by Nacho Iborra

Development Environments
Block 2
Unit 7: Abstract classes and interfaces

In this unit we are going to have an overview of two important concepts closely related with inheritance: abstract classes and interfaces, and some practical examples about how to apply them to real life problems.

2.7.1. Abstract classes

An **abstract class** is a class that can't be instantiated directly (this is, we can't create objects of this class), because its code or functionality is not completely defined. For instance, let's suppose that we have a class called `Animal`, in which we specify some attributes of every animal (such as color, or number of legs). However, there are some other attributes or operations, such as `talk` that can't be specified unless we know the concrete type of animal that we are talking about. In this case, we could (should) define `Animal` class as an abstract class.

If we want to define an abstract class in Java, we just add the reserved word `abstract` before `class` element. We can specify its attributes, constructors and other methods if we want to:

```
public abstract class Animal
{
    String color;
    int numberOfLegs;

    public Animal(String color, int numberOfLegs)
    {
        this.color = color;
        this.numberOfLegs = numberOfLegs;
    }

    public String getColor()
    {
        return color;
    }

    public void setColor(String color)
    {
        this.color = color;
    }

    ...
}
```

We can also add as many *abstract* methods as we want to. An abstract method is a method that is not implemented, so we set it as `abstract` and with no code. For instance, we could add an abstract method to previous class called `talk`:

```
public abstract class Animal
{
    String color;
    int numberOfLegs;

    ...

    public abstract void talk();
}
```

Keep in mind that **an abstract class does NOT need to have abstract methods in order to be abstract**, so we just add them if we need them. In this case, we can't make an animal talk unless we know the concrete type of animal, but we want every animal to talk as soon as they are created, so let's define this abstract method to be implemented later.

2.7.1.1. Inheriting from an abstract class

In order to create subtypes of an abstract class, we just inherit from it. These subtypes can be:

- **abstract**, so that we keep on adding the `abstract` modifier to the class, and we can even add more abstract methods if we want to. For instance, we could define a subclass called `Bird` with an abstract method called `fly`:

```
public abstract class Bird extends Animal
{
    public abstract void fly();
}
```

- **concrete**, so that we MUST implement (*override*) every abstract method defined in parent class(es). In our case, we could define an `Animal` subclass called `Dog` that needs to implement `talk` method, or a `Duck` subclass that inherits from `Bird` and then needs to implement both abstract methods (`talk` and `fly`).

```
public class Dog extends Animal
{
    public Dog(String color, int numberOfLegs)
    {
        super(color, numberOfLegs);
    }

    @Override
    public void talk()
    {
        System.out.println("Woof woof!!");
    }
}

public class Duck extends Bird
{
    public Duck(String color, int numberOfLegs)
    {
        super(color, numberOfLegs);
    }

    @Override
    public void talk()
    {
        System.out.println("Quack quack!!");
    }

    @Override
    public void fly()
    {
        System.out.println("I'm flying like a duck!");
    }
}
```

2.7.1.2. Abstract classes and polymorphism

We have just said that we can't instantiate objects of an abstract class. However, we can define an object of an abstract class from any of its concrete subclasses. This way, we can assign a variable of an abstract type any object of a subtype. For instance, if we look at previous example, we can't create an `Animal` object, since this class is abstract; but we can create a `Dog` object and assign it to an `Animal` variable, because of polymorphism.

```
Animal a1 = new Animal("red", 2);   // Error!!
Animal a2 = new Dog ("white", 4);   // OK
```

Proposed exercises:

2.7.1.1. Create a project in IntelliJ called **Animals**, with a main class called `AnimalsMain` within a package called `animals.main`. Then, add the classes seen before in a package called `animals.types`. Define the abstract classes `Animal` and `Bird` with their corresponding subclasses `Dog` and `Duck` and any other class that you may want to add (such as `Cat` or `Lion`, for instance). Then, define an array of 5 animals (type *Animal*) and fill it with some information (you don't need to ask it to the user if you don't want to). Then, explore the array and make each animal talk.

2.7.1.2. Go back to exercise 2.6.5.5 of previous unit (cultural organization) and make these changes to it (create a backup of the original project before making these changes):

- Define `CulturalObject` class as abstract
- In the main application, define an array of 6 `CulturalObject` objects and then add three books and three music discs to it. Then, print the whole array in the screen.

2.7.2. Interfaces

An interface can be considered as a special type of class with no code implemented (actually, we could add some code inside them, but this is not the aim of this section). So, they can't be directly instantiated either. We use interfaces to define a bunch of methods that need to be implemented by any class that wants to inherit from that interface.

For instance, let's suppose that we have an interface called `Shape` to represent any type of shape, such as circles, squares and so on. We don't want to store any specific information about a shape, but we want every shape to calculate its own area, and get drawn. So we can define an interface like this one:

```
public interface Shape
{
    public float calculateArea();
    public void draw();
}
```

So, any class that wants to "inherit" from that interface must implement these two methods. Actually, we are not inheriting from the interface, but **implementing** it, so we don't use *extends*, but `implements` reserved word.

```
public class Circle implements Shape
{
    float radius;

    public Circle(float radius)
    {
        this.radius = radius;
    }

    @Override
    public float calculateArea()
    {
        return Math.PI * radius * radius;
    }

    @Override
    public void draw()
    {
        System.out.println("Drawing a circle!");
    }
}
```

2.7.2.1. Extending vs implementing

We have just read about abstract classes and interfaces. None of them can be instantiated, and both can have some parts of unimplemented code. But... how to decide which one we must use in a given program?

- Abstract classes are **inherited**, so whenever we are wondering if we should create an abstract class, we need to be sure that:
    - The subclasses that we will define later **are subtypes** of the abstract class
    - We don't need to inherit from anything else (Java only lets us inherit from one class)
- Interfaces are **implemented**, and a class can implement as many interfaces as it needs, and also inherit from any other (single) class

```
public class MyShape extends AnotherClass
                     implements Shape, Comparable
{
    ...
}
```

The only drawback that interfaces have is the (almost) total lack of code: we can't define attributes, constructors or other methods. We can implement some static methods and other

special code, and leave unimplemented a set of methods, so that any implementing class can fill them and **act as** the interface. In our previous example, a *Circle* is not considered a *Shape* (since it does not inherit from it), but we can say that a *Circle acts as a Shape*, because it implements the corresponding interface. Also, we can define an array of an interface object and fill it with concrete objects implementing this interface:

```
Shape[] shapes = new Shape[10];
shapes[0] = new Circle(...);
shapes[1] = new Square(...);
...
shapes[0].draw();
```

Proposed exercises:

2.7.2.1. Create a project in IntelliJ called *Shapes*, with a main class called `ShapesMain` within a package called `shapes`. Then, add the `Shape` interface to that same package, and all the implementing classes (such as `Circle`, `Rectangle` or `Square`) in a subpackage called `shapes.types`. Implement both methods `calculateArea` and `draw` in all of them (just print a message in the `draw` method of each shape). Then, define an array of 5 shapes (type `Shape`) and fill it with some information (you don't need to ask it to the user if you don't want to). Finally, explore the array and calculate the area of each shape.

2.7.2.2. Example: sorting complex objects

You may be wondering how interfaces can be useful in your day to day work. Let's suppose that you are managing a list or array of complex objects, such as `Person` objects with their names and ages:

```
class Person {

    String name;
    int age;

    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }

    ...
}

...

public static void main(String[] args) {
    Person[] people = new Person[50];

    people[0] = new Person("Nacho", 40);
    people[1] = new Person("Juan", 70);
    ...
}
```

What if we want to sort this array by people's age, in descendant order? We may define a method to manually sort the array, by applying some of the well-known algorithms (such as *bubblesort* algorithm):

```
for (int i = 0; i < people.length - 1; i++) {
    for (int j = i + 1; j < people.length; j++) {
        if (people[i].getAge() < people[j].getAge()) {
            People aux = people[i];
            people[i] = people[j];
            people[j] = aux;
        }
    }
}
```

However, there is a faster way (in terms of efficiency) to get this result, although we may type some more code than in previous example. There are a couple of interfaces available in the Java core that lets us sort any kind of object. These interfaces are Comparator and Comparable. They both have one method to be implemented:

- If we choose `Comparable` interface, we must implement a method called `compareTo`, which receives a single object as a parameter, and compares it with current object (`this`). It also returns an integer indicating which one will go first in the

array: current object (negative number), the object received as a parameter (positive number), or zero if they are equal. As we are working with `this`, we will use this interface applied to the class whose objects need to be sorted.

- Regarding `Comparator` interface, we need to implement a method called `compare`. It receives two objects as parameters, and returns an integer indicating which will go first in the array: the first one (negative number), the second one (positive number) or zero if both are equal. We usually define an additional class to implement this interface, and sort objects of another different class.

Let's see how to use them with our `Person` class. First of all, we need to implement the chosen interface for our class. If we choose `Comparable` interface, this class needs to be `Person` class, and in the code of `compareTo` method we just return a number depending on which age is greater: remember, we need to sort people by age in descendant order, so we need to return a negative number if *this* object is older than the parameter:

```
class Person implements Comparable<Person> {

    String name;
    int age;

    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }

    @Override
    public int compareTo(Person p) {
        if (this.getAge() > p.getAge())
            return -1;
        else if (p.getAge() > this.getAge())
            return 1;
        else
            return 0;
    }
}
```

If you take a look at Integer API, there's an static method called `compare` that gets two integers as parameters and returns an integer determining which one is lower or greater (negative if the first one is lower, positive if the first one is greater). So, we can take advantage of this static method in order to sort persons by age. If we want to sort them in ascending order, we can do this:

```
@Override
public int compareTo(Person p) {
    return Integer.compare(this.getAge(), p.getAge());
}
```

But, if we want to sort the array in descending order, we just swap the order of the parameters:

```
@Override
public int compareTo(Person p) {
    return Integer.compare(p.getAge(), this.getAge());
}
```

There are similar methods in classes such as `Float`, `Character`, `Double`, etc, that let us compare primitive data types.

Then, in our *main* method, we just need to call `Arrays.sort` method from `Arrays` class (we need to import `java.util.Arrays` class in our code), and then our array will be automatically sorted:

```
import java.util.Arrays;
...
public static void main(String[] args) {
    Person[] people = new Person[50];
    ... // Fill array
    Arrays.sort(people);
    // Here our array is already sorted by age
}
```

If we choose `Comparator` interface instead of previous one, we usually define an external class that implements it (other than `Person` class)...

```
public class PersonComparator implements Comparator<Person> {
    @Override
    public int compare(Person p1, Person p2) {
        return Integer.compare(p2.getAge(), p1.getAge());
    }
}
```

Then, we just need to call `Arrays.sort` method with two arguments: the array to be sorted (our `people` array), and a comparator object that we will use to sort it (an instance

of `PersonComparator` class):

```
import java.util.Arrays;
...
public static void main(String[] args) {
    Person[] people = new Person[50];
    ... // Fill array
    Arrays.sort(people, new PersonComparator());
    // Here our array is already sorted by age
}
```

This way, we can leave the original class `Person` unchanged, and use another class to compare `Person` objects. This is particularly useful when we have no access to the code of the original class to modify it, or if we want to use an anonymous class, as we will see in next section.

> **Proposed exercises:**
>
> **2.7.2.2.** Go back to exercise 2.6.1.1 of previous unit, and create a copy of it called *SortedVideoGameList*. Then, use either `Comparable` or `Comparator` interfaces to sort the video game array by price (in ascending order) and print the sorted array in the screen.

2.7.3. Anonymous classes

As abstract classes and interfaces can't be instantiated, we always need to define a subclass that extends or implements the given abstract class or interface, and then instantiate this subclass.

From Java 7, there is a shortcut to define these subclasses without having to define a new source file and/or a new `class` element. We are talking about anonymous classes.

An **anonymous class** is a class without name that is created at the point where we need to implement a given interface, or extend from a given abstract class, so that we don't need to define an additional class for it.

Let's see how anonymous classes work with a couple of examples. The first one creates a new instance of an `Animal`, to define a new type of animal that is not defined in previous classes (`Dog` or `Duck`, for instance).

```
Animal strangeAnimal = new Animal("yellow", 2)
{
    @Override
    public void talk()
    {
        System.out.println("Vote for Quimby!");
    }
};

strangeAnimal.talk();
```

Notice that we just define an instance of `Animal` class and, inside the curly braces, we need to override and implement every pending abstract method (`talk` , in this case). We can also define any additional attribute or method that we need:

```
Animal strangeAnimal = new Animal("yellow", 2)
{
    String name = "Joe Quimby"

    @Override
    public void talk()
    {
        anotherMethod();
        System.out.println("Vote for Quimby!");
    }

    private void anotherMethod()
    {
        System.out.println ("My name is " + name);
    }
};

strangeAnimal.talk();
```

What we are defining, anyway, is an **object** which is a subtype of `Animal` in this case. So we can`t re-use this code to define another animal later (we would need to duplicate the code).

Regarding interfaces, we can also instantiate them and implement its methods in an anonymous class. This example shows how to define a new subtype of shape with its own implemented methods (and any additional one that we may add):

```
Shape irregularShape = new Shape()
{
    @Override
    public float calculateArea()
    {
        return 0.5f;
    }

    @Override
    public void draw()
    {
        System.out.println("Drawing this particular shape!");
    }
};

irregularShape.draw();
System.out.println(irregularShape.calculateArea());
```

We can use anonymous classes in many situations. For instance, we can define "on the fly" the sorting method for an array of `Person` objects like the one seen in previous examples:

```
import java.util.Arrays;
...
public static void main(String[] args) {
    Person[] people = new Person[50];
    ... // Fill array
    Arrays.sort(people, new Comparator<Person>()
    {
        @Override
        public int compare(Person p1, Person p2)
        {
            return Integer.compare(p2.getAge(), p1.getAge());
        }
    });
    // Here our array is already sorted by age
}
```

2.7.3.1. When can we use anonymous classes?

Anonymous classes are particularly useful when we need to define a particular instance of an abstract class or interface at a given point of our code (and nowhere else). This way, we avoid defining a new class element with its associated code. However, if we are planning to use this class definition in more than one place, then a "normal" class is recommended, so that we don't duplicate the code.

Proposed exercises:

2.7.3.1. Update exercise 2.7.2.1 by adding a new shape through an anonymous class. This shape will act as a diamond and it will have two internal attributes: the short axis and the long axis. Calculate its area by applying the corresponding formula, and implement the `draw` method by printing in the screen the message "I'm a diamond!". Then, add an object of this type to the shapes array to use its methods.

2.7.3.2. Define a `Comparator` in previous exercise to compare shapes by their area, in descending order. You must use an anonymous class to implement the comparator. Then, sort the array with this comparator and print it in the screen.