The Story of Object-Oriented Programming

The interesting story of the object-oriented programming.

By Omar Elgabry, Oct 24, 2016 · 8 min read
from https://medium.com/omarelgabrys-blog/the-story-of-object-oriented-programming-12d1901a1825

This is an article on the main features and aspects of object-oriented programming. Mastering OOP is essential for any developer who wants to build a high quality software. So, let's get started!

In object-oriented programming, your program will be split into several small, manageable, reusable programs. Each small program has it's own identity, data, logic and how it's going to communicate with the rest of the other small programs.

Objects

When you think about Objects, you should think about the real world situations. Object-Orientation was intended to be closer to the real world, hence, make it easier and more realistic.

What are the Objects?, Well, Objects could be:

- Something visible for you, like
  the *car*, *phone*, *apple*, *duck*, *lamp*.
- Something that you can't touch, like the *Time*, *Event*, *Account*.

Objects

Each Objects has it's own *attributes*, and behavior. Objects are separate from one another. They have their own existence, their own identity that is independent of other objects.

What do we mean by attributes?

It's the characteristics or the properties of the Object, like for example, in case of a *duck*, it's *weight* & *color*. They describe the current state of an object. The state of one object is independent of another. Maybe you will have a *duck* that's *white* and another one is *black*.

What do I mean by behavior?

Behavior is things that the Object can do, in case of *duck*, It can *fly*. Another example, in case of *Account*, we can *deposit* or/and *withdraw* from

any *account*. Again, Each account is totally independent on the other. We may *deposit* from an account and *withdraw* from another account.

Another thing you have to notice before leaving Objects part is, Objects are *nouns*, they aren't behaviors nor properties. You should differentiate of what's actually an Object and what's a behavior, and a property.

Now, the questions is, How do we construct these objects in our program?

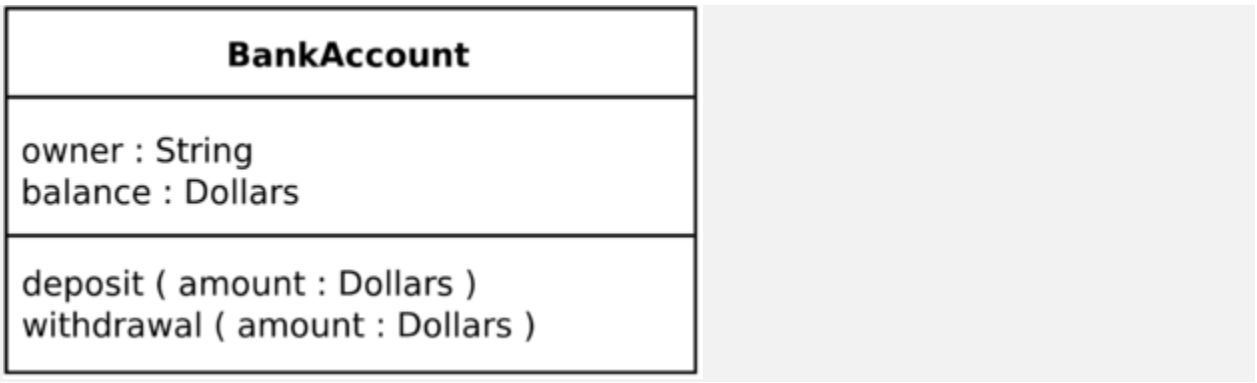
Well, They don't magically appear in our program, and to do that, there is another word we must explore that goes hand in hand with object, and that word is **Class**.

Class

When you start talking about Objects, you have to mention Classes.

So, *what is* a class?

Well, a class is the place where you can identify the behaviors and properties of an Object. So, the properties and behavior of an Object, will be defined inside a class.



Bank Account Class

So, a class is a *blueprint*, it provides the definition for an Object, the definition says what are the properties and the behaviors of the Object. Let's take an example, When you build a *house*, you start making the *blueprint*, in other words you start defining it's properties and behaviors, and then, you use this *blueprint* to build different *houses*, or more formally, different objects from the same *blueprint*.

So, Objects from the same *blueprint*, or more formally the same class, share the same properties and behaviors. However, even they share the same Class, each Object instantiated from this Class is independent on the other.

It's enough for Objects and Classes, Let's move on to other important concepts in OOP.

Abstraction

Abstraction means you start focusing on the common properties and behaviors of some Objects, and we automatically will discard what's unimportant or irrelevant.

What came into your mind when I say “Car”?

If we say Car, we didn’t explicitly say if the car is BMW or Audi, if it’s red or black, if it’s small or large. Your mental model of a car might have a color and size, but, it’s unlikely to have a smell or a flavor because those things are irrelevant to the idea of a car.

Cars

So, we abstracted the idea of what a car means. Abstraction is the heart of Object-Oriented Programming, because it’s what we are doing when we make a class.

In case of Cars, maybe you will have different brands like, BMW*,* Audi*,* Bugatti*,* Chevrolet*,* Citroen *and so on. But, we don’t create a class for each brand, instead, we will abstract; focus on the common essential qualities of the idea, and we will write* ***one*** *class called Car.*

And As a way to discard what’s unimportant, we should focus on how the class should look like for this application under these circumstances at this

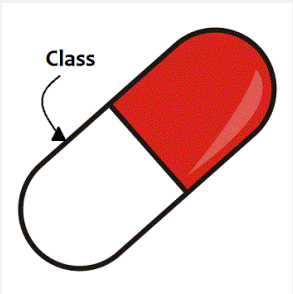
time. In Other words, you care only about the information that you need in this situation.

Later on, maybe you will be having a specific car brand that has a property or a behavior that's not common between other car brands, and it's specific to it's type. In this case, you will have to do of what's called "*Inheritance*", and we will get into it later.

Encapsulation

Instead of having a procedural, long program, we start to decompose our program into small reusable, manageable chunks. Encapsulation implies the idea of breaking down our program into small mini-programs; classes, where each class has a set of related attributes and behaviors.

Encapsulation also implies the idea of hiding the content of a Class, unless it's necessary to expose. We need to restrict the access to our class as much as we can, so that we can change the properties and the behaviors only from inside the class.



Encapsulation

But, how can we do that?

Start building our blueprints, or classes and define their properties and behaviors. Then, restrict access to the inner works of that class by hiding the attributes and methods so that they're only accessible from inside the class scope.

What's the difference between Encapsulation and Abstraction?

Encapsulation is a strategy used as part of abstraction. When we encapsulate, we *abstract* away the implementation.

Abstraction is a more generic term, and it's often not possible without hiding the object's class content by encapsulation; if a class exposes its internal content, thus, it cannot be abstracted.

Why should we hide attributes and methods of an object?

So other parts of the application won't change the current object attributes.

If we have a bank account object, we can use the deposit and withdraw methods from other parts of the application, and they can change the balance, but the attribute can't be directly changed from outside the class.

*Another example, If you have a lamp object, and assigned it's property called "*turnedOff*" to* **true**. *If there are some other class that have access to this property, they can change it accidentally to* **false***!.*

And, If we've restricted direct access to a piece of data, we only have to worry about this one class, and we don't have to worry about breaking the

other parts of the application that might depends on this piece of data. So, we can more safely change the way the object works.

If you have a bank account object, and you want to change how the balance is stored. Then, the other parts of the application don't have to worry about this change, Why? Because the balance attribute of the account object is already hidden; can't be accessed directly.

So, It's about reducing dependencies between different parts of the application.

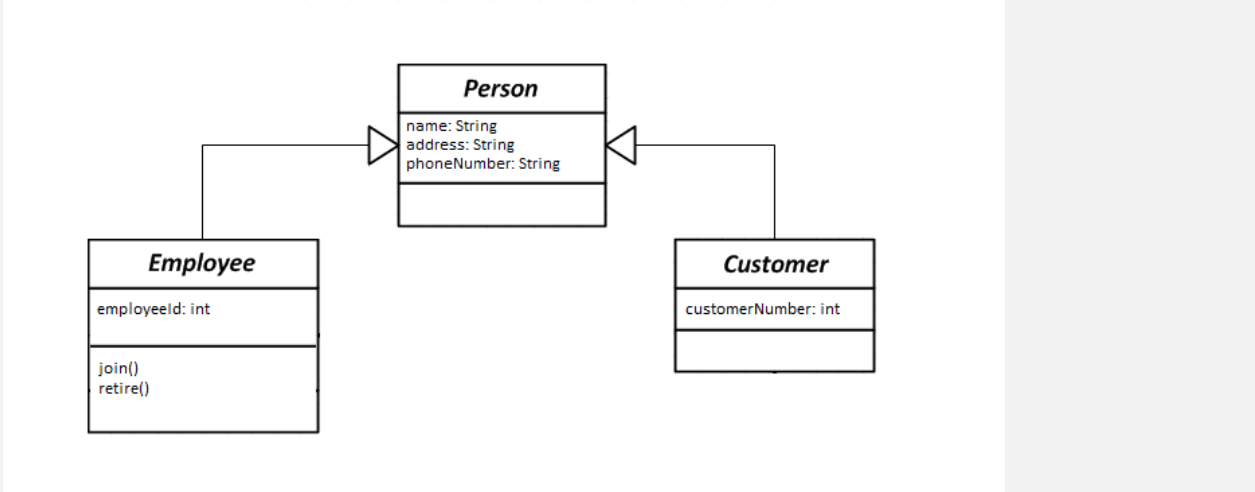
But how much should you hide?

Well, the rule is, **as much as possible**.

So, the idea of encapsulation is that you enclose your object's attributes and methods, and then you hide everything about that object except what is absolutely necessary to expose.

Inheritance

Abstraction is, instead of creating different classes, we can instead create one generic class that has the common, and essential properties and behavior of those classes, while Inheritance is inheriting these common properties and behaviors, so that we can create a new class, but instead of writing it from scratch, we can base it on an existing class.



Inheritance

A Person class with a attributes; name and address and phone number. Now, you need to create another one called Customer. It has the same attributes as Person, but it also has ***additional*** *attributes and methods.*

So, we are going to inherit from the Person class. And the Customer class will automatically has everything that the Person class has, all its attributes, all its behaviors, without us having to write any code. In addition, If we make a change in the Person class, it will automatically cascade down and affect the inheriting classes.

Now, we can add specific properties and behaviors that's not shared across the inheriting classes in their specific classes.

Overriding In Inheritance

Overriding the Super class methods is not a good case practice when the Super class is a Concrete class. That's why developers tend to use Interfaces instead.

Inheritance defines a relationship

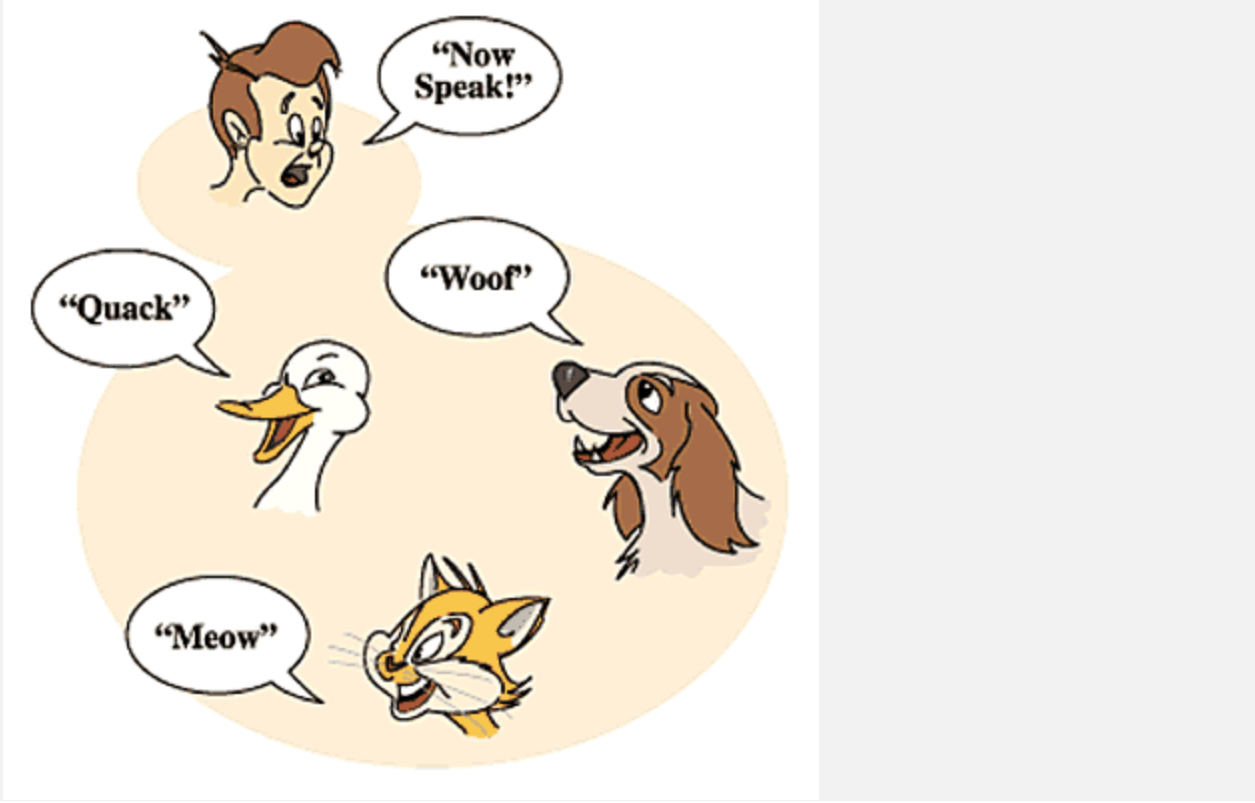
So, a Customer IS-A Person, and an Employee IS-A Person, and so on. The term that's most commonly used for this relationship is, Person class is the **Super** class, while the customer class is called the **Sub** class. Or, We can also hear this described as the **Parent** class and the **Child** class.

Multiple Inheritance

In C++, & Python allow you to inherit from more than one Super Class; Multiple Inheritance. But, it can get confusing as it's much more common to inherit from one Super class. In Java & C#, You only inherit from one Super Class.

Polymorphism

Polymorphism; is the state where an object can take the shape of many different forms, and lets us do the right thing at the right time.



Polymorphism

If you have Animal class, and inheriting classes; Dog, Cat, & Duck. Each of the inheriting classes override the ***speak*** *method (as each animal has a different voice).*

Now, we can call the speak method on any animal object, without knowing exactly what class the animal object was instantiated from, and it will do the correct behavior.

It's worth mentioning that the speak method in the Animal class will be triggered unless it's overridden.

Polymorphism is the flexibility, that triggers the correct behavior.

Putting All Together — Animals Project

I've introduced the concepts of OOP. Here is a complete snippet written in **Java** that shows a real example combining all of the OOP concepts together.

Download the Java code snippets, and share if you find them useful.
