Learning object-oriented programming

Learning Objectives

- To revise the concepts of class and object.
- To realize that a program that has been written without objects can also be written using objects.
- To realize that the use of objects can make a program more understandable.

So, what's object-oriented programming all about?

Let's step back for a moment and inspect how a clock works. The clock has three hands: hours, minutes and seconds. The second-hand grows once every second, the minute hand every sixty seconds, and the hour hand every sixty minutes. When the value of the second hand is 60, its value is set to zero, and the value of the minute hand grows by one. When the minute hand's value is 60, its value is set to zero, and the hour hand's value grows by one. When the hour hand's value is 24, it's set to zero.

The time is always printed in the form `hours: minutes: seconds`, where two digits are used to represent the hour (e.g., 01 or 12) as well as the minutes and seconds.

The clock has been implemented below using integer variables (the printing could be in a separate method, but that has not been done here).

```
int hours = 0;
int minutes = 0;
int seconds = 0;

while (true) {
    // 1. Printing the time
```

```
        if (hours < 10) {
            System.out.print("0");
        }
        System.out.print(hours);

        System.out.print(":");

        if (minutes < 10) {
            System.out.print("0");
        }
        System.out.print(minutes);

        System.out.print(":");

        if (seconds < 10) {
            System.out.print("0");
        }
        System.out.print(seconds);
        System.out.println();

        // 2. The second hand's progress
        seconds = seconds + 1;

        // 3. The other hand's progress when necessary
        if (seconds > 59) {
            minutes = minutes + 1;
            seconds = 0;

            if (minutes > 59) {
                hours = hours + 1;
                minutes = 0;

                if (hours > 23) {
                    hours = 0;
                }
            }
        }
    }
```

As we can see by reading the example above, how a clock consisting of three `int` variables works is not clear to someone reading the code. By looking at the code, it's difficult to "see" what's going on. A famous programmer once remarked *"Any fool can write code that a computer can understand .Good Programmers write code that humans can understand"*.

Our aim is to make the program more understandable.

Since a clock hand is a clear concept in its own right, a good way to improve the program's clarity would be to turn it into a class. Let's create a `ClockHand` class that describes a clock hand, which contains information about its value, upper limit (i.e., the point at which the value of the hand returns to zero), and provides methods for advancing the hand, viewing its value, and also printing the value in string form.

```java
public class ClockHand {
    private int value;
    private int limit;

    public ClockHand(int limit) {
        this.limit = limit;
        this.value = 0;
    }

    public void advance() {
        this.value = this.value + 1;

        if (this.value >= this.limit) {
            this.value = 0;
        }
    }

    public int value() {
        return this.value;
    }

    public String toString() {
        if (this.value < 10) {
            return "0" + this.value;
        }

        return "" + this.value;
    }
}
```

Once we've created the ClockHand class, our clock has become clearer. It's now straightforward to print the clock, i.e., the clock hand, and the hand's progression is hidden away in the ClockHand class. Since the the hand returns to the beginning automatically with the help of the upper-limit variable defined by the ClockHand class, the way the hands work together is slightly different than in the implementation that used integers. That one looked at whether the value of the integer that represented the clock hand exceeded the upper limit, after which its value was set to zero and the value of the integer representing the next clock hand was incremented. Using clock-hand objects, the minute hand advances when the second hand's value is zero, and the hour hand advances when the minute hand's value is zero.

```java
ClockHand hours = new ClockHand(24);
ClockHand minutes = new ClockHand(60);
ClockHand seconds = new ClockHand(60);

while (true) {
    // 1. Printing the time
    System.out.println(hours + ":" + minutes + ":" + seconds);

    // 2. Advancing the second hand
    seconds.advance();

    // 3. Advancing the other hands when required
    if (seconds.value() == 0) {
        minutes.advance();

        if (minutes.value() == 0) {
            hours.advance();
        }
    }
}
```

Object-oriented programming is primarily about isolating concepts into their own entities or, in other words, creating abstractions. Despite the previous example, one might deem it pointless to create an object containing only a number since the same could be done directly with `int` variables. However, that is not always the case.

Separating a concept into its own class is a good idea for many reasons. Firstly, certain details (such as the rotation of a hand) can be hidden inside the class (i.e., **abstracted**). Instead of typing an if-statement and an assignment operation, it's enough for the one using the clock hand to call a clearly-named method `advance()`. The resulting clock hand may be used as a building block for other programs as well - the class could be named `CounterLimitedFromTop`, for instance. That is, a class created from a distinct concept can serve multiple purposes. Another massive advantage is that since the details of the implementation of the clock hand are not visible to its user, they can be changed if desired.

We realized that the clock contains three hands, i.e., it consists of three concepts. The clock is a concept in and of itself. As such, we can create a class for it too. Next, we create a class called "Clock" that hides the hands inside of it.

```java
public class Clock {
    private ClockHand hours;
    private ClockHand minutes;
    private ClockHand seconds;

    public Clock() {
        this.hours = new ClockHand(24);
        this.minutes = new ClockHand(60);
        this.seconds = new ClockHand(60);
    }

    public void advance() {
        this.seconds.advance();

        if (this.seconds.value() == 0) {
            this.minutes.advance();

            if (this.minutes.value() == 0) {
                this.hours.advance();
            }
        }
    }

    public String toString() {
        return hours + ":" + minutes + ":" + seconds;
    }
}
```

The way the program functions has become increasingly clearer. When you compare our program below to the original one that was made up of integers, you'll find that the program's readability is superior.

```java
Clock clock = new Clock();

while (true) {
    System.out.println(clock);
    clock.advance();
}
```

The clock we implemented above is an object whose functionality is based on "simpler" objects, i.e., its hands. This is precisely the **great idea behind object-oriented programming: a program is built from small and distinct objects that work together**

Programming exercise:

One Minute

Points

1/1

The exercise template comes with the "ClockHand" class described above. Implement a `Timer` class based on the material's `Clock` class.

The timer has two hands, one for hundredths of a second and one for seconds. As it progresses, the number of hundredths of a second grows by one. When the hand corresponding to hundredths of a second reaches a value of 100, its value is set to zero, and the number of seconds grows by one. In the same way, when the value of the hand corresponding to seconds reaches the value of sixty, its value is set to zero.

- `public Timer()` creates a new timer.
- `public String toString()` returns a string representation of the timer. The string representation should be in the form "seconds: hundredths of a second", where both the seconds and the hundredths of a second are represented by two numbers. For example, "19:83" would represent the time 19 seconds, 83 hundredths of a second.
- `public void advance()` moves the timer forward by a hundredth of a second.

Once you've completed the task, return it to the server.

You can test out the timer's functionality in the main program whenever you like. The example code below provides you with a program where the timer is printed and it advances once every hundredth of a second.

```
Timer timer = new Timer();

while (true) {
    System.out.println(timer);
    timer.advance();

    try {
        Thread.sleep(10);
    } catch (Exception e) {

    }
}
```

NB! The program above will never stop running by itself. Press the red square to the left of the program's print window to turn it off.

Exercise submission instructions

How to see the solution

Next, let's review topic terminology.

Object

An **Object** refers to an independent entity that contains both data (instance variables) and behavior (methods). Objects may come in lots of different forms: some may describe problem-domain concepts, others are used to coordinate the interaction that happens between objects. Objects interact with one another through method calls — these method calls are used to both request information from objects and give instructions to them.

Generally, each object has clearly defined boundaries and behaviors and is only aware of the objects that it needs to perform its task. In other words, the object hides its internal operations, providing access to its functionality through clearly defined methods. Moreover, the object is independent of any other object that it doesn't require to accomplish its task.

In the previous section, we dealt with objects that represented people whose structure was defined in a "Person" class. It's a good idea to remind ourselves of what a class does: a **class** contains the blueprint needed to create objects, and also defines the objects' variables and methods. An object is created on the basis of the class constructor.

Our person objects had a name, age, weight and height property, along with a few methods. If we were to think about the structure of our person object some more, we could surely come up with more variables related to a person, such as a personal ID number, telephone number, address and eye color.

In reality, we can relate all kinds of different information and things to a person. However, when building an application that deals with people, the **functionality and features related to a person are gathered based on the application's use case**. For example, an application focused on personal health and well-being would probably keep track of the variables mentioned earlier, such as age, weight, and height, and also provide the ability to calculate a body mass index and a maximum heart rate. On the other hand, an application focused on communication could store people's email addresses and phone numbers, but would have no need for information such as weight or height.

The state of an object is the value of its internal variables at any given point in time.

In the Java programming language, a Person object that keeps track of name, age, weight, and height, and provides the ability to calculate body mass index and maximum heart rate would look like the following. Below, the height and weight are expressed as doubles - the unit of length is one meter.

```java
public class Person {
    private String name;
    private int age;
    private double weight;
    private double height;

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

    public double bodyMassIndex() {
        return this.weight / (this.height * this.height);
    }
    public double maximumHeartRate() {
        return 206.3 - (0.711 * this.age);
    }

    public String toString() {
        return this.name + ", BMI: " + this.bodyMassIndex()
            + ", maximum heart rate: " + this.maximumHeartRate();
    }
}
```

Determining the maximum heart rate and body mass index of a given person is straightforward using the Person class described above.

```
Scanner reader = new Scanner(System.in);
System.out.println("What's your name?");
String name = reader.nextLine();
System.out.println("What's your age?");
int age = Integer.valueOf(reader.nextLine());
System.out.println("What's your weight?");
double weight = Double.valueOf(reader.nextLine());
System.out.println("What's your height?");
double height = Double.valueOf(reader.nextLine());

Person person = new Person(name, age, weight, height);
System.out.println(person);
```

Sample output

```
What's your name?
Napoleone Buonaparte
What's your age
51
What's your weight?
80
What's your height?
1.70
Napoleone Buonaparte, BMI: 27.68166089965398, maximum heart
rate: 170.03900000000002
```

Class

A class defines the types of objects that can be created from it. It contains instance variables describing the object's data, a constructor or constructors used to create it, and methods that define its behavior. A rectangle class is detailed below which defines the functionality of a rectangle.

```java
// class
public class Rectangle {

    // instance variables
    private int width;
    private int height;

    // constructor
    public Rectangle(int width, int height) {
        this.width = width;
        this.height = height;
    }

    // methods
    public void widen() {
        this.width = this.width + 1;
    }

    public void narrow() {
        if (this.width > 0) {
            this.width = this.width - 1;
        }
    }

    public int surfaceArea() {
        return this.width * this.height;
    }

    public String toString() {
        return "(" + this.width + ", " + this.height + ")";
    }
}
```

Some of the methods defined above do not return a value (methods that have the keyword `void` in their definition), while others do (methods that specify the type of variable to be returned). The class above also defines the `toString` method, which returns the string used to print the object.

Objects are created from the class through constructors by using the `new` command. Below, we'll create two rectangles and print information related to them.

```java
Rectangle first = new Rectangle(40, 80);
Rectangle rectangle = new Rectangle(10, 10);
System.out.println(first);
System.out.println(rectangle);

first.narrow();
System.out.println(first);
System.out.println(first.surfaceArea());
```

Sample output

```
(40, 80)
(10, 10)
(39, 80)
3920
```

Programming exercise:

Book

Points 1/1

Create a "Book" class that describes a book. Each book has an author, title, and page count.

Make the class a:

- Constructor `public Book(String author, String name, int pages)`
- Method `public String getAuthor()` that returns the book's author's name.
- Method `public String getName()` that returns the name of the book.
- Method `public int getPages()` that returns the number of pages in the book.

In addition, make a `public String toString()` method for the book that will be used to print the book object. For example, the method call should produce the following output:

Sample output

```
J. K. Rowling, Harry Potter and the Sorcerer's Stone, 223 pages
```

Exercise submission instructions

How to see the solution

Programming exercise:

Cube

Points

1/1

Create a `Cube` class that represents a cube (i.e., a standard hexahedron). Create a `public Cube (int edgeLength)` constructor for the class, that takes the length of the cube's edge as its parameter.

Make a `public int volume()` method for the cube, which calculates and returns the cube's volume. The volume of the cube is calculated with the formula `edgeLength * edgeLength * edgeLength`. Moreover, make a `public String toString()` method for the cube, which returns a string representation of it. The string representation should take the form "`The length of the edge is l and the volume v`", where `l` is the length and `v` the volume - both the length and volume must be represented as integers.

Examples are provided underneath

```java
Cube oSheaJackson = new Cube(4);
System.out.println(oSheaJackson.volume());
System.out.println(oSheaJackson);

System.out.println();

Cube salt = new Cube(2);
System.out.println(salt.volume());
System.out.println(salt);
```

Sample output

```
64
The length of the edge is 4 and the volume 64
```

```
8
The length of the edge is 2 and the volume 8
```

Exercise submission instructions

How to see the solution

Programming exercise:

Fitbyte

Points

1/1

The Karvonen method allows you to calculate your target heart rate for physical exercise. The target heart rate is calculated with the formula `(maximum heart rate - resting heart rate) * (target heart rate percentage) + resting heart rate`, where the target heart rate is given as a percentage of the maximum heart rate.

For example, if a person has a maximum heart rate of `200`, a resting heart rate of `50`, and a target heart rate of `75%` of the maximum heart rate, the target heart rate should be about `((200-50) * (0.75) + 50)`, i.e., `162.5` beats per minute.

Create a class called `Fitbyte`. Its constructor takes both an age and a resting heart rate as its parameters. The exercise assistant should provide a method targetHeartRate, which is passed a number of type double as a parameter that represents a percentual portion of the maximum heart rate. The proportion is given as a number between zero and one. The class should have:

- A constructor `public Fitbyte(int age, int restingHeartRate)`
- A method `public double targetHeartRate(double percentageOfMaximum)` that calculates and returns the target heart rate.

Use the `206.3 - (0.711 * age)` formula to calculate the maximum heart rate.

Use case:

```
Fitbyte assistant = new Fitbyte(30, 60);

double percentage = 0.5;

while (percentage < 1.0) {
    double target = assistant.targetHeartRate(percentage);
    System.out.println("Target " + (percentage * 100) + "% of maximum: " +
    percentage = percentage + 0.1;
}
```

Sample output

```
Target 50.0% of maximum: 122.48500000000001
Target 60.0% of maximum: 134.98200000000003
Target 70.0% of maximum: 147.479
Target 80.0% of maximum: 159,976
Target 89.99999999999999% of maximum: 172.473
Target 99.99999999999999% of maximum: 184.97000000000003
```

Exercise submission instructions

How to see the solution

You have reached the end of this section! Continue to the next section:

→ 2. Removing repetitive code (overloading methods and constructors)

Remember to check your points from the ball on the bottom-right corner of the material!

In this part: