Usando funções para otimizar o código

Até o momento aprendemos a utilizar a parte básica de funções, que se resume em criar e utilizar dentro do código.

Nesta etapa vamos fazer o "bicho pegar" e explorar o lado negro das funções 😃

Funções e desvios de fluxos

As funções podem ser atribuídas a variáveis. Isso quer dizer que de certa forma podemos alterar o funcionamento do programa principal caso a variável que armazena a função seja sobrescrita.

Veja este caso americano:

```
var pais = 'ALGUM';

var lancarMissilNuclear = function() {
    console.log("Lançando míssil em 3..2..1");
}

if (pais == 'RUSSIA') {
  lancarMissilNuclear = function() {
      console.log("Usar o telefone vermelho...");
  }
}

lancarMissilNuclear();
// → Lançando míssil em 3..2..1
```

Esta característica única do JavaScript permite fazer muitas coisas interessantes que iremos aprender mais para frente, entre elas algo chamado Metaprogramação.

Pilha de chamadas

Para permitir o uso de funções, o JavaScript (assim como outros) implementa uma pilha de chamadas (*call stack*).

Esta pilha de funções funciona como uma espécie de "GPS" para o compilador. A cada chamada é armazenado na *call stack* a posição da função atual, para que ao terminar a execução o compilador possa continuar executando de onde parou.

Veja:

```
function ola(quem) {
    console.log("Olá " + quem);
}
```

```
ola("Fulaninho");
console.log("Tchau");
```

A pilha de funções do programa acima se resume em:

- Principal
  - `ola`
    - `console.log`
  - `ola`
- Principal
  - `console.log`
- Principal

O armazenamento dessa pilha de chamadas necessita de espaço na memória do computador, que no caso é limitada a poucos bytes. Quando a pilha aumenta de tamanho e "estoura" teremos a mensagem *maximum call stack size exceeded*.

Veja um programa que tenta desvendar um mistério do universo:

```
function galinha() {
  return ovo();
}

function ovo() {
  return galinha();
}
console.log(galinha() + " veio primeiro.");
```

Argumentos opcionais

É bastante comum algumas funções implementarem parâmetros opcionais. O próprio `alert` implementa e executa sem problemas, veja:

```
alert("Olá", "Tudo bem?", "Como vai!?");
```

Oficialmente a função `alert` exibe apenas uma mensagem na tela (nesse caso o "Olá"), mas aceita sem problemas os outros parâmetros.

Esse comportamento é válido para parâmetros excessivos ou ausentes, como é o caso da função abaixo:

```
var ola = function(quem) {
    console.log("Olá " + quem);
}
ola("Felipe");
// → Olá Felipe
```

```
ola();
// → Olá undefined
```

Com as funções, os parâmetros que são passados "a mais" são ignorados e os que forem passados "de menos" simplesmente assumem o valor `undefined`.

Isso dá um tremendo poder para o programador, mas como o próprio Tio Ben já cantou a bola uma vez: "Com grandes poderes, vem grandes responsabilidades".

E aproveitando a parte dos "poderes" da linguagem, podemos construir funções mais inteligentes que podem se adequar a medida que forem sendo chamadas no código.

Veja no caso onde eu quero derivar o comportamento de uma função desmembrando em duas:

```
var ola1 = function() {
  console.log("Olá tudo bem?");
}

var ola2 = function(quem) {
  console.log("Olá tudo bem " + quem + "?");
}

ola1();
// → Olá tudo bem?

ola2("Felipe");
// → Olá tudo bem Felipe?
```

Como você viu para derivar o comportamento da função `ola` eu precisei criar 2 funções. E como você também percebeu este não é o melhor método.

Agora utilizando os poderes dos argumentos opcionais, o código acima seria reescrito da seguinte forma:

```
var ola = function(quem) {
  if(quem == undefined) {
    console.log("Olá tudo bem?");
  } else {
    console.log("Olá tudo bem " + quem + "?");
  }
}

ola();
// → Olá tudo bem?

ola("Felipe");
// → Olá tudo bem Felipe?
```

Agora que você já entendeu os poderes que tem em mãos chegou a hora de aplicar isso aos argumentos opcionais.

Veja o exemplo abaixo:

```
function potencia(base, expoente) {
  if (expoente == undefined) {
    expoente = 2; // define o expoente padrão
  }

  var resultado = 1;
  for (var cont = 0; cont < expoente; cont++) {
    resultado *= base;
  }

  return resultado;
}
console.log(potencia(2));
// → 4
console.log(potencia(2, 3));
// → 8
```

Recursão

Na programação em geral é OK uma função "chamar" ela mesmo. Ou seja, dentro do código existir uma chamada para ele mesmo:

```
function ola() {
  console.log("Olá");
  ola();
}

ola();
// → Uncaught RangeError: Maximum call stack size exceeded
```

Como você já percebeu é perfeitamente possível, contudo precisamos assumir diversas ressalvas com relação a isso. Principalmente com relação a Pilha de Chamadas.

Como você lembra (lembra né??) a Pilha de Chamadas é um pequeno espaço de memória do seu programa que armazena as posições de chamadas para quando uma função terminar.

Mas como lidar com um código que apenas invoca função como se não houvesse o amanhã?

Exatamente por isso que as funções recursivas são tão problemáticas e poderosas. É como domar um garanhão selvagem a pelo, caso consiga terá um verdadeiro urco.

Recursões em gerais podem ser extremamente complexas e depende de cada algoritmo, vamos entender o nosso pequeno algoritmo `potencia` e como ele poderia ser reescrito de uma forma mais selvagem e

performática.

```
function potencia(base, expoente) {
  if (expoente == 0) {
    return 1;
  } else {
    return base * potencia(base, expoente - 1); // nessa parte a recursão
dá lugar ao for
  }
}
console.log(potencia(2, 3));
// → 8
```

Por via de regra a recursão é mais rápida em algumas linguagens. Mas em implementações típicas do JavaScript é mais lenta (devido a velocidade do `if`).

No JavaScript a única vantagem de se escrever funções recursivas é quando:

1. Você quer deixar o seu código bonito.
2. Você trabalha em uma estrutura de dados dinâmica ou não conhecida (a recursão vai tratar cada dado e não o todo como o `for`). Um caso prático são os menus de sites com *n* níveis.

Desafio

Como parte da nossa viagem matemática implemente a função fatorial (sem a chamada da biblioteca `Math`) com recursão.

Lembrando que o fatorial se calcula dessa forma: https://pt.wikipedia.org/wiki/Fatorial

Resposta

```
function fatorial(n) {
  if (n == 0) {
    return 1;
  }
  else {
    return (n * fatorial(n - 1));
  }
}

fatorial(1);
// → 1
fatorial(5);
// → 120
```