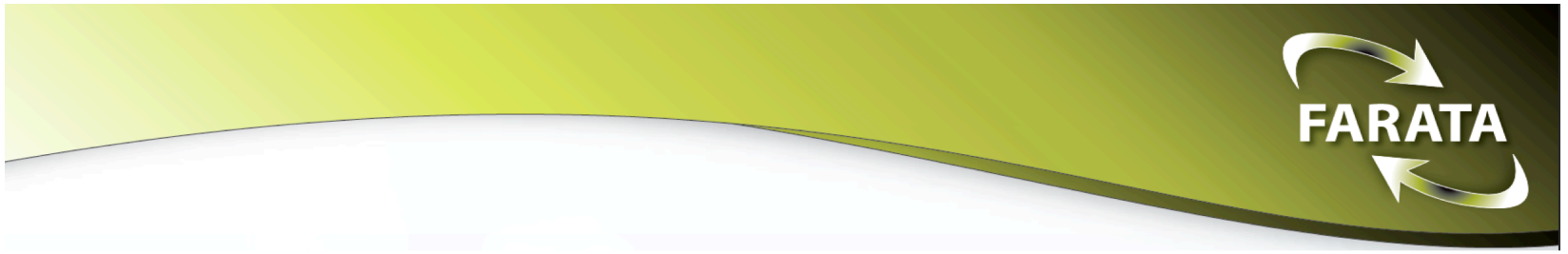

Java Programming
Unit 11

Intro to Concurrency.
Multithreading.

Intro to Multi-Threading

- A program may need to execute some tasks **concurrently**, e.g. get market news and the user's portfolio data.
- *Concurrent* means parallel execution
- A Java program is a process
- A *thread* is a light-weight process
- One Java program can start (*spawn*) multiple threads

Intro to Multi-Threading (cont.)

- One server instance can process multiple clients' request by spawning multiple threads of execution (one per client).

- My notebook has 4 CPUs. Things can run in parallel.

- Even on a single-CPU machine you can benefit from the multi-threading – one thread needs CPU, the other waits for the user's input, the third one works with files.

The class Thread

```
public class MarketNews extends Thread {
 public MarketNews (String threadName) {
    super(threadName);  // name your thread
 }

  public void run() {
      System.out.println(
       "The stock market is improving!");
  }
}
```

```
public class Portfolio extends Thread {
 public Portfolio (String threadName) {
        super(threadName);
    }

 public void run() {
       System.out.println(
       "You have 500 shares of IBM ");
 }
}
```

```
public class TestThreads {
    public static void main(String args[]){
      MarketNews mn = new MarketNews("Market News");
      mn.start();

      Portfolio p = new Portfolio("Portfolio data");
      p.start();
      System.out.println( "TestThreads is finished");
    }
}
```

Interface Runnable

```
public class MarketNews2
                    implements Runnable {
     public void run() {
       System.out.println(
       "The stock market is improving!");
     }
}
```

```
public class Portfolio2
                            implements
Runnable {
    public void run() {
       System.out.println(
        "You have 500 shares of IBM");
     }
}
```

```
public class TestThreads2 {
    public static void main(String args[]){

     MarketNews2 mn2 = new MarketNews2();
     Thread mn = new Thread(mn2,"Market News");
     mn.start();

     Runnable port2 = new Portfolio2();
     Thread p = new Thread(port2, "Portfolio Data");
     p.start();

     System.out.println( "TestThreads2 is finished");
    }
}
```

Sleeping Threads

```
public class MarketNews3 extends Thread {
  public MarketNews3 (String str) {
   super(str);
  }

 public void run() {
   try{
    for (int i=0; i<10;i++){
     sleep (1000);  // sleep for 1 second
     System.out.println( "The market is improving " + i);
    }
   }catch(InterruptedException e ){
     System.out.println(Thread.currentThread().getName()
                              + e.toString());
   }
 }
}
```

```
public class Portfolio3 extends Thread {
  public Portfolio3 (String str) {
      super(str);
  }

  public void run() {
    try{
     for (int i=0; i<10;i++){
      sleep (700);    // Sleep for 700 milliseconds
      System.out.println( "You have " +  (500 + i) +
                              " shares of IBM");
     }
    }catch(InterruptedException e ){
     System.out.println(Thread.currentThread().getName()
                              + e.toString());
   }
  }
```

```
public class TestThreads3 {

   public static void main(String args[]){

       MarketNews3 mn = new MarketNews3("Market News");
       mn.start();

       Portfolio3 p = new Portfolio3("Portfolio data");
       p.start();

       System.out.println( "The main method of TestThreads3 is finished");
   }
}
```

Walkthrough 1

- Download and import the project for the lesson 20 of the textbook
- Review the code of the threads created by subclassing `Thread` and implementing `Runnable`.
- Run the programs `TestThreads` and `TestThreads2`. Observe the output.
- Review the code that uses sleeping threads.
- Run the program TestThereads3 several times. Observe the output – is it always the same?
- Change the sleeping time in `Portfolio3` and `MarketNews3`. Re-run `TestThreads3` – the output should be different now.

Race Conditions

- A *race condition*: multiple threads try to modify the same program resource at the same time (*concurrently*).

- Husband and wife are trying to withdraw cash from different ATMs at the same time.

Thread Synchronization: the Old Way

- **The old** way to prevent race conditions is by using the keyword `synchronized`.

- The **better** way is using the class `ReentrantLock`, which offers better performance in high-concurrency apps.
  It also offers different options for locking.

- Both the `synchronized` and `ReentrantLock` place a lock (*a monitor*) on an important object or a piece of code to allow only one thread at a time access this code.

Minimize the locking periods

```
class ATMProcessor extends Thread{
  …
 synchronized withdrawCash(int accountID, int amount){
   // Some thread-safe code goes here, i.e. reading from
   // a file or a database
     …
  boolean allowTransaction = validateWithdrawal(accountID,
                                    amount);
  if (allowTransaction){
     updateBalance(accountID, amount, “Withraw”);
  }
  else {
     System.out.println(“Not enough money on the account”);
  }
 }
}
```

Synchronizing the entire method

Synchronizing the code block

```
class ATMProcessor extends Thread{
  …
 withdrawCash(int accountID, int amount){
   // Some thread-safe code goes here, i.e. reading from
   // a file or a database
   …
 synchronized(this) {
 boolean allowTransaction =
         validateWithdrawal(accountID, amount);
  if (allowTransaction){
   updateBalance(accountID, amount, “Withdraw”);
  }
  else {
   System.out.println(
                    “Not  enough money on the account”);
  }
 }
 }
}
```

Thread Synchronization: the Better Way

The class `ReentrantLock` is a better performing solution than the `synchronized`.

Place a lock before the section of your program that may result in a race condition, and to remove the lock afterward.

```
private Lock accountLock = new ReentrantLock();

witdrawCash(int accountID, int amount){
    // Some thread-safe code goes here, i.e. reading from
    // a file or a database
   …

   accountLock.lock(); // place a lock when this thread enters the code

   try{
   if (allowTransaction){
    updateBalance(accountID, amount, "Withdraw");
   }
   else {
    System.out.println("Not  enough money on the account");
   }
  }finally {
     accountLock.unlock(); // allow other threads to update balance
  }
 }
```

How to Kill a Thread

- The method `Thread.stop()` is deprecated. Don't use it as it may leave your program resources locked.

- You can declare in your thread a flag variable, say `stopMe`, and test it periodically in a loop condition – exit the loop if this flag is set.

- You can call the method `interrupt()` on the `Thread` instance to stop it.

Killing a thread by raising a flag

```
class KillTheThread{

  public static void main(String args[]){
        Portfolio4 p = new Portfolio4("Portfolio data");

        p.start();

        // Some other code goes here
        //   …
        // and now it's time to kill the thread

        p.stopMe();

       // This method won't work if a thread is not
       //  doing any processing, e.g. waits for the user's
       //  input. In such cases use p.interrupt();

  }
}
```

```
class Portfolio4 extends Thread{

private volatile boolean stopMe = false;

    public void stopMe() {
       stopMe = true;
    }

    public void run() {
       while (!stopMe) {
         try{
           //Do some portfolio processing here
         }catch(InterruptedException e ){
           System.out.println(Thread.currentThread().getName()
                                  + e.toString());
       }

     }
 }
```

The variable `stopMe` is declared with as `volatile`, which warns the Java compiler that another thread can modify it, and that this variable **should not be cached in registers**, so that all threads must always see its "fresh" value.

Wait and Notify

The class `Object` also has methods relevant to threads: `wait()`, `notify()`, and `notifyAll()`.

For example, `wait(1000)` means that the current thread has to wait for a notification from another thread for **up to** 1 second (a thousand milliseconds).

A thread can notify other thread(s) using methods `notify()` or `notifyAll()`.

```
synchronized (this) {
  try{
    wait(1000);
  } catch (InterruptedException e){…}
}
```

The Condition interface offers an alternative to wait(), notify(), notifyAll(). See http://goo.gl/Wx8Vd

Wait-Notify Sample

```
class ClassA  {
 String marketNews = null;

 void  someMethod(){

  // The ClassB needs a reference to the locked object
  // to be able to notify it

  ClassB myB=new ClassB(this);
  myB.start();
  synchronized(this) {
   wait();
  }

  // Some further processing of the MarketData
  // goes here
 }

 public void setData (String news){
   marketNews = news;
 }
}
```

```
class ClassB extends Thread{

  ClassA parent = null;

  ClassB(ClassA caller){
   parent = caller; // store the reference to the caller
  }

  public void run(){
   // Get some data, and, when done, notify the parent
   parent.setData("Economy is recovering…");
   …
   synchronized (parent){
    parent.notify();  //notification of the caller
   }
  }
}
```

Walkthrough 2 (start)

1. In Eclipse, Copy/Paste the class `TestThreads3` from Lesson20 Eclipse project, and rename it into the class `TestThreadsWait`

2. Add the following code above the `println()` line:

```
Object theLockKeeper = new Object();

  synchronized (theLockKeeper) {
       try{
           System.out.println("Starting to wait…");
           theLockKeeper.wait(15000);
       } catch (InterruptedException e){
          System.out.println("The main thread is interrupted");
       }
    }
```

3. Run the program. It won't print the "finished " message sooner than 15 sec. What has to be done to make this program to stop waiting earlier?

Walkthrough 2 (end)

1. Let's replace `synchronized/wait()` with `Condition/await()`.

2. Comment out the synchronized block and theLockKeeper line. Add the following code above the `println()` line:

```
ReentrantLock theLock = new ReentrantLock();
        final Condition notDone  = theLock.newCondition();

        theLock.lock();
        try{
           System.out.println(" Starting to wait...");
          notDone.await(15, TimeUnit.SECONDS);
        } catch (InterruptedException e){
          System.out.println("The main thread is interrupted");
        }

        theLock.unlock();
```

3. Run the program. It'll wait for 15 seconds before printing the "finished " message.

Joining Threads

If one thread needs to wait for the completion of another, you can use the method `join()`.

```
public class TestThreadJoin {

    public static void main(String args[]){

         MarketNews3 mn = new MarketNews3("Market News");
         mn.start();

         Portfolio3 p = new Portfolio3("Portfolio data");
         p.start();

         try{
          mn.join();
          p.join();

         }catch (InterruptedException e){
          e.printStackTrace();
         }

        System.out.println( "The main method of TestThreadJoin is finished");

    }
}
```

Walkthrough 3

1. Import the code samples from Lesson 21 of the textbook.

2. Review the code of the program `TestThreadsJoin` and run it.

3. Observe the output – the main thread waits for the completion of the threads `MarketNews3` and `Portfolio3`.

Executor Framework

Creating threads by subclassing `Thread` or implementing `Runnable` has drawbacks:

1. The thread's method `run()` cannot return a value.

2. An application may spawn too many threads - it can take up all system resources.

To overcome the first drawback use the `Callable` interface, and the second one by using classes from the *Executor framework*.

The `Executors` class spawns the threads from Runnable objects.

`ExecutorService` knows how to create `Callable` threads.

`ScheduledExecutorService` allows to schedule threads for future execution.

The class Executors

The class `Executors` has static methods to create an appropriate executor.

Its method `newFixedThreadPool()` creates a pool of threads of a specified size, e.g. you can create a pool that allows not more than 5 threads:

```
Executors.newFixedThreadPool(5);
```

Fork Join and Streams API.

Java 7 introduced fork/join framework, which is an implementation of `ExecutorService` for parallel work on multiple processors.
For more info read Oracle tutorial on Fork/Join: http://bit.ly/1lzn2rF

If you're using Java 8, consider Stream API for parallelism as it doesn't require to manually perform partitioning and combining.
For more info read Oracle Tutorial on Parallelism: http://bit.ly/1kyElaK

Callable Interface

If you need to return the data from a thread use `Callable` defined as:

```
public interface Callable <V>{
    V call() throws Exception;
}
```

The class `Executors` has a number of static methods that will create a thread from your `Callable` class and return the result as an object implementing the interface `Future`.

```
public class Portfolio implements Callable<Integer>{

  public Integer call() {
    // Perform some actions
    return someInteger;
  }
}


public class MarketData implements Callable<Integer>{

  public Integer call() {
    // Perform some actions
    return someInteger;

  }
}
```

The Future Object

To create `Future` object, submit an instance of the `Callable` thread to the `Executor`. Call the function `get()` on the `Future` instance, and **it'll block** on the thread until its `call()` method returns the result(s):

```
List<Future<Integer>> threadResults = new ArrayList<Future<>>();
```

```
// Submit two Callables for execution
threadResults.add(myExecutorService.submit(new Portfolio()));
threadResults.add(myExecutorService.submit(new MarketData()));


//Wait for threads to complete
for (Future<Integer> future : threadResults) {

      future.get();  // an equivalent of joining threads
}
```

JDK 8 introduced `CompleatableFuture`.

How to Spawn a Callable Thread

1. Declare and instantiate a class that implements the `Callable` interface, and program the business logic in its method `call()`.

2. Create an instance of the `Future` object.

3. Create an instance of an `ExecutorService` using `Executors.newFixedThreadPool()`.

4. Call the function `submit()` on the `ExecutorService`, providing an instance of the `Callable` object as an argument.

5. Call the function `get()` on the `Future` object from Step 2.
   This function will wait until the thread returns the result (or throws an exception).

6. Accept the result of the thread execution into a variable of the data type used in Step 1.

7. Call the function `shutdown()` on the `ExecutorService` from Step 3.

Callable Portfolio and MarketNews

```
class PortfolioCallable implements Callable<Integer> {

 public Integer call() throws Exception {
  for (int i=0; i<5;i++){
        Thread.sleep (700);    // Sleep for 700 milliseconds

     System.out.println( "You have " +  (500 + i) +
                          " shares of IBM");
  }

  // Just return some number as a result
  return 10;
 }
}
```

```
class MarketNewsCallable implements Callable<Integer> {

 public Integer call() throws Exception {

  for (int i=0; i<5;i++){
     Thread.sleep (1000);  // sleep for 1 second
     System.out.println( "The market is improving " + i);
   }

  // Just return some number as a result
  return 12345;
 }
}
```

Testing Callable Portfolio and MarketNews

```
public class TestCallableThreads {

  public static void main(String[] args)
   throws InterruptedException, ExecutionException {

   //A placeholder for Future objects
     List<Future<Integer>> futures =    new ArrayList<Future<Integer>>();

    // A placeholder for results
     List<Integer> results = new ArrayList<Integer>();

   final ExecutorService service =
     Executors.newFixedThreadPool(2); // pool of 2 threads

   try {
    futures.add(service.submit(new PortfolioCallable()));
    futures.add(service.submit(new MarketNewsCallable()));

    for (Future<Integer> future : futures) {
     results.add(future.get());
    }

   } finally {
    service.shutdown();
   }

   for (Integer res: results){
      System.out.println("\nGot the result: " + res);
   }
  }
 }
```

Walkthrough 4

- Review the code of the `TestCallableThreads` form the Lesson21 project. Run the examples and observe the results.

- Change the code of the thread `PortfolioCallable` as follows:

  a) Its constructor should have an argument to receive the price of the IBM stock: double price

  b) The thread should return the total amount based on the price multiplied by 504.

  c) Change the code of the `TestCallableThreads` to pass 164.22 as the price of IBM

  d) Run and the `TestCallableThreads` and observe the results

Homework

1. Study the materials from the lessons 20 and 21 from the textbook and do the assignments from theirs Try It sections.

2. Study the tutorial by Lars Vogel on Java Concurrency and Multithreading:
http://www.vogella.de/articles/JavaConcurrency/article.html

Additional reading

Read Oracle's tutorial about Executors:
http://docs.oracle.com/javase/tutorial/essential/concurrency/executors.html

Study the presentation and project of Viktor Grazi, "Concurrency Animated":
http://sourceforge.net/projects/javaconcurrenta/ and jconcurrency.com.

Read about more flexible locking in Java 5:
http://www.ibm.com/developerworks/java/library/j-jtp10264/

Learn how to use `BlockingQueue` instead of wait/notify:
http://tutorials.jenkov.com/java-util-concurrent/blockingqueue.html

Additional reading (cont.)

Read the article “Java 8. Definite Guide to CompletableFuture”: http://www.nurkiewicz.com/2013/05/java-8-definitive-guide-to.html

Watch the presentation “Modern Java Concurrency”: http://bit.ly/HIJsH8

From Java Code to Java Heap (in depth coverage of Java memory usage): http://goo.gl/0dIZK