Lesson 11

Unit Testing and Exception Handling

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.

Overview

- Test-Driven Development and Unit Testing (Optional Module #1)
- Unit-testing Stream Pipelines
- Introduction to Annotations
- Review of Exception Handling in Java (Optional Module #2)
- Handling Exceptions Using Java 8's try-with-resources
- Review of Basic JDBC Programming & Advanced JDBC Programming (Optional Module #3)
- Handling Exceptions Arising in Stream Pipelines
- Concurrent Processing and Parallel Streams

The Test-Driven Development (TDD) Paradigm (Optional Module #1)

1. The TDD paradigm says that the best testing strategy is to develop tests as part of the implementation process. Some even say that test code for a method or a class should be written before the actual code for the method or class is written. (TDD originated in the Extreme Programming development process, but is used these days regardless of the process used.)

2. TDD follows these steps

   a. First the developer writes an (initially failing) automated test case that defines a desired improvement or new function
   b. Next he produces the minimum amount of code to pass that test
   c. Finally, he refactors the new code to acceptable standards.

3. Benefits of TDD (See the Wikipedia article)

   a. Studies
      a. A 2005 study found that using TDD meant writing more tests and, in turn, programmers who wrote more tests tended to be more productive.
      b. Madeyski (Springer, 2010) provided an empirical evidence (involving 200 developers) that the TDD approach led to bettter OO design and more thorough and effective testing of a code base than the Test-Last approach.
      c. Surveys of developers (many examples of this) reveal that developers find significantly less need to debug code when they use TDD.
      d. Müller and Padberg ("About the Return on Investment of Test-Driven Development" – pdf available online) showed that, while it is true that more code is required with TDD, the total code implementation time is often shorter.
   b. *Better handling of requirements.* Since coding is closely related to testing, developer tends to think more effectively about how the code will be used – the result is that requirements are met with at a higher success rate.
   c. *Catch defects early* – this leads to a reduction in overall cost
   d. *Modular, flexible, extensible code.* TDD leads to more modularized, flexible, and extensible code because developers think of the software in terms of small units that can be written and tested independently and integrated together later

4. Example (see package lesson10.lecture.tdd)

Best Practices When Unit-Testing (Optional Module #1)

- Separate common set up logic into support methods.
- Keep each test focused on only the results necessary to do the necessary validation..
- Treat your test code with the same respect as your production code. It also must work correctly for both positive and negative cases, last a long time, and be readable and maintainable. In particular, tests should not depend on data or other aspects of the system that are likely to change (example: relying on the number of records in a table as part of a test)

Things to avoid

- Having test cases depend on system state manipulated from previously executed test cases.
- Dependencies between test cases. A test suite where test cases are dependent upon each other is brittle and complex. Execution order should not be presumed.
- Building “all-knowing oracles.” An oracle that inspects more than necessary is more expensive and brittle over time.
- Slow running tests.

Unit-Testing Tools

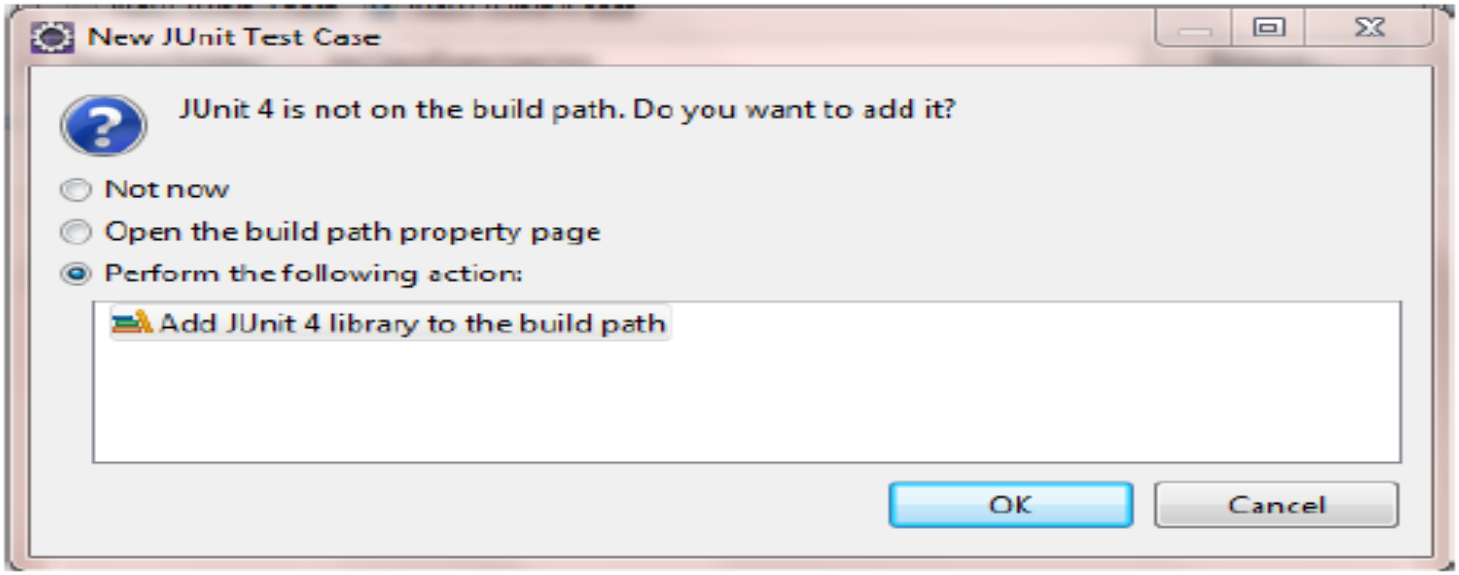
(Optional Module #1)

Tool	Description
Cactus	A JUnit extension for testing Java EE and web applications. Cactus tests are executed inside the Java EE/web container.
GrandTestAuto	Comprehensive Java software test tool not related to xUnit series of tools
Jtest	Commercial unit test tool that provides test generation and execution with code coverage and runtime error detection
JUnit 4.0	**Standard unit test tool; more flexible with release of version 4.0**
JUnitEE	A variant of JUnit that provides JEE testing
Mockito	Unit testing with mock objects
TestNG	Actually a multi-purpose testing framework, which means its tests can include unit tests, functional tests, and integration tests. Further, it has facilities to create even non-functional tests (as loading tests, timed tests). It uses Annotations since first version and is a framework more powerful and easy to use than the most used testing tool in Java: JUnit

Review: Setting Up JUnit 4.0

(Optional Module #1)

- 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.



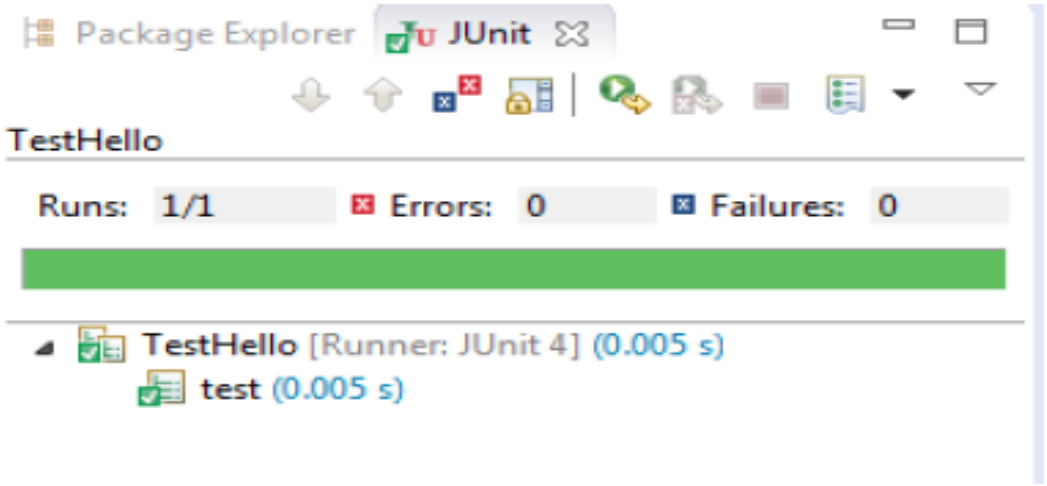
- Select "Perform the following action" → Add JUnit 4 library to the build path → OK
- Then you will see that JUnit 4 has been added to the Java Build Path by right clicking your project name → properties → Java Build Path.

- 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. ☺



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

  assertTrue(boolean test)

  assertFalse(boolean test)

  assertEquals(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:
- see junittestpack;

Overview

- Test-Driven Development and Unit Testing (Optional Module #1)
- Unit-testing Stream Pipelines
- Introduction to Annotations
- Review of Exception Handling in Java (Optional Module #2)
- Handling Exceptions Using Java 8's try-with-resources
- Review of Basic JDBC Programming (Optional Module #3)
- Advanced JDBC Programming
- Handling Exceptions Arising in Stream Pipelines
- Concurrent Processing and Parallel Streams

How to Unit Test Stream Pipelines?

- 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.

- Two Guidelines:
  - If the pipeline is simple enough, we can name it as an expression and unit test it directly.
  - 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 Stream Pipelines:

- Complex Expressions : Sometimes you want to use a lambda expression that exhibits complex functionality. (Here inside map)

```
public static List<String> uppercaseFirstChar1(List<String> words) {
   return words.stream()
           .map(value -> {
              char firstChar = value.charAt(0);
   firstChar = Character.toUpperCase(firstChar);
              return firstChar + value.substring(1);
           })
           .collect(Collectors.toList());
}
```

Do use method references. Any method that would have been written as a lambda expression can also be written as a normal method and then directly referenced elsewhere in code using method references.

```
public static List<String> uppercaseFirstChar(List<String> words) {
   return words.stream()
           .map(FunctionClass::firstToUppercase)
           .collect(Collectors.toList());
}
@Test
public void test1(){
    String input = "ab";
   String result = fc.firstToUppercase(input);
   assertEquals("Ab", result);
}
```

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

Unit-Testing Lambdas: Complex Expressions

- The example suggests a best practice for unit testing when lambdas are involved:
  - Replace a lambda that needs to be tested with a method reference plus an auxiliary method
  - 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

- Test-Driven Development and Unit Testing (Optional Module #1)
- Unit-testing Stream Pipelines
- Introduction to Annotations
- Review of Exception Handling in Java (Optional Module #2)
- Handling Exceptions Using Java 8's try-with-resources
- Review of Basic JDBC Programming (Optional Module #3)
- Advanced JDBC Programming
- Handling Exceptions Arising in Stream Pipelines
- Concurrent Processing and Parallel Streams

Introduction to Annotations

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

3. Annotations can be applied to a class, a method(@Override,@Test), a variable (@FXML)– in fact, anywhere qualifiers like public and static may be used.

4. If the annotations have the same type, then this is called a repeating annotation: This tag useful in JavaDoc.

@author(name = "Jane Doe") or **@author Jane Doe**
@author(name = "John Smith")
class MyClass { ... }
Repeating annotations are supported as of the Java SE 8 release

Java Pre Defined Annotations

- **@SuppressWarnings** specifies that one or more warnings that might be issued by the compiler are to be suppressed. The warnings to suppress are specified by name, in string form. This annotation can be applied to any type of declaration.
- It occurs when a method or attribute that should be no longer used.
- Example :

```
@SuppressWarnings("unused")
    int x = 10;
@SuppressWarnings({ "rawtypes", "unused" })
    List words = new ArrayList();
```

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")]
words.add("hello"); // this causes unchecked warning, because list is raw type
```

@Deprecated is a marker annotation. It indicates that a declaration is obsolete and has been replaced by a newer version.

```
Date d = new Date();
d.getDay(); // Deprecated Method
```

Own Annotation

User-defined annotations. The `@interface` keyword is the way the Java compiler knows you are creating an annotation; such "classes" extend the `Annotation` interface.

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

See the demo `lesson10.lecture.annotation`

@Retention `@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 `@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.

Another Example

```
@Documented
@Retention(RetentionPolicy.RUNTIME)
public @interface Author {
   String firstName();
   String lastName();
   boolean internalEmployee();
}
```

It can be used in your class as

```
@Author(firstName="John",lastName="Guddell",internal
Employee=true)
public class Test1 {
}
```

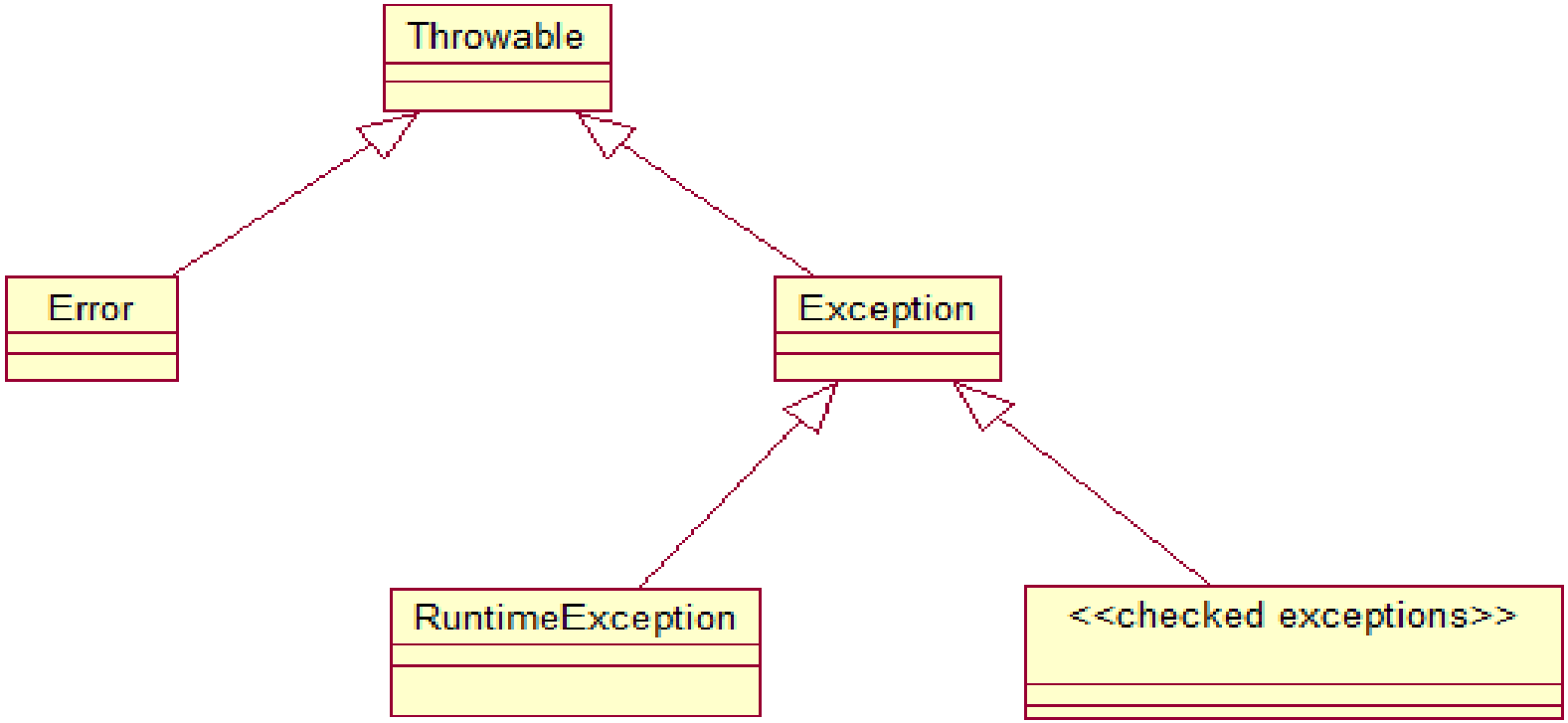
Overview

- Test-Driven Development and Unit Testing (Optional Module #1)
- Unit-testing Stream Pipelines
- Introduction to Annotations
- Review of Exception Handling in Java (Optional Module #2)
- Handling Exceptions Using Java 8's try-with-resources
- Review of Basic JDBC Programming (Optional Module #3)
- Advanced JDBC Programming (Optional Module #3)
- Handling Exceptions Arising in Stream Pipelines
- Concurrent Processing and Parallel Streams

Review of Exception-Handling in Java

(Optional Module #2)

- In Java, error conditions are represented by Java classes, all of which inherit from Throwable.



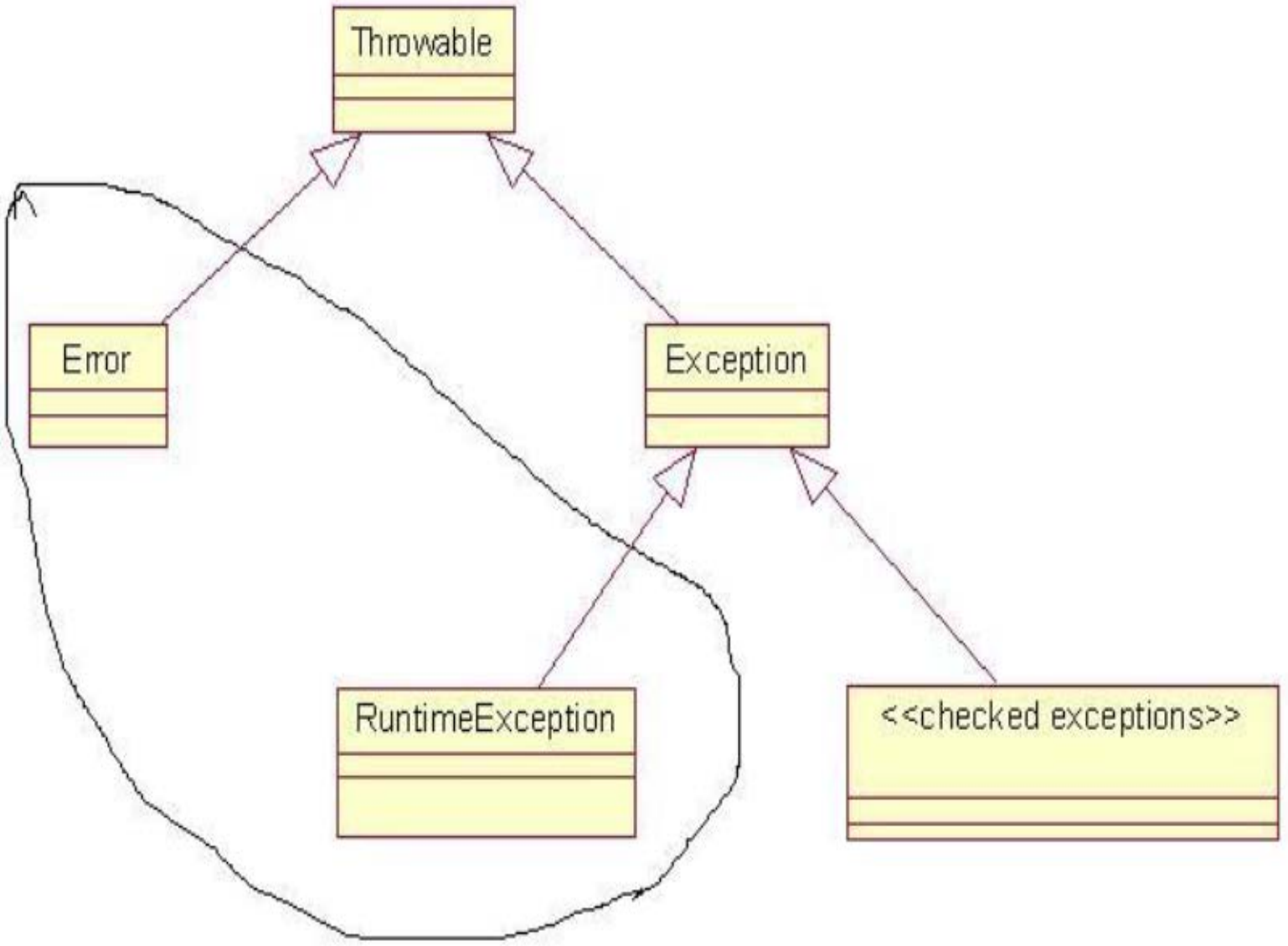
The Hierarchy of Exception Classes

Classification of Error-Condition Classes

- In Java, error-condition classes belong to one of three categories:
  - ***Error –*** Objects in this category belong to the inheritance hierarchy headed by the Error class. Examples include OutOfMemoryError, StackOverflowError, VirtualMachineError. If these are thrown, it indicates an unrecoverable error and the application should terminate.

  - ***Other Unchecked Exceptions*** – Besides Error objects, unchecked exceptions include all objects that belong to the inheritance hierarchy headed by the class RuntimeException. Examples include NullPointerException, ArrayIndexOutOfBoundsException, NumberFormatException. When these are thrown, it indicates that a programming error has occurred and needs to be fixed. Typically, these types of exceptions are not caught and handled – they simply indicate that some logic error needs to be corrected.

  - ***Checked Exceptions –*** Exceptions in this category are subclasses of Exception but not subclasses of RuntimeException. Examples include FileNotFoundException, IOException, SQLException. The idea behind checked exceptions is that it should be possible to handle them in such a way that the application can continue; for example, if a database is unavailable, a SQLException would be thrown, and the user could be given the option to continue on to other features of the application even if the database is down for awhile.



"Unchecked Exceptions"

Four Ways to Deal with Checked Exceptions

1. Declare that your method throws the same kind of exception (and do not handle the exception)
2. Put the exception-creating code in a try block, and write a catch block to handle the exception in case it is thrown – in other words, use try/catch blocks.
3. Use try/catch blocks – catch block can log information about the current state – and then re-throw the exception. In this case, as in (1), you must declare that the method throws this type of exception
4. Use try/catch blocks as in (3), but, when an exception is caught, wrap it in a new instance of another type of exception class and re-throw

- See Demo lesson10.lecture.checkedexceptions

Overview

- Test-Driven Development and Unit Testing (Optional Module #1)
- Unit-testing Stream Pipelines
- Introduction to Annotations
- Review of Exception Handling in Java (Optional Module #2)
- Handling Exceptions Using Java 8's try-with-resources
- Review of Basic JDBC Programming (Optional Module #3)
- Advanced JDBC Programming
- Handling Exceptions Arising in Stream Pipelines
- Concurrent Processing and Parallel Streams

Using Java 8's try-with-resources Construct

- We return to the example lesson10.lecture.trickycatch1:

```
public void handleFile(File f) throws IOException {
    FileReader fileReader = null;
    BufferedReader buf = null;
    try{
        fileReader = new FileReader(f);                    // Open a resource
        buf = new BufferedReader(fileReader);
        String line = buf.readLine();
        System.out.println("Line from file: " + line);
    } catch(IOException e) {
        log.config("Caught 1st IOException: " + e.getMessage());
        throw(e);
    } finally {
        try {                                              // close the  resource
            if(buf != null) buf.close();
            if(fileReader != null) fileReader.close();
            throw new IOException("Caught 2nd IOException: error closing readers");
        } catch(IOException e) {
            log.fine(e.getMessage());
            throw(e);
        }
    }
}
```

- **Exercise.** We recall this scenario: A calling class for this code successfully opens the FileReader, but an IOException is thrown when readLine() is called. When the finally clause executes, another IOException is thrown when an attempt is made to close the Readers. The calling class catches an IOException and prints the message to the console.

  However, only one exception can be thrown, and in this scenario, the message displayed will be "Caught 2nd IOException: error closing reader." The IOException indicating a failure to execute readLine() is lost.

  See demo lesson10.lecture.trickycatch1.MyClass2
- This failing in pre-Java 8 has been corrected (and the entire try/catch construct greatly simplified) by the introduction of *try-with-resources.*

The try-with-resources Solution

- Here is the Java 8 alternative to the previously displayed code:

```
public void handleFile(File f) {
    //try with resources construct
    try (BufferedReader buf = new BufferedReader(new FileReader(f))) {
        String line = buf.readLine();
        System.out.println(line);
    } catch(IOException e) {
        log.warning("Main exception: " + e.getMessage());
        List<Throwable> suppressed = Arrays.asList(e.getSuppressed());
        suppressed.forEach(except -> log.warning("Suppressed message: "
                                                 + except.getMessage()));
    }

}
```

- Demo: lesson10.lecture.trickycatch4_trywithres.MyClass

  The resources named in the arguments to try – in this case, a BufferedReader – will be closed at the completion of the try block. If an exception is thrown during execution of try and an error occurs in closing these resources, the close exceptions will automatically be appended to the main exception as *suppressed exceptions*. If no such main exception occurs, but a close exception occurs, the close exception is thrown in the usual way. Notice how it is now possible to read the suppressed exceptions using the getSuppressed() method, which returns a list of Throwables. These can be read and handled if desired, but the main exception is the one that is thrown and caught.

- A *resource* is an object that must be closed after the program is finished with it. The try-with-resources statement ensures that each resource is closed at the end of the statement.

Resources in Java 8 Which Can Be Used with try-with-resources

- Any object that implements java.lang.AutoCloseable, which includes all objects that implement java.io.Closeable, can be used as a resource. The classes in Java 8 that implement AutoCloseable are listed here:

AbstractInterruptibleChannel, AbstractSelectableChannel, AbstractSelector, AsynchronousFileChannel, AsynchronousServerSocketChannel, AsynchronousSocketChannel, AudioInputStream, BufferedInputStream, BufferedOutputStream, BufferedReader, BufferedWriter, ByteArrayInputStream, ByteArrayOutputStream, CharArrayReader, CharArrayWriter, CheckedInputStream, CheckedOutputStream, CipherInputStream, CipherOutputStream, DatagramChannel, DatagramSocket, DataInputStream, DataOutputStream, DeflaterInputStream, DeflaterOutputStream, DigestInputStream, DigestOutputStream, FileCacheImageInputStream, FileCacheImageOutputStream, FileChannel, FileImageInputStream, FileImageOutputStream, FileInputStream, FileLock, FileOutputStream, FileReader, FileSystem, FileWriter, FilterInputStream, FilterOutputStream, FilterReader, FilterWriter, Formatter, ForwardingJavaFileManager, GZIPInputStream, GZIPOutputStream, ImageInputStreamImpl, ImageOutputStreamImpl, InflaterInputStream, InflaterOutputStream, InputStream, InputStream, InputStream, InputStreamReader, JarFile, JarInputStream, JarOutputStream, LineNumberInputStream, LineNumberReader, LogStream, MemoryCacheImageInputStream, MemoryCacheImageOutputStream, MLet, MulticastSocket, ObjectInputStream, ObjectOutputStream, OutputStream, OutputStream, OutputStream, OutputStreamWriter, Pipe.SinkChannel, Pipe.SourceChannel, PipedInputStream, PipedOutputStream, PipedReader, PipedWriter, PrintStream, PrintWriter, PrivateMLet, ProgressMonitorInputStream, PushbackInputStream, PushbackReader, RandomAccessFile, Reader, RMIConnectionImpl, RMIConnectionImpl_Stub, RMIConnector, RMIIIOPServerImpl, RMIJRMPServerImpl, RMIServerImpl, Scanner, SelectableChannel, Selector, SequenceInputStream, ServerSocket, ServerSocketChannel, Socket, SocketChannel, SSLServerSocket, SSLSocket, StringBufferInputStream, StringReader, StringWriter, URLClassLoader, Writer, XMLDecoder, XMLEncoder, ZipFile, ZipInputStream, ZipOutputStream

Review of JDBC
(Optional Module #3)

- Another example of a resource that can be managed with try-with-resources is the Connection object, invoked in interacting with a database, using JDBC. Handling exceptions and closing the connection in the right way and in the right sequence has tended to be error-prone. Using try-with-resources, it is straightforward to write code in the correct way.
- ***Review of JDBC.*** JDBC is a mechanism that makes it possible for a Java program to communicate with a database system (and other similar data sources) by way of SQL (structured query language) commands.

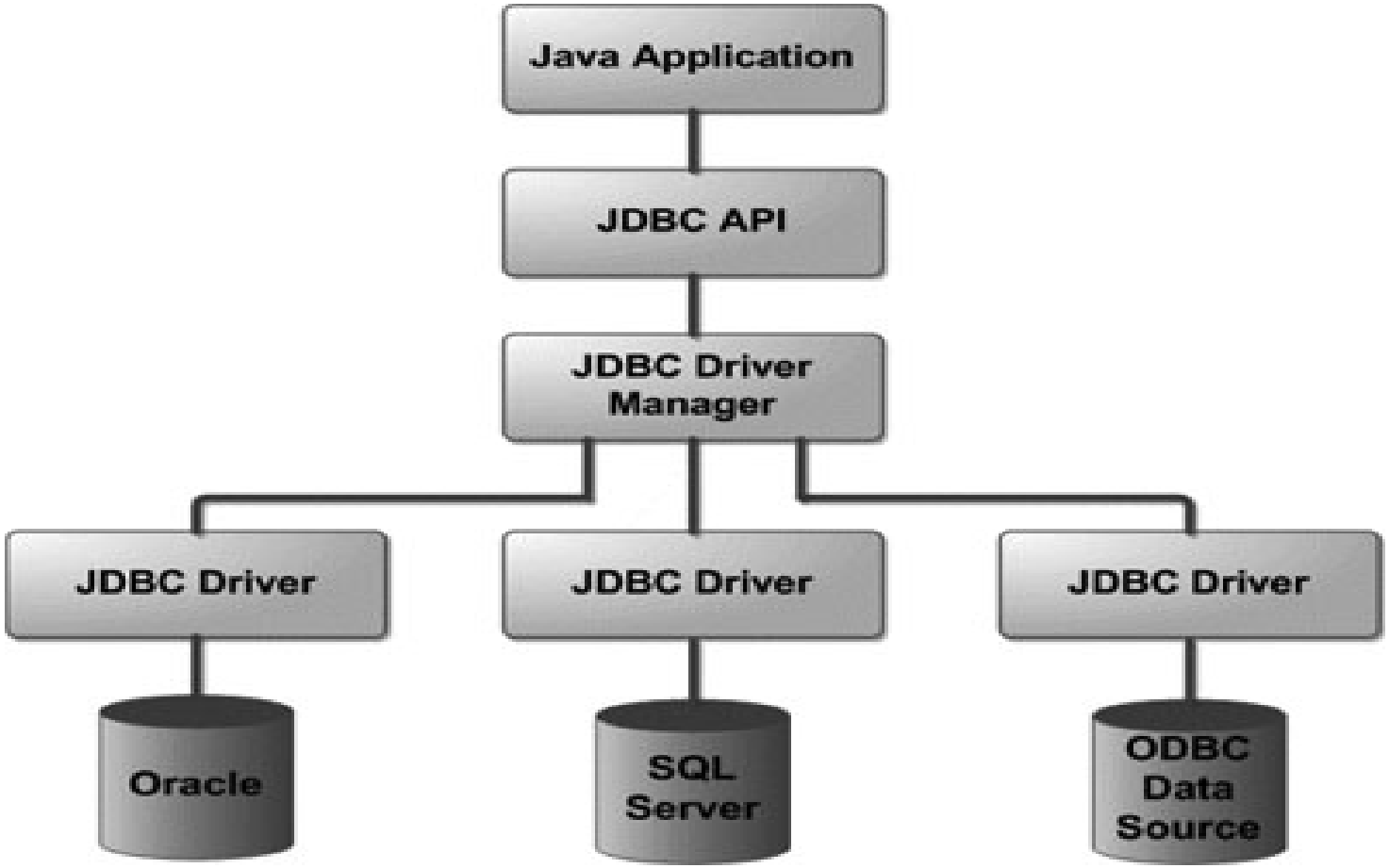
- JDBC APIs facilitate
  - Making a connection to a database.
  - Creating SQL or MySQL statements.
  - Executing SQL or MySQL queries in the database.
  - Viewing and modifying the resulting records.

- Simplified code sample (from Oracle tutorial)

```
public void connectToAndQueryDatabase(String username, String password) {

    Connection con = DriverManager.getConnection(
                     "jdbc:myDriver:myDatabase",
                     username,
                     password);

    Statement stmt = con.createStatement();
    ResultSet rs = stmt.executeQuery("SELECT a, b, c FROM Table1");

    while (rs.next()) {
        int x = rs.getInt("a");
        String s = rs.getString("b");
        float f = rs.getFloat("c");
    }
}
```

The JDBC Architecture

- **JDBC API:** This provides the application-to-JDBC Manager connection.
- **JDBC Driver API:** This supports the JDBC Manager-to-Driver Connection.



Overview

- Test-Driven Development and Unit Testing (Optional Module #1)
- Unit-testing Stream Pipelines
- Introduction to Annotations
- Review of Exception Handling in Java (Optional Module #2)
- Handling Exceptions Using Java 8's try-with-resources
- Review of Basic JDBC Programming (Optional Module #3)
- Advanced JDBC Programming (Optional Module #3)
- Handling Exceptions Arising in Stream Pipelines
- Concurrent Processing and Parallel Streams

Advanced JDBC Programming

- Begin with the example given in lesson10.lecture.jdbc.read_trywithres

- As in the case of closing a Reader, if a SQLException is thrown in reading the database and another is thrown in attempting to close the Connection, the close exception is appended to the database exception as a suppressed exception.

- The implementation style uses try-with-resources just to manage the Connection object; the steps of forming and executing a Statement are handled in an embedded try/catch block.

Handling Transactions and Auto-Generated Keys with JDBC

- Transactions enable you to control if, and when, changes are applied to the database. It treats a single SQL statement or a group of SQL statements as one logical unit, and if any statement fails, the whole transaction fails.
- To transaction support set the Connection object's autoCommit flag to false**.** For example, if you have a Connection object named conn, code the following to turn off auto-commit –

  ```
  conn.setAutoCommit(false);
  ```

- Once you have executed your SQL code (for insertions, deletions,etc), you *commit* the changes with a call to the Connection object's **commit()** method like this:

  ```
  conn.commit( );
  ```

- If an exception is thrown, you can rollback your changes to the database with a call to **rollback**:

  ```
  conn.rollback( );
  ```

- Exception-handling for transaction management can be tricky, but try-with-resources simiplifies the steps.
- DEMO: lesson10.lecture.jdbc.transact

- NOTES:
  - The demo illustrates the use of PreparedStatements. In a nutshell (for security reasons) whenever an SQL statement has parameters that need to be filled at runtime, the statement should be written using a PreparedStatement.
  - The Customer unique key field id is *auto-generated*. After you do an insert, you will often want to know what the value of the id that was generated by the database. The demo illustrates the technique for retrieving this value

RESOURCES:

- JDBC and working with transactions are big topics. Here are two excellent follow-up resources:

  http://docs.oracle.com/javase/tutorial/jdbc/basics/transactions.html

  http://www.tutorialspoint.com/jdbc/index.htm

  For practice with SQL, try:

  http://www.w3schools.com/sql/sql_intro.asp

Data Access and Frameworks

- 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.

- Typically, you gain access to a framework solution simply by adding one or more jar files to your project.

- *Two Approaches.* There are two styles of framework support these days
  - *ORM* (object-relational mapping) – JPA and Hibernate use this approach
  - *DAOs* (data access objects) – Spring supports this approach with its JDBC templating

JPA sample.

- Classes that need to be persisted (like Address, Customer, etc) are called *entities*
- In JPA, you insert annotations in an entity class to tell the framework information about reading and writing its data.
- 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.

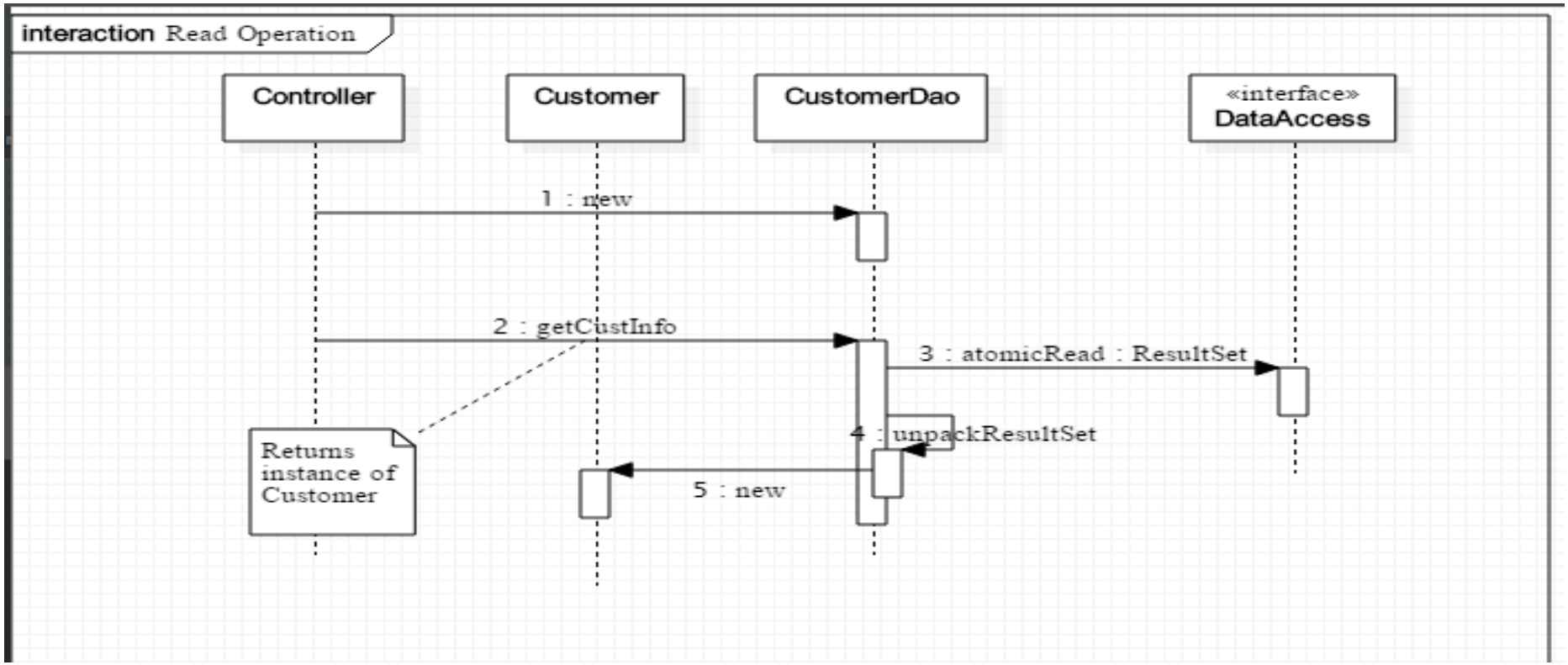
Persisting/Removing

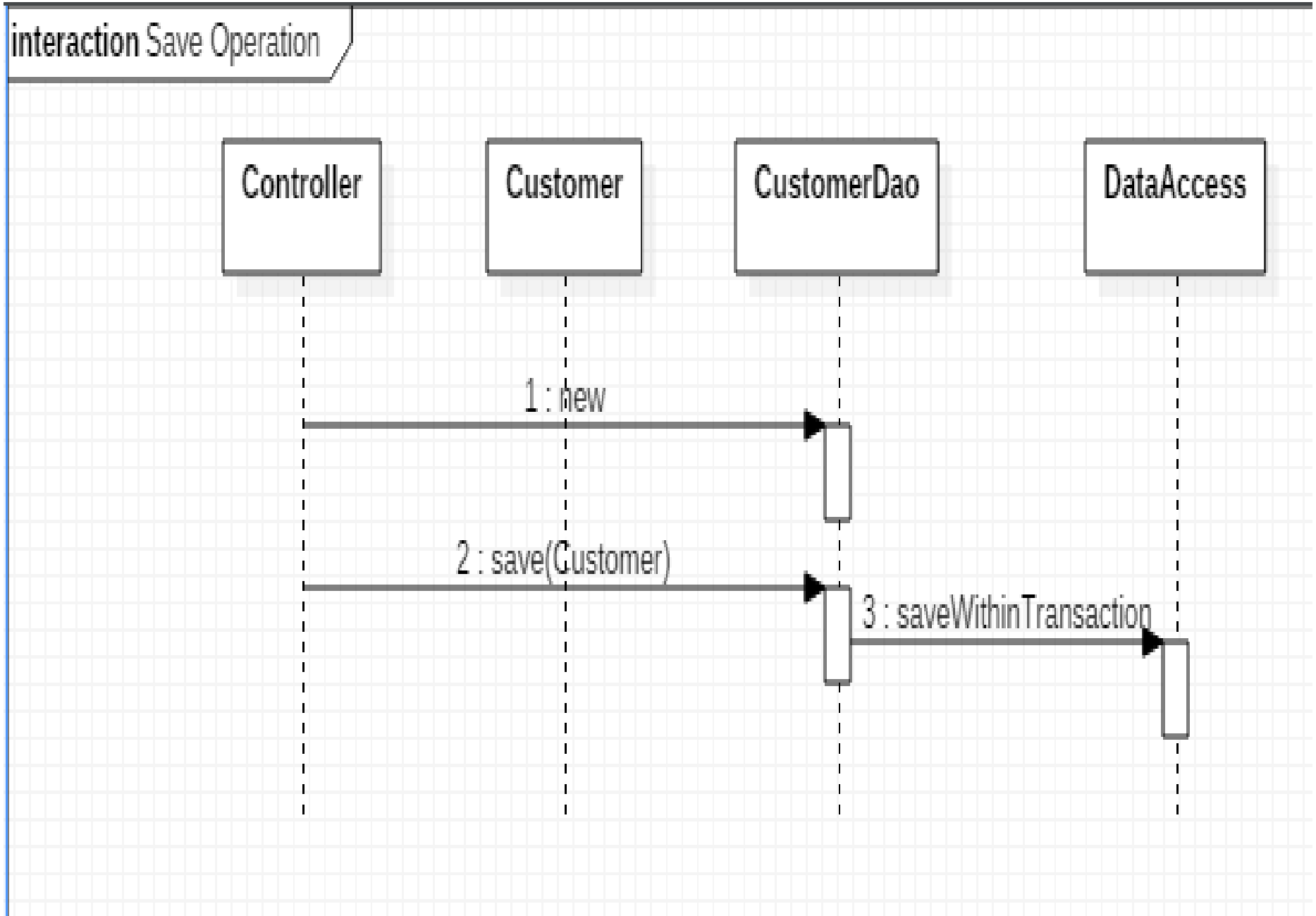
```
EntityManager em = ...;

em.persist(entity1);

em.remove(entity2);
```

Reading

```
Query query = em.createQuery(
   "SELECT c FROM Credentials c WHERE c.username = :username")
   .setParameter("username", name);
@SuppressWarnings("unchecked")
List<Credentials> result = query.getResultList();
```

- 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




interaction Save Operation
Controller
Customer
CustomerDao
DataAccess
1 : new
2 : save(Customer)
3 : saveWithinTransaction

Overview

- Test-Driven Development and Unit Testing (Optional Module #1)
- Unit-testing Stream Pipelines
- Introduction to Annotations
- Review of Exception Handling in Java (Optional Module #2)
- Handling Exceptions Using Java 8's try-with-resources
- Review of Basic JDBC Programming (Optional Module #3)
- Advanced JDBC Programming
- Handling Exceptions Arising in Stream Pipelines
- Concurrent Processing and Parallel Streams

Handling Exceptions Arising in Stream Pipelines

- However, stream operations, like map and filter, that require a functional interface whose unique method *does not have a throws clause*, make exception-handling more difficult. See demo code to see issues and best possible solutions. lesson10.lecture.exceptions2

- The best one can do in these situations is to convert checked exceptions to RuntimeExceptions. Because Java 8 streams does not support checked exceptions.
- For the streams the exception code can be written in an auxiliary method. It made more readable and compact if the try/catch clause that is needed.

Problem 1 : Exception not handled for stream operations, throw exception at runtime

```
public class TheProblem {

public List<String> getCanonicalPaths(String[] dirs) throws IOException {
//return Stream.of(dirs)
// .map(path ->
// new File(path).getCanonicalPath()) //UNHANDLED IOEXCEPTION
//.collect(Collectors.toList());
return new ArrayList<String>();
}

public static void main(String[] args) {
String[] localDirs = {"//usr", ".temp", "/etc"};
try {
List<String> canonicalPaths
= (new TheProblem()).getCanonicalPaths(localDirs);
System.out.println(canonicalPaths);
} catch(IOException e) {
e.printStackTrace();
}
}
```

Solution 1 : Handle inside Lambdas

```
public List<String> getCanonicalPaths(String[] dirs) {
return Stream.of(dirs).map(path -> {
try {
return new File(path).getCanonicalPath();
} catch (IOException ex) {
System.out.println("GOT IOException");
return ex.getMessage(); //inserts Exception message into return list
}
}).collect(Collectors.toList());
}
```

Solution 2 : Write the Exception in separate method – Easier Solution

```
public List<String> getCanonicalPaths(String[] dirs) {
return Stream.of(dirs).map(
path -> uncheckedGetCanonicalPath(path))
.map(Object::toString).collect(Collectors.toList());
}

public static String uncheckedGetCanonicalPath(String path) {
try {
return new File(path).getCanonicalPath();
} catch(IOException e) {
throw new RuntimeException(e);
}
```

Solution 3 : Using Functional Interface

```
@FunctionalInterface
public interface FunctionWithException<T, R> {
R apply(T t) throws Exception;
}

public List<String> getCanonicalPaths(String[] dirs)  {
return Stream.of(dirs).map(
path -> unchecked((String p) -> new File(p).getCanonicalPath()).apply(path))
.map(Object::toString).collect(Collectors.toList());
}

public static <T, R> Function<T,R> unchecked(FunctionWithException<T,R> f) {
return x -> {
try {
return f.apply(x);
} catch(Exception e) {
throw new RuntimeException(e);
}
};
```

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

- Test-Driven Development and Unit Testing (Optional Module #1)
- Unit-testing Stream Pipelines
- Introduction to Annotations
- Review of Exception Handling in Java (Optional Module #2)
- Handling Exceptions Using Java 8's try-with-resources
- Review of Basic JDBC Programming (Optional Module #3)
- Advanced JDBC Programming
- Handling Exceptions Arising in Stream Pipelines
- Concurrent Processing and Parallel Streams

Concurrent Processing and Parallel Streams Overview

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

Introduction to Threads

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

Creating Thread in Java - Example

```
public class ThreadTest {
public static void main(String[] args) {
Runnable r1 = new Runnable() { // Anonymous Implementation
 @Override public void run() {
System.out.println("Old Java Way"); }
 };
Runnable r2 = () -> {
System.out.println("New Java Way");
};
new Thread(r1).start();
new Thread(r2).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

- 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).
- Code is said to be *thread-safe* if it is guaranteed to be free of race conditions when accessed by multiple threads simultaneously.
- We can say therefore that this Singleton implementation is *not thread-safe*.

Forcing Serialized Access with synchronized

- 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.
- 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.

- *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++;
    }
}
```

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.

```
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

- 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.
- Some Guidelines
  - 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.
  - Operations that depend on the order of elements in the underlying collection should not be done in parallel. Example: findFirst, limit.
  - 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.
  - 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.
  - Certain data structures can be divided up more efficiently than others – ArrayList, HashSet and TreeSet can be partitioned efficiently, but LinkedLists cannot.
  - 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();
}
```

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.

Demo : lesson10.lecture.paralellstream;

Connecting the Parts of Knowledge With the Wholeness of Knowledge

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.**