

INFO1113 / COMP9003

Object-Oriented Programming

Lecture 6



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**

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.

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**.

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");
    }

}
```

```
> 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 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.

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

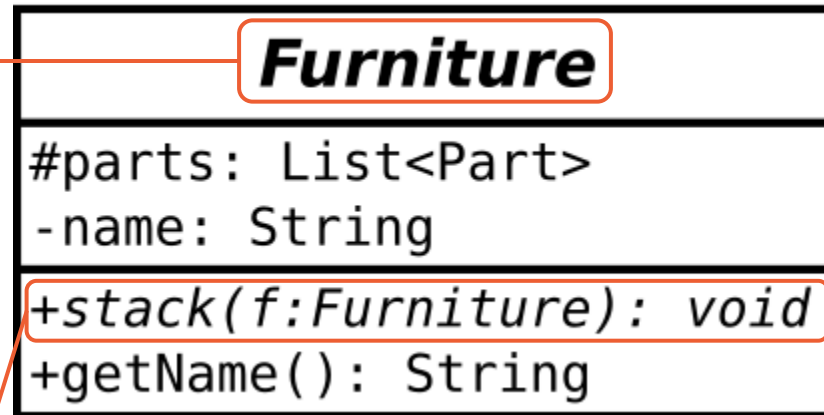
We can now declare and invoke **stack** through **Chair** class.

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.

Let's take a break!



THE UNIVERSITY OF
SYDNEY

Topics: Part B

- **Interfaces**
- **Interfaces and UML**
- **Default Method in Interfaces**

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.

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:

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

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

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

```
public class Dog implements 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);  
        }  
    }
```

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

```
    public double getKMTravelled() {
```

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);  
        }  
    }
```

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

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;  
    private double waterSpeed_kmh = 60.0;  
    private double kmTravelled = 0.0;  
  
    public 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;  
    private double waterSpeed_kmh = 60.0;  
    private double kmTravelled = 0.0;  
  
    public 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;  
    private double waterSpeed_kmh = 60.0;  
    private double kmTravelled = 0.0;  
  
    public 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;  
    private double waterSpeed_kmh = 60.0;  
    private double kmTravelled = 0.0;  
  
    public 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;  
    private double waterSpeed_kmh = 60.0;  
    private double kmTravelled = 0.0;  
  
    public 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)

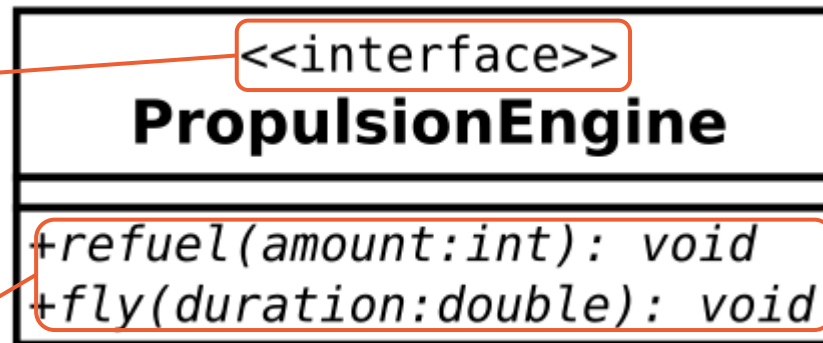
Therefore we cannot use them for instances.

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.

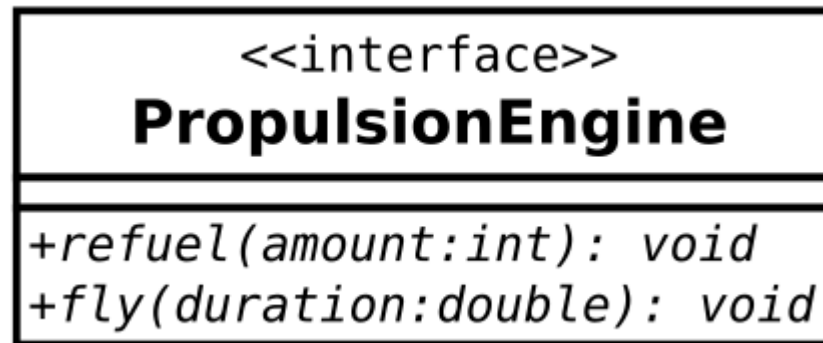
Italicised font shows that it is a polymorphic method



Interface and UML

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

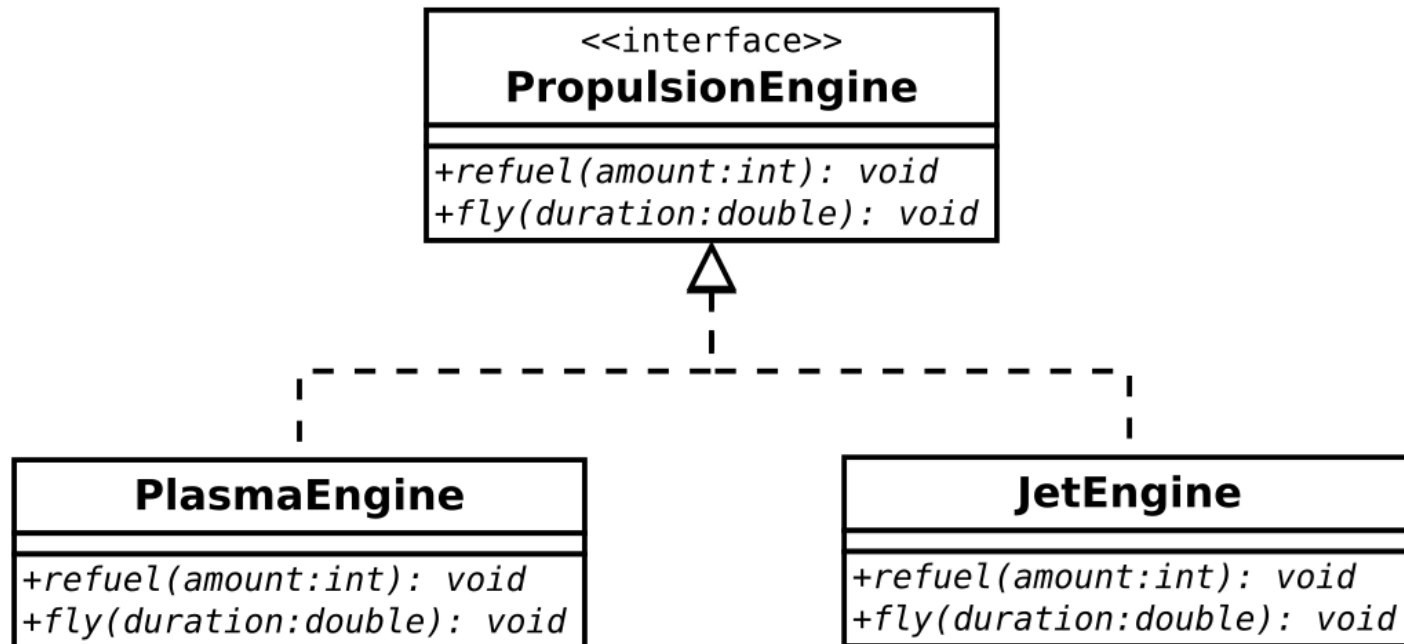
However! The relationship link is different than that of a classes.



Interface and UML

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

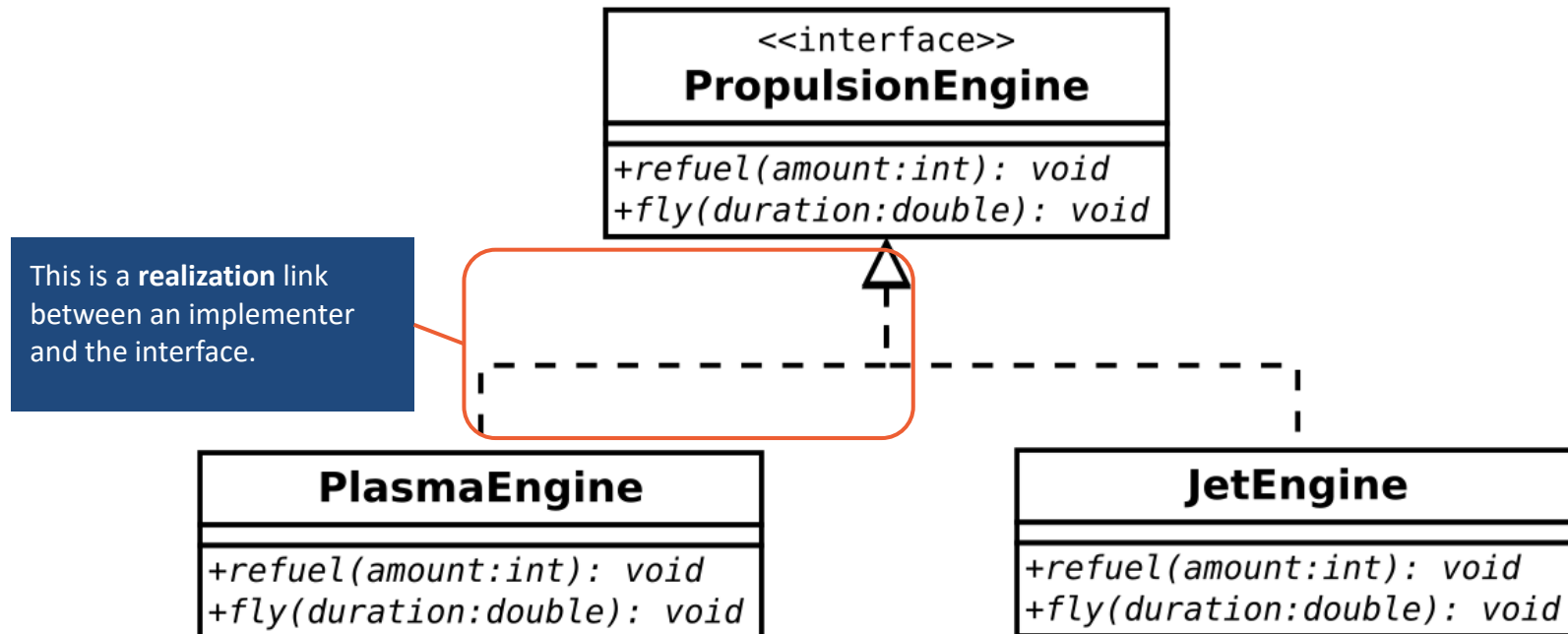
However! The relationship link is different than that of a classes.



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.



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

See you next time!



THE UNIVERSITY OF
SYDNEY