```
//HERANÇA:
//O conceito de herança em Javascript é semelhante ao conceito de herança nas demais linguagens de programação. Porém a maneira como a herança funciona no Javascript é totalmente diferente.
//Como sabemos, todo objeto em Javascript pertence a função interna Object do Javascript. E como uma função construtora, Object possuí propriedades, uma delas é o "prototype", essa propriedade é responsável por referenciar a um determinado objeto pai.
//Object é a função môr de todos os objetos, ou seja, ela não pode ter um pai, ela é o pai de todos, por isso, qualquer objeto criado á partir dele terá o Object como seu prototype.
//E isso vai acontecendo de função construtora para função construtora, por exemplo, se temos um objeto pai e desejamos criar um objeto filho que herde algumas características do pai, o objeto filho terá como prototype os atributos da função pai, e o pai por sua vez terá como prototype os atributos da função avô, e isso vai acontecendo até que chegue a função Object que é o último na lista de prototypes.
//Entender essa sequência de filiações é importante para saber referenciar heranças em Javascript.
//O prototype é uma propriedade privada do Object, por isso, para referenciá-la devemos usar o método de acesso: .__proto__
//Mas cuidado para não confundir, a propriedade [[prototype]] utilizada nas heranças de objetos é totalmente diferente da propriedade "prototype" que existe nas funções, uma coisa não tem a ver com a outra.
//OBS: Quando o assunto é Javascript, é sempre melhor priorizar o uso de composição no lugar de herança.

//ATRIBUTO PROTOTYPE DE OBJECT ESTÁ PRESENTE EM QUALQUER OBJETO:
const teste = {atributo: 'oi'} //Perceba que temos um objeto comum
console.log('\n1)', teste.__proto__ === Object.prototype) //Ao fazer a comparação entre o seu prototype e o de Object temos o mesmo prototype...



//OBJECT NÃO POSSUÍ UM PROTOTYPE SUPERIOR A ELE:
console.log('\n2)', Object.prototype.__proto__) //Veja que o resultado é null, pois Object não possuí um prototype superior...



//ATRIBUÍNDO PROTÓTIPOS SOBRE UMA CADEIA DE OBJETOS QUE HERDA UM DO OUTRO:
const avo = {attr1: 'A'}
const pai = {__proto__: avo, attr2: 'B'} //Perceba que atribuímos todos os atributos de avo em pai literalmente usando o __proto___
const filho = {__proto__: pai, attr3: 'C'} //Fizemos o mesmo em filho, onde filho herda os atributos de pai, e ainda por cima herda todos os atributos que o pai herdou de avo.
```

```
//VENDO HERANÇA ACONTECER DE ELEMENTO FILHO PARA ANCESTRAL:
console.log('\n3)', filho.attr1) //Perceba que existe uma herança onde filho procura pelo atributo attr1, ao não encontrar procura no protótipo dele que é pai, ao nao encontrar procura no protótipo de pai que é "avô", por ter encontrado no avô ele retorna a busca com o resultado do valor do atributo...



//VENDO QUE SE UM OBJETO HERDEIRO NÃO ENCONTRAR UM ELEMENTO NO ÚLTIMO ANCESTRAL ELE PROCURA NO OBJECT
Object.prototype.attr0 = 'Z' //Veja que atribuímos um atributo diretamente sobre o Object, com o valor 'Z'
console.log('\n4)', filho.attr0) //Por não encontrar o atributo nem em pai e nem em avo, ele procura no Object...



//SHADOWING NA HERANÇA:
//O shadowing é um princípio que acontece na herança quando temos um atributo com o mesmo nome em objetos diferentes dentro de uma mesma cadeira de heranças, nesse caso, o objeto herdeiro irá referenciar ao elemento mais próximo dele na cadeia...
const rei = {cabelo: 'preto', olhos: 'castanhos'}
const principe = {__proto__: rei, cabelo: 'loiro'} //Perceba que o príncipe herda os atributos do rei, mas o cabelo dele sobrescreve o atributo cabelo para "loiro", isso que acabou de acontecer é o shadowing...
const futuroPrincipe = {__proto__: principe}
console.log('\n5)', futuroPrincipe.cabelo, futuroPrincipe.olhos) //Quando puxamos os atributos do futuroPrincipe, ele herda o cabelo loiro do pai por causa do shadowing, mas os olhos ele herda do seu avo...



//ATRIBUINDO HERANÇA DE PROTOTYPE NO FORMATO MAIS MODERNO DO ECMASCRIPT 2015:
//O formato de herança mais moderno do ECMA Script 2015 trouxe a função "setPrototypeOf()", usar essa função é mais indicado do que utilizar o método literal utilizando o método __proto__, um método que não é suportado se o usuário estiver usando um browser muito antigo...
const carro = { //Perceba que temos um objeto carro que possuí atributo para...
    velMax: 200, //Velocidade máxima, com o padrão de 200Km/h...
    velAtual: 0, //Velocidade atual comb o padrão e 0...
    aceleraMais(delta) { //Uma função que pega o this atual dela e compara com a velocidade máxima para não deixar ultrapasar
        if (this.velAtual + delta <= this.velMax) {
            this.velAtual += delta
        } else {
            this.velAtual = this.velMax
```

```
        }
    },
    status() {//E uma função de status que mostra qual a velocidade em que o carro está e quanto ele pode atingir...
        return `${this.velAtual}Km/h de ${this.velMax}Km/h`
    }

}

//Criamos um objeto ferrari que irá herdar os atributos de métodos de carro, porém, irá sobrescrever o atributo "velMax" assim que a prototipação acontecer...
const ferrari = {
    modelo: 'F40',
    velMax: 324
}

//Temos também um objeto volvo que vai sobrescrever somente o método status do objeto carro quando a prototipação acontecer...
const volvo = {
    modelo: 'V40',
    status() {
        return `${this.modelo}: ${super.status()}`//Perceba 2 coisas aqui:
        //1º: lembre-se que o objetivo da criação desse método é sobrescrever o já existente no protótipo de carro, então, assim que a herança acontecer, o this irá referenciar ao próprio objeto volvo, que será o this da vez.
        //2º: estamos usando "super" para chamar o próprio status do protótipo, que é o método status do objeto carro. Sempre que usarmos super, vamos referenciar ao elemento pai em uma herança, e não ao objeto chamador. Se colocássemos "this" aqui teríamos um estouro de pilha, pois ele iria chamar ao próprio status do volvo infinitamente...
    }
}

Object.setPrototypeOf(ferrari, carro) //Aqui temos a prototipação de fato com a função "setPrototypeOf()", onde colocamos sempre o objeto que desejamos que ser o herdeiro, e depois o objeto que desejamos que tenha seus elementos herdados...
Object.setPrototypeOf(volvo, carro)

//Veja como a herança ocorreu corretamente entre os objetos e o objeto pai "carro"
ferrari.aceleraMais(300)
console.log('\n6)', ferrari.status())
```

```
volvo.aceleraMais(180)
console.log('\n7)', volvo.status()) //Veja como o método status causou uma sobrescrita sobre o status do objeto pai "carro"



//USANDO OBJECT.CREATE() PARA GERAR HERANÇA:
//Outra forma de gerar herança é usando o método "create()" da função Object, da seguinte forma:
const paiCreate = {nome: 'João', cabelo: 'Preto'} //Temos um objeto pai...
const filhaCreate = Object.create(pai) //E criamos um novo objeto para filha, onde o Object.create recebe como parâmetro o nome do
objeto que desejamos herdar...
console.log('\n8)', filhaCreate.nome, filhaCreate.cabelo) //Veja que a herança já aconteceu, os atributos foram herdados, mas ele
está em vazio por que o create() só permite que um objeto herde as chaves, ele não permite que valores sejam passados entre o
elemento pai e o elemento filho...

filhaCreate.nome = 'Ana' //Agora sim nós passamos valores para as chaves...
filhaCreate.cabelo = 'Loiro'
console.log('8)', filhaCreate.nome, filhaCreate.cabelo) //E eles vão receber normalmente os valores como podemos ver no console...



//ATRIBUINDO VALORES DIRETAMENTE NO OBJECT.CREATE E AINDA MEXENDO NAS PROPRIEDADES:
//É possível atribuir valores a um objeto filho assim que usamos o método create(), e ainda por cima, podemos alterar as propriedades
de chave desse elemento...
const filhaCreate2 = Object.create(pai, { //Veja que podemos atribuir valores as chaves herdadas abrindo um objeto dentro do campo de
parâmetros do método create e colocando as chaves, seus valores, se podem ser sobrescritos e sua visibilidade...
    nome: {value: 'Rafaela', writable: false, enumerable: true}
})
filhaCreate2.nome = 'Carla' //Perceba que, como colocamos que a chave nome não pode ser sobrescrita, não podemos fazer a mudança de
nome...
console.log('\n9)', filhaCreate2.nome, filhaCreate2.cabelo) //Perceba que os valores das chaves só ficam visíveis quando declaramos
valores para elas...
console.log('9)', Object.keys(filhaCreate2)) //Apesar de ter o atributo cabelo herdado, para o javascript é como se o objeto
filhaCreate2 tivesse somente a chave "nome dentro dela..."



//VERIFICANDO QUE CHAVES SÃO HERDADAS COM O MÉTODO HASOWNPROPERTY():
```

```
//O método "hasOwnProperty()" é usado para verificar que valores realmente pertencem a um objeto e quais são herdados de um outro objeto, veja como podemos utilizá-lo:
const paiHasOwn = {nome: 'Pedro', cabelo: 'Preto'}
const filhaHasOwn = Object.create(paiHasOwn, { //Veja que filhaHasOwn pegou o atributo nome e o sobrescreveu, fazendo com que a herança agora alterada a sua maneira
    nome: {value: 'Patrícia', writable: false, enumerable: true}
})
filhaHasOwn.altura = 1.75 //E criamos uma chave nova para o elemento "altura"
console.log('\n10)', filhaHasOwn.hasOwnProperty('nome')) //Perceba que todas as chaves alteradas ou criadas no próprio objetos retornam valor true
console.log('10)', filhaHasOwn.hasOwnProperty('altura'))
console.log('10)', filhaHasOwn.hasOwnProperty('cabelo')) //Mas chaves que não foram criadas nem alteradas são retornam valor false...



//USANDO HASOWNPROPERTY DE FORMA MAIS INTELIGENTE:
console.log('\n11)')
for (let i in filhaHasOwn) { //Usando um laço for podemos iterar mais facilmente...
    filhaHasOwn.hasOwnProperty(i) ? console.log(`Pertence a mim ${i}`) : console.log(`Não pertence a mim ${i}`)
}



//PROTOTYPE EM FUNÇÕES:
//As funções possuem um prototype próprio delas, mas elas também possuem um prototype que referencia a função Object afinal, as funções em Javascript também são tratadas como objetos. Mas as formas de referenciá-las são diferentes.
//Quando usamos "prototype" diretamente sobre uma função, nós referenciamos ao prototype do elemento Function do Javascript, mas quando referenciamos ao [[prototype]] da função, por através do __proto__ nós referenciamos a função Object.
function MeuObjeto() {} //Criamos aqui uma função, toda função têm um prototype interno...
console.log('\n12)', MeuObjeto.prototype === Object.prototype) //Mas o prototype de uma função é somente dela, não pode ser referenciado pelo prototype da função Object...

const objFuncaoProto1 = new MeuObjeto //Quando instanciamos um objeto sobre uma função, mesmo que ela são seja construtora de fato, como é o exemplo da função acima, ela irá se comportar como construtora e ela irá referenciar ao prototype da função, isso faz com que objetos possam herdar atributos de suas classes por assim dizer...
console.log('12)', MeuObjeto.prototype === objFuncaoProto1.__proto__)
```

```
//PROTOTYPE DAS FUNÇÕES É DIFERENTE DOS PROTOTYPES DOS OBJECTS:
console.log('\n13)', MeuObjeto.__proto__ === Function.prototype) //Perceba que o __proto__ de uma função construtora não referencia ao prototype da função Object, afinal uma função é criada á partir do tipo Function, referenciado a função Function...

console.log('13)', Function.prototype.__proto__ === Object.prototype) //Porém, o objeto Function tem como prototype de referência a função Object, o que quer dizer que dentro do javascript toda função é um objeto...



//GERANDO NOVAS PROPRIEDADES PARA A MINHA FUNÇÃO CONSTRUTORA:
//É possível criar mais propriedades para uma função construtora, mas temos que fazer isso acessando o prototype da função...
MeuObjeto.prototype.nome = "Anônimo" //Veja que após ter criado a função "MeuObjeto" adicionamos um atributo "nome" usando o prototype...
MeuObjeto.prototype.falar = function () { //E também adicionamos uma função falar() que fala o nome da pessoa. ATENÇÃO!!! ARROW FUNCTIONS NÃO FUNCIONAM BEM DENTRO DE FUNÇÕES CONSTRUTORAS QUANDO UTILIZAMOS THIS, POIS O THIS DE UMA ARROW FUNCTION SEMPRE VAI REFERENCIAR AO CONTEXTO DE ONDE A FUNÇÃO É CRIADA...
    console.log(`Meu nome é ${this.nome}`)
}

console.log('\n14:')
objFuncaoProto1.falar() //Veja que assim que adicionamos novos atributos, eles já estão disponíveis para que as instâncias possam utilizá-los...



//CRIANDO O PRÓPRIO NEW - PARA INSTANCIAR - COM O AUXÍLIO DO PROTOTYPE:
//Esse exemplo abaixo é meramente ditático, para mostrar como o prototype tem uma missão importante quando falamos de instanciar objetos. Para isso vamos criar o nosso próprio new usando o prototype...
const aula = function(nome, videoID) { //Criamos aqui uma função construtora para receber novas instâncias, onde o objetivo dela é retornar um objeto com nome de uma aula e o Id para o vídeo da aula
    this.nome = nome
    this.video = videoID
}
```

```
const aula1 = new aula('Bem vindo', 1234) //Naturalmente, podemos criar o objeto usando o new naturalmente...
console.log('\n15)', aula1) //E teremos o objeto...

//mas lembre-se que o nosso objetivo é criar o nosso próprio new, e isso é possível da seguinte forma...
const novo = function(funcao, ...params) { //Perceba que o nosso new recebe o nome de "novo", e ele recebe uma função como parâmetro e também um campo de conjunto de parâmetros com a ajuda o operador spread "..." - O operador spread é um operador que recebe vários argumentos e os junta em um único argumento, esses argumentos ficam armazenados em um array...
    const objeto = {} //Nossa função cria um objeto vazio...
    objeto.__proto__ = funcao.prototype //Esse objeto recebe como __proto__ de referência o prototype da função que for passada como parâmetros, seja ela qual for...
    funcao.apply(objeto, params) //Ao final, nós aplicamos á função passada como parâmetro o objeto vazio que nós acabamos de criar e qualquer quantidade de parâmetros que esse objeto precise serão passados ao spread por através do argumento "params". E usamos também o método "apply()", esse método é usado sempre que desejamos aplicar uma função construtora - que contenha o elemento "this" - a um determinado objeto, e também aplicamos um array - geralmente armazenado numa variável como é o caso aqui - onde o 1º argumento do apply é o objeto que iremos instanciar e o 2º argumento é a variável que contém o array de argumentos da função...
    return objeto //Por fim, retornamos o objeto em si, que virá já instanciado...
}

const aula2 = novo(aula, 'Até breve', 4567) //Perceba que a forma de usar nossa função new criada e a função new convencional é praticamente a mesma, com a excessão de que o nome da função construtora vai dentro dos parâmetros da nossa função new criada...
console.log('15)', aula2) //Mas o resultado é o mesmo...
```

RESULTADO NO CONSOLE...

```
[Running] node "c:\Users\Almoxarifado\Documents\JAVASCRIPT\arquivos_das_aulas\084-Heranca.js"

1) true

2) null

3) A

4) Z
```

```
5) loiro castanhos

6) 300Km/h de 324Km/h

7) V40: 180Km/h de 200Km/h

8) undefined undefined
8) Ana Loiro

9) Rafaela undefined
9) [ 'nome' ]

10) true
10) true
10) false

11)
Pertence a mim nome
Pertence a mim altura
Não pertence a mim cabelo
Não pertence a mim attr0

12) false
12) true

13) true
13) true

14:
Meu nome é Anônimo

15) aula { nome: 'Bem vindo', video: 1234 }
15) aula { nome: 'Até breve', video: 4567 }

[Done] exited with code=0 in 0.122 seconds
```