JUnit API (cont'd)

Key API in JUnit: `Assertions`

- `org.junit.jupiter.api.Assertions`
  - Contains a series of *static* assertion methods.
  - `assertTrue( boolean condition )`
  - `assertFalse( boolean condition )`
    - » Returns if `condition` is true/false.
      - »
```
Calculator cut = new Calculator();
assertTrue( cut.multiply(3, 4) > 0 );
assertTrue( exception instanceof illegalArgumentException);
```
    - » Throws an `org.opentest4j.AssertionFailedError` if `condition` is not equal to the expected boolean state.
      - » JUnit catches it; your test method (test case) doesn't have to.
  - JUnit judges that a test method (test case) passes if it normally returns without `AssertionFailedError`.

- `assertEquals( int expected, int actual )`
- `assertEquals( float expected, float actual )`
- `...`
- `assertEquals( Object expected, Object actual )`
  - » Defined for each primitive type and `Object`
  - » Returns if two values (expected and actual results) match.
    - »
```
float expected = 12;
float actual = cut.multiply(3,4);
assertEquals( expected, actual );
```
  - » Throws an `org.opentest4j.AssertionFailedError` if two values do not match.
    - » JUnit catches it; your test method (test case) doesn't have to.
- JUnit judges that a test method (test case) passes if it normally returns without `AssertionFailedError`

- Assertion methods perform *auto-boxing* and *auto-unboxing* wherever necessary and possible.
  - `assertEquals( int expected, int actual )`
  - expected: int, actual: int
    - »
```
int expected = 10;
int actual = 20;
assertEquals( expected, actual );
```
  - expected: int, actual: Integer
    - »
```
int expected = 10;
Integer actual = 20;
assertEquals( expected, actual );
```
  - expected: Integer, actual: Integer
    - »
```
Integer expected = 10;
Integer actual = 20;
assertEquals( expected, actual );
```

Just in case…
Auto-boxing and Auto-unboxing

- Automatic conversion between a primitive type value and a wrapper class instance
  - `int numInt = 10;`
    `Integer numInteger = numInt;`
    - No need to write…
    - ```
      int numInt = 10;
      Integer numInteger = new Integer(numInt);
      // OR
      Integer numInteger = Integer.valueOf(numInt);
      ```
  - `Integer numInteger = Integer.valueOf(10);`
    `int numInt = numInteger;`
    - No need to write…
    - ```
      Integer numInteger = Integer.valueOf(10);
      int numInt = numInteger.intValue();
      ```

Primitive type	Wrapper class
boolean	Boolean
byte	Byte
char	Character
float	Float
int	Integer
long	Long
short	Short
double	Double

Key API in JUnit: `Assertions`

- **`org.junit.jupiter.api.Assertions`**
  - Contains a series of *static* assertion methods.
  - `assertNull( Object actual )`
  - `assertNonNull( Object actual )`
    - » Defined for `Object` (not for primitive types)
    - » Returns if a value is null (or NOT null).
      - » `Calculator actual;`
        `assertNull( actual );`
      - » `Calculator actual = new Calculator();`
        `assertNonNull( actual );`
      - » `float actual = 12;`
        `assertNonNull( actual );`
    - » Throws an `org.opentest4j.AssertionFailedError` if two values do not match.
  - JUnit judges that a test method (test case) passes if it normally returns without `AssertionFailedError`

- **`org.junit.jupiter.api.Assertions`**
  - Contains a series of *static* assertion methods.
  - `assertNull( Object actual )`
  - `assertNonNull( Object actual )`
    - » Defined for `Object` (not for primitive types)
    - » Returns if a value is null (or NOT null).
      - » `Calculator actual;`
        `assertNull( actual );`
      - » `Calculator actual = new Calculator();`
        `assertNonNull( actual );`
      - » `float actual = 12;`
        `assertNonNull( actual );`
    - » Throws an `org.opentest4j.AssertionFailedError` if two values do not match.
  - JUnit judges that a test method (test case) passes if it normally returns without `AssertionFailedError`

- `assertArrayEquals(int[] expected, int[] actual)`
  `assertArrayEquals(float[] expected, float[] actual)`
  `...`
  `assertArrayEquals(Object[] expected, Object[] actual)`
  - » Defined for each primitive type and `Object`
  - » Returns if two (expected and actual) arrays are equal (i.e., if they have equal elements in the same order.)
    - ```
      String[] s1 = {"UMass", "Boston"};
      String[] s2 = {"UMass", "Amherst"};
      assertArrayEquals(s1, s2);      // NOT PASS
      ```
  - » Throws an `org.opentest4j.AssertionFailedError` if two arrays are not equal.
- JUnit judges that a test method (test case) passes if it normally returns without `AssertionFailedError`
      ```

Positive and Negative Tests

- *Positive* tests
  - Verifying tested code runs without throwing exceptions
- *Negative* tests
  - Testing is not always about ensuring that tested code runs without errors/exceptions.
  - Sometimes need to verify that tested code throws an exception(s) as expected.

Positive Tests

```
@Test
public void writeToTestFile(){
 Path path = Paths.get("test.txt");
 List<String> lines = ...; //some data

 try{
    Files.write(path, lines, StandardOpenOption.CREATE);
 }
 catch(IOException ex){
    fail();
 }
}
```

- When `write()` throws an `IOException`, this test method fails with `fail()`. Otherwise, the test case passes.
- Clear, logic-wise, but try-catch-finally blocks may clutter a test case.

- Alternative strategy
  - Have a test method *re-throw* an exception
    - rather than *catching* it.

```
@Test
public void writeToTestFile() throws IOException {
  Path path = Paths.get("test.txt");
  List<String> lines = ...; //some data

  Files.write(path, lines, StandardOpenOption.CREATE);
}
```

- JUnit catches an `IOException` and judges that the test method fails.
  - `write()` throws it originally, and `readFromTestFile()` re-throws it (to JUnit)

Negative Tests

- Verify that tested code throws an exception(s) as expected.
  - Understand the conditions that cause tested code to throw an exception and test those conditions in test methods
- 2 common ways
  - Write a test case with try-catch blocks
  - Use `Assertions.assertThrows()`
    - To be introduced later in this semester.

```
@Test
public void divide5By0(){
  Calculator cut = new Calculator();
  try{
    cut.divide(5, 0);
    fail("Division by zero");
  }
  catch(IllegalArgumentException ex){
    assertEquals("division by zero",
                 ex.getMessage() );
  }
}
```


Exercise: `Calculator`

- Given `Calculator`, add addition and subtraction methods.
- Given `CalculatorTest`, add extra test methods for addition and subtraction.
  - The Java source files are implemented in `edu.umb.cs680.junit5intro`
  - Change it to `edu.umb.cs680.ex1`
- Place source code files (.java files) and binary files (.class files) in different directories.
  - Follow this directory structure:
    - `<proj dir>/src/edu/umb/cs680/ex01/Calculator.java`
    - `<proj dir>/src/edu/umb/cs680/ex01/CalculatorTest.java`
    - `<proj dir>/bin/edu/umb/cs680/ex01/Calculator.class`
    - `<proj dir>/bin/edu/umb/cs680/ex01/CalculatorTest.class`
  - Your IDE has a way to set up these "src" and "bin" directories. Figure that out.
- Compile your code on your IDE
- Run test cases with JUnit on your IDE
- No need to use Ant yet. No need to turn in your solution.

Exercise: `PrimeGenerator`

- Understand how this class works.
  - It generates prime numbers in between two input numbers (`from` and `to`)

```
Class PrimeGenerator {
  protected long from, to;
  protected LinkedList<Long> primes;

  public void generatePrimes(){ ... }
  public LinkedList<Long> getPrimes(){ return primes };
  ...
```

- Client code (c.f. `main()`)

```
PrimeGenerator gen = new PrimeGenerator(1, 10);
gen.generatePrimes();
// 2, 3, 5, 7
```


- Place `PrimeGenerator` in the package `edu.umb.cs680.ex2`
  - NOT `edu.umb.cs680.junit5intro`
- Implement `PrimeGeneratorTest` in the package `edu.umb.cs680.ex2`
  - NOT `edu.umb.cs680.junit5intro`
- Place source code files and binary files in different directories.
  - `<proj dir>/src/edu/umb/cs680/ex2/PrimeGenerator.java`
  - `<proj dir>/src/edu/umb/cs680/ex2/PrimeGeneratorTest.java`
  - `<proj dir>/bin/edu/umb/cs680/ex2/PrimeGenerator.class`
  - `<proj dir>/bin/edu/umb/cs680/ex2/PrimeGeneratorTest.class`

- In your test class, **`PrimeGeneratorTest`**, write more than one test method.
  - Test a regular case (positive test)

```
PrimeGenerator gen = new PrimeGenerator(1, 10);
gen.generatePrimes();
Long[] expectedPrimes = {2L, 3L, 5L, 7L};
assertArrayEquals( expectedPrimes,
                   gen.getPrimes().toArray() );
```

  - Test error cases where wrong ranges are given (negative test)
    - e.g., [-10, 10], [100, 1]
  - You can name test methods as you like. Make sure to give them specific (not vague) names.

- Compile your code on your IDE
- Run test cases with JUnit on your IDE
- No need to use Ant.

- No need to turn in your solution.

Automated Build

Code Sharing
(Simple Team Development)

- Goal
  - Share your code with others (e.g., me) and have everyone build and run your code in the same way.

- Challenge
  - Different people use different environments (e.g., OSes, IDEs).
  - Need to make sure that everyone
    - follows the same directory and package structure,
    - imports the same set of JAR files (libraries),
    - compiles the same set of code in the same way (with the same options),
    - runs the same set of test cases, and
    - generates test reports in the same way.

- It is too time-consuming and error-prone to
  - Manually write a document that specifies the expected build configurations
    - e.g., directory/package structure, JAR files to be imported, and test cases to be used
  - Have everyone meet these expectations *manually* on their environments.

- Automated build
  - Intends to follow and satisfy those expectations *automatically* (not manually)
  - A few well-known tools: Ant, Maven, Gradle, etc.

Automated Build with Ant

- Use Ant (http://ant.apache.org/) to build all of your Java programs in every HW.
  - Turn in *.java and a build script (e.g. build.xml).
    - Turn in a ***single*** build script that
      - configures all settings (e.g., CLASSPATH, a directory of source code, a directory to generate binary code),
      - compiles all source code (*.java files) from scratch,
      - generates binary code (*.class files), and
      - runs compiled code [OPTIONAL]
    - DO **NOT** turn in byte code (class files).
    - DO **NOT** use any other ways for configurations and compilation.
      - DO NOT set up CLASSPATH and other paths manually with a GUI/IDE
      - DO NOT set up binary code directories with a GUI/IDE
      - DO NOT click the "compile/run" button manually on an IDE

- Goal: Fully automate configuration and compilation process to
  - Speed them up
  - Avoid potential human errors
  - Make it easy and clear for team mates to run your code properly.

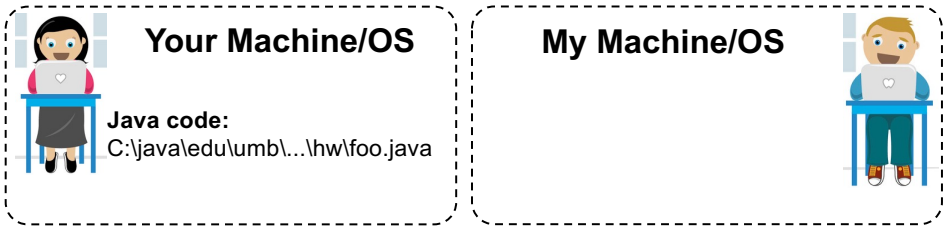
Exercise

- Place a Java source file and a build script as follows:
  - `<proj dir>/src/edu/umb/cs680/junit5intro/Calculator.java`
  - `<proj dir>/build-calc.xml`
- Run the build script (on your command-line shell) in the directory where it is located (i.e., at <proj dir>).
  - Type: **`ant -f build-calc.xml`**
- Understand how it builds and runs `Calculator`
  - Set up the directory where `Calculator.class` is placed.
    - `<proj dir>/bin/edu/umb/cs680/junit5intro`
  - Set up CLASSPATH
    - `<proj dir>/bin`
  - Compile `Calculator.java` and generate `Calculator.class` to `<proj dir>/bin/edu/.../junit5intro`
    - Use `<javac>` task
  - Run `Calculator.class`
    - Use `<java>` task to run Calculator's main()

- Make sure that the build script runs properly.
  - On both your shell and IDE
- No need to do unit testing yet.
- Change the package structure from
  - `edu.umb.cs680.junit5intro`
  - to:
  - `edu.umb.cs680.ex3`
- Revise your build script accordingly.
- Make sure your revised build script runs properly.

Important Note

- Never include absolute paths and OS/IDE-specific paths in your build script.
  - My OS would not be able to recognize them.
  - Always use relative paths.
- Keep your build script OS/IDE-independent.



Your build.xml
<project name="calculator" **basedir="C:\java"** …>

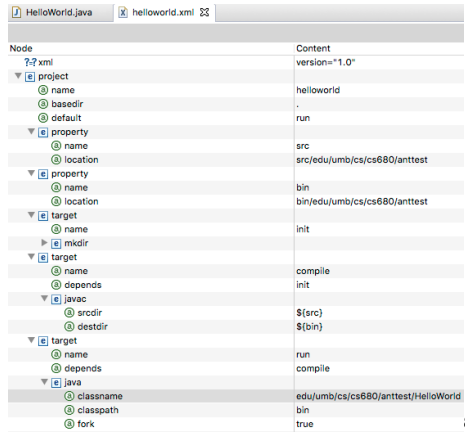
Your OS may recognize this absolute path, but mine may not.
Correct it to: **basedir="."** and place your build script at C:\java

Grading Policy

- To grade your work, I will simply run your build script with the "ant" command (on my shell) in the directory where your build script is located.
  - You can name your build script as you like.
    - No need to name it build.xml.
      - I will type: ant -f abc.xml
  - If the "ant" command fails, I will **NOT** grade your work.

Ant in IDE

- Ant is integrated in your IDE.
  - To work on HWs, you can use the integrated version of Ant.
- However, I will run your build script on a shell.
  - Make sure that your build script works on your shell.



Appendix:
NIO-based File/Path Handling and Try-with-resources Statement

(1) Dealing with File/Directory Paths in NIO

- **`java.nio.Paths`**
  - A utility class (i.e., a set of static methods) to create a path in the file system.
    - Path: A sequence of directory names
      - Optionally with a file name in the end.
  - A path can be *absolute* or *relative*.
    - `Path absolute = Paths.get("/Users/jxs/temp/test.txt");`
    - `Path relative = Paths.get("temp/test.txt");`
- **`java.nio.Path`**
  - Represents a path in the file system.
  - Given a path, *resolve* (or determine) another path.
    - `Path absolute = Paths.get("/Users/jxs/");`
      `Path another = absolute.resolve("temp/test.txt");`
    - `Path relative = Paths.get("src");`
      `Path another = relative.resolveSibling("bin");`

Just in Case: Passing a Variable # of Parameters to a Method

- **`Paths.get()`** can receive a variable number of parameter values (1 to many values)
  - c.f. Java API documentation
  - `Paths.get(String first, String... more)`
    - `Paths.get("temp/test.txt");` // relative path
    - `Paths.get("temp", "test.txt");` // relative path
    - `Paths.get("/", "Users", "jxs");` // absolute path
  - `String... More` → Can receive zero to many String values.
  - Introduced in Java 5 (JDK 1.5)

- Parameter values are handled with an array.
  - ```
    class Foo{
        public void varParamMethod(String... strings){
            for(int i = 0; i < strings.length; i++){
                System.out.println(strings[i]); } } }
    ```
  - ```
    Foo foo = new Foo();
    foo.varParamMethod("U", "M", "B");
    ```
- **`String... Strings`** is a syntactic sugar for **`String[] strings`**.
  - Your Java compiler transforms the above code to:
    - ```
      class Foo{
          public void varParamMethod(String[] Strings){
              for(int i = 0; i < strings.length; i++){
                  System.out.println(strings[i]); } } }
      ```
    - ```
      Foo foo = new Foo();
      String[] strs = {"U", "M", "B"};
      foo.varParamMethod(strs);
      ```

Reading and Writing into a File w/ NIO

- `java.nio.file.Files`
  - A utility class (i.e., a set of static methods) to process a file/directory.
  - Reading a byte sequence and a char sequence from a file

```
Path path = Paths.get("/Users/jxs/temp/test.txt");
byte[] bytes = Files.readAllBytes(path);
String content = new String(bytes);
```

```
List<String> lines = Files.readAllLines(path);
for(String line: lines){
      System.out.println(line); }
```

  - Writing into a file

```
Files.write(path, bytes);
Files.write(path, content.getBytes());
Files.write(path, bytes, StandardOpenOption.CREATE);
Files.write(path, lines);
Files.write(path, lines, StandardOpenOption.WRITE);
```

```
StandardOpenOption: CREATE, WRITE, APPEND, DELETE_ON_CLOSE, etc.
```


NIO (java.nio) v.s. Traditional I/O (java.io)

- NIO provides simpler or easier-to-use APIs.
  - Client code can be more concise and easier to understand.
- NIO:

```
Path path = Paths.get("/Users/jxs/temp/test.txt");
byte[] bytes = Files.readAllBytes(path);
String content = new String(bytes);
```

- java.io:

```
File file = ...;
FileInputStream fis = new FileInputStream(file);
int len = (int)file.length();
byte[] bytes = new byte[len];
fis.read(bytes);
fis.close();
String content = new String(bytes);
```


NIO (java.nio) v.s. Traditional I/O (java.io)

- NIO:

```
Path path = Paths.get("/Users/jxs/temp/test.txt");
List<String> lines = Files.readAllLines(path);
```

- java.io:

```
int ch=-1, i=0;
ArrayList<String> contents = new ArrayList<String>();
StringBuffer strBuff = new StringBuffer();
File file = ...;
InputStreamReader reader = new InputStreamReader(
                                new FileInputStream(file));
while( (ch=reader.read()) != -1 ){
    if( (char)ch == '\n' ){        //**line break detection
        contents.add(i, strBuff.toString());
        strBuff.delete(0, strBuff.length());
        i++;
        continue;
    }
    strBuff.append((char)ch);
}
reader.close();
```

** The perfect (platform independent) detection of a line break should be more complex.
Unix: '\n', Mac: '\r', Windows: '\r\n' c.f. BufferedReader.read()

NIO (java.nio) v.s. Traditional I/O (java.io)

- NIO:

```
Path path = Paths.get("/Users/jxs/temp/test.txt");
List<String> lines = Files.readAllLines(path);
```

- java.io (a bit simplified version):

```
int ch=-1, i=0;
ArrayList<String> contents = new ArrayList<String>();
StringBuffer strBuff = new StringBuffer();
File file = ...;
FileReader reader = new FileReader(file); //***
while( (ch=reader.read()) != -1 ){
    if( (char)ch == '\n' ){  //** Line break detection
        contents.add(i, strBuff.toString());
        strBuff.delete(0, strBuff.length());
        i++;
        continue;
    }
    strBuff.append((char)ch);
}
reader.close();
```

*** FileReader: A convenience class for reading character files.

Files in Java NIO

- readAllBytes(), readAllLines()
  - Read the whole data from a file without buffering.
- write()
  - Write a set of data to a file without buffering.
- When using a large file, it makes sense to use **`BufferedReader`** and **`BufferedWriter`** with **`Files`**.
  - ```
    Path path = Paths.get("/Users/jxs/temp/test.txt");
    BufferedReader reader = Files.newBufferedReader(path);
    while( (line=reader.readLine()) != null ){
        // do something
    }
    reader.close();
    ```
  - ```
    BufferedWriter writer = Files.newBufferedWriter(path);
    writer.write(...);
    writer.close();
    ```


Just in case: Buffering

- At the lowest level, read/write operations deal with data *byte by byte*, or *char by char*.
  - File access occurs *byte by byte*, or *char by char*.
- Inefficient if you read/write a lot of data.
- Buffering allows read/write operations to deal with data in a coarse-grained manner.
  - Chunk by chunk, not byte by byte or char by char
  - Chunk = a set of bytes or a set of chars
    - The size of a chunk: 512 bytes by default, but configurable

Getting Input/Output Streams from `Files`

- Input and output streams can be obtained from **`Files`**.
  - ```
    Path path = Paths.get("/Users/jxs/temp/test.txt");
    InputStream is = Files.newInputStream(path);
    ```
    - `is` contains an instance of ChannelInputStream, which is a subclass of InputStream.
    - Make sure to call `is.close()` in the end.
- Can decorate the input/output stream with filters.
  - ```
    ZipInputStream zis = new ZipInputStream(
                                Files.newInputStream(path) );
    ```
    - Make sure to call `zis.close()` in the end.

Never Forget to Call close()

- Need to call `close()` on each input/output stream (or its filer) in the end.
  - Must-do: Follow the *Before/After* design pattern.
    - In Java, use a *try-catch-finally* or *try-finally* statement.
      - ```
        Open a file here.
        try{
            Do something with the file here.
            Throw an exception if an error occurs.
        }catch(...){
            Error-handling code here.
        }finally{
            Close the file here.
        }
        ```
  - Note: No need to call `close()` when using **`readAllBytes()`**, **`readAllLines()`** and **`write()`** of **`Files`**.
        ```

```
Path path = Paths.get("/Users/jxs/temp/test.txt");
BufferedReader reader = Files.newBufferedReader(path);
try{
  while( (line=reader.readLine()) != null ){
      // do something
  }
}catch(IOException ex){
  ... // Error handling
}finally{
  reader.close();
}
```


(2) Try-with-resources Statement

- Allows you to skip calling close() explicitly in the finally block.
  - *Try-catch-finally*

```
Open a file here.
try{
    Do something with the file here.
}catch(...){
    Handle errors here.
}finally{
    Close the file here.
}
```

  - *Try-with-resources*

```
try ( Open a file here ){
    Do something with the file here.
}
```


- `close()` is automatically called on a resource used for reading or writing to a file, when exiting a try block.

```
try( BufferedReader reader =
        Files.newBufferedReader( Paths.get("test.txt")) ){
    while( (line=reader.readLine()) != null ){
        // do something }
 }
```

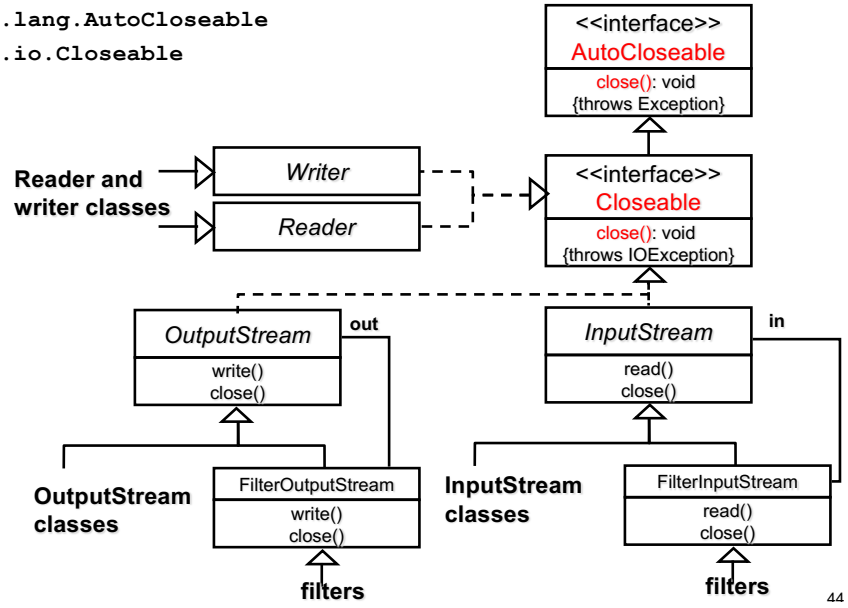
  - No explicit call of close() on `reader` in the finally block. `reader` is expected to implement the `AutoCloseable` interface.

```
try( BufferedReader reader = Files.newBufferedReader(...);
     PrintWriter writer = new PrintWriter(...) ){
    while( (line=reader.readLine()) != null ){
        // do something
        writer.println(...); }
}
```

  - Can specify multiple resources in a try block. close() is automatically called on all of them. They all need to implement `AutoCloseable`.

AutoCloseable Interface

- `java.lang.AutoCloseable`
- `java.io.Closeable`



- Recap: No need to call `close()` when using **`readAllBytes()`**, **`readAllLines()`** and **`write()`** of **`Files`**.
  - Those methods internally use the try-with-resources statement to read and write to a file.

Try-with-resources-Catch-Finally

- Catch and finally blocks can be attached to a try-with-resources statement.

```
try( BufferedReader reader =
        Files.newBufferedReader( Paths.get("test.txt")) ){
    while( (line=reader.readLine()) != null ){
        // do something. This part may throw an exception.}
}catch(...){
    //This block runs if the try block throws an exception.
}finally{
    ...
    //No need to do reader.close() here.
}
```

  - The catch and finally blocks run (if necessary) AFTER close() is called on **`reader`**.