CPSC 5011: Object-Oriented Concepts

Lecture 2: Classes & Methods, Defensive Programming, Exceptions, Unit Testing, Javadoc

Bloch – Item 15

Item 15: Minimize the accessibility of classes and members

- Make each class or member as inaccessible as possible
- Access levels
  - **private** : the member is accessible only from the top-level class where it is declared
  - **package-private** : the member is accessible from any class in the package where it is declared; technically know as default access, this is the access level you get if no access modifier is specified (except for interface members which are public by default)
  - **protected** : the member is accessible from subclasses of the class where it is declared (subject to a few restrictions) and from any class in the package where it is declared
  - **public** : the member is accessible from anywhere

Bloch – Item 16

Item 16: In public classes, use accessor methods, not public fields

- Public fields violates the prioritization of encapsulation
- Encapsulate data by accessors and mutators
- If a class is accessible outside its package, provide accessor methods
- If a class is package-private or is a private nested class, there is nothing inherently wrong with exposing it's data fields

Example:

```
class Point {
   public double x;
   public double y;
}
```

=>

```
class Point {
   private double x;
   private double y;
   public Point(double x, double y) {
      this.x = x;  this.y = y;
   }
   public double getX() { return x; }
   public double getY() { return y; }
   public void setX(double x) { this.x = x; }
   public void setY(double y) { this.y = y; }
}
```

Bloch – Item 17

Item 17: Minimize mutability (*for immutable classes*)

- Don't provide methods that modify the object's state
- Ensure that the class can't be extended
- Make all fields final
- Make all fields private
- Ensure exclusive access to any mutable components

Example:

```
public final class PhoneNumber {
   private final short areaCode, prefix, lineNum;
   public PhoneNumber(int areaCode, int prefix, int lineNum) { ... }
   ...
}
public final class Complex {
   private final double re, im;
   public Complex (double re, double im) { ... }
   public double getRealPart() { return re; }
   public double getImaginaryPart() { return im; }
   public Complex plus(Complex c) { ... }
   ...
}
```

Bloch – Item 22

An **interface** defines the protocol for certain behavior but does not provide an implementation

- You cannot instantiate an interface
- An interface does not contain any constructors
- All of the methods in an interface are abstract
- An interface cannot contain instance fields. The only fields that can appear in an interface must be declared both static and final
- An interface is not extended by a class; it is implemented by a class
- An interface can extend multiple interfaces

Example:

```
interface Animal {
   public void eat();
   public void travel();
}
```

Item 22: Use interfaces only to define types

- When a class implements an interface, the interface serves as a type that can be used to refer to instances of the class, and should say something about what a client can do with instances of the class.

Defensive Programming

Defensive programming is a form of defensive design intended to ensure the continuing function of a piece of software under unforeseen circumstances. Defensive programming practices are often used where high availability, safety or security is needed.

Defensive programming is an approach to improve software and source code, in terms of:

- General quality – reducing the number of software bugs and problems.
- Making the source code comprehensible – the source code should be readable and understandable so it is approved in a code audit.
- Making the software behave in a predictable manner despite unexpected inputs or user actions.

Overly defensive programming, however, may safeguard against errors that will never be encountered, thus wasting runtime and maintenance costs. There is also the risk that the code traps or prevents too many exceptions, potentially resulting in unnoticed, incorrect results. [see Wikipedia]

In summary:

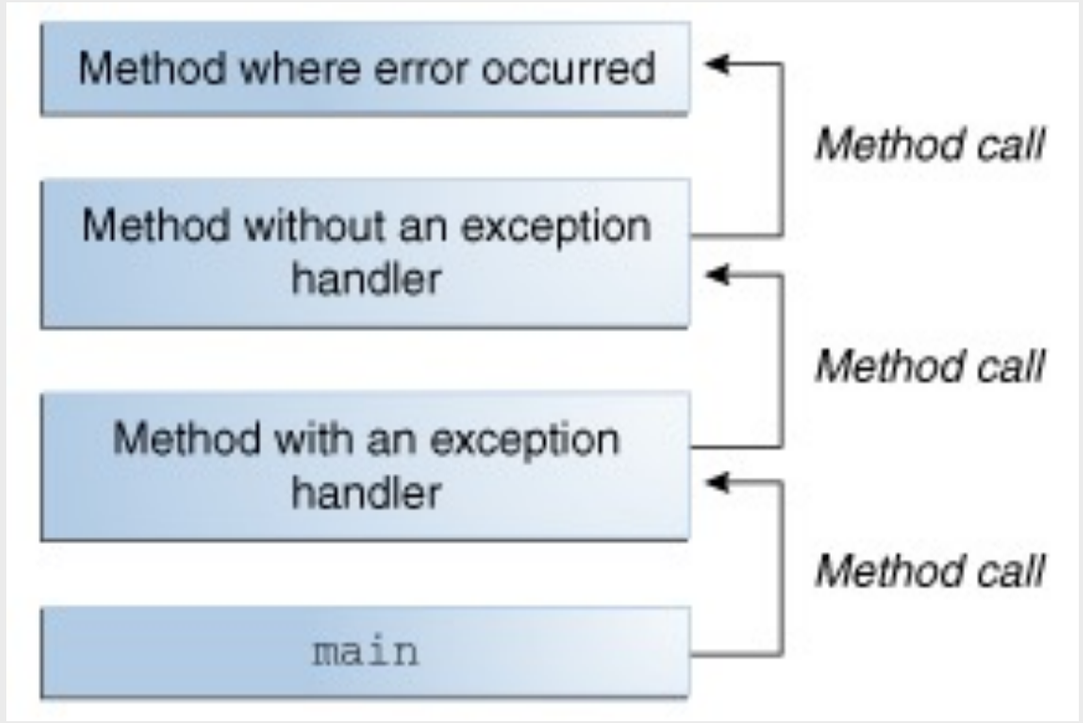
- Few assumptions made about data, state or environment
  - Little assumed to be correct (form, content, value,…)
  - Implicit assumption is that errors abound
- Extensive testing needed to ensure correct execution
- Significant overhead (and clutter code)
- Effectively contains error

Exceptions

An **exception** is an event, which occurs during the execution of a program, that disrupts the normal flow of the program's instructions. (see The Java™ Tutorials)

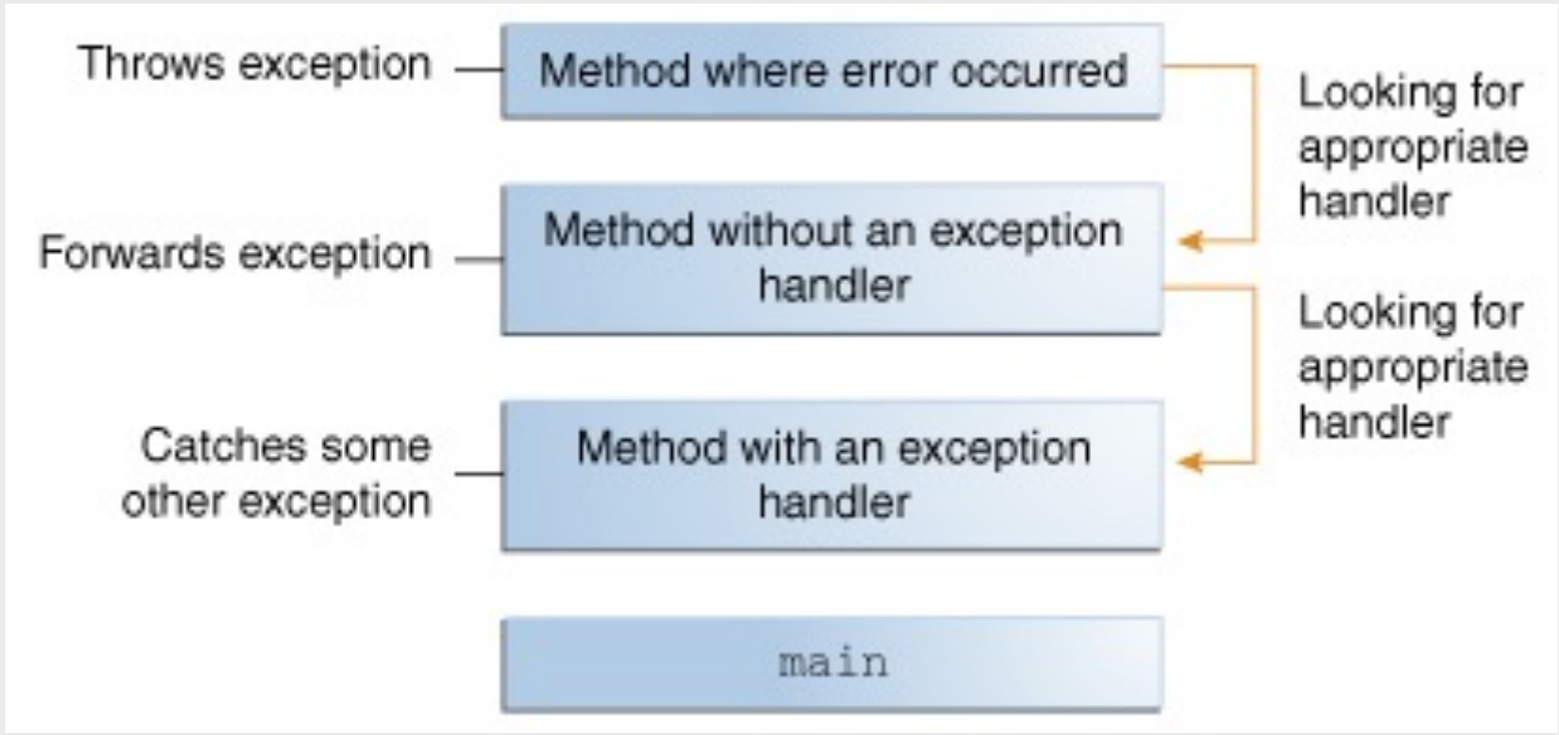
Exceptions

- When an error occurs within a method, the method creates an object and hands it off to the runtime system. The object, called an exception object, contains information about the error, including its type and the state of the program when the error occurred. Creating an exception object and handing it to the runtime system is called throwing an exception.
- After a method throws an exception, the runtime system attempts to find something to handle it. The set of possible "somethings" to handle the exception is the ordered list of methods that had been called to get to the method where the error occurred. The list of methods is known as the call stack.



Exceptions

- The runtime system searches the call stack for a method that contains a block of code that can handle the exception. This block of code is called an exception handler. The search begins with the method in which the error occurred and proceeds through the call stack in the reverse order in which the methods were called. When an appropriate handler is found, the runtime system passes the exception to the handler. An exception handler is considered appropriate if the type of the exception object thrown matches the type that can be handled by the handler.



- The exception handler chosen is said to catch the exception. If the runtime system exhaustively searches all the methods on the call stack without finding an appropriate exception handler, as shown in the next figure, the runtime system (and, consequently, the program) terminates.

The Catch or Specify Requirement

- Valid Java programming language code must honor the **Catch or Specify Requirement**. This means that code that might throw certain exceptions must be enclosed by either of the following:
  - A `try` statement that catches the exception. The `try` must provide a handler for the exception, as described in Catching and Handling Exceptions.
  - A method that specifies that it can throw the exception. The method must provide a throws clause that lists the exception, as described in Specifying the Exceptions Thrown by a Method.
- Code that fails to honor the Catch or Specify Requirement will not compile.
- Not all exceptions are subject to the Catch or Specify Requirement. To understand why, we need to look at the three basic categories of exceptions, only one of which is subject to the Requirement.

The Three Kinds of Exceptions

TYPE	CHECKED EXCEPTION	ERROR	RUNTIME EXCEPTION
Example	`FileNotFoundException` due to trying to open/read a file that does not exist	`IOError` due to not being able to read a file because of hardware or system malfunction	`NullPointerException` due to a logic error when passing null to constructor of FileReader

Example (Will not Compile)

```
import java.io.*;
import java.util.List;
import java.util.ArrayList;

public class ListOfNumbers {

    private List<Integer> list;
    private static final int SIZE = 10;

    public ListOfNumbers () {
        list = new ArrayList<Integer>(SIZE);
        for (int i = 0; i < SIZE; i++) {
            list.add(new Integer(i));
        }
    }

    public void writeList() {
        // The FileWriter constructor throws IOException
        PrintWriter out = new PrintWriter(new FileWriter("OutFile.txt"));
        for (int i = 0; i <= SIZE; i++) {
            // The get(int) method throws IndexOutOfBoundsException
            out.println("Value at: " + i + " = " + list.get(i));
        }
        out.close();
    }
}
```

Try block

The first step in constructing an exception handler is to enclose the code that might throw an exception within a `try` block. In general, a `try` block looks like the following:

```
try {
    // code
}
catch and finally blocks . . .
```

The segment in the example labeled code contains one or more legal lines of code that could throw an exception.

Catch block

You associate exception handlers with a `try` block by providing one or more `catch` blocks directly after the `try` block. No code can be between the end of the `try` block and the beginning of the first `catch` block.

```
try {

} catch (ExceptionType name) {

} catch (ExceptionType name) {

}
```

Each `catch` block is an exception handler that handles the type of exception indicated by its argument. The argument type, `ExceptionType`, declares the type of exception that the handler can handle and must be the name of a class that inherits from the `Throwable` class. The handler can refer to the exception with name.

The `catch` block contains code that is executed if and when the exception handler is invoked. The runtime system invokes the exception handler when the handler is the first one in the call stack whose `ExceptionType` matches the type of the exception thrown. The system considers it a match if the thrown object can legally be assigned to the exception handler's argument.

Finally block

The `finally` block always executes when the `try` block exits. This ensures that the `finally` block is executed even if an unexpected exception occurs. But finally is useful for more than just exception handling — it allows the programmer to avoid having cleanup code accidentally bypassed by a `return`, `continue`, or `break`. Putting cleanup code in a `finally` block is always a good practice, even when no exceptions are anticipated.

```
try {
} catch (ExceptionType name) {
} catch (ExceptionType name) {
} finally {
}
```

Example (with Exception handling)

```
public void writeList() throws IOException {
    PrintWriter out = null;

    try {
        System.out.println("Entering" + " try statement");
        out = new PrintWriter(new FileWriter("OutFile.txt"));
        for (int i = 0; i <= SIZE; i++) {
            out.println("Value at: " + i + " = " + list.get(i));
        }
    } catch (IndexOutOfBoundsException e) {
        System.err.println("Caught IndexOutOfBoundsException: "
                           +  e.getMessage());
    } catch (IOException e) {
        System.err.println("Caught IOException: " +  e.getMessage());
    } finally {
        if (out != null) {
            System.out.println("Closing PrintWriter");
            out.close();
        }
        else {
            System.out.println("PrintWriter not open");
        }
    }
}
```

IndexOutOfBoundsException is an unchecked exception; including it in the throws clause is not mandatory.

How to Throw Exceptions

- Before you can catch an exception, some code somewhere must throw one. Any code can throw an exception: your code, code from a package written by someone else such as the packages that come with the Java platform, or the Java runtime environment. Regardless of what throws the exception, it's always thrown with the `throw` statement.
- As you have probably noticed, the Java platform provides numerous exception classes. All the classes are descendants of the `Throwable` class, and all allow programs to differentiate among the various types of exceptions that can occur during the execution of a program.
- You can also create your own exception classes to represent problems that can occur within the classes you write. In fact, if you are a package developer, you might have to create your own set of exception classes to allow users to differentiate an error that can occur in your package from errors that occur in the Java platform or other packages.

Throw Statement

All methods use the `throw` statement to throw an exception. The `throw` statement requires a single argument: a throwable object. Throwable objects are instances of any subclass of the `Throwable` class. Here's an example of a `throw` statement.

```
throw someThrowableObject;
```

Let's look at the `throw` statement in context. The following `pop` method is taken from a class that implements a common stack object. The method removes the top element from the stack and returns the object.

```
public Object pop() {
    Object obj;
    if (size == 0) {
        throw new EmptyStackException();
    }
    obj = objectAt(size - 1);
    setObjectAt(size - 1, null);
    size--;
    return obj;
}
```

Creating Exception Classes

When faced with choosing the type of exception to throw, you can either use one written by someone else — the Java platform provides a lot of exception classes you can use — or you can write one of your own. You should write your own exception classes if you answer yes to any of the following questions; otherwise, you can probably use someone else's.

- Do you need an exception type that isn't represented by those in the Java platform?
- Would it help users if they could differentiate your exceptions from those thrown by classes written by other vendors?
- Does your code throw more than one related exception?
- If you use someone else's exceptions, will users have access to those exceptions? A similar question is, should your package be independent and self-contained?

Bloch – Items 69-72, 74, 76-77

Item 69: Use exceptions only for exceptional conditions

Item 70 : Use checked exceptions for recoverable conditions and runtime exceptions for programming errors

Item 71 : Avoid unnecessary use of checked exceptions

Item 72 : Favor the use of standard exceptions

Item 74 : Document all exceptions thrown by each method

Item 76 : Strive for failure atomicity

Item 77 : Don’t ignore exceptions

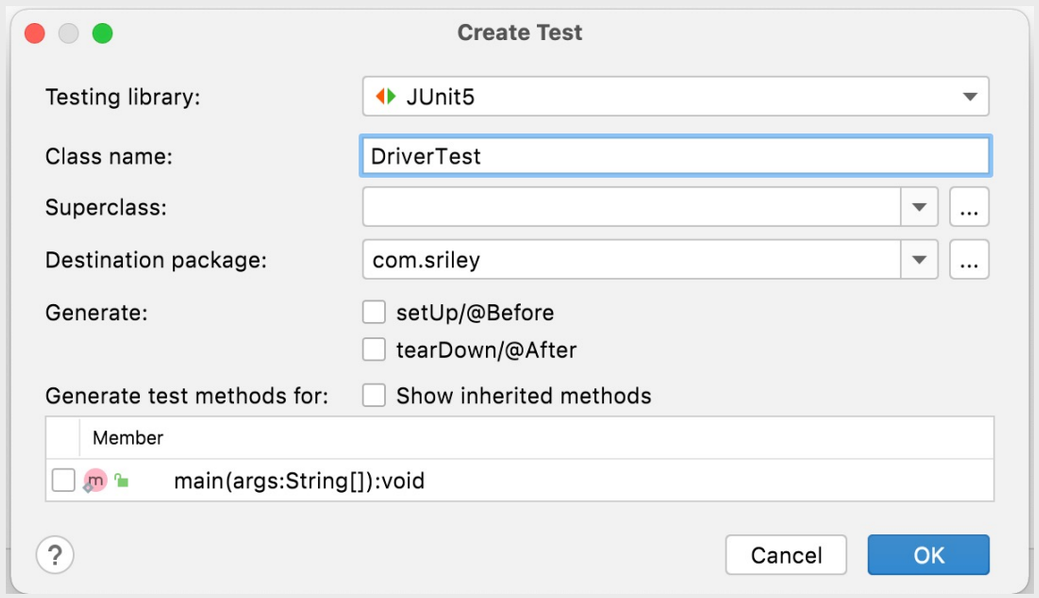
Unit Testing

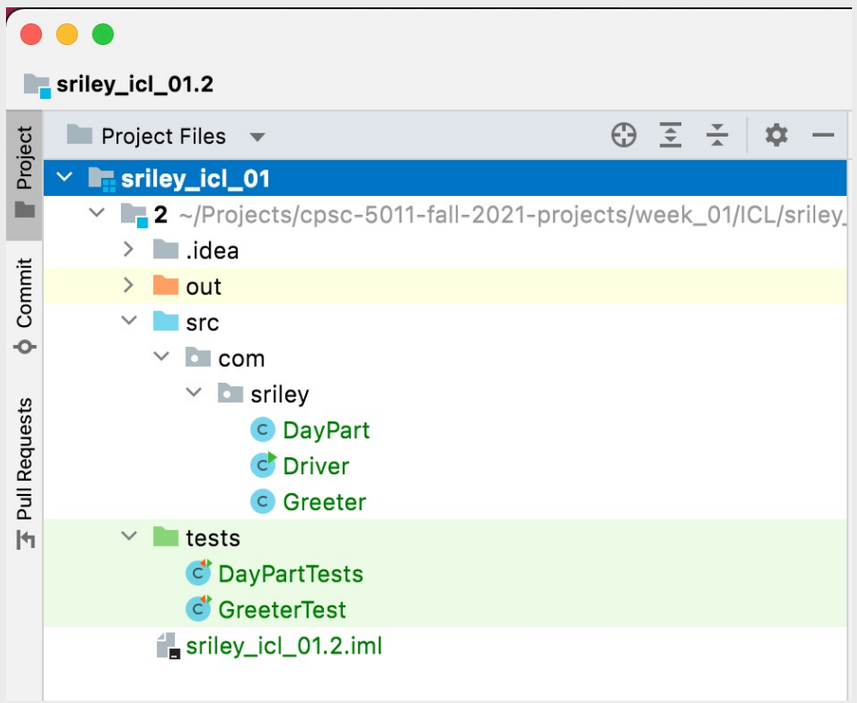
- Unit testing is a software testing method by which individual units of source code, sets of one or more computer program modules together with associated control data, usage procedures, and operating procedures, are tested to determine whether they are fit for use. [see Wikipedia]

JUnit (in IntelliJ)

- Create a “test sources root” per https://www.jetbrains.com/help/idea/testing.html

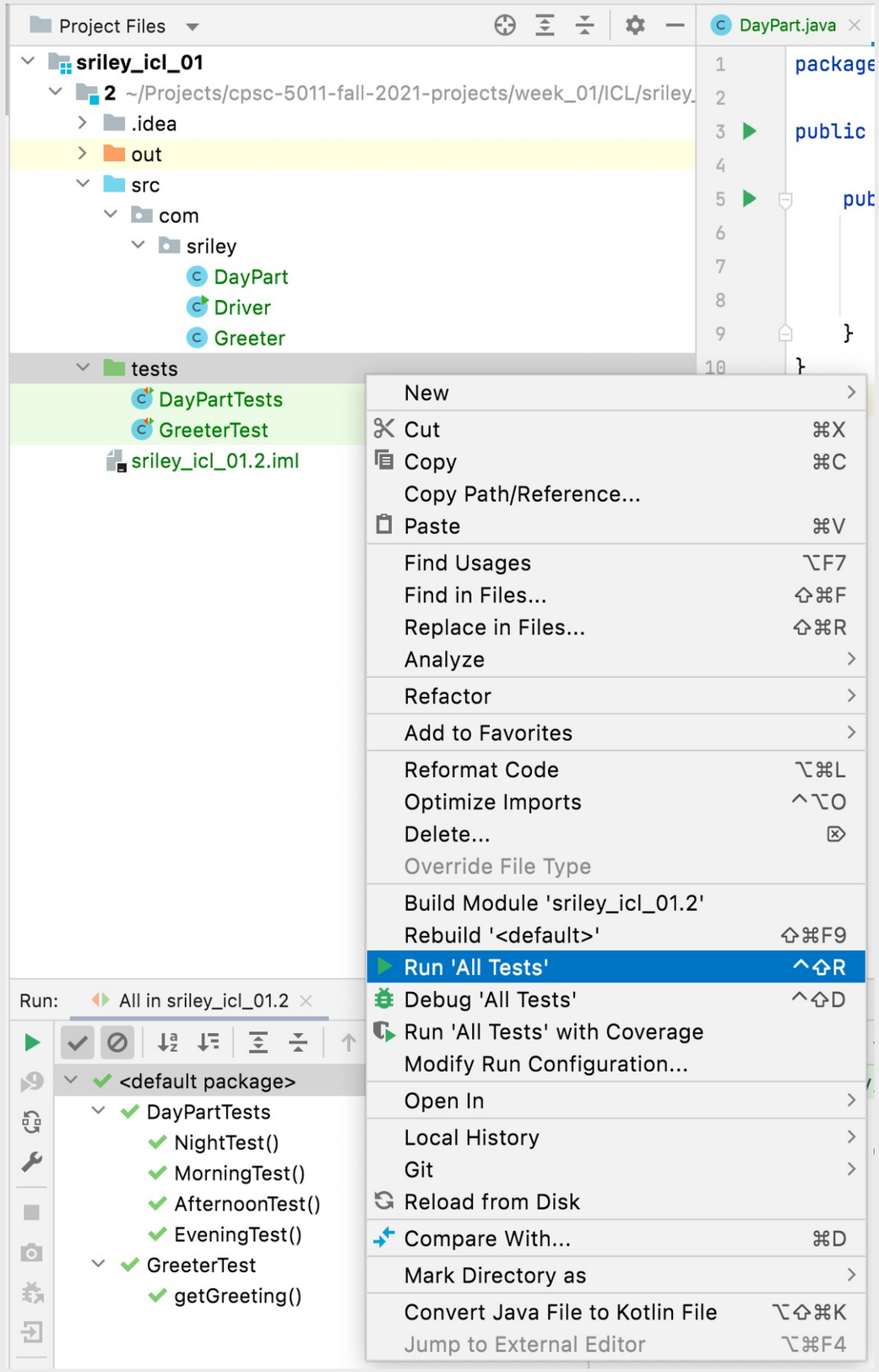
- In IntelliJ, right click on the class you’re trying to test, select “Generate…”, then select “Test…”.





JUnit (in IntelliJ)

- Now you can “Run ‘All Tests’” on the tests folder.



Writing Tests

METHOD NAME / PARAMETERS	DESCRIPTION
assertFalse(test) assertFalse("message", test)	Causes this test method to fail if the given boolean test is not false.
assertNotEquals(value1, value2) assertNotEquals("message", value1, value2)	Causes this test method to fail if the given two values are equal to each other. (For objects, it uses the equals method to compare them.)
assertNotNull(value) assertNotNull("message", value)	Causes this test method to fail if the given value is null.
fail() fail("message")	Causes this test method to fail.

Javadoc Documentation

Javadoc is a tool for generating API documentation in HTML format from doc comments in source code.

References:

- How to Write Doc Comments for the Javadoc Tool
- (Java) Documentation & Style Standards on Canvas

Javadoc (in IntelliJ)

- Make a folder for JavaDoc output
- Select “Tools” -> “Generate JavaDoc…”
- Select your folder in the “Output directory” text box
- Click “OK”
- Docs will load in your browser