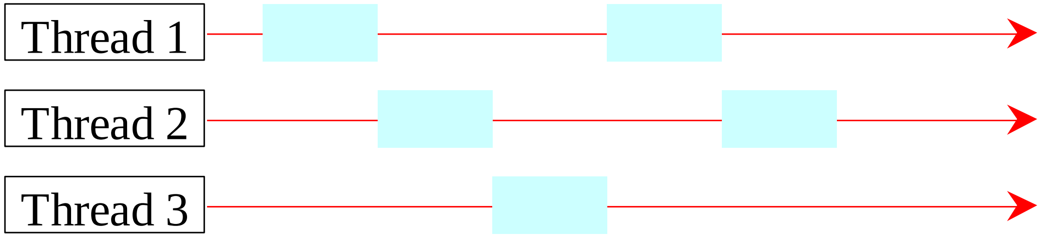
- Basic thread concept:
  - A **thread** is a flow of execution, from beginning to end, of a task in a program.
  - A **task** is a program unit that is executed independently of other parts of the program.
  - A **thread** provides the mechanism for running a **task**.

Concurrent Execution of Threads

- Multiple threads on multiple CPUs



- Multiple threads sharing a single CPU
  - Multiple threads share CPU time. The operating system is responsible for scheduling and allocating resources to threads.



Introduction to Multithreading

- Multithreading can make your program more responsive and interactive as well as enhance performance

- When a program executes as application, the Java interpreter starts a thread for the main method.

- In Java, each task is an instance of the **`Runnable`** interface, also called a runnable object. A thread is essentially an object that facilitates the execution of a task.

Creating Tasks & Threads

- Tasks are objects.

- To create tasks, you have to first declare a class of tasks. A task class must implement the **`Runnable`** interface.
  - **`Runnable`** interface contains a **`run`** method.
  - Once you have declared a task class class, you can create a task using its constructor

```
TaskClass task = new TaskClass();
```

Task Class

```
public class TaskClass implements Runnable {
...
      //constructor
      public TaskClass(){
            ...
      }

      //implement the run method in Runnable
      public void run(){
        // tell the system how to run
        // custom thread
            ...
      }
...
}
```

Creating Tasks & Threads

- A task must be executed in a method. The **`Thread`** class contains the constructor for creating threads and many useful methods for controlling threads.

```
Thread thread = new Thread(task);
```

- You can then invoke the **`start()`** method to tell the JVM that the thread is ready to run, as follows:

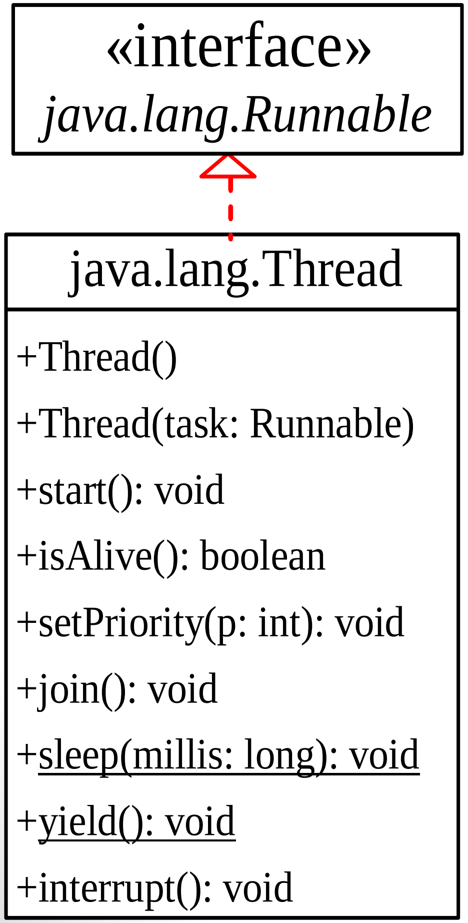
```
thread.start();
```

- The JVM will execute the task by invoking the task's **`run()`** method

TaskThreadDemo.java

Thread Class

- The **Thread** class contains the constructors for creating threads for tasks, and the methods for controlling threads



«interface»
java.lang.Runnable

java.lang.Thread

+Thread()	Creates a default thread.
+Thread(task: Runnable)	Creates a thread for a specified task.
+start(): void	Starts the thread that causes the run() method to be invoked by the JVM.
+isAlive(): boolean	Tests whether the thread is currently running.
+setPriority(p: int): void	Sets priority p (ranging from 1 to 10) for this thread.
+join(): void	Waits for this thread to finish.
+sleep(millis: long): void	Puts the runnable object to sleep for a specified time in milliseconds.
+yield(): void	Causes this thread to temporarily pause and allow other threads to execute.
+interrupt(): void	Interrupts this thread.

Thread Class

- Since the **Thread** class implements **Runnable**, you could declare a class that extends Thread and implements **run()** method.

- Then, create an object from the class and invoke its start method in a client program to start the thread.

TaskThreadDemo2.java

yield() Method

- You can use the **`yield()`** method to temporarily release time for other threads.

- In the **TaskThreadDemoYield.java** program:
  - Every time a number is printed, the **print100** thread is **yielded**.
  - So, the numbers are printed after the characters.

sleep() Method

- The **sleep(long)** method is used to put the thread to sleep for the specified time in milliseconds to allow other threads to execute.
- The **sleep(long)** method may throw an **InterruptedException**, which is a checked exception.
- This exception may occur when a sleeping thread's **interrupt()** method is called.
- We usually don't interrupt the thread so **InterruptedException** is unlikely to occur.

TaskThreadDemoSleep.java

join() Method

- The **`join()`** method is used to force one thread to wait for another thread to finish

- In the **TaskThreadDemoJoin.java** program:
  - The numbers after 25 are printed after thread **printY** is finished.

Thread Priority

- By default, a thread inherits the priority of the thread that produced it.
- You can increase or decrease the priority of any thread by using the **`setPriority()`** method.
- You can also get the thread's priority by using **`getPriority()`** method.
- Priorities are numbers ranging from 1 to 10.
- There are constant like **`MIN_PRIORITY`**, **`NORM_PRIORITY`** and **`MAX_PRIORITY`**, representing 1, 5, and 10.

Thread Priority

- The JVM always picks the currently runnable thread with the highest priority.
- If several runnable threads have equally high priority, the CPU is allocated to all in round-robin fashion.
- Of course, a low priority thread can only be run when no higher-priority threads are running.

TaskThreadDemoPriority.java

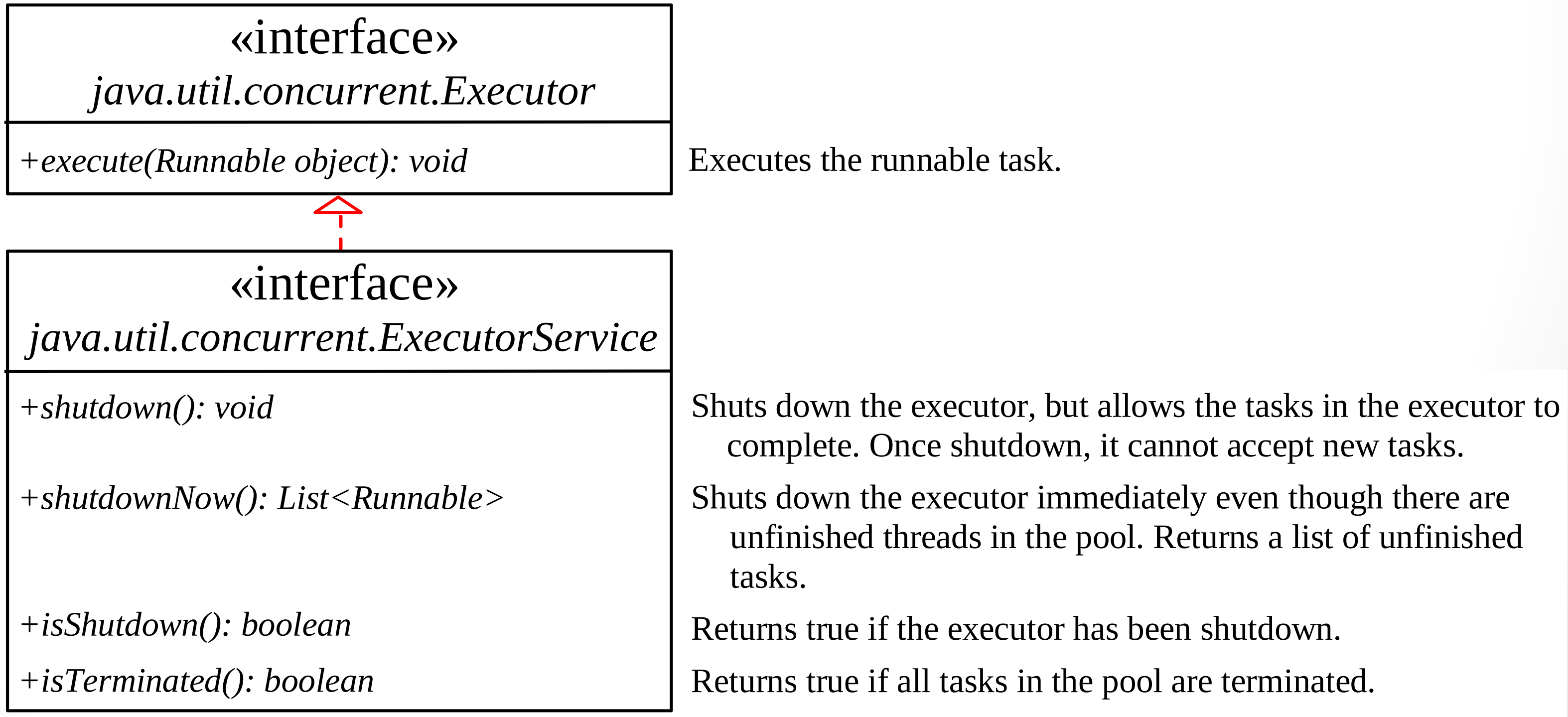
Thread Pools

- You have seen how to declare a task class by implementing **`Runnable`**, and how to create a thread to run a task.

- This approach is convenient for a single task execution, but it isn't efficient for large number of tasks.

Thread Pools

- A **thread pool** is ideal to manage the number of tasks executing concurrently.
- Java uses the **`Executor`** interface for executing tasks in a thread pool and the **`ExecutorService`** interface for managing and controlling tasks.



«interface»
java.util.concurrent.Executor

+execute(Runnable object): void

Executes the runnable task.

«interface»
java.util.concurrent.ExecutorService

+shutdown(): void

Shuts down the executor, but allows the tasks in the executor to complete. Once shutdown, it cannot accept new tasks.

+shutdownNow(): List<Runnable>

Shuts down the executor immediately even though there are unfinished threads in the pool. Returns a list of unfinished tasks.

+isShutdown(): boolean

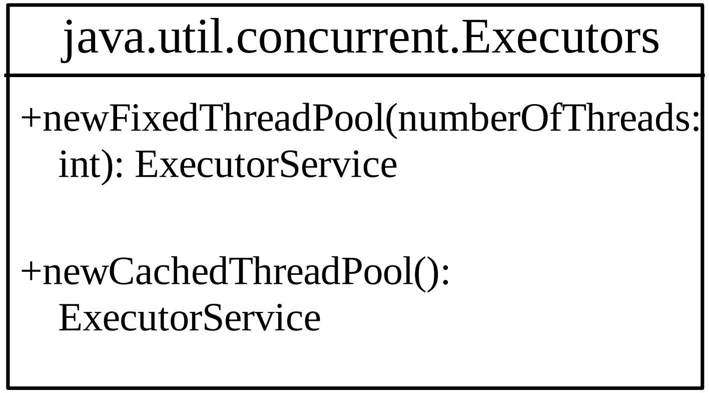
Returns true if the executor has been shutdown.

+isTerminated(): boolean

Returns true if all tasks in the pool are terminated.

Creating Executors

- To create an **Executor** object, use the static methods in the **Executors** class.



java.util.concurrent.Executors

+newFixedThreadPool(numberOfThreads: int): ExecutorService

+newCachedThreadPool(): ExecutorService

Creates a thread pool with a fixed number of threads executing concurrently. A thread may be reused to execute another task after its current task is finished.

Creates a thread pool that creates new threads as needed, but will reuse previously constructed threads when they are available.

ExecutorDemo.java

Thread Synchronization

- A shared resource may be corrupted if it is accessed simultaneously by multiple threads.
- The program **AccountWithoutSync.java** demonstrate the data corruption problem as all the threads have access to the same data source simultaneously.
- You will see the result is unpredictable.

Thread Synchronization

- In **AccountWithoutSync.java**, Thread 1 and Thread 2, etc are accessing a common resource in a way that causes conflict.

- This is common problem, known as *race condition* in multithreaded program.

- A class is said to be thread-safe if an object of the class does not cause a race condition in the presence of multiple threads.

- Account class is not thread-safe in the example.

Thread Synchronization

- To avoid race condition, it is necessary to prevent more than one thread from simultaneously entering a certain part of the program (known as *critical region*)

- The critical region in **AccountWithoutSync.java** is **`deposit()`** method.

- Use the keyword **`synchronized`** to synchronize the method so that only one thread can access the method at a time.

`synchronized` Method

- A synchronized method acquires a lock before it executes.

- There are 2 cases:
  - The instance method, the lock is on the object when its invoked.
  - The static method, the lock is on the class.

synchronized Method

- If a thread invokes a synchronized instance method on an object, the lock of that object is acquired first, then the method is executed, and finally the lock is released.

- Another thread invoking the same method of that object is blocked until the lock is released.

- Put the synchronized keyword on the **deposit** method as follows:

```
public synchronized void deposit(int amount) {...}
```

AccountWithSyncMethod.java

synchronized Statement

- 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.

- This is referred to as a synchronized block.

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

`synchronized` Statement

- 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, then the lock is released.

synchronized Statement

- Synchronized statements enable you to synchronize part of the code in a method instead of the entire method.
  - This increases concurrency.

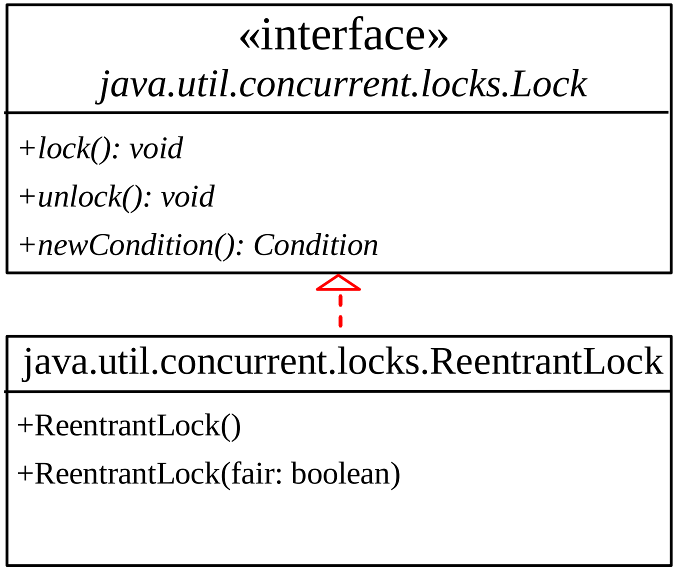
- Synchronized statements enable you acquire a lock on any object so that you can synchronize the access to an object instead of to a method.

```
synchronized (account){
    account.deposit(1);
}
```

AccountWithSyncStatement.java

Synchronization Using Locks

- A synchronized instance method implicitly acquires a lock on the instance before it executes the method.
- With the use of locks explicitly, the locking features are flexible and give you more control for coordinating threads.



«interface»
java.util.concurrent.locks.Lock

+lock(): void
+unlock(): void
+newCondition(): Condition

java.util.concurrent.locks.ReentrantLock

+ReentrantLock()
+ReentrantLock(fair: boolean)

Acquires the lock.

Releases the lock.

Returns a new Condition instance that is bound to this Lock instance.

Same as ReentrantLock(false).

Creates a lock with the given fairness policy. When the fairness is true, the longest-waiting thread will get the lock. Otherwise, there is no particular access order.

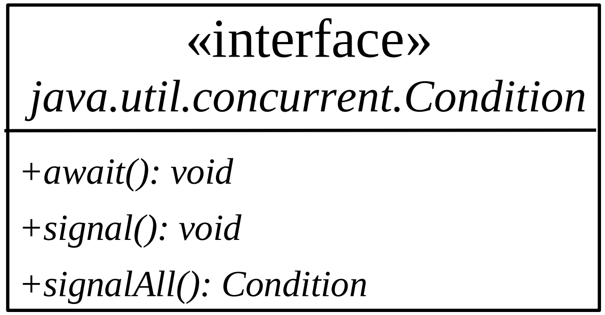
AccountWithSync.java

Thread Cooperation

- Sometimes, you need a way for threads to cooperate.
- Conditions can be used to facilitate communications among threads. The thread specify what to do under a certain condition.
- Once condition is created, you can use its **`await()`**, **`signal()`**, and **`signalAll()`** methods for thread communications.

ThreadCooperation.java

Thread Cooperation



«interface»

java.util.concurrent.Condition

+*await(): void* — Causes the current thread to wait until the condition is signaled.

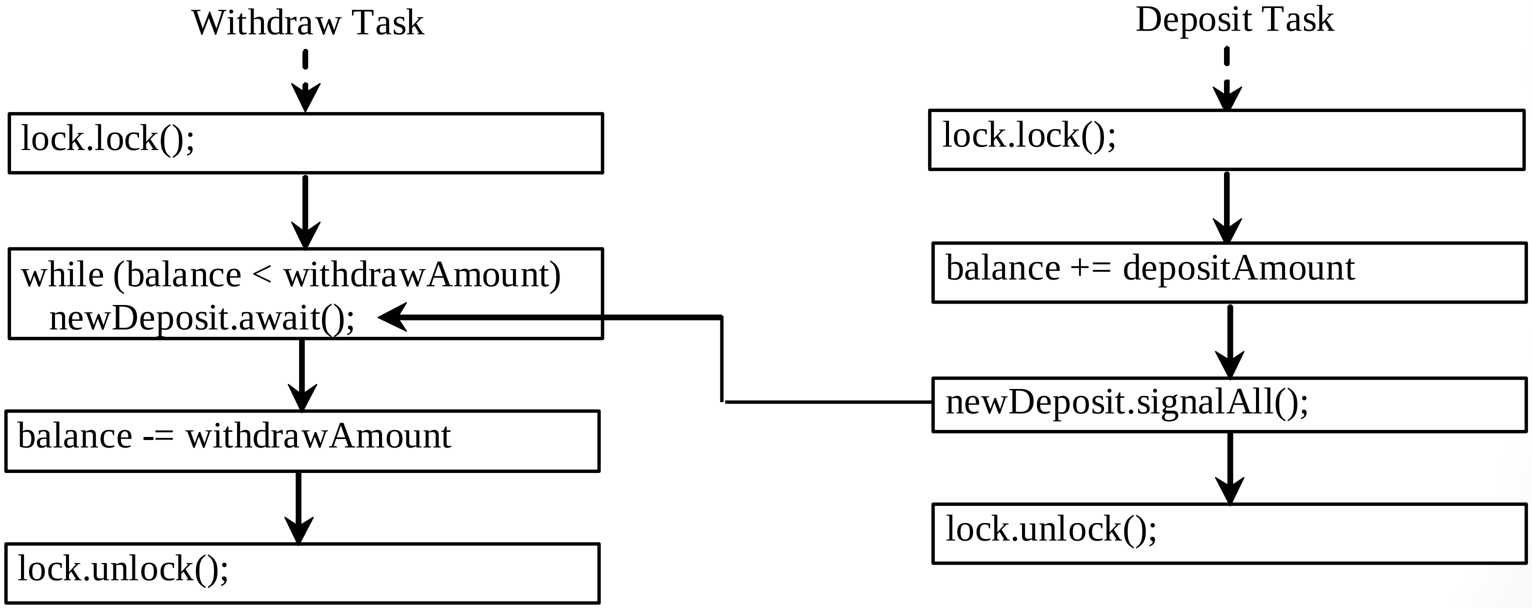
+*signal(): void* — Wakes up one waiting thread.

+*signalAll(): Condition* — Wakes up all waiting threads.

Thread Cooperation

- To synchronize the operations, use a lock with a condition.
- Example:
  - Condition: **`newDeposit`** (i.e., new deposit added to the account).
  - If the balance is less than the amount to be withdrawn, the withdraw task will wait for the **`newDeposit`** condition.
  - When the **`deposit`** task adds money to the account, the task signals the waiting withdraw task to try again.

Thread Cooperation



Withdraw Task

lock.lock();

while (balance < withdrawAmount)
  newDeposit.await();

balance -= withdrawAmount

lock.unlock();

Deposit Task

lock.lock();

balance += depositAmount

newDeposit.signalAll();

lock.unlock();