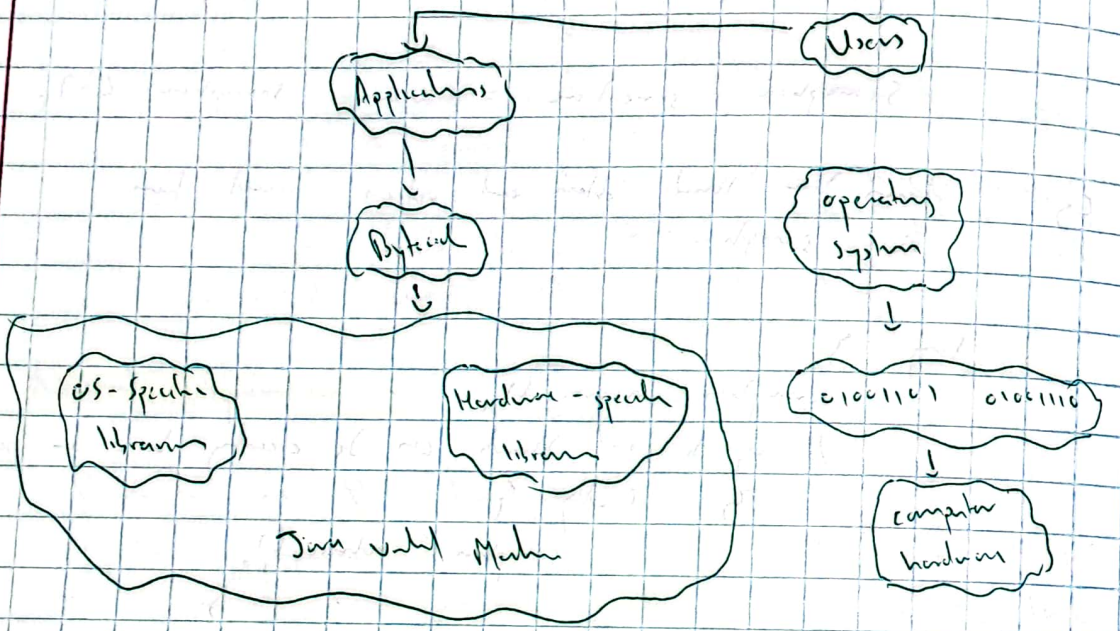
Running Java Applications

Introduction

Java Virtual Machine (JVM)

Diagram of how interaction between users, operating system and JVM



Java Bytecode

What is Java Bytecode?

It is assembly language for the Java

Eg: public static main ([Ljava / lang / String ;) V

LC

LINENUMBER 5 LC

GETSTATIC java / lang / System.out : Ljava / io / PrintStream ;

LDC " Hello world! "

INVOKEVIRTUAL java / io / PrintStream. println

Bytecode is content in file that have .class extension

That means each java class you will have a separate file of bytecode

Except of compiling a .java file into a class file using:

$ javac MainClassFile.java

Java Development Kit

What is the Java Development Kit?

A package that contains the tools to compile and package Java programs, along with the core libraries used to run Java programs. It also contains the Java Runtime Environment (JRE) which is used to run Java programs

Useful JDK tools:

- javac: compile source file into class file
- java: execute java program on JRE
- JRE: collection of environment-specific libraries
- More: jar, jpackage, jshell, jdeps, keytool

Building Java Applications

Building Java Applications

After compiling a java application into a .class file, you can package them together to install a variety of different types of archives. These archives are designed to run in different environments

Java Archive File (JAR)

A JAR file is a zip file full of java bytecode in the form of a bunch of .class file. Also contains a directory called META-INF, which holds a file containing program metadata, called MANIFEST.MF

Web Archive File (WAR)

Designed to be deployed to a Java Web Application Server.

Android package file (APK)

Variation of a JAR that also contains a number of file that specifically declare how the application should run on Android

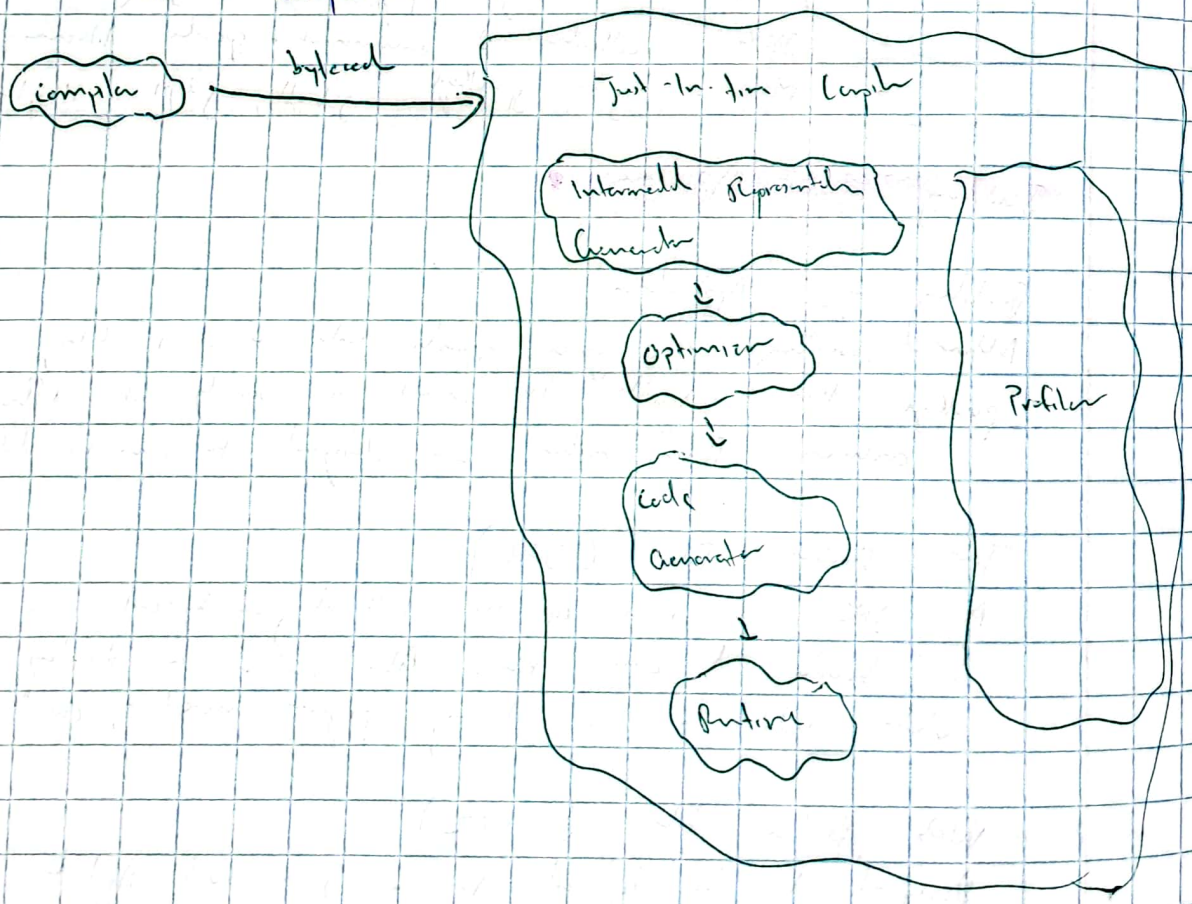
Running Java Applications

Interpreter vs Compilation

Interpreter:
- parses bytecode line-by-line
- performs all operations requested by each bytecode statement
- starts fast, but runs slow

Compilation:
- JVM compiles bytecode into machine code in advance
- saves machine code for reuse
- takes longer to start, but runs faster after beginning

Just-in-time compilation



Tiered Compilation

As of Java 8, JVM uses two different compilers:
- Client Compiler or C1 quickly compiles bytecode into machine code and lets application run right away

- Server Compiler or C2 recompile the application based on the information from the profiler

JShell

Read - Eval - Print loop (REPL)
- Saves a interface a REPL with the program JShell
- To start JShell simply type 'jshell'
- Once you enter JShell, you can type any normal java code to have it immediately executed. Pressing 'tab' will offer code completion suggestions

Useful Commands
- /vars - print a list of all variables you declared
- /types - print all classes you declared
- /methods - print all methods you declared
- /list - print all the source code you entered
- /exit - close JShell

JVM Development
- Java get a new version every 6 months

Dependency Management with Maven

Introduction to Maven

Maven phases

Maven organizes the build process

Validate → Compile → Test → Package → Install → Deploy

Executing a maven phase will run all the preceding phases in order. When maven reaches each phase, it will process all the goals attached to the phase

Introduction to XML

XML objects

- XML is a markup language that allows us to define object data using tags

Eg:
```
<person>
    <name> Alice </name>
    <age> 342 </age>
</person>
```

- XML that closes all open tags properly and has only a single element at the top level is considered well-formed.
- Attributes are a way to represent a nested element inside the definition of its enclosing tag.

Eg:
```
<person isCat = "false">
    <name> Alice </name>
    <age> 342 </age>
</person>

<person isCat = "false" name = "Alice" age = "342" />
```

* Both are equivalent *

Maven Project

pom.xml

Each maven project is defined by a file called pom.xml.
This stands for Project Object Model. Has 4 required elements:

- modelVersion : format of current pom.
- groupId : group identifier for your project. Can be shared
- artifactId : specific identifier for the project. Combination of artifactId and groupId uniquely identifies your project
- version : arbitrary additional identifier indicating what version of your artifact you are on. By incrementing this, you can use Maven to keep track of different versions of your project

Minimal pom.xml

```
<project>
    <modelVersion> 4.0.0 </modelVersion>

    <groupId> com.udacity.jpnd </groupId>
    <artifactId> maven-test </artifactId>
    <version> 1.0.0 </version>
</project>
```

Maven phases and pom.xml

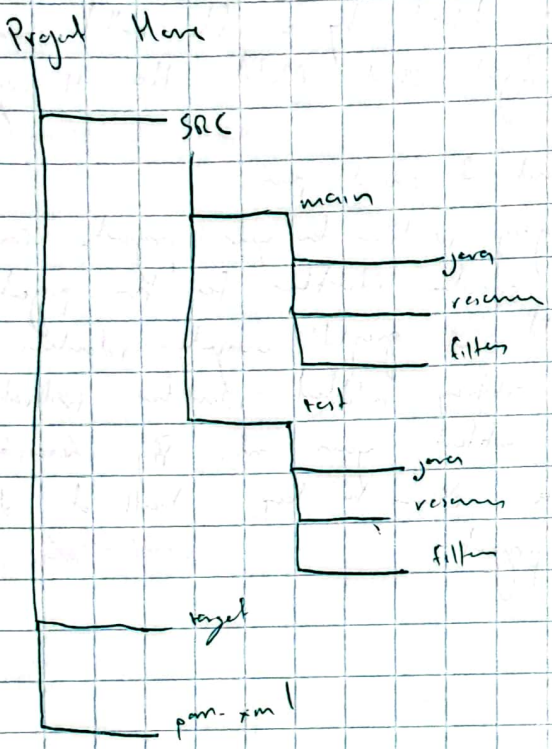
- you can run mvn by using the 'mvn' command followed by the phase you wish to run

Creating a new maven project

1) Use command mvn archetype:generate
2) Press enter to accept default location
3) Press enter to accept latest version
4) Fill in required elements

Maven Project Structure

Standard Directory Layout



SRC :

main : contains source code and resources related to project

test : contains source code and resources for testing your project

java :

Both java folder hold your .java source files

resources : contains non-java file related to running and building your project

filters : contains resource file which that main will use when filtering resources

Importing Dependencies

Eg:
```
<dependencies>
  <dependency>
    <groupId>
    <artifactId>
    <version>
  </dependency>
</dependencies>
```

Maven Dependency location: location & local maven dependencies: /usr/.m2/repository

Dependency Scope and Type

Dependency Scope

The scope of a dependency tells Maven when to include that dependency

- compile : Default Scope
- test : only available when building and running unit tests
- Runtime : only available when application runs
- provided : only available during compilation (not when run)
- System : DEPRECATED.
- import : import all dependencies from another pom

Dependency Type

Tells maven what type of artifact provided by a dependency. Value of this element should correspond to the type provided by the packaging element in the dependencies pom.

- jar : default java archive
- war : web archive
- ear : enterprise archive
- rar : resource adapter
- maven-plugin : package a project to be used as maven plugin
- pom : pom of the project is the primary artifact to produce

Exclusions and Transitive Dependencies

Transitive Dependencies

A transitive dependency is a version required by one of the dependencies included in your project. Eg: If I declare a dependency on the testing utility Mockito, its dependency JUnit will become a transitive dependency of my project.

If your project has multiple transitive dependencies, the nearest definition will win.

Resolving Transitive Dependencies

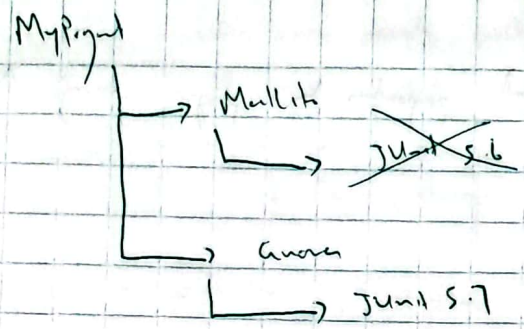
Two options:

① Directly include dependency in pom. The version you use will become the nearest definition and be selected by Maven

② Use the <exclusions> tag to specifically exclude versions you do not wish to use

Example: This code will exclude the version of JUnit from Mockito, resulting in our included version of 5.7 being selected

```
<dependency>
    <groupId>org.mockito</groupId>
    <artifactId>mockito</artifactId>
    <exclusions>
        <exclusion>
            <groupId>org.junit.jupiter</groupId>
            <artifactId>junit</artifactId>
        </exclusion>
    </exclusions>
</dependency>
```



pom inheritance

Pom Inheritance

- all poms in Maven inherit from Super Pom
- can create an inheritance hierarchy among your own poms
- first you have to define a pom to be used as a parent by specifying pom (packaging) type
- then you create a pom that inherit from that pom using the (parent) tag

Multi - Module Project

- to create a Maven project containing multiple modules, create a (modules) tag at the top level of the pom

Maven plugins

All maven goals, even the default behaviors are performed by plugins.

Plugin management vs Plugins

- you can customize the version of the plugin in the (pluginManagement) element of the build
- if you want to customize a plugin for the project only, place the plugin in the (plugins) element instead

Binding Goal to Phase

- to bind a plugin goal to a phase, add an (execution) element to the plugin definition

Configuring Plugins

- can pass additional properties to plugins using the <configuration> element

Maven Properties

Accessing Properties

- can configure the plugins directly

Creating and using properties

You can define your own properties and reference their values in your pom

Eg: <properties>
<sonar.plugin.version>2.0</sonar.plugin.version>
</properties>

Eg of using the property:

<plugin>
<groupId>my.group</groupId>
<artifactId>sonar-plugin</artifactId>
<version>${sonar.plugin.version}</version>
</plugin>

Automatic properties

There are 4 other types of properties Maven creates automatically

- Environment Variables: any variable in the shell environment can be referenced with ${env.VAR_NAME}
- pom elements: properties in the pom can be referenced accessing to the pom in the object structure.

Eg <project><groupId>...</groupId></project> -> ${project.groupId}

- settings.xml : users can provide customization to their Maven profile in a file called settings.xml. These values can be referenced with:
  ${settings.propName}

- Java system properties : anything provided by java.lang.System.getProperties() can be accessed using
  ${java.propName}

Repository

Maven Site

- maven repository helper does a phase called 'site'. Not part of default JAR lifecycle :. must run it manually using it command : mvn site

- this site phase generate documentation about your project. Can customize this behavior by adding additional plugins to the <reporting> element of the pom.xml & by level of the <project> element in your pom.

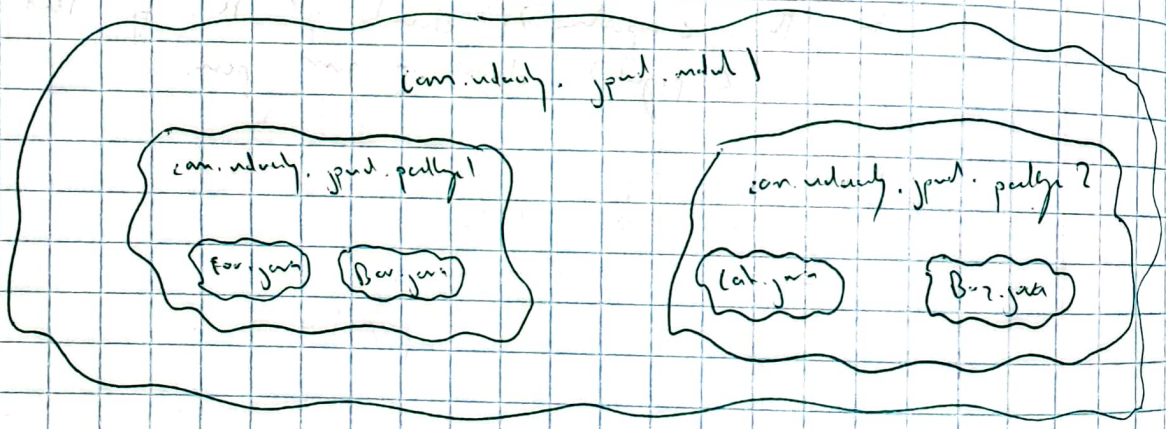
Java Modules

Introduction to Modules

Project Jigsaw
- was the effort to declare dependency relationships in a way that would allow Java to recognize library dependencies and control package-level visibility based on those dependencies
- eventually the changes implemented by Project Jigsaw was merged into Java 9 and became the Module System

What is a module?

Modules as organizational structure
- Modules serve a similar role for organizing packages.
- In the module below, each package contains two classes, and the module (com.udacity.jpwd.module) contains both packages



Module Contents
- a module is still a JAR -> contains classes in packages and use a MANIFEST.MF to declare class metadata
- also contains one additional class called a module descriptor

Module Descriptor: a Java class that provides information about the module. It is stored in a class called module-info.java and compiles into a class file called module-info.class

Defining Java Modules

Module Keywords

- module : declares a new class of the type module.
  Accepts a module name parameter with the same naming restrictions as packages

- exports : identifies a package that should be available to users of this module. Takes a package name as a parameter
  Does not include subpackages

- requires : marks the package in the specified module available in this module. Takes a module name as a parameter.
  You cannot require a package that also requires you.
  Circular dependencies will not compile

Example : module-info.java

```
module com.udacity.jpwd.module1 { export
com.udacity.jpwd.module1.somepackage; requires
com.udacity.jpwd.module2; }
```

Additional Module Keywords

- to : can be used to modify exports to limit with modules can access the specified package

- transitive : modifies the requires keyword and specifies that the required module will also be made available to any other module that includes this module

- opens : makes the specified package available for both public and private class reflection access. Can optionally use the to keyword to limit which modules have this access

Eg:
```
module com.udacity.jpms.module1 {
    exports com.udacity.jpms.publicpkg1;
    exports com.udacity.jpms.internalpkg to com.udacity.jpms.module2;
    opens com.udacity.jpms.internalpkg to com.udacity.jpms.module3;
    requires transitive com.udacity.jpms.module2;
}
```

Exports is used to grant compile time access to pkgs, but open is used to grant runtime access to pkgs

Access granted by command:

	Compile-Time	Run-Time
Module Export	public only	public only
Non-module export	public only	public or private
Module open	none	public or private

Service Provider Interface

Modules also interact with JPMS Java Service Provider Interface (SPI), which is a way for Java to dynamically discover and load implementation for a specified interface.

To declare or use a service defined by SPI, you can use JPMS provides / with and uses keywords

```
module module1 {
    exports module1.servicepkg1;
    provides module1.servicepkg1.MyInterface with module1.servicepkg1.MyInterfaceImpl;
}

module module2 {
    requires module1;
    uses module1.servicepkg1.MyInterface;
}
```

Module Types

Classpath vs Modulepath

- before modules, building a Java project involved placing all project classes and jars on the CLASSPATH. This is the path when your compiler looks for all your class files
- with modules, you place all modules at different locations, called the MODULEPATH. This is the path where the compiler looks for all your modules

Unnamed Module

- all your applications, use module system
- even if they contain no modules, everything placed on the CLASSPATH are placed into what is called the Unnamed Module

Automatic Modules

- unnamed module cannot access content in Modules, because it cannot use the require statement.
- named module cannot access unnamed module, because unnamed module cannot be referenced by require

- The solution is Automatic Modules
- automatic modules are created by placing non-modules jars on the module path.
- default name of automatic module is the name of the jar, but the name can be overridden by the Automatic-Module-Name property in the JAR Manifest.

Module Access

- Automatic modules can access everything in the unnamed module

	Named Module	Unnamed Module	Automatic Module
Created by	module-info.java	Jar on CLASSPATH	Jar on MODULEPATH
Export	In module-info	n/a	Everything
Opens	In module-info	n/a	Everything
Requires	In module-info	n/a	Everything
Access	Named and Automatic	Unnamed and Automatic	Named, Unnamed and Automatic

Java Linker

Jlink

- a tool that allows us to create a custom JRE containing the minimal components necessary to run a specific java module.
- can run it by simply typing jlink

Analyze Dependencies

- use tool called jdeps
- Eg: jdeps Example.jar

Building Runtime

- to run jlink, must put all required modules on module path
- We use --add-modules flag to specify which module to add to our JRE
- tool present on output directory
- modules comprising the Java Standard library can be found in the /jmods directory of your java install

Eg:

```
jlink --module-path "$JAVA_HOME/jmods" --module-path target/classes
      --add-modules com.study.jpms.module1 --output tinyJRE
```

Using the runtime

- the new tinyJRE directory is created and contains a complete Java Runtime that can run our program

```
tinyJRE/bin/java.exe -jar Example.jar
```

Migration to Java 9 modules

Should you upgrade
- you don't need to upgrade to modules to run a newer version of Java. All versions will without a module-info.java

When to create modules:
- many legacy dependencies
- infrequent changes
- not shared

When to use modules:
- new projects
- applications used by other applications
- hide internal method and data types

Resolving Duplicate Packages
- placing multiple project with the same packages on the classpath is not a problem, because the packages simply coexist
- you cannot place packages with the same name on the module path
- if you have different projects that share a package name, you will need to refactor one of them to use a different package name

Create module-info.java
1) Create a minimal module-info.java in the src/main/java directory containing only your new module name
2) Restart errors in IntelliJ
3) Build project with maven using main package

Common Problems
- revolve around the relationship between dependencies and transitive dependencies in what are or both are non-modular projects

Module Transitive Dependencies & non-module Dependencies

- If your non-module dependency returns transitive dependencies that are modular, those dependencies will become modules and be inaccessible to your module unless you explicitly require them

Reflective Access to Non-Module

- cannot open your module to the unnamed module
- If a transitive dependency requires reflective access to your module, it will be unable to access the packages unless you fully open the package to everyone

One solution for this case is to limit the scope in which you open your packages by only opening the module during the execution of the test plugin

Eg:

```
<plugin>
<artifactId> maven-surefire-plugin </artifactId>
<version> 3.0.0-M5 </version>
<configuration>
<argLine>
--add-opens com.udacity.jpnd.mymodule/com.udacity.jpnd.mypackage
=ALL-UNNAMED
</argLine>
</configuration>
</plugin>
```

Libraries removed from JDK

- some libraries are not exposed by the core JDK as modules
- can check for these classes using:

```
jdeps -jdkinternals pathToClasses
```

Java Unit Testing

Introduction to unit testing

Unit Test : A piece of code that verifies the behavior of another piece of code

JUnit

Junit
- main testing framework for java

Adding JUnit as a Maven dependency

```
<dependency>
    <groupId> org.junit.jupiter </groupId>
    <artifactId> junit-jupiter </artifactId>
    <version> 5.7.0 </version>
    <scope> test </scope>
</dependency>
```

Creating a test class

Test class : class containing all JU unit tests for a corresponding application class
- location : src/test/java
- package : same as your application class

Writing Unit Tests

Identifying Test Methods
- mark methods as test by using @Test annotation

Using Assertions
- Junit provides JU Assertion class to cause conditional test failure
- you can use Assertion.assertEquals to result each test method.

```
@Test
public void additionNumber_firstNum_returnSum() {
    int input1 = 5;
    int input2 = 6;
    int expected = 11;
    App app = new App();
    Assertion.assertEquals(expected, app.additionNumber(input1, input2),
        "5+6 should always equal 11");
}
```

Type of Assertions

- assertEquals(expected, actual) - fails if expected and actual don't match
- assertArrayEquals(expected, actual) - fails if any element of two arrays not equal
- assertIterableEquals(expected, actual) - for iterable collections
- assertTrue(condition) - fails if condition doesn't match what method expects
- assertNull(object) - fails if object null status does not match
- assertThrows(class exceptionType, executable) - runs the executable and fails if provided exception type is not thrown
- assertAll(Executable ... executables) - runs all executables and fails if any of the executable fail
- fail - always fails

Using assertAll

- this test verifies that all three properties match, and report on this, test will indicate all properties that do not match instead of failing in the middle

```
@Test
public void getCat_returnBeigeSomeAge10() {
    Cat cat = catTest.getCat();
    assertAll("This message will print in test report",
        () -> assertEquals("some", cat.getName()),
        () -> assertEquals(Color.BEIGE, cat.getColor()),
        () -> assertEquals(10, cat.getAge())
    );
}
```

Test Naming Conventions

- don't use word 'test'
- start test name with method
- include condition you're testing
- end with expected result
- separate fragment with underscore

Parameterized and Repeated Tests

Parameterized Tests

- use @ParameterizedTest annotation to return same test with different input parameter multiple times.
- used in combination with an ArgumentSource

Example: use a @ValueSource to pass in initial value

```
@ParameterizedTest
@ValueSource(ints={1, 5, 100, 1000, 100000})
public void addTwoNumber_addsTwo_returnNumberPlusTwo(int number) {
        int input1 = 1;
        int expected = number + 1;
        App app = new App();
        Assertions.assertEquals(expected, app.addTwoNumber(number, input1));
```

CsvSource

- can pass multiple value at once using a @CsvSource, which accepts a list of Strings containing comma-delimited values.
- will automatically attempt to parse this list and cast it value to primitive types

eg:

```
@ParameterizedTest(name = "[{index}] {0} + {1} = {2}")
@DisplayName("Add 1 to named number \uD83D\uDCA4")
@CsvSource({
        "1, 5, 6",
        "1, 7, 8",
        "1, 100, 101",
        "1, 1000, 1001",
        "1, 10000, 10001"
})
```

```
public void addTwoNumbers_addsTwoNumbersPlusOne(int input1, int input2, int expected) {
    App app = new App();
    Assertions.assertEquals(expected, app.addTwoNumbers(input1, input2));
}
```

MethodSource

- can create an ArgumentSource programmatically, using @MethodSource, which takes a single string parameter specifying the name of the method that should be used to provide Arguments

```
@MethodSource("fancyArgumentProvider")
```

For example, the above annotation would tell a @ParameterizedTest to use this method to provide its arguments:

```
private static Stream<Arguments> fancyArgumentProvider() {
    return Stream.of(
        Arguments.of(6, 5, 11),
        Arguments.of(6, 1, 7)
    );
}
```

EnumSource

- iterates over all the members of an enum and passes them as an input parameter to your test

```
enum SomeNumber {
    one(1), five(5), ONE_HUNDRED(100);
    int number;
    SomeNumber(int number) { this.number = number; }
    public int getNumber() { return this.number; }
}
```

@ Parameterized Test

@ EnumSource (SomeNumber.class)

public void addTwoNumber - addWithEnumSource returnValuePlusSix (SomeNumber num){

App app = new App ();

int expected = switch (num) {

case ONE -> 7;

case FIVE -> 11;

case ONE-HUNDRED -> 101;

default -> 0;

};

Assertion. assertEquals (expected, app. addTwoNumber (6, num. getNumber()));

}

Repeated Tests

- to run a same test multiple time, you can use

@ RepeatedTest annotation

- takes single parameter specifying number of iterations
- allows you to pass an optional parameter to your test method called RepetitionInfo.

Example: Tests number 1-10

@ RepeatedTest (10) public void

addTwoNumber - addWithReturnValuePlusOne (RepetitionInfo repetitionInfo) {

int input1 = repetitionInfo. getCurrentRepetition();

int input2 = 1;

int expected = 1 + input1;

App app = new App();

Assertion. assertEquals (expected, app. addTwoNumber (input1, input2));

}

Unit Test Lifecycle

JUnit lifecycle stages

① Construct class
② Run any method with @BeforeAll
③ For each test:
   - run any method with @BeforeEach
   - run test method
   - run any method with @AfterEach
④ Run any method with @AfterAll

Writing Good Unit Tests

Unit Test Tips:
- short - simple unit test that verify a single, concrete requirement
- name
- success and failure - test for failure and invalid states
- speed - tests should run quickly
- minimize preconditions - don't overuse @BeforeEach
- Thorough - test all branching conditions

Code Coverage

Testing the Tests
- code coverage is a tool for helping evaluate your test
- right-clicking on a test method or class and selecting Run with Coverage -> IntelliJ

Tests in Maven

Unit Tests in Maven

- our test will run when my phase later than test phase are reached
- can add reports of unit test to the site report generated by 'mvn site'
- this is accomplished by adding the surefire-report plugin to your <reports> tag in your pom.xml

Test Doubles, mocking and Integration Testing

Testing and Class design

Testing Around Dependencies

- what if we want to test another class that uses that fizzBuzz method
- move fizz buzz into a new class called SalesSensor and then reference it from inside another class

How do we test UserSensor.sayHello without relying on behavior of SalesSensor.fizzBuzz?

① Create a custom SalesSensor that does what we tell it to do
② Get the fake SalesSensor into UserSensor so it uses that instead of its normal behavior

Code: `public interface SalesSensor {`
`String fizzBuzz(int i);`
`}`

Creating an interface for sensor mocks it easy to create fake versions of those sensors. The fakeSalesSensor returns whatever value we want to create it when fizzBuzz is called

```
public class FakeSalesSensor implements SalesSensor {
    private String returnValue;
    public FakeSalesSensor(String returnValue) {
        this.returnValue = returnValue;
    }
```

@Override

```
public String fizzBuzz(int i) {
    return returnValue;
}
}
```

Decouple Dependencies: update UserSensor so that we can replace SalesSensor with it on we want

```
public class UserServiceImpl implements UserService {
    SalesService salesService;

    public UserServiceImpl ( SalesService salesService ) {
        this.salesService = salesService;
    }

    public String fizzBuzz (int n) { /* Implementation */ }
}
```

Testing:

① If salesService.fizzBuzz returns "Fizz", "Buzz", or "FizzBuzz", then fizzBuzz just returns that String:

```
@ ParameterizedTest
@ ValueSource ( Strings = { "Fizz", "Buzz", "FizzBuzz" } )
public void fizzBuzzCallsFizzBuzzService_returnsExpectedString ( String returnString) { }

    SalesService salesService = new FakeSalesService ( returnString );
    UserService userService = new UserServiceImpl ( salesService );

    assertEquals ( returnString, userService.fizzBuzz (1));
```

Test Doubles

Types of Test Doubles:

A test double is any kind of object that acts as a stand-in for an expected dependency in order to facilitate testing

Dummy: a class that doesn't actually do anything, but fills a required parameter of some method

Stubs: a simple class that always returns a hard-coded value

Fake: A test double that allows you to customize its response to imitate actual functionality

Spy: A test double that can keep track of what methods was called sometimes they can also track what parameters were used & how often the methods were called.

Mock: A general term for test double, or more specifically, a test double that can prepare special responses for different kind of input

Mockito

Why write our own Mocks?
- Mockito is a framework that can automatically generate mock object
- A Mockito Mock is an object that can programmatically implement any behavior from any of the test doubles we learned earlier. Mockito Mocks can be spies, dummies, stubs & mocks
- A Mockito spy can do everything a Mock can do, but it also inherits the behaviors of the class it extends. Sometimes the created is referred to as a partial mock, because the part mock, part original class

Mock vs Spy
- generally -> use Mockito Mock.
- only reason to use spies is if you need to verify that internal methods are called on your tested class

Adding Mockito
- first add dependency to Maven

```
<dependency>
    <groupId> org.mockito </groupId>
    <artifactId> mockito-junit-jupiter </artifactId>
    <version> 3.6.0 </version>
    <scope> test </scope>
</dependency>
```

- need add M Mockito test runner to your unit test by using M @ExtendWith

@ExtendWith (MockitoExtension.class) public class UserServiceMockitoTest { }

Creating a Mock
- annotate class attribute with @Mock

Eg: @ExtendWith (MockitoExtension.class)
public class UserServiceMockitoTest {
private UserService userService;

@Mock
private SchoolService schoolService;

@BeforeEach
void init() {
userService = new UserServiceImpl (schoolService);
}
}

Alternatively create a @Mock schoolService for each unit test and inject it into a new UserService in the init method

Eg: @ParameterizedTest
@ValueSource (strings = { "Fizz", "Buzz", "FizzBuzz" })
public void fizzBuzzShouldReturnGivenString (String returnString) {
SchoolService schoolService = new FakeSchoolService (returnString);
UserService userService = new UserServiceImpl (schoolService);

assertEquals (returnString, userService.fizzBuzz(1));
}

This created a fake using dependency which it should return. Some behaviour can be emulated by using Mockito using 'when' method.

@ ParameterizedTest
@ ValueSource (strings = { "1", "2", "FizzBuzz" })
public void fizzBuzz_getFizzBuzzString_returnsCorrectString (String input) {
Mockito.when (sutSpy.fizzBuzz(1)).thenReturn (input);
Assertions.assertEquals (input, sutSpy.fizzBuzz(1));
}

Argument Matchers

Matching Real Input

In previous example, can use ArgumentMatchers.anyInt()

when(sutSpy.fizzBuzz (anyInt())).thenReturn ("2");

Types of ArgumentMatchers

Type Matchers:
- primitive: anyInt(), anyFloat(), anyBoolean() etc
- object: any(), any(class)
- collection: anyCollection(), anyIterable(), anyList(), anyMap(), anySet()

Value Matchers:
- primitive: eq(val)
- object: eq(obj), same(obj), refEq(obj))

Mockito Verify

Verify Method Called

- Mockito.when allows us to create if then behaviour of method, but we need a different tool to tell if a method was called.
That tool is 'verify'

Eg: check if the method that save results to a database is called

```
public class PersistentSchSave implements SchSave {
    private SchRepository schRepository;

    public PersistentSchSave(SchRepository schRepository) {
        this.schRepository = schRepository;
    }

@Around
public String fizzBuzz(int i) {
    String retVal = fizzBuzz.Intern(i);
    schRepository.saveResult(i, retVal); // verify the method
    return retVal;
}
}

@ExtendWith(MockitoExtension.class)
class PersistentSchSaveTest {

    @Mock
    private SchRepository schRepository;
    private PersistentSchSave persistentSchSave;

    @BeforeEach
    void init() {
        persistentSchSave = new PersistentSchSave(schRepository);
    }

@RepeatedTest(15)
void persistentSchSave_fizzBuzz_saveToDb(RepetitionInfo repetitionInfo) {
    int i = repetitionInfo.getCurrentRepetition();
    String result = persistentSchSave.fizzBuzz(i);
    verify(schRepository).saveResult(i, result);
}
}
```

Verify with ArgumentMatchers

// Checks that any string is sent for 1st and integer
verify(salesRepository).save(1, anyString());

Verify with Number of Interactions

// fail if save() not called exactly 2 times
verify(salesRepository, times(2)).save(1, anyString());

// fail if save() called more than 10 times
verify(salesRepository, atMost(10)).save(1, anyString());

// fail if save() not called 3 or more times
verify(salesRepository, atLeast(3)).save(1, anyString());

Verify Order of interactions

// fail if delete() and save() not called in order
InOrder inOrder = Mockito.inOrder(salesRepository);
inOrder.verify(salesRepository).delete(1);
inOrder.verify(salesRepository).save(1, result);

Integration Testing

Automated Testing Scope:
- unit test: easier to write and maintain, narrowest scope. Write most of these
- integration Testing: test the interaction between units. Span multiple components
- Functional tests: verify behavior of application from start to finish

Continuous Integration

Continuous Integration / Continuous Delivery

Continuous Integration: process of automatically running unit test on every time you code change. It usually is performed by a server that monitors the state of your repository and runs all your unit test whenever you commit new changes

continuous delivery : process that follows CI and involves creating a deployable build artifact every time the project changes

Containerization : means creating a self-contained environment with all the parameters to build and run your application

Docker : is an application that provides a container format and an engine that can run these containers