**Lesson 10**Best Programming Practices with Java 8:*Living Life in Accord with Natural Law*

**Wholeness of the Lesson**

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.

**W2D4 : What Are Annotations?**

1. We have seen them in various contexts already:
   1. @Test - JUnit 4
   2. @Override – to indicate (with compiler check) that a method is being overridden
   3. @FunctionalInterface – to indicate that an interface is functional and may be used with lambdas
   4. @Deprecated – to discourage use of a method or class
   5. @SuppressWarnings – to hide warning messages of various kinds
   6. Javadoc annotations:
      1. @author
      2. @since
      3. @version
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

1. Annotations can be applied to a class, a method, a variable – in fact, anywhere qualifilers like public and static may be used
2. 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 users-defined annotation, it would **require an external tool to process it**.]

1. 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")]

1. If the annotations have the same type, then this is called a repeating annotation:

@Author(name = "Jane Doe")

@Author(name = "John Smith")

class MyClass { ... }

Repeating annotations are supported as of the Java SE 8 release

**NOW GO TO POWERPOINT FILE!! (slides 5-6)**

**DO THIS W2D4 : Summary of Best Practices**(Optional Module #2)

1. *Log when exception first arises.* When an exception is first caught, information about the state of the object should be logged – logging can be done in a catch block. However, if the try/catch are inside a loop that has many iterations or that could even possibly fail to terminate, logging should be done outside the loop.

|  |  |
| --- | --- |
|  |  |
| **Obtain instance of jdk Logger** | **Write to the LOG. Use LOG.severe, LOG.warning, LOG.info, LOG.fine** |

1. ***Handler of exception should be chosen carefully.* An exception should be handled by an object that “knows” what to do with the exception – typically, it should have the responsibility of communicating a message to the user. Therefore, when an exception is thrown, it should propagate up the call stack until it reaches an appropriate handler**.
2. *Never create an “empty” catch block.* Exceptions should never be ignored, as in

This is *so* bad…

try { . . .

} catch {}

If nothing needs to be done, there should at least be a comment stating this fact – and probably some message to the log – rather than dead silence.  
 try { . . .

This is acceptable

} catch {

//nothing needed here

LOG.info(“Exception thrown by . . .”);

}

**W2D4 : DO THIS - Read :**

1. *Never catch Exception or Throwable.* Your code should (almost) never catch Exception or Throwable. One reason is that doing so means that you will be handling any RuntimeExceptions that are thrown (like NullPointerException), and **these should not** be caught. [One exception to this rule arises sometimes when communicating with external APIs – it may not always be possible to anticipate which types of Exceptions will be thrown by API methods, and you may want to make sure your application does not shut down because of an uncaught exceptions coming from the outside.]
2. ***Always*** *validate input arguments.* Important methods that take input arguments should validate input values and throw an IllegalArgumentException in case of invalid inputs.

void myMethod(String arg) {

if(arg == null || arg.length() == 0)

throw new IllegalArgumentException(“Input must be nonempty”);

//more

}

Note that throwing any type of RuntimeException never requires a throws declaration (in the first line of the method).

1. *Don’t throw instances of RuntimeException.* If you need to throw some kind of runtime exception, either use one of the specific subclasses of RuntimeException available in the Java libraries (as in the previous example: IllegalArgumentException, or others: IllegalStateException, NumberFormatException) or, if nothing fits, create your own subclass of RuntimeException. **Never simply throw a RuntimeException – it is too general.**

|  |  |
| --- | --- |
|  | This is *bad*… |
|  |  |
|  | This is *good*… |

7. W2D4 : *Using a finally block*.

A. *Always executes*. When finally is used, the code in the finally block is executed even if the try block succeeds and returns (finally block executes before performing the return) or an exception is thrown. When an exception is thrown and caught, before control is passed up the stack, finally clause executes; when it is not caught, before a stack trace is displayed, finally clause is executed.

B. *Used for cleanup.* Traditionally, finally is used to clean up resources before exiting the application. Files are closed, database connections closed, etc. Java 7/8 provides a new approach (*try with resources*) to handle this pattern – discussed below

C. ***No return statement in a finally block****.* A return statement should not occur in a finally block – if the try block also has a return statement, then the finally block’s return statement will be the one that executes. (Could have a different argument in there (JL)).

D. ***Do not throw an exception within a finally block****.* An exception should not be thrown from within a finally block – if an exception is thrown during execution of the try block, and then in the finally block another exception is thrown, the exception from the finally block is the one that is actually thrown.

**W2D4 - Setting up the JDK Logger**(Optional Module #2)

1. The JDK Logger can always be accessed like this:

private final static Logger LOG = Logger.getLogger(<any string>)

The string argument should be the current package name. Typical way of obtaining this string:  
 MyClass.class.getPackage().getName();  
  
The top-level logger is indicated by the empty string “”:  
 Logger LOG = Logger.getLogger(“”);  
  
There is also a global logger that can be obtained like this:  
 Logger LOG = Logger.getGlobalLogger();  
For smaller applications, this one is fine to use.

1. Configuring the Logger
2. For production-quality logging, configuration should be done using the logging.properties file that comes with Java. Details about this are available in the setup folder for this course in the directory logging. (This is an FPP topic.)
3. Log configuration can also be done in code. For this course, the logger can be configured using logsetup.jar. The global logger can be configured using this jar file with a call (in application startup):  
    LogSetup.*setup*();  
   The setup method does the following:
4. Provides simple output messages to the console. You can create one of these messages with these calls:  
    LOG.config(<message>), LOG.info(<message>),

LOG.warning(<message>), LOG.severe(<message>)

1. Provides XML-formatted messages to a log file, placed at the top level of your src directory: src\logs

Demos: lesson10.lecture.logging.defaultlogging, lesson10.lecture.logging.defaultlogging2

**Tricky try/catch/finally Situations**(Optional Module #2)

1. *Avoid memory leaks.* When your application uses external resources, like files or a database connection, it is important to close the connections after your application has finished using them. Typically, using these connections involves checked exceptions; but whether or not an exception is thrown, your application must disconnect from the resource, or there can be a memory leak, causing memory to fill up.
2. ***Do clean-up in a finally block****.* The usual way to clean up resources is in a finally block, which will execute whether or not an exception is thrown.
3. But, what if closing a resource is also capable of throwing a checked exception? How should this be handled?  
     
   Demo: lesson10.lecture.trickycatch1

**Problems with the lesson10.lecture.trickycatch1 Solution:**

1. It’s messy
2. It is possible to make it more readable by separating the part that does interesting work (opening and reading a file) from the part that handles exceptions.   
     
   See lesson10.lecture.trickycatch2

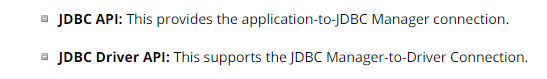
In this approach, the code is separated into inner and outer try blocks. Inner try/finally block does file processing and outer try/catch block takes care of exception-handling.

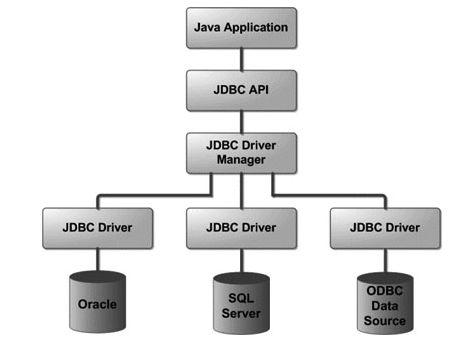
**Problems with Both Solutions:**

In both solutions, an IOException could be thrown for two different reasons. The first of these would indicate a difficulty in finding or reading the file; the second would arise because the readers could not be closed. Only the first is of any interest, but if both are triggered, only the second one is thrown.

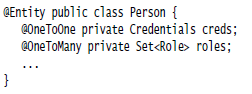
In Java 7, a new feature was added that allows you to add “suppressed” exceptions to a main exception, and then access them from the main exception as desired. The examples in lesson10.lecture.trickycatch3\_suppressed show how this can be done here. Note that the code is rather complicated!

1. W4D1 - The JDBC Architecture



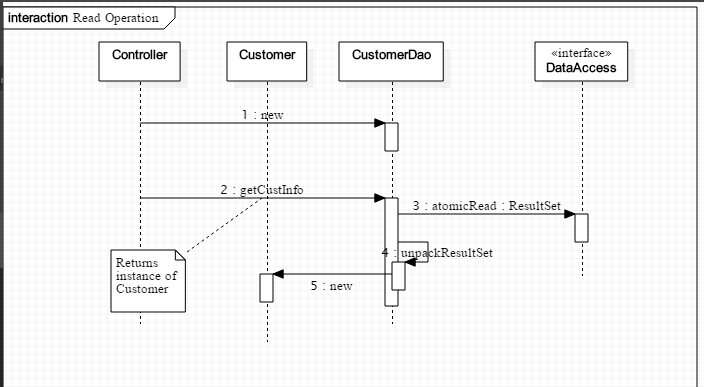
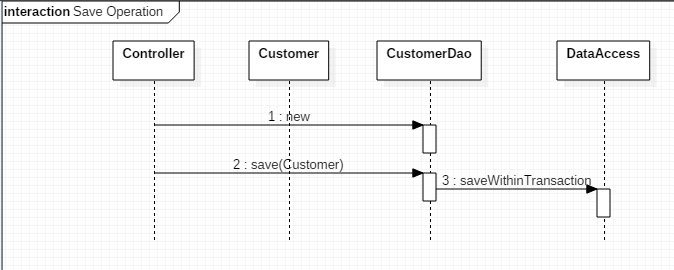


**Data Access and Frameworks**

1. For real production code, a more systematic approach to accessing the data source is followed. These days, systems rely on framework support, which provides an interface that gives you access not only to the **database**, but to a robust context for working with data, including transactional support, security and access control, data visualization, and perhaps most importantly, a uniform way of reading and writing data that hides the details of creating connections and executing statements.
2. Typically, you gain access to a framework solution **simply by adding one or more jar files** to your project.
3. *Two Approaches.* There are two styles of framework support these days
   1. *ORM* (object-relational mapping) – JPA and Hibernate use this approach
   2. *DAOs* (data access objects) – Spring supports this approach with its JDBC templating
4. JPA sample.
   1. Classes that need to be persisted (like Address, Customer, etc) are called *entities*
   2. In JPA, you insert annotations in an entity class to tell the framework information about reading and writing its data.  
       ****
   3. An EntityManager is invoked to save entity classes to the database and also to read data from a table into one of the entity classes.

|  |  |
| --- | --- |
|  |  |
| **Persisting/Removing** | **Reading** |

1. DAO Sample. In the DAO approach, classes that are persistent (Address, Customer, etc) are associated with corresponding DAO classes, which know how to interact with the data access layer. For instance, a Customer class would be associated with a CustomerDao. Reads and writes of Customer are then facilitated by CustomerDao.

See demo: lesson10.lecture.jdbc.framework; your must add dataaccess.jar to the project  
  
  
  
  


**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.  
   lesson10.lecture.exceptions2
3. The best one can do in these situations is to convert checked exceptions to RuntimeExceptions. 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. Examples are provided in lesson10.lecture.exceptions3, lesson10.lecture.exceptions.connectold, and lesson10.lecture.exceptions.connectnew

**Do This One - 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 a 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, 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.

**Concurrent Processing and Parallel Streams  
Overview**

1. Introduction to threads
2. Working with threads: the Runnable interface
3. Thread safety and the synchronized keyword
4. Java 8 convenience class for invoking threads: the Executor class
5. When should you use parallel streams?

**W2D4 - 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.

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. 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.”

**Done Before - 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!”);

}

}

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::

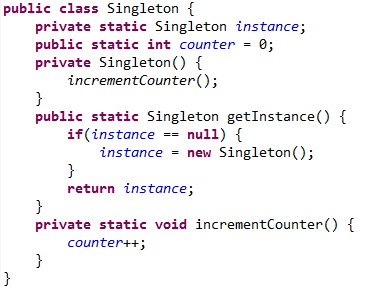
MyRunnable myRunnable = new MyRunnable();

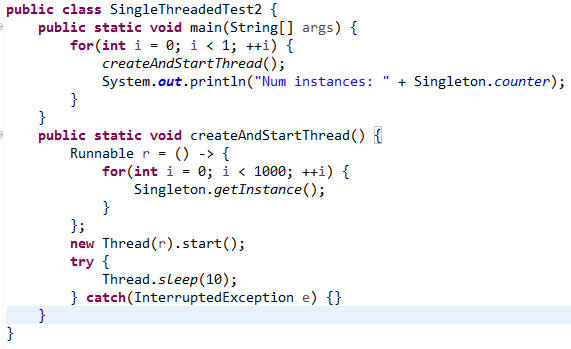
Thread t = new Thread(myRunnable);

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();

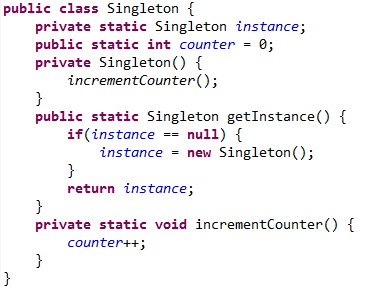
**Testing Singleton Using a Single Thread - DO THIS!!**

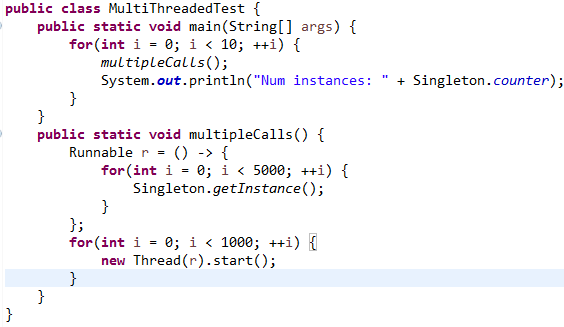




As expected, only 1 instance is ever created.  
  
Note: We have put each thread to sleep for 10 milliseconds before allowing the next one to start. If we do not do this, then the first 1 or 2 calls of createAndStart will record *0 instances.* This is because the change made by each thread may not be 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**





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 (before the first instance is ever created JL), and so the constructor is called multiple times(We have a ‘race’ condition(JL)).

**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.***

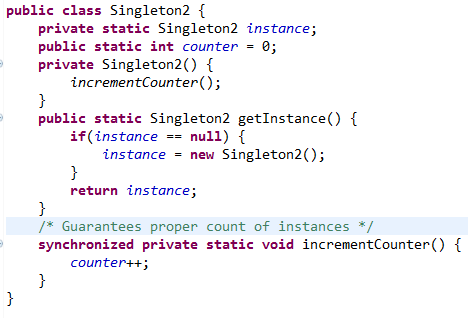
**W2D4 - Forcing Serialized Access with synchronized**

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 (a monitor).

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 finishes executing 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.

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

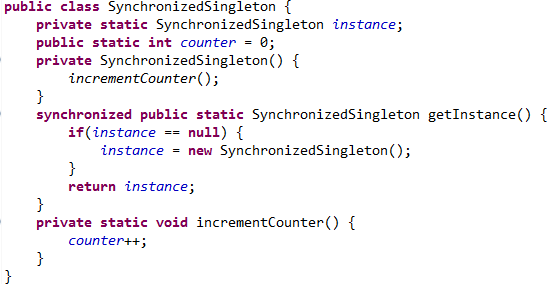
*Note:* We have seen that competing threads could corrupt the “== null” test. However, competing threads could also corrupt the counter since the increment operation counter++ is not atomic (it is in fact the assignment counter = counter + 1). Therefore, to be sure that the MultiThreadedTest is really 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.**

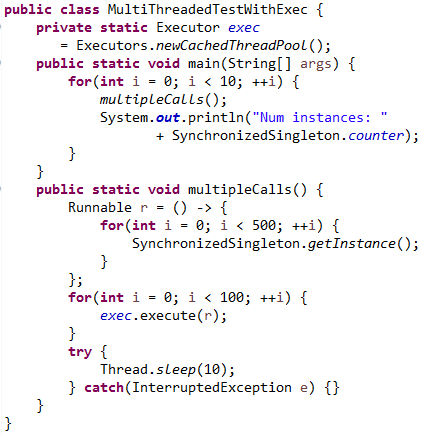
  
**The getInstance method should also have the keyword synchronized (a monitor)(JL).**

**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).

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




**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.

**Give ‘my’ threads materials on W2D4!**

**Give my threads homework!**

**Finish the Powerpoint file here!**

# Connecting the Parts of Knowledge With the Wholeness of Knowledge

# Best Programming Practices :

|  |  |
| --- | --- |
| 1. Executing a Java program results in algorithmic, predictable, concrete, testable behavior.   2. By using best programming practices from experience, our projects are more stable, more flexible, and more robust.  3*. Transcendental Consciousness* is the field of 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. | arrow |