**Inheritance**

**Learning objectives**

* Understand how inheritance works in Java
* Understand the super keyword usage
* Override methods from parent class
* Create an Inheritance chain

**Intro**

When we were learning Ruby, we saw how useful it was to gather classes together when they share properties or functionality. We used *inheritance* to describe the relationship between superclasses and subclasses.

Scan the room for blank expressions. Have a quick recap of inheritance in Ruby if necessary.

**class** **Vehicle**

**...**

**def** **start\_engine**

"Vrmmmm!"

**end**

**end**

**class** **Car** **<** Vehicle

**end**

The same concept exists in Java, and although the syntax is a little different, the idea is much the same. We can use the ‘extends’ keyword to use inheritance.

Lets use inheritance to model a system to track CodeClan instructors and students.

Hand out Starting point

So here we have a tested classes for an Instructor and Student at CodeClan

We can change both names and the cohort they are attached to.

All of this seems a bit wasteful though as we are having to code everything twice even though they are doing essentially the same thing.

A better way would be to have both of these classes inherit from a parent class of Person.

Create a Person class and a test file to go with it.

*//PersonTest.java*

**Person** person**;**

**@Before**

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

person **=** **new** **Person(**"Darren"**,** "E16"**);**

**}**

**@Test**

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

assertEquals**(**"Darren"**,** person**.**getName**());**

**}**

**@Test**

**public** **void** **hasCohort(){**

assertEquals**(**"E16"**,** person**.**getCohort**());**

**}**

**@Test**

**public** **void** **canChangeName(){**

person**.**setName**(**"Tony"**);**

assertEquals**(**"Tony"**,** person**.**getName**());**

**}**

**@Test**

**public** **void** **canChangeCohort(){**

person**.**setCohort**(**"E19"**);**

assertEquals**(**"E19"**,** person**.**getCohort**());**

**}**

**@Test**

**public** **void** **canTalk(){**

assertEquals**(**"I love Java"**,** person**.**talk**(**"Java"**));**

**}**

**Task:** Create the person class and get the tests to pass.

Ok so now we have a Person class we can change our Instructor and Student classes to inherit from it.

We can remove all the methods and move to the Person class.

To inherit in Java we use the extends keyword.

Lets start with the Instructor.

*// Instructor.java*

**public** **class** **Instructor** **extends** **Person** **{**

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

**private** **String** cohort**;**

**public** **Instructor(String** name**,** **String** cohort**)** **{**

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

**this.**cohort **=** cohort**;**

**}**

**}**

Ok so we now get an error that there is no default constructor in Person.

What is happening here is that we have inherited from person and when we do that we need to take the arguments passed into the constructor in Instructor and pass them to the Person class to assign the details.

We do this using the super() keyword and pass the properties we want to use in to this.

We can also now remove the properties we defined at top of the class.

*// Instructor.java*

**public** **class** **Instructor** **extends** **Person** **{**

**public** **Instructor(String** name**,** **String** cohort**)** **{**

**super(**name**,** cohort**);**

**}**

**}**

So how does this work??

When we create a new instance of Instructor we pass in the name and cohort. As Instructor extends from Person, Java will try and make a new instance of Person to work with. As Person’s constructor expects the name and cohort we can simply pass those up the chain and assign to the new Person instance that is attached to our Instructor.

As Instructor extends from Person we can now access all of the properties and methods in the Person class from Instructor.

Don’t believe me? Run the tests again.

What?? They still pass! Awesome.

**Task:** Change the Student class to inherit from Person and re-run tests to make sure it all works.

*//Student.java*

**public** **class** **Student** **extends** **Person{**

**public** **Student(String** name**,** **String** cohort**)** **{**

**super(**name**,** cohort**);**

**}**

**}**

Cool. It doesn’t just have to be Getters and Setters that the child classes can access. We could add any method into Person and it would become available to call from Instructor and Student.

Any methods written in Instructor or Student though will only be available to that class.

Ok so all good here. But what about if we wanted to add in an extra property just for the Instructor. Say a module team?

So we don’t want to add this to Person as students wouldn’t be members of the module team. We can add this to the Instructor class only.

*//Instructor.java*

**public** **class** **Instructor** **extends** **Person** **{**

**private** **String** moduleTeam**;**

*// As Before*

**}**

And lets change the constructor to take this in and assign it.

To do this we need to ensure that we call super first before assigning the moduleTeam value.

*//Instructor.java*

**private** **String** moduleTeam**;**

**public** **Instructor(String** name**,** **String** cohort**,** **String** moduleTeam**)** **{**

**super(**name**,** cohort**);**

**this.**moduleTeam **=** moduleTeam**;**

**}**

*//As Before*

And change our test to pass in the team name.

*//InstructorTest.java*

**public** **class** **InstructorTest** **{**

**Instructor** instructor**;**

**@Before**

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

instructor **=** **new** **Instructor(**"Ally"**,** "G3"**,** "JavaScript"**);**

**}**

**}**

*//As Before*

**Task:** Write tests for a getter and setter for moduleTeam in Instructor class.

**Overriding methods**

We can also override any methods from the parent class.

Lets add in a talk method to the the Student class and override it.

*//Student.java*

**public** **class** **Student** **extends** **Person{**

*//As Before*

**public** **String** **talk(String** language**){**

**return** "I love learning " **+** language**;**

**}**

**}**

When we do this you will notice a small o with an arrow next to the method. This indicates that we are now overriding the method from Person.

Now if we run the StudentTest you will see that the canTalk() test fails as it is expecting the string value that we have created in Student.

**Task:** Change the test in StudentTest to make it pass.

*//StudentTest.java*

**@Test**

**public** **void** **canTalk(){**

assertEquals**(**"I love learning Java"**,** student**.**talk**(**"Java"**));**

**}**

Neat huh? When we call student.talk() it will look in the Student class for a talk() method. If it doesn’t find one there it will go to next level up the inheritance chain to Person and look there and call that if it’s found.

**Types of inheritance**

So far, we have been using single inheritance, where our Instructor class inherits from a single parent, Person. We can also create an *inheritance chain*, where a class inherits from more than one layer of parent classes. For example:

*//Person.java*

**public** **class** **Person** **{**

**public** **Person(String** name**,** **String** cohort**){**

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

**this.**cohort **=** cohort**;**

**}**

**}**

*//Instructor.java*

**public** **class** **Instructor** **extends** **Person** **{**

**public** **Instructor(String** name**,** **String** cohort**,** **String** moduleTeam**){**

**super(**name**,** cohort**);**

**this.**moduleTeam **=** moduleTeam**;**

**}**

**}**

*//SeniorInstructor.java*

**public** **class** **SeniorInstructor** **extends** **Instructor** **{**

**private** **String** hiringTeam**;**

**public** **SeniorInstructor(String** name**,** **String** cohort**,** **String** moduleTeam**,** **String** hiringTeam**){**

**super(**name**,** cohort**,** moduleTeam**);**

**this.**hiringTeam **=** hiringTeam**;**

**}**

**}**

Because SeniorInstructor inherits from Instructor and Instructor inherits from Person this means that SeniorInstructor still has access to the properties and methods on Person through this inheritance chain.

However, we can never inherit directly from more than one parent class. This would be illegal:

*//SeniorInstructor.java*

**public** **class** **SeniorInstructor** **extends** **Instructor,** **Assessor** **{**

**}**

**SOLID: Liskov Substitution**

What we have just done adheres to the Liskov Substitution principle.

This principle states that if you substitute a parent class with a child class then it shouldn’t break your application.

So in this instance we are able to replace an instance of Person with an instance of either Student or instructor and there are no issues.

If it broke the application then Liskov would be very unhappy!

Instructor: Write up Liskov Substitution on the board.

**Task:** (Optional) - 20 minutes

Create your own inheritance chain for modelling various types of animal.

* Your base class should be Animal, and should have methods for eat() and breathe().
* Create a subclass of Animal called Mammal. Mammals should have a talk() method.
* Create two further subclasses of Mammal: Human and Chimpanzee. They should have a walk() method.

Make each method return a suitable string. Check that you can create a human and a chimpanzee object, and make sure that they can each eat, breathe, walk, and talk.

**Abstract Classes**

Lesson Duration: 60-75 minutes

**Learning Objectives**

* Understand what an abstract class is
* Be able to create an abstract class
* Understand that abstract classes cannot be instantiated
* Understand what an abstract method is
* Understand how abstract classes can be used as a type
* Understand that in inheritance parent classes can be abstract, but don’t have to

**Intro**

**What are abstract classes for?**

You learned that classes are blueprints for creating objects. But sometimes we want to have a class that we will never create an object of, because it is too vague, too abstract.

Some examples of that would be:

* a Person class could be a superclass of Student and Instructor but we never really want to create an instance of Person in itself.
* a Shape class could be a superclass of Square and Circle but we never really want to create an instance of Shape in itself.

This is where the idea of an Abstract class type comes in. On a practical level an abstract class is a class that is never instantiated - we will never create an object of that class’s type. However, we will often use abstract classes as a parents for other classes (via inheritance).

Give out the starter code and let the students read it for 2-5ish minutes and maybe ask students to draw class diagrams

UML: draw class diagrams for the starting code and refer to if throughout the lesson.

**Starting Code**

We have a class of Car and Motorbike classes that inherit from a parent Vehicle class.

* Vehicle has numberOfWheels and a getter for it
* Vehicle has String drivingInstructions() which gives some instructions, but these are very simple and could be not enough.
* Vehicle has String startEngine() which makes a sound of starting an engine.
* Car and Motorbike extend Vehicle
* Because all vehicles share certain behaviour in terms of driving instructions, Car and Motorbike override Vehicle’s method drivingInstructions. Both classes use parent’s implementation (with super.drivingInstructions()) as well as add their own specific information.
* Car has numberOfDoors and a getter for it
* Car has String openDoors() method

optional: below is a note on how child constructors use parent constructors:

Note that parent class Vehicle has a constructor with model and number of wheels, that is used by subclasses Car and Motorbike in such way that they provide default value for numberOfWheels. Car and Motorbike have their own constructors, which take other attributes (eg. numberOfDoors) but do not take numberOfWheels for which they will provide defaults.

Also note that new Car("Audi",2) calls Car’s constructor, not Vehicle’s, so the number 2 is number of doors, not of wheels.

All three classes have test files to accompany them.

**What If Classes Are Too ‘Abstract’ To Be Instantiated?**

Notice that at the moment we can create a new instance of Vehicle class (eg. in VehicleTest). But unfortunately in our scenario the driving instructions are not quite detailed enough - they do not tell us how to steer something that is a Vehicle.

Like above, it happens sometimes in our code that a parent class is Abstract when it would not make much sense for it to be instantiated on its own.

For example in a graphical application you could have an abstract Shape class extended by Square, Triangle and Circle. While we can create these three shapes and add it to our drawing, it would make no sense to add a Shape. Same as you can create and interact with a Chicken running around in your game, but its parent class Animal is too abstract to have it instantiated.

It’s the same with Vehicle and its children.

We have two needs that will be fulfilled by two OO methods:

* we want to have the ability to implement common code in Vehicle that can be then used in subclasses using super. calls (by other classes inheriting Vehicle)
* we want to make sure that we can’t instantiate the Vehicle class (by making Vehicle abstract)

**Making a class abstract**

As described above Vehicle should be abstract. We can make a class abstract by using the abstract keyword,

UML: In class diagrams the notation for abstract classes is putting their name in italics (eg. *Vehicle*). Since this is very hard to distinguish in handwriting, you can also put <<ABSTRACT>> marker next to your class name. It is up to you what method you use, as long as the diagrams help you to plan and communicate with your team

*//Vehicle.java*

**public** **abstract** **class** **Vehicle** **{** *//MODIFIED*

**public** **String** **drivingInstructions(){**

**return** "Turn the key to start."**;**

**}**

**}**

Notice that IntelliJ now complains that we can’t create in instance of the Vehicle class inside of out VehicleTest:

Error:(14, 29) java: Vehicle is abstract; cannot be instantiated

**Can we test abstract classes?**

Since it is not possible to create new instances of an abstract class, it is literally impossible to test if they work. This is why we do not write test files for abstract classes, but instead we test their behaviours in non-abstract child classes that extend these abstract parent classes.

In our example we’ll need to comment out the whole VehicleTest class. We’ll come back to it later.

Comment out all tests in VehicleTest class.

Note: While there exist special ways to test some behaviours of abstract classes (like creating a dummy for-testing-only implementation class VehicleImplementation), we do not teach that on this course.

**Abstract Methods**

**Non-abstract Methods**

Note that abstract classes still benefit from all benefits of inheritance. All methods and variable of parent classes are still available to abstract class’ children. In your starting code, there are two methods defined by the parent class Vehicle:

* drivingInstructions() is defined in parent and overridden by subclasses and it’s functionality is extended by using super calls.
* startEngine() is is defined in parent and not modified by subclasses, but can be still used by the,

When extending abstract classes we get all the normal “benefits” of inheritance still apply. We can share properties across all of our objects, e.g. type of fuel, color and other information that every vehicle has.

**Abstract Methods as Promises**

You already know that a class can be abstract - it cannot be initialised and serves as a blueprint for other classes. You can follow a similar pattern for methods.

The new concept is an abstract method: it is a ‘promise’ of a method, that all subclasses need to implement. It is used when the parent needs to ensure that all children can do something, but parent does not want to provide a default way to do it.

* abstract method *HAS TO* be defined in every subclass
* abstract methods have no method body (default implementation) - body will be defined in all subclasses.

Optional: explain what a method body is:

What’s method body? Have a look at this method: (just write this on a whiteboard)

*//method with method body*

**public** **double** **divide(double** divident**,** **double** divisor**)** **{**

**return** divident **/** divisor**;**

**}**

*//abstract method with no method body*

**public** **double** **divide(double** divident**,** **double** divisor**);**

you can then point to parts of method declaration on whiteboard describe parts of method definition

* public - Scope
* double - Return value
* divide - function name
* (double divident, double divisor) - arguments’ types and names
* { ... } - method body, the actual code that this method will run
* ; - if your method does not have a body, it can end with a ;. It means that this method will have a body defined elsewhere. You’ll see this in Abstract methods and later in Interfaces.

**Using Abstract Methods**

UML: on whiteboard add boardingInstructions() to Vehicle class

Let’s add an abstract method boardingInstructions() to our Vehicle class:

*//Vehicle.java*

**public** **abstract** **class** **Vehicle** **{**

**public** **String** **drivingInstructions(){**

**return** "Turn the key to start."**;**

**}**

**public** **String** **boardingInstructions();** *//MODIFIED*

**}**

Note that this method does not contain any body. What we are basically saying is that any class which inherits from Vehicle must implement a boardingInstructions() method which takes no parameters and returns a string. It doesn’t matter what the method does but it must be called boardingInstructions(), take the declared number and type of parameters and return a the declared return type. In our example it takes no parameters and returns a String.

If we were to try an compile our code now we will get an error saying that the subclasses of Vehicle do not implement the boardingInstructions() method. So lets’s go and fix this now.

UML: on whiteboard add boardingInstructions() to Car and Motorbike classes

Now that we have our class diagram, let’s write the tests:

Note that when we write tests IntelliJ’s auto complete will already suggest us the abstract method boardingInstructions() on a class Car even thou it was not implemented yet. That’s because it will have to be implemented at some point.

*//CarTest.java*

**@Test**

**public** **void** **hasBoardingInstructions()** **{**

assertEquals**(**"Enter via one of the 4 doors."**,** car**.**boardingInstructions**());**

**}**

*//MotorbikeTest.java*

**@Test**

**public** **void** **hasBoardingInstructions()** **{**

assertEquals**(**"Just hop on."**,** motorbike**.**boardingInstructions**());**

**}**

Now let’s add the implementation of this method in the Car and Motorbike classes to remove method not implemented error.

*//Car.java*

**public** **class** **Car** **extends** **Vehicle** **{**

**public** **String** **boardingInstructions(){**

**return** "Enter via one of the " **+** numberOfDoors **+** " doors."**;**

**}**

**}**

Now let’s fix the Motorbike class so it compiles and the test passes

Task for students: Fix the Motorbike class so it compiles and the test passes

SOLUTION:

*//Motorbike.java*

**public** **class** **Motorbike** **extends** **Vehicle** **{**

**public** **String** **boardingInstructions(){**

**return** "Just hop on."**;**

**}**

**}**

**Using Abstract Class as a Type**

**Keeping benefits of inheritance**

Abstract parent class is still a parent class. We get all the useful things that came with inheritance. Let’s come back to our VehicleTest.java and let’s uncomment these tests that still make sense when the parent class is abstract.

Go into VehicleTest.java and uncomment bottom 5 tests (all below the ` //BELOW: Inheritance related tests:` marker)

UNCOMMENT

private Vehicle vehicle;

LEAVE COMMENTED:

// public void before()

// public void vehicleHasNumberOfWheels()

// public void vehicleHasModel()

// public void hasDrivingInstructions()

UNCOMMENT:

public void carAsVehicle() //UNCOMMENT

public void motorbikeAsVehicle() //UNCOMMENT

public void changeTypeOfVehicle()//UNCOMMENT

public void collectionOfParentClassObjects() //UNCOMMENT

public void objectRemembersItsType() //UNCOMMENT

**Using the type of abstract parent as a variable type**

As you remember, with inheritance we could declare a variable of a parent type, and put an object of a child object in there.

Although we cannot create an instance of an abstract class, we are allowed to declare a variable of its type. When assigning a value to that variable we have to assign it with an instance of one of its subclasses.

UML: refer to class diagrams when you describe the below

Scroll to this code in VehicleTest.java. Code is already in start code, no need to type it: There will be an error, but we’ll get to it in a minute

We can:

* Declare a variable of an abstract class (but we can’t initialise an object of that class)

*//VehicleTest.java*

**Vehicle** vehicle**;**

* Put an object of a subclass into that variable
* **@Test**
* **public** **void** **carAsVehicle()** **{**
* vehicle **=** **new** **Car(**"Audi A8"**,** 2**);**
* assertEquals**(**"Audi A8"**,** vehicle**.**getModel**());**
* **}**
* **@Test**
* **public** **void** **motorbikeAsVehicle()** **{**
* vehicle **=** **new** **Motorbike(**"Harley"**);**
* assertEquals**(**"Harley"**,** vehicle**.**getModel**());**
* **}**
* **@Test**
* **public** **void** **changeTypeOfVehicle()** **{**
* vehicle **=** **new** **Car(**"Audi A8"**,** 2**);**
* vehicle **=** **new** **Motorbike(**"Harley"**);**
* assertEquals**(**"Harley"**,** vehicle**.**getModel**());**
* **}**
* Declare a collection of objects of the abstract class Vehicle , but we cannot create an object of Vehicle and put it in there.

Let’s look at the last test collectionOfParentClassObjects

to remove the error comment out the line: // vehicles.add( new Vehicle("Mini Morris", 4));

**@Test**

**public** **void** **collectionOfParentClassObjects()** **{**

**ArrayList<Vehicle>** vehicles **=** **new** **ArrayList<>();**

vehicles**.**add**(** **new** **Car(**"Audi A8"**,** 2**));**

vehicles**.**add**(** **new** **Motorbike(**"Harley"**));**

*// vehicles.add( new Vehicle("Mini Morris", 4)); //MODIFIED*

assertEquals**(**3**,** vehicles**.**size**());**

**}**

optional: a quick note on casting. You can show it in your code, and mention that they can read up on it and that we’ll talk about it later.

* Note that when we put a subclass object into a parent class variable, it does not lose any of the details that came from the subclass. But to access it’s subclass specific information, you’ll need to cast it (ensure the compiler that you know what you’re doing). Syntax for casting is: put a subclass name in brackets just before the assignment like in below code.

**@Test**

**public** **void** **objectRemembersItsType()** **{**

vehicle **=** **new** **Car(**"Audi A8"**,** 2**);**

**Car** car **=** **(Car)** vehicle**;** *// Example of type casting Vehicle to Car*

assertEquals**(**2**,** car**.**getNumberOfDoors**());**

**}**

Note that there is an error if you try to instantiate a Vehicle because it’s abstract: Error:(23, 28) java: Vehicle is abstract; cannot be instantiated.. Yes, that is correct, we cannot instantiate an abstract class - let’s comment this line out.

**Subclass objects as method parameters**

Just a quick note on this common usage of abstract classes. Like with any other parent class, we can use the abstract type as a parameter in a method, and then pass in an instance of the subclass to that method.

just write come code on a whiteboard as you explain the below You can draw a UML diagram of Garage and its method too to explain this

Let’s imagine we have a Garage class. Which has a method addVehicle(Vehicle vehicleToAdd). Note that this is possible, even thou we will never actually pass an instance of a Vehicle (because Vehicle is abstract), but rather instances of Vehicle’s children.

**SOLID: Open Closed Principle**

What we have seen here is how we adhere to the Open Closed principle from our SOLID list.

Instructor: Write Open Closed under our SOLID heading.

Now our classes should be Open for extension and Closed for modification.

UML: when explaining below, add Truck class to your diagram, which is a child of Vehicle. Make note of how easy that is

This means that if we wanted to add in another type of vehicle, let’s say Truck, we won’t have to change our Vehicle class. We can simply create a new class for the Truck which extends the Vehicle class.

**Disadvantages**

There is a problem however. We are still constrained by the same problems as inheritance. We can only have one superclass, and all the methods and properties on that class bleed down to all of our children whether we want them or not.

**In Summary**

When describing these learning takeaways, show them in your end code:

* If a class is declared abstract, you cannot instantiate objects of its type. (in VehicleTest you cannot have a new Vehicle() call)
* Other classes can be subclasses (children) of an abstract class, and it is possible to instantiate objects of these classes, as long as the child itself is not abstract (Car and Motorbike)
* Abstract classes may or may not contain abstract methods, i.e., methods without body ( boardingInstructions in Vehicle)
* When you inherit from an abstract class, you need to provide implementations to its abstract methods in it.( boardingInstructions in Car and Motorbike)
* if a class has at least one abstract method, then the class must be declared abstract.
* You can declare variables and method parameters where the type is an abstract class, but they have to take the value of in instance of one its subclasses. (Garage class example)

**Task (Optional) - 20 minutes**

Create your own inheritance chain for modelling various types of farm animals. Use Class diagrams and TDD to test all classes apart of the abstract ones.

Task: You will create 3 classes: FarmAnimal, Sheep and Horse. Animals will make noises and be able to introduce themselves.

* your base class should be abstract and called FarmAnimal. FarmAnimal should have the following:
  + a species,
  + a getter for the species
  + a introduceYourself() method which returns the same String like 'Hi, I am a cow'.
  + an abstract method called makeANoise which returns a String. This method is abstract, because there is no ‘default’ sound that an animal makes.
* create a subclass of FarmAnimal: Sheep. Sheep should also have
  + in constructor, make species ‘sheep’
  + an implementation of the makeANoise which returns an appropriate String (eg. baa)
  + override of introduceYourself returning 'Hi, I am a sheep, baa baa'. Note that species and noise should come from variable.
* create another subclass of FarmAnimal called Horse. Horse should have the following:
  + in constructor, make species ‘horse’
  + a breed e.g. pony
  + a getter for the breed
  + an implementation of the makeANoise which returns an appropriate String (eg. neigh)
  + override of introduceYourself returning 'Hi, I am a horse from pony family, neigh neigh'. Note that species, noise and breed should come from variable.

**Packages and Access Modifiers**

**Learning Objectives**

* Understand that you can organise your code in packages

**Introduction to packages.**

Packages are groups of classes in a folder. Their purpose is to help organise your classes in a structured way. You have already seen them when you created a new project. You had 2 packages, one to store your code, and one to store your unit tests.

Now we are going to work off a simple project that has a few packages.

Hand out packages start point

[Draw a box similar to one from this site] (https://docs.oracle.com/javase/tutorial/java/javaOO/accesscontrol.html)

We can see we have two packages, one for animals and one for humans. These packages will contain the respective code. A dog is inside the animal package and has a public string name and bark, and human class inside the human package and has a teach method that accepts a dog class.

If we we were look at the folder structure of our codebase, it would look like this (each package name is its own folder):

- codeclan

| - com

| - packagesexample

| - animal

| - Dog.java

| - human

| - Trainer.java

In JavaScript, when we wanted the Trainer class to see the Dog class, we would write a require("./../animal/dog"). In Java we need to be more explicit so we say

import animal.Dog;

We could say

import animal.\*;

This will import all classes inside the animal package. This, however, is considered bad practice if we only need one of those classes. So lets change the import back to only import the Dog class.

We can create sub-packages in packages (they will just become subfolders).

Lets create a package called air inside animal. Package names are written in lower camel case to avoid conflict with the names of classes or interfaces.

In the air package create Bird class.

In the bird class we are going to create a method called speak() that returns the string “Tweet Tweet”.

public String speak() {

return "Tweet tweet";

}

Now in the Trainer class we are going to create a trainBird() method that takes in a bird and returns the result of calling the birds speak method.

public String trainBird(Bird bird) {

return bird.speak();

}

Inside Trainer, we can import the Bird like so

animal.air.Bird;

Note we cannot say animal.\*; because \* is not recursive, only pulls it in from that folder and not subfolders.

**Protected keyword**

So far we have only really dealt with private and public access modifiers.

There is a third on in Java known as protected.

Unlike private, which can only be accessed within the class itself, protected can only be accessed from any class within the package.

Change the dogs bark() method to protected.

Note now that in the Trainer class there is an error telling us that bark() is protected. We would only be able to invoke bark() from Trainer if they were in the same package.

Change bark() back to public

**Summary**

Packages are a good way to structure our files and group common classes or behaviours together.

The protected keyword gives us an additional level of security for our variables and methods should we only wish them to be accessed from within the same package.

**Inheritance Lab.**

Your task is to model and create a system to store information for employees of a large development company. You should use TDD, inheritance, packages and abstract classes for this.

**Create an abstract Employee parent class in a package called staff.**

* Employee will have a name, NI number and salary.
* Create Getters for all properties.
* Add a method named raiseSalary with takes in a parameter of type double to increment the salary.
* Add a method called payBonus which returns 1% of the employees salary.
* Don’t worry about testing just now until you create the subclasses.

**Create a subclass of Employee called Manager in a package called management.**

* Create a class for Manager.
* Add a property to store the department name in a property called deptName.
* Create a constructor that includes all the parameters needed for Employee and deptName.
* Add a getter method for deptName.
* Test all methods including raiseSalary and payBonus

**Create subclasses of Employee: Developer and DatabaseAdmin in a package called techStaff.**

Create 2 new classes for Developer and DatabaseAdmin.

* These should take in the same parameters as Employee and pass them to Employee constructor.
* Again test all methods for both classes.

**Create a subclass of Manager called Director in the management package.**

* Add a private property to store a double value budget.
* Create a constructor for Director that includes the parameters needed for Manager and the budget parameter.
* Create a getter method for this property.
* Test all methods.

**Extensions**

**Prevent unwanted values**

* Prevent a negative value for the raiseSalary method.
* Allow the user to change the Employees name and prevent a null value from being entered.
* Override the payBonus in director to return 2% of their salary.

**Interfaces**

**Lesson Duration: 90 minutes**

**Learning Objectives**

* Know what an interface is in Java
* Know why and when you would use an interface (interface as a contract)
* Be able to create an interface
* Be able to implement an interface in a class
* Understand that a class can implement multiple interfaces

**Intro**

Sometimes we want a number of classes to implement similar type of functionality, but we do not want then to inherit from the same parent. In such a situation, classes can make a promise to implement certain methods - we call a promise like that an **Interface**.

An interface is implemented as a class, but it only contains descriptions of methods, not their implementation - like an abstract method in an abstract class.

Once an interface is defined, all classes that **implement** it promise to contain bodies of all the methods specified in the interface.

**Why use interfaces?**

Often in programming, when creating classes and objects we are more concerned with **what** an object does, i.e. what functionality it has, rather than being concerned about exactly **how** it does it. Interfaces allow us to separate what a class does from how it does it. When we use interfaces we can write code which interacts directly with a class which implements that interface, or we can write code that interacts with the interface i.e. it only accesses functionality specified in the interface. This is called coding to the interface, rather than to the implementation. It is more favourable to do this as it makes our code more flexible. Rather than our code being able to work with only one specific class, our code will be able to work with any class which implements the interface, even if it hasn’t been created yet.

An example of this is the Java Database Connectivity (JDBC) API. It is made up nearly entirely of interfaces. The concrete implementations are provided as “JDBC drivers”. This lets us write all the JDBC code independent of the database (DB) vendor. We can just change the JDBC driver without changing any line of our Java code (except of any hardcoded DB-specific SQL code) whenever we want to switch DB vendor.

**How to declare an interface**

Hand out start code. It has an simple Runner class, and the tests for the getters. Maybe ask students to draw class diagrams.

UML: Ask one of the students or instructor to draw the UML on whiteboard, leaving a lot of space for other classes we’ll build during the lesson. This will be used throughout the class

Let’s start with running. At first we will make sure that the Runner promises to be able to run. It will implement an interface called IRun.

UML: add IRun interface to your UML diagram on the board and connect it with Runner

A note on interface naming conventions

The naming convention we are going to use for our interfaces is an old one. We are going to put an ‘I’ before for the interface name. We are going to create an interface for a ‘Runner’, so our interface will be called IRun. Let’s create a new interface called IRun in the main package.

NOTE: another naming convention that is commonly used is, rather than putting an ‘I’ at the start of the interface name, is to put the word ‘able’ at the end e.g. in our example the interface name would be Runnable - a bit of a mouthful. This means that we can end up with weird looking words.

#IntelliJ

Create a new class called IRun.java and set its type to Interface from drop down.

Right click > new > Java Class.

name class IRun and in drop down change to Interface.

This will give us the following:

*//IRun.java*

**public** **interface** **IRun** **{**

**}**

Let’s add a method to this interface called run that takes an int distance which does not return anything.

UML: add run(int) to IRun interface

Note that we are not providing an implementation for the method. We are leaving this up to the classes which use the interface (just like with abstract methods in abstract classes)

*//IRun.java*

**public** **interface** **IRun** **{**

**public** **void** **run(int** distance**);** *// MODIFIED*

**}**

NOTE: later versions of Java do allow us to provide default implementations of methods which can be overriden in the classes using the interface. We won’t be doing this on this course.

We use this interface with the **implements** keyword. So let’s get our Runner class to use this interface:

*//Runner.java*

**public** **class** **Runner** **extends** **Athlete** **implements** **IRun** **{** *// MODIFIED*

*//...*

**}**

Now our code won’t compile. WHY?

we now get the following error: Error:(1, 8) java: Runner is not abstract and does not override abstract method run() in IRun

**Interface as a Contract**

An interface forms a contract. Any class which implements an interface must implement **ALL** the methods in that interface.

If any are missing then we’ll get a compiler error as the class is not fulfilling the contract. The contract doesn’t say **HOW** these methods will be implemented, just that they **ARE** implemented.

Perhaps think of a Bank Account example. I may have accounts at several banks, but I expect that for each account, I should be able to pay-in money and withdraw money - that is the ‘contract’ I have with each bank. Each bank may do things differently, and to be honest, I don’t care how, but they all do the things I expect them to do. We could say that all banks promise to implement an interface for paying-in and withdrawing money.

So what we are saying in our example is that the Runner class, since it implements the IRun interface **MUST** have a method called run(int distance) which does not return anything.

In fact **ANY** class which implements the IRun interface must have a method called run(int distance) which does not return anything.

**How to implement an interface**

So we now need to get our Runner class to fulfil the contract i.e. we need to add a run(int distance) method, which does not return anything.

UML: add run(int) to Runner

We’ll write the test first:

*//RunnerTest.java*

**@Test**

**public** **void** **canRun()** **{** *// NEW TEST*

runner**.**run**(**10**);**

assertEquals**(**10**,** runner**.**getDistanceTravelled**());**

**}**

And then implement the method:

*//Runner.java*

**public** **class** **Runner** **extends** **Athlete** **implements** **IRun** **{**

*//...*

**public** **void** **run(int** distance**){**

**this.**distanceTravelled **+=** distance**;**

**}**

**}**

**Interface becomes even more useful when many classes implement it**

Let’s create another class that will also implement the IRun interface: a Triathlete.

Give students 5 minutes to complete this task:

make the Triathlete class also implement IRun interface and all functions that it promises. Write your tests first.

UML: connect IRun with Triathlete. Add run(int) to Triathlete

Solution:

*//TriathleteTest.java*

**public** **class** **TriathleteTest** **{**

**@Test**

**public** **void** **canRun()** **{** *//NEW TEST*

triathlete**.**run**(**10**);**

assertEquals**(**10**,** triathlete**.**getDistanceTravelled**());**

**}**

**}**

*//Triathlete.java*

**public** **class** **Triathlete** **extends** **Athlete** **implements** **IRun** **{**

**public** **void** **run(int** distance**){** *// MODIFIED*

**this.**distanceTravelled **+=** distance**;**

**}**

**}**

Note: in this class, all the implementations of run(), swim() etc will be very similar, but thanks to separating them we are able to add to them class-specific behaviours. Like bike() could check if you actually have a bike, but in triathlon it would also check if you already swam (in triathlon cycling comes after swimming). But in this simple example, all the sport methods, will only increase the distance for now.

Going back to the IRun interface, if we think of the the Olympics then we have different types of runners, for example we could have a Marathon Runner and a Sprinter. Both can implement the the IRun interface, but in different ways.

**public** **class** **Sprinter** **extends** **Athlete** **implements** **IRun** **{**

**public** **void** **run(int** distance**)** **{**

**this.**runAsFastAsPossible**(**distance**);**

**this.**distanceTravelled **+=** distance**;**

**}**

**private** **void** **runAsFastAsPossible(int** distance**)** **{**

*// DO STUFF*

**}**

**}**

**public** **class** **MarathonRunner** **extends** **Athlete** **implements** **IRun** **{**

**private** **int** marathonPace**;**

**public** **void** **run(int** distance**)** **{**

**this.**runAtMarathonPace**(**distance**);**

**this.**distanceTravelled **+=** distance**;**

**}**

**private** **void** **runAtMarathonPace(int** distance**)** **{**

*// DO STUFF*

**}**

**}**

**Implementing Multiple Interfaces**

**Unique advantage of interfaces**

Did you notice that what we’ve built up until now could have been just as well implemented with inheritance? **Theoretically Runner and Triathlete could have inherited the running bevaviour from the same parent class**. This is where unique advantage of interfaces comes into play: a class can implement multiple interfaces at the same time.

* When we use inheritance, our **class can only inherit from ONE superclass**.
* When we use interfaces, our **class can implement as many interfaces as it wants**.

**Interface segregation**

Let’s consider a following task:

UML: try to use your UML diagrams to illustrate below:

Just like we had our Runner and Triathlete promise that they will implement a run(int distance) method, let’s have our Swimmer and Triathlete promise that they will implement a swim(int distance) method.

There are two bad ways to do it, and I would like you to tell me why these are not the best:

* We could add a swim(int distance) method on the Athlete parent class
* We could add a swim(int distance) method to the IRun interface,

Ask students to come up with ideas why these solutions would not work

Neither of these really makes much sense.

Why? This would mean that every class which inherits form Athlete of implements IRun, like our Runner would now also need to have a swim(int distance) method. One thing about using inheritance or interfaces is that we shouldn’t force methods onto classes that they don’t need.

In real life, it should be possible to be a runner without having to be able to swim.

When creating an interface we should think about the classes which are going to implement it. We might add a lot of methods to an interface, but we need to ask ourselves “Do we really want to have to implement every method of this interface in every class that implements it?”. Keeping things simple is often a good idea.

**Implementing another interface**

Let’s instead add a new interface ISwim which will promise that classes which implement it will have a swim(int distance) method. This interface will be implemented by Swimmer but also by Triathlete.

Give students 5-ish minutes to complete this task:

UML: as you describe below, add these elements to UML on whiteboard

Create two interfaces: a ISwim (promising swim(int distance)) and ICycle (cycle(int distance)) interfaces.

Swimmer and Triathlete should implement ISwim

Cyclist and Triathlete should implement ICycle. Implement methods these interfaces promise, so that your code compiles.

Write your tests first.

UML: by now you should have new interfaces: ISwim and ICycle. They are connected with Cyclist, Swimmer and Triathlete where appropriate. You also added swim(int) and cycle(int) where appropriate.

As mentioned above it is a good practice, to have smaller highly cohesive interfaces rather than larger less specific ones.

*//TriathleteTest.java*

**@Test**

**public** **void** **canSwim()** **{***//NEW TEST*

triathlete**.**swim**(**10**);**

assertEquals**(**10**,** triathlete**.**getDistanceTravelled**());**

**}**

**@Test**

**public** **void** **canCycle()** **{***//NEW TEST*

triathlete**.**cycle**(**10**);**

assertEquals**(**10**,** triathlete**.**getDistanceTravelled**());**

**}**

*//SwimmerTest.java*

**@Test**

**public** **void** **canSwim()** **{***//NEW TEST*

swimmer**.**swim**(**10**);**

assertEquals**(**10**,** swimmer**.**getDistanceTravelled**());**

**}**

*//CyclistTest.java*

**@Test**

**public** **void** **canCycle()** **{***//NEW TEST*

cyclist**.**cycle**(**10**);**

assertEquals**(**10**,** cyclist**.**getDistanceTravelled**());**

**}**

*//Triathlete.java*

**public** **class** **Triathlete** **extends** **Athlete** **implements** **IRun,** **ISwim,** **ICycle** **{** *//AMENDED*

**public** **void** **run(int** distance**){**

distanceTravelled **+=** distance**;**

**}**

**public** **void** **swim(int** distance**)** **{** *//AMENDED*

distanceTravelled **+=** distance**;**

**}**

**public** **void** **cycle(int** distance**)** **{** *//AMENDED*

distanceTravelled **+=** distance**;**

**}**

**}**

*//Swimmer.java*

**public** **class** **Swimmer** **extends** **Athlete** **implements** **ISwim** **{**

**public** **void** **swim(int** distance**){**

distanceTravelled **+=** distance**;**

**}**

**}**

*//Cyclist.java*

**public** **class** **Cyclist** **extends** **Athlete** **implements** **ICycle** **{**

**public** **void** **cycle(int** distance**){**

distanceTravelled **+=** distance**;**

**}**

**}**

**SOLID: Interface segregation**

Notice what we’ve done - we separated sport activities into three small interfaces IRun, ISwim and ICycle. This way classes that want to be able to swim, promise that they will be able to swim. But classes who do not want to swim, do not promise swimming.

The Interface Segregation rules basically states that classes should not implement methods that they do not need.

UML: you can use UML to describe the below

* **When an interface is large and complicated, try to separate it into smaller interfaces**: Just explain, or show in code, but ask students to NOT code along and show this in IntelliJ:  Imagine we did not have IRun, ISwim and ICycle but instead we had just one large generic interface ICanDoTriathlon. ICanDoTriathlon would promise run(), swim() and cycle() Now imagine each of Runner, Cyclist and Swimmer implemented that interface. To make the compiler happy, you’d probably need to implement the required methods and have them do nothing interesting (since normal Runner cannot really swim or cycle). The same thing would happen if Athlete implemented that interface - each of Athlete’s subclasses would need to implement all these redundant methods.

Which takes us to the second important part of interface segregation:

* **Classes should not implement interfaces that are not needed or are too generic**, because you will have to write empty methods to satisfy these interfaces:

So with Interface Segregation we need to ensure that no class should be forced to depend on methods it does not use. We should split interfaces that are very large into smaller and more specific ones so that clients will only have to know about the methods that are of interest to them.

When creating an interface we should think about the classes which are going to implement it. We might add a lot of methods, but we need to ask ourselves “Do we want to have to implement every method in our class?”

Instructors: Write Interface Segregation on the board under our SOLID header.

**Interface as a Type**

A class implementing an interface also takes on the type of that interface. This means that once a Java class implements a Java interface you can use an instance of that class as an instance of that interface. Let’s add a new test to show this

start writing below code, and when you encounter the problem, continue with explanation below this code snippet

*//RunnerTest.java*

**@Test**

**public** **void** **canBeReferredToAsInterfaceType()** *// NEW TEST {*

**IRun** somethingThatRuns **=** **new** **Runner();**

somethingThatRuns**.**run**(**10**);**

assertEquals**(**10**,** somethingThatRuns**.**getDistanceTravelled**());**

*//OH NO! When you try to run this, you will get below error*

**}**

We’re getting the following error:

Error:(44, 53) java: cannot find symbol

symbol: method getDistanceTravelled()

location: variable somethingThatRuns of type IRun

Note that when we use an interface as a type (eg. IRun), we can only access those methods which are listed as part of the interface. If a class which implements the interface has other methods that are not a part of that interface (eg. getDistanceTravelled()), then we cannot access these methods when simply using the type of the interface.

In our example, if we create an variable of type IRun and put a Runner object in it. We cannot access Runner’s method getDistanceTravelled() because it’s not in IRun. This is because compiler only knows about methods relevant to the variable type, which in our example is IRun.

One possible solution is to cast the object back into it’s original object type. But that approach can be error prone and dangerous, so you will rarely use it for things other than testing.

change assertEquals line into:

*//RunnerTest.java*

**@Test**

**public** **void** **canBeReferedToAsInterfaceType()** **{**

**IRun** somethingThatRuns **=** **new** **Runner();**

somethingThatRuns**.**run**(**10**);**

assertEquals**(**10**,** **((Runner)** somethingThatRuns**.**getDistanceTravelled**()));** *//MODIFIED, added casting*

**}**

Ask students if they have an idea why could casting be dangerous?

Answer: imagine what would happen if you try to cast somethingThatRuns into a Runner but actually it is a Triathlete?

Change the line below and try to run the program (you’ll get an error) java.lang.ClassCastException: Triathlete cannot be cast to Runner at RunnerTest.canBeReferedToAsInterfaceType(RunnerTest.java:30)

*//RunnerTest.java*

**@Test**

**public** **void** **canBeReferedToAsInterfaceType()** **{**

**IRun** somethingThatRuns **=** **new** **Triathlete();** *//MODIFIED*

somethingThatRuns**.**run**(**10**);**

assertEquals**(**10**,** **((Runner)** somethingThatRuns**.**getDistanceTravelled**()));**

**}**

We’ll talk more about this when we talk about polymorphism

**Abstract classes and superclasses can implement interfaces**

**Superclasses implementing an interface**

One thing to note is that abstract classes and superclasses can also implement interfaces.

UML: you can use UML to describe the below

If we created an interface ITravelDistance which promise a method getDistanceTravelled() this interface could be implemented by Athlete class. This would mean that Athlete needs to implement above method, but also it would mean that all subclasses of Atlhete can be referred to using ITravelDistance as type (eg. Runner, Swimmer), because they would also inherit implementation of that interface.

**Abstract classes implementing an interface**

UML: this could be best explained with a simple class diagram

Abstract classes can also implement interfaces. Let’s imaging an abstract class Shape which has two non abstract children Square and Circle. Let’s have Shape implement an interface IHasArea which promises a method double area().

There are two options whenever an abstract class implements an interface:

* Provide a full implementation of the area() method in the abstract class. That implementation will cascade to all subclasses. This is not the best solution for the Shape, because each shape has a different formula for its area, hence it’s hard to come up with a default formula for the area.
* Make the area() method and abstract inside of Shape class and provide full implementations of it in each of the sub-classes (Square, Circle).

**Interfaces inheriting other Interfaces**

Optional Section: Feel free to skip this section during the class.

Java classes can only inherit one super class. But fortunately interfaces are not exactly Java classes, so as an exception they can inherit from many other interfaces.

Let’s imagine two scenarios:

UML: you can use UML to describe the below

* Interface extending another interface in IntelliJ Create a new Java class called ICanDoHurdles and write:

**public** **interface** **ICanDoHurdles** **extends** **IRun** **{**

**public** **void** **jump();**

**}**

A class that implements interface `IHurdles` needs to have all methods promissd by `IHurdles` and its parent `IRun`, so both `jump()` and `run(int distance)`.

* Interface combining multiple other interfaces in IntelliJ Create a new Java class called ICanDoTriathlon and write:

**public** **interface** **ICanDoTriathlon** **extends** **IRun,** **ISwim,** **ICycle{**

**}**

Now if we'd like some class to implement all 3 interfaces (`IRun`, `ISwim`, `ICycle`) we could simply have that class implement `ICanDoTriathlon`.

* A combination of both of these approaches (where you both add new methods and combine many other interfaces).

As we mentioned above in the SOLID section. It is good to separate interfaces into small ones which promise just a few methods each.

**Recap**

* An interface is a bit like a class except that it only contains descriptions of methods, not their full implementations.
* Interfaces are implemented by classes and form a contract in that, if a class implements an interface, then it must implement *ALL* the methods declared by that interface.
* Classes can implement multiple interfaces
* It is better to implement multiple, smaller interfaces, than larger less specific ones.

**Interfaces vs Abstract Classes**

**Objectives**

* Understand the difference between an abstract class and an interface

**What’s the difference?**

Often in interviews you will be asked to explain the difference between an abstract class and an interface.

**Guidelines**

Here are some guidelines on when to use an abstract class and when to use interfaces in Java:

* An abstract class is good if you think you will plan on using inheritance since it provides a common base class implementation to derived classes.
* An abstract class is also good if you want to be able to declare non-public methods. In an interface, all methods must be public.
* If you think you will need to add methods in the future, then an abstract class is a better choice. Because if you add new method headings to an interface, then all of the classes that already implement that interface will have to be changed to implement the new methods. That can be quite a hassle.
* Interfaces are a good choice when you think that the API will not change for a while.
* Interfaces are also good when you want to have something similar to multiple inheritance, since you can implement multiple interfaces.

**In Summary**

If you ever get asked the difference in an interview…

Abstract classes:

* Subclass can have only ONE superclass
* Can contain properties
* Can contain default implementations
* Can define abstract methods which must be implemented on subclass
* Give the superclass type to the subclass

Interfaces:

* Implementing class can have MANY interfaces
* Cannot contain properties
* Cannot contain default implementations
* Defines method signatures which must be implemented on class using it
* Gives the implementing class the interface type

**Lab - Stereo**

You are being asked to model a stereo system made up of separate components. The stereo will contain a radio and at least one of the following: record deck, CD player, cassette deck(if it’s a really old stereo). It will also be possible to connect an external sound source e.g. mp3 player, phone.

**Objectives**

* Practice creating abstract classes
* Practice creating abstract methods
* Practice using abstract classes
* Practice creating interfaces
* Practice implementing interfaces in classes

You will need to:

* Create a class Radio, which has a method tune. This may simply return a string of the station being tuned to (e.g. Radio 1).
* Create classes for different components e.g. record deck, CD player etc. Each class will have instance variables particular to that component e.g. a record deck may have play speeds, a CD player may have a number of CDs it can play at a time (think of the old-fashioned multi-changers). As the stereo is made up of separate components, each component should have its own make and model, and methods to operate them e.g. a CD player might have a play method. (If different components have the same methods then this might suggest that they implement the same interface);
* Create an abstract class Component which contains attributes you see as being common to all the main items in the stereo system. The classes created in step 2 above can then inherit from this Component class.
* Create a Stereo class, which contains all the components created above. It should also a name, have methods to use all the components in the stereo e.g. tune radio, play CD etc, raise, lower volume.
* Create an interface IConnect. This should have a method connect which takes in an instance of a Stereo and return a string which contains the name of the stereo it is connected to.
* Create a device e.g. mp3 player which implements the IConnect interface.

**Possible Extensions**

* Perhaps you could add output devices e.g. speakers, headphones, and have one connected to the stereo.
* Anything else you can think of.

**Remember**

Use TDD, with separate test files for each class.

**Polymorphism**

**Learning Objectives**

* Understand the need for polymorphism
* Understand the power of polymorphism
* Use polymorphic methods and collections

**Intro**

Hand out start code and let students look through it for a few minutes

Basically we’re modelling a simple computer network. We’ve got a class Network which has two collections both stored in ArrayLists, one to store Desktops, and the other to store Printers.

**ASIDE - Method Overloading**

You will notice that there are two methods called connect, each taking a different parameter type, one takes a Desktop, the other takes a Printer. Java lets us do this because it knows that depending on the argument we pass in, the correct version of the method is called. This is known as method **overloading**.

*//Network.java*

**public** **void** **connect(Desktop** desktop**){**

devicesDesktop**.**add**(**desktop**);**

**}**

**public** **void** **connect(Printer** printer**){**

devicesPrinter**.**add**(**printer**);**

**}**

**The need for polymorphism**

Our solution works but it isn’t very DRY and or scaleable. What happens if want to add other network devices like servers, laptops, mobile phones?? Are we going to make new collections and new method overloads every time?

Luckily, there is a better way. Polymorphism to the rescue!!

We can solve all of our problems with “polymorphism”. This isn’t as scary as it sounds, it just means we can wrap our objects up in an enclosing type that defines a contract between them all.

**What is polymorphism?**

The term polymorphism comes from two Greek words: ‘poly’ meaning ‘many’ and “morph” meaning ‘change’. When we talk of something being ‘polymorphic’ we mean that it can have ‘many forms’.

I know… let’s look at a real world example - a person. A person can have many ‘hats’ or ‘roles’, but they are not all who that person is. For example, Wilma is a person. She is married to Fred and they have one child, Pebbles. Wilma also works as a brain surgeon, and at weekends follows the Bedrock baseball team. Thus we can look at Wilma in many ways: \* Wilma the wife(of Fred) \* Wilma the parent(of Pebbles) \* Wilma the brain surgeon \* Wilma the baseball fan

When Wilma is at home she is Wilma the wife and/or Wilma the parent, but not Wilma the brain surgeon or baseball fan. When at work she is Wilma the brain surgeon etc.

Draw diagram on board of a person with different ‘roles’ or ‘forms’

We can do the same thing in programming. We can treat an instance of a class as if it is also another class/type at the same time.

**How to implement polymorphism in Java**

Polymorphism can be implemented using both abstract classes and interfaces. Remember that all classes which inherit from a class can take the type of the superclass. To use an abstract class we create a superclass which all the classes we want to treat as being the same can inherit from. But, inheritance is fraught with problems and we can quickly get into a mess. We can also just have one superclass.

Interfaces also allow us to treat a class as being of another type. When a class implements an interface it gains the type of the interface without having a horrible inheritance chain. We can have as many interfaces as we like, too. Rather than just one super class.

**Modify our example to be polymorphic**

What we want to do in our Network class is to be able to just create a single ArrayList for devices and be able to add different things e.g. Desktops and Printers to the ArrayList, but to do this they need to be of the same type. This is where polymorphism comes in. We need to be able to treat anything we add to the ArrayList as being of the one type. We can do this using an interface. We can create an interface which both Desktop and Printer inherit, and then have a single ArrayList where the type is the interface that we implement.

Let’s have an interface, IConnect that both of our connectable types implement from. Both a Desktop and a Printer are connectable. Once we have created our interface and made our classes implement it, we can than create an ArrayList of IConnect which means that it can contain both Desktop and Printer objects.

Lets create a new interface called IConnect in the main package. This interface will have one method connect which will take a single String parameter, and return a String.

#IntelliJ

Create a new interface in the main package called IConnect.

*//IConnect.java*

**public** **interface** **IConnect** **{**

**public** **String** **connect(String** data**);**

**}**

TASK: get the Desktop and Printer classes to implement this interface. Remember to write tests.

*//ComputerTest.java*

**@Test**

**public** **void** **canConnect(){**

assertEquals**(**"connecting to network: CodeClan"**,** desktop**.**connect**(**"CodeClan"**));**

**}**

*//Desktop.java*

**public** **class** **Desktop** **implements** **IConnect** **{**

*// AS BEFORE*

**public** **String** **connect(String** data**)** **{**

**return** "connecting to network: " **+** data**;**

**}**

**}**

*//PrinterTest.java*

**@Test**

**public** **void** **canConnect(){**

assertEquals**(**"connecting to CodeClan network"**,** printer**.**connect**(**"CodeClan"**));**

**}**

*//Printer.java*

**public** **class** **Printer** **implements** **IConnect{**

*//AS BEFORE*

**public** **String** **connect(String** data**)** **{**

**return** "connecting to " **+** data **+** " network"**;**

**}**

**}**

Great. Now, in Ruby this didn’t really mean much since we could pass anything to any method, and put anything we wanted into Arrays. In Java, this concept is extremely important. Our Desktop is both a Desktop AND an IConnect. Our Printer is both a Printer AND an IConnect.

What does this mean for us though? Let’s revist our networks device list.

**Polymorphic collections**

We can now delete some of our stinky code!

*//Network.java*

**public** **class** **Network** **{**

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

**private** **ArrayList<Desktop>** devicesDesktop**;** *//DELETED*

**private** **ArrayList<Printer>** devicesPrinter**;**

**public** **Network(String** name**){**

**this.**devicesDesktop **=** **new** **ArrayList<Desktop>();** *//DELETED LINE*

**this.**devicesPrinter **=** **new** **ArrayList<Printer>();**

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

**}**

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

**return** name**;**

**}**

**public** **int** **deviceCount(){**

**return** devicesDesktop**.**size**()** **+** devicesPrinter**.**size**();**

**}**

**public** **void** **connect(Desktop** desktop**){** *//DELETED FULL METHOD*

devicesDesktop**.**add**(**desktop**);**

**}**

**public** **void** **connect(Printer** printer**){**

devicesPrinter**.**add**(**printer**);**

**}**

**public** **void** **disconnectAll(){**

devicesDesktop**.**clear**();** *// DELETED LINE*

devicesPrinter**.**clear**();**

**}**

**}**

After deletion:

*//Network.java*

**public** **class** **Network** **{**

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

**private** **ArrayList<Printer>** devicesPrinter**;**

**public** **Network(String** name**){**

**this.**devicesPrinter **=** **new** **ArrayList<Printer>();**

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

**}**

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

**return** name**;**

**}**

**public** **int** **deviceCount(){**

**return** devicesDesktop**.**size**()** **+** devicesPrinter**.**size**();**

**}**

**public** **void** **connect(Printer** printer**){**

devicesPrinter**.**add**(**printer**);**

**}**

**public** **void** **disconnectAll(){**

devicesPrinter**.**clear**();**

**}**

**}**

Didn’t that feel good?? Now we can fix our code to be nice and polymorphic. Our ArrayList can renamed devices

*//Network.java*

**public** **class** **Network** **{**

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

**private** **ArrayList<Printer>** devices**;** *//UPDATED*

**public** **Network(String** name**){**

**this.**devices **=** **new** **ArrayList<Printer>();** *//UPDATED*

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

**}**

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

**return** name**;**

**}**

**public** **int** **deviceCount(){**

**return** devicesDesktop**.**size**()** **+** devicesPrinter**.**size**();**

**}**

**public** **void** **connect(Printer** printer**){**

devices**.**add**(**printer**);** *//UPDATED*

**}**

**public** **void** **disconnectAll(){**

devices**.**clear**();** *//UPDATED*

**}**

**}**

Our deviceCount() method can go be modified to just return the size of the devices ArrayList.

*//Network.java*

**public** **int** **deviceCount(){**

**return** devices**.**size**();**

**}**

We are going to do something controversial - we are going to make our ArrayList contain IConnect, NOT Desktops or Printers.

*//Network.java*

**private** **ArrayList<IConnect>** devices**;** *//UPDATED*

**public** **Network(String** name**){**

**this.**devices **=** **new** **ArrayList<IConnect>();** *//UPDATED*

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

**}**

Similarly, our connect() method is going to accept IConnect.

*//Network.java*

**public** **void** **connect(IConnect** device**){** *//UPDATED*

devices**.**add**(**device**);** *//UPDATED*

**}**

Believe or not, all of our tests still pass!!!!

How can this be?

**Magic**

The beauty of “polymorphism” is that any type can behave as if it is any of it’s super class types as well as it own. Our Desktop can be both a Desktop and an IConnect, like we mentioned earlier. So anything accepting IConnect can accept a Desktop.

However, it does NOT apply the other way round. Anything accepting a Desktop does NOT accept an IConnect. Not every IConnect is a Desktop! But every Desktop is an IConnect.

**Theme Park - Homework Revising Abstract Classes, Interfaces and Polymorphism.**

**Learning Objectives**

Practice modelling a large real world domain and what you have learned about using:

* classes
* inheritance
* abstract classes
* interfaces

**ThemePark**

**MVP**

**Classes and Abstract Classes**

Look through the start code and understand what is set up for you already.

* Visitor class with age, height and money.
* Attraction abstract class with name, rating and visitCount (which starts at 0).
* Rollercoaster, Dodgems, Park and Playground classes which inherit from Attraction
* Stall abstract class which has name, owner name, rating and parking spot.
* CandyFlossStall. TobaccoStall and IceCreamStall which inherit from Stall.

All of these classes have tests set up for them.

**Interfaces**

Introduce some Interfaces to enable charging, restricting and reviewing attractions and stalls:

* ITicketed promises double defaultPrice() and double priceFor(Visitor)
* ISecurity promises boolean isAllowedTo(Visitor)
* IReviewed promises int getRating() and String getName() Have some of the classes implement these interfaces. Below are the rules about what the implementations should be:
* Playground implements ISecurity because it has a maximum age of 15
* TobaccoStall implements ISecurity because it has a minimum age of 18
* Rollercoster implements ISecurity and requires a visitor to be over 145cm tall and over 12 years of age
* All Stalls and Attractions implement IReviewed.

**Note: Starting prices are:**

* £8.40 for Rollercoaster
* £4.50 for Dodgems
* Rollercoaster implements ITicketed and charges tall people over 200cm double fee
* Dodgems implements ITicketed and charge half price to children under 12 years old

**Extensions**

Create a ThemePark class to manage the attractions and stalls.

* ThemePark stores all Attractions and Stalls in it.
* ThemePark has a method getAllReviewed() which returns an ArrayList of IReviewed objects.
* Add an empty visitedAttractions ArrayList to Visitor which stores a list of Attractions and a method to add an attraction to the list.
* ThemePark has a method visit(Visitor, Attraction) calls for the attraction to increment its visitCount by 1 and adds the attraction to the visitors visitedAttractions list.

**Advanced Extensions**

* ThemePark has a method that can return a new HashMap<String, Integer> with all reviews. The HashMap will have a key of the name and value of review.
* ThemePark has a method getAllAllowedFor(Visitor) which takes a Visitor and returns an ArrayList of IReviewed that the visitor is allowed to go on.

**Composition**

**Objectives**

* Understand what composition is
* Understand how inheritance can sometimes lead us into dead-ends
* Learn how we can use composition over inheritance to resolve these
* Understand how to compose objects using interfaces

**Intro**

**What is Composition**

We’ve looked at inheritance, and seen how we can create our programs by looking at what something ***IS*** e.g. a car ***IS A*** vehicle.

We can also think about what something ***HAS*** e.g. a car ***HAS AN\_\_ engine, a car \_\_\_HAS A*** gearbox. In this sense we are seeing an object as being made up of, or ***composed*** of other objects. This is what we mean when we talk of composition in the object-oriented sense.

Perhaps write this on the board.

**class** **Engine** **{**

**private** **int** capacity**;**

**public** **void** **rev()** **{**

**}**

**}**

**class** **Gearbox** **{**

**private** **String** type**;**

**private** **int** numberOfGears**;**

**}**

**class** **Car** **{**

**private** **String** make**;**

**private** **String** model**;**

**private** **Engine** engine**;**

**private** **GearBox** gearBox**;**

**}**

**Composition and ownership**

One important thing to know about composition is that it is also about ownership. When we assign an object to be part of the composition of another object e.g. when we assign an Engine object to our Car class, then we are handing ownership of our Engine object over to the Car object of which it is now a part.

The consequence of this is that when the Car object is destroyed, all objects that it owns, or composed of, are also destroyed i.e. the Engine object is also destroyed.

**Composition and Behaviours**

In terms of what a Car does, i.e. its behaviour, we can think about what it HAS to carry our that behaviour i.e. an Engine, rather than what it IS that gives us that behaviour e.g. a drive() method in an Vehicle superclass.

We compose objects from other objects in order to get us to the functionality we need. We’ve already been doing this for a while without actually calling it Composition

Ask the class if they can think of any examples.

Thinking back to the weekend homework, we had Hotel objects composed of Room objects.

**Composition over Inheritance**

Inheritance can be useful, but it needs to be used with caution. We can very easily get into a nightmare of tangled classes and overriden methods. We also can only inherit from ONE thing and one thing only. An object can’t be both a Car and a Motorcycle. It can be one or the other.

In programming we have the saying favour composition over inheritance. We should compose our classes from other classes that implement the behaviours we need.

So we’re going to look at an example where our inheritance is starting to weigh us down and we can’t get the behaviour we need without breaking our hierarchy or making things ugly. And how we can use composition over inheritance to solve this problem.

We’re also going to use interfaces to help us with this solution.

Give out starter code.

Get the students to read it for 15 mins to half an hour.

**Code Structure**

The students should:

* draw a diagram of the classes and how they are related.

draw the diagram on the board

**The problem**

Key point:

We want the computer to output data using a printer. Or a speaker. Or anything that outputs data.

* Pair up and discuss it

Possible solutions:

* Common superclass for types which output data? Nope, already existing class hierarchy so it doesn’t make sense.
* Ancestor for all objects? We don’t want everything to outputs data!
* Multiple inheritance? It’s not supported and there’s all sorts of issues. If two superclasses have the same method, which does it use?

We want a common set of abilities that our things can do without polluting their class hierarchy which things that don’t concern it.

More than that, we don’t want this thing to have any clue about that behaviour, just to make sure that it exists on our object. A printer NEEDS to output data, a monitor NEEDS to output data but we don’t want our structure to care HOW they output data.

**Changing our code to use interfaces**

To solve our problem we are going to

* Add an IOutput interface
* Printer, Monitor, Speaker all implement it
* Change to IOutput in the computer

perhaps modify the diagram on the board to show this

First things first, let’s add the IOutput interface. We’re going to put this in a new package, called ‘behaviours’

#IntelliJ

> right click on java folder and select New > package

> Call the new package behaviours. (Need to do this so that behaviours and device\_management are in same folder.)

> Right click behaviours folder and create an IOutput interface

*//IOutput.java*

**package** behaviours**;**

**public** **interface** **IOutput** **{**

**String** **outputData(String** data**);**

**}**

Now, we need to make our Printer and Monitor implement it.

*//Printer.java*

**package** device\_management**;**

**import** behaviours.IOutput**;** *//NEW*

**public** **class** **Printer** **extends** **PrintingDevice** **implements** **IOutput** **{** */\* UPDATED \*/* **}**

*//Monitor.java*

**package** device\_management**;**

**import** behaviours.IOutput**;** *//NEW*

**public** **class** **Monitor** **extends** **VideoDevice** **implements** **IOutput** **{** */\*UPDATED \*/* **}**

You will notice that IntelliJ automatically imports the IOutput interface from the behaviours package for us.

If it doesn’t then type the following above the class declaration:

**import** behaviours.IOutput**;** *//NEW*

We already have tests to see if these things outputData, so if we run them they should all still pass.

Let’s now add the test for outputting data using the printer.

*//ComputerTest.java*

**@Test**

**public** **void** **canOutputDataViaPrinter(){**

**Printer** printer **=** **new** **Printer(**"Epson"**,** "Stylus"**,** 120**,** 4**);**

computer **=** **new** **Computer(**8**,** 512**,** printer**);**

assertEquals**(**"printing: space invaders"**,** computer**.**outputData**(**"space invaders"**));**

**}**

If we build this, we will see an error because the parameters being passed to the constructor of our Computer class do not match. This is quite correct, a Printer and Monitor are completely unrelated and we are passing a Printer to a method which is expecting a Monitor. We need to change this to be IOutput in the method signature.

*//Computer.java*

**package** device\_management**;**

**import** behaviours.IOutput**;**

**public** **class** **Computer** **{**

**private** **int** ram**;**

**private** **int** hddSize**;**

**private** **IOutput** outputDevice**;** *//UPDATED*

**public** **Computer(int** ram**,** **int** hddSize**,** **IOutput** outputDevice**)** **{** *//UPDATED*

**this.**ram **=** ram**;**

**this.**hddSize **=** hddSize**;**

**this.**outputDevice **=** outputDevice**;**

**}**

**public** **int** **getRam()** **{**

**return** **this.**ram**;**

**}**

**public** **int** **getHddSize()** **{**

**return** **this.**hddSize**;**

**}**

**public** **IOutput** **getOutputDevice()** **{** *//UPDATED*

**return** **this.**outputDevice**;**

**}**

**public** **String** **outputData(String** data**)** **{**

**return** **this.**outputDevice**.**outputData**(**data**);** *//UPDATED*

**}**

**}**

Since we have changed the name of getMonitor to getOutputDevice, we need to go update our test.

*//ComputerTest.java*

**@Test** *//DELETED*

**public** **void** **hasMonitor()** **{**

assertEquals**(**22**,** computer**.**getMonitor**().**getScreenSize**());**

assertEquals**(**786432**,** computer**.**getMonitor**().**getPixels**());**

**}**

**@Test**

**public** **void** **hasOutputDevice()** **{**

**IOutput** outputDevice **=** computer**.**getOutputDevice**();**

assertNotNull**(**outputDevice**);**

**}**

We have successfully output data using a printer. Let’s see if we can output data using a speaker:

*//ComputerTest.java*

**@Test**

**public** **void** **canOutputDataViaSpeaker(){**

**Speaker** speaker **=** **new** **Speaker(**100**);**

**Computer** computer **=** **new** **Computer(**8**,** 512**,** speaker**);**

assertEquals**(**"playing: Beep!"**,** computer**.**outputData**(**"Beep!"**));**

**}**

This doesn’t compile, since Speaker does not implement IOutput. Let’s go fix that!

*//Speaker.java*

**package** device\_management**;**

**import** behaviours.IOutput**;**

**public** **class** **Speaker** **extends** **SoundDevice** **implements** **IOutput** **{** */\*UPDATED \*/* **}**

**Strategy Pattern**

Wouldn’t it be nice if we could set the output device on the computer? Then on the same instance of a computer we could swap out the monitor for a printer and vice versa. We are no longer tied to one output device for the life of the computer object.

*//ComputerTest*

**@Test**

**public** **void** **canSetOutputDevice(){**

**Printer** printer **=** **new** **Printer(**"Epson"**,** "Stylus"**,** 120**,** 4**);**

computer**.**setOutputDevice**(**printer**);**

assertEquals**(**"printing: space invaders"**,** computer**.**outputData**(**"space invaders"**));**

**}**

This is a fairly easy change.

*//Computer.java*

**public** **void** **setOutputDevice(IOutput** outputDevice**){**

**this.**outputDevice **=** outputDevice**;**

**}**

This ability to set the behaviour of the computer’s outputData method (using setOutputDevice) is an example of an architectural pattern. In this case, it’s the Strategy Pattern. That is, we know our computer can outputData and by setting the IOutput object that they output data with, we can change the ‘strategy’ they use to do so.

e.g. save method, with Storage interface, so can either save to cloud or to hard drive, by using a setStorage method.

**SOLID: Dependency Inversion**

You will notice that we are passing in an instance of IOutput into our computer class.

This is adhering to the dependency inversion principle.

Most instincts will tell you to new up an outputDevice in the constructor of the computer but this would be the wrong way to do it.

So for example:

**public** **class** **Computer** **{**

**private** **int** ram**;**

**private** **int** hddSize**;**

**private** **IOutput** outputDevice**;** *//UPDATED*

**public** **Computer(int** ram**,** **int** hddSize**)** **{** *//UPDATED*

**this.**ram **=** ram**;**

**this.**hddSize **=** hddSize**;**

**this.**outputDevice **=** **new** **Printer();**

**}**

**}**

Would be the wrong way to go about this as the computer now depends on the instance of the printer already created.

Whereas the original way means that as long as there is an IOutput ready to pass in we can create a computer with no issue.

Instructors: Write up Dependency Inversion under SOLID headings on board.

**Recap**

* Composition refers to a ***has-a*** relationship where an object is made up of one or more other objects.
* Composition allows a class to use behaviour from a group of other classes, and makes it possible for that behaviour to change at runtime.
* In composition, the object composed of other behaviours ***owns*** those behaviours. This means that when the object is destroyed ***all*** of its behaviours are also destroyed.

**CodeClan Cars**

**Duration 2.5 hours**

Model a car dealership (e.g. Arnold CodeClan), making sure to use the concepts that we’ve covered in Java so far as appropriate.

You should be working in a TDD fashion and creating tests for your classes. Make sure you plan effectively in preparation for the extensions if you have enough time to attempt them.

**MVP**

* Create classes for different components (e.g. engine, tyres) that can be used to compose a vehicle
* Create different vehicle types (e.g. Car, Electric Car, Hybrid Car, etc)
* Assign them various properties (e.g. price, colour, engine etc.)
* Create classes for Customer and Dealership.
* Customers will have money and a collection of owned vehicles.
* Dealership will have stock of different types of cars and a till.

**Extension**

* Allow a customer to buy a vehicle.
* Allow dealership to buy vehicles
* Allow Dealerships to sell vehicles to customers.
* Allow damage to be added to a vehicle. This should be a cost and be removed from vehicles price.
* Allow dealership to repair vehicles. This should take the cost from the dealer and add back to value of vehicle.

**Lab - Fantasy Adventure**

The task is to model a fantasy adventure. The game will have players, who will have a number of health points and weapons/spells. Players will fall into different categories:

* Dwarves, Barbarians, Knights
  1. each of these will have a weapon and fight with other players (Sword, Axe, Club, etc)
  2. should be able to change their weapon at any point in the game.
* Warlocks and Wizards
  1. they will be able to cast spells on others (Fireball, Lightning Strike, etc).
  2. they will have a mythical creature (e.g. Ogre, Dragon) who will defend them from attackers.
  3. should be able to change spells or creature at any point in the game.
* Clerics
  1. they will have healing tools with which they can heal other players (potion, herbs, etc).
  2. should be able to change healing tool at any point in the game.

The game will also have rooms to work through. Each room should either have some kind of treasure (Gold, Gems) to collect or an Enemy (Troll, Orc) to defeat before moving on.

**Remember**

* Use TDD
* Git commit regularly

**MVP**

* Create the players and rooms for the adventure. (Don’t worry about giving the rooms any exits.)
* Give different players properties/methods that are specific to their own class. (e.g. a Knight could have armour that could reduce damage they take)
* The user should complete rooms by either collecting treasure or defeating an enemy and a new room will be presented to them.

**Extensions**

* Create a Quest for the players to work through. Possibly a collection of rooms to go through and the quest completes when all rooms are completed
* Expand any other way you wish.

**Lab - Fantasy Adventure**

The task is to model a fantasy adventure. The game will have players, who will have a number of health points and weapons/spells. Players will fall into different categories:

* Dwarves, Barbarians, Knights
  1. each of these will have a weapon and fight with other players (Sword, Axe, Club, etc)
  2. should be able to change their weapon at any point in the game.
* Warlocks and Wizards
  1. they will be able to cast spells on others (Fireball, Lightning Strike, etc).
  2. they will have a mythical creature (e.g. Ogre, Dragon) who will defend them from attackers.
  3. should be able to change spells or creature at any point in the game.
* Clerics
  1. they will have healing tools with which they can heal other players (potion, herbs, etc).
  2. should be able to change healing tool at any point in the game.

The game will also have rooms to work through. Each room should either have some kind of treasure (Gold, Gems) to collect or an Enemy (Troll, Orc) to defeat before moving on.

**Remember**

* Use TDD
* Git commit regularly

**MVP**

* Create an abstract class Player. Each player will have a name and a number of health points.
* Create an abstract class for Enemy. Each enemy will have a number of health points and a method to take damage.
* Create some subclasses of Enemy (Troll, Orc).
* Create an interface IWeapon which has a method attack
* Create classes which implement the IWeapon interface, e.g. Sword, Club, Axe.
  1. Each of these could have a different damage value, which affects the health of the enemy attacked.
* Create an abstract subclass of Player called Fighter. This should have an IWeapon and an attack method.
* Create some subclasses of Fighter - Barbarian, Dwarf, Knight.
  1. Each will have a weapon of type IWeapon, which they can change as the game goes on i.e. a Dwarf might swap an Axe for a Club.
  2. Add methods, properties specific to each
* Create an interface ISpell which has a method cast
* Create classes which implement the ISpell interface
  1. Each of these could have a different power value, which affects the health of the enemy they are cast upon.
* Create an interface IDefend which has a method defend
* Create classes which implement the IDefend interface e.g. Ogre, Dragon
  1. Each should have a defend method which affects the health value of the enemy they attack.
* Create an abstract subclass of Player called Mage. This should have an IDefend and a method called defend.
* Create subclasses of Mage, e.g. Warlock and Wizard
  1. Each will have a spell of type ISpell and a defender of type IDefend, which they can change as the game goes on. \* Warlock and Wizard should have IDefend instead of IWeapon.
* Create an interface IHeal which has a single method Heal
* Create classes which implement the IHeal interface e.g. Potion, Herbs etc.
* Create a Cleric class. This class will have an IHeal with which they can heal other players. This can be changed as the game goes on.

**Extensions**

* Create a class for Room.
* Create an Enum called Exit, this should have 4 values (WEST, EAST, NORTH, SOUTH)
* Room should have an array of exits which should create a random number of unique exit locations when initialised.
* When a user chooses an exit it should take them to a new Room.
* Create a class for Quest. This should have an array of Rooms to work through.
* Quest should complete when the array of rooms is empty.

**Advanced Extensions**

* Create a Party class which should have an ArrayList of type Person and an int for treasure value.
* Create method to add a Person to the Party.
* Create an interface ITreasure. This should have a getValue method defined.
* Create some classes to implement ITreasure. These should have default value.
* Create method that takes in treasure and adds value to party treasure.
* Create an attack method that allows every party member to attack or cast a spell.
* Create a heal method that allows any medic to heal all members of the party.
* Allow rooms to randomly assign an enemy to defeat or treasure to find once user enters the room.

**ending Machine Lab**

In this exercise you will build the brains of a vending machine. It will accept money, make change, maintain inventory, and dispense products. All the things that you might expect a vending machine to accomplish.

The point of this is to practice the concepts you have learned so far and can be used to practice TDD.

**MVP**

* The vending machine will accept valid coins (10p, 20p, 50p, etc) and reject invalid ones (1p, 2p).
* When a valid coin is inserted the amount of the coin will be added to the current amount. Rejected coins are returned.
* There are three products: cola for £1.00, crisps for £0.50, and sweet for £0.65. When the respective code is entered and enough money has been inserted, the product is returned and the current amount will be set to £0.00. If there is not enough money inserted then the machine returns the amount still required.

**Extensions**

* When a product is selected that costs less than the amount of money in the machine, then the correct change is placed in the coin return.
* When the return coins button is pressed, the money the customer has placed in the machine is returned into the coin return.

**Advanced Extensions**

* When the machine is not able to make change with the money in the machine for any of the items that it sells, it will only accept exact change.

**Weekend Homework - Ray’s Music Exchange**

:star: This is a PDA Homework :star:

You are being asked to model a music shop. This shop will contain items which can be sold. These items might be instruments, which someone may play e.g. guitar, saxaphone, piano, etc, or, they might be other items, e.g. guitar strings, drum sticks, sheet music etc. Each item which can be sold will have both a price at which the shop bought the item, and a price at which the item will be sold.

You will need to:

* Create a IPlay interface that could be applied to instruments. This should have a play method that returns the sound of the instrument being played as a String.
* Create classes for different types of instruments e.g. guitars, pianos, etc. (any you can think of). Each class will have its own instance variables for attributes particular to that instrument e.g. a guitar may have a number of strings, a trumpet may have a number of valves etc.
* Create an Instrument superclass which contains attributes you see as being common to all instruments e.g. material it is made from, colour, type (Brass, String, Woodwind, Keyboard etc). The classes created in step 2 above can then inherit from this Instruments class.
* Create a ISell interface which has a calculateMarkup method, which returns an int/double, based on the buying price and the selling price.
* Create classes for items the shop may sell, i.e. which implement the ISell interface. Such items should have the following:
  1. a type/description e.g. guitar, drum sticks, guitar strings, sheet music etc.
  2. a price at which the shop bought the item.
  3. a price at which the shop plans to sell the item.

Some ISell items will be musical instruments i.e. instruments can be both IPlay and ISell.

* Create a Shop class, which has a collection of ISell items, called stock. In your shop you should be able to:
  1. add items to stock.
  2. remove items from stock.

**Possible Extensions**

* Using the calculateMarkup method for each item in stock, create a method which gives the total potential profit for the shop.
* Create and use an enum for instrument types e.g. keyboard, wind, brass etc
* Any other extensions you may wish to add.

**Remember**

Use TDD, with separate test files for each class.

**PDA Reminder:**

As part of this homework you are required to take screenshots of the following:

- Demonstrate the use of Polymorphism in a program.

- An Inheritance Diagram

Take a screenshot of the use of Inheritance in a program. Take screenshots of:

- A Class

- A Class that inherits from the previous class

- An Object in the inherited class

- A Method that uses the information inherited from another class.

* 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 2, I.T 7, A.D 5