**Introduction**

**Learning Objectives**

* Install IntelliJ IDE
* Create a Java project in IntelliJ
* Know what a statically typed language is
* Understand how compiled languages differ from interpreted languages
* Be able to write, compile and run a Java program.

**Introduction**

So we have looked at Object Orientated programming in Ruby so far. This is fine as Ruby does lend itself well to OO programming for lightweight applications.

Imagine you are a developer working on a large enterprise code base where there are tens to hundreds of classes and millions of lines of code.

Ruby is a Dynamically Typed language though and this means that, while it can be forgiving, it can sometimes lead to mistakes at a more enterprise level of programming.

No type declaration on a method means you do not have any type information coming in or going out of it. Now the question is, can you consume this function without any documentation or going through/understanding the code completely?

Also when you are creating instances can you be sure that you are creating an instance of the right thing? And that it can’t be changed later.

For example…

**Dynamically Typed**

*#terminal*

touch ruby\_play.rb

*#ruby\_play.rb*

**class** **Cat**

**end**

**class** **Dog**

**end**

myCat **=** Cat.**new**

myCat **=** Dog.**new**

*#terminal*

ruby ruby\_play

The variable myCat has been assigned to first a Cat object and then a dog object. Ruby has no problem with this. Variables are just variables they don’t care what type of object they refer to. Ruby is chill.

**Statically Typed**

Statically types languages are much more uptight. Variables need to know what kind of thing they are referring to, and always have to refer to that type of thing.

**Intro to Java**

Give out starter code and open in Atom

In Java unlike Ruby, we can’t just run any java files directly from the terminal. We need to have an entry point for the program to start from.

Look at Runner.java. In here we have a Main method. Every java console program needs this method in order to run in the console. (Unless we are just using tests in a TDD fashion which we will cover later).

Don’t worry about the void/static etc as we will understand what all those words mean over the week.

*#terminal*

cd intro\_start\_point

java Runner

Note we don’t need to specify the file type i.e. Runner.java as java will look for a specific file type.

We get an error if we do this telling us Error: Could not find or load main class Runner

What needs to happen first is that we need to compile the java files into something that the computer can use. These are called class files.

We do this by typing the word javac. This stands for java compiler. Most statically typed languages involve a compilation step. It is here that the type checking happens, among other checks. Remember those ‘no method errors’ in Ruby. All these errors will be caught at compile time in a statically typed compiled language. It also gives us the ability for much more powerful tooling as our editor/IDE can tell us when something is of the wrong type immediately. We will see this later on.

Using \*.java runs javac on all the files with a .java extension, saving us having to type out each file separately.

*#terminal*

javac **\***.java

Our compilation succeeds. The compiler is happy that the code is free from any syntax errors that may have otherwise caused compilation to fail. The compiler produces ‘not for humans’ files, which can now be executed.

*#terminal*

java Runner

Yay we have some output.

So this is all good and well but there are a lot of problems you can face if you are trying to write larger java applications using a text editor.

If there are any errors which may cause a compilation error you won’t find out until you run javac on the files. These can be mainly things like syntax errors and missing methods etc. The kind of things that we would maybe not notice until compilation time. Also there are a lot of extra things that we may need to do in order to get our application up and running such as importing libraries to use,

Wouldn’t it be nice if we had an environment where any errors are highlighted straight away so we can fix them and where the majority of our imports are done for us to make sure we don’t forget?

Enter the IDE…..

**IntelliJ**

We are going to be using an IDE called IntelliJ for the next 2 weeks.

An integrated development environment (IDE) is an application that facilitates application development. IDEs offer a central interface featuring all the tools a developer needs, including the following:

* Code editor: This feature is a text editor designed for writing and editing source code. Source code editors are distinguished from text editors because they enhance or simplify the writing and editing of code.
* Compiler: This tool transforms source code written in a human readable/writable language into a form executable by a computer.
* Debugger: This tool is used during testing to help debug application programs.

So first off we need to install IntelliJ if we don’t already have it.

*#terminal*

brew cask install intellij-idea-ce

Open IntelliJ and get to the home screen.

We will start by adding some settings that will help us out later on. Don’t worry too much about what these settings do for the moment we will explain as and when we use them.

# IntelliJ

Click Configure and select Preferences

Select Build, Execution and Deployment > Build Tools > Maven > Importing

Check the box that says `Import Maven projects automatically`

Next go to Editor > General > Auto Import

Check the box that says `Add unambiguous imports on the fly`

Click `Apply` then `OK`

Now let’s start a new Java project.

#IntelliJ

Create New Project

We are going to use a tool called Maven to help us with our applications.

Maven is a dependancy management tool that will allow us to add libraries to our application that we will need to use. Remember in ruby we did all those gem install commands to add libraries to our programs? Maven does a similar thing.

We will explore Maven in more depth as we go through this course.

# IntelliJ

Click on Maven in the left hand pane

Make sure the project SDK is showing 1.8

Next

Expand Artifact Coordinates

GroupId: com.codeclan.example

ArtifactId: IntroToJava (Our application name)

Version: Leave as it is.

Change project location to your codeclan\_work/week\_11/day\_1 folder and add the name of the project to the end of the file path.

(So should end with codeclan\_work/week\_11/day\_1/IntroToJava)

Finish

IntelliJ now creates our project for us and installs what we need to get started.

Expand IntroToJava folder and then expand src folder.

You will see that we have two folders in here. There is one called main where all our java files will go and another called test where our test files will go.

There are a lot of other folders in here but we don’t need to worry about those for just now.

So lets start off by creating some Java classes

#IntelliJ

Expand main folder

Right click on java folder.

New > Java Class

Name: Cat

Kind: Class

Ok

So now IntelliJ creates our new class called Cat and pre-populates it for us with a basic class structure.

Please note that in Java the class names are always capitalised and that the file name has to match the class name in the same way. If the file name wasn’t capitalised we would get an error.

**Task**

Create 2 more java classes in this folder. One called Dog and one called Runner.

*//Cat.java*

**class** **Cat{**

**}**

*//Dog.java*

**class** **Dog{**

**}**

Open the Runner class and let’s add a main method so we can start our application.

*// Runner.java*

**class** **Runner{**

**public** **static** **void** **main(String[]** args**){** *//added*

**Cat** myCat **=** **new** **Cat();** *//added*

myCat **=** **new** **Dog();** *//added*

**}**

**}**

Let’s focus on lines 4 and 5 in Runner.java. IntelliJ is being helpful and showing us that there is an error on line 5 by showing a red underscore.

Hover over it and you will see an error message.

**Compile time vs Runtime errors**

This is an example of a Compile time error.

There are two types of errors we may see. Compile time means any error that will cause compilation to fail. These can be things like trying to call missing methods, syntax errors, passing wrong number of arguments to methods.

The other type of error is a runtime error. These won’t fail compilation as they look fine but when the program tries to run it will fail. These can be things like trying to divide by 0, accessing a file that is missing, accessing an index of an array that isn’t there. More on these later.

In Ruby because there is no compilation stage all errors are highlighted when you try and run the script.

In above example we are creating a myCat variable and assigning it to a Cat object. What’s different here from the Ruby version? The variable has been given a type Cat. Because this variable is for this type and this type only, we can’t assign it to a Dog object.

[Expected q: Why would a language ask for this]

Statically typed languages give us an extra level of protection. We can be sure that the type we are given is what we expect. This will come more apparent when we are creating functions/methods.

Remove line 5, as we can’t do this now.

*//Runner.java*

**class** **Runner{**

**public** **static** **void** **main(String** args**[]){**

**Cat** myCat **=** **new** **Cat();**

*// deleted line 5*

**}**

**}**

**Running the Program in IntelliJ**

Ok so how do we run this program?

#IntelliJ

Right click on the Runner file in the project pane.

Select Run Runner.main()

You may see a warning about JavaLaunchHelper being implemented in 2 places. We can ignore this warning.

We have run our first program. No output! Let’s give the cat a little meow method.

*//Cat.java*

**class** **Cat{**

**public** **String** **meow(){**

**return** "Hello Meooooow"**;**

**}**

**}**

Let’s ask our cat to meow!

Notice that when you start to type in cat in runner it will show suggestions for autocomplete listing available methods in the cat class.

*//Runner.java*

**class** **Runner{**

**public** **static** **void** **main(String** args**[]){**

**Cat** myCat **=** **new** **Cat();**

**System.**out**.**println**(**myCat**.**meow**());**

**}**

**}**

System.out is just like puts (ruby) or console (JS).

Because we have already run our Runner file that at the top right of IntelliJ there is a green arrow and the drop down now says Runner. This means we can now re run the program by simply clicking on the arrow.

Yay!, our cat now Meows.

**SOLID and 4 Pillars of OOP**

As we go through the next few weeks you will get to see more and more features of the IDE and understand how it can be a very powerful tool for developers. We will also delve further into the structure, keywords and syntax of Java to see how it all comes together to create type-checked applications and the benefits this can bring.

There are 2 concepts we will be covering throughout the next 2 weeks. SOLID and the 4 pillars of object orientated programming.

These concepts are not really anything new as, so far, we have already been adhering to most of these.

SOLID is a set of 5 principles to adhere to when writing code. They make it easy to maintain your applications and make them more robust.

In order to manage so many classes and to reduce the complexity, system designers use several techniques which can be grouped under four main concepts. These are the 4 pillars of OOP.

We won’t go into any depth about these just now but we will fill in the blanks for each as we go along.

Instructor: Write SOLID and 4 pillars as headings on board

For Reference. SOLID stands for:

* **S**ingle responsibility principle
* **O**pen/closed principle
* **L**iskov substitution principle
* **I**nterface segregation principle
* **D**ependency inversion principle

4 Pillars are:

* Abstraction,
* Encapsulation,
* Polymorphism,
* Inheritance

**Summary**

We’ve seen:

* How to install IntelliJ IDE
* How to create a Java project in IntelliJ
* What a statically typed language is
* How compiled languages differ from interpreted languages
* How to write, compile and run a Java program.

P

**OO Single Class**

**Learning Objectives**

* Demonstrate create a class
* Show creating instance Variables
* Show creating methods.
* Show defining parameters, and return type of methods.
* Revise catching type errors at compile time.

**Creating a bear class**

Our file name must match the class name.

Start a new IntelliJ project called SingleClasses.

#IntelliJ

Create two new Java Classes in main folder

Runner

Bear

*//Runner.java*

**class** **Runner{**

**public** **static** **void** **main(String[]** args**){**

**Bear** bear **=** **new** **Bear(**"Balu"**);**

**String** name **=** bear**.**getName**();**

**System.**out**.**println**(**name**);**

**}**

**}**

*//Bear.java*

**class** **Bear{**

**}**

Oh thank you compiler. You caught the problem, that there is no ‘getName’ method on a bear!

What do we need to give our bear this functionality.

* Constructor with one argument - the name
* Instance Variable
* Getter

**Instance Variables**

Instance variable are the state of our object. We want to give our bear a name instance variable. How would we do this in Ruby? (@name)

In Java, we put them at the top of our class. Remember we need to define a type.

*// Bear.java*

**class** **Bear{**

**String** name**;**

**}**

When we create the bear, how can we we assign it a name? Yes! Constructor function.

**Constructor Function**

Ruby, initialize

In Java we define a constructor function by writing a method with the same name as the class.

*//Bear.java*

**class** **Bear{**

**String** name**;**

**public** **Bear(String** name**){**

**this.**name **=** name**;**

**}**

**}**

Public allows this method to be accessed from outside the object.

**Getter**

*//Bear.java*

**class** **Bear{**

**String** name**;**

**public** **Bear(String** name**){**

**this.**name **=** name**;**

**}**

**public** **String** **getName(){**

**return** **this.**name**;**

**}**

**}**

Cool, now the compiler is happy with us and we can see the bear’s name. The getter is a typical Java method. We define the type of what a method returns. In this case a String.

**Setter**

What if we want to update the bear’s name after we have created the object? What do we need?

Yes! A setter!

*//Bear.java*

**class** **Bear{**

*//same as above*

**public** **void** **setName(String** newName**){**

**this.**name **=** newName**;**

**}**

**}**

Void? Even if a function doesn’t return anything, Java still expects us to tell it this.

The parameter, “newName”, needs to have it’s type defined. In this case a string.

[Task:]Alter bear’s name to “Baloo” using the setter

*//Runner.java*

**class** **Runner{**

**public** **static** **void** **main(String[]** args**){**

**Bear** bear **=** **new** **Bear(**"Balu"**);**

bear**.**setName**(**"Baloo"**);**

**String** name **=** bear**.**getName**();**

**System.**out**.**println**(**name**);**

**}**

**}**

**Public and Private**

We are going to show you something terrifying and awful.

*//Runner.java*

**class** **Runner{**

**public** **static** **void** **main(String[]** args**){**

**Bear** bear **=** **new** **Bear(**"Baloo"**);**

**System.**out**.**println**(**bear**.**name**);**

**}**

**}**

What!! We directly access the name of the bear. Even worse…

*//Runner.java*

**class** **Runner{**

**public** **static** **void** **main(String[]** args**){**

**Bear** bear **=** **new** **Bear(**"Baloo"**);**

**System.**out**.**println**(**bear**.**name**);**

bear**.**name **=** ""**;** *// UPDATED*

**System.**out**.**println**(**bear**.**name**);**

**}**

**}**

Defining our instance variables as private protects against unwanted access from outside. Define all your instance variables as private and create public getters and setters when needed. This allows for defensive programming.

*//Bear.java*

**class** **Bear{**

**private** **String** name**;**

*//same*

**}**

**Encapsulation**

What we have just gone through is one of the 4 key concepts of OO programming, Encapsulation.

Encapsulation in Java is a mechanism of wrapping the data (variables) and code acting on the data (methods) together as a single unit. In encapsulation, the variables of a class will be hidden from other classes, and can be accessed only through the methods of their current class. Therefore, it is also known as data hiding.

To achieve encapsulation in Java −

* Declare the variables of a class as private.
* Provide public setter and getter methods to modify and view the variables values.

**Benefits of Encapsulation**

* The fields of a class can be made read-only or write-only.
* A class can have total control over what is stored in its fields.
* The users of a class do not know how the class stores its data. A class can change the data type of a field and users of the class do not need to change any of their code.

**Types and Testing**

**Learning Objectives**

* Exposure to numeric types and booleans.
* Using JUnit to test.

**Testing with JUnit**

We’re going to start doing TDD just like we used to in Ruby. This will help us practice our new language and see how we can do the same thing we used to do, but in a Java friendly way.

Create new Maven project called ‘types\_testing’ Create a Bear class with property name and a getter

**public** **class** **Bear** **{**

**private** **String** name**;**

**public** **Bear(String** name**){**

**this.**name **=** name**;**

**}**

**public** **String** **getName(){**

**return** **this.**name**;**

**}**

**}**

Now we’re going to write some tests for our Bear class, and the convention is to name the test file the class name plus ‘Test’, so in this case BearTest.

To add test classes to our program we will create them in the test folder.

#IntelliJ

Expand test folder in project pane

Right Click on java folder

New > java Class

Name: BearTest

We need to import both the junit testing package and also the AssertEquals function. The classes we want to test also need to be available to this file, just like require\_relative in Ruby. In this case we’ll have all the files in the same directory so we don’t need to do anything, but when we come to packaging up our classes we need to make sure they’re imported.

Each test should be started by writing ‘@Test’. (This tells JUnit to run the following function as a test, so if any exceptions are thrown the test fails - this is a Java annotation, which can be used for a range of things, not just testing).

We then write a function which returns void and has a name descriptive of what is being tested.

*//BearTest.java*

**public** **class** **BearTest{**

**@Test**

**public** **void** **hasName(){**

**Bear** bear **=** **new** **Bear(**"Baloo"**);**

assertEquals**(** "Baloo"**,** bear**.**getName**()** **);**

**}**

**}**

You will notice two things here.

IntelliJ has auto imported the JUnit framework for us when we wrote the @Test annotation.

*//BearTest.java*

**import** org.junit.Test**;**

But assertEquals shows an error. This is because there are many assertEquals methods in JUnit and we need to specify where this is coming from.

Position the cursor somewhere in the assertEquals text and press alt+enter.

This shows us how to fix this error and offers us options to import the method.

junit.framework - For JUnit before version 4. org.junit - For JUnit versions 4 and after

We are using version 4 so we will select org.junit.

*//BearTest.java*

**import** static org**.**junit**.**Assert**.**assertEquals**;**

**import** org.junit.Test**;**

Here we have used assertEquals, which compares an expected value with the result of something, in this case the bear’s name with the result of the getName() function.

#IntelliJ

Open Bear.java

*//Bear.java*

**class** **Bear{**

**private** **String** name**;**

**public** **Bear(String** name**){**

**this.**name **=** name**;**

**}**

**public** **String** **getName(){**

**return** **this.**name**;**

**}**

**}**

**Running Tests with Junit in IntelliJ**

We can run these tests from IntelliJ in a similar way to how we ran the Runner class.

We can choose to run just the one test file by right clicking on the class and selecting Run or we can run all the test files in the java folder.

#IntelliJ

Right click on the java folder in test

Select Run 'All Tests'

**int**

Java gives us the int type to handle integer values. Let’s give the bear an age. We’ll follow TDD and write the test first.

*//BearTest.java*

**public** **class** **BearTest{**

**@Test**

**public** **void** **hasName(){**

**Bear** bear **=** **new** **Bear(**"Baloo"**);**

assertEquals**(** "Baloo"**,** bear**.**getName**()** **);**

**}**

**@Test**

**public** **void** **hasAge(){**

**Bear** bear **=** **new** **Bear(**"Baloo"**,** 25**);**

assertEquals**(** 25**,** bear**.**getAge**()** **);**

**}**

**}**

Solution:

*//Bear.java*

**class** **Bear{**

**private** **String** name**;**

**private** **int** age**;**

**public** **Bear(String** name**,** **int** age**){**

**this.**name **=** name**;**

**this.**age **=** age**;**

**}**

**public** **String** **getName(){**

**return** **this.**name**;**

**}**

**public** **int** **getAge(){**

**return** **this.**age**;**

**}**

**}**

But wait! We now get a compile error in our test class because we are creating a new Bear in our first test with just a name. Let’s alter our first test to pass in an age argument too.

*//BearTest.java*

**@Test**

**public** **void** **hasName(){**

**Bear** bear **=** **new** **Bear(**"Baloo"**,** 25**);**

assertEquals**(** "Baloo"**,** bear**.**getName**()** **);**

**}**

inevitably someone will ask why String is a capital and int is lowercase. This is going to be covered in the (quick) reference types lesson coming up next!

If you need a bigger number than int can hold, you can use long.

**double(float)**

For non-integer numbers Java gives us the double type. double stands for double precision float. They can contain twice the amount of data as a float. This allows for doubles to store larger numbers to more decimal points.

When we’re testing for a double, we should always include a third parameter in the assert for a delta, which is a range that the expected value can be lower or higher by. E.g. if we said 1.5, 0.01, it could be between 1.49 and 1.51.

Remember the delta for the double test!

*//BearTest.java*

**public** **class** **BearTest{**

*//AS BEFORE*

**@Test**

**public** **void** **hasWeight(){**

**Bear** bear **=** **new** **Bear(**"Baloo"**,** 25**,** 95.62**);**

assertEquals**(** 95.62**,** bear**.**getWeight**(),** 0.01 **);**

**}**

**}**

You might have noticed we’re being a bit repetitive by creating a new bear at the start of each test. To save doing this we can add in a setup step using @Before. This signals to JUnit a piece of code to run before each test is run.

when setting this up go over scope of class vs functions.

Now we can write the code to pass the Weight test.

*//BearTest.java*

*// Change import org.junit.Test to be org.junit.\**

**public** **class** **BearTest{**

**Bear** bear**;**

**@Before**

**public** **void** **before(){**

bear **=** **new** **Bear(**"Baloo"**,** 25**,** 95.62**);**

**}**

*//remove Bear setup from each test*

**}**

**}**

Solution

*//Bear.java*

**class** **Bear{**

**private** **String** name**;**

**private** **int** age**;**

**private** **double** weight**;**

**public** **Bear(String** name**,** **int** age**,** **double** weight**){**

**this.**name **=** name**;**

**this.**age **=** age**;**

**this.**weight **=** weight**;**

**}**

**public** **String** **getName(){**

**return** **this.**name**;**

**}**

**public** **int** **getAge(){**

**return** **this.**age**;**

**}**

**public** **double** **getWeight(){**

**return** **this.**weight**;**

**}**

**}**

**boolean**

Java gives us a boolean type to handle truthiness.

Unlike Ruby and Javascript where every object has a truthy or falsey value, Java is more uptight. If evaluating the truthiness of something it needs to be a conditional statement or a boolean object.

Create a readyToHibernate method based on weight. Let’s do the tests together:

*//BearTest.java*

**public** **class** **BearTest{**

*//AS BEFORE*

**@Test**

**public** **void** **readyToHibernateIfGreaterThan80(){**

assertEquals**(** **true,** bear**.**readyToHibernate**()** **);**

**}**

**@Test**

**public** **void** **notReadyToHibernateIfLessThan80(){**

**Bear** thinBear **=** **new** **Bear(**"Baloo"**,** 25**,** 65.44**);**

assertEquals**(** **false,** thinBear**.**readyToHibernate**()** **);**

**}**

**}**

To get this test passing we will use an if statement.

If statements in Java are very similar to every other language you have seen.

*//Bear.java*

**class** **Bear{**

*// AS BEFORE*

**public** **boolean** **readyToHibernate(){** *// NEW*

**if** **(this.**weight **>=** 80.00**){**

**return** **true;**

**}** **else** **{**

**return** **false;**

**}**

**}**

**}**

However as we only have 2 possible outcomes we could do this without the else ….

*//Bear.java*

**class** **Bear{**

*// AS BEFORE*

**public** **boolean** **readyToHibernate(){** *// NEW*

**if** **(this.**weight **>=** 80.00**){**

**return** **true;**

**}**

**return** **false;**

**}**

**}**

In this case if the condition is true the method will return true and the remainder of the method isn’t run.

**char**

The char type is used to store a single character. We use the SINGLE quotes to define the value.

**char** gender **=** 'f'**;**

Optional: Add a height (double) property. Add a BMI method (divide weight in kilograms (kg) by height in metres (m) then divide the answer by height again). Change readyToHibernate so it’s based on BMI > 30

**Quiz**

Select all statements where the declaration type does not match the value.

**char** gender **=** 'G'**;**

**boolean** fact **=** **true;**

**boolean** number **=** 17**;**

**String** name **=** "Yogi"**;**

**double** price **=** 17.89**;**

**long** total **=** 100.1**;**

**Solution**

number and total are wrong

**Types and Testing**

**Learning Objectives**

* Exposure to numeric types and booleans.
* Using JUnit to test.

**Testing with JUnit**

We’re going to start doing TDD just like we used to in Ruby. This will help us practice our new language and see how we can do the same thing we used to do, but in a Java friendly way.

Create new Maven project called ‘types\_testing’ Create a Bear class with property name and a getter

**public** **class** **Bear** **{**

**private** **String** name**;**

**public** **Bear(String** name**){**

**this.**name **=** name**;**

**}**

**public** **String** **getName(){**

**return** **this.**name**;**

**}**

**}**

Now we’re going to write some tests for our Bear class, and the convention is to name the test file the class name plus ‘Test’, so in this case BearTest.

To add test classes to our program we will create them in the test folder.

#IntelliJ

Expand test folder in project pane

Right Click on java folder

New > java Class

Name: BearTest

We need to import both the junit testing package and also the AssertEquals function. The classes we want to test also need to be available to this file, just like require\_relative in Ruby. In this case we’ll have all the files in the same directory so we don’t need to do anything, but when we come to packaging up our classes we need to make sure they’re imported.

Each test should be started by writing ‘@Test’. (This tells JUnit to run the following function as a test, so if any exceptions are thrown the test fails - this is a Java annotation, which can be used for a range of things, not just testing).

We then write a function which returns void and has a name descriptive of what is being tested.

*//BearTest.java*

**public** **class** **BearTest{**

**@Test**

**public** **void** **hasName(){**

**Bear** bear **=** **new** **Bear(**"Baloo"**);**

assertEquals**(** "Baloo"**,** bear**.**getName**()** **);**

**}**

**}**

You will notice two things here.

IntelliJ has auto imported the JUnit framework for us when we wrote the @Test annotation.

*//BearTest.java*

**import** org.junit.Test**;**

But assertEquals shows an error. This is because there are many assertEquals methods in JUnit and we need to specify where this is coming from.

Position the cursor somewhere in the assertEquals text and press alt+enter.

This shows us how to fix this error and offers us options to import the method.

junit.framework - For JUnit before version 4. org.junit - For JUnit versions 4 and after

We are using version 4 so we will select org.junit.

*//BearTest.java*

**import** static org**.**junit**.**Assert**.**assertEquals**;**

**import** org.junit.Test**;**

Here we have used assertEquals, which compares an expected value with the result of something, in this case the bear’s name with the result of the getName() function.

#IntelliJ

Open Bear.java

*//Bear.java*

**class** **Bear{**

**private** **String** name**;**

**public** **Bear(String** name**){**

**this.**name **=** name**;**

**}**

**public** **String** **getName(){**

**return** **this.**name**;**

**}**

**}**

**Running Tests with Junit in IntelliJ**

We can run these tests from IntelliJ in a similar way to how we ran the Runner class.

We can choose to run just the one test file by right clicking on the class and selecting Run or we can run all the test files in the java folder.

#IntelliJ

Right click on the java folder in test

Select Run 'All Tests'

**int**

Java gives us the int type to handle integer values. Let’s give the bear an age. We’ll follow TDD and write the test first.

*//BearTest.java*

**public** **class** **BearTest{**

**@Test**

**public** **void** **hasName(){**

**Bear** bear **=** **new** **Bear(**"Baloo"**);**

assertEquals**(** "Baloo"**,** bear**.**getName**()** **);**

**}**

**@Test**

**public** **void** **hasAge(){**

**Bear** bear **=** **new** **Bear(**"Baloo"**,** 25**);**

assertEquals**(** 25**,** bear**.**getAge**()** **);**

**}**

**}**

Solution:

*//Bear.java*

**class** **Bear{**

**private** **String** name**;**

**private** **int** age**;**

**public** **Bear(String** name**,** **int** age**){**

**this.**name **=** name**;**

**this.**age **=** age**;**

**}**

**public** **String** **getName(){**

**return** **this.**name**;**

**}**

**public** **int** **getAge(){**

**return** **this.**age**;**

**}**

**}**

But wait! We now get a compile error in our test class because we are creating a new Bear in our first test with just a name. Let’s alter our first test to pass in an age argument too.

*//BearTest.java*

**@Test**

**public** **void** **hasName(){**

**Bear** bear **=** **new** **Bear(**"Baloo"**,** 25**);**

assertEquals**(** "Baloo"**,** bear**.**getName**()** **);**

**}**

inevitably someone will ask why String is a capital and int is lowercase. This is going to be covered in the (quick) reference types lesson coming up next!

If you need a bigger number than int can hold, you can use long.

**double(float)**

For non-integer numbers Java gives us the double type. double stands for double precision float. They can contain twice the amount of data as a float. This allows for doubles to store larger numbers to more decimal points.

When we’re testing for a double, we should always include a third parameter in the assert for a delta, which is a range that the expected value can be lower or higher by. E.g. if we said 1.5, 0.01, it could be between 1.49 and 1.51.

Remember the delta for the double test!

*//BearTest.java*

**public** **class** **BearTest{**

*//AS BEFORE*

**@Test**

**public** **void** **hasWeight(){**

**Bear** bear **=** **new** **Bear(**"Baloo"**,** 25**,** 95.62**);**

assertEquals**(** 95.62**,** bear**.**getWeight**(),** 0.01 **);**

**}**

**}**

You might have noticed we’re being a bit repetitive by creating a new bear at the start of each test. To save doing this we can add in a setup step using @Before. This signals to JUnit a piece of code to run before each test is run.

when setting this up go over scope of class vs functions.

Now we can write the code to pass the Weight test.

*//BearTest.java*

*// Change import org.junit.Test to be org.junit.\**

**public** **class** **BearTest{**

**Bear** bear**;**

**@Before**

**public** **void** **before(){**

bear **=** **new** **Bear(**"Baloo"**,** 25**,** 95.62**);**

**}**

*//remove Bear setup from each test*

**}**

**}**

Solution

*//Bear.java*

**class** **Bear{**

**private** **String** name**;**

**private** **int** age**;**

**private** **double** weight**;**

**public** **Bear(String** name**,** **int** age**,** **double** weight**){**

**this.**name **=** name**;**

**this.**age **=** age**;**

**this.**weight **=** weight**;**

**}**

**public** **String** **getName(){**

**return** **this.**name**;**

**}**

**public** **int** **getAge(){**

**return** **this.**age**;**

**}**

**public** **double** **getWeight(){**

**return** **this.**weight**;**

**}**

**}**

**boolean**

Java gives us a boolean type to handle truthiness.

Unlike Ruby and Javascript where every object has a truthy or falsey value, Java is more uptight. If evaluating the truthiness of something it needs to be a conditional statement or a boolean object.

Create a readyToHibernate method based on weight. Let’s do the tests together:

*//BearTest.java*

**public** **class** **BearTest{**

*//AS BEFORE*

**@Test**

**public** **void** **readyToHibernateIfGreaterThan80(){**

assertEquals**(** **true,** bear**.**readyToHibernate**()** **);**

**}**

**@Test**

**public** **void** **notReadyToHibernateIfLessThan80(){**

**Bear** thinBear **=** **new** **Bear(**"Baloo"**,** 25**,** 65.44**);**

assertEquals**(** **false,** thinBear**.**readyToHibernate**()** **);**

**}**

**}**

To get this test passing we will use an if statement.

If statements in Java are very similar to every other language you have seen.

*//Bear.java*

**class** **Bear{**

*// AS BEFORE*

**public** **boolean** **readyToHibernate(){** *// NEW*

**if** **(this.**weight **>=** 80.00**){**

**return** **true;**

**}** **else** **{**

**return** **false;**

**}**

**}**

**}**

However as we only have 2 possible outcomes we could do this without the else ….

*//Bear.java*

**class** **Bear{**

*// AS BEFORE*

**public** **boolean** **readyToHibernate(){** *// NEW*

**if** **(this.**weight **>=** 80.00**){**

**return** **true;**

**}**

**return** **false;**

**}**

**}**

In this case if the condition is true the method will return true and the remainder of the method isn’t run.

**char**

The char type is used to store a single character. We use the SINGLE quotes to define the value.

**char** gender **=** 'f'**;**

Optional: Add a height (double) property. Add a BMI method (divide weight in kilograms (kg) by height in metres (m) then divide the answer by height again). Change readyToHibernate so it’s based on BMI > 30

**Quiz**

Select all statements where the declaration type does not match the value.

**char** gender **=** 'G'**;**

**boolean** fact **=** **true;**

**boolean** number **=** 17**;**

**String** name **=** "Yogi"**;**

**double** price **=** 17.89**;**

**long** total **=** 100.1**;**

**Solution**

number and total are wrong

**Primitive Types v Reference Types**

You might have been wondering why in previous lessons we spelled String with a capital S but int, double and boolean with a lowercase first letter. This is because that’s Java’s way of differentiating between primitive and other types.

In Java there are eight primitive types. You might also hear these called value types. This is because all they do is hold a value, and have no other methods available.

The types are:

* boolean (for true/false)
* char (for unicode single characters, use single quotes)
* byte, short, int, long (all for integers/whole numbers)
* float, double (for float/decimal point numbers)

All have a set amount of memory they take up.

In Ruby / JS conversion of number sizes is done automatically and the appropriate amount of memory is allocated.

Info table for reference: http://docstore.mik.ua/orelly/java-ent/jnut/ch02\_04.htm )

Any other data types are actually Java classes which have extra functionality built in. We’ve already seen the String class, other examples are arrays and HashMaps. These will all start with a capital letter like any other class, and have methods like length() or toUppercase() or isEmpty().

These other data types are known as reference types. The big difference between these and primitive/value types is that when you assign a variable to one of these they just make a REFERENCE to the information, rather than actually HOLDING the value themselves. This is important because if you make a copy of a reference type variable, such as when you pass some data in as an argument to a function, it is only copying a new reference to the same data, rather than making a clone of the data, which happens if you pass a primitive type in as an argument. Therefore if you do anything to change the data using this copied reference, it also changes the original data set, which can cause problems.

Link to docs for anyone who’s particularly interested: http://docstore.mik.ua/orelly/java-ent/jnut/ch02\_10.htm

# Single Class - minilab (45 mins)

Create a new directory ‘planets’

Convert the following Ruby class into Java and get it to print out the same information. (Hint: You will need a Runner class with a main method).

class Planet

def initialize(name, size)

@name = name

@size = size

end

def get\_name()

return @name

end

def get\_size()

return @size

end

def explode()

puts "Boom! #{@name} has exploded."

end

end

mars = Planet.new("Mars", 908973)

puts "#{mars.get\_name()} is #{mars.get\_size()}"

mars.explode()

# Homework - Simple Classes

Create 3 single classes with methods to perform some functionality.

This is a TDD exercise. Write the test first then create the method to get the test passing.

## MVP:

#### Calculator

* Create a Calculator class. This should have functions for Add, Subtract, Multiply and Divide. Your methods should take in two ints to perform the calculations on (except the Divide method. This should take two doubles as arguments).

#### Water Bottle

* Create a water bottle class with a volume property.
* The volume should start at 100.
* Add a drink function that takes 10 from the volume each time it is called.
* Create an empty function that brings the volume down to 0.
* Create a fill function that fills the volume back to 100.

## Extension:

#### Printer

* Create a Printer class that has a property for number of sheets of paper left.
* Add a method to print that takes in a number of pages and number of copies.
* The print method will only run if the printer has enough paper. If it runs it will reduce the value of the paper left by number of copies \* number of pages.
* Add a toner volume property to the class.
* Modify the printer so that it reduces the toner by 1 for each page printed.

# Basic Arrays

### Learning Objectives

* Know how to create a basic array
* Understand null
* Be able to use arrays in Java
* Know how to create an ArrayList in Java
* Be able to use ArrayLists in Java
* Understand standard and enhanced for loops
* be able to use for loops with ArrayLists

## Intro

|  |  |
| --- | --- |
| Draw up ArrayList | Array table |
|  | | Array | ArrayList |
| **Need to specify type of contents?** | | **Yes** | **Yes** |
| **Need to specify size?** | | **Yes** | **No** |
| **How a length property?** | | **Yes** | **No** |
| **Has a size method?** | | **No** | **Yes** |
| **Has useful methods?** | | **No** | **Yes** |
| **Can change size once declared?** | | **No** | **Yes** |

In this lesson we are going to have a look at two types of collections in Java. The most basic Java collection - the Array and a more enhanced collection - the ArrayList. We’ve encountered arrays before and now we are going to see how they work in Java. They are exactly the same in principle - a container to hold a set of items.

In Ruby and many other dynamically typed languages, we could put a mixture of things into an array:

*# irb*

myArray **=** [1,2,3,"banana", **true**];

In Java world, we can’t do this. We’re going to have a look at why this is the case in this lesson.

We need to declare the type of things that go into the array, followed by square brackets then the name of the array variable. This means that if we create an array of Strings then only Strings can be stored in the array.

The main concern with arrays is that we need to specify the number of items that can be stored in the array. This must be done when we initialise the array and can’t be changed after.

Let’s make one of these and see it in action.

Create new project called ‘arrays’

Create class WordCollection

*//WordCollection.java*

**class** **WordCollection** **{**

**private** **String[]** words**;**

**public** **WordCollection(){**

**this.**words **=** **new** **String[**5**];**

**}**

**}**

So here we have an array of Strings that will only hold 5 string objects inside.

Let’s write a test to get the number of words in this collection and see what’s going on with our array.

Create WordCollectionTest ```java public class WordCollectionTest {

private WordCollection myWords;

@Before public void before(){ myWords = new WordCollection(); }

@Test public void canGetWordCount(){ assertEquals(0, myWords.getWordCount()); } }

In the `WordCollectionTest` we have a test to get the word count back. Uncomment this and let's make it pass.

Arrays don't have many methods attached to them like we had in Ruby. It does however have a length property that we can use.

Let's write the method for this.

```java

//WordCollection.java

class WordCollection {

private String[] words;

public int getWordCount(){

return this.words.length;

}

}

Run the test and see what happens….

Expected 5?? But our array should be empty?

This is because we have specified the length as 5 so Java arrays will automatically store something in the 5 spaces.

So what is actually in the array is [null, null, null, null, null]

null is a special java type meaning nothing. (This is not the same as integer 0 or an empty string as these have a specific type. Nulls have no type.)

In Java, a variable is a reference to an object. A null value indicates an unset reference (i.e. a reference to nothing).

So this could be a problem when it comes to getting the size of the array accurately. We would need conditions for if the entry was null.

Not having methods on arrays has a big effect on how we manipulate them.

They have no add method so we would need to write this ourselves…. to do this we need to specify the index to add it to.

*//WordCollection*

**public** **class** **WordCollection** **{**

**public** **void** **add(String** word**){**

**this.**words**[**0**]** **=** word**;** *// Added*

**}**

**}**

Also when we add we are always setting the first element in the array so we would need some kind of counter to check if the entry was null and if so add it at that index. Messy!!

Also we don’t have any methods so remove an item so we would have to write that ourselves as well.

The reason we are showing you this is that arrays are still used in some cases to store data where you know the exact size of the collection, but very rarely.

Some Java methods do return Arrays so it is likely that you will still encounter them. Such methods as Enum.values(), String.split(), main method all return arrays.

When we are creating a collection of our own though the more standard collection to use is an ArrayList.

Comment out the test in WordCollectionTest so that it doesn’t affect next section.

## ArrayLists

The ArrayList class is an out of the box Java class that implements the List interface so we can do a lot more with it. ArrayList supports dynamic arrays that can grow as needed.

Array lists are created with an initial size. When this size is exceeded, the collection is automatically enlarged. When objects are removed, the array may be shrunk.

Lets create another class to use these and a test.

#IntelliJ

Create a class called NumberList

Create a test called NumberListTest

When we create a new ArrayList we need to specify again what type is held in the ArrayList. This is done within angular brackets <>. Again only this type can be stored in here.

ArrayLists cannot hold primitive types so we couldn’t do something like ArrayList<int>. However, as we looked at in Reference types, there is normally a class associated to primitive types. So Integer for ints, Double for double etc. So ArrayList would be fine.

*//NumberList.java*

**import** java.util.ArrayList**;**

**public** **class** **NumberList** **{**

**private** **ArrayList<Integer>** numbers**;**

**public** **NumberList(){**

**this.**numbers **=** **new** **ArrayList<>();**

**}**

**}**

Yep, ArrayLists live in the java.util “namespace”. Namespaces are just a way of bundling up code, like Ruby modules. Util is indeed a shocking name and it should really be “collections” or something. Don’t be like Java. Don’t call stuff util, or utils. For me. Please.

In the constructor we have to create a new instance of the ArrayList. We can’t simply say something like numbers = [] like we could in Ruby because the ArrayList is a Class. Like any other class we write we have to create a new instance.

Lets test getting the number of entries from the ArrayList.

We can do this using the size() method attached to ArrayList class.

*// NumberListTest*

**import** org.junit.Before**;**

**import** org.junit.Test**;**

**import** static org**.**junit**.**Assert**.**assertEquals**;**

**public** **class** **NumberListTest** **{**

**private** **NumberList** myNumbers**;**

**@Before**

**public** **void** **before(){**

myNumbers **=** **new** **NumberList();**

**}**

**@Test**

**public** **void** **hasNumberOfEntries(){**

assertEquals**(**0**,** myNumbers**.**getNumberCount**());**

**}**

**}**

Task Write the method so that the test passes.

*// NumberList*

**import** java.util.ArrayList**;**

**public** **class** **NumberList** **{**

**private** **ArrayList<String>** numbers**;**

**public** **NumberList(){**

**this.**numbers **=** **new** **ArrayList<>();**

**}**

**public** **int** **getNumberCount()** **{**

**return** **this.**numbers**.**size**();**

**}** *// NEW*

**}**

Sweet our test passes!

Lets add a number to the list.

*// NumberListTest*

**import** org.junit.Before**;**

**import** org.junit.Test**;**

**import** static org**.**junit**.**Assert**.**assertEquals**;**

**public** **class** **NumberListTest** **{**

*//AS Before*

**@Test**

**public** **void** **canAddNumberToList(){**

myNumbers**.**addNumber**(**12**);**

assertEquals**(**1**,** myNumbers**.**getNumberCount**());**

**}**

**}**

Task: Write the method to add to the ArrayList.

Investigate the methods on the ArrayList class to see if there is anything to help you here.

*// NumberList*

**import** java.util.ArrayList**;**

**public** **class** **NumberList** **{**

*//As Before*

**public** **void** **addNumber(int** number**){**

**this.**numbers**.**add**(**number**);**

**}**

**}**

Awesome we can now add to the ArrayList.

Every time we add to the list it will always put the new number at the next index number.

Lets see if we can get the first number back out.

*// NumberListTest*

**import** org.junit.Before**;**

**import** org.junit.Test**;**

**import** static org**.**junit**.**Assert**.**assertEquals**;**

**public** **class** **NumberListTest** **{**

*//AS Before*

**@Test**

**public** **void** **canGetFirstNumber(){**

myNumbers**.**addNumber**(**12**);**

assertEquals**(**12**,** myNumbers**.**getNumberAtIndex**(**0**));**

**}**

**}**

Task: Write the method to get the number from the ArrayList.

*// NumberList*

**import** java.util.ArrayList**;**

**public** **class** **NumberList** **{**

*//As Before*

**public** **int** **getNumberAtIndex(int** index**){**

**return** **this.**numbers**.**get**(**index**);**

**}**

**}**

So we can get the number at a specific index.

There are numerous methods we can use on ArrayLists:

* clear(). Removes all of the elements from this list.
* contains(Object o). Returns true if this list contains the specified element.
* remove(int index). Removes and returns the element at the specified position in this list.

## Loops

A for loop is a repetition control structure that allows you to efficiently write a loop that needs to be executed a specific number of times.

A for loop is useful when you know how many times a task is to be repeated.

Let’s say we wanted to get the total of all the numbers in our ArrayList.

*// NumberListTest*

**public** **class** **NumberListTest** **{**

*//AS Before*

**@Test**

**public** **void** **canGetTotal(){**

myNumbers**.**addNumber**(**1**);**

myNumbers**.**addNumber**(**2**);**

myNumbers**.**addNumber**(**3**);**

myNumbers**.**addNumber**(**4**);**

assertEquals**(**10**,** myNumbers**.**getTotal**());**

**}**

**}**

The first kind of loop to look at is a standard for loop.

The syntax of a for loop is −

**for(**initialization**;** **Boolean** expression**;** update**)** **{**

*// Statements*

**}**

So let’s try using this loop to get the total.

*// NumberList.java*

**public** **class** **NumberList{**

*// AS BEFORE*

**public** **int** **getTotal(){**

**int** total **=** 0**;**

**for** **(int** i **=** 0**;** i **<** getNumberCount**();** i**++){**

total **+=** getNumberAtIndex**(**i**);**

**}**

**return** total**;**

**}**

**}**

So here we are setting an initial counter i to 0. The loop will run as long as i is less than the number of entries in the list. After each loop runs the value of i is increased by 1.

i can now be used as our index for each item in the list.

Run the test….

Sweet it passes.

Is there maybe an easier way though?

Yes there is an enhanced for loop which will look more familiar to you.

The syntax for the enhanced for loop is as follows:

**for(**type name **:** listName**){**

**}**

Looks similar to the Ruby loop doesn’t it?

This is the most common type of loop used with collections.

Let’s amend our getTotal() method.

**public** **class** **NumberList{**

*// AS BEFORE*

**public** **int** **getTotal(){**

**int** total **=** 0**;**

**for** **(int** number **:** **this.**numbers**){**

total **+=** number**;**

**}**

**return** total**;**

**}**

**}**

Ah, that’s better.

## Passing lists into Constructors

Ok so before we created a new ArrayList in the constructor. But what if we wanted to create new instance of our NumberList class with a pre-populated list of numbers.

Well we can create a new ArrayList in our test and pass it to the NumberList when we create the new instance. Just like we did with the Bear and passed it a name, age and weight.

In the test class we will change the before method to do this.

*// NumberListTest.java*

**public** **class** **NumberListTest** **{**

**private** **NumberList** myNumbers**;**

**@Before**

**public** **void** **before(){**

**ArrayList<Integer>** testNumbers **=** **new** **ArrayList<>();** *// NEW*

**}**

**}**

Now let’s add some numbers to this list.

*// NumberListTest.java*

**public** **class** **NumberListTest** **{**

**private** **NumberList** myNumbers**;**

**@Before**

**public** **void** **before(){**

**ArrayList<Integer>** testNumbers **=** **new** **ArrayList<>();**

testNumbers**.**add**(**1**);** *// NEW*

testNumbers**.**add**(**2**);** *// NEW*

testNumbers**.**add**(**3**);** *// NEW*

testNumbers**.**add**(**4**);** *// NEW*

**}**

**}**

And finally we will pass it to the NumberList class.

*// NumberListTest.java*

**public** **class** **NumberListTest** **{**

**private** **NumberList** myNumbers**;**

**@Before**

**public** **void** **before(){**

**ArrayList<Integer>** testNumbers **=** **new** **ArrayList<>();**

testNumbers**.**add**(**1**);**

testNumbers**.**add**(**2**);**

testNumbers**.**add**(**3**);**

testNumbers**.**add**(**4**);**

myNumbers **=** **new** **NumberList(**testNumbers**);**

**}**

**}**

We will also need to amend some of our previous tests now as our list will now start with 4 numbers.

*// NumberListTest.java*

**public** **class** **NumberListTest** **{**

*// AS BEFORE*

**@Test**

**public** **void** **hasNumberOfEntries(){**

assertEquals**(**4**,** myNumbers**.**getNumberCount**());** *// MODIFIED*

**}**

**@Test**

**public** **void** **canAddNumberToList(){**

myNumbers**.**addNumber**(**12**);**

assertEquals**(**5**,** myNumbers**.**getNumberCount**());** *// MODIFIED*

**}**

**@Test**

**public** **void** **canGetFirstNumber(){**

*// DELETED addNumber*

assertEquals**(**1**,** myNumbers**.**getNumberAtIndex**(**0**));** *// MODIFIED*

**}**

**@Test**

**public** **void** **canGetTotal(){**

assertEquals**(**10**,** myNumbers**.**getTotal**());** *// MODIFIED*

**}**

**}**

Ok so we are good to go. Now we will go and change our class to take in the list and assign it to the instance variable.

*// NumberList.java*

**public** **class** **NumberList{**

**private** **ArrayList<Integer>** numbers**;**

**public** **NumberList(ArrayList<Integer>** numbers**){**

**this.**numbers **=** numbers**;**

**}**

**}**

So we have created an ArrayList with the values 1,2,3,4 in it. Passed it to the NumberList class and then assigned that same list to the numbers property of the class.

Don’t get confused by the fact that we are declaring ArrayList<Integer> in the constructor parameters. We need to let java know what kind of thing is being passed in.

Run the tests and we should now be all green!

## Summary

While arrays still have their place in Java and you may see them in use the preferred collection to use is ArrayList<>.

We can perform more functions on an ArrayList without having to write the specific methods ourselves.

ArrayLists also grow in size dynamically.

ArrayLists, like Array, can only hold one specific type of Object. (String, Integer, etc). But this can also include our own objects (e.g. Bear).

# Multiple Classes

### Learning Objectives

* Be able to create multiple classes.
* Be able to create a class with a collection of another class type.
* Be able to create collections of different classes.

## Intro

In this lesson we are going to have a look at creating multiple classes and allowing one class to store a collection of the second class in an ArrayList.

Create a project called BearRiverFish

Create a Bear Test class

**public** **class** **BearTest** **{**

**private** **Bear** bear**;**

**@Before**

**public** **void** **before()** **{**

bear **=** **new** **Bear(**"Baloo"**);**

**}**

**}**

Create a Bear class

**public** **class** **Bear** **{**

**private** **String** name**;**

**public** **Bear(String** name**)** **{**

**this.**name **=** name**;**

**}**

**}**

Create an empty Salmon class

**public** **class** **Salmon** **{**

**}**

## Declaring an Array List

First we need to tell our code that we want to use the ArrayList class, in a similar way that we use require in Ruby.

*//Bear.java*

**import** java.util.ArrayList**;**

**public** **class** **Bear** **{**

**private** **String** name**;**

**private** **ArrayList<Salmon>** belly**;** *//UPDATED*

**public** **Bear(String** name**){**

**this.**belly **=** **new** **ArrayList<Salmon>();** *//UPDATED*

**this.**name **=** name**;**

**}**

**}**

So now we have given our bear an ArrayList belly that can hold instances of our Salmon objects.

### Counting the items

We want to see how much food is in the bear’s belly - initially it should be zero.

Let’s write the test

*//BearTest.java*

**@Test**

**public** **void** **bellyStartsEmpty(){**

assertEquals**(**0**,** bear**.**foodCount**());**

**}**

And now we will write a method in the Bear class to return an int value with the size of the belly.

*//Bear.java*

**public** **int** **foodCount(){**

**return** belly**.**size**();**

**}**

### Eating a salmon

Let’s try to get our bear to eat a salmon. First, let’s add a test salmon to our test file.

*//BearTest.java*

**import** static org**.**junit**.**Assert**.**assertEquals**;**

**import** org.junit.\***;**

**public** **class** **BearTest** **{**

**private** **Bear** bear**;**

**private** **Salmon** salmon**;** *//NEW*

**@Before**

**public** **void** **before(){**

bear **=** **new** **Bear(**"Baloo"**);**

salmon **=** **new** **Salmon();** *//NEW*

**}**

*//same as before*

**}**

Cool, now we can use this salmon in our tests.

*//BearTest.java*

**@Test**

**public** **void** **canEatSalmon(){**

bear**.**eat**(**salmon**);**

assertEquals**(**1**,** bear**.**foodCount**());**

**}**

Let’s add the eat() method. For this we will use the add() method from the ArrayList class to add a salmon to the belly.

*//Bear.java*

**public** **void** **eat(Salmon** salmon**){**

belly**.**add**(**salmon**);**

**}**

### Resetting the array

Lastly, let’s add a way for our Bear to go to sleep and let his belly settle. What we want to happen here is that when the bear sleeps the belly will empty.

*//BearTest.java*

**@Test**

**public** **void** **shouldEmptyBellyAfterSleeping(){**

bear**.**eat**(**salmon**);**

bear**.**sleep**();**

assertEquals**(**bear**.**foodCount**(),** 0**);**

**}**

And lets add the sleep method in the bear class. For this we can use the clear() method from the ArrayList class to empty the list.

*//Bear.java*

**public** **void** **sleep(){**

belly**.**clear**();**

**}**

Cool our Bear is a happy little chappy now.

### Adding a third class

So lets extend this out a bit more now and add a third class of River to interact with the other two.

#Intellij

Create a new class called River and a test called TestRiver

Our River is going to contain the salmon and when a bear eats we will get it to take a salmon from the river.

*// River.java*

**import** java.util.ArrayList**;**

**public** **class** **River** **{**

**private** **ArrayList<Salmon>** fish**;**

**public** **River()** **{**

**this.**fish **=** **new** **ArrayList<>();**

**}**

**}**

Lets write some tests for our River. Lets start by adding some Salmon to the river.

*// RiverTest.java*

**import** org.junit.Before**;**

**import** org.junit.Test**;**

**import** static org**.**junit**.**Assert**.**assertEquals**;**

**public** **class** **RiverTest** **{**

**private** **River** river**;**

**private** **Salmon** salmon**;**

**@Before**

**public** **void** **before(){**

river **=** **new** **River();**

salmon **=** **new** **Salmon();**

**}**

**@Test**

**public** **void** **canAddSalmon(){**

river**.**addFish**(**salmon**);**

assertEquals**(**1**,** river**.**fishCount**());**

**}**

**}**

Now lets get the test to pass.

*// River.java*

*//As Before*

**public** **void** **addFish(Salmon** salmon**){**

**this.**fish**.**add**(**salmon**);**

**}**

**public** **int** **fishCount()** **{**

**return** **this.**fish**.**size**();**

**}**

Now lets test that we can remove and get a Salmon back out of the River.

*// RiverTest.java*

**import** org.junit.Before**;**

**import** org.junit.Test**;**

**import** static org**.**junit**.**Assert**.**assertEquals**;**

**public** **class** **RiverTest** **{**

*//As Before*

**@Test**

**public** **void** **canGetSalmon(){**

river**.**addFish**(**salmon**);**

river**.**removeFish**();**

assertEquals**(**0**,** river**.**fishCount**());**

**}**

**}**

*// River.java*

*//As Before*

**public** **Salmon** **removeFish()** **{**

**return** **this.**fish**.**remove**(**0**);**

**}**

Cool so we can get a Salmon back from the River. So how do we get the Bear to get this salmon from the River?

Discuss with the students and hope they get to the below solution

So when the bear eats now we will pass in a river for it to eat from.

To make what is happening clear we will change the name of our eat() method in Bear to something more explicit. So we will now call it eatFishFromRiver().

Lets amend our bear test to set this up.

*// BearTest.java*

**public** **class** **BearTest** **{**

**private** **Bear** bear**;**

**private** **Salmon** salmon**;**

**private** **River** river**;** *//ADDED*

**@Before**

**public** **void** **before(){**

bear **=** **new** **Bear(**"Baloo"**);**

salmon **=** **new** **Salmon();**

river **=** **new** **River();** *// ADDED*

river**.**addFish**(**salmon**);** *// ADDED*

**}**

**@Test**

**public** **void** **bellyStartsEmpty(){**

assertEquals**(**0**,** bear**.**foodCount**());**

**}**

**@Test**

**public** **void** **canEatSalmon(){**

bear**.**eatFishFromRiver**(**river**);** *//AMENDED*

assertEquals**(**0**,** river**.**fishCount**());**

**}**

**@Test**

**public** **void** **shouldEmptyBellyAfterSleeping(){**

bear**.**eatFishFromRiver**(**river**);** *//AMENDED*

bear**.**sleep**();**

assertEquals**(**bear**.**foodCount**(),** 0**);**

**}**

**}**

Task: Amend the Bears eat function to be called eatFishFromRiver and make it so that it takes in a river and adds a salmon from the river to its belly.

*// Bear.java*

*//As Before*

**public** **void** **eatFishFromRiver(River** river**){** *//AMENDED*

**Salmon** salmon **=** river**.**removeFish**();** *//ADDED*

belly**.**add**(**salmon**);**

**}**

Great stuff! So now our bear gets the salmon from the River and the river deals with removing and returning the salmon.

The way that we have done this works well as the bear doesn’t have access to the rivers collection of salmon directly to amend it.

So the River should be the only class responsible for removing a fish from its ArrayList of fish. The river is responsible for amending its collection of salmon and not the bear. The Bear class shouldn’t directly manipulate any of the properties in the River, like its fish count.

## SOLID: Single Responsibility

In our example, Bear should be only responsible for Bear related information and actions. Just as the River should be only responsible for river related information and actions.

**Single Responsibility principle states that one class or method should do just one simple job**.

To achieve Single Responsibility we require decoupling where classes are not closely entangled and only depend on each other when they need to. Another important part of Single Responsibility is encapsulation where all classes have their inner variables set to private and only enable editing them via public methods, such as getters and setters. These two practices enable exact definition of which class is responsible for which functionality.

Lack of Single Responsibility complicates:

* **maintenance of code** - especially where different functionality is programmed by different departments. Programmers can accidentally break other people’s code.
* **deployment** - if a class has many responsibilities, it is more often changed, so needs more frequent releases and updates. Which complicates maintenance of other code that depend on this class.
* **testing** - in a multi-responsibility class there are more possible fringe scenarios that that will need testing.

Look at the below method in the Bear class:

**public** **void** **eatFishFromRiver(River** river**){**

**Salmon** salmon **=** river**.**removeFish**();**

belly**.**add**(**salmon**);**

**}**

The bear object is communicating with other objects, but only for the purposes of bear-related actions, like eating a fish. And it’s the river is responsible for amending its collection of salmon and not the bear.

This depicts a correct **decoupling** of classes (separating of Bear and River functionality) and **encapsulation** of class variables (only River being able to change River’s own array of fish). These combined produce a correct example of **Single Responsibility**.

## Summary

We have seen how to

* Create multiple classes in Java.
* Create a class with a collection of another class type.
* Create ArrayLists, and add and remove items from them.
* What the Single Responsibility principle is

**Lab**

Create a TDD model of a bus with passengers.

**MVP**

* Create a Bus class with a Destination, Capacity and an initially empty ArrayList passengers of type Person
* Create a method to return the number of passengers on the bus.
* Create a method to add a passenger onto the bus only if the passenger count is less than the capacity.
* Create a method to remove a passenger from the bus.

**Extensions:**

* Add a BusStop class which interacts with the other two.
* BusStop should have a name and an initially empty ArrayList of type “Person” called Queue.
* Add a method to add a Person to the queue.
* Add a method to remove a person from the queue.
* Create a pickUp method in bus that gets person from bus stop and adds to bus passengers.

# HashMaps

Duration - 30 minutes

## Learning Objectives

* Describe the purpose of a HashMap
* Use some important HashMap methods
* Use a HashMap in your projects

### Intro

When we were learning Ruby, we saw how useful it was to store keys and values in a structured way - using hashes:

db\_details **=** { dbname: "pizza\_shop", host: "localhost" }

It allowed us to store and retrieve data without having to worry about the order of the data.

Most languages have a similar construct, and in Java, these are called HashMaps. (In other languages they might be called hashes, dictionaries, or associative arrays.)

Let’s look at how we would initialise a HashMap in Java.

Create a new project in IntelliJ and call it HashMapDemo.

Create a new Java class in the main package again called HashMapDemo.

*// HashMapDemo.java*

**import** java.util.HashMap**;**

**public** **class** **HashMapDemo** **{**

**public** **static** **void** **main(String[]** args**)** **{**

**HashMap** favouriteFruits **=** **new** **HashMap();**

favouriteFruits**.**put**(**"Alice"**,** "Apple"**);**

favouriteFruits**.**put**(**"Sarah"**,** "Banana"**);**

favouriteFruits**.**put**(**"Bob"**,** "Strawberry"**);**

**System.**out**.**println**(**favouriteFruits**.**get**(**"Alice"**));**

**}**

**}**

So this works OK; the program outputs Alice’s favourite fruit as expected.

Notice that we’re initialising the HashMap as being empty, then using the .put() method to add keys and values.

However, if we build our project, rather than just running it, you might notice that the compiler is giving us a warning.

Choose Build > Build Project

Note: If the warnings don’t appear as expected, it can be introduced by doing the following:

* Introduce a blatant error
* Build > Build Project
* The errors list will appear
* Fix the blatant error
* Build > Build Project
* Cmd + 0 (Command + zero)
* Should now be able to see the warning from the compiler.

Note: HashMapDemo.java uses unchecked or unsafe operations.

The compiler is warning us that we should specify the types of the keys and values we are putting into the HashMap. Let’s try again.

*// HashMapDemo.java*

**import** java.util.HashMap**;**

**public** **class** **HashMapDemo** **{**

**public** **static** **void** **main(String[]** args**)** **{**

**HashMap<String,** **String>** favouriteFruits **=** **new** **HashMap<String,** **String>();**

favouriteFruits**.**put**(**"Alice"**,** "Apple"**);**

favouriteFruits**.**put**(**"Sarah"**,** "Banana"**);**

favouriteFruits**.**put**(**"Bob"**,** "Strawberry"**);**

**System.**out**.**println**(**favouriteFruits**.**get**(**"Alice"**));**

**}**

**}**

Much better! Now Java will complain loudly if we try to set the key or value to anything other than a String.

## Keys

A note about HashMap keys: you can use any class as a key, provided that it implements the .equals() and .hashCode() methods. In the example above, String fits the bill, but we can use any class that implements these two methods.

## Values

When you store a value in a HashMap, it will always store an object, rather than a primitive type. Take a look at the following code.

**import** java.util.HashMap**;**

**public** **class** **HashMapDemo** **{**

**public** **static** **void** **main(String[]** args**)** **{**

**HashMap<String,** **Integer>** ages **=** **new** **HashMap<String,** **Integer>();**

ages**.**put**(**"Alice"**,** 52**);**

ages**.**put**(**"Bob"**,** 24**);**

**Integer** aliceAge **=** ages**.**get**(**"Alice"**);**

**String** output **=** "Alice's age is " **+** aliceAge**.**toString**();**

**System.**out**.**println**(**output**);**

**}**

**}**

Because the value of Alice’s age is a full integer object, we can call toString() on it. (We couldn’t do this if it was a primitive type!)

## Methods

Let’s take a look at some of the most common methods we can call on our HashMap:

favouriteFruits**.**put**(**key**,** value**)** *// inserts a new entry into the HashMap*

favouriteFruits**.**get**(**key**)** *// gets the value for the given key*

favouriteFruits**.**size**()** *// returns the size of the HashMap as an integer*

favouriteFruits**.**clear**()** *// clears all entries from the HashMap*

favouriteFruits**.**containsValue**(**value**)** *// returns true if the HashMap contains the value*

favouriteFruits**.**remove**(**key**)** *//removes the entry with the given key*

### Task

1. Create a HashMap of keys and values for the populations of some countries. Here is some sample data (don’t forget to think about the types of your keys and values!):

UK: 64,100,000

Germany: 80,620,000

France: 66,030,000

Japan: 127,300,000

1. Output some values from the HashMap using .get(key) and system.out.println().
2. Investigate the use of the .values() and .keySet() methods on HashMap.

# Homework

Create your own library class with an internal collection of books.

## MVP

* Books should have title, author and genre.
* Write a method to count the number of books in the library.
* Write a method to add a book to the library stock.
* Add a capacity to the library and write a method to check if stock is full before adding a book.

## Extensions:

* Add a third class which interacts with the other two. E.g. you could add a Borrower with a method that takes a Book and moves to the Borrower’s collection.

## Advanced Extension

* The library wants to keep track of it’s number of books by genre. Using a HashMap, store the genre of each book as the key - and a count of how many books of that genre as the value.

# Homework

Create your own library class with an internal collection of books.

## MVP

* Books should have title, author and genre.
* Write a method to count the number of books in the library.
* Write a method to add a book to the library stock.
* Add a capacity to the library and write a method to check if stock is full before adding a book.

## Extensions:

* Add a third class which interacts with the other two. E.g. you could add a Borrower with a method that takes a Book and moves to the Borrower’s collection.

## Advanced Extension

* The library wants to keep track of it’s number of books by genre. Using a HashMap, store the genre of each book as the key - and a count of how many books of that genre as the value.

# CodeClan Towers

You are being asked to build a booking system for the new CodeClan Towers hotel.

Use TDD in Junit to model the Hotel with Java classes, with separate test files for each class.

## MVP

* Create a Guest class to represent a visitor to the hotel, they’ll at least need a name, you can add more properties later if and when they become necessary
* Create 2 different types of rooms:
  + Bedrooms will have a room number, capacity, collection of Guests and a type (e.g. Single/Double.)
  + ConferenceRooms will have a capacity, collection of Guests, name and any other properties you wish.
* Create a Hotel class, which has collections of rooms of different types.
* The Hotel will be able check guests in/out of rooms.

### Extensions

* Create a Booking class which contains a Bedroom and a number of nights booked.
* Create a bookRoom method in your Hotel. This should book a given Bedroom for a number of nights. This should return a new Booking object.
* Add a nightly rate to your Bedrooms and write a method to return the total bill for the Booking.
* Add a DiningRoom class with a name, and collection of guests
* Hotel will have a HashMap based collection of DiningRooms.
* Hint ^

</details>

### Advanced Extensions

* Add functionality to the Hotel so it can return a collection of only the vacant Bedrooms.
* Update the check-in process so that Hotel will only be able to check guests into empty Bedrooms.
* Any other extensions you can think of!

# ntroduction to Debugging in IDE

## Learning Objectives

* Know how to debug a Java application
* Understand breakpoints
* Understand the debugger console
* Know how to evaluate code fragments

## Task

As an introduction to debugging:

* watch the following video: [Intro to Debugging video](https://youtu.be/ErVZrVWZrko)

Answer the following questions:

1. What is the purpose of a breakpoint?
2. Does the line of code on a breakpoint run when you start debugging?
3. How do we debug the next line of code?
4. What does the step into command do?
5. What is the difference between evaluate expression and evaluate code fragment?

# Enums

### Objectives

* Know what an enum is
* Be able to use an enum
* Know how to give values to enums
* Know how to add fields and methods to an enum

### Duration - 45 minutes

# Intro

Previously, we have often used “magic strings” to represent sets of possible properties. For example, our bank account might be “personal” or “business”. An order status might be “received”, “dispatched” or “processing”.

Let’s create an example

Create a project called Enums

Create a CardTest class

**public** **class** **CardTest** **{**

**Card** card**;**

**@Before**

**public** **void** **before(){**

card **=** **new** **Card(**"Hearts"**);**

**}**

**@Test**

**public** **void** **canGetSuit(){**

assertEquals**(**"Hearts"**,**card**.**getSuit**());**

**}**

**}**

Create a Card class

**public** **class** **Card** **{**

**private** **String** suit**;**

**public** **Card(String** suit**)** **{**

**this.**suit **=** suit**;**

**}**

**public** **String** **getSuit(){**

**return** **this.**suit**;**

**}**

**}**

Let’s add another test.

*//CardTest.java*

**@Test**

**public** **void** **suitCanBeMispelled(){**

card **=** **new** **Card(**"Heeaarts"**);**

assertEquals**(**"Heeaarts"**,** card**.**getSuit**());**

**}**

If we were to write a method on a card game to find a card of a given suit, it would match “Hearts” but not “Heeaarts”. What if we added it as lowercase, uppercase? It becomes a nightmare.

Let’s try another little test.

*//CardTest.java*

**@Test**

**public** **void** **suitCanBeBananas(){**

card **=** **new** **Card(**"Bananas"**);**

assertEquals**(**"Bananas"**,** card**.**getSuit**());**

**}**

Uh, we’ve managed to set Bananas as the suit for the Card… that’s not great either. Cards don’t tend to come in Banana… unless you’ve get a set for a particular game.

If we use strings, we can’t stop users passing invalid values.

This is where enums come in. Enums allow us to define a set of possible values, and nothing outside of that set is permitted. This is great news for searching and stopping unexpected things happening.

//IntelliJ

Create a new class in the java folder called SuitType.java and set its type to enum from drop down.

Right click > new > Java Class.

name class SuitType and in drop down change to Enum.

*//SuitType.java*

**public** **enum** **SuitType** **{**

HEARTS**,**

DIAMONDS**,**

SPADES**,**

CLUBS

**}**

The “enum” keyword sits where we used to declare a class. An enum is different, it has no properties and you’re restricted to enum-specific methods. It simply acts as a container of values we can use.

We tend to use uppercase for the value names.

Let’s refactor our Card to use the SuitType enum.

*//Card.java*

**public** **class** **Card** **{**

**private** **SuitType** suit**;** *//UPDATED*

**public** **Card(SuitType** suit**)** **{** *//UPDATED*

**this.**suit **=** suit**;**

**}**

**public** **SuitType** **getSuit(){** *//UPDATED*

**return** **this.**suit**;**

**}**

**}**

Now our tests won’t compile, because we are throwing around strings all over the place. We’ve got a bit of work to do. Let’s comment out the banana and misspell test for now.

*//CardTest.java*

**import** static org**.**junit**.**Assert**.\*;**

**import** org.junit.\***;**

**public** **class** **CardTest** **{**

**Card** card**;**

**@Before**

**public** **void** **before(){**

card **=** **new** **Card(**"Hearts"**);**

**}**

**@Test**

**public** **void** **canGetSuit(){**

assertEquals**(**"Hearts"**,** card**.**getSuit**());**

**}**

*//@Test*

*//public void suitCanBeMispelled(){*

*// card = new Card("Heeaarts");*

*// assertEquals("Heeaarts", card.getSuit());*

*//}*

*//@Test*

*//public void suitCanBeBananas(){*

*// card = new Card("Bananas");*

*// assertEquals("Bananas", card.getSuit());*

*//}*

**}**

Let’s go about fixing our test. We need to use our Enum instead of the string, since an enum declares a new type, in our case SuitType. SuitType is it’s own thing, not a String or an int or a Card. Just like a class behaves.

To use our shiny new enum, we need to use the enum name then the key we want to access.

*//CardTest.java*

**import** static org**.**junit**.**Assert**.\*;**

**import** org.junit.\***;**

**public** **class** **CardTest** **{**

**Card** card**;**

**@Before**

**public** **void** **before(){**

card **=** **new** **Card(SuitType.**HEARTS**);** *//UPDATED*

**}**

**@Test**

**public** **void** **canGetSuit(){**

assertEquals**(**"Hearts"**,** card**.**getSuit**());**

**}**

*// same as before*

**}**

This will still fail, since our test is comparing it with the string “Hearts”. One way to fix this is to call toString() on the value, but this negates the point of using an enum. We can actually just compare Enum values directly using the type itself.

*//CardTest.java*

**import** static org**.**junit**.**Assert**.\*;**

**import** org.junit.\***;**

**public** **class** **CardTest** **{**

**Card** card**;**

**@Before**

**public** **void** **before(){**

card **=** **new** **Card(SuitType.**HEARTS**);** *//UPDATED*

**}**

**@Test**

**public** **void** **canGetSuit(){**

assertEquals**(SuitType.**HEARTS**,** card**.**getSuit**());**

**}**

*// same as before*

**}**

Cool we are all good!

Enums are extremely powerful for giving us more control over our code and what gets passed to our methods.

[TASK:] Create an enum for card ‘rank’ (ACE, ONE, TWO, etc) that can be passed to the card

[SOLUTION:] Create an Enum file called RankType:

*// RankType.java*

**public** **enum** **RankType** **{**

ACE**,**

TWO**,**

THREE**,**

FOUR**,**

FIVE**,**

SIX**,**

SEVEN**,**

EIGHT**,**

NINE**,**

TEN**,**

JACK**,**

QUEEN**,**

KING

**}**

Then modify the Card class:

*//Card.java*

**public** **class** **Card** **{**

**private** **SuitType** suit**;**

**private** **RankType** rank**;** *//UPDATED*

**public** **Card(SuitType** suit**,** **RankType** rank**)** **{** *//UPDATED*

**this.**suit **=** suit**;**

**this.**rank **=** rank**;**

**}**

*//same as before*

**public** **RankType** **getRank()** **{** *//UPDATED*

**return** **this.**rank**;** *//UPDATED*

**}**

**}**

Finally, include the new Enum in the existing test file.

*//CardTest.java*

**public** **class** **CardTest** **{**

**Card** card**;**

**@Before**

**public** **void** **before(){**

card **=** **new** **Card(SuitType.**HEARTS**,** **RankType.**QUEEN**);** *//UPDATED*

**}**

**@Test** *//UPDATED*

**public** **void** **canGetRank(){** *//UPDATED*

assertEquals**(RankType.**QUEEN**,** card**.**getRank**());** *//UPDATED*

**}**

**}**

# Enum Fields and Methods

One of the things you can do is add a value to an enum, known as a **field**. This means that we can create something which looks like a key-value pair in our enum. For example, say we wanted to associate each item in our RankType enum (e.g. ‘KING’) with the value (e.g. 10) for that card. We could add some kind of switch statement wherever we use the enum in our code, but we can do something like this inside our enum.

When we add such a value to our enum, then we also need to add a getter to the enum, so that we can get this value back.

So let’s write a test that, say, if we have a Queen then it’s value should be 10.

*//CardTest.java*

**@Test**

**public** **void** **queenHasValue10(){**

card **=** **new** **Card(SuitType.**HEARTS**,** **RankType.**QUEEN**);**

assertEquals**(**10**,** card**.**getValueFromEnum**());**

**}**

So we are testing a getValueFromEnum() method on our Card class. This method is going to call a ‘getter’ on our enum. Let’s add the getValueFromEnum() method now:

*//Card.java*

**public** **int** **getValueFromEnum()** **{** *//ADDED METHOD*

**return** **this.**rank**.**getValue**();**

**}**

So now we need to add our value to our field. We can add a variable (like an instance variable in a class) to our enum:

*//RankType.java*

**public** **enum** **RankType** **{**

ACE**,**

TWO**,**

THREE**,**

FOUR**,**

FIVE**,**

SIX**,**

SEVEN**,**

EIGHT**,**

NINE**,**

TEN**,**

JACK**,**

QUEEN**,**

KING**;** *//MODIFIED TO ADD SEMI-COLON*

**private** **final** **int** value**;** *//ADDED*

**}**

note that we now need to add a semi-colon after the final entry in the enum list.

This value is declared as final, which means that it will never change.

As our enum has a field, we now need to create a constructor. This constructor takes an argument, which is used to set the value of the field.

*//RankType.java*

**public** **enum** **RankType** **{**

ACE**,**

TWO**,**

THREE**,**

FOUR**,**

FIVE**,**

SIX**,**

SEVEN**,**

EIGHT**,**

NINE**,**

TEN**,**

JACK**,**

QUEEN**,**

KING**;**

**private** **final** **int** value**;**

**RankType(int** value**)** **{** *//ADDED*

**this.**value **=** value**;**

**}**

**}**

So we are passing a value to our constructor. But where does this value come from? How do we call this constructor?

When we add an item to our enum, the constructor is called for that item. Since our constructor takes a value we then need to put that value in round brackets after the item we are adding e.g. if we are creating an item in our enum called QUEEN with the value 10 then we would have:

This means that when we add this item to our enum, it will call the constructor for the enum item QUEEN, passing it the value 10. We can then do the same for the other items in our enum e.g.:

*//RankType.java*

**public** **enum** **RankType** **{**

ACE**(**1**),** *//UPDATED all values for Enum*

TWO**(**2**),**

THREE**(**3**),**

FOUR**(**4**),**

FIVE**(**5**),**

SIX**(**6**),**

SEVEN**(**7**),**

EIGHT**(**8**),**

NINE**(**9**),**

TEN**(**10**),**

JACK**(**10**),**

QUEEN**(**10**),**

KING**(**10**);**

**private** **final** **int** value**;**

**RankType(int** value**)** **{**

**this.**value **=** value**;**

**}**

**}**

We can now add a getter for our value, which is an enum-specific method. This will simply return the value associated with the item in the enum:

*//RankType.java*

**public** **enum** **RankType** **{**

ACE**(**1**),**

TWO**(**2**),**

THREE**(**3**),**

FOUR**(**4**),**

FIVE**(**5**),**

SIX**(**6**),**

SEVEN**(**7**),**

EIGHT**(**8**),**

NINE**(**9**),**

TEN**(**10**),**

JACK**(**10**),**

QUEEN**(**10**),**

KING**(**10**);**

**private** **final** **int** value**;**

**RankType(int** value**)** **{**

**this.**value **=** value**;**

**}**

**public** **int** **getValue()** **{** *//UPDATED*

**return** **this.**value**;** *//UPDATED*

**}**

**}**

Our test should now pass.

You can now see that our enum is starting to look like a class. In fact an enum is a special type of a Java class.

# Other useful things

Say wanted to get a list of all the entries in our enum. We can do this using the values method e.g.:

**SuitType[]** suits **=** **SuitType.**values**();**

This returns an array of all the items in the enum. We could then do useful things with this array, like loop through it.

## How can enums help us?

Enums are useful when we we know in advance a variable will have a small set of possible values.

Ask the class to suggest some examples

Some examples -

* Days of the week - MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY
* Directions on a compass - NORTH, EAST, SOUTH, WEST
* Menu options - FILE, EDIT, VIEW, HISTORY, BOOKMARKS

An example where an enum **would not** work -

* Names on a Passport - ALLY, SANDY, ALEX, FINN

Only four names are allowed! If our client the passport office wants to allow an additional name we’d have to make a new build for them! We’d have millions of new feature requests - one for each new name!

Enums help us detect errors before we run our code - when we compile it. In more technical terms - we’ll detect errors at compile-time, not run-time. We saw when we used strings for our Suit that errors occurred when we **ran** our tests and not before.

We also get nice auto-completion in IntelliJ to remind us what the allowed values are. And warnings when we try and use an undefined value -

*//CardTest.java - add anywhere to demo*

**Card** brokenCard **=** **new** **Card(SuitType.**STARS**,** **RankType.**STUDENT**);**

*// won't compile - IntelliJ colours STARS and STUDENT red*

An added bonus is that our code is documenting itself - helping others understand how they should use it and what values we allow.

## Recap

* An enum is a special type of data type which is basically a collection (set) of constants.
* An enum allows a variable to be a set of pre-defined constants
* An enum can contain constants, which allow us to associate a specific value for element in an enum, and methods.

### Card Game Lab

This lab is to model a basic card game and implement the rules.

We aren’t looking for a running game, just good TDD to demonstrate that the game works as it should.

#### Highest Card

### MVP

* Create a Deck class with an ArrayList of cards. (Deck should start of empty)
* Come up with a method to populate the deck of cards.
* Find a way to shuffle the cards.
* Create a method to deal a card from the deck.

#### Extensions

Build a simple card game that has a dealer deal one card to every player and the player with the highest card value in their hand wins.

## Considerations / Restrictions.

1. Think about how to model a deck of cards. What are the constituent parts?
   * A deck which contains 52 cards. (What kind of data structure best models this?).
   * A card with a suit and a value. (Given the suits and number of cards is fixed, this could be a job for enums).
   * Can you think of a way to ‘populate’ the deck with cards? (A nested loop?).
2. Model the game.
   * Think about how to model the rules of the game (determine which hand wins etc).
   * Think about how to implement the logistics of the game (deal cards, gather the players’ totals, etc).
3. Use test driven development. Fully test your model.

## Weekend Homework - Blackjack

:star: This is a PDA Homework :star:

Extend your High/Low card game Using your existing code (fork it!) simulate a game of “BlackJack”.

To start off with you should be working in a TDD fashion and creating tests for your classes.

## MVP

* Deal two cards to a dealer and a player
* Compare the hands
* Determine the winner from who has the highest value hand

## Extension

* Allow the player to “twist” or “stick” (Player go bust if hand value exceeds 21 and they automatically lose the round).
* Dealer will twist if hand < 16
* Compare hands once both dealer and player have stuck.
* Allow for more players to play.

### BlackJack rules

* Aces may be counted as 1 or 11 points, 2 to 9 according to card value, and tens and face cards count as ten points.
* The value of a hand is the sum of the point values of the individual cards.
* Except, a “blackjack” is the highest hand, consisting of an ace and any 10-point card, and it outranks all other 21-point hands.
* To start dealer will give two cards to each player and two cards to himself. One of the dealer cards is dealt face up.
* Play begins with the player to the dealer’s left. The following are the choices available to the player:
  + Stand: Player stands pat with his cards.
  + Twist: Player draws another card (and more if he wishes). If this card causes the player’s total points to exceed 21 (known as “breaking” or “busting”) then he loses.
* After each player has had his turn, the dealer will turn over his hole card. If the dealer has 16 or less, then he will draw another card.
* If the dealer goes over 21 points, then any player who didn’t already bust will win.
* If the dealer does not bust, then the higher point total between the player and dealer will win.

### PDA Reminder:

As part of this homework you are required to take screenshots of the following:

- An example of encapsulation in a program.

Demonstrate testing in your program. Take screenshots of:

- Example of test code

- The test code failing to pass

- Example of the test code once errors have been corrected

- The test code passing

* Go to your [PDA Checklist](https://github.com/codeclan/pda/tree/master/Evidence%20Gathering%20Portfolio)
* Submit your PDA evidence (screenshots, etc.) to your own PDA repo
* PDA Reference: I.T 1, P18

# Airline!

Your task is to model a system for the world renowned online travel booking agent **TravelJava**. You should use the tools you have learnt this week where appropriate and useful. Remember to **TDD**!

TravelJava doesn’t really exist (sadly) - don’t stress about how a “real” example of this would work differently. This is just an exercise to practice some different concepts.

**Assumptions**:

* Each passenger bag weighs the same
* Planes reserve half of their total weight for passenger bags
* The weight of bag per person is the weight reserved for passenger bags divided by the capacity
* Passengers exist for a single flight only

## MVP

Create a Passenger class which has:

* a name
* a number of bags

Create a Plane class which has:

* an enum PlaneType (e.g. BOEING747) which stores capacity and total weight

Create a Flight class which has:

* an empty list of booked Passenger’s
* a Plane
* flight number (i.e. “FR756”)
* destination (i.e. GLA, EDI)
* departure airport (i.e. GLA, EDI)
* departure time (use a String)

The Flight class should have methods to:

* return the number of available seats
* book a passenger (if there are remaining seats)

### Extensions

Create a FlightManager class which has methods to:

* calculate how much baggage weight should be reserved for each passenger for a flight
* calculate how much baggage weight is booked by passengers of a flight
* calculate how much overall weight reserved for baggage remains for a flight

### More Extensions

* Refactor the Flight’s departure time to use the Date class (HINT: Look into **Type Migration** in IntelliJ to refactor faster)
* Add a ‘flight’ property to the Passenger class which is assigned when a passenger is added to a flight
* Add a ‘seat number’ property to the Passenger class as an integer. Set it to a random number which is assigned when a Passenger is booked on a flight
* Make sure the flight doesn’t double book the same seat number to more than one passenger

### Super Extensions

* Write a method in FlightManager to sort the passengers of a Flight by seat number by implementing a Bubble Sort
* Write a method in FlightManager that uses a Binary Search to find a Passenger by seat number. Remember to use the previous method to sort the list first

### PDA Reminder:

As part of this homework you are required to take screenshots of the following:

- An example of encapsulation in a program.

Demonstrate testing in your program. Take screenshots of:

- Example of test code

- The test code failing to pass

- Example of the test code once errors have been corrected

- The test code passing

* Go to your [PDA Checklist](https://github.com/codeclan/pda/tree/master/Evidence%20Gathering%20Portfolio)
* Submit your PDA evidence (screenshots, etc.) to your own PDA repo
* PDA Reference: I.T 1, P18