```
//PROTOTYPE:
//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 OBJETC ESTÁ PRESENTE EM QUALQUER OBJETO:
//OBS: o método __proto__ não é o mais indicado, é preferível usarmos a função "setPrototypeOf()", falamos mais dela na lição "086"...
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...



//UMA FUNCTION GERA UM PROTOTYPE PRÓPRIO DELA:
const funcao = () => {} //Perceba que o prototype de uma função é diferente do prototype do Object...
console.log('\n3)', funcao.prototype === Object.prototype)



//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('\n4)', 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('4)', MeuObjeto.prototype === objFuncaoProto1.__proto__)



//PROTOTYPE DAS FUNÇÕES É DIFERENTE DOS PROTOTYPES DOS OBJECTS:
console.log('\n5)', 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('5)', 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('\n6):')
objFuncaoProto1.falar() //Veja que assim que adicionamos novos atributos, eles já estão disponíveis para que as instâncias possam utilizá-los...
```

```
//INCLUÍNDO MUDANÇAS NOS TIPOS PRIMITIVOS DO JAVASCRIPT:
//ATENÇÃO!!! ISSO É ALTAMENTE NÃO RECOMENDADO!!! POIS PODEM GERAR EFEITOS COLATERIAS IMENSURÁVEIS...
//Só a título de exemplo, como todos os tipos primitivos em Javascript são funções é possível nós alterarmos essas funções
adicionando comportamentos a elas por através do prototype, vejamos alguns exemplos:
String.prototype.reverse = function() { //Não existe um método no elemento String para reverter as letras de uma string, mas veja que
criamos um método ingessado diretamento no tipo primitivo usando o prototype...
    return this.split('').reverse().join('') //Veja que a função pega uma string, gera um array com os caracteres dela, reverte o
array usando o reverse() e depois junta tudo com o join() em uma string invertida...
}

const nomeRevetido = 'Gabriel'
console.log('\n7)', nomeRevetido.reverse()) //Veja que, embora o método reverse não fosse existente no tipo String, agora ele existe,
tudo por causa do prototype...
```

RESULTADO NO CONSOLE...

```
[Running] node "c:\Users\Almoxarifado\Documents\JAVASCRIPT\arquivos_das_aulas\085-Prototype.js"

1) true

2) null

3) false

4) false
4) true

5) true
5) true

6):
Meu nome é Anônimo
```

```
7) leirbaG

[Done] exited with code=0 in 0.114 seconds
```