LECTURE 4: ADVANCED CONCURRENCY IN THE JAVA LANGUAGE
Lecture Contents

• Recap on Threads and Monitors in Java
  – Example: Queue Class based on Monitors
  – Example: ReadersWriters Class

• Features in java.util.concurrent:
  – Semaphore class, Example 3: Thread Throttling
  – Lock/Class Condition Objects
    • Example: Bounded Buffers
    • Example: Dining Philosopers using ReentrantLocks
    • Example: Bank Account Implementation
  – Interface Executor, ForkJoin, Future etc
  – Concurrent Annotations
SECTION 4.1: MONITORS IN JAVA
Monitors in Java

• Java implements a slimmed down version of monitors.
• Java's monitor supports two kinds of thread synchronization: mutual exclusion and cooperation:
  – Mutual exclusion:
    • Supported in JVM via object locks (aka ‘mutex’),
    • Enables multiple threads to independently work on shared data without interfering with each other.
  – Cooperation:
    • Supported in JVM via the `wait()`, `notify()` methods of class `Object`,
    • Enables threads to work together towards a common goal.
Monitors in Java: Recap on Threads (/2)

- Java thread is a lightweight process with own stack & execution context, access to all variables in its scope.
- Can be programmed by extending `Thread` class or implementing `Runnable` interface.
- Both of these are part of standard `java.lang` package.
- `Thread` instance is created by:
  ```java
  Thread myProcess = new Thread ();
  ```
- New thread started by executing:
  ```java
  MyProcess.start ();
  ```
  `start` method invokes a `run` method in the thread.
- As `run` method is undefined as yet, code above does nothing.
Monitors in Java: Recap on Threads (/3)

• We can define the `run` method by extending the `Thread` class:

```java
class myProcess extends Thread {
    public void run () {
        System.out.println("Hello from the thread");
    }
}
myProcess p = new myProcess();
p.start();
```

• Best to terminate threads by letting `run` method to terminate.

• If don’t need a ref to new thread omit `p` and simply write:

```java
new myProcess().start();
```
Monitors in Java: Recap on Threads (/4)

- As well as extending `Thread` class, can create lightweight processes by implementing `Runnable`.
- Advantage is can make your own class, or a system-defined one, into a process.
- Not possible with threads as Java only allows for one class at a time to be extended.
- Using the `Runnable` interface, previous example becomes:

```java
class myProcess implements Runnable {
    
    public void run () {
        System.out.println("Hello from the thread");
    }

    Runnable p = new myProcess();
    New Thread(p).start();
}
```
Monitors in Java: Recap on Threads (/5)

• If it has nothing immediate to do (e.g., it updates the screen regularly) should suspend thread by putting it to sleep.

• 2 flavours of `sleep()` method (specifying different times)

• `join()` awaits specified thread finishing, giving basic synchronisation with other threads.
  – i.e., "join" start of a thread's execution to end of another thread's execution
  – thus thread will not start until other thread is done.

• If `join()` is called on a Thread instance, the currently running thread will block until the Thread instance has finished executing:

```java
try {
    otherThread.join (1000); // wait for 1 sec
} catch (InterruptedException e) {}  
```
Monitors in Java: Synchronization

• Conceptually threads in Java execute concurrently, so could simultaneously access shared variables (aka A Race Condition)

• To prevent when updating a shared variable, Java provides synchronisation via a slimmed-down monitor.

• Java’s keyword synchronized provides mutual exclusion and can be used with a group of statements or with an entire method.

• The following class will potentially have problems if its update method is executed by several threads concurrently:

```java
class Problematic {
    private int data = 0;
    public void update() {
        data++;
    }
}
```
Monitors in Java: Synchronization (/2)

- Conceptually threads in Java execute concurrently and therefore could simultaneously access shared variables.

```java
class ExclusionByMethod {
    private int data = 0;
    public synchronized void update (){
        data++;
    }
}
```

- This is a simple monitor where the monitor’s permanent variables are private variables in the class;
- Monitor procedures are implemented as `synchronized` methods.
- Only 1 lock per object in Java thus if a `synchronized` method is invoked the following occurs:
  - it waits to obtain the lock,
  - executes the method, and then
  - releases the lock.
- This is known as `intrinsic locking`. 
Monitors in Java: Synchronization (/3)

• Can also have Mutual exclusion with **synchronized** statement in method’s body:

```java
class ExclusionByGroup {
    private int data = 0;
    public void update (){
        synchronized (this) {
            // lock this object for
            data++;
            // the following group of
            // statements
        }
    }
}
```

• The keyword **this** refers to object invoking the update method.
• The lock is obtained on the invoking object.
• A **synchronized** statement specifies that the following group of statements is executed as an atomic, non interruptible, action.
• A **synchronized** method is equivalent to a monitor procedure.
Monitors in Java: Condition Variables

- While Java does not explicitly support condition variables, there is one *implicitly* declared for each synchronised object.

- Java’s `wait()` & `notify()` resemble can only be executed in `synchronized` code parts (when object is locked):
  - `wait()` releases object lock, suspending the executing thread in a FIFO delay queue (one per object).
  - thus gets it to yield the monitor & sleep until some thread enters monitor & calls `notify()`
  - so `notify()` wakes thread at the front of object’s delay queue.
  - `notify()` has signal-and-continue semantics, so thread calling `notify()` still holds the object lock.

- Awakened thread goes later when it reacquires the object lock

- Java has `notifyAll()`, waking all threads blocked on same object.
Monitors in Java: Example 1: `Queue` Class

- `wait()` & `notify()` in Java are used in `Queue` implementation:

```java
/**
 * One thread calls push() to put an object on the queue. Another calls pop() to
 * get an object off the queue. If there is none, pop() waits until there is
 * using wait()/notify(). wait() and notify() must be used within a synchronized
 * method or block.
 */
import java.util.*;

public class Queue {
    LinkedList q = new LinkedList(); // Where objects are stored

    public synchronized void push(Object o) {
        q.add(o); // Append the object at end of the list
        this.notify(); // Tell waiting threads data is ready
    }

    public synchronized Object pop() {
        while(q.size() == 0) {
            try { this.wait(); }
            catch (InterruptedException e) { /* Ignore this exception */ }
        }
        return q.remove(0);
    }
}
```
class ReadersWriters
{
    private int data = 0;  // our database
    private int nr = 0;

    private synchronized void startRead()
    {
        nr++;
    }

    private synchronized void endRead()
    {
        nr--;            
        if (nr == 0) notify();  // wake a
        }                //waiting writer

    public void read()
    {
        startRead();
        System.out.println("read"+data);
        endRead();
    }

    public synchronized void write()
    {
        while (nr > 0)
        {
            try {
                wait();  //wait if any
                }      //active readers
        }
        catch (InterruptedException ex)
        {
            return;
        }
        data++;  
        System.out.println("write"+data);
        notify();  // wake a waiting writer
    }
}
Example 2: Readers/Writers Class (/2)

```java
class Reader extends Thread {
    int rounds;
    ReadersWriters RW;

    Reader(int rounds, ReadersWriters RW) {
        this.rounds = rounds;
        this.RW = RW;
    }

    public void run() {
        for (int i = 0; i < rounds; i++)
            RW.read();
    }
}

class Writer extends Thread {
    int rounds;
    ReadersWriters RW;

    Writer(int rounds, ReadersWriters RW) {
        this.rounds = rounds;
        this.RW = RW;
    }

    public void run() {
        for (int i = 0; i < rounds; i++)
            RW.write();
    }
}

class RWProblem {
    static ReadersWriters RW = new ReadersWriters();

    public static void main(String[] args) {
        int rounds = Integer.parseInt(args[0], 10);
        new Reader(rounds, RW).start();
        new Writer(rounds, RW).start();
    }
}
```

• This is the Reader Preference Solution. How to make this fair?
SECTION 4.2: DEVELOPMENTS IN JAVA.UTIL.CONCURRENT
Developments in `java.util.concurrent`

- Thus far, have focused on low-level APIs that were part of Java from the onset.
- These are ok for basic tasks, but need higher-level constructs for more advanced tasks
  - esp for many-thread parallel apps exploiting multi-core systems.
- In this lecture we focus on some high-level concurrency features of more recent Java releases.
- Most of these are implemented in `java.util.concurrent`
- Also have concurrent data structures in the Java `Collections` Framework.
Features in Brief

• **Semaphore** objects resemble those seen already; except `acquire()` & `release()` instead of `P`, `V` (resp)

• **Lock** objects support locking idioms that simplify many concurrent applications (don’t mix up with *implicit* locks!)

• **Executor**s give high-level API for launching, managing threads.

• **Executor** implementations provide thread pool management suitable for large-scale applications.

• Concurrent **Collections** support concurrent management of large data collections in Hash Tables, different kinds of Queues etc.

• **Future** objects are enhanced to have their status queried and return values when used in connection with asynchronous threads.

• Atomic variables (eg **AtomicInteger**) support atomic operations on single variables
  – features that minimize synchronization & help avoid memory consistency errors
  – i.e. useful in applications that call for atomically incremented counters
Semaphore Objects

• **Constructors for semaphore**

1. **Semaphore** object maintains a set of permits:

   ```java
   Semaphore(int permits);
   ```
   - Each acquire blocks till permit is available; Each release adds a permit
   - No permit objects *per se* – just keeps a count of available permits

2. **Semaphore** constructor also accepts a fairness parameter:

   ```java
   Semaphore(int permits, boolean fair);
   ```
   
   - permits: initial value
   - fair:
     - if true semaphore uses FIFO to manage blocked threads
     - if set false, class doesn’t guarantee order threads acquire permits.
     - In particular, lets barging (ie, thread doing `acquire()` can get a permit ahead of one waiting longer)
Example 3: Semaphore Example

```
//SemApp: code to demonstrate throttling with semaphore class © Ted Neward
import java.util.*; import java.util.concurrent.*;

public class SemApp {
    public static void main(String[] args) {
        Runnable limitedCall = new Runnable {
            final Random rand = new Random();
            final Semaphore available = new Semaphore(3);  // semaphore obj with 3 permits
            int count = 0;
            public void run() {
                int time = rand.nextInt(15);
                int num = count++;
                try {
                    available.acquire();
                    System.out.println("Executing " + "long-run action for " + time + " secs. #" + num);
                    Thread.sleep(time * 1000);
                    System.out.println("Done with # " + num);
                    available.release();
                }
                catch (InterruptedException intEx) {
                    intEx.printStackTrace();
                }
            }
        };
        for (int i=0; i<10; i++)
            new Thread(limitedCall).start();  // kick off worker threads
    } // end main
} // end SemApp
```

Lecture 4: Advanced Concurrency in Java
Example 3: Semaphore Example (/2)

- *Throttling* with Semaphore class

  Often must throttle number of open requests for a resource.
  
  – Can improve throughput of a system
    
    Does this by reducing contention for that particular resource.
  
  – Alternatively it might be a question of starvation prevention.
  
  – This was shown in the room case of Dining Philosophers (above)
    
    Only want to let 4 philosophers in the room at any one time
  
  – Can write the throttling code by hand, but it's often easier to use *semaphore* class - does it for you.
Example 3: **Semaphore Class (/2)**

- Even though the 10 threads in this code are running, only three are active.
- You can verify by executing `jstack` against the Java process running `SemApp`.
- The other seven are held at bay pending release of one of the semaphore counts.
- Actually, the **Semaphore** class supports acquiring and releasing more than one *permit* at a time,
- That wouldn't make sense in this scenario, however.
Interface Lock

- **Lock** implementations operate like the implicit locks used by `synchronized` code (only 1 thread can own a **Lock** object at a time\(^1\).)

- Unlike intrinsic locking all **lock** and **unlock** operations are explicit and have bound to them explicit **Condition** objects.

- Biggest advantage over implicit locks is can back out of an attempt to acquire a **Lock**:
  - i.e. livelock, starvation & deadlock are not a problem

- **Lock** methods:
  - `tryLock()` returns if lock is not available immediately or before a timeout (optional parameter) expires.
  - `lockInterruptibly()` returns if another thread sends an interrupt before the lock is acquired.

\(^1\) A thread can’t get a lock owned by another thread, but it can get a lock that it already owns. Letting a thread acquire the same lock more than once enables **Reentrant Synchronization** (i.e. tread with the lock on a synchronized code snippet can invoke another bit of synchronized code e.g. in a monitor.)
Interface Lock

- **Lock** interface also supports a *wait/notify* mechanism, through the associated **Condition** objects.

- Thus do away with basic monitor methods (*wait()*, *notify()* & *notifyAll()*) with specific objects:
  - **Lock** in place of *synchronized* methods and statements.
  - An associated **Condition** in place of Object’s monitor methods.
  - A **Condition** instance is intrinsically bound to a **Lock**.

- To obtain a **Condition** instance for a particular **Lock** instance use its **newCondition()** method.
Reentrantlocks & synchronized Methods

• Reentrantlock implements lock interface with the same mutual exclusion guarantees as synchronized.

• Acquiring/releasing a Reentrantlock has the same memory semantics as entering/exiting a synchronized block.

• So why use a Reentrantlock in the first place?
  – Using synchronized gives access to the implicit lock an object has, but forces all lock acquisition/release to occur in a block-structured way: if multiple locks are acquired they must be released in the opposite order.
  – Reentrantlock allows a more flexible locking/releasing mechanism.
  – Reentrantlock supports scalability and is nice where there is high contention among threads.

• So why not get rid of synchronized?
  – Firstly, a lot of legacy Java code uses it
  – Secondly, there are performance implications to using Reentrantlock
Example 4: Bounded Buffer

Using Lock & Condition Objects

```java
class BoundedBuffer {
    final Lock lock = new ReentrantLock();
    final Condition notFull = lock.newCondition();
    final Condition notEmpty = lock.newCondition();
    final Object[] items = new Object[100];
    int putptr, takeptr, count;

    public void put(Object x) throws InterruptedException {
        lock.lock(); // Acquire lock on object
        try {
            while (count == items.length)
                notFull.await();
            items[putptr] = x;
            if (++putptr == items.length)
                putptr = 0;
            ++count;
            notEmpty.signal();
        } finally {
            lock.unlock(); // release the lock
        }
    }

    public Object take() throws InterruptedException {
        lock.lock(); // Acquire lock on object
        try {
            while (count == 0)
                notEmpty.await();
            Object x = items[takeptr];
            if (++takeptr == items.length)
                takeptr = 0;
            --count;
            notFull.signal();
            return x;
        } finally {
            lock.unlock(); // release the lock
        }
    }
}
```
package net.jcip.examples;
import java.util.*;
import java.util.concurrent.*;
import java.util.concurrent.locks.*;
import static java.util.concurrent.TimeUnit.NANOSECONDS;
/**
 * DeadlockAvoidance: * Avoiding lock-ordering deadlock using tryLock *
 * @author Brian Goetz and Tim Peierls
 */
public class DeadlockAvoidance {
    private static Random rnd = new Random();
    public boolean transferMoney(Account fromAcct,
                                   Account toAcct,
                                   DollarAmount amount,
                                   long timeout,
                                   TimeUnit unit)
        throws InsufficientFundsException, InterruptedException {
            long fixedDelay = getFixedDelayComponentNanos(timeout, unit);
            long randMod = getRandomDelayModulusNanos(timeout, unit);
            long stopTime = System.nanoTime() + unit.toNanos(timeout);
            while (true) {
                if (fromAcct.lock.tryLock()) {
                    try {
                        if (toAcct.lock.tryLock()) {
                            try {
                                if (fromAcct.getBalance().compareTo(amount) < 0)
                                    throw new InsufficientFundsException();
                                else {
                                    fromAcct.debit(amount);
                                    toAcct.credit(amount);
                                    return true;
                                }
                            } finally {
                                toAcct.lock.unlock();
                            }
                        }
                    } finally {
                        fromAcct.lock.unlock();
                    }
                }
            }
        }
    }
}
Example 5: Bank Account Example using **Lock & Condition Objects** (/2)

```java
    } finally {
        fromAcct.lock.unlock();
    }

    if (System.nanoTime() < stopTime)
        return false;
    NanoSECONDS.sleep(fixedDelay + rnd.nextLong() % randMod);
}

private static final int DELAY_FIXED = 1;
private static final int DELAY_RANDOM = 2;

static long getFixedDelayComponentNanos(long timeout, TimeUnit unit;
    return DELAY_FIXED;
}

static long getRandomDelayModulusNanos(long timeout, TimeUnit unit)
    return DELAY_RANDOM;
}

static class DollarAmount implements Comparable<DollarAmount> {
    public int compareTo(DollarAmount other) {
        return 0;
    }
    DollarAmount(int dollars) {
    }
}

class Account {
    public Lock lock;

    void debit(DollarAmount d) {
    }

    void credit(DollarAmount d) {
    }

    DollarAmount getBalance() {
        return null;
    }
}

class InsufficientFundsException extends 
```
Example 5: Bank Account Example using Lock & Condition Objects (/3)

• With intrinsic locks deadlock can be serious, so `tryLock()` is used to allow control to be regained if all the locks cannot be acquired.

• `tryLock()` returns if lock is unavailable immediately or before a timeout expires (parameters specified).

• At `fromAcct.lock.tryLock` thread tries to get lock on `fromAcct`:
  – If successful, tries and acquire the lock on `toAcct`.
  – If former is successful but the latter is unsuccessful, can back off, release the one acquired and retry at a later time.
  – On acquiring both & with enough US$ in the `fromAcct`, call `debit()` on this object for the sum amount & `credit()` on `toAcct` is called for the same amount & boolean `TransferMoney()` comes back as true
  – If there are insufficient funds, an exception to that effect is returned.
Example 6: Dining Philosophers Using `Lock` Objects

```java
public class Fork {
    private final int id;
    public Fork(int id) {
        this.id = id;
    }
    // equals, hashCode, and toString() omitted
}

public interface ForkOrder {
    Fork[] getOrder(Fork left, Fork right);
} // We will need to establish an order of pickup

// Vanilla option with set pickup order implemented
class Philo implements Runnable {
    public final int id;
    private final Fork[] Forks;
    protected final ForkOrder order;

    public Philo(int id, Fork[] Forks, ForkOrder order) {
        this.id = id;
        this.Forks = Forks;
        this.order = order;
    }

    public void run() {
        while (true) {
            eat();
        }
    }

    protected void eat() {
        Fork[] ForkOrder = order.getOrder(getLeft(),
        getRight());
        synchronized(ForkOrder[0]) {
            synchronized(ForkOrder[1]) {
                Util.sleep(1000);
            }
        }
    }

    Fork getLeft() { return Forks[id]; }
    Fork getRight() { return Forks[(id+1) % Forks.length]; }
}
```

- This can, in principle, be run & philosophers just eat forever: choosing which fork to pick first; picking it up; then picking the other one up then eating etc.
- If you look at the code above in the `eat()` method, ‘grab the fork’ by synchronizing on it, locking the fork's monitor.
Example 6: Dining Philosophers Using Lock Objects (/2)

```java
class Philo implements Runnable {
    public final int id;
    private final Fork[] Forks;
    protected final ForkOrder order;

    public Philo(int id, Fork[] Forks, ForkOrder order) {
        this.id = id;
        this.Forks = Forks;
        this.order = order;
    }

    public class GraciousPhilo extends Philo {
        private static Map ForkLocks = new ConcurrentHashMap();

        public GraciousPhilo(int id, Fork[] Forks, ForkOrder order) {
            super(id, Forks, order);
            // Every Philo creates a lock for their left Fork
            ForkLocks.put(getLeft(), new ReentrantLock());
        }

        protected void eat() {
            Fork[] ForkOrder = order.getOrder(getLeft(),
                getRight());
            Lock firstLock = ForkLocks.get(ForkOrder[0]);
            Lock secondLock = ForkLocks.get(ForkOrder[1]);
            firstLock.lock();

            try {
                secondLock.lock();
                try {
                    Util.sleep(1000);
                } finally {
                    secondLock.unlock();
                }
            } finally {
                firstLock.unlock();
            }
        }
    }
}
```

- Just replace `synchronized` with `lock()` & end of `synchronized` block with a `try` { } finally { } unlock() }.
- This allows for timed wait (until finally successful) or
- `lockInterruptibly()` - block if lock already held, wait until lock is acquired; if another thread interrupts waiting thread `lockInterruptibly()` - will throw `InterruptedException`
Dining Philosophers Using **ReentrantLocks** (/3)

- Can leverage additional power of **ReentrantLock** to do some niceties:
  - First, don't have to block forever on the `lock` call.
  - Instead we can do a timed wait using `tryLock()`.
  - One form of this method returns immediately if the lock is already held.
  - Other can wait for some time for the lock to become available before giving up.
  - In both, could effectively loop and retry the `tryLock()` until it succeeds.

- Another nice option is to **lockInterruptibly()**
  - Calling this allows for waiting indefinitely but reply to thread being interrupted.
  - Possible to write an external monitor that either watches for deadlock or allows a user to forcibly interrupt one of the working threads.
  - Could be provided via JMX to allow a user to recover from a deadlock.
Pre-History of Executors

- As seen above, one method of creating a multithreaded application is to implement `Runnable`.
- In J2SE 5.0, this became the `preferred` means (using package `java.lang`).
- Built-in methods and classes are used to create Threads that execute the Runnables.
- As also seen, the `Runnable` interface declares a single method named `run`.
- Runnables are executed by an object of a class that implements the `Executor` interface.
- This can be found in package `java.util.concurrent`. 
Executors (new)

• Seen how to create multiple threads and coordinate them via synchronized methods and blocks, as well as via Lock objects.

• But how do we execute the threads to different cores on a multicore machine?

• There are 2 mechanisms in Java
  – Executor Interface and Thread Pools
  – Fork/Join Framework
Executors: **Executor** Interface & Thread Pools

- *java.util.concurrent* package provides 3 executor interfaces:
  - **Executor**: Simple interface that launches new tasks.
  - **ExecutorService**: Subinterface of **Executor** that adds features that help manage tasks’ lifecycle.
  - **ScheduledExecutorService**: Subinterface of **ExecutorService** supporting future and/or periodic execution of tasks.

- The **Executor** interface provides a single method, **execute**.

- For runnable object *r*, Executor object *e* then
  ```java
  e.execute (r);  
  ```
  may simply execute a thread,
  or it may use an existing worker thread to run *r*,
  or, with thread pools, queue *r* to wait for available worker thread.
Thread pool threads execute **Runnable** objects passed to `execute()`.

The **Executor** assigns each **Runnable** to an available thread in the thread pool.

If none available, it creates one or waits for one to become available & assigns that thread the **Runnable** passed to method `execute`.

Depending on the **Executor** type, there may be a limit to the number of threads that can be created.

A subinterface of Executor (Interface **ExecutorService**) declares other methods to manage both **Executor** and task/ thread life cycle.

An object implementing the **ExecutorService** sub-interface can be created using static methods declared in class **Executors**.
Example 7: Executors

//From Deitel & Deitel PrintTask class sleeps a random time 0 - 5 seconds
import java.util.Random;

class PrintTask implements Runnable {
    private int sleepTime; // random sleep time for thread
    private String threadName; // name of thread
    private static Random generator = new Random();
    // assign name to thread
    public PrintTask(String name)
        threadName = name; // set name of thread
        sleepTime = generator.nextInt(5000); // random sleep 0-5 secs
    } // end PrintTask constructor

    // method run is the code to be executed by new thread
    public void run()
        try // put thread to sleep for sleepTime {
            System.out.printf("%s sleeps for %d ms.\n", threadName, sleepTime);
            Thread.sleep( sleepTime ); // put thread to sleep
        } // end try
        // if thread interrupted while sleeping, print stack trace
        catch ( InterruptedException exception ) {
            exception.printStackTrace();
        } // end catch
        // print thread name
        System.out.printf( "%s done sleeping\n", threadName );
    } // end method run
} // end class PrintTask
Example 7: Executors (/2)

- When a `PrintTask` is assigned to a processor for the first time, its `run` method begins execution.
- Static method `sleep` of class `Thread` is called to place the thread into the timed waiting state.
- At this point, thread loses the processor & system lets another execute.
- When the thread awakens, it re-enters the runnable state.
- When the `PrintTask` is assigned to a processor again, thread’s name is output saying thread is done sleeping; `run` terminates.
Example 7: Executors Main Code

//RunnableTester: Multiple threads printing at different intervals
import java.util.concurrent.Executors;
import java.util.concurrent.ExecutorService;

public class RunnableTester {
    public static void main(String[] args) {
        // create and name each runnable
        PrintTask task1 = new PrintTask("thread1");
        PrintTask task2 = new PrintTask("thread2");
        PrintTask task3 = new PrintTask("thread3");

        System.out.println("Starting threads");

        // create ExecutorService to manage threads
        ExecutorService threadExecutor
            = Executors.newFixedThreadPool(3);

        // start threads and place in runnable state
        threadExecutor.execute(task1); // start task1
        threadExecutor.execute(task2); // start task2
        threadExecutor.execute(task3); // start task3

        threadExecutor.shutdown(); // shutdown worker threads

        System.out.println("Threads started, main ends\n");
    } // end main
} // end RunnableTester
Example 7: Executors Main Code (/2)

- The code above creates three threads of execution using the `PrintTask` class.
- `main`
  - creates & names three `PrintTask` objects.
  - creates a new `ExecutorService` using method `newFixedThreadPool()` of class `Executors`, which creates a pool consisting of a fixed number (3) of threads.
  - These threads are used by `threadExecutor` to run the execute method of the `Runnables`.
  - If `execute()` is called and all threads in `ExecutorService` are in use, the `Runnable` will be placed in a queue
  - It is then assigned to the first thread completing its previous task.
Example 7: Executors Main Sample Output

Starting threads
Threads started, main ends

thread1 sleeps for 1217 ms.
thread2 sleeps for 3989 ms.
thread3 sleeps for 662 ms.
thread3 done sleeping
thread1 done sleeping
thread2 done sleeping
Executors: Futures/Callables

- Pre-Java 8 version of Futures was quite weak, only supporting waiting for Future to complete.
- Also executor framework uses Runnables & Runnable can’t return a result.
- A Callable object allows return values after completion.
- Callable uses generics to define type of object returned.
- If you submit a Callable object to an Executor, framework returns java.util.concurrent.Future object.
- This Future object can be used to check the status of a Callable and to retrieve the result from the Callable.
So writing asynchronous concurrent programs that return results using executor framework requires:

- Define class/task implementing either `Runnable` or `Callable` interface
- Configure & implement `ExecutorService`
- (This because need `ExecutorService` to run the `Callable` object.)
- The service accepts `Callables` to run using `submit()` method
- Submit task using `Future` class to retrieve result if task is `Callable`

Difference between a `Runnable` and `Callable`:

- `Runnable` interfaces do not return a result while `Callable` permits returning values after completion.
- When a `Callable` is submitted to the executor framework, it returns an object of type `java.util.concurrent.Future`.
- The `Future` can be used to retrieve results
Executors: Futures/Callables (/3)
Example 8

```java
package de.vogella.concurrency.callables;
import java.util.concurrent.Callable;
public class MyCallable implements Callable<Long> {
    @Override
    public Long call() throws Exception {
        long sum = 0;
        for (long i = 0; i <= 100; i++) {
            sum += i;
        }
        return sum;
    }
}
```

1 This code and associated piece on the next page were written and are Copyright © Lars Vogel. Source Code can be found at `de.vogella.concurrency.callables`.
package de.vogella.concurrency.callables;
import java.util.ArrayList;
import java.util.List; import java.util.concurrent.Callable;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors; import java.util.concurrent.Future;

public class CallableFutures {
    private static final int NTHREADS = 10;
    public static void main(String[] args) {
        ExecutorService executor = Executors.newFixedThreadPool(NTHREADS);
        List<Future<Long>> list = new ArrayList<Future<Long>>();
        for (int i = 0; i < 20000; i++) {
            Callable<Long> worker = new MyCallable();
            Future<Long> submit = executor.submit(worker);
            list.add(submit);
        }
        long sum = 0;
        System.out.println(list.size());
        // now retrieve the result
        for (Future<Long> future : list) {
            try {
                sum += future.get(); // get() method of Future will block until task is completed
            } catch (InterruptedException e) {
                e.printStackTrace();
            } catch (ExecutionException e) {
                e.printStackTrace();
            }
        }
        System.out.println(sum); executor.shutdown();
    }
}

1 Executors: Futures/Callables (/4)

Example 8
ForkJoin Framework

- Since Java 7, the Fork/Join framework can be used to distribute threads among multiple cores.
- It’s an implementation of `ExecutorService` interface designed for work that can be broken into smaller pieces recursively.
- Goal: use all available processors to enhance application performance
- This framework thus adopts a divide-and-conquer approach:
  - If task can be easily solved
    - current thread returns its result.
  - Otherwise ->
    - thread divides the task into simpler tasks and forks a thread for each sub-task.
    - When all sub-tasks are done, the current thread returns its result obtained from combining the results of its sub-tasks.
- Key difference between Fork/Join framework and Executor Interface is the former implements a work stealing algorithm.
  - This allows idle threads to steal work from busy threads (i.e. pre-empting).
ForkJoin Framework (/2)

• A key class is the ForkJoinPool (an implementation of ExecutorService implementing work-stealing.)

• A ForkJoinPool is instantiated thus:
  ```java
  numberOfCores = Runtime.getRuntime().availableProcessors();
  ForkJoinPool pool = new ForkJoinPool(numberOfCores);
  ```

• Pool size is changed automatically at any time giving enough active threads.

• Unlike ExecutorService, ForkJoinPool needn’t be explicitly shutdown.

• There are 3 ways to submit tasks to a ForkJoinPool
  – `execute()` : asynchronous execution
  – `invoke()` : synchronous execution - wait for the result
  – `invoke()` : asynchronous execution - returns a Future object that can be used to check the status of the execution and obtain the results.
ForkJoin Framework (/3)

- Thus, `ForkJoinPool` facilitates tasks to split work up into smaller tasks.
- These smaller tasks are then submitted to the `ForkJoinPool` too.
- This aspect differentiates `ForkJoinPool` from `ExecutorService`.
- Task only splits itself up into subtasks if work it was given is large enough for this to make sense.
- Reason for this is the overhead to splitting up a task into subtasks.
- So for small tasks this may be greater than speedup from executing subtasks concurrently.
ForkJoin Framework (/4)

• Submitting tasks to a ForkJoinPool is like submitting tasks to an ExecutorService.

• Can submit two types of tasks.
  – A task that does not return any result (aka an "action"), and
  – One which does return a result (a "task").

• These two types of tasks are represented by RecursiveAction and RecursiveTask classes, respectively.

• To use a ForkJoinPool to return a result:
  1. first create a subclass of RecursiveTask<V> for some type V
  2. In the subclass, override the compute() method.
  3. Then you call the invoke() method on the ForkJoinPool passing an object of type RecursiveTask<V>

• The use of tasks and how to submit them is summarised in the following example.
Example 9: Returning a Result from a ForkJoinPool

```java
import java.util.concurrent.ForkJoinPool;
import java.util.concurrent.RecursiveTask;

class Globals {
    static ForkJoinPool fjPool = new ForkJoinPool();
}

// This is how you return a result from fjPool
class Sum extends RecursiveTask<Long> {
    static final int SEQ_LIMIT = 5000;

    int low;
    int high;
    int[] array;

    Sum(int[] arr, int lo, int hi) {
        array = arr;
        low = lo;
        high = hi;
    }

    protected Long compute() {
        // override the compute() method
        if(high - low <= SEQ_LIMIT) {
            long sum = 0;
            for(int i=low; i < high; ++i)
                sum += array[i];
            return sum;
        } else {
            int mid = low + (high - low) / 2;
            Sum left = new Sum(array, low, mid);
            Sum right = new Sum(array, mid, high);
            left.fork();
            long rightAns = right.compute();
            long leftAns = left.join();
            return leftAns + rightAns;
        }
    }
}

static long sumArray(int[] array) {
    return Globals.fjPool.invoke(new Sum(array, 0, array.length));
}
```

- This example sums all the elements of an array, using parallelism to potentially process different 5000-element segments in parallel.
Example 9: Returning a Result from a `ForkJoinPool(/2)`

- **Sum** object gets an array & its range; **compute** sums elements in range.
  - If range has < **SEQ_LIMIT** elements, use a simple for-loop
  - Else, create two **Sum** objects for problems of half the size.
- Uses **fork** to compute left half in parallel to computing the right half, which this object does itself by calling **right.compute()**.
- To get the answer for the left, it calls **left.join()**.
- Create more **Sum** objects than available processors as it’s framework's job to do a number of parallel tasks efficiently
- But also to schedule them well - having lots of fairly small parallel tasks can do a better job.
- Especially true if number of processors available varies during execution (e.g., due to OS is also running other programs)
- Or maybe, despite load balancing, tasks end up taking different time.
Concurrent Annotations

• Annotations were added as part of Java 5.
• Java comes with some predefined annotations (e.g. @override), but custom annotations are also possible (e.g. @GuardedBy).
• Many frameworks and libraries make good use of custom annotations. JAX-RS, for instance, uses them to turn POJOs into WS resources.
• Annotations are processed at compile time or at runtime (or both).
• Good programming practice to use annotations to document code.
• Here is an Example:

```java
public class BankAccount {
    private Object credential = new Object();
    @GuardedBy("credential") // amount guarded by credential because
    private int amount; // access only if synch lock on credential held
}
```

• Will revisit annotations again later with Web Services.
Lecture Summary

• Concurrency support in Java has developed greatly since early versions:
  – Native **semaphore** class
  – Extra functionality in explicit **Lock/Condition** objects

• Perhaps in terms of large-scale thread have there been greatest strides since the **Runnable** interface

• **Interface Executor** provides many different support mechanisms for threads:
  – For allocation of threads to different cores on a multicore machine
  – For returning future results from an asynchronous task
  – For pre-empting/work-stealing using **ForkJoin**
  – Annotations have many applications. E.g., JAX-WS uses annotated POJOs for generating WS resources