JUNE 27, 2018/#PROGRAMMING

How to explain object-oriented programming concepts to a 6-year-old

by Alexander Petkov

Have you noticed how the same cliche questions always get asked at job interviews — over and over again?

I'm sure you know what I mean.

For example:

Where do you see yourself in five years?

or, even worse:

What do you consider to be your greatest weakness?

Ugh…give me a break. I consider answering this question a great weakness! Anyway, not my point.

As trivial as questions like these may be, they are important because they give clues about you. Your current state of mind, your attitude, your perspective.

When answering, you should be careful, as you may reveal something you later regret.

Today I want to talk about a similar type of question in the programming world:

What are the main principles of Object-Oriented Programming?

I've been on both sides of this question. It's one of those topics that gets asked so often that you can't allow yourself to not know.

Junior and entry-level developers usually have to answer it. Because it's an easy way for the interviewer to tell three things:

1. **Did the candidate prepare for this interview?**
   Bonus points if you hear an answer immediately — it shows a serious approach.
2. **Is the candidate past the tutorial phase?**
   Understanding the principles of Object-Oriented Programming (OOP) shows you've gone beyond copy and pasting from tutorials — you already see things from a higher perspective.
3. **Is the candidate's understanding deep or shallow?**
   The level of competence on this question often equals the level of competence on **most other subjects**. Trust me.
   The four principles of object-oriented programming are **encapsulation**, **abstraction**, **inheritance**, and **polymorphism**.
   These words may sound scary for a junior developer. And the complex, excessively long explanations in Wikipedia sometimes double the confusion.

   That's why I want to give a simple, short, and clear explanation for each of these concepts. It may sound like something you explain to a child, but I would actually love to hear these answers when I conduct an interview.

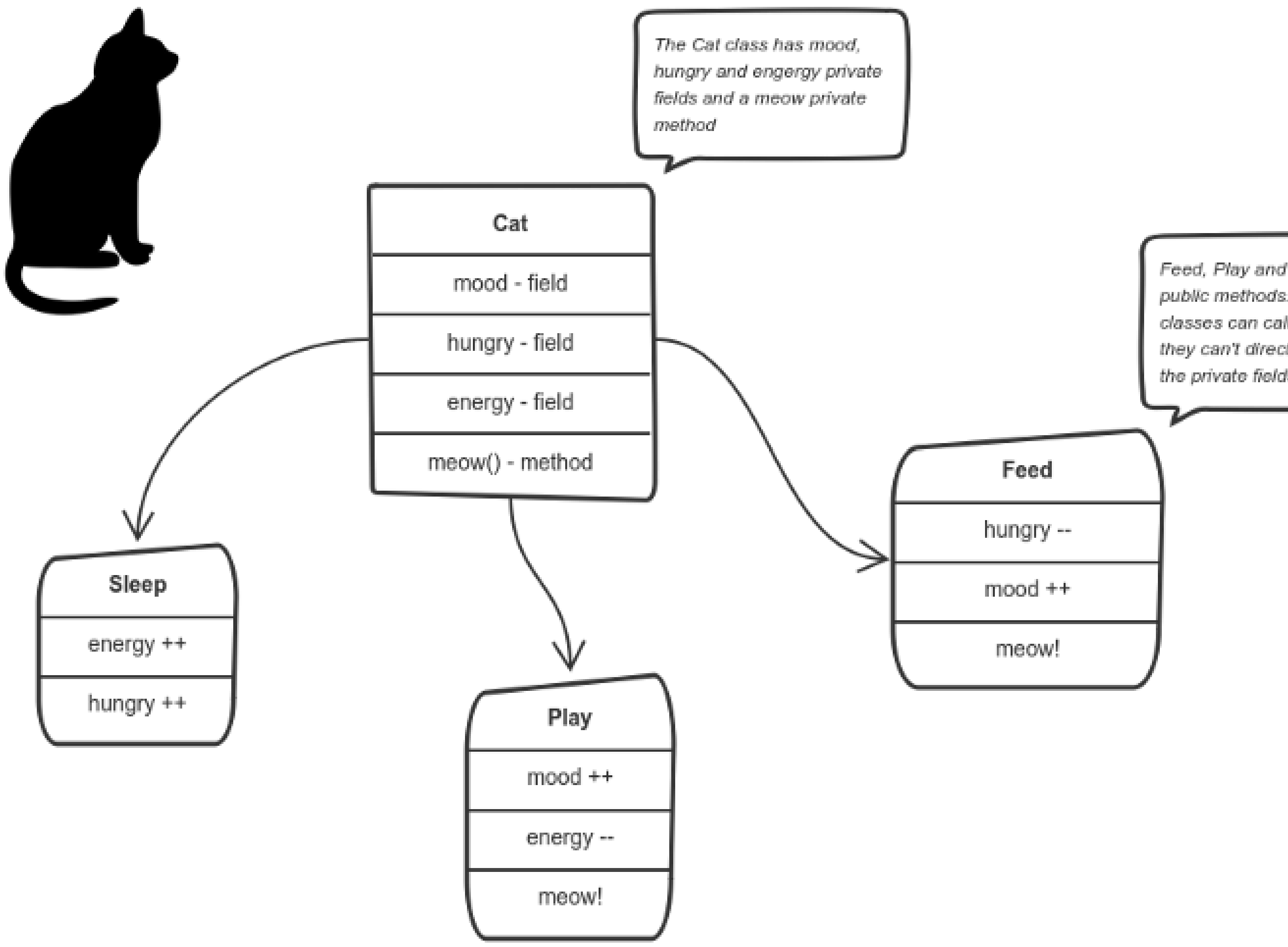
   Encapsulation

   Say we have a program. It has a few logically different objects which communicate with each other — according to the rules defined in the program.

   Encapsulation is achieved when each object keeps its state **private**, inside a class. Other objects don't have direct access to this state. Instead, they can only call a list of public functions — called methods.
   So, the object manages its own state via methods — and no other class can touch it unless explicitly allowed. If you want to

communicate with the object, you should use the methods provided. But (by default), you can't change the state.

Let's say we're building a tiny Sims game. There are people and there is a cat. They communicate with each other. We want to apply encapsulation, so we encapsulate all "cat" logic into a `Cat` class. It may look like this:



You can feed the cat. But you can't directly change how hungry the cat is.
Here the "state" of the cat is the **private variables** `mood`, `hungry` and `energy`. It also has a private method `meow()`. It can call it whenever it wants, the other classes can't tell the cat when to meow.

What they can do is defined in the **public methods** `sleep()`,`play()` and `feed()`. Each of them modifies the internal state somehow and may invoke `meow()`. Thus, the binding between the private state and public methods is made. This is encapsulation.

Abstraction

Abstraction can be thought of as a natural extension of encapsulation.

In object-oriented design, programs are often extremely large. And separate objects communicate with each other a lot. So maintaining a large codebase like this for years — with changes along the way — is difficult.

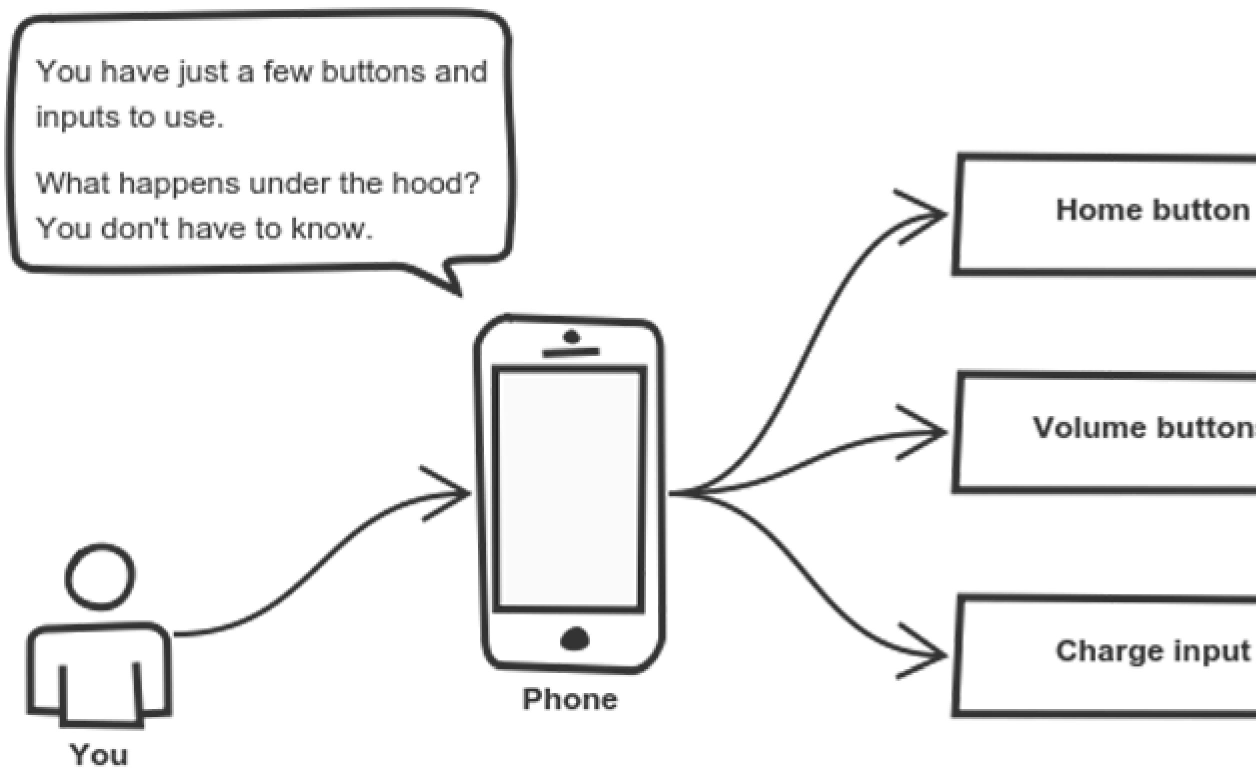
Abstraction is a concept aiming to ease this problem.

Applying abstraction means that each object should **only** expose a high-level mechanism for using it. This mechanism should hide internal implementation details. It should only reveal operations relevant for the other objects.

Think — a coffee machine. It does a lot of stuff and makes quirky noises under the hood. But all you have to do is put in coffee and press a button.

Preferably, this mechanism should be easy to use and should rarely change over time. Think of it as a small set of public methods which any other class can call without "knowing" how they work.

Another real-life example of abstraction?
Think about how you use your phone:


You have just a few buttons and inputs to use.
What happens under the hood? You don't have to know.
Home button
Volume button
Charge input
Phone
You

Cell phones are complex. But using them is simple.
You interact with your phone by using only a few buttons.
What's going on under the hood? You don't have to know — implementation details are hidden. You only need to know a short set of actions.

Implementation changes — for example, a software update — rarely affect the abstraction you use.

Inheritance

OK, we saw how encapsulation and abstraction can help us develop and maintain a big codebase.

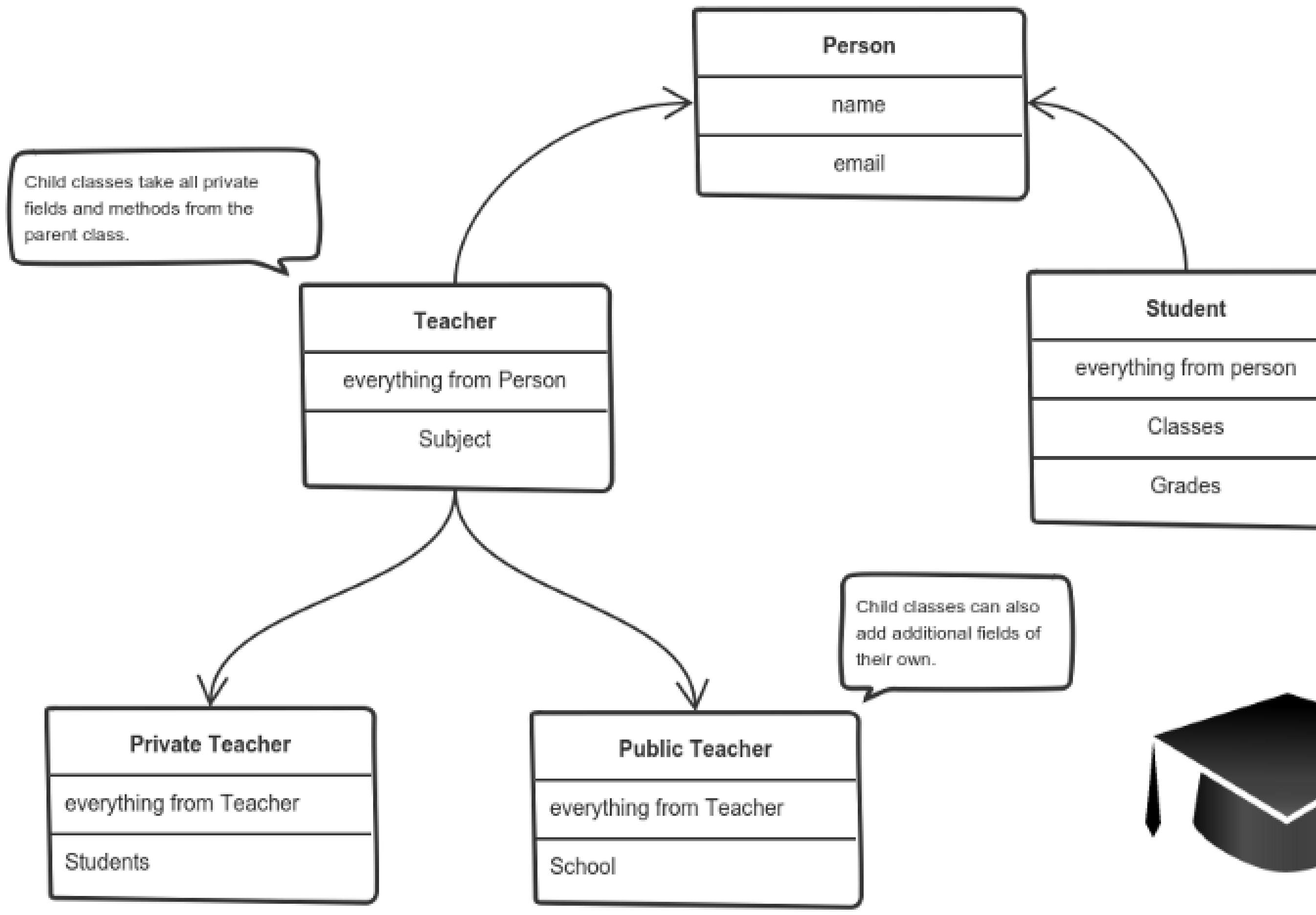
But do you know what is another common problem in OOP design?

Objects are often very similar. They share common logic. But they're not **entirely** the same. Ugh…
So how do we reuse the common logic and extract the unique logic into a separate class? One way to achieve this is inheritance.

It means that you create a (child) class by deriving from another (parent) class. This way, we form a hierarchy.

The child class reuses all fields and methods of the parent class (common part) and can implement its own (unique part).

For example:



A private teacher is a type of teacher. And any teacher is a type of Person.
If our program needs to manage public and private teachers, but also other types of people like students, we can implement this class hierarchy.

This way, each class adds only what is necessary for it while reusing common logic with the parent classes.

Polymorphism

We're down to the most complex word! Polymorphism means "many shapes" in Greek.

So we already know the power of inheritance and happily use it. But there comes this problem.

Say we have a parent class and a few child classes which inherit from it. Sometimes we want to use a collection — for example a list — which contains a mix of all these classes. Or we have a method implemented for the parent class — but we'd like to use it for the children, too.

This can be solved by using polymorphism.

Simply put, polymorphism gives a way to use a class exactly like its parent so there's no confusion with mixing types. But each child class keeps its own methods as they are.
This typically happens by defining a (parent) interface to be reused. It outlines a bunch of common methods. Then, each child class implements its own version of these methods.

Any time a collection (such as a list) or a method expects an instance of the parent (where common methods are outlined), the language takes care of evaluating the right implementation of the common method — regardless of which child is passed.

Take a look at a sketch of geometric figures implementation. They reuse a common interface for calculating surface area and perimeter:

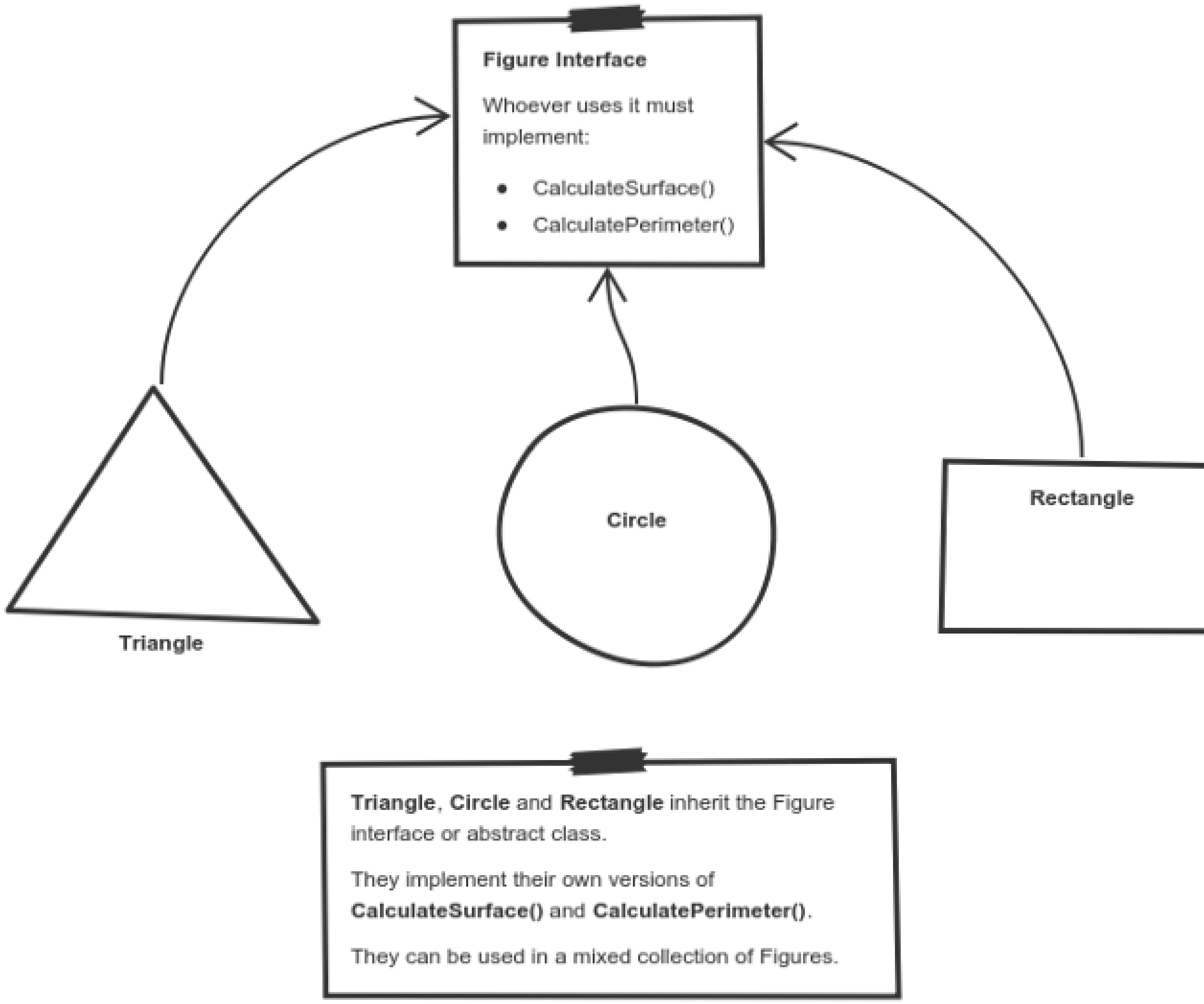


Figure Interface
Whoever uses it must implement:
• CalculateSurface()
• CalculatePerimeter()
Triangle
Circle
Rectangle
Triangle, **Circle** and **Rectangle** inherit the Figure interface or abstract class.
They implement their own versions of **CalculateSurface()** and **CalculatePerimeter()**.
They can be used in a mixed collection of Figures.

Triangle, Circle, and Rectangle now can be used in the same collection
Having these three figures inheriting the parent `Figure Interface` lets you create a list of mixed `triangles`, `circles`, and `rectangles`. And treat them like the same type of object. Then, if this list attempts to calculate the surface for an element, the correct method is found and executed. If the element is a triangle, triangle's `CalculateSurface()` is called. If it's a circle — then circle's `CalculateSurface()` is called. And so on.

If you have a function which operates with a figure by using its parameter, you don’t have to define it three times — once for a triangle, a circle, and a rectangle.

You can define it once and accept a `Figure` as an argument. Whether you pass a triangle, circle or a rectangle — as long as they implement `CalculateParamter()`, their type doesn’t matter.
I hope this helped. You can directly use these exact same explanations at job interviews.

If you find something still difficult to understand — don’t hesitate to ask in the comments below.

What’s next?

Being prepared to answer one of the all-time interview question classics is great — but sometimes you never get called for an interview.

Next, I’ll focus on what employers want to see in a junior developer and how to stand out from the crowd when job hunting.

Stay tuned.
