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

funcao();
```

A gente pode passar parâmetros pra essa função por exemplo:

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

funcao('Valor');
```

A gente obteria um erro do JavaScript fazendo isso mas o que acontece é o contrário porque eu estou enviando um argumento para suprir o valor do parâmetro, mas não existe parâmetro. Então o JS não liga com que eu estou fazendo com os parâemtros então eu posso ou não ter um parâmetro na minha função.

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

funcao('Valor', 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12);
```

Mas para onde que foram todos esses meus argumentos?

O JavaScript tem uma carta na manga que quando a gente define a função somente com a palavra function (não funciona pra Arrow Function) então quando eu defino uma função com a palavra function ou uma function expression que é quando a gente joga a função numa variável a gente tem dentro disponível uma variável chamada *arguments,* essa variável sustenta todos os argumentos que eu enviei:

```
[Arguments] {
  '0': 'Valor',
  '1': 1,
  '2': 2,
  '3': 3,
  '4': 4,
  '5': 5,
```

```
  '6': 6,
  '7': 7,
  '8': 8,
  '9': 9,
  '10': 10,
  '11': 11,
  '12': 12
}
```

Tudo o que eu enviei está dentro de um objeto. Eu posso pegar o argumento de qualquer índice e isso é meio poderoso porque eu posso por exemplo criar uma função que soma todos os valores recebidos, eu crio uma variável pra sustentar todos o meu total de valores e daí eu vou criar um loop for of para *arguments* e daí eu vou somar o meu total de valores com o valor do argumento:

```
function funcao() {
   let total = 0;
   for (let argument of arguments) {
     total += argument;
   }
   console.log(total);
}

funcao(1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12);
```

Agora eu vou ter a soma total de todos os argumentos que eu passei pra função, mesmo eu não tendo criado nenhum parâemtro. Então a gente pode fazer isso é um pouco estranho porque ninguém que isso aconteça em uma linguagem de programação a gente esperaria um erro já que a função não possuí um parâmetro.

Resumindo: A função definida com a palavra function tem uma variável especial que chama *arguments* que sustenta todos os argumentos enviados.

E isso é interessante porque nada vai mudar mesmo se eu colocar parâmetros então isso indica que para cada um desses itens eu tenho um parâmetro então a partir do 4 em diante vai estar em *arguments,* aliás, todos os itens continuam nessa variável então mesmo eu criando parâmetros para a função o valor não vai ser alterado: 78.

Mas agora eu posso ver que a, b e c sustentam os valores: 1, 2 e 3.

```
function funcao(a, b, c) {
```

```
    let total = 0;
    for (let argument of arguments) {
      total += argument;
    }
    console.log(total, a, b, c);
}

funcao(1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12);
```

Então até agora o que a gente sabe é que a gente pode ou não enviar argumentos para a função e a gente pode ou não criar os parâmetros se a gente quiser. E a gente sabe que a variável *arguments* sustenta todos os argumentos enviados mas só para funções criadas com a palavra function, isso não funciona em Arrow Function.

Parte 2

Imagine que eu tenho mais 3 parâmetros então eles são os argumentos que eu pretendo esperar na minha função:

```
function funcao(a, b, c, d, e, f) {
    console.log(a, b, c, d, e, f)
}

funcao(1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12);
```

Ele vai exibir até o 6. Mas imagina o contrário agora se eu mando só até o 3:

```
function funcao(a, b, c, d, e, f) {
    console.log(a, b, c, d, e, f);
}

funcao(1, 2, 3);
```

O que vai acontecer com o resto das variáveis -> d, e, f é isso:

```
1 2 3 undefined undefined undefined
```

Então o que o JavaScript foi criar essas variáveis e assim como a gente declara uma variável sem valor, sem inicializar no JS ele setou o valor dos parâmetros como undefined, ou seja, se eu não enviei um valor ele vai colocar o valor padrão na

variável *undefined.* Então o JS não vai gerar nenhum erro na minha função, então quando eu divirjo números de argumentos com número de parâmetros a ordem não importa pode ser mais argumentos do que parâmetros e pode ser mais parâmetros do que argumentos, eles são valores que eu envio para os parâmetros.

Agora, suponha que eu tenha uma função que soma e aí eu mando somente 1 argumento de 2:

```
function funcao(a, b) {
   console.log(a + b);
}

funcao(3);
```

O resultado vai ser: NaN, porque não é uma conta aritmética válida. Então quando eu faço isso eu obtenho esse resultado e não era o que eu queria então eu deveria colocar um valor padrão na variável b. A maneira mais antiga de se fazer seria:

```
b = b || 0;
```

Ele verifica se b tem algum valor, se não ele é 0. Então fazendo isso agora a gente tem o valor 2 porque agora b obteve um valor padrão que é 0. Um caso mais moderno de se fazer que foi adicionado ao JS seria:

```
function funcao(a, b = 2)
```

```
function funcao(a, b = 2) {
   console.log(a + b);
   console.log('We have to overcome our difficulties.');
   // JS is crowded with methods of arrays
   // Parameter is different from other languages
   // The more the merrier!
}

funcao(3);
```

Se eu tento passar uma string vazia ele junta os números:

```
function funcao(a, b = 2, c = 33) {
   console.log(a + b + c);
   console.log('We have to overcome our difficulties.');
```

```
}

funcao(3, '', 1);
```

-> 31

A única maneira do b assumir o valor padrão é colocando undefined:

```
function funcao(a, b = 2, c = 33) {
   console.log(a + b + c);
   console.log('We have to overcome our difficulties.');
}

funcao(3, undefined, 1); // 6
```

O null é assumido como 0 se eu colocar, ele não assume o valor padrão. Então se eu tiver um parâmetro padrão no meio do caminho eu quero que ele assuma esse valor padrão é enviando o undefined. Não é muito bom fazer isso, o ideal seria mudar a minha lógica de alguma maneira porque essa lógica não está boa mas é uma maneira de assumir o valor padrão usando o undefined.

Atribuição via desestruturação

```
function funcao({nome, sobrenome, idade}) {
   console.log(nome, sobrenome, idade);
   console.log('We have to overcome our difficulties.');

}

funcao({ nome: "Léo", sobrenome: "Machado", idade: 18}
);
```

Eu estou enviando um objeto literal como argumento da minha função e nos parâmetros eu estou fazendo a desestruturação e retirando as variáveis.

Também poderia fazer assim:

```
let obj = { nome: "Léo", sobrenome: "Machado", idade: 18};

funcao(obj);
```

Além de objeto também posso fazer a desestruturação de array. Eu poderia jogar o array numa variável e joga-lá como argumento da minha função, mas aqui eu vou mandar os valores literais:

```
function funcao([valor1, valor2, valor3]) {
   console.log(valor1, valor2, valor3);
   console.log('We have to overcome our difficulties.');

}

funcao(['Leonardo Mancilha', 'Machado', 18]);
```

Outra coisa muito interessante que podemos fazer e que é muito similar ao *arguments* que a gente viu no começo que são os que sustentam os argumentos da função, uma outra coisa que eu posso fazer é -> suponha que eu tenha uma função que faça conta que recebe um operador, acumulador e números:

```
function funcao([operador, acumulador, numeros]) {
   console.log(operador, acumulador, numeros);
   console.log('We have to overcome our difficulties.');

}
```

A ideia seria que no operador eu mando um operador: + / - ou *;
No acumulador eu mando um valor inicial pra fazer uma acumulação; Por exemplo se eu quero somar todos os valores eu tenho um acumulador que começa do 0 e daí eu somo todos os números;
E daí eu faço a conta nesse acumulador que nem como a gente fez com o total em uma função anterior;

```
function conta([operador, acumulador, numeros]) {
   console.log(operador, acumulador, numeros);
   console.log('We have to overcome our difficulties.');

}

conta(['+', 0, [20, 30, 42]]);
```

Mas eu não preciso mandar o array de números e eu posso utilizar números com operador de resto (rest operator), então eu quero receber o operador e o acumulador e todo o resto dos argumentos que eu quero receber eu quero receber no array numeros:

```
function conta(operador, acumulador, ...numeros) {
   console.log(operador, acumulador, numeros);
   console.log('We have to overcome our difficulties.');

}

conta('+', 0, 20, 30, 42);
```

Agora todo o resto depois do operador e do acumulador 0 vai estar dentro de um array. A gente tem o mesmo resultado usando o *rest operator.*

Agora a gente pode fazer a funcionalidade da função:

```
function conta(operador, acumulador, ...numeros) {
   for (let numero of numeros) {
      console.log(numeros);
   }
   console.log('We have to overcome our difficulties.');

}

conta('+', 0, 20, 30, 42);
```

Então o que quero fazer é fazer o meu acumulador acumular esses valores, então a cada volta desse laço eu vou somar cada um desses valores no meu acumulador:

```
function conta(operador, acumulador, ...numeros) {
   for (let numero of numeros) {
      acumulador += numero;
   }
   console.log('We have to overcome our difficulties.');
   console.log(acumulador);

}

conta('+', 0, 20, 30, 42);
```

Mas não era exatamante isso que que queria eu queria saber qual é o operador pra eu saber qual conta fazer usando o if:

```
function conta(operador, acumulador, ...numeros) {
   for (let numero of numeros) {
      if (operador === '+') acumulador += numero;
```

```
      if (operador === '-') acumulador -= numero;
      if (operador === '/') acumulador /= numero;
      if (operador === '*') acumulador *= numero;
   }
   console.log('We have to overcome our difficulties.');
   console.log(acumulador);

}

conta('/', 11, 20, 30, 42);
```

Então esse é o rest operator e ele sempre deve ser o último parâmetro da função, porque ele é o resto então não tem como ele não ser o resto se ele não for o último.

Como eu não te falei a gente pode utilizar qualquer tipo de expressão de função para criar essa função acima, eu poderia fazer assim utilizando *function expression*:

```
const conta = function (operador, acumulador, ...numeros) {
   for (let numero of numeros) {
      if (operador === '+') acumulador += numero;
      if (operador === '-') acumulador -= numero;
      if (operador === '/') acumulador /= numero;
      if (operador === '*') acumulador *= numero;
   }
   console.log('We have to overcome our difficulties.');
   console.log(acumulador);

};

conta('+', 11, 20, 30, 42);
```

Agora, mesmo com rest operator a gente continua tendo os *arguments*:

```
const conta = function (operador, acumulador, ...numeros) {
   console.log(arguments);
}
conta('+', 11, 20, 30, 42);
```

Então isso vai funcionar em qualquer lugar que eu usar a palavra function mas eu muda para Arrow Function eu não saio mais os argumentos dentro:

```
const conta = (operador, acumulador, ...numeros) => {
   console.log(arguments);
```

```
}
conta('+', 11, 20, 30, 42);
```

Na verdade ele mostra uma coisa do Node completamente diferente do que estamos vendo em parâmetros de função.

Então não existe *arguments* em Arrow Functions;

Como eu te falei então todos os modos que você tem para declarar funções todos eles vão funcionar de maneira extremamente similar, a [unica diferença que vai ter de Arrow Function para function normal é que a palavra **this** vai modificar um pouco, o resto vai funcionar perfeitamente.

Dica sobre os *arguments*: Desde que você tem o rest operator você pode usar ele para pegar seus argumentos:

```
const conta = (...args) => {
   console.log(args);
}
conta('+', 11, 20, 30, 42);
```

Você tem a mesma coisa que você teria com *arguments* só não pode esquecer de colocar os 3 pontinhos … para que ele seja o rest operator, se eu tiro o args vai virar o meu primeiro argumento que é o sinal de +. O rest operator vai funcionar em qualquer tipo de função ele não tem a restrição do Arrow Function que o *arguments* tem. Então sempre que você precisar parâmetros interminados pra uma função como foi o caso da operação que a gente fez de receber muitos valores e fazer a conta de todos eles usa o *rest operator* que já tem pronto na função.