OOP, much simplified introduction

Object-oriented paradigm

It is a philosophy and way of thinking, design and implementation, where we put the emphasis on reusability.

Access finds inspiration in the Industrial Revolution - the invention of the basic components that we will use (eg. while we build a house, you will not burn bricks and turning of the screws, simply buy already done).

Advantages of OOP

Folded component of the program is better and cheaper.

We believe it is tested (the components are known to work are tested and maintained).

If there is a problem, you fix it in one place.

We are motivated to write code clearly and properly, because we use it for others or ourselves in other projects.

Note: man is by nature lazy and if he did not know that his code will be used again, would not try to write it well ☺

Principles of OOP

We try to simulate reality as we are accustomed to perceive it. We can say that we leave seeing the program from the

perspective of computer (machine) and we will see the program rather from the perspective of a programmer.

As we replaced assembler by humanly readable mathematical notation, now let's go even further and replace the rest.

It is therefore a certain level of abstraction of the program.

It has considerable advantages in the fact that it is natural for us and clearer.

Object, class, instance

Let us have some concept of the real world, such as "man" or "database". This concept can be understood in two senses:

1. First as an abstraction that summarizes all the common elements of the concept. An example might be a "man" who has two arms, two legs, head and body, sometimes also sense and the like. This meaning is called a **class**; we speak about the class of man, the class of database, etc.. Generally speaking, the class determines which features and capabilities will be.

2. The second as a particular person; in this sense we are speaking about an **instance** or just the **object**. Each object has its own specific properties and its specific methods.

We can imagine that the class is "template", which produces different instances (=objects).

Both classes and objects have **attributes** and **methods**.

Attributes

Attributes are the properties of an object or data that stores (eg. humans name and age, the database password). It is a simple variable, with which we have worked a hundred times. Sometimes we talk about them as about the internal state of the object.

Objects (= instances) have the same attributes as has their class. Attributes of objects can have individual values for each object.

Methods

Methods are abilities that an object can perform. In humans, it could be methods: **go()**, **read()** or **say()**.

From a programmer‘s perspective, it is actually a function.

In older languages, methods did not belong to objects, but are freely found in the modules.

The disadvantage is that there is no way to retrieve the list of what can be done with the object.

In addition, we could not have two methods with the same name, while in OOP we have both **user.delete()** and **data.delete()**

Class

A class is a pattern, it is the template from which objects are created. Defines the properties and capabilities.

Instance is called an object that is created by the class.

Instances have the same interface as the class under which was produced, but each have different data (attributes).

Example: Class x Instance

For example, assume a class man and from it you create instances Jan Novák and Milan Potměšil.

Both instances will have the same methods and attributes as theit class (eg. name and date of birth) and methods (**go()** and **say()**), but the values in them are different (the first instance has the attribute name set to "Jan Novak" and the birth date 30.5.1983, the second "Milan Potmesil" and 11.9.1989).

Basic features of OOP

encapsulation

inheritance

polymorphism

events

Encapsulation

Encapsulation allows you to hide some of the methods and attributes so that they remain applicable only inside the class. The object can be thought of as a black box, which has some interface (interface), through which we pass it instruction / data and it is processed.

We do not know how it works inside, but we know how it behaves externally and how it is used. We cannot therefore cause an error, because we use and we see only what the author made available for us.

Example

An example might be a class like *person* who will have attribute birthDate, and based on it, other attributes: age and adult.

If someone from outside changed birthDate, he would reset the variables age and adult. We say that the internal state of the object would be inconsistent.

In the structured programming this may well be.

In OOP, however, we encapsulate the object and attribute birthDate mark as private, thus invisible from the outside.

Conversely, we will issue out method changeBirthDate(), which substitutes the new date of birth to a variable BirthDate and

also performs the necessary conversion and re-evaluation of the age and adult.

Using the object makes application safe and stable.

Encapsulation thus forces programmers to use object just the right way.

Interface (interface) divides classes into public (public) part and internal structure (private).

Inheritance

-

Suppose we have a general category (*class*) **Man**, that describes the general characteristics of us humans.

You are asked to write another class that will describe adult, class **Adult**. At first glance, we see that class Man and class Adult does not differ much from each other; the main difference is that adults have an identity card and can smoke.

Of course we could write Adult class again from the beginning. But it would have significant drawbacks:

- we're too lazy to do
- all writing code, testing, documentation generation, etc.. would have to do it again
- code would be poorly maintainable, because any change would have to be done in both classes Man and Adult

Much simpler is now to take a class Man and just extend it a bit. In our example – to add the information on the identity card and a function *smoke()*.

This derivation of childs from their parents is named **inheritance**.

There are three interesting rules for inheritance.

Cannot delete

Methods and functions can be modified and/or added, but never deleted. Because parental class already could be used elsewhere - if we modify it, we could create an error.

In our example, we could add more than one function; we could add functions *drive_car()* and *drink_alcohol()*. Also, we could add any number of attributes.

Methods and attributes, when once they are in a class, by no inheritance can removed. If a class Man has a method *say()*, all its inherited methods must have such method.

However, when speaking about methods, we can change the meaning, content of inherited methods. For example, the method *say()* in class Adult can be changed to do something completely different from person than in the classroom. Also from the same class (from Man) we can inherit several different classes Adult1, Adult2, ... etc., and each may have different *say()*.

Note 1: Only methods can be changed, not attributes.

Note 2: We cannot delete a method. But we can change the content of method so that it does nothing. In effect, this is (almost) the same.

Modified type control

The types and type checking, we will talk later in detail. Now just a brief explanation.

Data are of different types, such as numbers and strings (texts). Each type of data is handled differently on your computer. For example, you can multiply numbers (not strings), but the string can be divided into words (it is not possible by numbers).

Therefore, it is natural to process each data type separately and strictly check types not to confuse.

< But what about inheritance? Strictly speaking, the class Man is of a different type than class Adults. But if we were not allowed

to work with Adults as if they were of class Man, we could not find out his name, for example:/EN>

```
Adult.name
```

it will not work, propper would be only

```
Man.name
```

This would be a nonsense, of course, because Adult has an artibute *name* as Man has.

Therefore, the type checking is relaxed so that everything goes to the parent, it is also to do with his child.

The explanation is simple. All attributes and all methods that has the parent, are also included in the child.

But beware! Converse is not true. Child may contain methods and attributes that the parent does not, and therefore type compatibility in the opposite direction does not apply!

Single parent only

The big problem with inheritance is a structure called diamond structure.

When you inherit from parent A two sons, B1 and B2, everything is fine. The son B2 can madify function **func()** to do something else. It is also fine.

But if the grandson had inherited from B1 from B2, was the problem: what function there is the right one? The original or the modified?

This problem is usually preceded by prohibiting inheritance from multiple ancestors.

Polymorphism

Polymorphism is closely linked with inheritance.

We have shown that using inheritance we can change the behaviour of methods. Imagine, then, that from classe Man we inherit two new classes: RedMan and BlueMan. They differ in that their methods will do something completely different than the methods of the parent class Man. This is perfectly legal.

We have also shown that on all placec where a parent class can be used, also an arbitrary descendant can be used. For instance, where we can write `Man.read()`, we can also write `RedMan.read()` or `BlueMan.read()`.

So into some variable `x`, we can insert instance of first, second or third class. All following lines are correct:

`x = instanceMan` or

`x = instanceRedMan` or

`x = instanceBlueMan`

In run-time, into `x` can be stored first, second or third instance. We say that variable is assigned *dynamically*.

But this is a problem. Somewhere later in the program, a command like this `x.read()` can be used. Does this command call *x.read()*, or *x.read()*, or *x.read()* ?

Method of that class, that is just in `x`, is called. This means that the called method also changes dynamically.

Messaging, events

Communication between objects is via message passing, which makes program‘s syntax clear. The code usually looks like this:

```
receiver.methodName (parameters).
```

For example **charles.greeting (neighbor)** could cause the instance **charles** greets instance **neighbor**.

When an event occurs, e.g. user moves the mouse, a hardware interrupt is generated and a message made to circulate around the computer. We say that the **event** was created. Every second there is a lot of events.

It is up to us whether we want to respond to events or not. Mostly it is done so that we choose only the events that interest us and to each such event we will write a piece of program that processes it (serves it). These sections of the program we call **handlers**.

When programming using the IDE, e.g. when working with graphics, it is customary that the creator of IDE creates a **framework**. It contains all the basic functionality and all the complicated parts. The application programmers then is left to complete their handlers through which they will operate events.

This kind of programming is called **event-driven programming**. We'll try it at work with *SceneBuilder*.

Excercise

Give an example why is polymorphism useful.

Give an example how to draft an OOP database of books.

Content