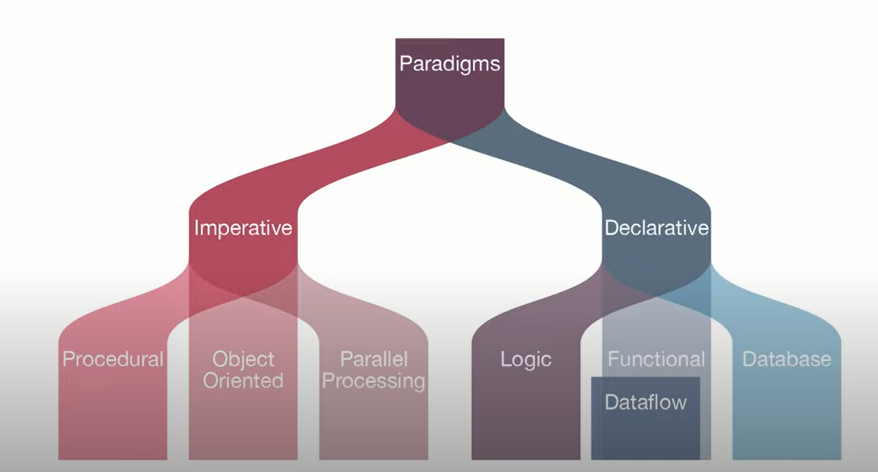
**Paradigmas e Pilares**

A programação possui dois paradigmas: o **imperativo** e o **declarativo:**



O **imperativo** foca sempre em como você vai resolver os problemas.

O **declarativo** foca no que você vai fazer para resolver os problemas.

Um exemplo de declarativo seria um banco de dados em SQL e programação funcional onde apenas declaramos as coisas e elas acontecem (diferente da programação orientada a objetos onde precisamos dizer como as coisas serão chamadas, a ordem etc.)

Então os paradigmas basicamente são essas diferentes formas para programar.

Os programas são “objetos” que possuem uma série de propriedades:



Ex: eu posso ter o objeto “pessoa” e ela tem várias propriedades, tais como nome, altura, cidade, etc.

Agora vamos ver os 4 pilares para a Orientação à Objetos:

1. Herança

São objetos filhos que herdam propriedades e métodos do objeto pai (objetos ascendentes à ele), mas os objetos filhos também possuem suas próprias características.

1. Polimorfismo

Objetos podem herdar a mesma classe do pai porém podem se comportar de forma diferente quando invocam seus métodos. Por ex:

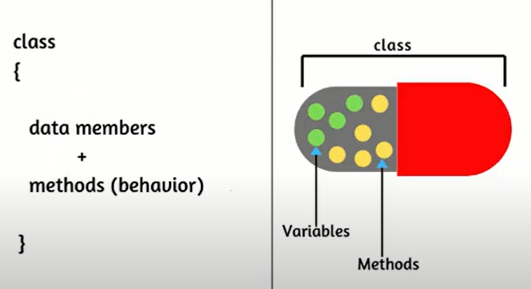


Na imagem temos três animais, a **classe pai** é “**animal”**, e tem o **método “falar”** só que quando o pato, o cachorro e o gato falam, eles falam de formas diferentes, e isso é o **polimorfismo**.

O polimorfismo possui um pouco de abstração, é como se o elemento filho abstraísse algumas heranças do pai e se tornasse um pouco diferente dele.

1. Encapsulamento

Cada classe tem propriedades e métodos independentes do restante do código. Ex. uma propriedade pertence a uma classe e se a propriedade mudar de valor vai alterar apenas a classe não influenciando no restante do código.



1. Abstração

A definição, de uma forma geral, para abstração é “processo mental que consiste em isolar um aspecto determinado de um estado de coisas relativamente complexo, a fim de simplificar a sua avaliação, classificação ou para permitir a comunicação do mesmo”.

Ou seja, temos um problema complexo e a abstração serve para torná-lo mais simples. Por exemplo, temos um carro que é um objeto complexo (cheio de propriedades, etc) mas se abstrairmos vamos dizer que ele é um veículo de rodas, se abstrairmos mais ainda vamos ver que ele é apenas um veículo.

Então abstraindo estamos generalizando algo, e quando especificamos estamos complexando o objeto. Por exemplo, o barco também é um veículo, porém não é um veículo de rodas como o carro, temos a bicicleta que é também um veículo de rodas porém diferente do carro.

**Protótipos e Classes**

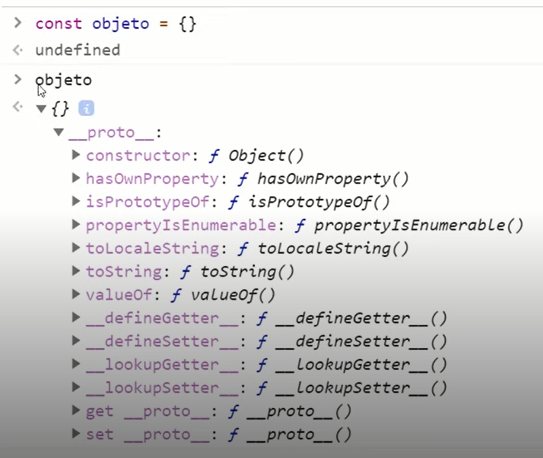
Protótipos:

Vamos falar de OOJS (Object Oriented JavaScript) ou JavaScript Orientado a Objetos.

Os Protótipos são basicamente um esqueleto de todos os objetos. Todos os objetos do javascript vão herdar propriedades e métodos de um *prototype* (que é um protótipo).

O objeto **Object.prototype** está no topo dessa cadeia.

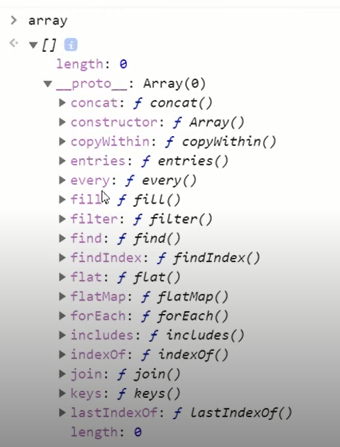
Sempre que tiver uma constante em javascript ou um objeto que é do tipo não primitivo, ou seja, do tipo complexo ou composto, ele terá essa propriedade **\_proto\_** veja abaixo:



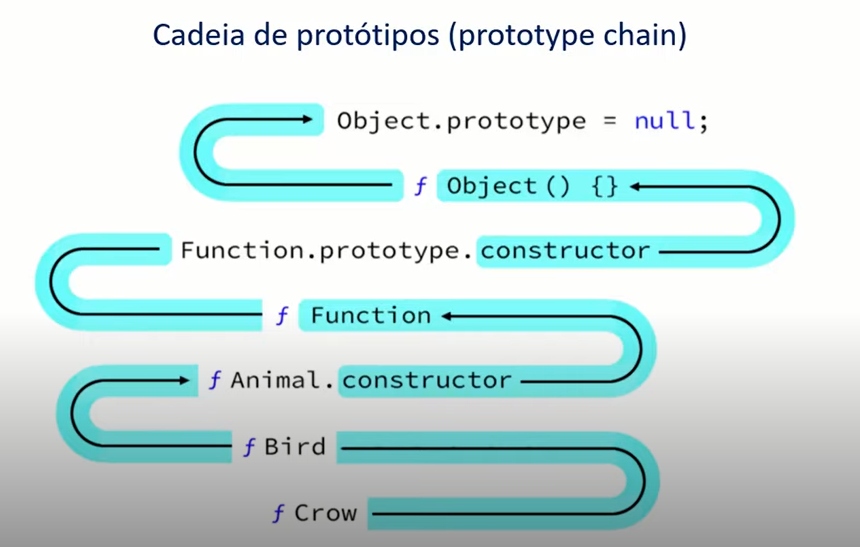
A propriedade **proto** (prototype) tem também uma série de propriedades. Então conseguimos utilizar por exemplo **objeto.hasOwnProperty** para ver se o objeto tem alguma chave com aquele nome por causa do prototype, pois isso está na classe pai dele. Outro exemplo: podemos converter um objeto para **toString** pois sua classe pai (a classe prototype) tem esse método!

Então, mesmo que a constante **objeto** (declarada na imagem acima) não tenha uma função toString, conseguimos acessar isso, pois seu pai (seu protótipo) possui esse método listado.

Abaixo temos uma lista com todos os métodos que conseguimos acessar do array, tais como **map,** **reduce,**  **join,** etc. O array herda os métodos do protótipo do objeto array que é o que define objetos array em javascript.



Abaixo temos uma figura como exemplo da cadeia de protótipos:



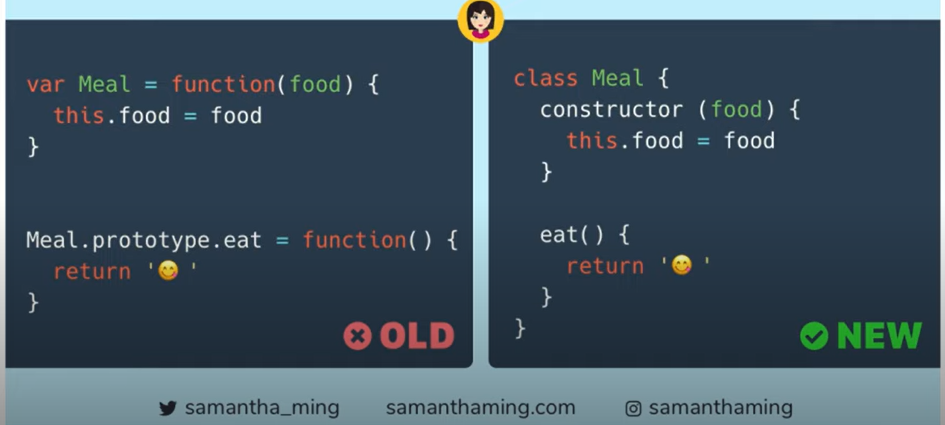
Temos o corvo (crow) que herda de pássaro (bird), que herda de animal, que herda da **Function prototype** que herdaria também do **Object.prototype**. A cadeia vai sempre procurar pelas funções (ou métodos) dentro de todos os objetos “pai” até chegar no **nul**

Classes:

Classes no javascript não existem nativamente, elas são **syntatic sugar** (açúcar sintático) que nada mais é do que uma sintaxe feita para facilitar a escrita.

Mas sempre utilizamos objetos e objetos que tem protótipos, esse é o tipo de herança que existe em javascript nativamente (figura da esquerda).

Depois da atualização do javascript (depois do ECMAScript 6) se tornou possível fazer essa sintaxe de classe mais parecida com outras linguagens (figura da direita) que são feitas para acomodar o paradigma de orientação a objetos como por exemplo o JAVA.



Então, apesar de chamar de **classe, na verdade** ela é um objeto que também dará acesso ao protótipo herdando os métodos deste objeto.

Aqui temos um print de uma classe chamada **Animal** e da filha dela chamada **Cat** e elas têm a anatomia de uma classe em javascript, que sempre terá:

* um ***constructor*** que constrói a classe e quando passamos parâmetros ele vai atribuir esses parâmetros a certos valores dentro dessa classe. No exemplo foi passado o parâmetro “type = animal”, então, caso não tenha tipo definido, será animal, mas caso tenha um tipo definido será esse outro tipo.
* para ter acesso a propriedades de um objeto, utilizamos ***get*** e ***set***. Com o get estamos pegando o tipo do objeto e com set estamos definindo que tipo tem um outro valor.
* já na classe filha Cat, que foi determinada como filha usando **class Cat extends Animal**, temos um método chamado **super (importante para fazer herança)** que manda pra cima (pra função pai) os parâmetros que estão ali. Quando falo **super(‘cat’)** como na figura, estou construindo o cat e no momento que ele for construído o tipo não será mais **animal,** será **cat**. Então o super serve para utilizar propriedades do construtor que existem na classe pai.
* podemos sobrescrever um método, por ex o **makeSound** está na classe pai, mas utilizei ele na classe filha, trazendo um comportamento diferente do pai

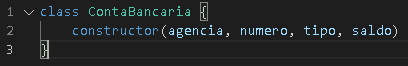


**Atividade Prática**

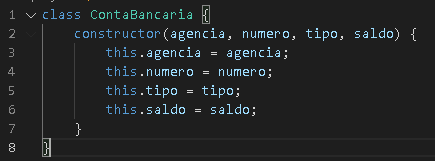
Nesta atividade, vamos testar os conceitos de Orientação a Objetos simulando a criação de diversos tipos de contas bancárias e operações disponíveis em cada uma.

1. Crie a classe ContaBancaria, que possui os parâmetros agencia, numero, tipo e saldo;

Sempre que formos colocar os parâmetros da classe (o que vamos passar para esta classe), quando formos declarar uma nova instância dela, criar uma nova conta bancária por exemplo, tudo que a gente vai mandar no momento da criação a gente precisa colocar dentro do construtor. Então no nosso construtor vamos declarar: agência, número, tipo e saldo, pois nossa conta terá essas informações:



Agora vamos defini-los:

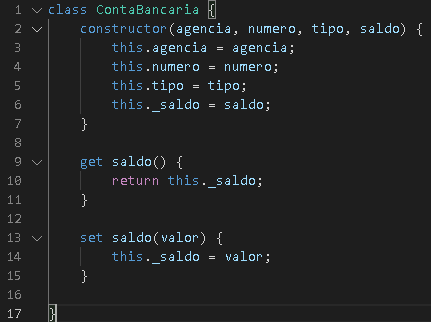


O this sempre se refere ao objeto, que neste caso é a ContaBancaria.

1. Dentro de ContaBancaria, construa o getter e o setter de saldo;

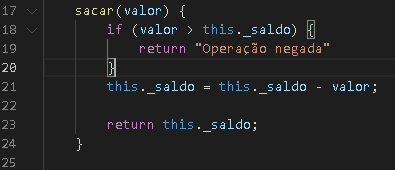
ContaBancaria será a classe mais abstrata que teremos, depois vamos criar outros tipos de contas. Agora vamos manipular **saldo**.

Sempre que tivermos **getter** e **setter** colocamos um underline na frente do nome da propriedade (**this.\_saldo**), assim a nossa função **get e set** terá nome **saldo** (veja abaixo **get saldo()** e **set saldo()**):

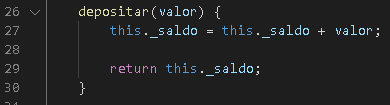


1. Dentro de ContaBancaria, crie os métodos sacar e depositar;

Se o valor for maior que o saldo não será feita a operação. Precisamos atualizar o saldo também, diminuindo dele o valor sacado, caso o saque aconteça, e depois retornamos o valor do saldo atualizado:



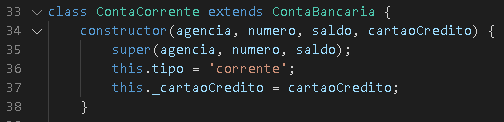
Criando o método depositar:



1. Crie uma classe-filha chamada ContaCorrente que herda todos esses parâmetros e ainda possua o parâmetro cartaoCredito;

Agora fechamos a classe ContaBancaria e vamos criar a filha fora dela, claro.

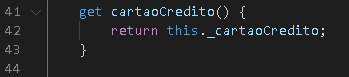
Vamos copiar todos os parâmetros da ContaBancaria, exceto tipo (mais pra frente entenderemos porque) e vamos usar o **super** para mandar tudo isso para a classe pai que vai setar para esta conta corrente todas essas propriedades:



O tipo é conta corrente, então definimos aqui nosso tipo usando this e também teremos a propriedade cartão de crédito nesse tipo de conta.

1. Ainda em ContaCorrente, construa o getter e o setter de cartaoCredito;

Já que vamos usar get e set, foi preciso mudar no código acima o código para **this.\_cartaoCredito** para ficar com nomes diferentes!



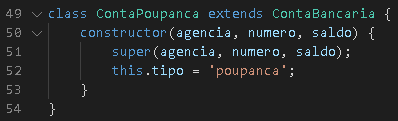
O valor para nosso cartao de credito será false ou true, ou seja, se ela tem cartão de crédito ou não:



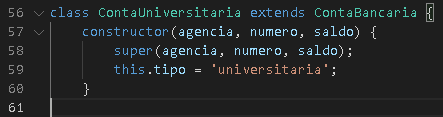
1. Ainda em ContaCorrente, faça com que o tipo seja 'conta corrente' por padrão;

Já fizemos isso quando definimos o tipo da conta corrente.

1. Crie uma classe-filha chamada ContaPoupanca que herda todos os parâmetros de ContaBancaria;

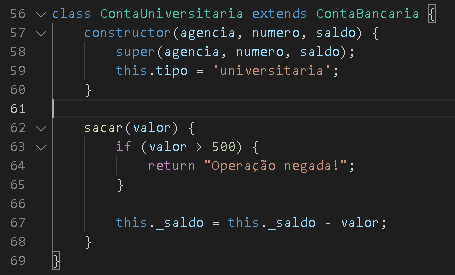


1. Crie uma classe-filha chamada ContaUniversitaria que herda todos os parâmetros de ContaBancaria;

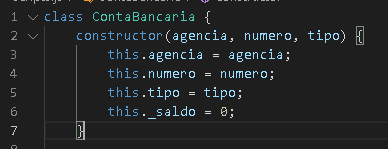


1. Faça com que o método saque de ContaUniversitaria apenas seja capaz de sacar valores menores que 500 reais.

Aqui será preciso sobrescrever o método sacar (alterar o “original” que foi feito anteriormente. Ele vai receber um valor também só que vamos colocar uma condicional para caso ele seja maior que 500 a operação seja negada.

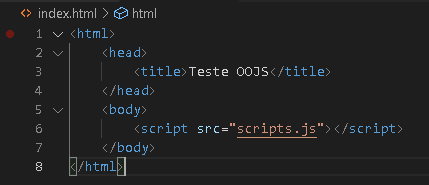


Stephany achou melhor retirar a propriedade **saldo** de tudo, pois quando abrimos uma conta não temos saldo, ele é igual a zero, então vamos só colocar ele em Contabancaria com valor igual a zero:

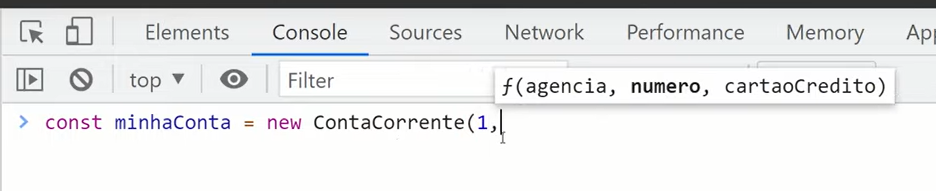


Nosso código javascript está pronto, agora só precisamos colocar ele no nosso HTML para poder criar contas, e fazer todas as operações no navegador.

Vamos fazer nosso **index.html** básico só pra rodar no navegador e abrir o console para simular a abertura de contas, etc.

****

No console vou fazer um teste criando uma conta nova como constante e dando o nome que eu quiser pra ela (neste caso minhaConta), depois em new defino se é corrente, universitaria, etc.

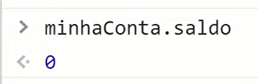


Ao digitar parenteses já consigo acessar as propriedades, o console já mostra o que precisa inserir de dados da conta bancária.

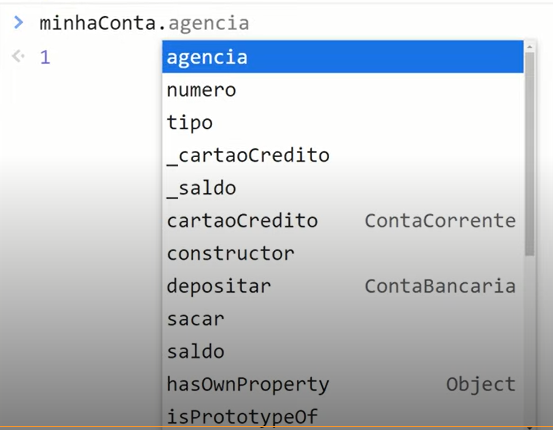
Inserindo os dados:



Posso acessar informações de saldo da minha conta por ex:



Só de colocar ponto, aparecem as opções que tenho para manipular os dados:



depositando 1000 :



Sacando 500:



Verificando saldo:



Todas as operações funcionam.