```
function sayHi() {
    console.log('Hi');
}
```

Essa é uma declaração literal de função.

Ela declarada dessa maneira ocorre uma coisa que a gente já falou uma das diferenças entre **var** e **let**. Ocorre o function hoisting, isso significa que quando a gente declara a função da maneira que está na imagem acima só dessa maneira que ocorre o hoisting seria, a engine do JavaScript vai elevar as declarações de Funções e Variáveis declaradas com a palavra **var** pro topo do nosso JavaScript. Isso significa que eu posso chamar a função acima dessa maneira:

sayHi();

Ou eu posso usar a função antes de chamar ela:

```
sayHi();

function sayHi() {
    console.log('Hi');
}
```

Se você criar dessa maneira com várias funções o código vai funcionar porque ocorre o hositing. Todas as declarações vão ser elevadas pelo topo do JavaScript para o topo do arquivo na hora da execução, então não vai fazer a mínima diferença você chamar a função antes de declarar ela ou depois de declarar ela, vai funcionar perfeitamente.

Uma coisa interessante sobre funções em JavaScript é que elas são:

First-class objects

As funções são objetos de primeira classe

Isso quer dizer que a gente quer falar que a função pode ser tratada em JavaScript e em outras linguagens de programação por exemplo eu posso falar com propriedade do Python que também faz isso, mas você tratar as funções como dado.

Então, assim como eu faço:

```
const nome = 'Leo';
```

Eu também posso usar a expressão de função que no caso seria criar uma função como um dado:

Function expression

Que seria assim:

```
const souUmDado = function() {
    console.log('Sou um dado.');
}
```

Explicação: Eu estou jogando uma função como resultado de uma constante, então eu estou falando que uma constante recebe uma função como dado.

E agora eu posso executar essa variável como uma função normal.

souUmDado();

Porque a minha variável recebeu uma função e ela passou a ser uma função então por isso que eu posso tratar uma função como um dado em JavaScript. Isso é muito poderoso porque agora eu posso jogar essa variável como parâmetro de outra função e fazer essa outra função executar a minha função. Por exemplo:

```
function executaFuncao(funcao) {
    funcao();
}
```

Isso é uma coisa muito louca na programação pois eu estou recebendo um parâmetro (que é como se fosse uma variável que eu vou receber na minha função), mas nesse caso eu estou esperando uma função e vou executá-la.

```
function executaFuncao(funcao) {
    funcao();
}
executaFuncao(souUmDado);
```

Ele está executando meu parâmetro;

Então por isso que a gente fala que a função em JavaScript é um objeto de primeira classe que a gente pode tratar a função como um dado naturalmente a gente pode passar ela pra outra função, retornar ela de outra função, a gente pode fazer várias coisas com a função. A gente pode fazer também **Arrow Functions,** ele é um recurso mais novo do EcmaScript (ES6) 2015, **Arrow Function** seria uma

declaração de função, seria uma function expression só que de uma forma bem mais curta:

```
const funcaoArrow = () => {
    console.log('Sou uma arrow function.')
};
```

Então isso é uma função normal também que você trataria exatamente como qualquer outra função em JavaScript. Só que vai dar uma diferença quando a gente for falar sobre a palavra **this** você vai entender melhor a diferença dessa função acima das outras funções que vimos anteriormente. As que usam a palavra **function** vão divergir um pouquinho na palavra **this**.

Neste momento, todas essas funções fazem exatamente a mesma coisa: executar uma tarefa. Então eu poderia fazer:

```
funcaoArrow();
```

Todas elas (funções) são tratadas como função de primeira classe, todas elas são tratadas como dado. Tanto que nesse exemplo abaixo também estamos tratando como um (anônima) dado, porque a gente está passando a função como parâmetro pra outra função que seria o setInterval:

```
setInterval(function() {

}, 1000);
```

E nós vimos também como se declararia uma função externa e chamaria a função:

```
setInterval(afuncao, 1000);
```

A função como parâmetro dessa maneira em cima. A gente não poderia executar a função colocando os (), mas passar ela como um dado.

A gente viu também um pouco em objetos mas a gente não viu a fundo que, dentro de um objeto eu posso ter uma função, então eu posso falar:

```
const obj = {

}
```

E aí eu posso jogar em um atributo desse objeto eu posso criar uma função, por exemplo:

```
const obj = {
    falar: function () {
        console.log('Estou falando...');
    }
};
obj.falar();
```

Essa é a maneira clássica de jogar uma função como um método de um objeto. Então eu posso usar anotação de ponto pra executar minha função que está dentro do meu objeto.

Mas agora, nas versão mais novas do JavaScript eu posso também já criar esse método sem a palavra function e sem os :

```
const obj = {
    falar() {
        console.log('Estou falando...');
    }
};
obj.falar();
```

Então essa função falar() já vai ser um método dentro do objeto.