Lecture 11:Best Programming Practices with Java 8

Living Life in Accord with Natural Law

Wholeness Statement

Best practices in the world of OO programming are a way of ensuring quality in code. Code that adheres to best practices tends to be easier to understand, easier to maintain, more capable of adapting to change in the face of changing requirements and new feature requests, and more reusable for other projects. This simple theme is reflected in individual life. There are laws governing life, both physical laws and laws pertaining to all kinds of relationships and interactions. When life flows in accordance with the laws of nature, life is supported for success and fulfillment. When awareness becomes established at its deepest level, actions and behavior springing from this profound quality of awareness spontaneously are in accord with the laws of nature.

Overview

1. Create your first JUnit test
2. Unit-testing Stream Pipelines
3. Handling Exceptions Arising in Stream Pipelines
4. Concurrent Processing and Parallel Streams

Overview

1. Create your first JUnit test
2. Unit-testing Stream Pipelines
3. Handling Exceptions Arising in Stream Pipelines
4. Concurrent Processing and Parallel Streams

JUnit

- A quick way to test your code as you develop is to run it from the main method as we have been doing.
- A better way that is more reusable and useful for larger-scale team development is to have parallel tests that use JUnit.
- JUnit is a unit testing framework for the Java programming language. JUnit becomes more flexible with release of version 4.0.

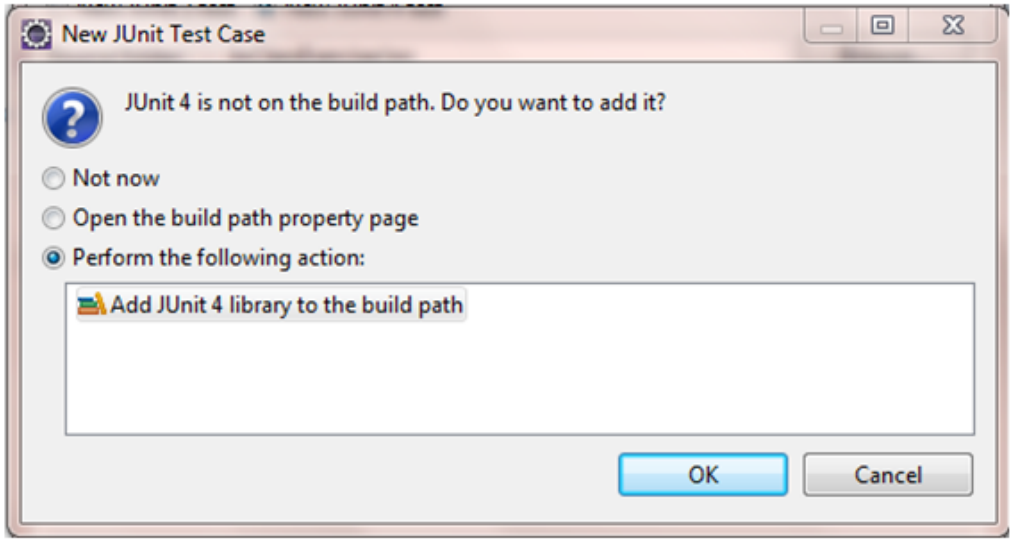
Setting Up JUnit 4.0

- For this demo, add the following method to your Hello.java above the main method.

```
public static String hello(){
    return "Hello";
}
```

Steps to set up JUnit

- Right click the default package where your Hello.java class is, and create a JUnit Test Case TestHello.java by clicking New→JUnit Test Case. (After we introduce the package concept, we will put the tests in a separate package.)
- Name it TestHello and click finish. You will see the following popup window.



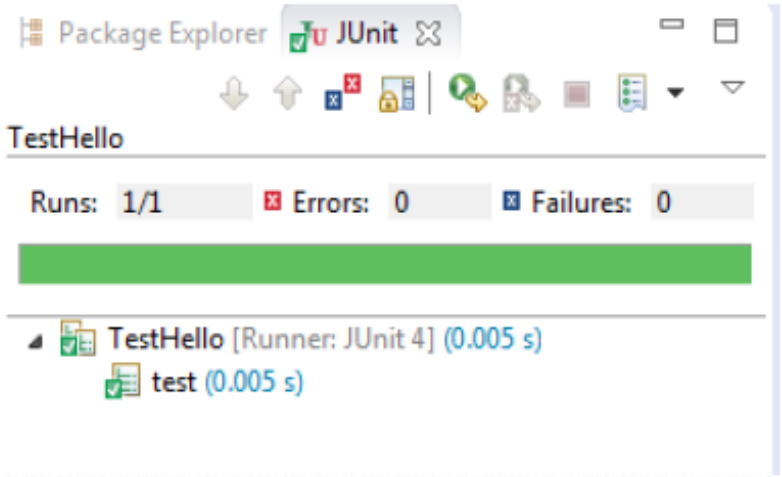
- Select "Perform the following action" → Add JUnit 4 library to the build path → OK
- Then you will see that JUnit 4 has been added to the Java Build Path by right clicking your project name → properties → Java Build Path.

- Create a testHello method in TestHello.java (see below) and remove the method test() that has been provided by default:

```
@Test
public void testHello() {
    assertEquals("Hello", Hello.hello());
}
```

If you have compiler error, you can hover over to the workspace where you have error and Eclipse will give you hints of what to do. (In this case, Add import 'org.junit.Assert.assertEquals'.)

- Run your JUnit Test Case by right clicking the empty space of the file → Run As → JUnit Test. If you see the result like below, it means you have successfully created your first unit test. ☺



Setting Up JUnit 4.0

- Methods that are annotated with @Test can be unit-tested with JUnit.
- Three most commonly used methods:

  assertTrue(String comment, boolean test)
  assertFalse(String comment, boolean test)
  assertEquals(String comment, Object ob1, Object ob2)

  In each case, when test fails, comment is displayed (so it should be used to say what the expected value was).
- Demo: lesson11.lecture.junit.test.*

Overview

1. Create your first JUnit test
2. Unit-testing Stream Pipelines
3. Handling Exceptions Arising in Stream Pipelines
4. Concurrent Processing and Parallel Streams

How to Unit Test Stream Pipelines?

A. The Problem. Stream pipelines, with lambdas mixed in, form a single unit; it is not obvious how to effectively unit test the pieces of the pipeline.

B. Two Guidelines:
   a. If the pipeline is simple enough, we can name it as an expression and unit test it directly.
   b. If the pipeline is more complex, pieces of pipeline can be called from support methods, and the support methods can be unit-tested.

Unit-Testing Stream Pipelines:

Simple Expressions

- Example: Test the expression:

```
Function<List<String>, List<String>> allToUpperCase =
    words -> words.stream()
    .map(word ->  word.toUpperCase())
    .collect(Collectors.toList());
```

- Can do ordinary unit testing of the allToUpperCase

```
@Test
public void multipleWordsToUppercase() {
  List<String> input = Arrays.asList("a", "b", "hello");
  List<String> result = Testing.allToUpperCase.apply(input);
  assertEquals(asList("A", "B", "HELLO"), result);
}
```

Unit-Testing Lambdas:

Complex Expressions

- Example:

```
Function<List<String>, List<String>> elementFirstToUpperCaseLambdas
    = words -> words.stream()
      .map(word -> {
          char firstChar =
          Character.toUpperCase(word.charAt(0));
          return firstChar + word.substring(1);
      })
      .collect(Collectors.toList());
```

- The key point to test is whether the expression for transforming a word so that its first letter becomes upper case is working.
- This can be done by replacing the lambda expression with a method reference together with an auxiliary method which can be placed in a companion class `LibraryCompanion`.

Unit-Testing Lambdas:
Complex Expressions (cont.)

```
public static List<String> elementFirstToUpperCaseLambdas(
                                    List<String> words) {
        return words.stream()
                .map(LibraryCompanion::firstToUpper)
                .collect(Collectors.toList());
}
```

//auxiliary method, used in method reference, in the class LibraryCompanion

```
public static String firstToUpper(String value) {
        char firstChar = Character.toUpperCase(value.charAt(0));
        return firstChar + value.substring(1);
}
```

- Now the key element of the original lambda can be tested directly.

```
@Test
public void twoLetterStringConvertedToUppercase() {
  String input = "ab";
  String result = LibraryCompanion.firstToUpper(input);
  assertEquals("Ab", result);
}
```

Unit-Testing Lambdas:

Complex Expressions

The example suggests a best practice for unit testing when lambdas are involved:

1. Replace a lambda that needs to be tested with a method reference plus an auxiliary method
2. Then you can test the auxiliary method

Main Point 1

Unit testing, in conjunction with Test-Driven Development, make it possible to steer a mistake-free course as programming code is developed. The self-referral mechanism of anticipating logic errors in unit tests, developed as the main code is developed, is analogous to the mechanism that leads awareness to be established in its self-referral basis; action from such an established awareness tends to make fewer mistakes.

Overview

1. Create your first JUnit test
2. Unit-testing Stream Pipelines
3. Handling Exceptions Arising in Stream Pipelines
4. Concurrent Processing and Parallel Streams

Handling Exceptions Arising in Stream Pipelines

- Review of exception handling - See demo code in lesson11.lecture.exceptions.

- Stream operations, like map and filter, that require a functional interface whose abstract method *does not have a throws clause* (like Function and Predicate), make exception-handling more difficult. See demo code to see issues and best possible solutions - lesson11.lecture.exceptions2

Main Point 2

Associated with exception-handling in Java are many well-known best-practices. For example: exceptions that can be caught and handled – *checked exceptions* – reflect the philosophy that, if a mistake can be corrected during execution of an application, this is better result than shutting the application down completely. Secondly, one should never leave a caught exception unhandled (by leaving a catch block empty). Third, one should never ask a catch block to catch exceptions of type Exception because doing so tends to be meaningless.

Likewise, Maharishi points out that, in life, it is better not to make mistakes, but, if a mistake is made, it is best to handle it, to apologize, so that the situation can be repaired; it is never a good idea to simply "ignore" a wrongdoing that one has done. Repairing a wrongdoing requires proper use of speech; an "apology" that does not really address the issue may be too general and may do more harm than good.

Overview

1. Create your first JUnit test
2. Unit-testing Stream Pipelines
3. Handling Exceptions Arising in Stream Pipelines
4. Concurrent Processing and Parallel Streams

Concurrent Processing and Parallel Streams - Overview

A. Introduction to threads
B. Working with threads: the Runnable interface
C. Thread safety and the synchronized keyword
D. Java 8 convenience class for invoking threads: the Executor class
E. When should you use parallel streams?

Introduction to Threads

1. A *process* in Java is an instance of a Java program that is being executed. It contains the program code and its current activity. A process has a self-contained execution environment. A process generally has a complete, private set of basic run-time resources; in particular, each process has its own memory space.

2. A *thread* is a component of a process. Multiple threads can exist within the same process, executing concurrently (one starting before others finish) and sharing memory (and other resources), while different processes do not share these resources. In particular, the threads of a process share the values of its variables at any given moment.

3. Every process has at least one thread, the *main thread* (the main method of a Java program starts up the main thread.) Other threads may be created from the main thread.

Introduction to Threads (cont.)

4. Examples of how multiple threads are used:
   a. One thread keeps a UI active while another thread performs a computation or accesses a database
   b. Divide up a long computation into pieces and let each thread compute values for one piece, then combine the results (computing in parallel)
   c. Web servers typically handle client requests on separate threads; in this way, many clients can be served "simultaneously."

Creating Threads in Java

Code that you wish to run in a new thread is contained in the run() method of a class that implements the Runnable interface.

```
interface Runnable {
    void run() ;
}
class MyRunnable implements Runnable {
    public void run() {
        System.out.println("Running a thread!");
  }
}
```

Creating Threads in Java (cont.)

The thread is then *spawned* when an instance of your class is used as an argument to the Thread constructor, and the start() method is called on the Thread instance.

The following code creates a thread and starts it:

```
MyRunnable myRunnable = new MyRunnable();
Thread t = new Thread(myRunnable);
t.start();
```

Runnable is a functional interface. Therefore, starting a new Thread can be done using lambdas:

```
Thread t = new Thread(() ->
    System.out.println("Running a thread"));
t.start();
```

Testing Singleton Using a Single Thread

```
public class Singleton {
    private static Singleton instance;
    public static int counter = 0;
    private Singleton() {
        incrementCounter();
    }
    public static Singleton getInstance() {
        if(instance == null) {
            instance = new Singleton();
        }
        return instance;
    }
    private static void incrementCounter() {
        counter++;
    }
}
```

```
public class SingleThreadedTest2 {
    public static void main(String[] args) {
        for(int i = 0; i < 1; ++i) {
            createAndStartThread();
            System.out.println("Num instances: " + Singleton.counter);
        }
    }
    public static void createAndStartThread() {
        Runnable r = () -> {
            for(int i = 0; i < 1000; ++i) {
                Singleton.getInstance();
            }
        };
        new Thread(r).start();
        try {
            Thread.sleep(10);
        } catch(InterruptedException e) {}
    }
}
```

As expected, only 1 instance is ever created.

Note: We have put each thread to sleep for 10 milliseconds before allowing the next one to start. If we do not do this, then the first 1 or 2 calls of createAndStart will record *0 instances.* This is because the change made by each thread may not be visible to the main thread immediately (this is most likely because processor memory is much faster than the RAM where the counter data is stored).

Testing Singleton Using Multiple Threads

```
public class Singleton {
    private static Singleton instance;
    public static int counter = 0;
    private Singleton() {
        incrementCounter();
    }
    public static Singleton getInstance() {
        if(instance == null) {
            instance = new Singleton();
        }
        return instance;
    }
    private static void incrementCounter() {
        counter++;
    }
}
```

```
public class MultiThreadedTest {
    public static void main(String[] args) {
        for(int i = 0; i < 10; ++i) {
            multipleCalls();
            System.out.println("Num instances: " + Singleton.counter);
        }
    }
    public static void multipleCalls() {
        Runnable r = () -> {
            for(int i = 0; i < 5000; ++i) {
                Singleton.getInstance();
            }
        };
        for(int i = 0; i < 1000; ++i) {
            new Thread(r).start();
        }
    }
}
```

The test shows that competing threads are creating multiple instances of the Singleton class. The test “instance == null” is being interrupted so that it appears to be true to more than one thread, and so the constructor is called multiple times.

Race Conditions and Thread Safety

1. When two or more threads have access to the same object and each modifies the state of the object, this situation is called a *race condition,* which arises when threads *interfere* with each other (the sequence of steps being executed by one thread is interrupted by another thread).
2. Code is said to be *thread-safe* if it is guaranteed to be free of race conditions when accessed by multiple threads simultaneously.
3. We can say therefore that this Singleton implementation is *not thread-safe*.

Forcing Serialized Access with synchronized

1. We can force threads to access the getInstance method of Singleton *one at a time* (this is called *serialized access*) by labeling getInstance with the keyword synchronized.

2. When a method is synchronized, in order for a thread to execute the method, it must *acquire the lock* for the instance of the object that the method is running on. Each object has an *instrinsic* lock, and a thread gains access to this lock when it calls the method, as long as no other thread has the lock. Once a thread executes the synchronzied method, the lock becomes available again and the next eligible thread (determined by the OS using thread priorities and other factors) then acquires the lock.

Forcing Serialized Access with synchronized (cont.)

***Note*:** This use of the word "serialized" has nothing to do with the Serializable interface that we examined earlier in the course

Note: We have seen that competing threads could corrupt the "== null" test. However, competing threads could also corrupt the counter since the increment operation counter++ is not atomic (it is in fact the assignment counter = counter + 1). Therefore, to be sure that the MultiThreadedTest is really producing multiple instances of Singleton, we must make the incrementCounter method synchronized, as in the code below. Running this test, and witnessing multiple calls to the Singleton constructor once again convinces us that multiple instances are being created.

```
public class Singleton2 {
    private static Singleton2 instance;
    public static int counter = 0;
    private Singleton2() {
        incrementCounter();
    }
    public static Singleton2 getInstance() {
        if(instance == null) {
            instance = new Singleton2();
        }
        return instance;
    }
    /* Guarantees proper count of instances */
    synchronized private static void incrementCounter() {
        counter++;
    }
}
```

Forcing Serialized Access with synchronized (cont.)

By making the `getInstance()` synchronized, we get a thread-safe implementation of Singleton.

```
public class SynchronizedSingleton {
    private static SynchronizedSingleton instance;
    public static int counter = 0;
    private SynchronizedSingleton() {
        incrementCounter();
    }
    synchronized public static SynchronizedSingleton getInstance() {
        if(instance == null) {
            instance = new SynchronizedSingleton();
        }
        return instance;
    }
    private static void incrementCour
        counter++;
    }
}
```

```
public class MultiThreadedTest3 {
    public static void main(String[] args) {
        for(int i = 0; i < 10; ++i) {
            multipleCalls();
            System.out.println(SynchronizedSingleton.getInstance());
            System.out.println("Num instances: " + SynchronizedSingleton.counter);
        }
    }
    public static void multipleCalls() {
        Runnable r = () -> {
            for(int i = 0; i < 5000; ++i) {
                SynchronizedSingleton.getInstance();
            }
        };
        for(int i = 0; i < 1000; ++i) {
            new Thread(r).start();
        }
        try {
            Thread.sleep(100);
        } catch(InterruptedException e) {}
    }
}
```

Starting and Managing Threads with Executor

1. When an application relies heavily on multi-threading, either because many threads are needed, or because threads need to be managed carefully, the “manual” approach to starting threads shown above is not optimal. To create and manage threads properly, Java has an Executor class.

2. Two examples of specialized Executor classes are those created by the factory methods Executors.newCachedThreadPool(), which is optimized for creation of threads for performing many small tasks or for tasks which involve long wait periods. For computationally intensive tasks, Java provides Executors.newFixedThreadPool(numThreads).

Starting and Managing Threads with Executor (cont.)

3. We modify our earlier code to make use of this the Executor class. We synchronize the getInstance method in SynchronizedSingleton.

```
public class SynchronizedSingleton {
    private static SynchronizedSingleton instance;
    public static int counter = 0;
    private SynchronizedSingleton() {
        incrementCounter();
    }
    synchronized public static SynchronizedSingleton getInstance() {
        if(instance == null) {
            instance = new SynchronizedSingleton();
        }
        return instance;
    }
    private static void incrementCounter() {
        counter++;
    }
}
```

```
public class MultiThreadedTestWithExec {
    private static Executor exec
       = Executors.newCachedThreadPool();
    public static void main(String[] args) {
        for(int i = 0; i < 10; ++i) {
            multipleCalls();
            System.out.println("Num instances: "
                    + SynchronizedSingleton.counter);
        }
    }
    public static void multipleCalls() {
        Runnable r = () -> {
            for(int i = 0; i < 500; ++i) {
                SynchronizedSingleton.getInstance();
            }
        };
        for(int i = 0; i < 100; ++i) {
            exec.execute(r);
        }
        try {
            Thread.sleep(10);
        } catch(InterruptedException e) {}
    }
}
```

Note: You may notice that the program waits a bit after the last printout. It terminates when the pooled threads have been idle for a while; after some time, the executor terminates them.

Guidelines for Using Parallel Streams

1. When you create a parallel stream from a Collection class in order to process elements in parallel, Java handles this request by partitioning the collection and processing each piece with a separate thread. Not every Stream operation, nor every underlying collection type, is amenable to parallel processing. We give general guidelines for choosing between sequential and parallel streams.

2. Some Guidelines
   A. Don't use parallel streams if the number of elements is small – the improved performance (if any) will not in this case outweigh the overhead cost of working with parallel streams.
   B. Operations that depend on the order of elements in the underlying collection should not be done in parallel. Example: findFirst, limit.
   C. If the terminal operation of the stream is expensive (example: collect(Collectors.joining)), you must remember that it will be executed repeatedly in a parallel computation – this could be a good reason to avoid parallel streams in this case.
   D. Translation between primitives and their object wrappers becomes very expensive when done in parallel. If you are working with primitives, use the primitive streams, like IntStream and DoubleStream.
   E. Certain data structures can be divided up more efficiently than others – ArrayList, HashSet and TreeSet can be partitioned efficiently, but LinkedLists cannot.

Connecting the Parts of Knowledge With the Wholeness of Knowledge

Annotations

1. Executing a Java program results in algorithmic, predictable, concrete, testable behavior.
2. Using annotations, it is possible for a Java program to modify itself and interact with itself.

---

3. *Transcendental Consciousness* is the field self-referral pure consciousness. At this level, only one field is present, continuously in the state of knowing itself.
4. *Impulses Within the Transcendental Field*. What appears as manifest existence is the result of fundamental impulses of intelligence within the field of pure consciousness. These impulses are ways that pure consciousness acts on itself, interacts with itself.
5. *Wholeness Moving Within Itself.* In Unity Consciousness, the diversity of creation is appreciated as the play of fundamental impulses of one's own nature, one's own Self.