# The General Concept of Polymorphism

## 1. Introduction

The [dictionary](https://www.lexico.com/en/definition/polymorphism) defines polymorphism as “the condition of occurring in several different forms”. Since polymorphism appears in multiple fields of science, it’s a general definition. Maybe the computing definition will be more meaningful for us: “**a feature of a programming language that allows routines to use variables of different types at different times**“.

In this tutorial, we’ll explore what this sentence means. But first, we should understand how we perceive the surrounding world.

## 2. Polymorphism in the Real World

**Our mind generalizes things around us.** It looks for similarities between them and puts them into categories. These categories can be general, therefore containing many items—for example, animals. On the other hand, they can be rather specific. Hence, they contain significantly fewer elements – for instance, black, three-legged tables.

**We call this categorization process classification. As a result, the categories are called classes.**

Evolution led us to develop this behavior because it helped us to make better and faster decisions. As a result, we had a better chance of surviving. Think of everyday situations a prehistoric man could face. If they saw a tiger, they ran or hid. If they saw a rabbit, they didn’t. They didn’t meet every rabbit or tiger, but they recognized one when they saw them because they saw enough of them to create classes, which described these animals accurately enough.

**But similar things don’t just have similar properties. Besides, they have similar behaviors. And that’s what makes polymorphism really powerful.**

Think of employees. From the employer’s perspective, the most important behavior of an employee is working. And every employer knows how to do their job. At least in an ideal world, but let’s assume that we live in one.

**It means that every morning the boss only has to tell the employees to start working. And they do, without the boss’ interference.** (Yes, it’s a ridiculous assumption. But let’s remember, it’s an ideal world.) The boss doesn’t have to tell every employee what to do. **They autonomously do what they have to.**

How would it look like if they weren’t autonomous? The boss would go around like this:

* What’s your role?
* I’m a backend developer.
* Then create an endpoint, which executes a batch job. When the job is done, store the result in the database and send a notification to the user.

Then he moves to the next employee:

* What’s your role?
* I’m a UX designer.
* Then create a wireframe, focusing on accessibility. Use the mobile-first approach. If it’s ready, run some research with a focus group.

And then this would go on and on. (We’re aware that this exists. It’s called micromanagement. But fortunately, it doesn’t exist in our little utopistic world.)

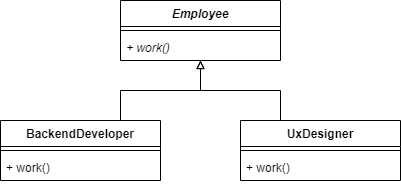
For larger companies, the second approach isn’t feasible. That’s why they rely on employees to know how to do their job.

## 3. Polymorphism in Programming

**In programming, they usually associate polymorphism with the object-oriented paradigm. Since it’s far the most popular paradigm, we’ll also talk about that. However, note that functional and structural programming has polymorphic capabilities, such as higher-order functions or function pointers.**

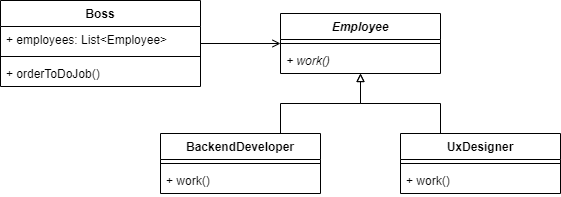
Let’s revisit our employee example in OO programming. How would we model this?

The simplest solution is to have an *Employee* superclass and two children: *BackendDeveloper* and *UxDesigner*:



As we can see, the *Employee* class has a single abstract method: *work()*. Both of the child classes implement that method differently.

With this in place, we can create a *Boss* class, which can easily order the employees to do their job:



## 4. Advantages

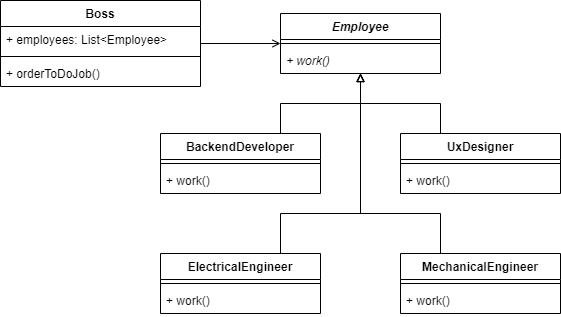
Micromanagement exists for a reason (even if it’s a bad one). There are bosses, who are control freaks and love to boss around (pun intended). In programming, we like to follow a different approach.

**When we want to control another class’s behavior completely, we have to know everything about it. Every little detail. That’s manageable for one or two external classes, but above that, things get messy. Furthermore, it makes it hard to extend the system with other classes.**

Let’s assume the company wants to expand and produce hardware. That would mean the boss had to know what an electrical engineer, a mechanical engineer, and other new roles do. (Which is even more challenging considering he doesn’t even know what current employees do. He only thinks he does know.)

In programming, that means we have to implement all new roles’ behavior in the *Boss* class. It would get out of hand pretty fast.

Instead, we use polymorphism. We implement the functionality in the new *ElectricalEngineer* and *MechanicalEngineer* classes:



**After that, the only thing we have to do is to hire them and add them to the *employee* list in the *Boss* class.**

## 5. Conclusion

In this short article, we saw what polymorphism is. With that knowledge, we can now revisit and interpret our definition: “[polymorphism is] a feature of a programming language that allows routines to use variables of different types at different times”.

A variable that has *Employee* type can contain any employee descendant. The only thing we know (and need) that we want them to work. Therefore, we call their *work()* method. We don’t know exactly how they will do it because it’ll be determined at runtime, depending on the exact type of employee the variable holds. But that’s enough for us because we have faith that all of our employees know how to do their job.