Exercise 1

- You have received:
  - `Calculator.java`
  - `ex1.xml`
- Place them on your machine as follows:
  - <proj dir>
    - `ex1.xml`
    - src/main [source code directory]
      - umbcs680/calc/`Calculator.java`
- Run `ex1.xml` on your IDE AND your shell.
  - On your shell, run: `ant -f ex1.xml`

- Make sure to understand how the build script builds and runs `Calculator`
  - Set up the directory where `Calculator.class` is placed.
    - `<proj dir>/build/main/`
  - Set up CLASSPATH
    - `<proj dir>/build/main/`
  - Compile `Calculator.java` and generate `Calculator.class` in `<proj dir>/build/main/umbcs680/calc/` with the `<javac>` task

When Your Turn in a HW solution…

- Prepare your own build script.
- Have it fully automate all configuration and compilation steps to build your work.
  - Do not include absolute paths. Always use relative paths.

Grading Policy in CS680

- I will run your build script on a shell (not on an IDE).
  - I will use the most recent stable version of the "ant" command, which is available at http://ant.apache.org/
- If your build script fails, I will **NOT** grade your work.
  - If you want, ask me if your build script runs properly on my machine.
    - You can correct any errors if you consult with me early enough against the deadline.

Exercise 2: `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
```


- Write a build script for `PrimeGenerator.java` yourself.

- Place the files on your machine as follows:
  - <proj dir>
    - **`ex2.xml`**
    - src/main [source code directory]
      - umbcs680/prime/**`PrimeGenerator.java`**

- Run `ex2.xml` on your IDE AND your shell.
  - On your shell, run: `ant -f ex2.xml`

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`

  - `assertIterableEquals( Iterable expected, Iterable actual )`

    – Returns if two (expected and actual) iterables are deeply equal (i.e., if they have equal elements in the same order).

    ```
    List<String> l1 = List.of("UMass", "Boston");
    List<String> l2 = List.of("UMass", "Amherst");
    assertIterableEquals(l1, l2);  // NOT PASS
    ```

    » c.f. assertArrayEquals()

    » Throws an `org.opentest4j.AssertionFailedError` if two iterables are not equal.

    - JUnit catches it; your test cases don't have to.

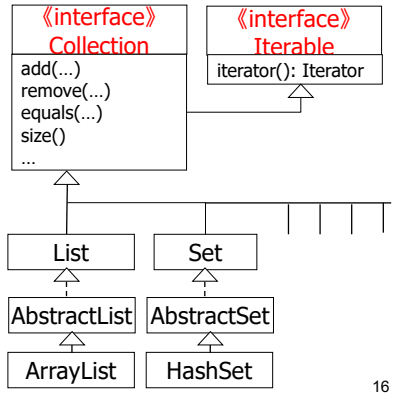
    - JUnit judges that a test method (test case) passes if it normally returns without `AssertionFailedError`

Iterable

- **`Collection`**
  – The root interface for all collection classes

- **`Iterable`**
  – The super interface of `Collection`.
  – All collection classes implements it.

- **`assertIterableEquals()`** takes any collections.



- `assertIterableEquals( Iterable expected, Iterable actual )`
  - Returns if two iterables are deeply equal.
    - They don't have to be of the same type.

```
List<String> l1 = Arrays.asList("UMass", "Boston");
List<String> l2 = Arrays.asList("UMass", "Boston");
assertIterableEquals(l1, l2);   // PASS
```

```
ArrayList<String> al = new ArrayList<>(
                                Arrays.asList("UMass", "Boston"));
LinkedList<String> ll = new LinkedList<>(
                                Arrays.asList("UMass", "Boston"));
assertIterableEquals(al, ll);   //PASS
```


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() );
  }
 }
```


Automating Unit Tests

Running Unit Tests in a Build Process

- From now, you will use BOTH Ant and JUnit.
  - You will run your Ant build script to
    - Compile your program(s),
      - e.g., `Calculator.java`
    - Compile your test class(es),
      - e.g., `CalculatorTest.java`
    - Run your test class(es) with Junit, and
      - e.g., `CalculatorTest.class`
- Your code base requires, or depends on, JUnit library files (i.e., JUnit JAR files).
  - Your build script needs to reference those JAR files and set their paths into CLASSPATH
    - so that it can compile and run your test classes.

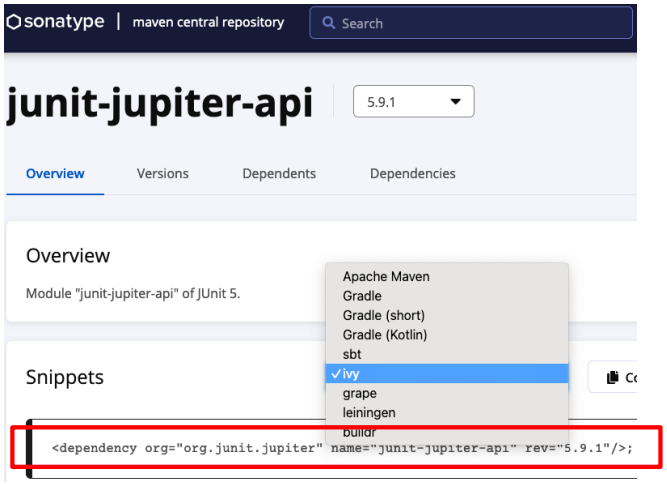
JUnit JAR Files

- JUnit API JAR files
  - `junit-jupitar-api.jar`
  - `junit-jupitar-engine.jar`
  - `junit-jupitar-params.jar`
  - `apiguardian-api.jar`
  - `opentest4j.jar`
  - DO NOT use `junit-vintage-*.jar` nor `junit.jar` (Those are for JUnit version 4.)
- JUnit Platform JAR files
  - `junit-platform-commons.jar`
  - `junit-platform-engine.jar`
  - `junit-platform.launcher.jar`
  - `junit-platform-runner.jar:`
  - `junit-platform-suite-api.jar`
- Your Ant build script needs to reference ALL these JAR files and set their paths in CLASSPATH.

How to Make JUnit JAR files Available for Your Build Script?

- Use Apache Ivy with Ant
  - https://ant.apache.org/ivy/
  - Install it in addition to Ant (c.f. Ivy's reference manual).
- Ivy does "dependency management" for Ant.
  - Ivy allows your build script to download any external library files (JAR files),
    - so the build script can set their paths into CLASSPATH.
  - ivy.xml lists the JAR files you want to download.
  - Your build script states `<ivy:retrieve/>`
    - This task will download the JAR files listed in ivy.xml into the "lib" directory by default.

- ivy.xml should specify each JAR file you want to download with `<dependency>`.
  - <dependency org="…" name="…" rev="…" />
- Search each JAR file you want at https://search.maven.org/



- Your build script:
  - `<ivy:retrieve/>`

```
<path id="classpath">
   ...
   <fileset dir="lib">
</path>

<javac …>
<classpath refid="classpath"/>
    …
</javac>

<junitlauncher …>
<classpath refid="classpath"/>
    …
</junitlauncher>
```

  - Useful task (for debugging) to print CLASSPATH
    - `<echo message="${toString:classpath}" />`

Exercise 3

- You have received:
  - `Calculator.java`
  - `CalculatorTest.java`
  - `ex3.xml`
  - `ivy.xml`
- Place them on your machine as follows:
  - <proj dir>
    - ex3.xml
    - ivy.xml
    - src/main [source code directory]
      - umbcs680/calc/Calculator.java
    - src/test [test code directory]
      - umbcs680/calc/CalculatorTest.java
- Run `ex3.xml` on your IDE AND your shell.
  - On your shell, run: `ant -f ex3.xml`

- Make sure to understand how the build script runs.
- Directory structure:
  - <proj dir>
    - `ex3.xml`
    - `ivy.xml`
    - `src/main`
    - `src/test`
    - `build/main/`
    - `build/test/`

HW 1

- Extend your Exercise 2 (ex2) work.
  - In ex2, you used Ant only. You didn't use Ivy and JUnit.
- Write a test class, `CalculatorTest.java`, to test `Calculator`. Place it under the `src/test` directory.
- Expected directory structure:
  - <HW 1 dir>
    - `build.xml`
    - `ivy.xml`
    - `src/main`
    - `src/test`
    - `build/main/`
    - `build/test/`

- Write one or more test methods for a positive test(s).

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

- Write one or more test methods for a negative test(s).
  - Verify your class throws an expected exception when wrong ranges (`from-to` pairs) are given
    - e.g., [-10, 10], [-10, -5], [100, 1]

- Prepare a build script by customizing `ex3.xml`.
  - No need to modify `ivy.xml`.

- Have your build script…
  - Download JUnit JAR files.
  - Do pre-compilation configurations (e.g., directory structures, CLASSPATH settings, etc.)
  - Compile `PrimeGenerator` and `PrimeGeneratorTest`
  - Run test cases (test methods) in `PrimeGeneratorTest`

- Make sure that your build script runs properly.
  - on both your IDE and shell.

- Turn in:
  - build script
  - ivy.xml
  - src/main
  - src/test

- DO NOT include binary files (the "lib" directory and .class files), IDE-specific files and OS-specific files.
  - Configure `.gitignore` if you like.
  - I will NOT grade your work if you turn in binary files.

- Due: March 17 (Sun), midnight
  - Firm deadlines are firm. Will NOT grade any late submissions.

- Place your HW solution at GitHub and email me an invitation to share your repo with me.
  - My GitHub account is jxsboston.
  - One repo for the entire set of HW solutions.

Important Notice

- You must work alone from scratch for each HW.
  - You can discuss HW assignments with others. However, DO NOT start with anyone else's code. You must write your code yourself.

- It is an academic crime to
  - Copy (or steal) someone else's code and submit it as your own work.
  - Allow someone else to copy (or steal) your code and submit it as his/her work.
    - Use a private repo to avoid this.

- You will end up with a serious situation if you commit this crime.
  - The University, College and Department have no mercy about it.

Suggestions

- Try your best to turn in your solutions early and regularly as HW assignments are given.
  - Rather than waiting for a deadline to turn them in.
- If you turn in your solutions early…
  - you can ask for my feedback about them, so you can revise them before a deadline.
- If you turn in your solutions regularly…
  - you can earn some extra points.

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*.
- 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`

- Can obtain an input/output stream 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();
}
```

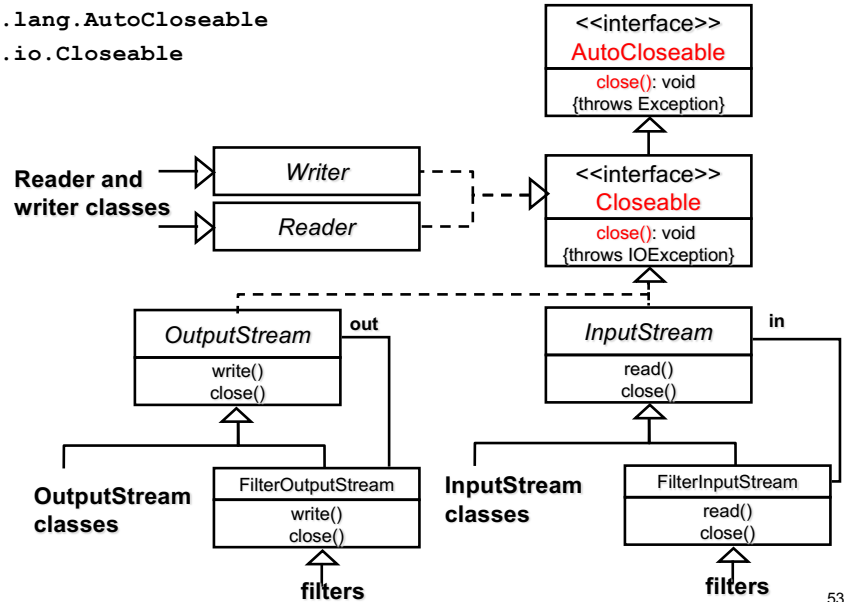

(2) Try-with-resources Statement

- 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 implicitly (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 implicitly (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.
    //Do something else if necessary.
}
```

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