Programming in Java (24/25)

– *Exercises Day 8* –

Learning goals

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

- Overload methods
- Prevent code repetition in overloaded methods
- Upcast objects to more general types
- Downcast objects to more specific types
- Understand how Java determines which method implementation is used for a method call
- Create generic classes
- Use generic classes

1 Don't Repeat Yourself

Look at the following code. Is there anything you can do to make this code better from a DRY perspective?

```
1  public class MaximumCalculator {
2      public int getMax(int n, int m) {
3          if (n > m) {
4              return n;
5          } else {
6              return m;
7          }
8      }
9      public Object[] getMax(Object[] o1, Object[] o2) {
10         if (o1.length > o2.length) {
11             return o1;
12         } else {
13             return o2;
```

```
        }
    }
    public String getMax(String number1, String number2) {
        int n1 = Integer.parseInt(number1);
        int n2 = Integer.parseInt(number2);
        if (n1 > n2) {
            return number1;
        } else {
            return number2;
        }
    }
}
```

2 Upcasting, downcasting

For this exercise, you will need to use again some classes and interfaces you created last day: Phone, OldPhone, MobilePhone, SmartPhone.

2.1 Start

Create a script that builds a new SmartPhone with the line:

```
        SmartPhone myPhone = new SmartPhone("ACME");
```

and then uses all its methods.

2.2 Direct upcasting

Change the script so that the SmartPhone is created with the line:

```
        MobilePhone myPhone = new SmartPhone("ACME");
```

Compile your code again. Are there any problems? Why do this problems happen? What are the possible solutions?

2.3 Indirect upcasting when calling a method

Pass this object to a method testPhone(Phone) that has only one parameter of type Phone. What methods can you call on the object inside the method?

2.4 Downcasting

Inside the former method, downcast the object to SmartPhone so that you can use all the public methods of SmartPhone.

2.5 Casting exception

Create a MobilePhone object and then pass it to the method testPhone(Phone) from Exercise 2.4. What happens?

3 Understanding ad-hoc polymorphism

Try solving this exercise first without running the code, just by looking at the code.

In the setting of our Phone exercises, consider the following classes:

```
1  public class SimpleTester {
2      public void test(OldPhone p) {
3          System.out.println("simple test old");
4      }
5      public void ring(OldPhone p) {
6          System.out.println("simple ring old");
7      }
8  }
```

```
1  public class SpecialTester extends SimpleTester {
2      public void test(OldPhone p) {
3          System.out.println("special test old");
4      }
5      public void test(MobilePhone m) {
6          System.out.println("special test mobile");
7      }
8      public void ring(MobilePhone m) {
9          System.out.println("special ring mobile");
10     }
11 }
```

```
1  public class TesterMain {
2      public static void main(String[] args) {
3          OldPhone old = new OldPhone("ACME classic");
4          MobilePhone mobile = new MobilePhone("ACME");
5          OldPhone mobileAsOld = mobile;
6
7          SimpleTester simple = new SimpleTester();
8          SpecialTester special = new SpecialTester();
9          SimpleTester specialAsSimple = special;
10
11         simple.test(old);
12         simple.test(mobile);
13         simple.test(mobileAsOld);
14         System.out.println();
```

```
15
16          special.test(old);
17          special.test(mobile);
18          special.test(mobileAsOld);
19          System.out.println();
20
21          specialAsSimple.test(old);
22          specialAsSimple.test(mobile);
23          specialAsSimple.test(mobileAsOld);
24          System.out.println();
25
26          simple.ring(old);
27          simple.ring(mobile);
28          simple.ring(mobileAsOld);
29          System.out.println();
30
31          special.ring(old);
32          special.ring(mobile);
33          special.ring(mobileAsOld);
34          System.out.println();
35
36          specialAsSimple.ring(old);
37          specialAsSimple.ring(mobile);
38          specialAsSimple.ring(mobileAsOld);
39      }
40  }
```

What does this program print? Why?

Hint: Remember that the Java compiler *irrevocably* selects the *signature* that will be used for the method call. The compiler uses the information from the type declarations of the references to do this, but it ignores what objects might be in memory when the program is run.

When we run the program, the JVM looks at the object in memory and selects the right implementation of the method for that object *only among the methods with the signature selected earlier by the compiler*. This "late binding" of a method call to the used implementation is also called *dynamic dispatch*.

4 Why is `static` called static?

Try solving this exercise first without running the code, just by looking at the code.

Consider the following classes:

```java
public class Bird {
    public static void print() {
        System.out.println("bird");
    }
}
```

```java
public class Owl extends Bird {
    public static void print() {
        System.out.println("owl");
    }
}
```

```java
public class OwlMain {
    public static void main(String[] args) {
        Bird b = new Bird();
        Owl o = new Owl();
        Bird owlAsBird = o;

        Bird.print();
        Owl.print();

        // you can call static methods also via object references
        // (calling them via the class name is clearer)
        b.print();
        o.print();
        owlAsBird.print();
    }
}
```

What is printed? Why?

Hint: One way of reading the keyword "**static**" is as "determined when the code is compiled". In Computer Science, the term "static" is often used to mean the opposite of "dynamic", which can be read as "when the code is run".

5 `Pair<...>`

Write a *generic* class `Pair` that can store **two** objects of arbitrary (and possibly different) types. Use generic type parameters where appropriate. How many type parameters do you need?

Your class should provide at least the following public functionality:

- A constructor with two arguments for the first and the second components of the Pair.
- A constructor with one argument for a Pair which copies its components into the new object.
- Getters for both components of the Pair.
- A method swap() that returns a new Pair object with the positions of the stored objects exchanged. So when we call swap() on a Pair object corresponding to the pair (42, "Thingy"), we will get a Pair object corresponding to the pair ("Thingy", 42). What does this mean for the result type?

When you choose the visibility modifiers for your instance variables, recall the principles of information hiding.

To avoid code repetition, make one constructor of your class call the other constructor.

Write a small script that uses all the constructors and methods of class `Pair`.

6 Box<T> revisited

Recall class Box<T> from the notes.

```
1  /**
2   * A Box<T> stores a reference to an object and can return the stored reference.
3   *
4   * @param <T> the type of the reference to store and later retrieve
5   */
6  public class Box<T> {
7      /** The stored reference. */
8      private T data;
9
10     /**
11      * Constructs a Box<T> that stores the given reference.
12      *
13      * @param data the reference to store
14      */
15     public Box(T data) {
16         this.data = data;
17     }
18
19     /**
20      * Returns the stored reference.
21      *
22      * @return the stored reference
23      */
24     public T getData() {
25         return this.data;
26     }
27 }
```

Add a `public` setter method `setData` with one parameter to the class. What parameter type does your method have?

7 Generics and subclasses

Consider the following three (very minimalistic) classes:

```
1 public class Animal {}
```

```
1 public class Cat extends Animal {}
```

```
1 public class Dog extends Animal {}
```

Now consider a class

```
1 public class AnimalHelper {
2     public static Animal doSomething(Box<Animal> animalBox) {
3         return animalBox.getData();
4     }
5 }
```

where class `Box<T>` is from the previous exercise.

Add the lines `animalBox.setData(new Animal());` and `animalBox.setData(new Dog());` at the beginning of the body of method `doSomething`. Does the compiler still accept the method? Why (not)?

Now try to compile and run the following lines in a script:

```
1         Box<Cat> catBox = new Box<>(new Cat()); // Box<Cat>, not Box<Animal>
2         Animal animal = AnimalHelper.doSomething(catBox);
```

Does the compiler accept your script? Why (not)?