Exercise 11

Inheritance

By the end of this exercise you will be able to

- Understand the idea of *inheritance* and know why it is useful.
- Understand how the `protected` modifier relates to inheritance.
- Use an `abstract` class.

Introduction

Inheritance is a way for one class to borrow methods and properties from another class. The borrower class is called the *subclass* and the class that is borrowed from is called the *superclass*. The code below shows a `Dog` class inheriting from an `Animal` class using the `extends` keyword:

```
class Animal {

   // Properties of the class...
   public int numberOfLegs;

   // Methods of the class...
   public void talk() {
      System.out.println("Hello");
   }
}

class Dog extends Animal {

   // Properties of the class...
   public int numberOfFleas;

   // Constructor of the class...
   public Dog() {
      numberOfLegs  = 4;
      numberOfFleas = 10;
   }

   // Methods of the class...
   public void bark() {
      System.out.println("Woof woof");
   }
   public void scratch() {
      if (numberOfFleas > 0) numberOfFleas--;
   }
}
```

Your previous experience of using references in Java will tell you that if we have a `Dog` reference `d` then we can use it to access the methods and properties of the `Dog` class like so:

```
d.bark();
d.scratch();
System.out.println(d.numberOfFleas);
```

Because the `Dog` class inherits from the `Animal` class, we can also use the `d` reference to access the methods and properties of the `Animal` class:

```
d.talk();
System.out.println(d.numberOfLegs);
```

By enabling the `d` reference to access methods and properties of the `Animal` class, the `Dog` class has effectively borrowed these methods and properties from the `Animal` class. Question 1 takes you through a detailed example and question 2 shows you a more practical example of inheritance being used in a program to reduce the amount of duplication of code.

Questions

1. **Theoretical example of inheritance.**

   (a) Fetch the file `AnimalTest.java` and examine the source code to see four classes: `Animal`, `Bird` and `Eagle` and a tester class `AnimalTest`. Class `Bird` inherits from `Animal` and class `Eagle` inherits from `Bird`.

   (b) Use your understanding of inheritance to say which of the statements will not compile and why. Then use the compiler to check that your guesses were correct and comment out all the statements with errors in them.

   For each statement that doesn't compile, comment it out and beside it write a brief reason why the compiler doesn't accept it.

   NOTE: It is essential for the next question that the incorrect lines are commented out.

   (c) Run the application `AnimalTest.class` to see what is printed to the screen.

   (d) Why is the number printed out for `e.numberOfLegs` equal to 2 when there is no mention of ever setting the value for `numberOfLegs` to 2 in the `Eagle` class?

   HINT: It has something to do with inheritance.

   (e) Add the following three lines to the `main` method:

   ```
   a = b;
   a.talk();
   a.fly();
   ```

   If you compile the code, one of these lines gives a compiler error. Which line is it and what is the reason for the error?

   (f) Add the following three lines to the `main` method:

   ```
   b = a;
   b.talk();
   b.fly();
   ```

   If you compile the code, one of these lines gives a compiler error. Which line is it and what is the reason for the error?

2. **Practical example of inheritance.**

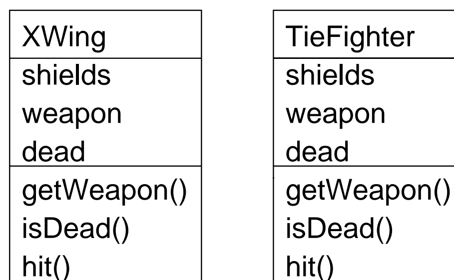
(a) Fetch the file `StarWars.java` which contains a program to simulate a battle between the two enemies of the *Star Wars* trilogy, the Rebel Alliance (the "goodies") and the Dark Side (the "baddies"). Run the program to see the battle unfold.

The file `StarWars.java` is comprised of three classes: `XWing`, `TieFighter` and `StarWars`. The first two represent spacecraft of the goodies and the baddies, respectively. The third class contains code for running the battle.

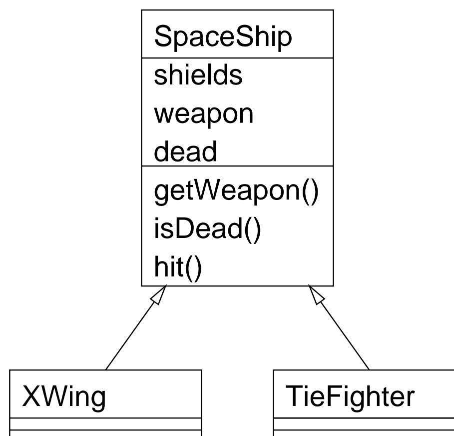
If you look at the Java code for the `XWing` and `TieFighter` classes you will notice that they are almost identical: They have the same methods and properties, the only difference is that the X-Wings are initialised with a different value for their `shields` and `weapon` properties to the `TieFighter` class.

The next few questions will guide you through the process of using inheritance to eliminate this unnecessary duplication of code.

(b) Here is the UML diagram for the `XWing` and `TieFighter` classes:



You will create a new class called `SpaceShip` and move most of code from the `XWing` and `TieFighter` classes into this class. Then you will make these two classes inherit from `SpaceShip` like so:



The first step in this process is to create the outer shell of the `SpaceShip` class, which you should now add to the file `StarWars.java`:

```
class SpaceShip {
}
```

(c) Move the properties `shields`, `weapon` and `dead` out of the `XWing` and `TieFighter` classes and into the `SpaceShip` class. You must change the privacy status of the properties from `private` to `protected`.

The `protected` modifier was invented as an intermediate level of privacy between `public` and `private`. Like `private`, it allows visibility to the same class in which the method or property was defined, but unlike `private` it also allows visibility to subclasses of the class in which the method or property was defined.

(d) Move the three methods `getWeapon`, `isDead` and `hit` out of the `XWing` and `TieFighter` classses and into the `SpaceShip` class. At this point, the `XWing` and `TieFighter` classes should contain nothing but a constructor.

(e) Finally, add the `extends` keyword to the first line of the `XWing` and `TieFighter` classes:

```
class XWing extends SpaceShip {
...
class TieFighter extends SpaceShip {
```

(f) Compile and run your program again, making sure that it produces the same results now that it is using inheritance.

(g) The `SpaceShip` class is a superclass of both `XWing` and `TieFighter` containing everything that X-Wings and Tie Fighters contain in common. Because the role of the `SpaceShip` class is simply to hold these commonalities, we might choose to label the class with the `abstract` keyword:

```
abstract class SpaceShip {
```

This prevents us from creating instances of the `SpaceShip` class. Without the `abstract` modifier, we could happily create a `new SpaceShip()`, which would be an object that is not an X-Wing, nor a Tie Fighter, but just a vague “space ship”. If we consider this to be a logical mistake then we can use `abstract` to prevent such calls to the `SpaceShip` constructor.

Change the class `SpaceShip` to be `abstract` and observe how the compiler will not accept any lines of the form:

```
SpaceShip s = new SpaceShip();
```

Remove the `abstract` keyword and notice how the compiler will then allow this line to compile.