INFO1113 / COMP9003
Object-Oriented Programming

Lecture 6

THE UNIVERSITY OF SYDNEY

Contents developed by Tyson Thomas

Reminder

- **Quiz** this week (week 6)
  - Released on Thursday, 30 March at 23:59
  - Due date Friday, 31 March at 23:59
  - Quiz duration → 1.5 hours
    - **Only one attempt is allowed**
  - Covers week 1 - week 5 contents
  - 10 MCQ questions
  - 3 extended response questions
    - Identify errors
    - Write code from specification
  - Practice quiz is available in Canvas

Acknowledgement of Country

I would like to acknowledge the Traditional Owners of Australia and recognise their continuing connection to land, water and culture. I am currently on the land of the Gadigal people of the Eora nation and pay my respects to their Elders, past, present and emerging.

I further acknowledge the Traditional Owners of the country on which you are on and pay respects to their Elders, past, present and future.

Copyright Warning

COMMONWEALTH OF AUSTRALIA

Copyright Regulations 1969

WARNING

This material has been reproduced and communicated to you by or on behalf of the University of Sydney pursuant to Part VB of the Copyright Act 1968 (**the Act**).

The material in this communication may be subject to copyright under the Act. Any further copying or communication of this material by you may be the subject of copyright protection under the Act.

Do not remove this notice.

Topics: Part A

- **Abstract Classes**
- **Abstract Classes UML**
- **Interfaces**
- **Interfaces and UML**

What is an abstract class?

Although similar to a **concrete class**, an **abstract** class cannot be instantiated.

It can define methods and attributes which can be inherited, inherit from super types and can be inherited from.

However, abstract classes can also enforce a method implementation for subtypes.

Refer to Chapter 8.4, pages 684-688, (Java, An Introduction to Problem Solving & Programming, Savitch & Mock)

Why would we use abstract?

The main case for **abstract** is that we have some **type** that we do not want instantiated but is a generalisation of many other types.

Example:

- **Shape** is a generalisation of **Triangle, Square, Circle** but we don't have a **concrete** instance of **Shape**

- **Furniture** is a generalisation of **Chair, Sofa, Table** and **Desk**.

Refer to Chapter 8.4, pages 684-688, (Java, An Introduction to Problem Solving & Programming, Savitch & Mock)

Sounds like abstract classes are quite different from classes!

What can we still do?

We still are able to specify:

- Constructors
- Define methods (static and instance)
- Attributes
- Use all the access modifiers
- … everything a regular class can do *except!*

We cannot instantiate the class ***but* we can specify methods subtypes must define.**

~~**AbstractClass a = new AbstractClass();**~~

Declaration of an abstract class

Simply we are able to define an **abstract** class by using the **abstract keyword**. This immediately marks the class as abstract and we do not need anything more.

Syntax:

[modifier] **abstract** class ClassName

Example:

```
public abstract class Furniture
```

What if we try to instantiate it?

Since it is marked as abstract, the compiler will refuse to allow this type of instantiation.

```
> javac FurnitureStore.java
FurnitureStore.java:22: error: Furniture is abstract; cannot be instantiated
        Furniture f = new Furniture("Table");
                      ^
1 error
<program end>
```

Abstract methods

We are able to declare an **abstract** method in **only abstract classes**. When we declare an abstract method we do not **define** a method body (the logic of the method).

```
public abstract void stack(Furniture f);
```

The class should not be instantiated and behaviour is defined by the subtypes and not the super type.

Declaration of an abstract class

We have an **abstract** class specified.

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

public          class Furniture {

   private String name;
   private List<Part> parts;

   public Furniture(String name) {
      this.name = name;
      this.parts = new ArrayList<Part>();
   }

   public void addPart(Part p) {
      parts.add(p);
   }

   public abstract void stack(Furniture f);
}
```

Notice we have declared an **abstract** method.

Declaration of an abstract class

We have an **abstract** class specified.

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

public abstract class Furniture {

   private String name;
   private List<Part> parts;

   public Furniture(String name) {
      this.name = name;
      this.parts = new ArrayList<Part>();
   }

   public void addPart(Part p) {
      parts.add(p);
   }

   public abstract void stack(Furniture f);
}
```

```
public class FurnitureStore {
   public static void main(String[] args) {
      Chair ch = new Chair();
      ch.stack(new Chair());
   }
}
```

```
public class Chair extends Furniture {

   public Chair() {
      super("Chair");
   }
}
```

Take notice of what's happening here.
That's what they meant by abstract classes;
You can't instantiate them through themselves,
but you also can't use the methods without defining it.

This is fundamentally different from a simple `extends` inheritance keyword

```
> javac FurnitureStore.java
Chair.java:1: error: Chair is not abstract and does not override abstract
method stack(Furniture) in Furniture
public class Chair extends Furniture {
       ^
1 error
```

Declaration of an abstract class

We have an **abstract** class specified.

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

public abstract class Furniture {

   private String name;
   private List<Part> parts;

   public Furniture(String name) {
      this.name = name;
      this.parts = new ArrayList<Part>();
   }

   public void addPart(Part p) {
      parts.add(p);
   }

   public abstract void stack(Furniture f);
}
```

```
public class FurnitureStore {
   public static void main(String[] args) {
      Chair ch = new Chair();
      ch.stack(new Chair());
   }
}
```

```
public class Chair extends Furniture {

   public Chair() {
      super("Chair");
   }

   public void stack(Furniture f) {
      System.out.println("Don't put furniture on chairs!");
   }
}
```

Now we have defined the method **stack** in the subclass.

```
> javac FurnitureStore.java
>
```

Declaration of an abstract class

We have an **abstract** class specified.

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

public abstract class Furniture {

   private String name;
   private List<Part> parts;

   public Furniture(String name) {
      this.name = name;
      this.parts = new ArrayList<Part>();
   }

   public void addPart(Part p) {
      parts.add(p);
   }

   public abstract void stack(Furniture f);
}
```

```
public class Chair extends Furniture {

   public Chair() {
      super("Chair");
   }

   public void stack(Furniture f) {
         System.out.println("Don't put furniture on chairs!");
      }
   }
```

Now we have defined the method **stack** in the subclass.

```
public class FurnitureStore {
   public static void main(String[] args) {
      Chair ch = new Chair();
      ch.stack(new Chair());
   }
}
```

We can now declare and invoke stack through **Chair** class.

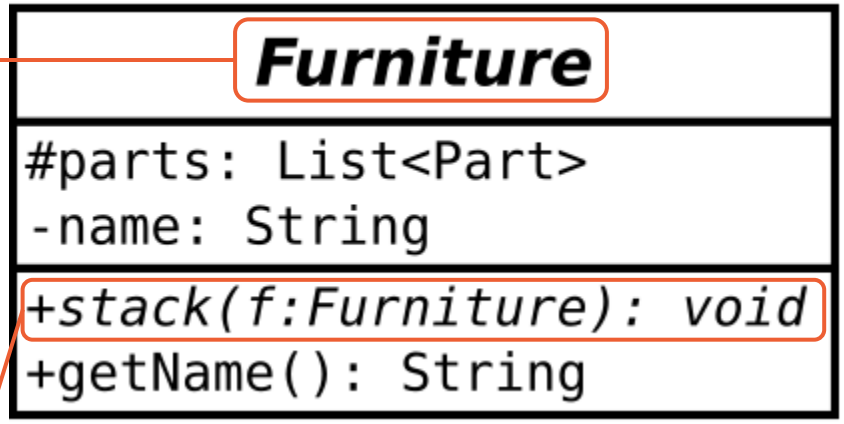
```
> java FurnitureStore
Don't put furniture on chairs!
```

Demonstration

Abstract Classes and UML

Within a UML class diagram, we can illustrate abstract classes with the following.

Italicised font shows that it is an abstract class.



We also show **polymorphic** method as italicised.

Notice that it's only the `stack` that is abstract which requires being defined;

Other methods can totally work without being defined

Interfaces

We will be introducing a new keyword **implements**.

Interfaces share a similarity with **Abstract Classes** in that they declare methods that a subclass must **implement** and **they cannot be instantiated.** However, unlike classes, they can be **implemented by classes** as many times as they like.

We are not bound to implementing a single interface, we can implement multiple interfaces.

Refer to Chapter 8.4, pages 659-669, (Java, An Introduction to Problem Solving & Programming, Savitch & Mock)

Interfaces

Interfaces

- Cannot specify any attributes only methods
- **Do not (typically) provide a method definition**
- **Cannot instantiate them**
- **Can be implemented multiple times**

From an application design perspective we need to consider how we can use interfaces and where they are appropriate.

Declaration of an interfaces

Simply we are able to define an interface by using the **interface** keyword.

Syntax:

[modifier] **interface** InterfaceName

Example:

public interface Swim

Declaration of an interfaces

Simply we are able to define an interface by using the **interface** keyword.

Syntax:

[modifier] **interface** InterfaceName

Example:

To be clear, an **interface** is **not a class.**

public interface Swim

Declaration of an interfaces

Simply we are able to define an interface by using the **interface** keyword.

Syntax:

[modifier] **interface** InterfaceName

Example:

```
public interface Swim {
    public void floating();
    public void diving();
}
```

To be clear, an **interface** is **not a class.** It defines a group a methods for implementers to define.

Declaration of an interfaces

Simply we are able to define an interface by using the **interface** keyword.

Syntax:

[modifier] **interface** InterfaceName

Example:

```
public interface Swim {
    public void floating();
    public void diving();
}

public class Dog implements Swim
```

To be clear, an **interface** is **not a class.** It defines a group a methods for implementers to define.

Since a **Dog** class **implements** the **Swim** interface it will need to define the methods for **Swim.**

Interfaces

So let's take a look at the following example

```
public interface Move {
   public void move(double hours);
}
```

```
public class Dog implements Move {
   private String region; //Water or Land
   private double landSpeed_kmh = 50.0;
   private double waterSpeed_kmh = 8.0;
   private double kmTravelled = 0.0;

   public Dog(String region) {
      this.region = region;
   }

   public void move(double hours) {
      if(region.equals("water"))
         kmTravelled += (waterSpeed_kmh *hours);
      else if(region.equals("land"))
         kmTravelled += (landSpeed_kmh * hours);
   }

   public double getKMTravelled() {
      return kmTravelled;
   }
}
```

```
public class Dolphin implements Move {
   private String region; //Water or Land
   private double landSpeed_kmh = 1.0;
   private double waterSpeed_kmh = 60.0;
   private double kmTravelled = 0.0;

   public Dolphin(String region) {
      this.region = region;
   }

   public void move(double hours) {
      if(region.equals("water"))
         kmTravelled += (waterSpeed_kmh * hours);
      else if(region.equals("land")) {
         kmTravelled += (landSpeed_kmh * hours);
   }

   public double getKMTravelled() {
      return kmTravelled;
   }
}
```

Interfaces

So let's take a look at the following example

```
public interface Move {
    public void move(double hours);
}
```

We have defined our **Interface Move** that will be implemented by **Dog** and **Dolphin**.

```
public class Dog implements Move {
    private String region; //Water or Land
    private double landSpeed_kmh = 50.0;
    private double waterSpeed_kmh = 8.0;
    private double kmTravelled = 0.0;

    public Dog(String region) {
        this.region = region;
    }

    public void move(double hours) {
        if(region.equals("water"))
            kmTravelled += (waterSpeed_kmh *hours);
        else if(region.equals("land"))
            kmTravelled += (landSpeed_kmh * hours);
    }

    public double getKMTravelled() {
        return kmTravelled;
    }
}
```

```
public class Dolphin implements Move {
    private String region; //Water or Land
    private double landSpeed_kmh = 1.0;
    private double waterSpeed_kmh = 60.0;
    private double kmTravelled = 0.0;

    public Dolphin(String region) {
        this.region = region;
    }

    public void move(double hours) {
        if(region.equals("water"))
            kmTravelled += (waterSpeed_kmh * hours);
        else if(region.equals("land")) {
            kmTravelled += (landSpeed_kmh * hours);
    }

    public double getKMTravelled() {
        return kmTravelled;
    }
}
```

Interfaces

So let's take a look at the following example

```
public interface Move {
    public void move(double hours);
}
```

We have defined our **Interface Move** that will be implemented by **Dog** and **Dolphin**.

```
public class Dog implements Move {
    private String region; //Water or Land
    private double landSpeed_kmh = 50.0;
    private double waterSpeed_kmh = 8.0;
    private double kmTravelled = 0.0;

    public Dog(String region) {
        this.region = region;
    }

    public void move(double hours) {
        if(region.equals("water"))
            kmTravelled += (waterSpeed_kmh *hours);
        else if(region.equals("land"))
            kmTravelled += (landSpeed_kmh * hours);
    }

    public double getKMTravelled() {
        return kmTravelled;
    }
}
```

```
public class Dolphin implements Move {
    private String region; //Water or Land
    private double landSpeed_kmh = 1.0;
    private double waterSpeed_kmh = 60.0;
    private double kmTravelled = 0.0;

    public Dolphin(String region) {
        this.region = region;
    }

    public void move(double hours) {
        if(region.equals("water"))
            kmTravelled += (waterSpeed_kmh * hours);
        else if(region.equals("land")) {
            kmTravelled += (landSpeed_kmh * hours);
    }

    elled() {
```

They both have a similar implementation but **their** land and water movement speed is different. We could change it completely between the two implementations.

Interfaces

So let's take a look at the following example

```
public interface Move {
    public void move(double hours);
}
```

We have defined our **Interface Move** that will be implemented by **Dog** and **Dolphin**.

```
public class Dog implements Move {
    private String region; //Water or Land
    private double landSpeed_kmh = 50.0;
    private double waterSpeed_kmh = 8.0;
    private double kmTravelled = 0.0;

    public Dog(String region) {
        this.region = region;
    }

    public void move(double hours) {
        if(region.equals("water"))
            kmTravelled += (waterSpeed_kmh *hours);
        else if(region.equals("land"))
            kmTravelled += (landSpeed_kmh * hours);
    }

    public double getKMTravelled() {
        return kmTravelled;
    }
}
```

Since they both implement **Move** interface, we can treat them as a **Move type.**

```
public class Dolphin implements Move {
    private String region; //Water or Land
    private double landSpeed_kmh = 1.0;
    te double waterSpeed_kmh = 60.0;
    te double kmTravelled = 0.0;

    c Dolphin(String region) {
        this.region = region;
    }

    public void move(double hours) {
        if(region.equals("water"))
            kmTravelled += (waterSpeed_kmh * hours);
        else if(region.equals("land")) {
            kmTravelled += (landSpeed_kmh * hours);
    }

    elled() {
```

They both have a similar implementation but **their** land and water movement speed is different. We could change it completely between the two implementations.

Interfaces

So let's take a look at the following example

```
public interface Move {
    public void move(double hours);
}
```

We have defined our **Interface Move** that will be implemented by **Dog** and **Dolphin**.

```
public class Dog implements Move {
    private String region; //Water or Land
    private double landSpeed_kmh = 50.0;
    private double waterSpeed_kmh = 8.0;
    private double kmTravelled = 0.0;

    public Dog(String region) {
        this.region = region;
    }
```

```
public class Dolphin implements Move {
    private String region; //Water or Land
    private double landSpeed_kmh = 1.0;
    te double waterSpeed_kmh = 60.0;
    te double kmTravelled = 0.0;

    c Dolphin(String region) {
        this.region = region;
    }
```

Since they both implement **Move** interface, we can treat them as a **Move type.**

```
public class MovingAnimals {
    public static void main(String[] args) {
        Dog dog = new Dog("land");
        Dolphin dolphin = new Dolphin("land");
        Move[] movingAnimals = {dog, dolphin};

        for(Move m : movingAnimals) {
            m.move(1.0);
        }

        System.out.println(dog.getKMTravelled());
        System.out.println(dolphin.getKMTravelled());
    }
}
```

We can create an **Move[]** array and add both **dog** and **dolphin** types to it. **Why?**

Interfaces

So let's take a look at the following example

```
public interface Move {
    public void move(double hours);
}
```

We have defined our **Interface Move** that will be implemented by **Dog** and **Dolphin**.

```
public class Dog implements Move {
    private String region; //Water or Land
    private double landSpeed_kmh = 50.0;
    private double waterSpeed_kmh = 8.0;
    private double kmTravelled = 0.0;

    public Dog(String region) {
        this.region = region;
    }
```

```
public class Dolphin implements Move {
    private String region; //Water or Land
    private double landSpeed_kmh = 1.0;
    te double waterSpeed_kmh = 60.0;
    te double kmTravelled = 0.0;

    c Dolphin(String region) {
        this.region = region;
    }
```

Since they both implement **Move** interface, we can treat them as a **Move type.**

```
public class MovingAnimals {
    public static void main(String[] args) {
        Dog dog = new Dog("land");
        Dolphin dolphin = new Dolphin("land");
        Move[] movingAnimals = {dog, dolphin};

        for(Move m : movingAnimals) {
            m.move(1.0);
        }

        System.out.println(dog.getKMTravelled());
        System.out.println(dolphin.getKMTravelled());
    }
}
```

We can create an **Move[]** array and add both **dog** and **dolphin** types to it.
Why?
Because they are of type Move.

Interfaces

So let's take a look at the following example

```
public interface Move {
    public void move(double hours);
}
```

We have defined our **Interface Move** that will be implemented by **Dog** and **Dolphin**.

```
public class Dog implements Move {
    private String region; //Water or Land
    private double landSpeed_kmh = 50.0;
    private double waterSpeed_kmh = 8.0;
    private double kmTravelled = 0.0;

    public Dog(String region) {
        this.region = region;
    }
```

```
public class Dolphin implements Move {
    private String region; //Water or Land
    private double landSpeed_kmh = 1.0;
    te double waterSpeed_kmh = 60.0;
    te double kmTravelled = 0.0;

    c Dolphin(String region) {
        this.region = region;
    }
```

Since they both implement **Move** interface, we can treat them as a **Move type.**

```
public class MovingAnimals {
    public static void main(String[] args) {
        Dog dog = new Dog("land");
        Dolphin dolphin = new Dolphin("land");
        Move[] movingAnimals = {dog, dolphin};

        for(Move m : movingAnimals) {
            m.move(1.0);
        }

        System.out.println(dog.getKMTravelled());
        System.out.println(dolphin.getKMTravelled());
    }
}
```

If they of **type Move** we are guaranteed to be able to use **move()** method.

Interfaces

So let's take a look at the following example

```
public interface Move {
    public void move(double hours);
}
```

We have defined our **Interface Move** that will be implemented by **Dog** and **Dolphin**.

```
public class Dog implements Move {
    private String region; //Water or Land
    private double landSpeed_kmh = 50.0;
    private double waterSpeed_kmh = 8.0;
    private double kmTravelled = 0.0;

    public Dog(String region) {
        this.region = region;
    }
```

```
public class Dolphin implements Move {
    private String region; //Water or Land
    private double landSpeed_kmh = 1.0;
    te double waterSpeed_kmh = 60.0;
    te double kmTravelled = 0.0;

    c Dolphin(String region) {
        this.region = region;
    }
```

Since they both implement **Move** interface, we can treat them as a **Move type.**

```
public class MovingAnimals {
    public static void main(String[] args) {
        Dog dog = new Dog("land");
        Dolphin dolphin = new Dolphin("land");
        Move[] movingAnimals = {dog, dolphin};

        for(Move m : movingAnimals) {
            m.move(1.0);
        }

        System.out.println(dog.getKMTravelled());
        System.out.println(dolphin.getKMTravelled());
    }
}
```

We can see the updated variables that have been applied to both objects.

Interfaces

So let's take a look at the following example

```
public interface Move {
    public void move(double hours);
}
```

We have defined our **Interface Move** that will be implemented by **Dog** and **Dolphin**.

```
public class Dog implements Move {
    private String region; //Water or Land
    private double landSpeed_kmh = 50.0;
    private double waterSpeed_kmh = 8.0;
    private double kmTravelled = 0.0;

    public Dog(String region) {
        this.region = region;
    }
```

```
public class Dolphin implements Move {
    private String region; //Water or Land
    private double landSpeed_kmh = 1.0;
    te double waterSpeed_kmh = 60.0;
    te double kmTravelled = 0.0;

    c Dolphin(String region) {
        this.region = region;
    }
```

Since they both implement **Move** interface, we can treat them as a **Move type.**

```
public class MovingAnimals {
    public static void main(String[] args) {
        Dog dog = new Dog("land");
        Dolphin dolphin = new Dolphin("land");
        Move[] movingAnimals = {dog, dolphin};

        for(Move m : movingAnimals) {
            m.move(1.0);
        }

        System.out.println(dog.getKMTravelled());
        System.out.println(dolphin.getKMTravelled());
    }
}
```

```
> java MovingAnimals
50.0
1.0
<program end>
```

We can then see that **move()** has changed an **internal** travelled variable.

Using interfaces!

Note: Interfaces

Okay, I lied a little, we can have attributes in an interface.

However! The attributes are:

- Static (They belong to the interface)
- Constant (have the **final** modifier applied to them)

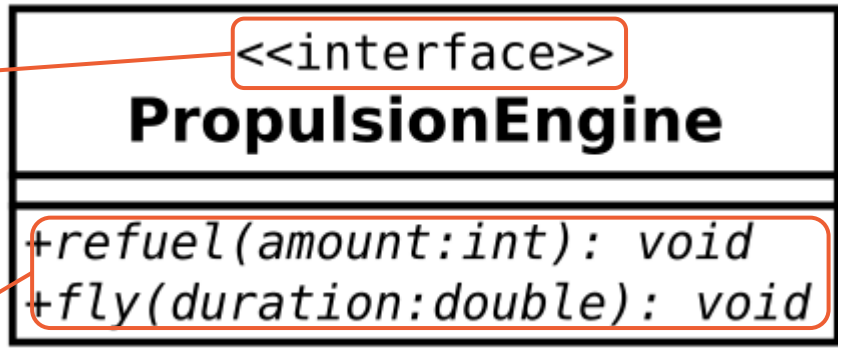
Therefore we cannot use them for instances.

Interface and UML

Just like abstract classes we can represent an **interface** within UML however it is slightly different than others.

We specify the stereotype in UML to be interface and this gives us specificity of language constructs.

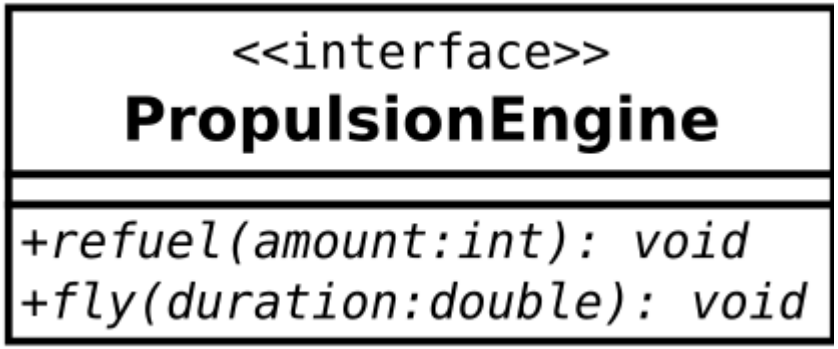
Italicised font shows that it is a polymorphic method



Interface and UML

Just like abstract classes we can represent an **interface** within UML however it is slightly different than others.

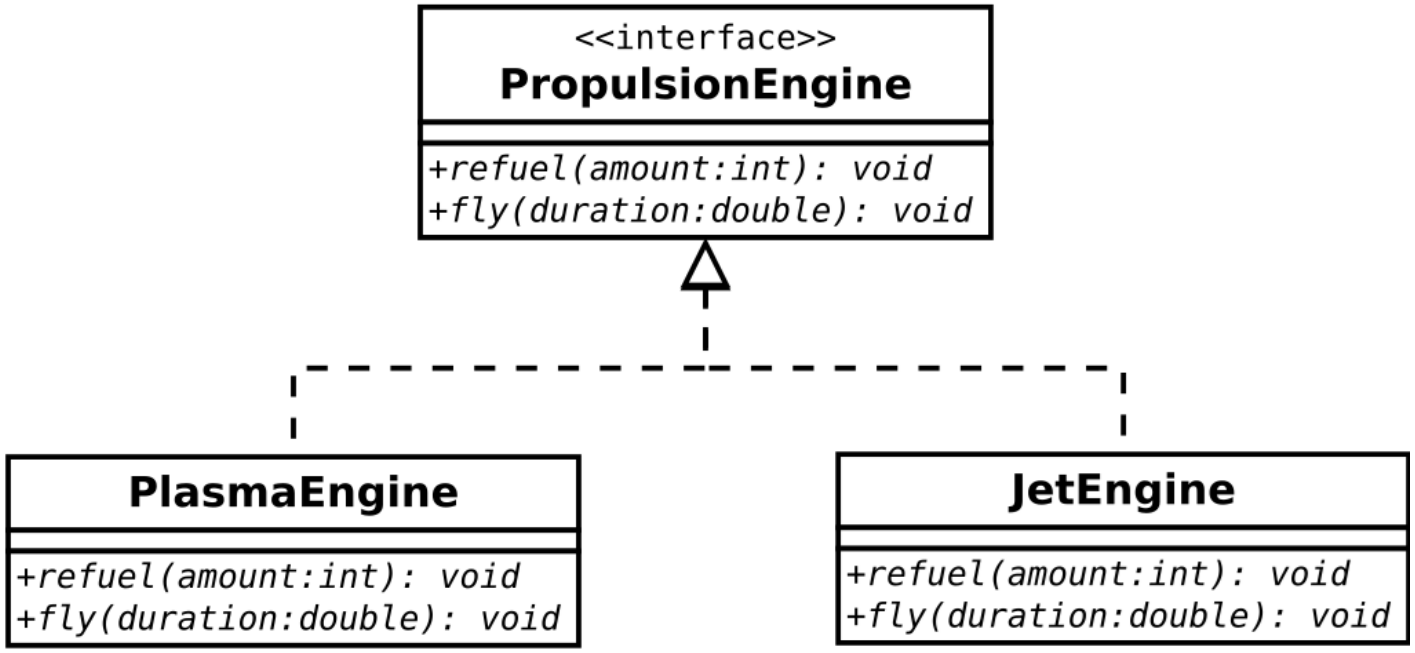
However! The relationship link is different than that of a classes.



Interface and UML

Just like abstract classes we can represent an **interface** within UML however it is slightly different than others.

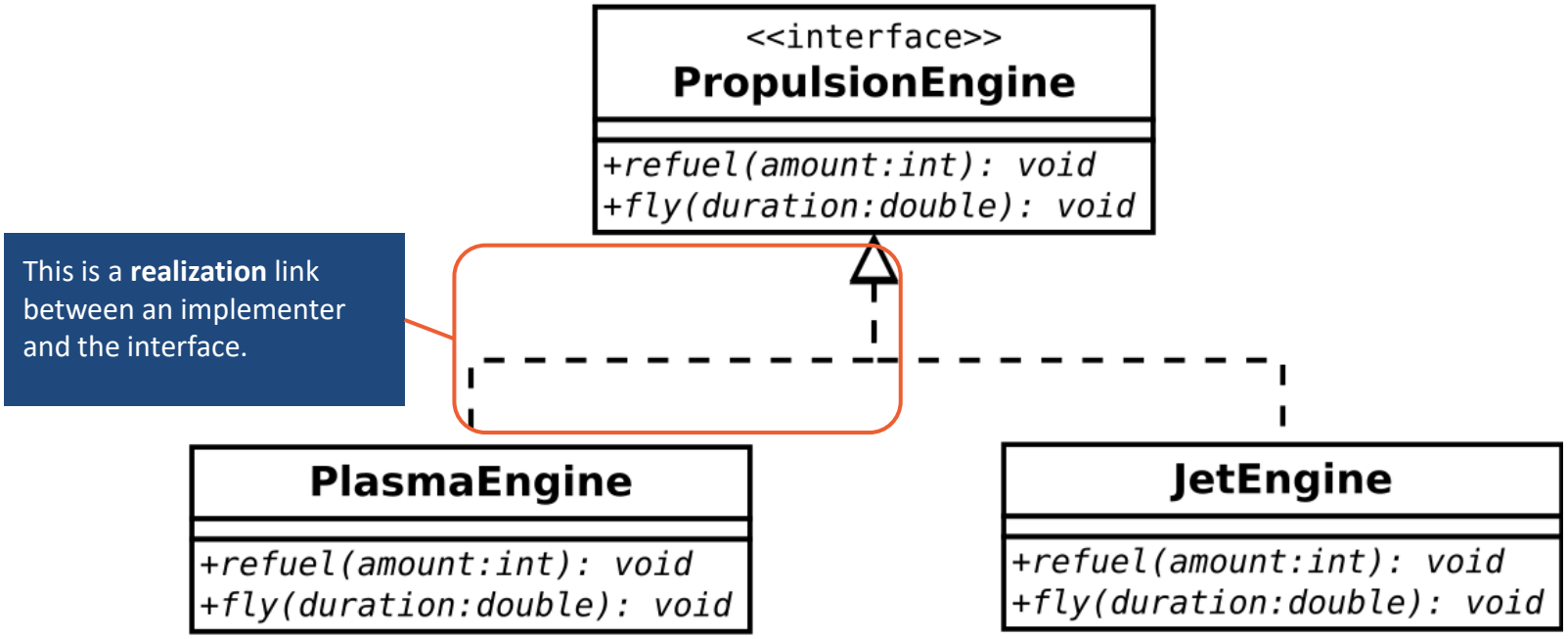
However! The relationship link is different than that of a classes.



Interface and UML

Just like abstract classes we can represent an **interface** within UML however it is slightly different than others.

However! The relationship link is different than that of a classes.



Let's take a break!

THE UNIVERSITY OF SYDNEY

Topics: Part B

- **Default Method in Interfaces**
- **Packaging**

Default Method

We know interfaces and now we will be visiting default methods with java and their utility. This is a new feature in **Java** that allows methods to be defined in an interface.

Prior to Java 8, interfaces just specified the method declaration and never a default method.

Syntax of a default method

Simply we are able to define a default method by using the **default** keyword.

Syntax:

[modifier] **default** <returntype> MethodName([parameters])

Example:

```
private default void swim();
```

Default Method

```
interface Talk {
    public void talk();
    public default void talking(){
            System.out.println("I am talking.");
    }
}
```

```
public class Alien implements Talk {

    public void talk() {
        System.out.println("zzzfer342aa");
    }
}
```

```
public class Cat implements Talk {

    public void talk() {
        System.out.println("meow");
    }
}
```

Both **Alien** and **Cat** implement the talk behaviour through the interface.

Default Method

```
interface Talk {
    public void talk();
    public default void talking(){
            System.out.println("I am talking.");
    }
}
```

```
public class Alien implements Talk {

    public void talk() {
        System.out.println("zzzfer342aa");
    }
}
```

```
public class Cat implements Talk {

    public void talk() {
        System.out.println("meow");
    }

    public void talking() {
        System.out.println("Overridden in Cat");
    }
}
```

Subclass may override the default method if needed

Demonstration

Organising your application

Classpath

A classpath defines a set of directories exposed to our program. It will allow us to use libraries constructed by others.

We are able to cleanly separate and structure our code into different directories and refer to different segments as if it was in the same directory.

Classpath

Given a directory that will allow us to store our class file, we will be able to use them directly within our project.

```
> javac -cp .:<Your directory or jar file here>[:<more>]
```

Demo, using multiple location of class files

Packages

Java defines a package keyword which will outline to the class which part of the package it resides in. It will self verify on compilation if it exists within the package.

Syntax:

package <identifier>[.<nested ident>[...]]

Packages

Java defines a package keyword which will outline to the class which part of the package it resides in. It will self verify on compilation if it exists within the package.

Syntax:

package <identifier>[.<nested ident>[...]]

Example:

```
package com.whiteboard;

package com.whiteboard.render;
```

Packages

Java defines a package keyword which will outline to the class which part of the package it resides in. It will self verify on compilation if it exists within the package.

Syntax:

package <identifier>[.<nested ident>[...]]

Example:

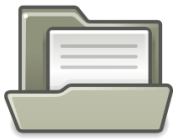
```
package com.whiteboard;

package com.whiteboard.render;
```

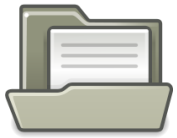
Typically set at the top of your java file, specifies directory it is in.

Packages

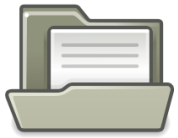
Let's look at the layout of a package



./src



./src/whiteboard



./src/whiteboard/render

./src/whiteboard/input

./src/whiteboard/Whiteboard.java

./src/whiteboard/render/Drawable.java

./src/whiteboard/render/PositionData.java

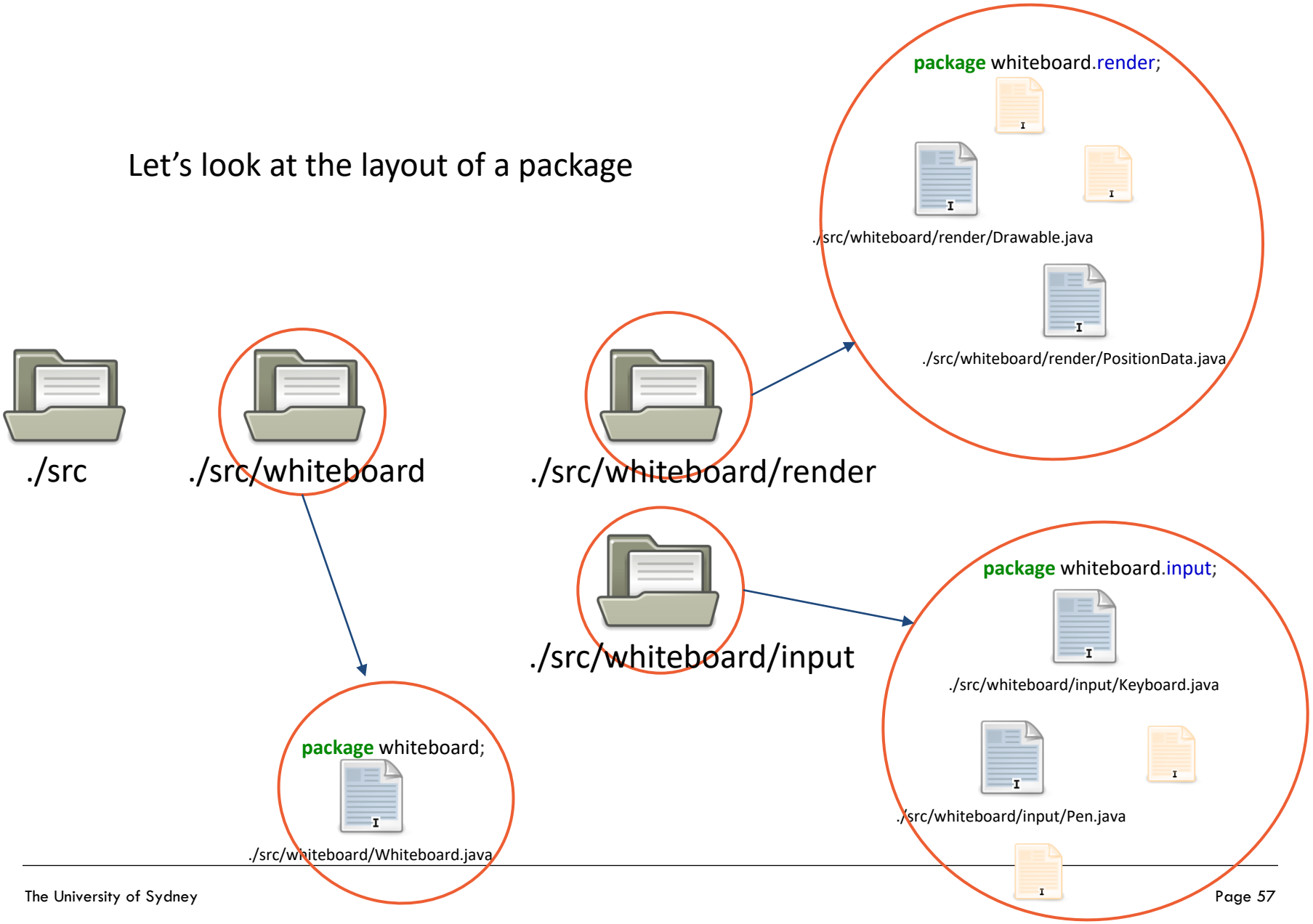
./src/whiteboard/input/Keyboard.java

./src/whiteboard/input/Pen.java

Given the current package layout, what would be the package name of each class?

Packages

Let's look at the layout of a package



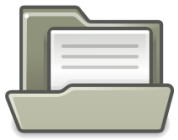
Packages

So we have laid out the package as the following:

./src

./src/telephone

./src/telephone/state



./src/telephone/exceptions

Packages

```
package telephone;
public class Telephone {

   private TelephoneState state;

   public Telephone() {
      state = new LineWaiting();
   }

   public void dial(String phonenumber) {
      state = state.dial(phonenumber);
   }

   public void hangup() {
      state = state.hangup();
   }

   public static void main(String[] args) {
      Telephone phone = new Telephone();
      phone.dial("12341234");
      phone.hangup();
   }

}
```

We specify above our above classes and typically above majority of our code, the package name for the file.

Packages

```
package telephone.state;
public abstract class TelephoneState {

   protected String numberDialed;

   public abstract TelephoneState dial(String phonenumber);

   public abstract TelephoneState hangup();

}
```

```
package telephone.state;
public class LineBusy extends TelephoneState {

   public LineBusy(String number) {
      super();
      numberDialed = number;
   }

   public TelephoneState dial(String phonenumber) {
      throw new InvalidPhoneState();
   }

   public TelephoneState hangup() {
      System.out.println("Hanging up: " + numberDialed);
      return new LineWaiting();
   }

}
```

```
package telephone.state;
public class LineWaiting extends TelephoneState {

   public TelephoneState dial(String phonenumber) {
      System.out.println("Dialing: " + phonenumber);
      return new LineBusy(phonenumber);
   }

   public TelephoneState hangup() {
      throw new InvalidPhoneState();
   }

}
```

We specify the package name within each state class.

Packages

However! We now need to import these classes into our code so we are able to use them.

```
package telephone.state;
public abstract class TelephoneState {

    protected String numberDialed;

    public abstract TelephoneState dial(String phonenumber);

    public abstract TelephoneState hangup();

}
```

```
package telephone.state;
public class LineBusy extends TelephoneState {

    public LineBusy(String number) {
        super();
        numberDialed = number;
    }

    public TelephoneState dial(String phonenumber) {
        throw new InvalidPhoneState();
    }

    public TelephoneState hangup() {
        System.out.println("Hanging up: " + numberDialed);
        return new LineWaiting();
    }

}
```

```
package telephone.state;
public class LineWaiting extends TelephoneState {

    public TelephoneState dial(String phonenumber) {
        System.out.println("Dialing: " + phonenumber);
        return new LineBusy(phonenumber);
    }

    public TelephoneState hangup() {
        throw new InvalidPhoneState();
    }

}
```

We specify the package name within each state class.

Packages

```
package telephone;
import telephone.state.TelephoneState;
import telephone.state.LineWaiting;

public class Telephone {

   private TelephoneState state;

   public Telephone() {
      state = new LineWaiting();
   }

   public void dial(String phonenumber) {
      state = state.dial(phonenumber);
   }

   public void hangup() {
      state = state.hangup();
   }

   public static void main(String[] args) {
      Telephone phone = new Telephone();
      phone.dial("12341234");
      phone.hangup();
   }
}
```

Our state classes exist is a different package space name, therefore it is unaware they exist.

We will need to import them into our application to utilise them in our code.

Package Demo

How could we create an archive?

Java Archives

Java provides an archiving format that allows you to compress the files you want to export and distribute to other.

This kind of format is similar to other OS/Package manager specific formats such as **.dmg, .apk, .xdg** and **.deb**.

Java Archives

To create an archive file, you will need to utilise the **jar** command. We are able to store any kind of data within a java archive but its typical case is bundling and packaging of libraries and applications.

```
> jar -cf MyProgram.jar <list of files>
```

Java Archives

To create an archive file, you will need to utilise the **jar** command. We are able to store any kind of data within a java archive but its typical case is bundling and packaging of libraries and applications.

```
> jar -cf MyProgram.jar <list of files>
```

Specifies the create and file flag for the **jar** program.

We specify the Jar file to produce and input .class files to be included in the archive.

Java Archives

.jar Manifest files provide a simple description of requirements your archive files needs.

A common setting is providing an Application Entry point for your .jar file.

By default, creating an archive file will only index the files you have added to it. It will not know what **.class** file you want to execute. You will need to specify that by hand.

Let’s generate a .jar file

Build Tools

Is there a better way?

Yes! you can look into using the following:

- Apache Ant
- Apache Maven
- Gradle

Gradle can be used for building more complex java applications that will involve testing

Each build system intends to make it easier to incorporate libraries, run tests and create multiple application builds.

See you next time!

THE UNIVERSITY OF SYDNEY