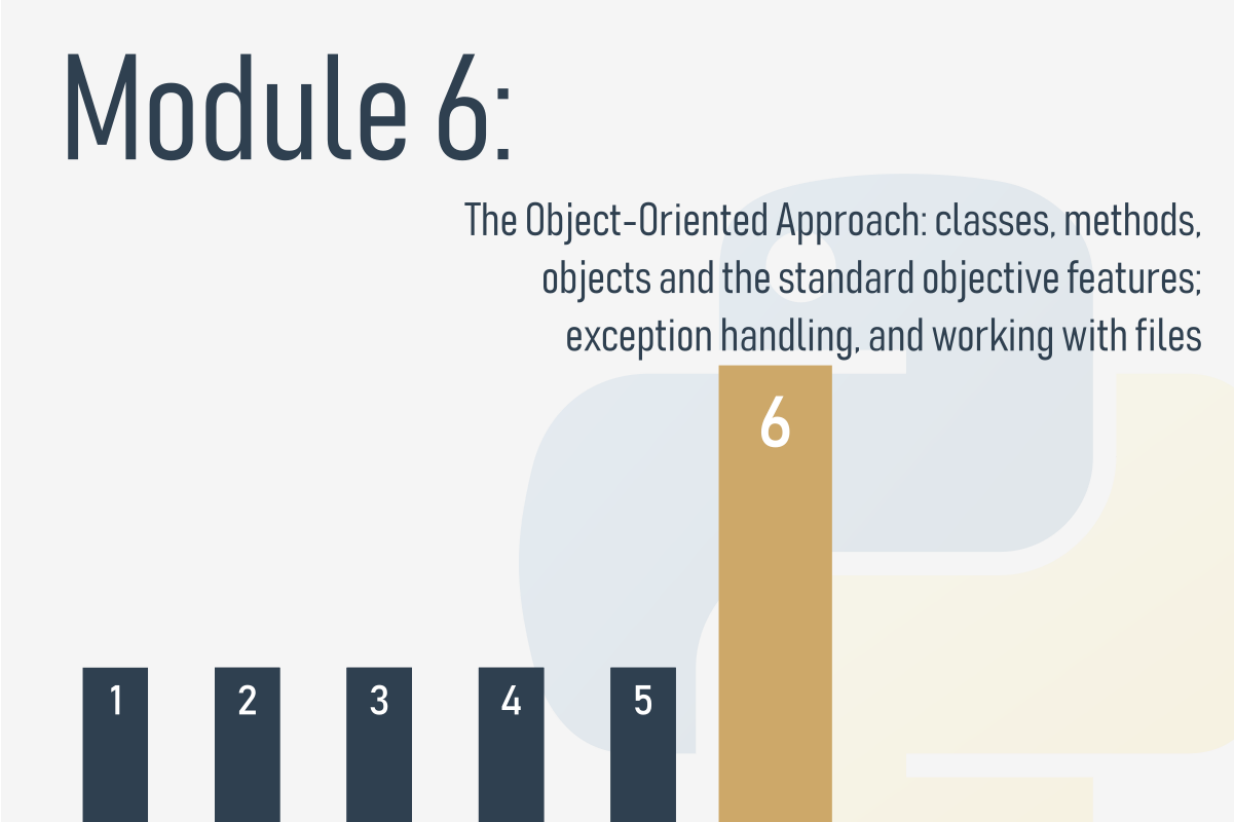
Programming Essentials in Python: Module 6

Python essentials

In this module, you will learn about:

- the object-oriented approach - foundations;
- classes, methods, objects, and the standard objective features;
- exception handling;
- working with files.



The foundations of OOP

The basic concepts of the object-oriented approach

Let's take a step outside of computer programming and computers in general, and discuss object programming issues.

Nearly all of the programs and techniques you have used till now fall under the procedural style of programming. Admittedly, you have made use of some built-in objects, but when referring to them, we just mentioned the absolute minimum.

The procedural style of programming was the dominant approach to software development for decades of IT, and it is still in use today. Moreover, it isn't going to disappear in the future, as it works very well for specific types of projects (generally, not very complex ones and not large ones, but there are lots of exceptions to that rule).

The object approach is quite young (much younger than the procedural approach) and is particularly useful when applied to big and complex projects carried out by large teams consisting of many developers.

This kind of understanding of a project's structure makes many important tasks easier, e.g., dividing the project into small, independent parts, and independent development of different project elements.

Python is a universal tool for both object and procedural programming. It may be successfully utilized in both spheres.

Furthermore, you can create lots of useful applications, even if you know nothing about classes and objects, but you have to keep in mind that some of the problems (e.g., graphical user interface handling) may require a strict object approach.

Fortunately, object programming is relatively simple.

Procedural vs. the object-oriented approach

In the **procedural approach**, it's possible to distinguish two different and completely separate worlds: **the world of data, and the world of code**. The world of data is populated with variables of different kinds, while the world of code is inhabited by code grouped into modules and functions.

Functions are able to use data, but not vice versa. Furthermore, functions are able to abuse data, i.e., to use the value in an unauthorized manner (e.g., when the sine function gets a bank account balance as a parameter).

We said in the past that data cannot use functions. But is this entirely true? Are there some special kinds of data that can use functions?

Yes, there are - the ones named methods. These are functions which are invoked from within the data, not beside them. If you can see this distinction, you've taken the first step into object programming.

The **object approach** suggests a completely different way of thinking. The data and the code are enclosed together in the same world, divided into classes.

Every **class is like a recipe which can be used when you want to create a useful object** (this is where the name of the approach comes from). You may produce as many objects as you need to solve your problem.

Every object has a set of traits (they are called properties or attributes - we'll use both words synonymously) and is able to perform a set of activities (which are called methods).

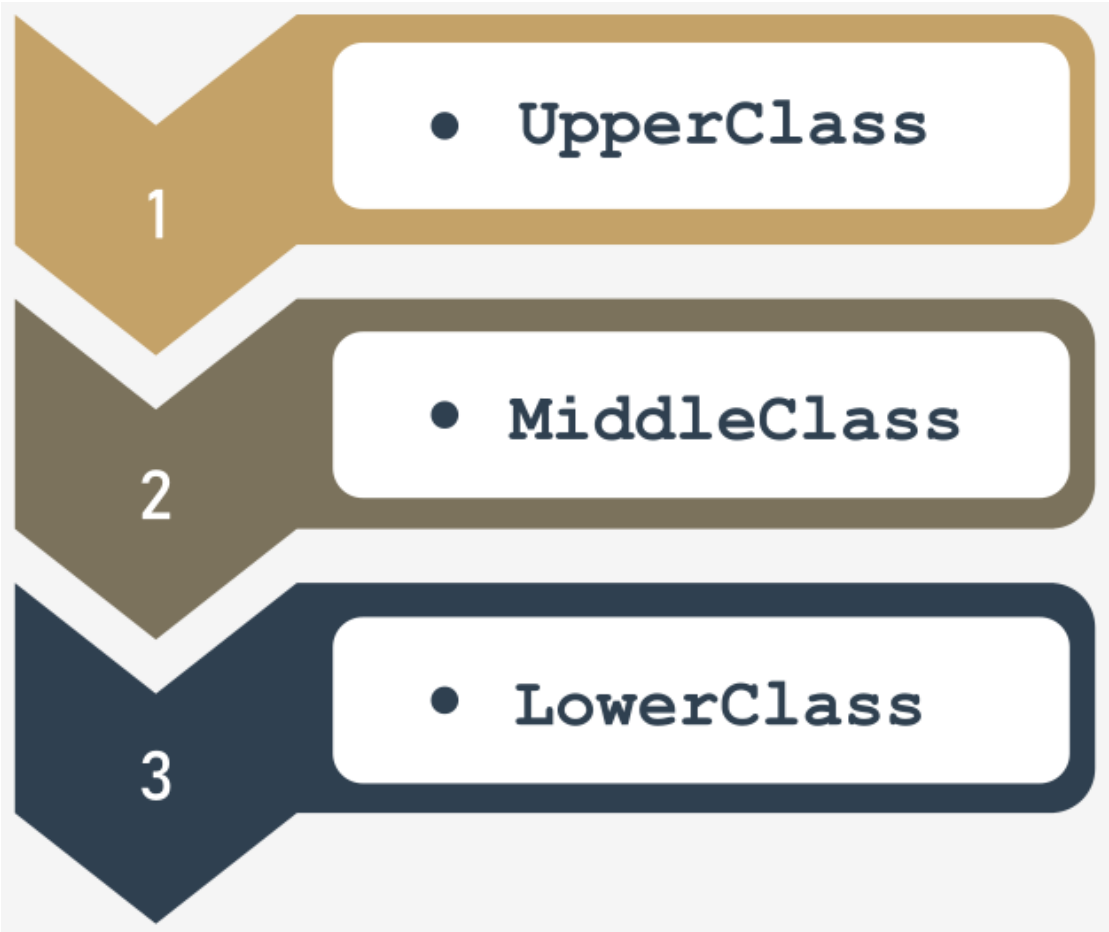
The recipes may be modified if they are inadequate for specific purposes and, in effect, new classes may be created. These new classes inherit properties and methods from the originals, and usually add some new ones, creating new, more specific tools.

Objects are incarnations of ideas expressed in classes, like a cheesecake on your plate is an incarnation of the idea expressed in a recipe printed in an old cookbook.

The objects interact with each other, exchanging data or activating their methods. A properly constructed class (and thus, its objects) are able to protect the sensible data and hide it from unauthorized modifications.

There is no clear border between data and code: they live as one in objects.

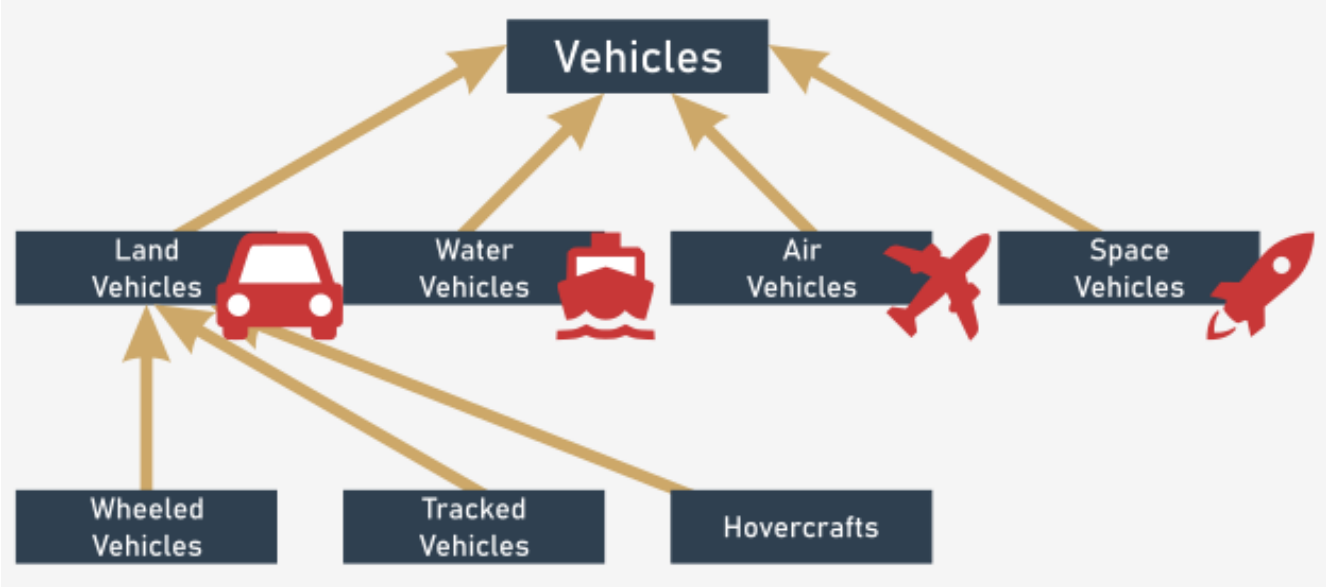
All these concepts are not as abstract as you may at first suspect. On the contrary, they all are taken from real-life experiences, and therefore are extremely useful in computer programming: they don't create artificial life - **they reflect real facts, relationships, and circumstances**.



Class hierarchies

The word *class* has many meanings, but not all of them are compatible with the ideas we want to discuss here. The *class* that we are concerned with is like a *category*, as a result of precisely defined similarities.

We'll try to point out a few classes which are good examples of this concept.



Let's look for a moment at vehicles. All existing vehicles (and those that don't exist yet) are **related by a single, important feature**: the ability to move. You may argue that a dog moves, too; is a dog a vehicle? No, it isn't. We have to improve the definition, i.e., enrich it with other criteria, distinguishing vehicles from other beings, and creating a stronger connection. Let's take the following circumstances into consideration: vehicles are artificially created entities used for transportation, moved by forces of nature, and directed (driven) by humans.

Based on this definition, a dog is not a vehicle.

The *vehicles* class is very broad. Too broad. We have to define some more **specialized classes**, then. The specialized classes are the **subclasses**. The *vehicles* class will be a **superclass** for them all.

Note: **the hierarchy grows from top to bottom, like tree roots, not branches**. The most general, and the widest, class is always at the top (the superclass) while its descendants are located below (the subclasses).

By now, you can probably point out some potential subclasses for the *Vehicles* superclass. There are many possible classifications. We've chosen subclasses based on the environment, and say that there are (at least) four subclasses:

- land vehicles;
- water vehicles;
- air vehicles;
- space vehicles.

In this example, we'll discuss the first subclass only - land vehicles. If you wish, you can continue with the remaining classes.

Land vehicles may be further divided, depending on the method with which they impact the ground. So, we can enumerate:

- wheeled vehicles;
- tracked vehicles;
- hovercrafts.

The hierarchy we've created is illustrated by the figure.

Note the direction of the arrows - they always point to the superclass. The top-level class is an exception - it doesn't have its own superclass.

Another example is the hierarchy of the taxonomic kingdom of animals.

We can say that all *animals* (our top-level class) can be divided into five subclasses:

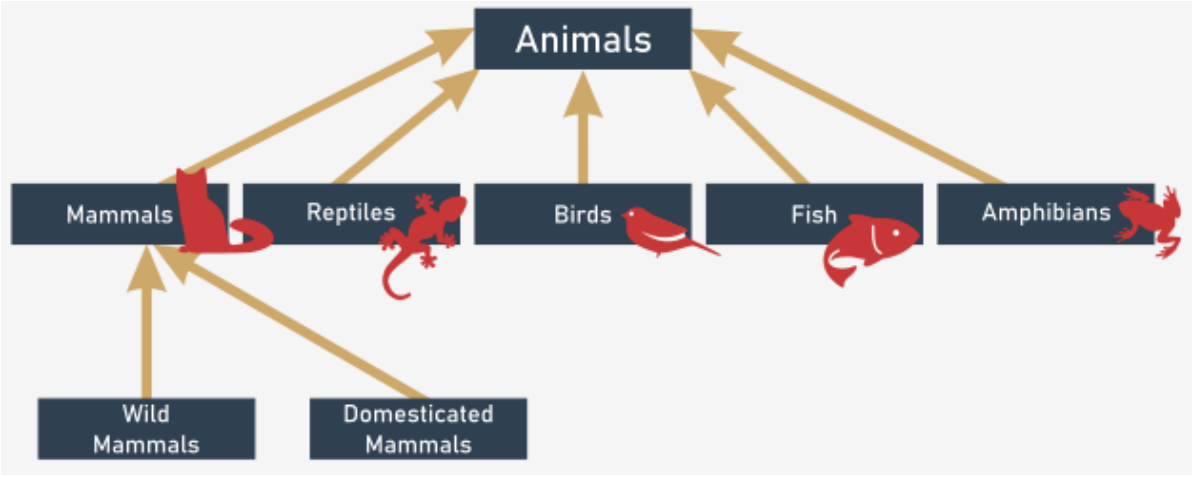
- mammals;
- reptiles;
- birds;
- fish;
- amphibians.

We'll take the first one for further analysis.

We have identified the following subclasses:

- wild mammals;
- domesticated mammals.

Try to extend the hierarchy any way you want, and find the right place for humans.



What is an object?

A class (among other definitions) is a **set of objects**. An object is **a being belonging to a class**.

An object is **an incarnation of the requirements, traits, and qualities assigned to a specific class**. This may sound simple, but note the following important circumstances. Classes form a hierarchy. This may mean that an object belonging to a specific class belongs to all the superclasses at the same time. It may also mean that any object belonging to a superclass may not belong to any of its subclasses.

For example: any personal car is an object belonging to the *wheeled vehicles* class. It also means that the same car belongs to all superclasses of its home class; therefore, it is a member of the *vehicles* class, too. Your dog (or your cat) is an object included in the *domesticated mammals* class, which explicitly means that it is included in the *animals* class as well.

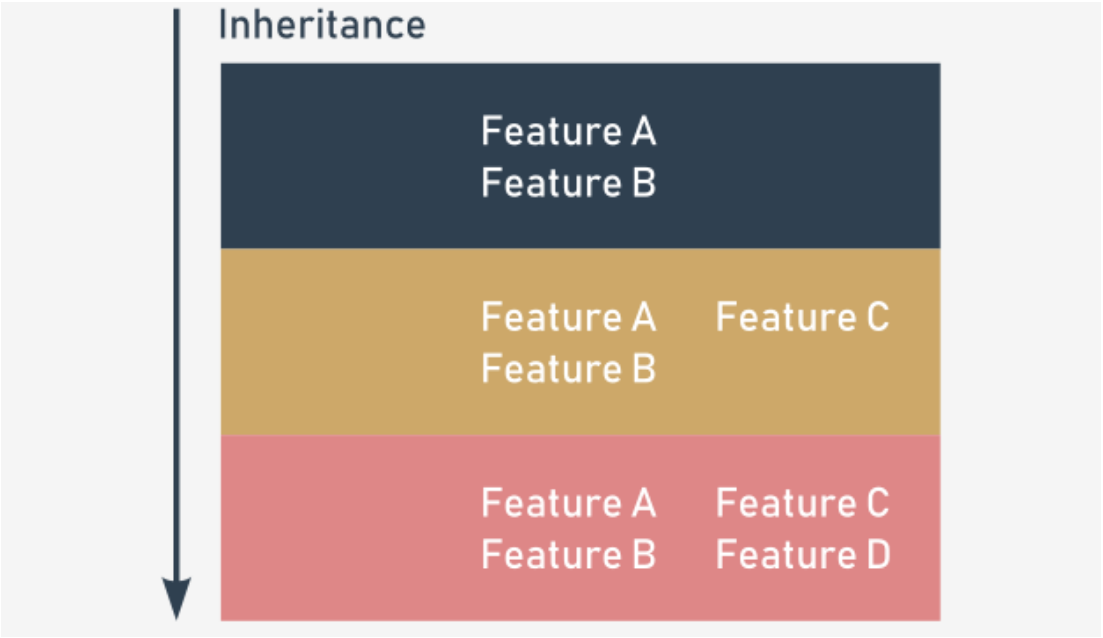
Each **subclass is more specialized** (or more specific) than its superclass. Conversely, each **superclass is more general** (more abstract) than any of its subclasses. Note that we've presumed that a class may only have one superclass - this is not always true, but we'll discuss this issue more a bit later.

Inheritance

Let's define one of the fundamental concepts of object programming, named **inheritance**. Any object bound to a specific level of a class hierarchy **inherits all the traits (as well as the requirements and qualities) defined inside any of the superclasses**.

The object's home class may define new traits (as well as requirements and qualities) which will be inherited by any of its superclasses.

You shouldn't have any problems matching this rule to specific examples, whether it applies to animals, or to vehicles.



What does an object have?

The object programming convention assumes that **every existing object may be equipped with three groups of attributes**:

- an object has a **name** that uniquely identifies it within its home namespace (although there may be some anonymous objects, too)
- an object has a **set of individual properties** which make it original, unique or outstanding (although it's possible that some objects may have no properties at all)
- an object has a **set of abilities to perform specific activities**, able to change the object itself, or some of the other objects.

There is a hint (although this doesn't always work) which can help you identify any of the three spheres above. Whenever you describe an object and you use:

- a noun - you probably define the object's name;
- an adjective - you probably define the object's property;
- a verb - you probably define the object's activity.

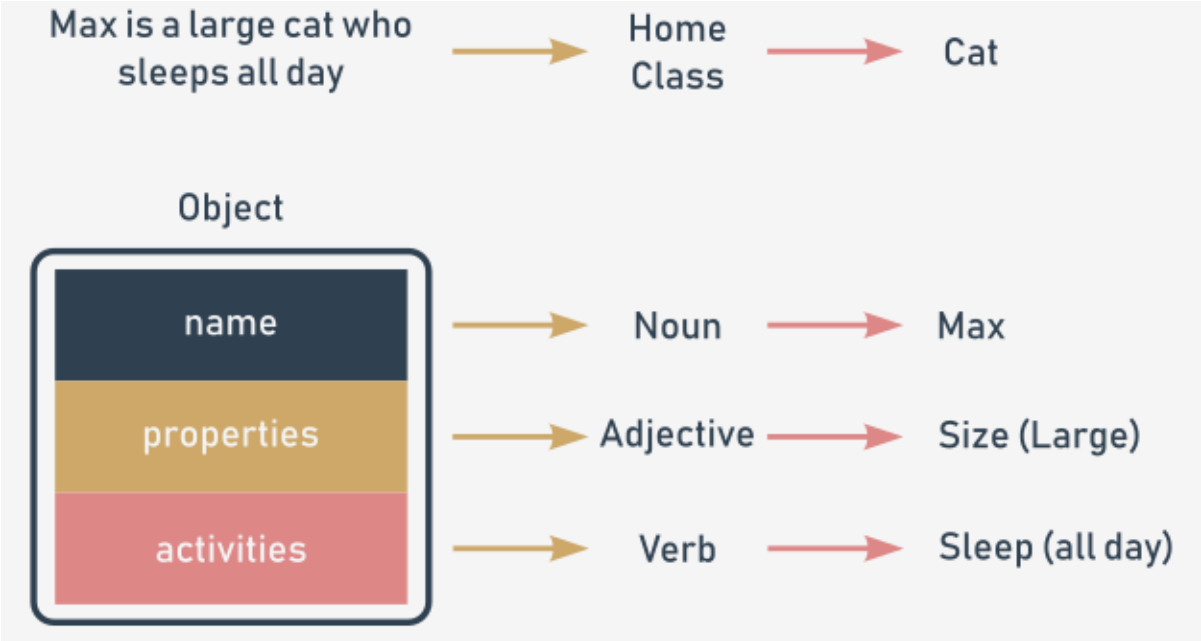
Two sample phrases should serve as a good example:

- Max is a large cat who sleeps all day.

  Object name = Max
  Home class = Cat
  Property = Size (large)
  Activity = Sleep (all day)

- A pink Cadillac went quickly.

  Object name = Cadillac
  Home class = Wheeled vehicles
  Property = Color (pink)
  Activity = Go (quickly)



Your first class

Object programming is **the art of defining and expanding classes**. A class is a model of a very specific part of reality, reflecting properties and activities found in the real world.

The classes defined at the beginning are too general and imprecise to cover the largest possible number of real cases.

There's no obstacle to defining new, more precise subclasses. They'll inherit everything from their superclass, so the work that went into its creation isn't wasted.

The new class may add new properties and new activities, and therefore may be more useful in specific applications. Obviously, it may be used as a superclass for any number of newly created subclasses.

The process doesn't need to have an end. You can create as many classes as you need.

The class you define has nothing to do with the object: **the existence of a class does not mean that any of the compatible objects will automatically be created**. The class itself isn't able to create an object - you have to create it yourself, and Python allows you to do this.

It's time to define the simplest class and to create an object. Take a look at the example below:

```
class TheSimplestClass:
    pass
```

We've defined a class there. The class is rather poor: it has neither properties nor activities. It's **empty**, actually, but that doesn't matter for now. The simpler the class, the better for our purposes.

The definition begins with the keyword `class`. The keyword is followed by an **identifier which will name the class** (note: don't confuse it with the object's name - these are two different things).

Next, you add a **colon**:), as classes, like functions, form their own nested block. The content inside the block define all the class's properties and activities.

The `pass` keyword fills the class with nothing. It doesn't contain any methods or properties.

Your first object

The newly defined class becomes a tool that is able to create new objects. The tool has to be used explicitly, on demand.

Imagine that you want to create one (exactly one) object of the `TheSimplestClass` class.

To do this, you need to assign a variable to store the newly created object of that class, and create an object at the same time.

You do it in the following way:

```
myFirstObject = TheSimplestClass()
```

Note:

- the class name tries to pretend that it's a function - can you see this? We'll discuss it soon;
- the newly created object is equipped with everything the class brings; as this class is completely empty, the object is empty, too.

  The act of creating an object of the selected class is also called an **instantiation** (as the object becomes an **instance of the class**).

  Let's leave classes alone for a short moment, as we're now going to tell you a few words about *stacks*. We know the concept of classes and objects may not be fully clear yet. Don't worry, we'll explain everything very soon.