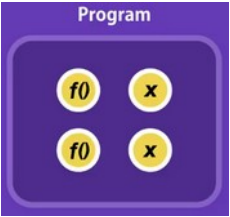
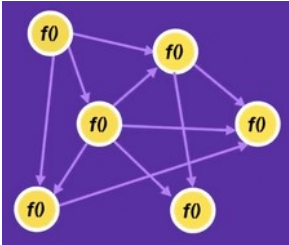
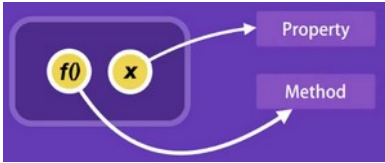
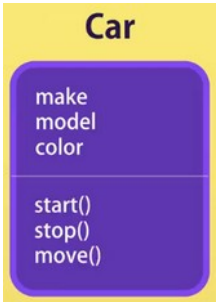




## Object-oriented programming in 7 minutes <sup>1</sup>

1. A popular interview question concerns the **four core concepts** in object-oriented programming. This concepts are: *encapsulation*, *abstraction*, *inheritance*, and *polymorphism*. Let's look at each of these concepts.
2. Before object-oriented programming, we had **procedural programming** that divided a program into a set of functions. So we have data stored in a bunch of variables and functions that operate on the data. This style of programming is very simple and straightforward. Often it's what you'd learn as part of your first programming subject at the university.
3. But as your programs grow, it will end up with a bunch of functions that are all over the place. You might find yourself copying and pasting lines of code over and over. You make a change to one function and then several other functions break: that's what we call "spaghetti code"! There is so much inter-dependency between all these functions it becomes problematic.
4. **Object-oriented programming** came to solve this problem. In object-oriented programming we combine a group of related variables and functions into a unit. We call that unit an *object*. We refer to these variables as *properties* and the functions as *methods*.
5. Here's an example. Think of a car: a car is an object with properties such as make, model, and color and methods like start, stop, and move. Now you might say: "But Mosh, we don't have cars in our programs, give me a real programming example." OK, think of the local storage object in your browsers. Every browser has a local storage object that allows you to store data locally. This local storage object has a property like length, which returns the number of objects in the storage, and methods like setItem and removeItem.
6. So, in object-oriented programming, we group related variables and functions that operate on them into objects, and this is what we call **encapsulation**.

Let me show you an example of this in action. So here we have three variables: baseSalary, overtime and rate. Below these, we have a function to calculate the wage for an employee. We refer to this kind of implementation as *procedural*. So we have variables on one side and functions on the other side: they are *decoupled*.

```
1 let baseSalary = 30_000;
2 let overtime = 10;
3 let rate = 20;
4
5 function getWage(baseSalary, overtime, rate) {
6   return baseSalary + (overtime * rate);
7 }
8
```

7. Now, let's take a look at the object-oriented way to solve this problem. We can have an employee object with three properties baseSalary, overtime and rate and a method called getWage. Now why is this better? Well, first of all, look at the getWage function: this function has **no parameters**. In contrast, in a procedural example, our getWage function has three parameters.

```
1 let employee = {
2   baseSalary: 30_000,
3   overtime: 10,
4   rate: 20,
5   getWage: function() {
6     return this.baseSalary + (this.overtime * this.rate);
7   }
8 };
9 employee.getWage();
```

<sup>1</sup> <https://www.youtube.com/watch?v=pTB0EiLXUC8>



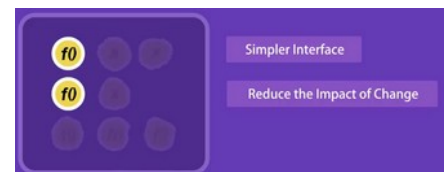
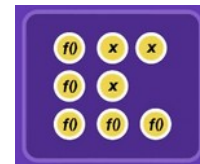
8. The reason in this implementation we don't have any parameters is because all these parameters are actually modeled as *properties* of this object. All these properties and the `getWage` function, they are highly related so they are part of one unit. So one of the symptoms of procedural code is functions with so many parameters.
9. When you write code in an object-oriented way, your functions end up having fewer and fewer parameters. As Uncle Bob says: "The best functions are those with no parameters!" The fewer the number of parameters, the easier it is to use and maintain that function.
10. So, that's encapsulation. Now, let's look at **abstraction**. Think of a DVD player as an object. This DVD player has a complex logic board on the inside and a few buttons on the outside that you interact with. You simply press the play button and you don't care what happens on the inside. All that complexity is hidden from you: this is abstraction in practice.
11. We can use the same technique in our objects. So we can hide some of the properties and methods from the outside, and this gives us a couple of **benefits**.

```
function getWage(baseSalary, overtime, rate) {
  return baseSalary + (overtime * rate);
}

let employee = {
  baseSalary: 30_000,
  overtime: 10,
  rate: 20,
  getWage: function() {
    return this.baseSalary + (this.overtime * this.rate);
  }
};
```



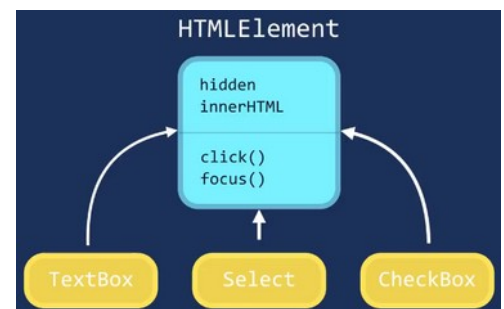
- First is that we'll make the interface of those objects simpler. Using and understanding an object with a few properties and methods is easier than an object with several properties and methods.
- The second benefit is that it helps us reduce the impact of change.



12. Let's imagine that tomorrow, we change these inner or private methods. None of these changes will leak to the outside, because we don't have any code that touches these methods outside of their containing object. We may delete a method or change its parameters, but none of these changes will impact the rest of the application's code. So with abstraction, we reduce the impact of change.
13. Now, the third core concept in object-oriented programming: **inheritance**. Inheritance is a mechanism that allows you to eliminate redundant code.

Here's an example. Think of HTML elements like text boxes, drop-down lists, check boxes and so on. All these elements have a few things in common: they should have properties like `hidden` and `innerHTML` and methods like `click` and `focus`.

14. Instead of redefining all these properties and methods for every type of HTML element, we can define them once in a generic object – call it `HTMLElement` – and have other objects inherit these properties and methods. So inheritance helps us eliminate redundant code.

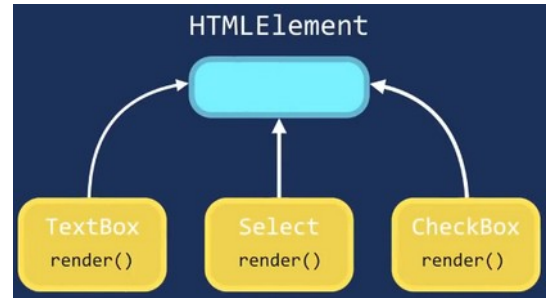


15. And finally: **polymorphism**. *Poly* means "many", *morph* means "form", so *polymorphism* means "many forms". In object-oriented programming, polymorphism is a technique that allows you to get rid of long *if-and-else* or *switch-and-case* statements. So, back to our HTML elements example, all these objects should have the ability to be rendered on a page. But the way each element is rendered is different from the others. If we want to render multiple HTML elements in a procedural way, our code would probably look like this.

```
switch (...) {
  case 'select': renderSelect();
  case 'text': renderTextBox();
  case 'checkbox': renderCheckBox();
  case ...
  case ...
  case ...
}
```



16. But with object orientation, we can implement a render method in each of these objects and the render method will behave differently depending on the type of the object you're referencing. So we can get rid of this nasty *switch-and-case* and use one line of code like this. You will see that later in the course.



17. So here are the **benefits** of object oriented programming:
- Using *encapsulation*, we group related variables and functions together, and this way we can reduce complexity. Now we can reuse these objects in different parts of a program, or in different programs.
  - With *abstraction*, we hide the details and the complexity and show only the essentials. This technique reduces complexity and also isolates the impact of changes in the code.
  - With *inheritance*, we can eliminate redundant code.
  - And with *polymorphism*, we can refactor ugly *switch/case* statements.
18. Well, hello! It's me, Mosh, again. I wanted to say thank you very much for watching this tutorial to the end. I hope you learned a lot! Please share and like this video to support me. If you want to learn more about the object-oriented programming, as I told you before, I have a **course** called "Object-oriented programming" in *JavaScript*. If you want to learn more, click on the link in the video description and enroll in the course. And if not, that's perfectly fine. Make sure to subscribe to my channel because I upload new videos every week. Thank you and have a great day!