OOP 1st lecture

Certainly! In programming, there are different ways or styles to write code, and these are called programming paradigms. One popular paradigm is called Object-Oriented Programming (OOP). OOP is all about organizing our programs using objects, which group together related data and functionality.

To understand the difference between OOP and another paradigm called Functional Programming, let's take an example. Suppose we need to write code to calculate the total cost of buying a pair of shoes, including tax.

In Functional Programming, we would separate the data (shoes price and tax) from the function that calculates the total price. We would define variables for the data and a function to perform the calculation.

In OOP, we would create an object to represent the purchase. This object would contain variables for the shoes price and tax, as well as a method (a function inside an object) to calculate the total price. We can then access the method on the object and get the result.

The advantage of using OOP is that we can create multiple objects, each representing a different purchase. We can reuse the same method for all objects, making our code more efficient. We can also use the "this" keyword to refer to the current object, which makes our code more flexible and easier to understand.

In summary, OOP is a way of organizing code using objects that contain both data and functionality. It allows for code reuse, flexibility, and efficient programming.

04:38 AM

………..

Coding of this Video

Certainly! Here's an example code that demonstrates the concepts discussed in the video:

// Functional Programming Approach

function calculateTotalPrice(shoes, tax) {

return shoes \* tax;

}

const shoesPrice = 100;

const stateTax = 1.2;

const totalPrice = calculateTotalPrice(shoesPrice, stateTax);

console.log(totalPrice); // Output: 120

// Object-Oriented Programming Approach

const purchase1 = {

shoes: 100,

stateTax: 1.2,

totalPrice: function() {

return this.shoes \* this.stateTax;

}

};

console.log(purchase1.totalPrice()); // Output: 120

const purchase2 = {

shoes: 50,

stateTax: 1.2,

totalPrice: purchase1.totalPrice // Reusing the same method

};

console.log(purchase2.totalPrice()); // Output: 60

In the functional programming approach, we define a function calculateTotalPrice that takes in the shoes price and tax as parameters and returns the total price. We then call this function with the appropriate values and log the result.

In the object-oriented programming approach, we create objects purchase1 and purchase2 that have properties for shoes price and state tax, as well as a method totalPrice that calculates the total price using the this keyword to refer to the current object. We can then access and invoke the totalPrice method on each object to get the respective total prices.

04:47 AM

………………

Classes

Certainly! In programming, there are times when you need to create many objects with the same properties and methods. For example, if you're making a car racing game, you might need to create hundreds of car objects. To do this efficiently, you can use something called classes.

A class is like a blueprint that you can use to create new objects of a certain kind. In JavaScript, you define a class using the class keyword, followed by the name of the class starting with a capital letter. Inside the class, you have a special function called the constructor. The constructor function is used to assign values to the properties of the objects created from the class.

Once you have defined the class and its constructor, you can add as many methods as you want. These methods are functions that can perform actions or calculations using the data stored in the object.

To create a new object from the class, you use the new keyword followed by the class name and any necessary arguments for the constructor. This new object is called an instance of the class.

Once you have an instance of the class, you can access its methods and properties using dot notation. For example, if you have a car object called carOne, you can access its methods like carOne.turboOn().

I hope this explanation helps! Let me know if you have any further questions.

04:56 AM

Questions

Certainly! Here are some practice questions based on the video content:

1. What is the purpose of using classes in programming?
2. How do you define a class in JavaScript?
3. What is the role of the constructor function in a class?
4. How do you instantiate an object from a class?
5. How do you access methods and properties of an object in JavaScript?

Feel free to answer these questions and let me know if you have any further questions or need more clarification!

04:57 AM

………………………

Reading

**Object Oriented Programming principles**

In this reading, you'll learn about the benefits of object-oriented programming (OOP) and the OOP principles.

**The Benefits of OOP**

There are many benefits to using the object-oriented programming (OOP) paradigm.

OOP helps developers to mimic the relationship between objects in the real world. In a way, it helps you to reason about relationships between things in your software, just like you would in the real world. Thus, OOP is an effective approach to come up with solutions in the code you write. OOP also:

* Allows you to write modular code,
* Makes your code more flexible and
* Makes your code reusable.

**The Principles of OOP**

The four fundamental OOP principles are inheritance, encapsulation, abstraction and polymorphism. You'll learn about each of these principles in turn. The thing to remember about Objects is that they exist in a hierarchal structure. Meaning that the original base or super class for everything is the Object class, all objects derive from this class. This allows us to utilize the Object.create() method. to create or instansiate objects of our classes.

1

2

3

4

5





A more common method of creating obbjects from classes is to use the *new* keyword. When using a default or empty constructor method, JavaScript implicitly calls the Object superclass to create the instance.

1

2

3

4

5





This concept is explored within the next section on inheritance

**OOP Principles: Inheritance**

Inheritance is one of the foundations of object-oriented programming.

In essence, it's a very simple concept. It works like this:

1. There is a base class of a "thing".
2. There is one or more sub-classes of "things" that inherit the properties of the base class (sometimes also referred to as the "super-class")
3. There might be some other sub-sub-classes of "things" that inherit from those classes in point 2.

Note that each sub-class inherits from its super-class. In turn, a sub-class might also be a super-class, if there are classes inheriting from that sub-class.

All of this might sound a bit "computer-sciency", so here's a more practical example:

1. There is a base class of "Animal".
2. There is another class, a sub-class inheriting from "Animal", and the name of this class is "Bird".
3. Next, there is another class, inheriting from "Bird", and this class is "Eagle".

Thus, in the above example, I'm modelling objects from the real world by constructing relationships between Animal, Bird, and Eagle. Each of them are separate classes, meaning, each of them are separate blueprints for specific object instances that can be constructed as needed.

To setup the inheritance relation between classes in JavaScript, I can use the *extends* keyword, as in *class B extends A*.

Here's an example of an inheritance hierarchy in JavaScript:

1

2

3





**OOP Principles: Encapsulation**

In the simplest terms, encapsulation has to do with making a code implementation "hidden" from other users, in the sense that they don't have to know how my code works in order to "consume" the code.

For example, when I run the following code:

1





I don't really need to worry or even waste time thinking about how the *toUpperCase()* method works. All I want is to use it, since I know it's available to me. Even if the underlying syntax - that is, the implementation of the *toUpperCase()* method changes - as long as it doesn't break my code, I don't have to worry about what it does in the background, or even how it does it.

**OOP Principles: Abstraction**

Abstraction is all about writing code in a way that will make it more generalized.

The concepts of encapsulation and abstraction are often misunderstood because their differences can feel blurry.

It helps to think of it in the following terms:

* An abstraction is about extracting the *concept* of what you're trying to do, rather than dealing with a specific manifestation of that concept.
* Encapsulation is about you not having access to, or not being concerned with, how some implementation works internally.

While both the encapsulation and abstraction are important concepts in OOP, it requires more experience with programming in general to really delve into this topic.

For now, it's enough to be aware of their existence in OOP.

**OOP Principles: Polymorphism**

Polymorphism is a word derived from the Greek language meaning "multiple forms". An alternative translation might be: "something that can take on many shapes".

So, to understand what polymorphism is about, let's consider some real-life objects.

* A door has a bell. It could be said that the bell is a property of the door object. This bell can be rung. When would someone ring a bell on the door? Obviously, to get someone to show up at the door.
* Now consider a bell on a bicycle. A bicycle has a bell. It could be said that the bell is a property of the bicycle object. This bell could also be rung. However, the reason, the intention, and the result of somebody ringing the bell on a bicycle is not the same as ringing the bell on a door.

The above concepts can be coded in JavaScript as follows:

10





So, I can access the *bell()* method on the *bicycle* object, using the following syntax:

1





I can also access the *bell()* method on the *door* object, using this syntax:

1





At this point, one can conclude that the exact same name of the method can have the exact opposite intent, based on what object it is used for.

Now, to make this code truly polymorphic, I will add another function declaration:

1

2

3





Now I have declared a *ringTheBell()* function. It accepts a *thing* parameter - which I expect to be an object, namely, either the *bicycle* object or the *door* object.

So now, if I call the *ringTheBell()* function and pass it the *bicycle* as its single argument, here's the output:

1





However, if I invoke the *ringTheBell()* function and pass it the *door* object, I'll get the following output:

1





You've now seen an example of the exact same function producing different results, **based on the context** in which it is used.

Here's another example,the concatenation operator, used by calling the built-in *concat()* method.

If I use the *concat()* method on two strings, it behaves exactly the same as if I used the *+* operator.

1





I can also use the *concat()* method on two arrays. Here's the result:

1





Consider using the *+* operator on two arrays with one member each:

1

2





This means that the *concat()* method is exhibiting polymorphic behavior since it behaves differently based on the context - in this case, based on what data types I give it.

To reiterate, polymorphism is useful because it allows developers to build objects that can have the exact same functionality, namely, functions with the exact same name, which behave exactly the same. However, at the same time, you can override some parts of the shared functionality or even the complete functionality, in some other parts of the OOP structure.

Here's an example of polymorphism using classes in JavaScript:

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21





The *Penguin* and *Eagle* sub-classes both inherit from the *Bird* super-class. The *Eagle* sub-class inherits the *useWings()* method from the *Bird* class, but extends it with an additional console log. The *Penguin* sub-class doesn't inherit the *useWings()* class - instead, it has its own implementation, although the *Penguin* class itself does extend the *Bird* class.

Do some practice with the above code, trycreating some of your own classes. (hint : think about things you know from everyday life)

1





Summary

…………….

Of course! I'd be happy to explain the topic of object-oriented programming (OOP) in simple terms. Here's a simplified explanation:

Object-oriented programming (OOP) is a way of organizing and structuring code to make it more efficient and reusable. It helps developers mimic real-world relationships between objects in their software.

Here are the benefits of using OOP:

* Modularity: OOP allows you to write code in smaller, self-contained modules, making it easier to understand and maintain.
* Flexibility: OOP makes your code more flexible, allowing you to easily modify or add new features without affecting the rest of the code.
* Reusability: OOP promotes code reuse, meaning you can use existing code to create new objects or classes, saving time and effort.

The four fundamental principles of OOP are:

1. Inheritance: Inheritance allows classes to inherit properties and methods from other classes, creating a hierarchical structure.
2. Encapsulation: Encapsulation hides the implementation details of a class, allowing users to interact with the class without needing to know how it works internally.
3. Abstraction: Abstraction involves writing code in a generalized way, focusing on the concept rather than the specific implementation.
4. Polymorphism: Polymorphism allows objects to take on different forms or behaviors based on the context in which they are used.

I hope this explanation helps! Let me know if you have any further questions.

05:01 AM

# Constructors

JavaScript has a number of built-in object types, such as:

 Math, Date, Object, Function, Boolean, Symbol, Array, Map, Set, Promise, JSON, etc.

These objects are sometimes referred to as "native objects".

Constructor functions, commonly referred to as just "constructors", are special functions that allow us to build instances of these built-in native objects. All the constructors are capitalized.

To use a constructor function, I must prepend it with the operator new.

For example, to create a new instance of the Date object, I can run: new Date(). What I get back is the current datetime, such as:

Thu Feb 03 2022 11:24:08 GMT+0100 (Central European Standard Time)

However, not all the built-in objects come with a constructor function. An example of such an object type is the built-in Math object.

Running new Math() throws an Uncaught TypeError, informing us that Math is not a constructor.

Thus, I can conclude that some built-in objects do have constructors, when they serve a particular purpose: to allow us to instantiate a specific instance of a given object's constructor. The built-in Date object is perfectly suited for having a constructor because each new date object instance I build should have unique data by definition, since it's going to be a different timestamp - it's going to be built at a different moment in time.

Other built-in objects that don't have constructors, such as the Math object, don't need a constructor. They're just static objects whose properties and methods can be accessed directly, from the built-in object itself. In other words, there is no point in building an instance of the built-in Math object to be able to use its functionality.

For example, if I want to use the pow method of the Math object to calculate exponential values, there's no need to build an instance of the Math object to do so. For example, to get the number 2 to the power of 5, I'd run:

Math.pow(2,5); // --> 32

There's no need to build an instance of the Math object since there would be nothing that needs to be stored in that specific object's instance.

Besides constructor functions for the built-in objects, I can also define custom constructor functions.

Here's an example:

1

2

3

4

5

6





No Output

Now I can make as many icecreams as I want:

1

2

3

4

5

6

7

8

9

10

11





I've just built two instance objects of Icecream type.

The most common use case of new is to use it with one of the built-in object types.

Note that using constructor functions on all built-in objects is sometimes not the best approach.

This is especially true for object constructors of primitive types, namely: String, Number, and Boolean.

For example, using the built-in String constructor, I can build new strings:

2

1





The apple variable is an object of type String.

Let's see how the apple object differs from the following pear variable:

1

2





The pear variable is a string literal, that is, a primitive Javascript value.

The pear variable, being a primitive value, will always be more performant than the apple variable, which is an object.

Besides being more performant, due to the fact that each object in JavaScript is unique, you can't compare a String object with another String object, even when their values are identical.

In other words, if you compare new String('plum') === new String('plum'), you'll get back false, while "plum" === "plum" returns true. You're getting the false when comparing objects because it is not the values that you pass to the constructor that are being compared, but rather the memory location where objects are saved.

Besides not using constructors to build object versions of primitives, you are better off not using constructors when constructing plain, regular objects.

Instead of new Object, you should stick to the object literal syntax: {}.

A RegExp object is another built-in object in JavaScript. It's used to **pattern-match strings** using what's known as "Regular Expressions". Regular Expressions exist in many languages, not just JavaScript.

In JavaScript, you can built an instance of the RegExp constructor using new RegExp.

Alternatively, you can use a pattern literal instead of RegExp. Here's an example of using /d/ as a pattern literal, passed-in as an argument to the match method on a string.

1

2





Instead of using Array, Function, and RegExp constructors, you should use their array literal, function literal, and pattern literal varieties: [], () {}, and /()/.

However, when building objects of other built-in types, we can use the constructor.

Here are a few examples:

1

2

3

4

5

6

7





The above list is inconclusive, but it's just there to give you an idea of some constructor functions you can surely use.

Note that there are links provided about RegExp and regular expression in the lesson item titled "Additional Reading".

…………………………………………………………………………………………………………………………………………………………………..

In this video, you will learn about inheritance in JavaScript and how it revolves around the concept of a prototype. The prototype is an object that can hold properties to be shared by multiple other objects. The video demonstrates how to create objects using the Object.create() method and how these objects inherit properties from their prototype. It also explains how properties can be overridden in the child objects without affecting the prototype or other objects. Overall, the video provides an introduction to inheritance in JavaScript and how it can be implemented using prototypes.

12:20 AM