THE UNIVERSITY OF SYDNEY

# INFO1113/COMP9003 Week 6 Tutorial

**Polymorphism and Packages**

## Abstract classes and methods

**Abstract** classes cannot be instantiated but can define methods and attributes to be inherited by sub classes. Abstract classes can also mark methods that must be implemented by subtypes, since the type is unable to be instantiated there is no risk that the JVM could execute the method without a definition.

Abstract classes are marked using the `abstract` keyword in the class declaration. This simply tells the compiler that the class must not be instantiated, regarldess if all methods are defined.

```
public abstract class DrawableObject
```

**Abstract** methods can use the same access modifiers and return types as methods we have seen and used before but do not carry a definition when declared. The `abstract` keyword annotation on methods marks the method to be implemented by any subtype.

```
public abstract void draw();
```

Since we only **declare** the method in the abstract class and, any type that inherits from the abstract class must implement this method and will be able to specify its own behaviour.

For example, if there exists an abstract class called `Media` which specifies an abstract method `interact`. Any subtype (such as `Book` or `Movie`) will need to implement this method.

# Question 1: Water

You are to implement the following.

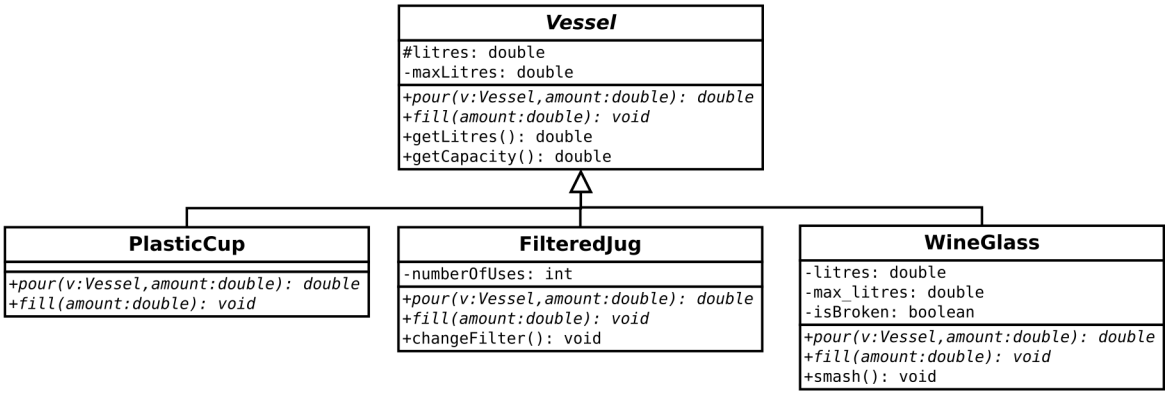


Figure 1: Vessel UML Diagram

Each class may contains a specific definition of the methods *pour* and *fill* but each method must be usable with other `Vessel` objects.

Each container has at least two attributes, `litres` and `maxLitres`. `maxLitres` is the total number of litres of water a vessel can hold, `litres` is the current litres of water contained in the vessel.

The `pour` method for each class corresponds to the action of pouring liquid from one vessel to another. If no vessel is specified, then it is assumed the liquid is wasted. This may be necessary if the container contains something contaminated or is smashed.

The `fill` method for each class correspond to the action to adding water to the vessel. The vessel must not exceed the `maxLitres` specified by the vessel.

The following classes have the specified behaviour:

- `FilteredJug` contains a filter with a number of uses. If the `numberOfUses` is > 0, the jug can be filled by other sources. The standard number of uses for a filter is 30, and after the jug has been filled 30 times, the jug must not be able to accept any more water
- `WineGlass` is a fragile container and if the glass is broken, it will not be able to hold any water. Therefore if any object attempts to pour water to the wineglass in a brokne state, that water is wasted.

# Interfaces

Interfaces declare methods that will describe a *behaviour* an object may have. When a class `implements` the interface it will need to **define** the methods that have been declared to satisfy the relationship.

```
public interface Talkable {
    //outputting what the object is saying
    public void talk();

    //What the object should say, if it can understand words
    public void script(String[] words);
}
```

**Wait... aren't they the same thing?** These two concepts are oftenly confused of being the same thing when really they are used in different scenarios. Class heirarchy infers to a clear generalisation of a type while interfaces infers to an shared behaviour between types.

- Example 1: a `Dog` and `Cat` both exhibit some way of `talk`ing but the noise they emit is different and it doesn't make sense for Cat and Dog to be in the same heirarchy.
- Example 2: `Person` can interact with other `Person` objects but also `Furniture`, `Singage` and `Animal` objects. It is clear are not suitable to be in the same class heirarchy but can facilitate a method of `Interact`ion.

Unlike class inheritance where a class can only inherit from only a single class, a class can implement as many interfaces it wants.

The following is where a `Person` class has specified it will implement a number of interfaces.

```
public class Person implements Swim, Talk, Jump, Run
```

Similar to **abstract** classes, when a class states it will implement an interface it will need to implement all methods declared.

# Question 2: Interacting with objects

Given the following UML diagram, implement the following interface and classes. Your classes must be able to interact with any object that implements the `Interactable` interface. When an object invokes `talkTo`, it will use Strings from the initiation script, for every elements in the initiation script the target object will respond with Strings in the response script.

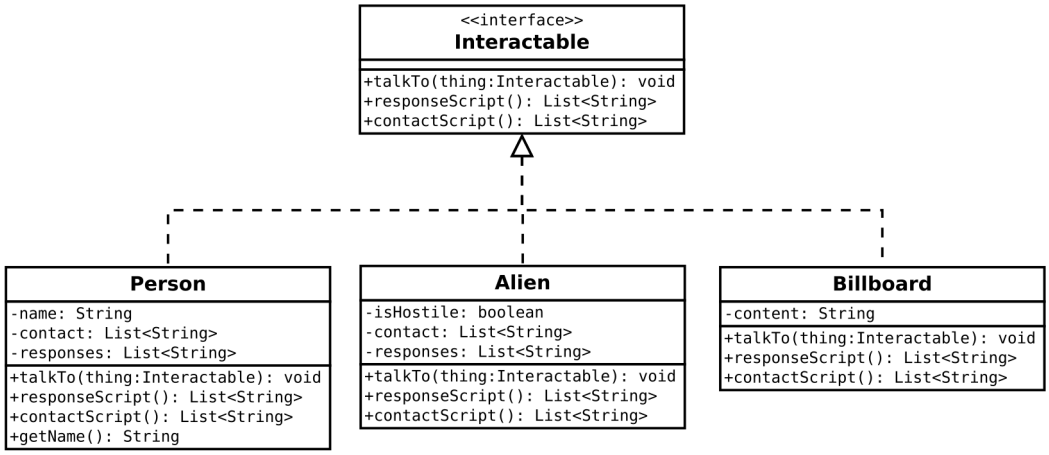


Figure 2: Interaction UML Diagram

There are a few conditions of interaction between these objects.

- Consider how the dialogue for all 3 classes could be inputted. This could be as simple as creating a constructor that allows for input or extending the interface.
- All objects have variable length scripts, if a script contains no Strings, then the dialogue may be cut short to someone just specifying the first string as part of their script.
- `Person` object will use their name at the beginning of their script.
- You may utilise the `isHostile` attribute in the `Alien` class however you see fit.
- `Billboard` usually has short messages to state that it is trying to sell to you immediately. It does not have a clever method of communication and will instead just responds to you with a whole string.

Output of a `Person` interact with another `Person` selling tickets.

```
How are you?
Fine, See you soon.
```

## Packages

A package is a grouping of related types providing access protection and name space management. It gives the programmer control over the namespace and access of classes. We can group classes together in a single package to eliminate naming conflicts between classes and to organise our code.

To specify a package name for class, we use the package keyword and specify it at the top of the .java source file.

```
package my.library;
```

We are able to import files specified outside of the current directory by using the class path flag during compilation and execution.

```
javac -cp .:<directory to .class/.java files> MyProgram.java
java -cp .:<directory to .class files> MyProgram
```

## Question 3: My First Collections

Since you have already implemented your own dynamic array, create a modular code base that will allow you to import your own libraries that you can use within your projects. Try and set the correct package name and correct compilation command for compiling your program with your own packages.

```
import my.collections.DynamicArray;
import java.util.Random;

public class TestProgram {

    public static void main(String[] args) {
        Random rand = new Random();
        DynamicArray array = new DynamicArray();

        for(int i = 0; i < 20; i++) {
            array.add(rand.nextInt(40));
        }

        for(int i = 0; i < array.size(); i++) {
            System.out.println(array.get(i));
        }

    }

}
```

## Question 4: Creating an archive

When distributing your own programs you will generate a java archive (.jar). A java archive contains and compresses class files and is a container format that is reconised by the JVM that can be executed.

Create a .jar file of the previous question, making sure you bundle all the necessary classes so it can execute without error.

You can create an archive using the `jar` command and specifying `-cf` flag. However, ensure you specify a manifest flag with the jar command and bundle a manifest file as part of step.

```
jar -cfm manifest.txt project.jar project/*.class
```

## Question 5: Assignment Preparation

Next week the assignment specification will be released. The assignment will involve implementing a game with graphics using the Processing library and Gradle. Refer to the Introduction to Processing guide under Ed lessons. Download the demo game from Ed resources and run it with the command `gradle run` in the root directory of the project (where the `build.gradle` file is located).

The main class `App` contains a `draw()` method that runs every frame. The `setup()` method occurs once at the beginning. Modify the program to:

- Have a different background colour.
- Change the movement keys from `a` and `d` to `q` and `e`.
- Change the logic to score a point whenever the ball it hit, rather than missed.
- Change the stopping condition so the game ends when the score differs by 2 points, rather than the first player to reach 5.
- Consider if any improvements could be made to the design to more easily facilitate simulations and unit testing.

## Question 6: Assessed Task: Quiz 1

Remember you are required to complete the quiz within the due date. Go to Canvas page of this unit and click on Quizzes to find out the quiz and the due date. This is a marked assessment.