Școala
informală
de IT

OOP

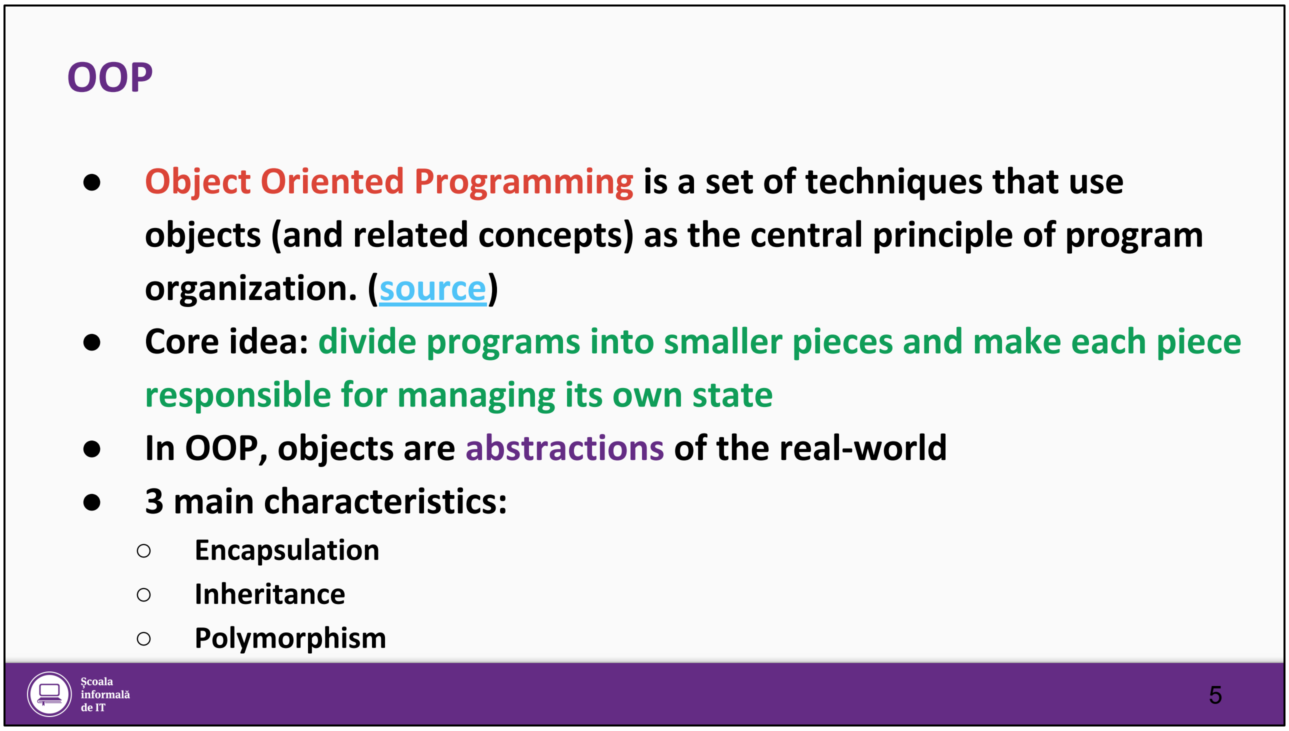
Recap

- An object has properties and methods.
- A class is a blueprint used to build a specific type of object.
- An object is an instance of a class.
- Prototypes are the mechanism by which objects inherit from one another.
- All functions have a prototype property (`User.prototype`)
- All objects have a prototype attribute (`Object.getPrototypeOf(userObj)`)
- When accessing an object property or method, it's searched at runtime on the object's prototype chain

Agenda

- What OOP is
- Encapsulation
- Inheritance
- Composition
- Polymorphism
- Static Properties and Methods
- Methods for Objects Creation

What OOP is



OOP

- Object Oriented Programming is a set of techniques that use objects (and related concepts) as the central principle of program organization. (source)
- Core idea: divide programs into smaller pieces and make each piece responsible for managing its own state
- In OOP, objects are abstractions of the real-world
- 3 main characteristics:
  - Encapsulation
  - Inheritance
  - Polymorphism

Școala informală de IT

(Although no one agrees on a definition) OOP is a programming paradigm that uses **objects** as the central principle of program organization.

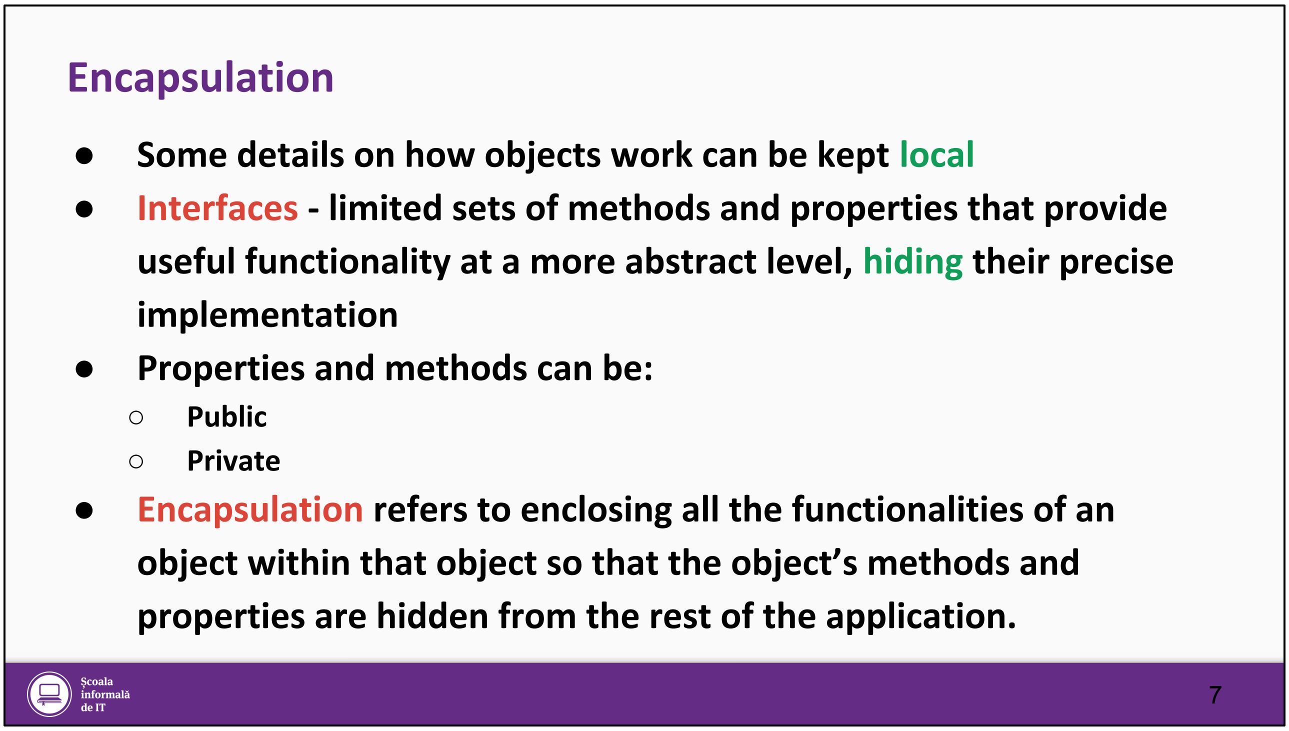
In OOP, the program is abstracted into objects and classes. In plain English:

- The main entities of the program are identified, based on the specifications. For instance, if we're building an ecommerce application, we will have Users and Products as some of the most important entities. They'll become classes in our program, and we'll instantiate objects of those types in our program.
- Although both users and products have a huge number of characteristics, we only keep in our program the ones that are relevant for the application that we're building. For instance, for a user we don't need to know the medical history, but it's good to know the address where to deliver the products.
- This process, of only storing what is relevant for the program that we're building, is called **abstraction**.

The core idea of object oriented programming is to divide programs into smaller pieces (objects and classes) and to make each piece as responsible for managing its own state. This is done via the object properties - their values represent the object state.

There are 3 main characteristics of OOP that we'll talk about next: Encapsulation, Inheritance and Polymorphism.

Encapsulation



Encapsulation

- Some details on how objects work can be kept local
- Interfaces - limited sets of methods and properties that provide useful functionality at a more abstract level, hiding their precise implementation
- Properties and methods can be:
  - Public
  - Private
- Encapsulation refers to enclosing all the functionalities of an object within that object so that the object's methods and properties are hidden from the rest of the application.

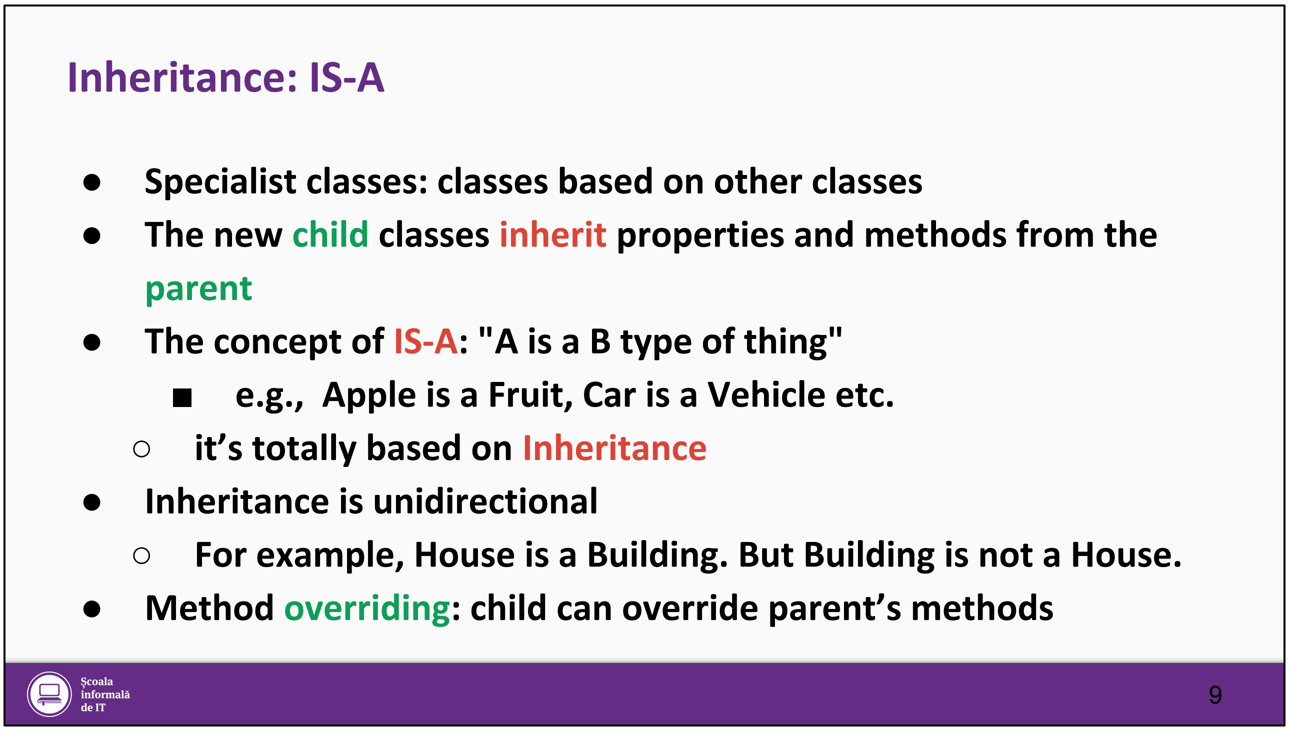
Going back to the core idea of OOP: to divide the program into smaller pieces and to make each one responsible for its own state, and more specifically to the last part, this means that some knowledge on how the piece of program works can be kept internally in that object (locally). For a car, how the engine is started and how it's turned off can be kept internal - no other part of the program cares about the details of that, they only want to be able to start or stop the engine and to know if the engine is started or not.

In a more formal way, *for an object it's important to know* ***what*** *another object does,* ***not how*** *it does it*. This is achieved via **interfaces**.

The properties and methods that are part of the interface are called **public**. The others, which outside code should not be touching, are called **private**. JavaScript does not have (yet) a standard way of having private properties and methods, but the programmers are using the concept successfully. There are several ways that can be used to achieve this, the simplest being to describe the available interface in comments and documentation, and adding an underscore (_) before private properties and methods.

Encapsulation can be defined as the separation between the interface and the implementation.

Inheritance



Inheritance: IS-A

- Specialist classes: classes based on other classes
- The new child classes inherit properties and methods from the parent
- The concept of IS-A: "A is a B type of thing"
    - e.g., Apple is a Fruit, Car is a Vehicle etc.
  - it's totally based on Inheritance
- Inheritance is unidirectional
  - For example, House is a Building. But Building is not a House.
- Method overriding: child can override parent's methods

Școala informală de IT

In OOP, we can create new classes based on other classes — these new child classes can be made to inherit the data and code features of their parent class, so you can reuse functionality common to all the object types rather than having to duplicate it. Where functionality differs between classes, you can define specialized features directly on them as needed.

Let's implement "Motorcycle is a Vehicle". First we need to define the "Vehicle" Class

```
function Vehicle(){
  this.wheels = 4;
  this.fuel = "Gas";
}

Vehicle.prototype.startEngine = function(){ console.log("starting vehicle engine");
}
Vehicle.prototype.drive = function(){ console.log("driving some random vehicle"); }
```

... nothing fancy for now. Let's implement "Motorcycle" class that extends Vehicle:

```
function Motorcycle() {
  Vehicle.call(this);
  this.wheels = 2;
}

Motorcycle.prototype = Object.create(Vehicle.prototype);
Motorcycle.prototype.drive = function() { console.log("driving a motorcycle"); }
```

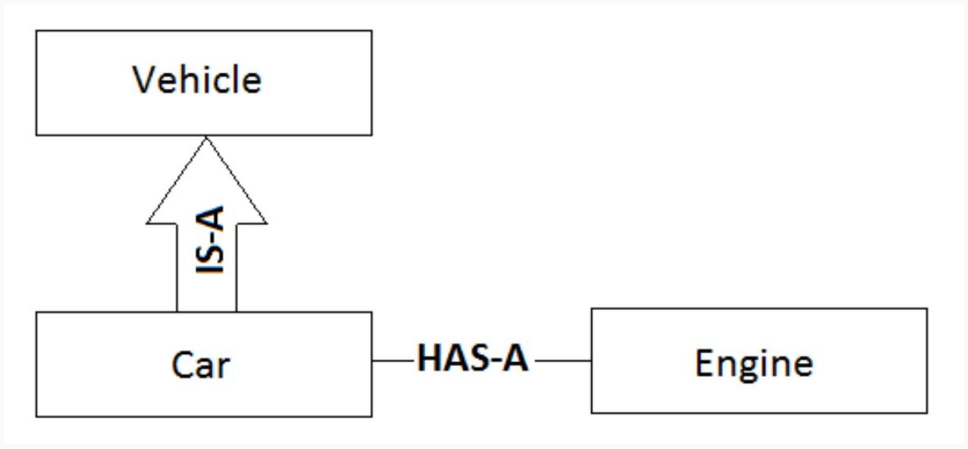
We used two concepts that we didn't study before: Vehicle.call and Object.create.

We’ll talk about them a little later.

When implementing inheritance, the class that inherits another class can override its methods. Take the motorcycle example where we are overriding the “drive” method.

Composition

Composition: HAS-A



Besides the **IS-A** relationship, another important relationship in OOP is **HAS-A**.

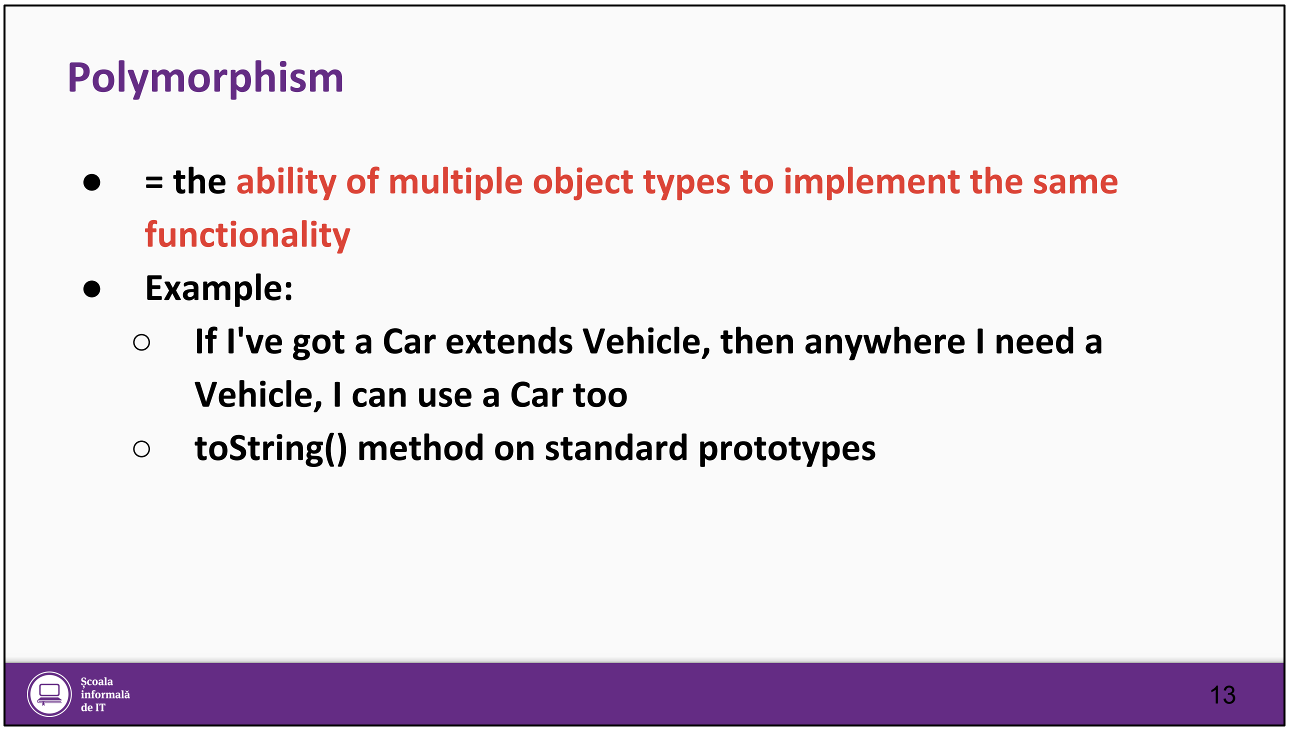
This simply means the use of instance variables that are references to other objects. For example Car has Engine, or House has Bathroom.

The HAS-A relationship is used to define **composition**.

Whereas encapsulation can be used to **separate** pieces of code from each other, reducing the tangledness of the overall program, inheritance fundamentally **ties** classes together, creating more tangle. When inheriting from a class, you usually have to know more about how it works than when simply using it.

Prefer composition over inheritance.

Polymorphism



Polymorphism

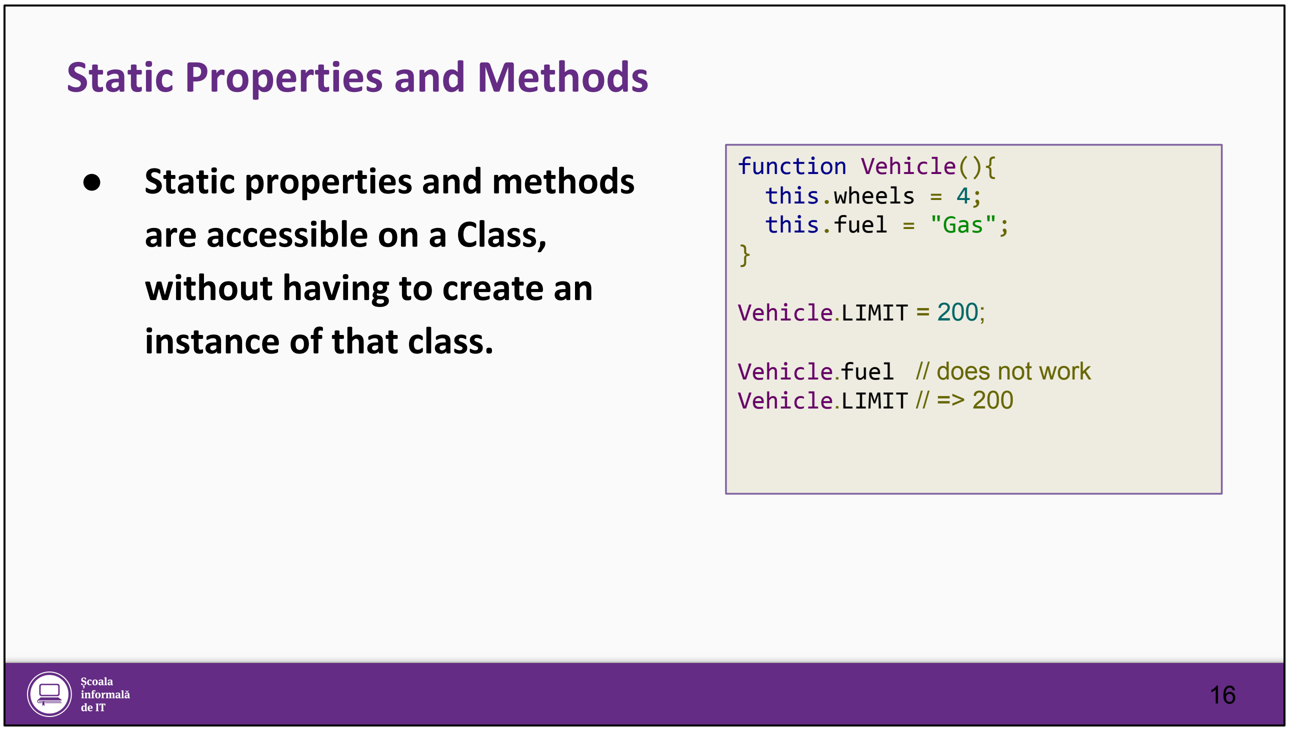
- = the ability of multiple object types to implement the same functionality
- Example:
  - If I've got a Car extends Vehicle, then anywhere I need a Vehicle, I can use a Car too
  - toString() method on standard prototypes

Şcoala informală de IT

When you call the String function (which converts a value to a string) on an object, it will call the toString method on that object to try to create a meaningful string from it. Some of the standard prototypes define their own version of toString so they can create a string that contains more useful information than "[object Object]". You can also do that yourself.

This is a simple instance of a powerful idea. When a piece of code is written to work with objects that have a certain interface—in this case, a toString method—any kind of object that happens to support this interface can be plugged into the code, and it will just work. This technique is called polymorphism.

Static Properties and Methods



Static Properties and Methods

- Static properties and methods are accessible on a Class, without having to create an instance of that class.

```
function Vehicle(){
  this.wheels = 4;
  this.fuel = "Gas";
}

Vehicle.LIMIT = 200;

Vehicle.fuel   // does not work
Vehicle.LIMIT // => 200
```

Școala informală de IT

Static variables and methods are accessible on a Class, without having to create an object with that class.

So on the Vehicle example we are not able to use:

```
var vehicleFuel = Vehicle.fuel; // This will not work!!
```

... but we have to do something like this:

```
var vehicle = new Vehicle();
var vehicleFuel = vehicle.fuel;
```

Let's add a static variable on our Vehicle class

```
function Vehicle(){
  this.wheels = 4;
  this.fuel = "Gas";
}

Vehicle.LIMIT = 200;
Vehicle.prototype.startEngine = function(){
  console.log("starting vehicle engine");
}

Vehicle.prototype.drive = function(){
  console.log("driving some random vehicle");
}
```

Now we're able to call:

```
Vehicle.LIMIT //200
```

Methods for Objects Creation

Method for Objects Creation

- Object.create
  - creates a new object with the specified prototype object and properties

```
Car.prototype =
Object.create(Vehicle.prototype);
```

- Call / Apply functions
  - = special functions that are helping us calling a function
  - a regular function called using () is running it it’s own scope
  - with call/apply functions we can call a function with a specific context (“this”)

```
function.call(thisArg, arg1, arg2, ...)
function.apply(thisArg, [argsArray])
```

Școala informală de IT

Resources

https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Objects/Object-oriented_JS

http://www.objectplayground.com/

https://medium.com/@kevincennis/prototypal-inheritance-781bccc97edb