Lesson 10
Unit-Testing and Exception Handling

Progress through Purification of the Path

Wholeness of the Lesson

Test-driven development (TDD) combines traditional coding with unit-testing to ensure that the code that is written is, at the same time, reliable. By testing code as it is written, the need for correcting mistakes later is greatly reduced. When mistakes do arise in Java code, the debugger can be used to track them down. When the code has been written, errors of various kinds may still arise, and to handle these, a robust exception-handling strategy is required. Using the Java SE 8 try-with-resources construct makes it possible to code exception-handling scenarios that were tricky to handle before Java 8.

Overview

1. The “Test-Driven Development” Approach to Software Development
2. JUnit and Annotations
3. Debugging Your Code
4. Handling Exceptions in Java
5. Tricky Exception-Handling Situations
    a. Introducing Try-With-Resources Construct
    b. Exceptions Using JDBC and Try-With-Resources
    c. Exceptions Thrown in a Stream Pipeline

The Test-Driven Development (TDD) Paradigm

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.

2. Robert Martin ("Professionalism and Test-Driven Development", IEEE Software, May/June 2007) describes a three-pronged TDD discipline:

   a. You may not write production code unless you've first written a failing unit test
   b. You may not write more of a unit test than is sufficient to fail
   c. You may not write more production code than is sufficient to make the failing unit test pass

3. Benefits of TDD

   a. Safe environment for code cleanup.
   b. Documentation.
   c. Minimal debugging.

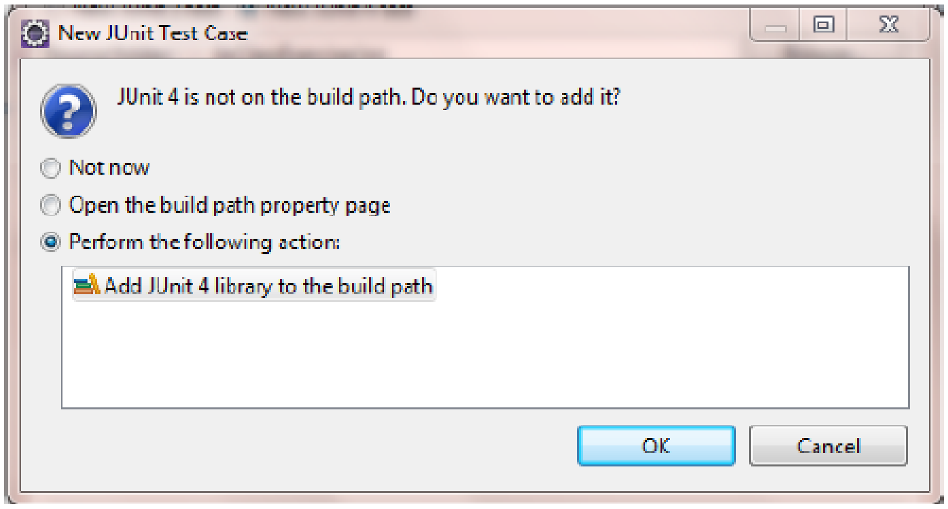
4. Example (see package lesson10.lecture.tdd)

Using Unit-Testing Tools for TDD

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 no-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

Setting Up JUnit 4.0

- 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.


New JUnit Test Case

JUnit 4 is not on the build path. Do you want to add it?

- Not now
- Open the build path property page
- [✓ Perform the following action:]

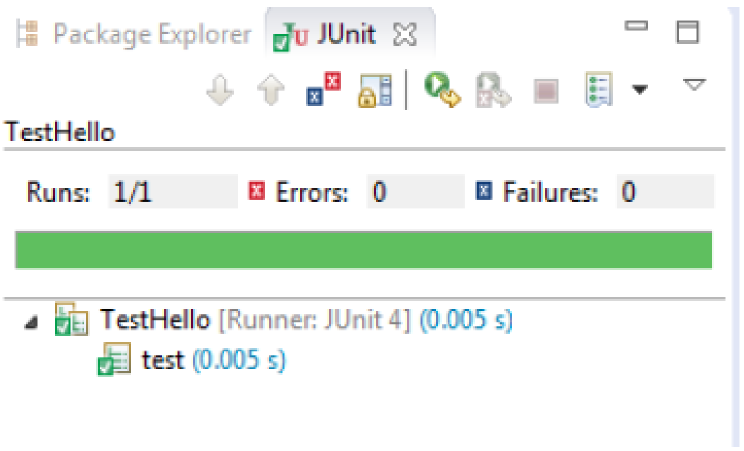
Add JUnit 4 library to the build path

- 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. ☺



```
TestHello
Runs: 1/1    Errors: 0    Failures: 0
TestHello [Runner: JUnit 4] (0.005 s)
    test (0.005 s)
```

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

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

  In each case, when test fails, comment is displayed (so it should be used to say what the expected value was).

Demo: TDD Example Using JUnit 4.0 – see lesson10.lecture.tddjunit

How to Unit Test with Lambdas?

1. 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.

2. Two Guidelines:
    a. If the pipeline is simple enough, it can wrapped in a method call, and it is enough to unit test the method call.
    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 Lambdas:

Simple Expressions

- Example: Test the expression:

```
words.stream().map(word -> word.toUpperCase()).collect(Collectors.toList());
```

- It is enough to execute this in the body of an ordinary method, and then test the method:

```
public static List<String> allToUpperCase(List<String> words) {
   return words.stream().map(word -> word.toUpperCase())
                       .collect(Collectors.toList());
}
```

- Can now do ordinary unit testing of the `allToUpperCase` method.

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

Unit-Testing Lambdas:

Complex Expressions

- Example:

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

- It is possible to test the method itself, but it requires testing lists. The key point to test, though, is whether the expression for transforming a word so that its first letter becomes upper case is working.

- This can be done by replacing the lambda expression with a method reference together with an auxiliary method:

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

//auxiliary method, used in method reference
public static String firstToUpper(String value) {
   char firstChar = Character.toUpperCase(value.charAt(0));
   return firstChar + value.substring(1);
}
```

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

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

Unit-Testing Lambdas:

Complex Expressions

The example suggests two best practices for unit testing when lambdas are involved:

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

Main Point 1

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

What Are Annotations?

1. We have seen them in various contexts already:
    a. @Test - JUnit 4
    b. @Override – to indicate (with compiler check) 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

2. Annotations are tags that you insert into your 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, 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 qualifilers like `public` and `static` may be used

5. Annotations may have 0 or more *elements.* Here is an example of a user-defined annotation that has two elements, `assignedTo` and `severity`.

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

   When an annotation has just one element and its name is “value”, the following more compact form can be used:

   ```
   @SuppressWarnings("unchecked")
   ```

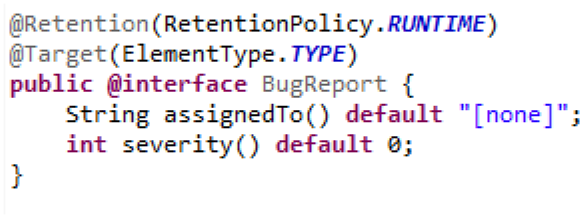
   [same as `@SuppressWarnings(value = "unchecked")`]

6. 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

7. User-defined annotations are possible. 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.

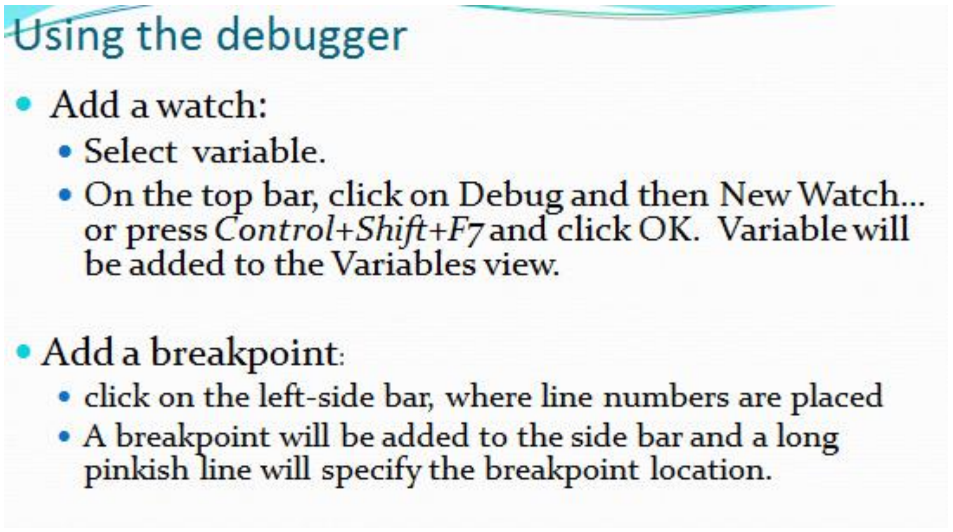
@Target `@Target` annotation marks another annotation to restrict what kind of Java elements the annotation can be applied to. A target annotation specifies one of the following element types as its value:

- `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.

Debugging Your Code

One purpose of test-driven development is to minimize the time spent debugging. Working with unit tests to get code right from the beginning has proven to be a more cost effective way of arriving at bug-free code.

Nevertheless, debugging will never go away, so it is important to know tools and techniques.

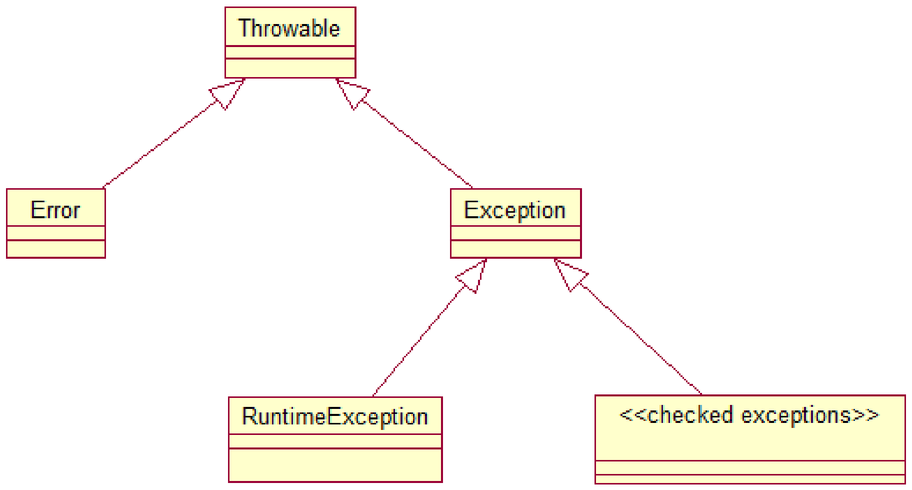


Using the debugger

- Add a watch:
  - Select variable.
  - On the top bar, click on Debug and then New Watch… or press *Control+Shift+F7* and click OK. Variable will be added to the Variables view.
- Add a breakpoint:
  - click on the left-side bar, where line numbers are placed
  - A breakpoint will be added to the side bar and a long pinkish line will specify the breakpoint location.

Exception-Handling in Java

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



The Hierarchy of Exception Classes

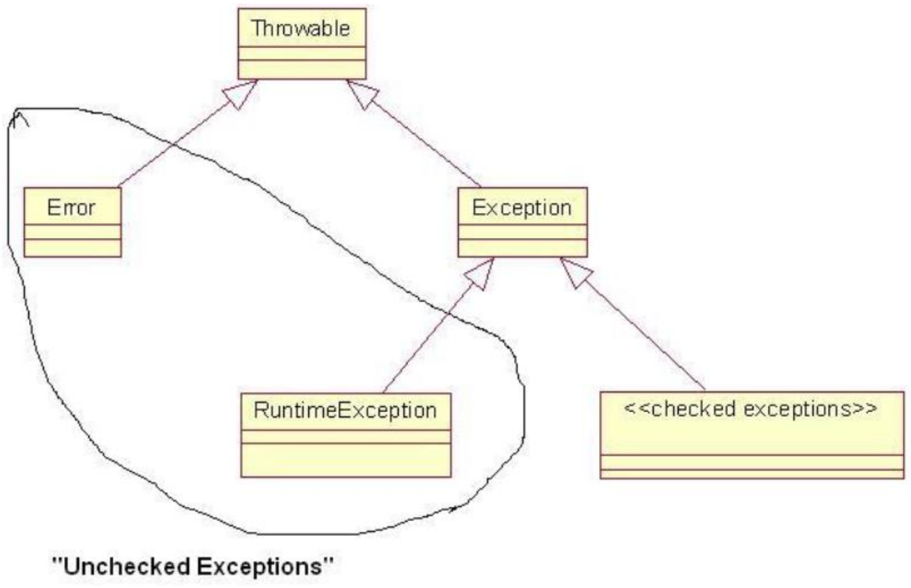
Classification of Error-Condition Classes

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.



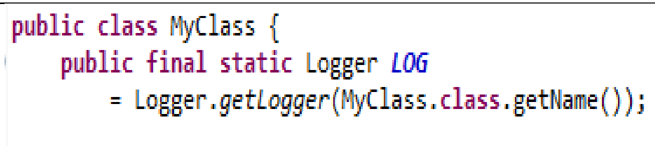
Four Ways to Deal with Checked Exceptions

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`

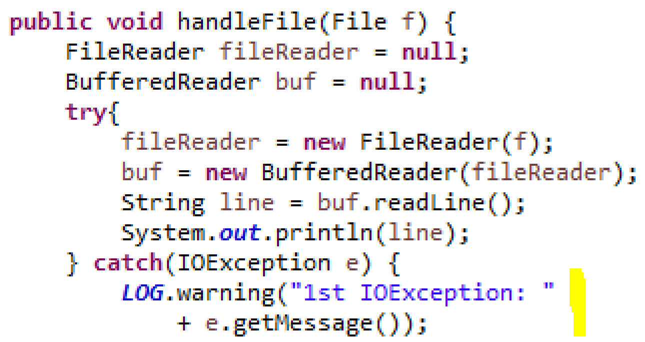
See Demo lesson10.lecture.checkedexceptions

Summary of Best Practices

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.



```
public class MyClass {
    public final static Logger LOG
        = Logger.getLogger(MyClass.class.getName());
```



```
public void handleFile(File f) {
    FileReader fileReader = null;
    BufferedReader buf = null;
    try{
        fileReader = new FileReader(f);
        buf = new BufferedReader(fileReader);
        String line = buf.readLine();
        System.out.println(line);
    } catch(IOException e) {
        LOG.warning("1st IOException: "
            + e.getMessage());
```

Obtain instance of jdk Logger

Write to the LOG. Use LOG.severe, LOG.warning, LOG.info, LOG.fine

2. *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.

3. *Never create an "empty" catch block*. Exceptions should never be ignored, as in

```
try { . . .
} catch {}
```

⇐ This is *so* bad…

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 { . . .
} catch {
   //nothing needed here
   LOG.info("Exception thrown by . . .");
}
```

⇐ This is acceptable

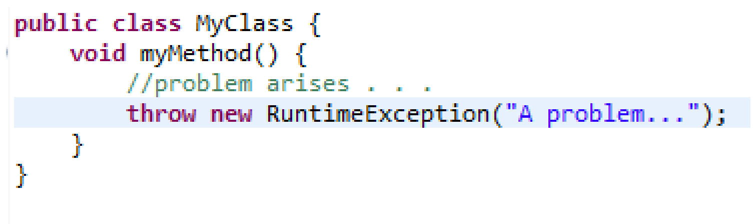
4. *Never catch Exception or Throwable*. Your code should (almost) never catch `Exception` or `Throwable`. One reason is that doing so means that you will be handling any `RuntimeException`s that are thrown (like `NullPointerException`), and these should not be caught. [One exception to this rule arises 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.]

5. *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
}
```

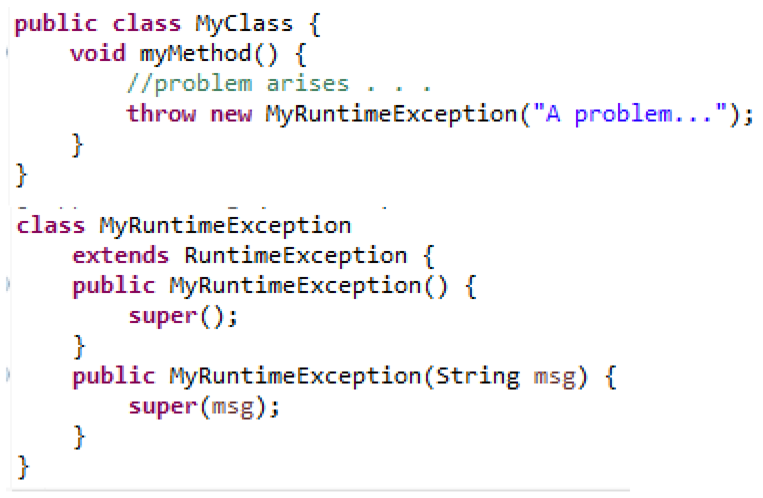
   Note that throwing any type of `RuntimeException` never requires a `throws` declaration.

6. *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.



```
public class MyClass {
    void myMethod() {
        //problem arises . . .
        throw new RuntimeException("A problem...");
    }
}
```

⇐ This is *bad*...



```
public class MyClass {
    void myMethod() {
        //problem arises . . .
        throw new MyRuntimeException("A problem...");
    }
}
class MyRuntimeException
    extends RuntimeException {
    public MyRuntimeException() {
        super();
    }
    public MyRuntimeException(String msg) {
        super(msg);
    }
}
```

⇐ This is *good*...

7. *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.

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.

Tricky try/catch/finally Situations

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!

Try-with-Resources

The `try-with-resources` construct accomplishes the same thing as the code in `lesson10.lecture.trickycatch3_suppressed`, much more compactly. See the package `lesson10.lecture.trickycatch4_trywithres`

The resources named in the arguments to `try` 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 (just as in `lesson10.lecture.trickycatch3_suppressed`). If no such main exception occurs, but a close exception occurs, the close exception is thrown in the usual way.

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. Any object that implements `java.lang.AutoCloseable`, which includes all objects which 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
```

Exception-Handling with JDBC

1. 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.

2. ***Introducing 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.

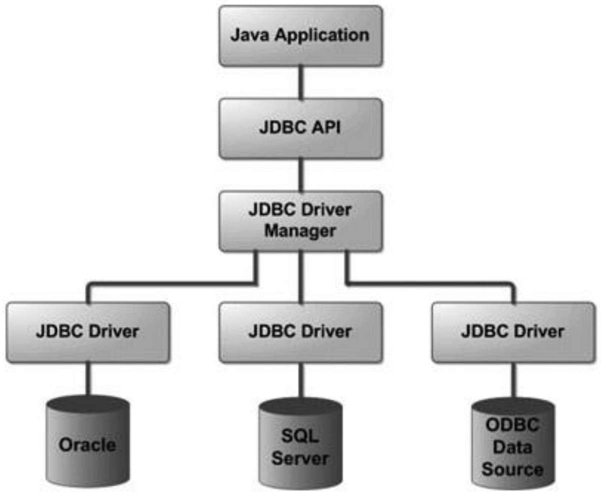
3. JDBC APIs facilitate
   - Making a connection to a database.
   - Creating SQL or MySQL statements.
   - Executing SQL or MySQL queries in the database.
   - Viewing and modifying the resulting records.

4. Over-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");
    }
}
```

5. The JDBC Architecture

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



SQL Examples

You use JDBC to transmit and execute SQL statements (for reading, inserting, updating, deleting) on a database. A pre-requisite to working with the JDBC APIs is a basic knowledge of SQL. Here are some basic examples:

1. SELECT. Select statements are used to read values from a table. Here is an example:

   ```
   SELECT custid, password
   FROM Customer
   WHERE custid = '1'
   ```

   This statement locates all records in the Customer table in which the custid column has value '1', and returns rows restricted to the two columns custid and password

2. INSERT. Insert statements are used to insert a new row into a table. Here is an example:

   ```
   INSERT INTO ShippingAddresses
   (custid,street,city,state,zip)
   VALUES('1','B St','Fairfield','IA','52556')
   ```

   This statement inserts a new record into ShippingAddresses table, placing values 1, B St, Fairfield, IA, and 52556 in the custid, street, city , state, and zip fields, respectively.

3. DELETE. Delete statements remove one or more rows in a table. Here is an example.

   ```
   DELETE FROM Company
   WHERE comp_name = 'Pepsi'
   ```

   This statement deletes from the Company table all rows in which the comp_name field is equal to 'Pepsi'.

4. UPDATE. Updates are used to change values in already-existing rows in a table. Here is an example.

   ```
   UPDATE Employees
   SET salary = salary * 1.10
   WHERE dep_code = 32
   ```

   This statement changes the value in the salary field to 1.10 times the original value, for all records in which the dep_code equals 32.

JDBC Example and Proper Exception-Handling

1. Begin with the example given in `lesson10.lecture.jdbc.read_trywithres`

2. 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.

3. 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.

Another JDBC Example:

Handling Transactions and Auto-Generated Keys

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:

1. 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.

2. 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:

I. http://docs.oracle.com/javase/tutorial/jdbc/basics/transactions.html
II. http://www.tutorialspoint.com/jdbc/index.htm

For practice with SQL, try:

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

More Tricky try/catch Situations:

Exceptions in a Lambda/Stream Pipeline

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 `lesson4lecture.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. `lesson4lecture.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. Examples are provided in `lesson4lecture.exceptions3`, `lesson4lecture.exceptions.connectold`, and `lesson4lecture.exceptions.connectnew`

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.

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.