
MAHARISHI UNIVERSITY of MANAGEMENT
Engaging the Managing Intelligence of Nature
Computer Science Department
CS401 Modern Programming Practices (MPP)
Professor Paul Corazza
1

Lecture 10: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 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. **Unit-testing Stream Pipelines**
2. Introduction to Annotations
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: Perform a unit test on the following pipeline:

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

- Can do ordinary unit testing of `allToUpperCase`.

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

Unit-Testing Stream Pipelines:

ComplexExpressions

- **Example**: Sorts Employees first by name (ascending), then by salary in descending order

```
//difficult to test
static Function<List<Employee>, List<Employee>> employeeSorter =
    list -> list.stream()
    .sorted(Comparator.comparing((Employee e) -> e.getName())
                      .thenComparing((Employee e) -> e.getSalary(),
                                     Comparator.reverseOrder()))
    .collect(Collectors.toList());
```

- The key point to test is whether any two Employees are being compared in the correct way.
- This can be done by replacing the lambda expression that defines the Comparator with a method reference, together with an auxiliary method that can be placed in a companion class `LibraryCompanion`.
- Demo: lesson10.lecture.libcompanion

Strategy: Refactor the pipeline using auxiliary methods that can be tested separately.

```
static Function<List<Employee>, List<Employee>> employeeSorter =
list -> list.stream()
        .sorted(LibraryCompanion::compareEmps)
        .collect(Collectors.toList());
```

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

```
public class LibraryCompanion {
    static Comparator<Employee> empComp
       = Comparator.comparing((Employee e) -> e.getName())
            .thenComparing((Employee e) -> e.getSalary(),
                Comparator.reverseOrder());
    public static int compareEmps(Employee e1, Employee e2) {
        return empComp.compare(e1, e2);
    }
}
```

Now the key element of the original lambda can be tested directly:

```
//verify that jim comes before joe
assert(LibraryCompanion.compareEmps(jim, joe1) < 0);

//verify that joe2 comes before joe1
assert(LibraryCompanion.compareEmps(joe2, joe1) < 0);
```

Unit-Testing Stream Pipelines:

ComplexExpressions

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

Exercise 10.1

You are developing unit tests and need to decide how to test the lambda expression shown below. Should you treat it as a simple expression or a complex expression? Create a unit test to test it. All necessary files are in package lesson10.exercise_1 in the InClassExercises project. Use the class TestLambda for your test.

```
// A list of Customers whose checking account balance is > 50,
//sorted by customer's last name
public static List<Customer> specialAccounts(List<Account> accounts) {
        return accounts.stream().filter(a -> a.getBalance() > 50)
                        .map((Account a) -> a.getCustomer())
        .sorted(Comparator.comparing((Customer c) -> c.getLastName()))
        .collect(Collectors.toList());
}
```

Overview

1. Unit-testing Stream Pipelines
2. **Introduction to Annotations**
3. Handling Exceptions Arising in Stream Pipelines
4. Concurrent Processing and Parallel Streams

What Are Annotations?

We have seen them in various contexts already:

a. @Test - JUnit 4
b. @Override – to indicate that a method is being overridden
c. @FunctionalInterface – to indicate that an interface is functional and may be used with lambdas
d. @Deprecated – to discourage use of a method or class
e. @SuppressWarnings – to hide warning messages of various kinds
f. Javadoc annotations:
    i. @author
    ii. @since
    iii. @version

What Are Annotations? (cont.)

2. Annotations are tags that are inserted into source code so that some tool can process them. The tools can operate on the source level, or they can process class files into which the compiler has placed annotations.

3. To benefit from annotations that you create, you need to select a *processing tool*. You need to use annotations that your processing tool understands, then apply the processing tool to your code.
   - JUnit processes its @Test annotation
   - Java compiler processes the others shown

4. Annotations can be applied to a class, a method, a variable – in fact, anywhere qualifiers like public and static may be used

What Are Annotations? (cont.)

5. Annotations may have zero or more *elements*. Here is an example of a user-defined annotation that has two elements, `assignedTo` and `severity`.

```
@BugReport(assignedTo="Harry", severity=10)
```

[This annotation could be applied at the class level. Like any user-defined annotation, it would require an external tool to process it.]

6. When an annotation has just one element and its name is "`value`", the following more compact form can be used:

```
@SuppressWarnings("unchecked")
[same as @SuppressWarnings(value = "unchecked")]
```

7. If two annotations of the same type are used on a class or elsewhere, then these two together are said to be a repeating annotation:

```
@Author(name = "Jane Doe")
@Author(name = "John Smith")
class MyClass { ... }
```

Repeating annotations are supported as of the Java SE 8 release

What Are Annotations? (cont.)

8. *User-defined annotations*. The @interface keyword is the way the Java compiler knows you are creating an annotation; such "classes" extend the Annotation interface. See the demo lesson10.lecture.annotation

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE)
public @interface BugReport {
    String assignedTo() default "[none]";
    int severity() default 0;
}
```

Note: the code shows how the elements "assigned to" and "severity" are defined.

See package `lesson10.lecture.annotation`

Annotation Reference

@Retention annotation specifies how the marked annotation is stored:

- RetentionPolicy.SOURCE – The marked annotation is retained only in the source level and is ignored by the compiler.
- RetentionPolicy.CLASS – The marked annotation is retained by the compiler at compile time, but is ignored by the Java Virtual Machine (JVM).
- RetentionPolicy.RUNTIME – The marked annotation is retained by the JVM so it can be used by the runtime environment.

@Target annotation marks another annotation to restrict what kind of Java elements the annotation can be applied to. A target annotation specifies one of the following element types as its value:

- ElementType.ANNOTATION_TYPE can be applied to an annotation type.
- ElementType.CONSTRUCTOR can be applied to a constructor.
- ElementType.FIELD can be applied to a field or property.
- ElementType.LOCAL_VARIABLE can be applied to a local variable.
- ElementType.METHOD can be applied to a method-level annotation.
- ElementType.PACKAGE can be applied to a package declaration.
- ElementType.PARAMETER can be applied to the parameters of a method.
- ElementType.TYPE can be applied to any element of a class.

Exercise 10.2

Work with the Bug Report code shown in class (available in package `lesson10.exercise_2` in `InClassExercises`). A class `StillMoreBadCode` has been added. Add to this class annotation values for the elements "assigned to" and "severity". Then integrate this new class into the AnnotationInfo class and run it. You should now see info about `StillMoreBadCode` in the output report.

Overview

1. Unit-testing Stream Pipelines
2. Introduction to Annotations
3. **Handling Exceptions Arising in Stream Pipelines**
4. Concurrent Processing and Parallel Streams

Handling Exceptions Arising in Stream Pipelines

1. Ordinary functional expressions, composed in a pipeline, may throw exceptions, but very often exception-handling can be done in the usual way. See demo code in `lesson10.lecture.exceptions`.

2. However, stream operations, like map and filter, that require a functional interface whose unique 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. See `lesson10.lecture.exceptions2`

3. The best one can do in these situations is to convert checked exceptions to `RuntimeException`s. The code can be made more readable and compact if the `try/catch` clause that is needed can be tucked away in an auxiliary method.

Overview

1. Unit-testing Stream Pipelines
2. Introduction to Annotations
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. Multiple threads are typically invoked to perform multiple tasks simultaneously, or to simulate simultaneous execution of multiple tasks. In a multiprocessor environment, different threads can access different processors; in a single processor environment, multiple threads can appear to work simultaneously by virtue of *time-slicing* – the operating system allots portions of time to competing threads.

5. Within a single process, only one thread can be running at a time. Java's Thread Scheduler, working with the operating system, decides which thread is allowed to run at any given time.

Introduction to Threads (cont.)

6. 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();
```

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 < 10; ++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 stopped and another thread begins to execute).
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*.

Atomic Operations Cannot Be Interrupted

- We have seen that the parts of an if-then statement can be interrupted by the thread scheduler.
- The Java Specification guarantees that "reading and writing values" are "atomic" operations – they cannot be interrupted. Examples:

- This assignment statement is atomic:

```
int x = 2;
```

- This setter is atomic:

```
private int x;
public void setX(int x) {
  this.x = x;
}
```

- The increment operator is *not* atomic:

```
int x = 2;
x++;  //not atomic since it is same as x = x + 1.
```

Compiling x = x +1:
- x + 1 is computed and stored in temp variable y
- y is stored in variable x

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++;
    }
}
```

Main Point 1

Code that will perform flawlessly in a single-threaded environment may bring disastrous consequences in a multi-threaded environment if shared data is not managed properly.

In life, a similar situation exists because the particulars of life are always changing. In any domain of life – work, family, health, society – a situation that appears favorable today may become unfavorable tomorrow. How does one prepare for change that cannot be predicted? The only way is to be established in a part of oneself that does not change, no matter how the world changes. That part is first experienced as simple silence in the practice of TM. In time, with regular practice, it becomes firmly established.

Starting and Managing Threads with Executor

1. When many threads need to be managed, starting threads in the way shown above is not optimal. To create and manage threads properly, Java has an `Executor` class.

2. Two examples of specialized `Executor` classes:
   - `Executors.newCachedThreadPool()` - optimized for creation of threads for performing many small tasks or for tasks which involve long wait periods.
   - `Executors.newFixedThreadPool(numThreads)` – for computationally intensive tasks

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, works well with parallel processing. We give general guidelines for choosing between sequential and parallel streams.

Guidelines for Using Parallel Streams (cont.)

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`.
   C. If the terminal operation of the stream is expensive, 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. Until you develop a degree of expertise in working with parallel computations, it is a good idea to benchmark the performance of a pipeline executed in parallel and compare with the performance of the sequential version.

Sample Benchmarks for Sequential vs Parallel Processing

Warburton, *Java 8 Lambdas* (p. 84) gives an example of a benchmark test that makes a convincing case for choosing parallel processing over sequential processing for a particular task.

Sequential Version of Code

```
public int serialArraySum() {
    return albums.stream()
                 .flatMap(Album::getTracks)
                 .mapToInt(Track::getLength)
                 .sum();
}
```

//counts the total number of
//tracks on all albums

Sample Benchmarks for Sequential vs Parallel Processing (continued)

Parallel Version of Code

```
public int parallelArraySum() {
    return albums.parallelStream()
                 .flatMap(Album::getTracks)
                 .mapToInt(Track::getLength)
                 .sum();
}
```

Warburton reports that, when running on a quad core Windows machine, when the number of albums was just 10, the sequential version was 8x faster; when number of albums was 100, the two versions were equally fast; when the number of albums was 10,000, the parallel version was 2.5x faster.

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.