First think, then program

The Java programming language completely changed (resp. hould completely change) the way in which you are working. Java introduces such a strong connection to the technology of object-oriented programming, object-oriented technique, that simply can not be ignored.

Let's look at a simple example. Consider a program model of a crossroad, where traffic lights will guide transport. An example of such a crossroad is shown on figure ..\files\03-21 Křižovatka 1.png

Classic programmatic approach would be that we first created four green areas (lawns) around which we of rings, rectangles and ovals began to create traffic lights. Those creative would have created only one traffic light and placed him in four different positions.

And then we would start writing the program logic of crossroad: when traffic on the horizontal street is on, both horizontal traffic lights must be green. Therefore we turn all horizontal green lights and other lights turn off.

That is completely wrong.

This procedure treats the solution as an image, but does not respect functionality. This means that the individual elements of the image do not describe things of the real world, but just their pictures. At the highest level, overlooks the fact that there are certain restrictions for traffic lights, such as not to let both directions simultaneously. The lower level also disregards the reality, for example at traffic lights never shine red and green at one moment.

Programming style like this is typical for BASIC programs.

Yes, the program must take care so that model worked correctly. But it is quite tedious, confusing and there is a great danger that we make a mistake. Moreover, such a program would be very difficult to modify. Imagine that we should expand the intersection as shown, it means to add more lights so that the bottom street had a separate traffic light to turn left. That would completely revise the entire solution algorithm, therefore it would be laborious and risky because of the possible errors.

The correct procedure is to split the application logic on different **levels of abstraction** and at every abstraction level to combine appearance (picture) with functionality.

For example, at the level of a semaphore we see that it has three lights and that it can have four states: red, yellow, green and red with yellow. Alternatively, a fifth state (no light is on). Semaphore for us is an object that has a **graphical form** (gray oval with colored circles) and at the same time with the ability to perform an **activity**. In our case, that activity was the "go to state 1", "go to state 2" and so on till "go to the state 5."

At the highest level, we noticed that the crossroad has four traffic lights and lights turn according to some rules. Therefore, we created a class TrafficLights that manages individual semaphores. This means that it will call functions "go to state 1", "go to state 2" and so on. Importantly, this class does not need to know anything about the internal structure of a traffic light (and does not know). If, say, we exchange for another semaphore, which wouldreplace the round lights had by text labels, class TrafficLights about it did not need to know anything. Model would work on.

Java directly supports good programming style, because it forces programmers to code classes. Of course, even in Java can be written a poorly structured code, but it is rather unusual. I tis still true that one can write BASIC-like programs in any language ☺.

Important

In practice, we would Semaphore class implement as a separate class that would include graphics. It would work similarly to the other components, ie. we would drag it from the library to the workspace as easy as, say, the Button component. But our knowledge is not enough to program something like this.

Therefore is used replacement solution where all classes are placed directly in the file *FXMLDocumentController.java*.

Class Semaphore

Now we will show you how you can create a simple class "Semaphore". We will accomodate 5 functions (abilities) to switch semaphore into five allowable states.

Graphics

We start by creating a new project and call it TrafficLights. Of course, our well knovn HelloWorld opens. Open SceneBuilder and both Buttonand Label leave in project, just shuffle away the way. (If you need to remove them, select them and delete by Del key.)

On left side click Shapes. Then find Rectangle and drag it into center of workspace. Auxiliary red centerlines will help us to find exact center. This rectangle will be a graphic shape of semaphore. This why we will narrow it a bit.

On right side of screen we see „Properties“. Label Fill shows the colour Semaphore is filled. Change it to loght grey.

By the way, we can modify all properties. For example, Rotate means to rotate and adjust the rotation of graphic element.

Similarly as before, drag a Circle from left pane onto the workplace. Position does not matter, the circle does not fit there anyway.

This is why we select tile Layout and set Radius=25. Place the circle correctly and change its colour to black (safe state). This is a base for class Semaphore.

SceneBuilder ↔ FXML ↔ NetBeans

NetBeans and SceneBuilder interchange data using FXML file. So do not forget to store this layout into file!

Now switch to NetBeans.

Changes will not appear automatically in NetBeans, we must read them from FXML file first!

We'll right-click on the file and select Edit FXMLDokument.xml. Opens text file in which (red arrow), we see that the class

Semaphore consists of one object Rectangle, three objects Circle and one objectButton. Each object has its own parameters. When you now try to run the program, it will show the image of traffic lights.

But that's not all. Because NetBeans and SceneBuilder are separate programs, so one does not know anything about the variables that are defined in the second. And vice versa. We must create a bidirectional connection between NetBeans and SceneBuilder to pass variables. This connection is via file FXML.

Lets take a look at connection NetBeans ↔ FXML first. In file *FXMLDocumentController.java* there are two lines as follows:

```
@FXML
```

```
private Label label;
```

Second line says, that Java will work with a variable, whose name is „label“ and its type is Label. And this is a private variable.

First line is tag `@FXML`. This tag marks a variable that is to be connected to FXML file. **Each variable, which is expected to link to file FXML must have `@FXML` tag**.

Something similar must do well. Define three variables that wittily called a redLight, greenLight and yellowLight. All will be of type Circle and in front of each will be tag `@FXML`.

Trying to do so, an error appears, becouse a library for Circle is not imported. We can do so by accepting NetBeans‘ proposal. Our program will look as this. This was the first step.

But when we look at the FXML file, we find there is no such thing! To change the file FXML, the program must be compiled. It is suitable icon for this purpose.

Now **again (!) open FXML file** in ScreenBuilder, select one light on the tab **Code:** in the combo-box **fx: id** assign a name that belongs to the light. It successively repeated for all the lights. Finally, do not forget to save the result.

After you open FXML file in NetBeans (use *Edit*), you can see that important information like `fx:id="redLight"` appeared.

For testing purposes, we eventually can enhance handler of Button like this:

```
redLight.setFill(Color.BLUE);
```

Please, do not forget to import library

```
import javafx.scene.paint.Color;
```

Program Logic

Now we start adding functionality into Semaphore class.

Program the traffic light will be solved during the exercises. An example of how it can look like a finished program is here.

Example: Calculator

Scene Builder is used for visual editing windows with graphics. It cooperates with NetBeans, but it is not part of NetBeans, it is a standalone application.

The link between NetBeans and SceneBuilder is mediated by FXML file. When you rightclick FXMLDocument.fxml and select **Open**, a SceneBuilder will open and grapics of this file is shown. In general: stuff and navigation to the left, picture in the middle, and properties on the right. We'll make some small modifications as follows.

Every JavaFX project starts with a Hello World program. This is a nice starting point for our experiments.

We'll make some small modifications as follows. First, click the entry for **Button** in the Hierarchy panel (bottom left), which selects the button in the middle Content panel (see figure). We alternatively can select Button by clicking it. Move it down and over a bit to make room for the second button.

Next, let's drag another button from the Library panel (top left) and drop it onto the canvas, lining it up with the first button using the red positioning lines that appear while dragging it (see figure).