INFO1113 Object-Oriented Programming

Week 6A: Abstract Classes and Interfaces

Copyright Warning

Topics

- Abstract Classes (s. 4)
- Abstract Classes UML (s. 24)
- Interfaces (s. 27)
- Interfaces and UML (s. 47)

Inheritance

As noted last week, we are able to extend a class to a subclass. We will be visiting how we are able to define **abstract** classes, how they are used and when to use them.

We will also be visiting interfaces and discussing about the different perspectives between class and inheritance based relationships.

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

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!*

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

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.**

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

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
```

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 typical of instantiation.

But there is a little more at work here when we mark something as **abstract**.

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

What if we try to instantiate it?

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

But there is a little more at work here when we mark something as **abstract**.

This is because we can mark methods as being abstract as well.

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

```
> 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);
```

If the class should not be instantiated or behaviour is defined by the subtypes and not the super type the class should be non-instantiable.

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);
}
```

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);
}
```

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

Declaration of an abstract class

We have an **abstract** class specified.

```java
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);
}
```

```java
public class WoodChair extends Furniture {

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

}
```

However, in this class we have not defined the method **stack**.

Declaration of an abstract class

We have an **abstract** class specified.

```java
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);
}
```

```java
public class WoodChair extends Furniture {

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

}
```

However, in this class we have not defined the method **stack**.

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

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

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

Declaration of an abstract class

We have an **abstract** class specified.

```java
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);
}
```

```java
public class WoodChair extends Furniture {

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

    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.

```java
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>();
    }
```

```java
public class WoodChair extends Furniture {

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

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

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

```java
public class FurnitureStore {

    public static void main(String[] args) {
        WoodChair f = new WoodChair();
        f.stack(new WoodChair());
    }
}
```

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

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

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 class WoodChair extends Furniture {

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

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

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

```
public class FurnitureStore {

    public static void main(String[] args) {
        Furniture f = new WoodChair();
        f.stack(new WoodChair());
    }
}
```

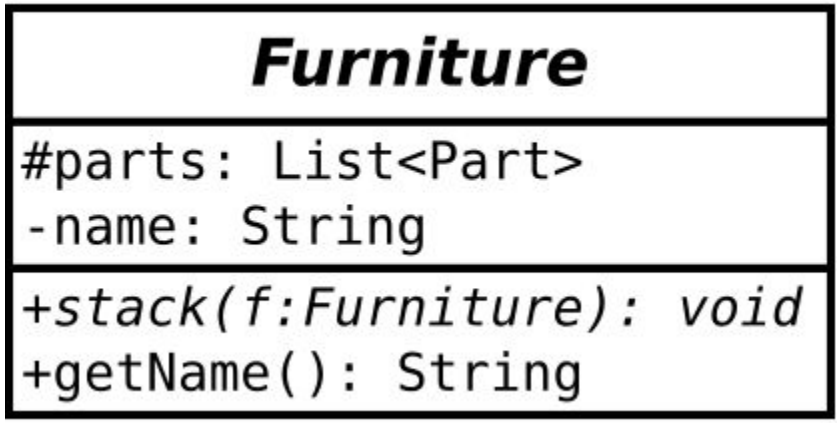
We can even bind it to **Furniture** type and **invoke** stack which will call the subtype's method

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

Neat! Let’s play around with it and see what other types we can create

Abstract Classes and UML

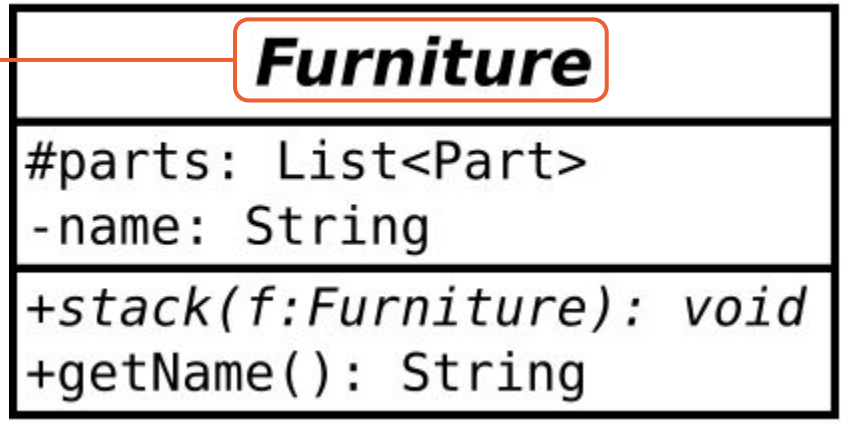
Within a UML class diagram, we can illustrate abstract classes with the following.



Abstract Classes and UML

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

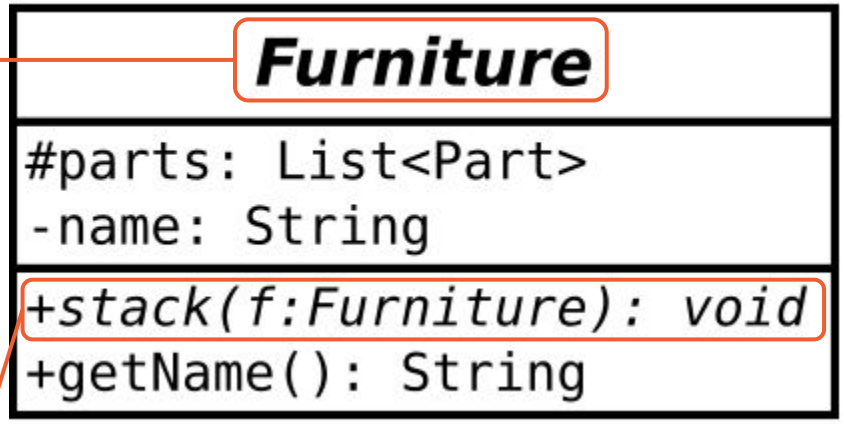
Italicised font shows that it is an abstract class.



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.

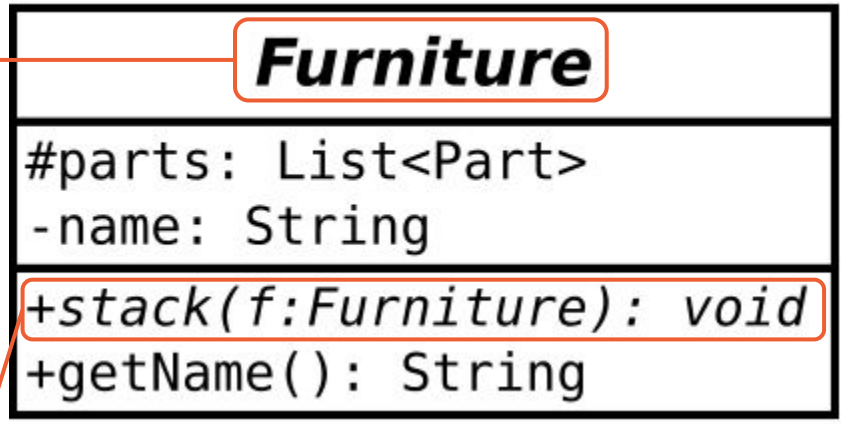


We also show **polymorphic** method as italicised.

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.

This is convention for UML 2.

Interfaces

We will now look at **interfaces.**

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

Interfaces share a similarity with **Abstract Classes** in that they declare methods that a class 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

Maybe it might be best to ask **Why would we want to do that?**

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.

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

Declaration of an interfaces

Simply we are able to define an **interface** by using the **interface keyword**. Afterwards we will need to define a few

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**. Afterwards we will need to define a few

Syntax:

```
[modifier] interface InterfaceName
```

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

Example:

```
public interface Swim
```

Declaration of an interfaces

Simply we are able to define an **interface** by using the **interface keyword**. Afterwards we will need to define a few

Syntax:

```
[modifier] interface InterfaceName
```

Example:

```
public interface Swim {
    public void swim();
    public void dive();
}
```

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**. Afterwards we will need to define a few

Syntax:

```
[modifier] interface InterfaceName
```

Example:

```
public interface Swim {
    public void swim();
    public void dive();
}
```

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.**

```
public class Dog implements Swim
```

Interfaces

So let's take a look at the following example (WHOA!)

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

```
public class Dog implements Move {
    private String region; //Water or Land
    private final double LAND_MOVEMENT_SPEED_KMH = 50.0;
    private final double WATER_MOVEMENT_SPEED_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 += (WATER_MOVEMENT_SPEED_KMH *hours);
        } else if(region.equals("land")) {
            kmTravelled += (LAND_MOVEMENT_SPEED_KMH * hours);
        }
    }

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

```
public class Dolphin implements Move {
    private String region; //Water or Land
    private final double LAND_MOVEMENT_SPEED_KMH = 1.0;
    private final double WATER_MOVEMENT_SPEED_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 += (WATER_MOVEMENT_SPEED_KMH * hours);
        } else if(region.equals("land")) {
            kmTravelled += (LAND_MOVEMENT_SPEED_KMH * hours);
        }
    }

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

Okay! It’s a lot, but let’s try and distill it

Interfaces

So let's take a look at the following example (WHOA!)

```
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 final double LAND_MOVEMENT_SPEED_KMH = 50.0;
    private final double WATER_MOVEMENT_SPEED_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 += (WATER_MOVEMENT_SPEED_KMH *hours);
        } else if(region.equals("land")) {
            kmTravelled += (LAND_MOVEMENT_SPEED_KMH * hours);
        }
    }

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

```
public class Dolphin implements Move {
    private String region; //Water or Land
    private final double LAND_MOVEMENT_SPEED_KMH = 1.0;
    private final double WATER_MOVEMENT_SPEED_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 += (WATER_MOVEMENT_SPEED_KMH * hours);
        } else if(region.equals("land")) {
            kmTravelled += (LAND_MOVEMENT_SPEED_KMH * hours);
        }
    }

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

Interfaces

So let's take a look at the following example (WHOA!)

```
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 final double LAND_MOVEMENT_SPEED_KMH = 50.0;
    private final double WATER_MOVEMENT_SPEED_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 += (WATER_MOVEMENT_SPEED_KMH *hours);
        } else if(region.equals("land")) {
            kmTravelled += (LAND_MOVEMENT_SPEED_KMH * hours);
        }
    }

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

```
public class Dolphin implements Move {
    private String region; //Water or Land
    private final double LAND_MOVEMENT_SPEED_KMH = 1.0;
    private final double WATER_MOVEMENT_SPEED_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 += (WATER_MOVEMENT_SPEED_KMH * hours);
        } else if(region.equals("land")) {
            kmTravelled += (LAND_MOVEMENT_SPEED_KMH * hours);
        }
    }

KMTravelled() {
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 (WHOA!)

```
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 final double LAND_MOVEMENT_SPEED_KMH = 50.0;
    private final double WATER_MOVEMENT_SPEED_KMH = 10.0;
    private double kmTravelled = 0.0;

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

    public void move(double hours) {
        if(region.equals("water")) {
            kmTravelled += (WATER_MOVEMENT_SPEED_KMH *hours);
        } else if(region.equals("land")) {
            kmTravelled += (LAND_MOVEMENT_SPEED_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 final double LAND_MOVEMENT_SPEED_KMH = 1.0;
    private final double WATER_MOVEMENT_SPEED_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 += (WATER_MOVEMENT_SPEED_KMH * hours);
        } else if(region.equals("land")) {
            kmTravelled += (LAND_MOVEMENT_SPEED_KMH * hours);
        }
    }

    KMTravelled() {
        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 (WHOA!)

```
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 final double LAND_MOVEMENT_SPEED_KMH = 50.0;
    private final double WATER_MOVEMENT_SPEED_KMH = 10.0;
    private double kmTravelled = 0.0;

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

```
public class Dolphin implements Move {
    private String region; //Water or Land
    private final double LAND_MOVEMENT_SPEED_KMH = 1.0;
    private final double WATER_MOVEMENT_SPEED_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 (WHOA!)

```
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 final double LAND_MOVEMENT_SPEED_KMH = 50.0;
    private final double WATER_MOVEMENT_SPEED_KMH = 10.0;
    private double kmTravelled = 0.0;

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

```
public class Dolphin implements Move {
    private String region; //Water or Land
    private final double LAND_MOVEMENT_SPEED_KMH = 1.0;
    private final double WATER_MOVEMENT_SPEED_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.
Because they are of type Move.

Interfaces

So let's take a look at the following example (WHOA!)

```
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 final double LAND_MOVEMENT_SPEED_KMH = 50.0;
    private final double WATER_MOVEMENT_SPEED_KMH = 10.0;
    private double kmTravelled = 0.0;

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

```
public class Dolphin implements Move {
    private String region; //Water or Land
    private final double LAND_MOVEMENT_SPEED_KMH = 1.0;
    private final double WATER_MOVEMENT_SPEED_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 (WHOA!)

```
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 final double LAND_MOVEMENT_SPEED_KMH = 50.0;
    private final double WATER_MOVEMENT_SPEED_KMH = 10.0;
    private double kmTravelled = 0.0;

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

```
public class Dolphin implements Move {
    private String region; //Water or Land
    private final double LAND_MOVEMENT_SPEED_KMH = 1.0;
    private final double WATER_MOVEMENT_SPEED_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.

Reflecting on your experience with **Python** or **Javascript. Why do we not need to treat them as a certain type in those languages?**

Interfaces

So let's take a look at the following example (WHOA!)

```
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 final double LAND_MOVEMENT_SPEED_KMH = 50.0;
    private final double WATER_MOVEMENT_SPEED_KMH = 10.0;
    private double kmTravelled = 0.0;

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

```
public class Dolphin implements Move {
    private String region; //Water or Land
    private final double LAND_MOVEMENT_SPEED_KMH = 1.0;
    private final double WATER_MOVEMENT_SPEED_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.

Reflecting on your experience with **Python** or **Javascript. Why do we not need to treat them as a certain type in those languages? Simply: They are not statically typed.**

Interfaces

So let's take a look at the following example (WHOA!)

```
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 final double LAND_MOVEMENT_SPEED_KMH = 50.0;
    private final double WATER_MOVEMENT_SPEED_KMH = 10.0;
    private double kmTravelled = 0.0;

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

```
public class Dolphin implements Move {
    private String region; //Water or Land
    private final double LAND_MOVEMENT_SPEED_KMH = 1.0;
    private final double WATER_MOVEMENT_SPEED_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 (WHOA!)

```
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 final double LAND_MOVEMENT_SPEED_KMH = 50.0;
    private final double WATER_MOVEMENT_SPEED_KMH = 10.0;
    private double kmTravelled = 0.0;

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

```
public class Dolphin implements Move {
    private String region; //Water or Land
    private final double LAND_MOVEMENT_SPEED_KMH = 1.0;
    private final double WATER_MOVEMENT_SPEED_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 variables in an interface.

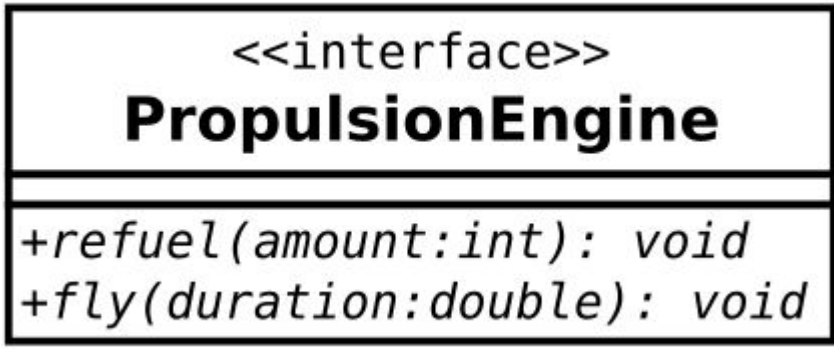
However! The variables are:

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

Therefore we cannot use them for instances.

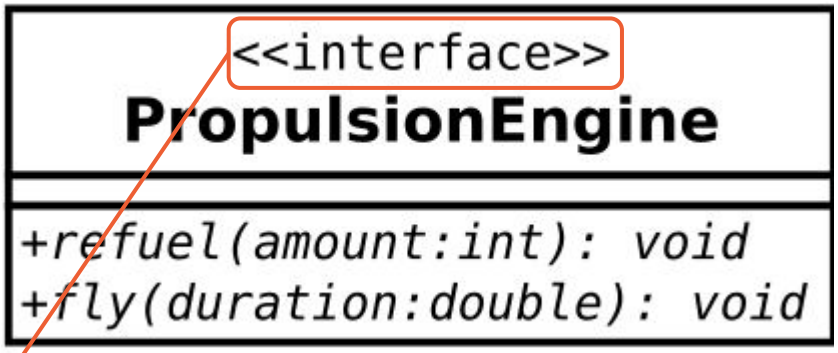
Abstract Classes and UML

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



Abstract Classes and UML

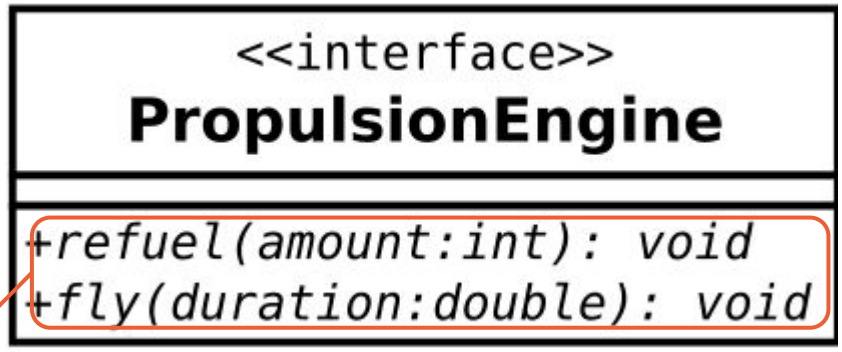
Just like **abstract** classes we can represent an interface within UML however it is slightly different than others.



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

Abstract Classes and UML

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

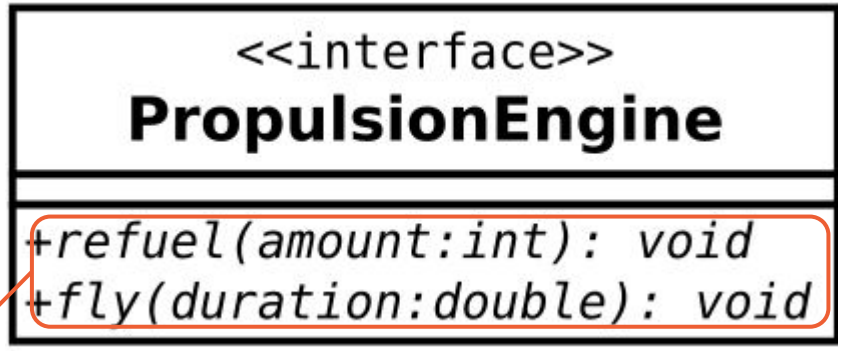


Italicised font shows that it is a polymorphic method

Abstract Classes 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.**

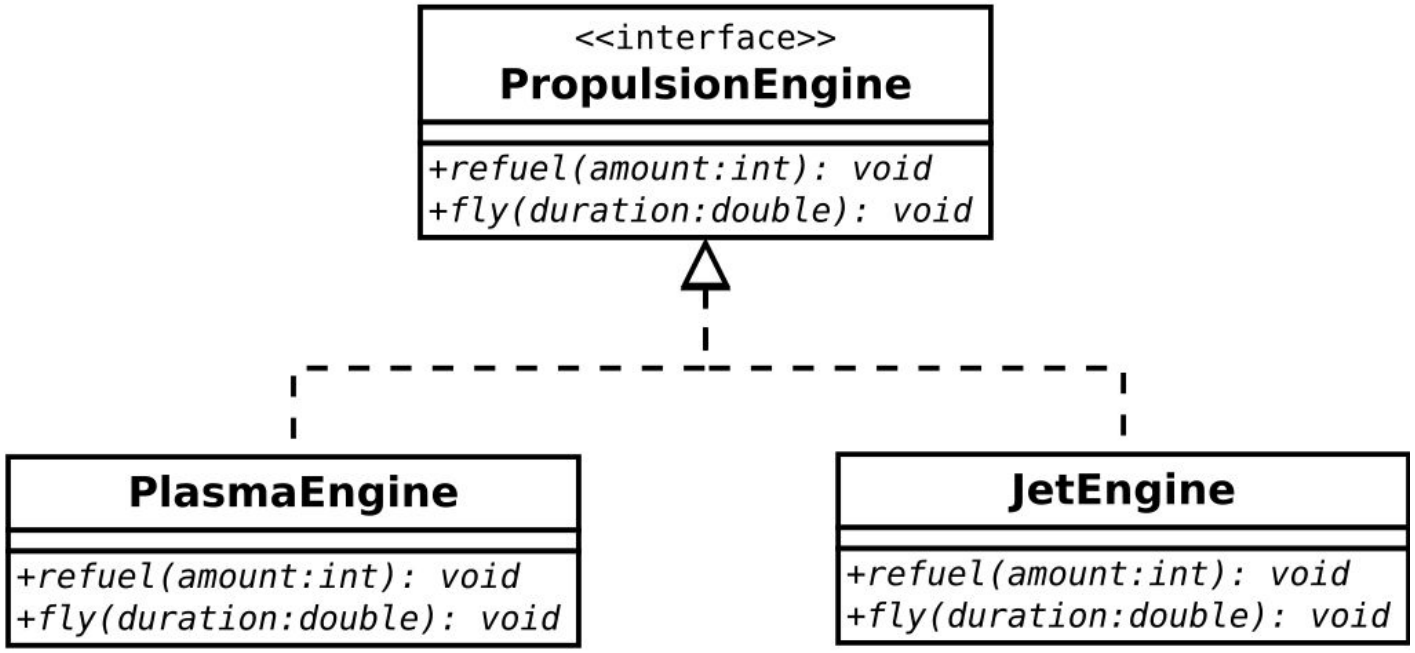


Italicised font shows that it is a polymorphic method

Abstract Classes and UML

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

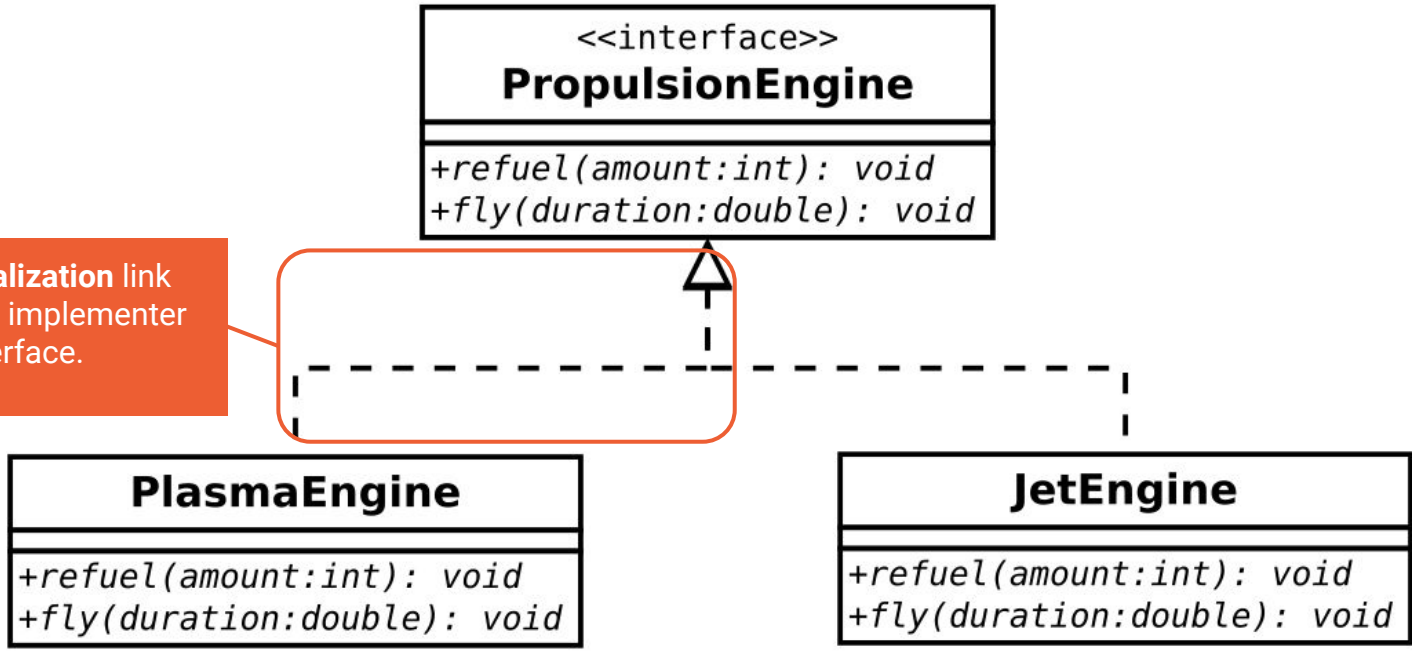
However! **The relationship link is different than that of a classes.**



Abstract Classes 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.



See you next time!