Programming in Java (24/25)

– Exercises Day 7 –

Learning goals

Before the next day, you should achieve the following learning goals:

- Understand the concept of inheritance.
- Extend classes.
- Override methods.
- Use `super` both for method calling and construction.
- Understand the use of `final` for classes and methods.
- Understand when to use composition vs inheritance.
- Understand the meaning of `private`, `public`, `protected`, and "default" visibility.

Note: Many exercises below instruct you to create methods. Unless the exercise description says otherwise, a very simple implementation (e.g., just printing something on screen) will be enough. The point today is on practising inheritance, not over-complicated algorithms for smartphones, musical instruments, etc.

1 Extension, extension...

Create a class `OldPhone` that implements the following interface. It is good practice to use the annotation "`@Override`" for a method whenever this is possible (i.e., you are implementing a method from an interface, like method `call(String)` from interface `Phone`, or you are overriding a method from a superclass).

```
/**
 * A Phone makes calls to given phone numbers.
 */
public interface Phone {

    /**
     * Just print on the screen: "Calling <number>...".
     *
     * @param number the phone number to call
     */
    void call(String number);
}
```

Now create a class `MobilePhone` that extends `OldPhone` and adds methods for things like `ringAlarm(String)` and `playGame(String)`. This class keeps a list of the last ten numbers that have been called, which can be printed with the method `printLastNumbers()`.

Then create a class `SmartPhone` that extends `MobilePhone` and adds methods for `browseWeb(String)` and `findPosition()`, the latter returning a (fictitious) GPS-found position as a `String`.

Create a small script (i.e., a separate Java class with a `main` method meant to exercise the classes that you have written) called `PhoneLauncher` in which you create a `SmartPhone` and use all its methods, including those inherited from its ancestor classes.

```
/**
 * Exercises some of the functionality in the Phone hierarchy.
 */
public class PhoneLauncher {

    /**
     * Launches the PhoneLauncher to exercise the Phone hierarchy.
     */
    public void launch() {
        // your code creating and using SmartPhone here...
    }

    /**
     * Creates and launches a PhoneLauncher.
     *
     * @param args ignored
     */
```

```
18      public static void main(String[] args) {
19          PhoneLauncher launcher = new PhoneLauncher();
20          launcher.launch();
21      }
22  }
```

2 Overriding

Save money by routing your international calls through the Internet! Modify your class `SmartPhone` so that it overrides the method `call(String)` inherited from its superclass. If the string parameter starts with "00", the method should output "Calling <number> through the internet to save money"; otherwise, the method should do the same as the original method (hint: use **`super`**).

3 Passing information to ancestor classes

Add the following field, constructor, and method to `OldPhone`:

```
1       /** The brand of this OldPhone. */
2       private String brand;
3
4       /**
5        * Creates a new OldPhone with a given brand.
6        *
7        * @param the brand of this OldPhone
8        */
9       public OldPhone(String brand) {
10          this.brand = brand;
11      }
12
13      /**
14       * @return the brand of this OldPhone
15       */
16      public String getBrand() {
17          return this.brand;
18      }
19      // ... there is no setter for brand
```

Add the appropriate constructors to `MobilePhone` and `SmartPhone` in order to be able to call the method `getBrand()` from an object of class `SmartPhone` and obtain the right answer, i.e., the brand provided in the constructor.

Do not introduce any additional fields to `MobilePhone` or `SmartPhone` (they are not needed).

4 Increasing visibility

Change the visibility of `playGame(String)` to **`private`** and check whether the script you wrote in Exercise 1 still works. Why does this happen? What are the minimal changes that you need to make on class `SmartPhone` so that the script still works?

5 Reducing visibility

Some parents are concerned that their children spend too much time playing with their smartphones. Create a class `RestrictedSmartPhone` that overrides `playGame(String)` to make it **`private`** and thus non-visible to external classes and scripts. Is this possible? Why?

6 Multiple inheritance

Create a class `MusicalInstrument` with a method `play()`. Now create another class `WoodenObject` with a method `burn()`.

Create a class `Guitar` that is at the same time a musical instrument and a wooden object. How would you do it in Java?

7 Java magic

Can you see what is wrong in the following code so that the compiler will refuse to accept it?

```
1  /**
2   * A Singer has a name and can sing songs.
3   */
4  public class Singer {
5
6      /** The name of this Singer. */
7      private String name;
8
9      /**
10      * Creates a new Singer with a given name.
11      *
12      * @param name the name of this Singer
13      */
14     public Singer(String name) {
15         this.name = name;
16     }
17
18     /**
19      * Returns the name of this Singer.
20      *
21      * @return the name of this Singer
```

```
     */
    public String getName() {
        return this.name;
    }

    /**
     * Sings a specific song.
     *
     * @param subject the song to sing
     */
    public void sing(String song) {
        System.out.println("Singing song: " + song);
    }
}
```

```
/**
 * A SingerSongwriter can both sing and write songs.
 */
public class SingerSongwriter extends Singer {

    /**
     * Writes a song with a given title.
     *
     * @param title the song title
     */
    public void writeSong(String title) {
        System.out.println("Writing song with title: " + title);
    }
}
```

If it is not evident, try to compile the code.

If it compiles without problems, write a script that creates an object of class `SingerSongwriter` and uses its methods for singing, writing songs, and querying the name. If it does not, modify class `SingerSongwriter` so that the program compiles, and then write the script to use these methods.

8 Packages and visibility

Create a package `pij.day7.artists` and put the classes `Singer` and `SingerSongwriter` that you created in Exercise 7 into this package. Now create a second package `pij.day7.driver` and put your script from Exercise 7 into this package. Compile your code. Does your script still run as before?

Change the visibility modifier of method `getName()` in class `Teacher` from **`public`** to **`protected`**, to **`private`**, and to no modifier. For each of the three variants, try to recompile all your classes for this exercise. What happens? Why?

9 `final` means no change

Create a class that extends `String` and adds a method `printEven()` that prints on screen the even-numbered characters of the string. Try to compile it and see what happens.

10 Refactoring: from inheritance to composition

At the end of Section 2 of the notes for Day 7, we said that the classes `DrinkRefrigerator` and `ChocBarVendingMachine` should rather use delegation than inheritance for access to the `buy(int money)` method. In a similar fashion to Section 1.3, sketch what classes you would have in this case and how they would be implemented. In the methods `releaseCan()` and `giveChocolateBar()`, just print a suitable message on the screen.

11 Noah's Ark (*)

Design and implement an application that represents the day that Noah's Ark was open, just before the rain started. In your script, create an animal of each species and then call them all in. Every animal must implement a method `call()` that prints on screen the appropriate message. You should keep in mind the following requirements:

- The application should contain at least: bears, beetles, cats, crocodiles, dogs, dolphins, eagles, flies, frogs, lizards, monkeys, owls (of course), pigeons, salmon, sharks, snakes, whales.
- All animals have at least a method `call()` and a method `reproduce()` (for after the rain).
- Insects, fish, amphibians, reptiles, and birds lay eggs (`layEggs()`). Mammals cannot lay eggs but give birth[1] (`giveBirth()`). The method `reproduce()` should call the appropriate method in each case.
- When called, all animals answer (i.e., print on screen) "<name of the animal> coming...". The exceptions are aquatic animals, which are not affected by the rain and answer "<name of the animal> will not come..."; and flying animals, which answer "<name of the animal> now flying, will come later when tired...". Method `call()` **must not** be implemented in every class: use inheritance to reuse methods and constructors to pass information to parent classes.
- All animals make a sound. If `Animal` is an `interface` in your design, `makeSound()` must be a method in there; if `Animal` is an `abstract class`, it must be an `abstract` method. The method can then be implemented by descendant classes.

[1] There were no platypus in Noah's Ark.