Estrutura de Dados: Objetos

Até agora os arrays comportaram muito bem o nosso problema de lista de convidados. Contudo o seu cliente pediu algumas alterações no "sisteminha".

De acordo com ele: "*as vezes as pessoas colocam seus nomes na lista mas não podem entrar no evento, alguns convidados tem menos de 18 anos...*".

No sistema temos isso:

```
var nomes = ["Felipe", "Zezinho", "Fulano"];
// como recuperar mais informações se nosso array guarda apenas nomes!?
```

Como você pode ver temos somente as informações de nome, nada mais. E isso está demandando muito trabalho no momento em que é feita a verificação da entrada.

O cliente ainda continua: "*[...] precisamos ter a possibilidade de cadastrar o CPF, data de nascimento e a idade para adiantar o meu processo [...]".

Claramente o nosso sistema não dá conta desse recado, armazenamos apenas a informação de nome em nossos cadastros.

Por isso se faz necessário uma forma melhor de cadastrar a informação de cada convidado, permitindo cadastrar não apenas o nome, mas outros dados para o controle.

Objetos e outros bichos

Os objetos são estruturas de dados que podem ser composta por propriedades e ações (técnicamente métodos). Esse comportamento é inspirado por objetos do mundo real, veja alguns exemplos:

Cachorros:

- Raça
- Gênero
- Cor do pelo

Computadores:

- Processador
- Memória
- Sistema Operacional

E mais algumas ações, um exemplo:

Humanos:

- Propriedades
    - Altura

    - Peso
    - Idade
    - Gênero
    - Cor do cabelo

Praticamente qualquer coisa no planeta Terra pode ser representado por objetos (com suas propriedades).

E não é diferente aqui, sendo assim seria muito interessante representar **nosso convidado** com a seguinte estrutura de propriedades:

- Propriedades
    - Nome
    - Idade
    - Gênero
    - Data de Nascimento
    - CPF

Isso cobre 101% o que o cliente necessita, mas como fazer isso em JavaScript?

Criando objetos

Um objeto no JavaScript pode ser declarado da seguinte forma:

```
var convidado = {nome: "Felipe", idade: "37", genero: "Masculino",
dataNascimento: "18/02/1982", cpf: "037.727.730-44"}
console.log(convidado);
// → {nome: "Felipe", idade: "37", genero: "Masculino", dataNascimento:
"18/02/1982", cpf: "037.727.730-44"}
```

A beleza da coisa é que você pode acrescentar qualquer propriedade, inclusive a propriedade da propriedade (que seria objeto de objeto):

```
var convidado = {nome: "Felipe", idade: "37", genero: "Masculino",
dataNascimento: {ano: 1982, mes: 2, dia: 18}, cpf: "037.727.730-44"}
console.log(convidado);
// → {nome: "Felipe", idade: "37", genero: "Masculino", dataNascimento:
{ano: 1982, mes: 2, dia:18}, cpf: "037.727.730-44"}
```

E depois de criado é fácil acessar uma ou mais propriedade:

```
var convidado = {nome: "Felipe", idade: "37", genero: "Masculino",
dataNascimento: {ano: 1982, mes: 2, dia: 18}, cpf: "037.727.730-44"}
console.log(convidado.nome);
// → Felipe
```

Isso muda o jogo do *programinha* de cadastro de convidados, já que agora o nosso cliente poderá cadastrar os dados de cada propriedade e assim barrar penetras mais facilmente.

Como criar uma estrutura de dados complexa

A coisa está fluindo muito bem, mas como colocar toda essa galera dentro de um array? Afinal estamos criando uma "lista de convidados". Certo!?

Voltando ao querido array:

```
var convidados = []; // crio o array

// adiciono cada convidado no array, usando o push
convidados.push({nome: "Felipe", idade: "37", genero: "Masculino",
dataNascimento: {ano: 1982, mes: 2, dia: 18}, cpf: "037.727.730-44"});
convidados.push({nome: "Zezinho", idade: "17", genero: "Masculino",
dataNascimento: {ano: 2002, mes: 9, dia: 1}, cpf: "510.741.300-57"});
convidados.push({nome: "Fulano", idade: "22", genero: "Masculino",
dataNascimento: {ano: 1997, mes: 3, dia: 19}, cpf: "876.056.530-69"});

console.log(convidados);
// → [
//   {nome: "Felipe", idade: "37", genero: "Masculino", dataNascimento:
{…}, cpf: "037.727.730-44"}
//   {nome: "Zezinho", idade: "17", genero: "Masculino", dataNascimento:
{…}, cpf: "510.741.300-57"}
//   {nome: "Fulano", idade: "22", genero: "Masculino", dataNascimento:
{…}, cpf: "876.056.530-69"}
// ]
```

Como sabemos o método `push` do array pode inserir qualquer coisa, inclusive um objeto.

Muito bom!

Operadores de objetos

Os objetos possuem uma série de operadores para auxiliar o trabalho. Como o operador `delete` remove atributos de um objeto:

```
var convidado = {nome: "Felipe", idade: 37};

delete convidado.idade;

console.log(convidado);
// → {nome: "Felipe"}
```

Ou então o operador binário `in`, que indica se um objeto possui uma determinada propriedade. Veja:

```
var convidado = {nome: "Felipe", idade: 37};

console.log("nome" in convidado);
// → true
```

Mutabilidade

Como vimos até aqui os objetos possuem propriedades que podem ser preenchidas com valores a nossa escolha e necessidade.

Quando temos em um programa, dois números, por exemplo dois indivíduos com 21 anos, podemos considerar que são iguais:

```
var idade1 = 21;
var idade2 = 21;

console.log(idade1 == idade2);
// → true
```

Entretanto, isso se aplica de uma forma um pouco diferente aos objetos:

```
var convidado1 = {idade: 21}
var convidado2 = convidado1;
var convidado3 = {idade: 21}

console.log(convidado1 == convidado2);
// → true
console.log(convidado1 == convidado3);
// → false
```

O objeto `convidado1` é igual ao objeto `convidado2` porque fazem referência a mesma região da memória, portanto são o mesmo objeto.

Logo o objeto `convidado3` foi criado a partir de novas propriedades, apesar de as mesmas em nome e valor, para o JavaScript pertence a outra região de memória.

Isso se comporta dessa forma porque o JavaScript não possuí uma comparação "profunda" (deep). Ele compara apenas se um objeto faz referência a outro (mesma posição de memória).

Para fazer uma comparação profunda devemos trabalhar com uma função usando magia negra:

```
function deepEqual(x, y) {
    // primeiro verifica se x é idêntico a y
    if (x === y) {
        return true;
    }
```

```
    // caso x seja um objeto, y também seja um objeto e ambos não sejam
nulos, inicia a verificação profunda
    else if ((typeof x == "object" && x != null) && (typeof y == "object"
&& y != null)) {
        // primeira estratégia é verificar a quantidade de propriedades com
o método key
        if (Object.keys(x).length != Object.keys(y).length) {
            return false;
        }

        // caso seja a mesma quantidade inicia um loop em cada propriedade
de x
        for (var prop in x) {
            // verifica se existe a mesma propriedade em y com o método
hasOwnProperty
            if (y.hasOwnProperty(prop)) {
                // caso exista a mesma propriedade utiliza o primeiro if na
recursão para verificar se são idênticas
                if (!deepEqual(x[prop], y[prop])) {
                    return false;
                }
            } else {
                // caso não exista, descarta o restante das propriedades
                return false;
            }
        }

        // caso passe em todas as verificações do for, retorna true
        return true;
    } else {
        return false;
    }
}

var convidado1 = {idade: 21};
var convidado2 = convidado1;
var convidado3 = {idade: 21};

console.log(deepEqual(convidado1, convidado3));
// → true

console.log(deepEqual(convidado1, {idade: 21}));
// → true
```

Objetos do tipo mapas (hash)

As propriedades de objetos podem ser criadas e acessadas com uma certa facilidade:

```
var convidado = {nome: "Felipe", idade: 37};
```

```
console.log(convidado.nome);
// → Felipe
```

Mas dentro da caixa de ferramentas do JavaScript também é possível acessar estas mesmas propriedades utilizando a notação de colchetes do array:

```
var convidado = {nome: "Felipe", idade: 37};

console.log(convidado["nome"]);
// → Felipe
```

Isso é bastante interessante porque podemos criar um "array" com índice em forma de mapa, também conhecido como Hash. Veja:

```
var convidados = {};

convidados["Felipe"] = {idade: 37};

console.log(convidados);
// → {Felipe: {idade: 37}}
```

Isso é muito útil quando queremos acessar diretamente um objeto cujo o ID seja conhecido. Normalmente quando recebemos este ID de um banco de dados, por exemplo:

```
var convidados = {};

convidados["da39a3ee5e6b4b0das55bfef95231890afd80709"] = {nome: "Felipe",
idade: 37};
convidados["da39a3ee5e6b4b0d3255bfef95601890afd80709"] = {nome: "Bruno",
idade: 21};
// ...

console.log(convidados);
// → {da39a3ee5e6b4b0d3255bfef95601890afd80709: {nome: "Bruno", idade: 21}
//    da39a3ee5e6b4b0das55bfef95231890afd80709: {nome: "Felipe", idade:
37}}
```

O que nos dá uma tremenda facilidade, já que podemos utilizar os mesmos métodos conhecidos:

```
var convidados = {};

convidados["da39a3ee5e6b4b0das55bfef95231890afd80709"] = {nome: "Felipe",
idade: 37};
convidados["da39a3ee5e6b4b0d3255bfef95601890afd80709"] = {nome: "Bruno",
idade: 21};
```

```
for (var id in convidados)
{
    console.log(convidados[id]);
}

// → {nome: "Felipe", idade: 37}
// → {nome: "Bruno", idade: 21}
```

Objeto `arguments` e seu uso em funções

Na sessão sobre `function` entendemos que podemos trabalhar com parâmetros opcionais, logo isso é possível:

```
function ola(quem) {
    if (quem == undefined) {
        console.log("Olá tudo bem com você?");
    } else {
        console.log("Olá " + quem + " como vai!? Tudo bem??");
    }
}

ola("Felipe"); // é ok
ola();         // também é ok
```

Ambas as chamadas de função são totalmente possíveis no JavaScript, graça aos conceitos de parâmetros opcionais e a existência do `undefined`.

Contudo ainda não falamos da possibilidade de tratar infinitos parâmetros. Para explicar isso vamos considerar a mesma função `ola`.

Suponhamos que temos que criar uma função que não pode esquecer de ninguém. Ou seja, deve receber não apenas um, mas infinitos nomes. Como isso ficaria?!?

```
ola("Felipe", "Gabriel", "Bruno", "Douglas");
// → Olá Felipe, Gabriel, Bruno e Douglas! Tudo bem!?
```

Com você sabe a nossa função super útil aceita apenas um parâmetro, que no caso chama-se `quem`. E se fosse usado iria capturar apenas o primeiro nome ("Felipe" no caso). Então, como ficaria o restante!?

Nesse caso entra um objeto muito especial chamado `arguments`. Que na prática é um array definido automaticamente no início da chamada de cada `function` com todos os parâmetros de chamada, veja:

```
function ola() {
    console.log(arguments);
}
```

```
ola("Felipe", "Gabriel", "Bruno", "Douglas");
// → ["Felipe", "Gabriel", "Bruno", "Douglas"]
```

Isso agora abre margem para poder tratar e criar a nossa esperada chamada coletiva:

```
function ola() {
    var mensagem = "Olá ";

    for(i = 0;i < arguments.length - 1;i++) {
        mensagem += arguments[i];

        if (i < arguments.length - 2) {
            mensagem += ", ";
        }
    }

    mensagem += " e " + arguments[arguments.length - 1] + "! Tudo bem!?";

    console.log(mensagem);
}

ola("Felipe", "Gabriel", "Bruno", "Douglas");
```

O objeto `arguments` nos dá uma margem de trabalho interessante para tratar parâmetros opcionais dentro das nossas funções.

Desafio

Com o conhecimento de objetos em mãos construa um programinha de controle de convidados mais elaborado, considerando as propriedades:

- Nome
- Idade
- CPF

Permita que o cadastrante digite estas informações em nosso array de convidados.

Resposta

```
var convidados = [];

while(true) {
    var nome = prompt("Digite o nome do convidado ou S para sair");
    var idade = Number(prompt("Digite a idade do convidado"));
    var cpf = prompt("Digite o CPF do convidado");

    if (nome == "S") {
        break;
    } else {
        convidados.push({nome: nome, idade: idade, cpf: cpf});
    }
}

console.log(convidados);
```