Linguagens de Programação
Roteiro de Laboratório 02 – JavaScript

Os roteiros de laboratório iniciais cobrem o uso de JavaScript tanto como uma linguagem de programação convencional quanto como uma linguagem de *script* que funciona dentro de um navegador.

Parte II
Tipos de Dados, Classes e Objetos

Nessa segunda parte vamos falar sobre classes, objetos e outras características OO de JavaScript, que diferem consideravelmente de LPs como Java, por exemplo.

O conteúdo desse roteiro foi adaptado do **The Modern JavaScript Tutorial**, disponível em `https://javascript.info/`.

Atividade

Como já indicado no roteiro anterior, é importante que você tente reproduzir todos os exemplos do texto para ir pegando familiaridade com a linguagem e as suas ferramentas.

1 Objetos: conceitos básicos

1.1 Objetos

Conforme visto na Parte I, sabemos que há oito tipos de dados em JavaScript. Sete deles são ditos "primitivos" porque os seus valores representam uma única "coisa", como uma *string* ou um número, por exemplo.

Por outro lado, objetos em JS são usados para armazenar qualquer coleção de dados e outras entidades mais complexas. Um objeto é criado com `{...}`, com uma lista opcional de *propriedades*. Uma propriedade é um par `key : value`, aonde `key` é uma *string* (também chamada de *property name*), e `value` pode ser um valor de qualquer tipo de dado.

(Conhecedores de Python já devem ter feito uma associação imediata dos objetos de JS com os dicionários de Python. Outras LPs chamam essa construção de *arrays* associativos, mapas ou *hashs*.)

Um objeto vazio pode ser criado com duas sintaxes distintas.

```
let userA = new Object(); // "object constructor" syntax
let userB = {};  // "object literal" syntax
```

Objetos também já podem ser criados com informações.

```
let user = {     // an object
  name: "John",  // by key "name" store value "John"
  age: 30        // by key "age" store value 30
};
```

Os valores das propriedades podem ser acessados com a notação de ponto usual.

```
console.log(user.name); // John
console.log(user.age);  // 30
```

Propriedades podem ser adicionadas ou removidas de forma dinâmica.

```
user.active = true; // new property pair active : true
delete user.age;    // remove property age
```

Uma `key` pode conter espaços mas nesse caso ela deve ficar entre aspas. (Lembre que chaves são só *strings*.) Nesse caso, não é possível usar a notação de ponto, devemos usar colchetes `[...]`.

```
user["is admin"] = true;
delete user.is admin // syntax error
```

Para usar o valor de uma variável como a chave de um objeto, basta empregar `[...]`.

```
let fruit = "apple";
let bag = {
  [fruit] : 5,
};
console.log(bag.apple); // 5
```

Teste de existência de propriedade, operador `in`

Ao contrário de outras linguagens, JavaScript permite o acesso a qualquer propriedade. Caso uma chave não exista, não há erro, só é retornado o valor `undefined`.

O operador `in` serve para testar se um objeto possui uma chave.

```
let user = { name: "John", age: 30 };
console.log( "age" in user );    // true, user.age exists
console.log( "blabla" in user );// false, user.blabla doesn't exist
console.log( user.blabla );      // undefined, user.blabla doesn't exist
```

Iteração sobre objetos

Para se percorrer as chaves de um objeto, usa-se um *loop* `for` com uma sintaxe especial.

```
let user = {
  name: "John",
  age: 30,
  "is admin": true
};

for (let key in user) {
  console.log(`( ${key} : ${user[key]} )`); // ( name : John ), ...
}
```

O tipo de dado visto até agora é chamado em JS de "objeto simples" (*plain object*), ou simplesmente `Object`. Há vários outros objetos já definidos na linguagem, como `Array`, `Error`, etc. Embora costuma-se dizer: "o tipo `Array`", ou "o tipo `Error`"; estritamente falando estes não são tipos em JavaScript, já que todos pertencem ao tipo de dados `object`.

Referências a objetos, cópias e coleção de lixo

A diferença fundamental entre objetos e tipos primitivos é que todos os tipos primitivos são armazenados por **valor**, enquanto os objetos são armazenados por **referência**, exatamente como em Java.

Como esperado, o comando de atribuição faz somente uma cópia da referência do objeto. Duas variáveis contendo objetos são iguais se ambas referenciam a mesma estrutura.

```
let a = {};
let b = a; // copy the reference
console.log( a === b ); // true, both vars reference the same object
```

E, obviamente, dois objetos distintos não são iguais, mesmo quando a sua estrutura é a mesma.

```
let a = {};
let b = {}; // two independent objects
console.log( a === b ); // false
```

Para se duplicar um objeto, podemos usar o método `Object.assign`, que recebe um objeto `dest` e uma lista de objetos cujas propriedades devem copiadas para `dest`. A sintaxe é:

```
Object.assign(dest, [srcA, srcB, srcC...])
```

A chamada do método retorna o objeto de destino `dest`. Segue um exemplo completo.

```
let user = {
  name: "John",
  sizes: {
    height: 182,
    width: 50
  }
};

let clone = Object.assign({}, user);
```

Note que o comando acima realiza uma **cópia/clonagem rasa** (*shallow copy/cloning*) das propriedades. Isso quer dizer que valores primitivos são duplicados, mas objetos aninhados têm somente a sua referência copiada.

```
clone.name = "Jack";
console.log( user.name === clone.name );     // false, different names
console.log( user.sizes === clone.sizes );   // true, same object

// user and clone share sizes
user.sizes.width++;                 // change a property from one place
console.log( clone.sizes.width ); // 51, see the result from the other
```

Não há uma forma padronizada para se realizar uma **cópia/clonagem profunda** (*deep copy/-cloning*) em JavaScript. Por conta disso, desenvolvedores utilizam funcionalidades de bibliotecas, como o método `_.cloneDeep(obj)` da biblioteca `https://lodash.com/`.

A gerência de memória em JS é feita de forma automática, através de coleção de lixo, como ocorre em praticamente todas as LPs modernas.

1.2 Métodos de objetos, `this`

Já vimos anteriormente que funções são elementos de primeira classe em JavaScript. Assim, o conceito de **métodos de objetos**, algo comum em LPs OO, é implementado em JS como uma propriedade de um objeto. A chave é o nome do método e o valor é o seu código.

```
let user = {
  name: "John",
  age: 30,
```

```
  sayHello: function() {
    console.log("Hello!");
  }
};

user.sayHello(); // Hello!
```

Existe uma sintaxe mais simples para declaração de métodos, como indicado abaixo.

```
user = {
  sayHello() { // same as "sayHello: function()"
    console.log("Hello!");
  }
};
```

Quase sempre a sintaxe simplificada é preferível.

`this` em métodos

Podemos usar a palavra reservada `this` dentro do corpo de um método para acessar a referência do objeto invocador, de forma similar a Java.

```
let user = {
  name: "John",
  age: 30,
  sayName() {
    console.log(this.name);
  }
};

user.sayName(); // John
let clone = Object.assign({}, user);
clone.name = "Jack";
clone.sayName(); // Jack
```

`this` é amarrado dinamicamente

Apesar da mesma sintaxe, o `this` do JavaScript se comporta de forma diferente do que o `this` do Java em alguns casos. Por exemplo, JS permite que `this` seja usado em qualquer função, mesmo quando ela não é um método. Não há erro de sintaxe no exemplo abaixo. (Note que a função foi declarada fora de um objeto!)

```
function sayName() {
  console.log(this.name);
}
```

O valor de `this` é avaliado em tempo de execução, dependendo do contexto. Isso pode levar a situações estranhas como a do código abaixo.

```
function sayHello() {
  console.log(`Hello from ${this.name}!`);
}

let user = { name: "John" };
let machine = { code: 42 };
```

```
user.f = sayHello;
machine.f = sayHello;

user.f();     // Hello from John!
machine.f(); // Hello from undefined!
```

No caso da execução da última linha, o objeto `machine` é associado ao `this` da função. Como esse objeto não possui um campo de nome, o acesso `this.name` retorna `undefined`.

Entusiastas de linguagens estaticamente tipadas, como Java, costumam surtar quando veem um código como acima... :P Em Java, o `this` é amarrado estaticamente, com a LP proibindo a sua utilização em locais inadequados. É por isso que em Java não é possível se usar `this` dentro de métodos estáticos, já que não há um objeto invocador nesse caso.

Funções *arrow* não possuem `this`

Para complicar mais as coisas, as funções *arrow* não possuem o seu próprio `this`.

```
let user = {
  name: "John",
  sayName: () => console.log(this.name)
};

user.sayName(); // undefined
```

Por outro lado, se a função é criada aninhada dentro de um objeto, ela "pega emprestado" o `this` do escopo externo. Mais um caso particular para lembrar... :/

```
let user = {
  name: "John",
  sayName() {
    let arrow = () => console.log(this.name);
    arrow();
  }
};

user.sayName(); // John
```

1.3 Construtores e operador `new`

A sintaxe de `{...}` permite criar um único objeto. Embora seja possível cloná-lo, essa forma não é muito conveniente quando precisamos criar vários objetos similares. Para isso existem funções construtoras e o operador `new`.

Funções construtoras

Funções construtoras são iguais a funções regulares, mas há duas convenções usuais em JS:

1. Nomes de funções construtoras começam com uma letra maiúscula.
2. Elas devem ser executadas somente por meio do operador `new`.

Por exemplo:

```
function User(name) {
  this.name = name;
```

```
  this.isAdmin = false;
}

let user = new User("Jack");

console.log(user.name);      // Jack
console.log(user.isAdmin);   // false
```

Quando uma função é executada com `new`, ocorrem os seguintes passos:

1. Um novo objeto é criado e associado ao `this`.
2. O corpo da função é executado, geralmente adicionando propriedades ao `this`.
3. O valor de `this` é retornado.

Vale reforçar que o uso de uma letra maiúscula no nome de um construtor é só uma convenção, não havendo nenhuma restrição sintática pelo interpretador. De fato, qualquer função pode ser chamada com `new`!

Criação de métodos no construtor

Além de inicializar os campos de um objeto, construtores em JavaScript também podem ser utilizados para se adicionar métodos.

```
function User(name) {
  this.name = name;
  this.sayHello = function() {
    console.log(`Hello from ${this.name}!`);
  };
}

let user = new User("John");
user.sayHello(); // Hello from John!
```

Até agora, cada objeto foi manipulado de forma independente, e portanto não há nenhuma garantia de uniformidade entre eles. Por exemplo, mesmo após uma clonagem, os dois objetos não necessariamente terão sempre os mesmos campos e métodos, já que propriedades podem ser adicionadas ou removidas pontualmente. O mesmo problema ainda acontece com construtores, mas pelo menos eles proveem uma forma mais conveniente de se garantir que todos os objetos "começam iguais". Além disso, uma organização como do exemplo acima aumenta a legibilidade do código, pois o construtor concentra em um único lugar as declarações dos métodos.

Para a criação de objetos complexos é mais conveniente se usar a sintaxe de classes, que será vista adiante.

1.4 Encadeamento opcional com `?.`

A versão ES2020 do padrão de JavaScript incluiu recentemente a possibilidade de encadeamento opcional (*optional chaining*), com a sintaxe `?.` para acesso a propriedades aninhadas de objetos.

Caso alguma propriedade de um objeto seja opcional, ao tentarmos acessá-la, é possível obter de volta um `undefined`. Nesse caso, se alguma outra propriedade aninhada for acessada, obtemos um erro.

```
let user = {}; // a user without "address" property
console.log(user.address);         // undefined
console.log(user.address.street); // TypeError!
```

É possível usar a construção `value?.prop` para se fazer um acesso seguro. Esse comando:

1. Funciona como `value.prop` se `value` existe.
2. Retorna `undefined` quando `value` é `undefined` ou `null`.

Com isso, é possível realizar uma tentativa de acesso sem perigo de erro, como abaixo.

```
console.log(user?.address);          // undefined
console.log(user?.address?.street); // undefined
```

Cuidado! Use o encadeamento opcional de forma criteriosa, somente aonde é razoável que algo não exista. Caso contrário, você pode ocultar erros no código que vão ser difíceis de debugar...

1.5 O tipo `Symbol`

Até agora, foi dito que as chaves das propriedades de objetos devem ser do tipo `string`. Mas a especificação de JavaScript também permite que as chaves sejam do tipo `symbol`, o que pode levar a alguns benefícios.

Símbolos

Um símbolo representa um identificador **único** e um valor deste tipo pode ser criado com a função `Symbol()`. Um símbolo também pode ser criado com uma descrição (também chamada de nome ou chave do símbolo), geralmente utilizada para propósitos de depuração.

```
let sym = Symbol();
let id = Symbol("id"); // var id is a symbol with description "id"
```

Símbolos são garantidamente únicos. Mesmo que vários símbolos sejam criados com a mesma descrição, todos serão valores diferentes. A descrição é só um rótulo que não tem nenhum efeito. Por exemplo, no código abaixo temos dois símbolos com a mesma descrição – eles não são iguais.

```
let idA = Symbol("id");
let idB = Symbol("id");

console.log(idA == idB);  // false
console.log(idA === idB); // false
```

Símbolos não são convertidos automaticamente para *strings*

A maioria dos valores em JS são convertidos de forma implícita para uma *string* quando necessário. Por conta disso, podemos passar praticamente qualquer valor para `console.log` e ver algo impresso. A exceção são os símbolos, que não são convertidos automaticamente para *strings* para evitar que esses tipos sejam misturados acidentalmente.

```
let id = Symbol("id");
console.log(id); // TypeError: can't convert symbol to string
```

Para conseguir mostrar um símbolo, precisamos chamar o seu método `.toString()` de forma explícita.

```
let id = Symbol("id");
console.log( id.toString() ); // Symbol(id), now it works
```

Ou então acessar a propriedade `symbol.description` para exibir somente a descrição passada na criação do símbolo.

```
let id = Symbol("Identifier");
console.log( id.description ); // Identifier
```

Propriedades "escondidas"

Símbolos podem ser usados para se criar propriedades "escondidas" de um objeto, que não podem ser acessadas ou sobrescritas acidentalmente por outra parte do código.

Poe exemplo, suponha que nós estamos trabalhando com objetos `user`, que foram criados em um código de terceiros, e nós queremos adicionar identificadores nesses objetos. Podemos usar um símbolo para isso, como no exemplo abaixo.

```
let user = { // belongs to another code
  name: "John"
};

let id = Symbol("id");
user[id] = 1;
console.log( user[id] ); // 1 ; access data using the symbol as key
console.log( user.id );  // undefined ; property is hidden
console.log( user["id"] );  // undefined ; symbol is not a string
```

Qual é a vantagem de ser usar `Symbol()` ao invés de uma *string*? Como objetos `user` pertencem a outro código, não se deve adicionar quaisquer propriedades a eles, pois isso é bastante inseguro. Mas como um símbolo não pode ser acessado acidentalmente, um código de terceiros provavelmente não vai enxergar esse símbolo, o que torna a sua inclusão menos insegura.

Esse ponto ilustra bem o contraste na filosofia de projeto de LPs. Linguagens como Java adotam uma postura mais conservadora, definindo e garantindo políticas de acesso a campos e métodos através de modificadores como `public` ou `private`. Já JavaScript está no outro ponto do espectro, adotando uma postura bastante relaxada quanto ao encapsulamento. Embora propriedades criadas com símbolos seja consideradas ocultas, isso não é 100% verdade, já que existem métodos em JS como `Object.getOwnPropertySymbols(obj)` que retornam todos os símbolos de um objeto. Essa postura é bastante comum em linguagens dinamicamente tipadas, aonde é dado preferência à liberdade do programador ao invés de garantias mais conservadoras de encapsulamento. Em que lado do espectro cada programador se encaixa vai muito da preferência pessoal. Python segue a mesma filosofia de JS, com a linguagem indicando que a questão de visibilidade (público vs. privado) normalmente é desnecessária. Programadores Python costumam dizer: *"we're all adults here"*.

Símbolos são pulados por `for...in`

Propriedades simbólicas não são consideradas em *loops* `for...in`, como indica o exemplo abaixo.

```
let id = Symbol("id");
let user = {
  name: "John",
  age: 30,
  [id]: 123
};

for (let key in user) console.log(key); // name, age (no symbols)
```

Da mesma forma, o método `Object.keys(user)` também ignora propriedades simbólicas. Por outro lado, `Object.assign` copia tanto as propriedades de *strings* quanto de símbolos, para garantir que um objeto clonado seja igual ao original.

Símbolos globais

Como já visto, símbolos são sempre valores distintos mesmo quando eles possuem a mesma descrição. Mas JavaScript também possui um **registro global de símbolos** (*global symbol registry*), aonde podemos armazenar e recuperar símbolos pela sua descrição (nome).

A chamada `Symbol.for(key)` consulta o registro global, buscando pelo símbolo com a descrição `key`. Se o símbolo existir ele é retornado. Caso contrário, ele é criado com `Symbol(key)`, armazenado no registro e por fim retornado.

```
// read from the global registry
let id = Symbol.for("id"); // if the symbol didn't exist, it's created

// read it again (maybe from another part of the code)
let idAgain = Symbol.for("id");

console.log( id === idAgain ); // true ; the same symbol.
```

1.6 Conversão de objetos para tipos primitivos

A conversão de objetos para tipos primitivos é feita de forma implícita na maioria dos casos em JavaScript. A linguagem permite coisas um pouco estranhas, como um objeto ser convertido para um número, por exemplo. Isso ocorre para simular a sobrecarga de operadores, mas não é recomendado por deixar o código muito obscuro. Na prática, geralmente é suficiente sobrescrever o método `obj.toString()` para se retornar uma *string* com a representação legível do objeto. A definição padrão do método não é muito útil, retornando apenas `[object Object]`.

```
let user = {
  name: "John",
  age: 30
};

console.log( user.toString() ); // [object Object]
console.log( user );            // implicit call to .toString()

user.toString = function () {
  return `User ${this.name} is ${this.age} years old.`;
};

console.log( user.toString() ); // User John is 30 years old.
```

2 Tipos de dados

2.1 Métodos de tipos primitivos

Linguagens que diferenciam entre tipos primitivos e objetos, como Java e JavaScript, acabam criando uma separação e tratamento diferenciado entre as duas categorias nas construções da LP. Isso fica muito aparente nos tipos genéricos de Java, aonde o tipo parametrizado deve necessariamente ser

uma classe. Isto força a realização de *boxing/unboxing* dos tipos primitivos. Por exemplo, em Java não é possível declarar uma lista de inteiros com `ArrayList<int>`, já que `int` é um tipo primitivo. Nesse caso, o tipo primitivo deve ser encaixotado em um objeto da classe `Integer`, levando à declaração `ArrayList<Integer>`.

Python 3, por outro lado, evita essa diferenciação de tipos, tratando todos uniformemente como objetos. Segundo a documentação da linguagem: *"Everything is an Object!"*. Uma implicação imediata disto é que a notação de ponto para invocação de métodos pode ser utilizada para quaisquer valores, inclusive aqueles que normalmente seriam considerados tipos primitivos, como números ou Booleanos.

```
$ python
>>> x = 4.2
>>> x.is_integer()
False
```

JavaScript segue um caminho intermediário entre Java e Python, que é um pouco esquisito mas funciona da seguinte forma.

- Existe uma distinção entre tipos primitivos e objetos, como em Java.
- JS permite o acesso a métodos e propriedades de *strings*, números, Booleanos e símbolos, como em Python.
- Para isso funcionar, JS cria um *"object wrapper"* especial que provê a funcionalidade extra e é destruído em seguida.

Existem *wrappers* para quatro tipos primitivos de JS. Eles são: `String`, `Number`, `Boolean` e `Symbol`, com cada um provendo diferentes métodos conforme cada tipo. Obviamente, os tipos primitivos `undefined` e `null` não possuem *wrappers* associados. Segue um exemplo.

```
let str = "Hello";
console.log( str.toUpperCase() ); // HELLO
```

Eis o que acontece no código acima.

1. A variável `str` contém um valor primitivo. Assim, na hora do acesso ao método `.toUpperCase()`, um objeto especial é criado automaticamente para ser o invocador do método.
2. O método `.toUpperCase()` executa e retorna uma nova *string*, exibida por `console.log`.
3. O *wrapper object* é destruído, restando somente a variável `str` original.

O motor JS tenta otimizar ao máximo esse processo, mas é claro que ele causa um certo *overhead*.

Talvez você esteja se perguntando a diferença entre o processo acima e o *boxing/unboxing* automático de Java. A distinção fundamental é o tratamento dos *wrappers* em cada linguagem. Por exemplo, em Java é perfeitamente razoável se criar um objeto do tipo `Integer`; ele é tratado como qualquer outro objeto. Já em JavaScript você **nunca** deve criar *wrapper objects* manualmente com `new`. Embora a linguagem permita isso, coisas malucas acontecem, como no código abaixo.

```
let zero = new Number(0);
if (zero) { // zero is true, because it's an object
  console.log( "zero is truthy!?! O_o" );
}
```

Cuidado para não se confundir com o uso das funções `String/Number/Boolean` sem o `new`. Como visto no roteiro anterior, neste caso elas servem para realizar conversão de tipos. Por exemplo, o código abaixo é perfeitamente válido.

```
let num = Number("123"); // convert a string to number
```

2.2 Vetores (*arrays*)

Objetos permitem o armazenamento de coleções de valores através de chaves, mas estas coleções não possuem ordem. Por outro lado, para prover uma coleção ordenada, a linguagem possui uma estrutura de dados específica, o `Array`.

Arrays em JavaScript não são nada mais que objetos com uma sintaxe e operações especiais para acesso aos elementos. Sendo assim, esta estrutura não possui muitas semelhanças com os vetores de C ou Java, sendo mais parecida com as listas de Python.

Há duas sintaxes para a criação de *arrays*, com a segunda sendo mais comum porque já permite inicialização. Os índices começam de 0, como na maioria das linguagens.

```
let arr = new Array();
let ay  = [];

let fruits = ["Apple", "Orange", "Banana"];
console.log( fruits[0], fruits[2] ); // Apple Banana
```

A diferença mais fundamental entre *arrays* em JS e C, é que aqui os vetores podem armazenar elementos de qualquer tipo, da mesma forma que as listas de Python. Todo *array* possui a propriedade `length`, que indica o seu tamanho. Além disso, *arrays* possuem uma implementação especializada de `.toString`, que retorna uma *string* com a sequência dos elementos.

```
let array = ["Apple", 42, true];
console.log( array.length ); // 3
console.log( array );        // Apple,42,true
```

Arrays podem ser usados para simular outras estruturas de dados, como pilhas e filas. Por exemplo, a combinação dos métodos `push` e `pop` (que adicionam e removem um elemento do final do *array*, respectivamente) pode ser usada para simular uma pilha. Da mesma forma, a combinação de `unshift` (que adiciona um elemento no começo do *array*) mais `pop` é suficiente para termos uma fila simples. Outras combinações obviamente são possíveis.

```
let fruits = ["Apple"];
fruits.push("Orange");
fruits.unshift("Pineapple");
console.log( fruits ); // Pineapple,Apple,Orange
```

Além das operações indicadas acima, também é possível incluir e remover elementos de um *array* em qualquer posição. Por se tratar de uma estrutura dinâmica, a sua implementação interna pode variar. Em geral o motor JS procura empregar a implementação mais eficiente possível. Por exemplo, se o *array* foi criado já com todos os elementos declarados, o motor **V8** faz o armazenamento em uma área contínua de memória. Por outro lado, se o vetor se tornar esparso, ou houverem muitas inserções e remoções no meio, a estrutura interna pode mudar para uma lista duplamente encadeada ou um *hash*.

Iteração sobre *Arrays*

A sintaxe clássica de C para se percorrer vetores também pode ser usada em JS.

```
let arr = ["Apple", "Orange", "Banana"];
for (let i = 0; i < fruits.length; i++) {
  console.log( fruits[i] );
}
```

Mas há uma forma mais conveniente de iteração, similar a Python.

```
for (let fruit of fruits) {
  console.log( fruit );
}
```

Como *arrays* são tecnicamente objetos, também é possível se usar `for...in` como um *loop* mas isto **não** é indicado porque `for...in` itera sobre **todas** as propriedades do *array*, não somente as numéricas. Assim, a regra geral é: `for...of` para *arrays* e `for...in` para os demais objetos.

Além disso, o método `.forEach` permite executar uma função para cada elemento do *array*. A função pode ter até três argumentos, na ordem `item, index, array`, como mostra o exemplo abaixo.

```
["Bilbo", "Gandalf", "Nazgul"].forEach((item, index, array) => {
  console.log(`${item} is at index ${index} in ${array}`);
});
```

Este tipo de construção em que o *loop* fica implícito tem se tornado bastante comum em LPs imperativas mais novas, aonde podemos ver a inclusão de diversas operações "copiadas" das linguagens funcionais. Por exemplo, o método funcional `map` é um dos mais úteis e utilizados. Ele chama (mapeia) a função do argumento sobre todos os elementos do *array*, retornando um novo vetor com os resultados.

```
let lengths = ["Bilbo", "Gandalf", "Nazgul"].map(item => item.length);
console.log(lengths); // 5,7,6
```

Esse movimento de "funcionalização" de LPs imperativas parece estar acelerando cada vez mais, aonde até mesmo linguagens conservadoras como Java estão incluindo novas funcionalidades (*pun intended! :P*) em suas versões mais recentes.

`Array.isArray`

Como já dito anteriormente, *arrays* não formam um tipo distinto em JavaScript, eles são baseados em objetos. Sendo assim, `typeof` não serve para distinguir um objeto de um *array*.

```
console.log( typeof {} ); // object
console.log( typeof [] ); // same
```

Mas *arrays* são tão comuns que a linguagem provê uma função que faz essa diferenciação.

```
console.log( Array.isArray({}) ); // false
console.log( Array.isArray([]) ); // true
```

Atividade

Esta foi só uma breve introdução das funcionalidades de *arrays* em JS. Existem várias outras que não serão discutidas aqui mas que podem ser vistas em `https://javascript.info/array-methods#summary` e no manual da linguagem (`https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array`). Dê uma passada de olho nessas referências se quiser saber mais.

2.3 Atribuição desestruturante

Acabamos de ver as duas estruturas de dados mais utilizadas em JavaScript: `Object` e `Array`.

Essas estruturas são amplamente empregadas para *agrupar* informações relacionadas. Já para fazer a *separação* dos dados, a linguagem possui uma sintaxe especial de atribuição desestruturante (*destructuring assignment*), que permite "desempacotar" vetores e objetos em várias variáveis.

Desestruturando *arrays*

Eis um exemplo de como um *array* pode ser desestruturado em variáveis.

```
let arr = ["John", "Smith"]
// sets firstName = arr[0] and surname = arr[1]
let [firstName, surname] = arr;
console.log(firstName, surname); // John Smith
```

Note que uma atribuição *desestruturante* não quer dizer *destrutiva*. No programa acima, o vetor `arr` continua existindo normalmente. Esse tipo de atribuição é muito útil quando combinada com o comando `split` que divide uma *string* segundo um delimitador, retornando um vetor.

```
let [firstName, surname] = "John Smith".split(' ');
console.log(firstName, surname); // John Smith
```

A sintaxe é simples mas há vários detalhes peculiares. Vamos ver mais alguns exemplos.

Elementos indesejados do vetor podem ser jogados fora com o uso de vírgulas extras.

```
let [firstName, , title] = ["Julius", "Caesar",
                            "Consul", "of the Roman Republic"];
console.log(title); // Consul
```

Essa mesma construção também pode ser utilizada em *loops*.

```
let user = {
  name: "John",
  age: 30
};

for (let [key, value] of Object.entries(user)) {
  console.log(`${key}:${value}`); // name:John, then age:30
}
```

Um truque muito comum em JS é usar uma atribuição desestruturante para trocar os valores de duas variáveis.

```
let guest = "Jane";
let admin = "Pete";
[guest, admin] = [admin, guest];
console.log(`${guest} ${admin}`); // Pete Jane (successfully swapped!)
```

No código acima um vetor temporário de dois elementos é criado para ser imediatamente desestruturado na atribuição.

Também é possível agrupar o restante de um *array* usando três pontos `...`.

```
let [nameA, nameB, ...rest] =
        ["Julius", "Caesar", "Consul", "of the Roman Republic"];
console.log(rest[0]);       // Consul
console.log(rest[1]);       // of the Roman Republic
console.log(rest.length);   // 2
```

Desestruturando objetos

A atribuição destrutiva também funciona com objetos. No código abaixo as propriedades `options.title`, `options.width` e `options.height` são atribuídas às variáveis correspondentes.

```
let options = {
  title: "Menu",
  width: 100,
  height: 200
};
let {title, width, height} = options;
```

A ordem das variáveis do lado esquerdo não importa. Se a gente quiser atribuir uma propriedade a uma variável com outro nome; por exemplo, fazer `options.width` ser atribuida à variável `w`, basta usar dois pontos.

```
let {width: w, height: h, title } = options;
```

Essa mesma construção também pode ser utilizada para se passar argumentos para uma função. No código abaixo o objeto `options` é passado para a função, que então desestrutura as opções nos parâmetros formais.

```
let options = {
  title: "My menu",
  items: ["Item1", "Item2"]
};

function showMenu({title = "Untitled", width = 200,
                   height = 100, items = []}) {
  // title, items - taken from options
  // width, height - defaults used
  console.log( `${title} ${width} ${height}` ); // My Menu 200 100
  console.log( items ); // Item1,Item2
}

showMenu(options);
```

3 Configurando propriedades de objetos

3.1 Descritores de propriedades

Como já visto, objetos podem armazenar propriedades. Até agora uma propriedade sempre foi um par `key:value`, mas ela pode ser algo diferente. Nesta seção nós vamos estudar opções adicionais de configurações de propriedades, e na próxima seção vamos ver como utilizá-las para criar métodos *getter*/*setter*.

Flags de propriedades

As propriedades de um objeto, além de um valor (`value`), possuem três atributos especiais, chamados de *flags*:

- `writable` – se `true`, o valor pode ser modificado, caso contrário será somente para leitura.
- `enumerable` – se `true`, o valor é listado em *loops*, caso contrário é oculto.
- `configurable` – se `true`, a propriedade pode ser apagada e estes atributos modificados, caso contrário eles ficam fixos na criação da propriedade.

Quando uma propriedade é criada da forma usual vista até aqui, todas as *flags* são `true`. Vamos aprender mais sobre elas. O exemplo abaixo mostra como o conjunto de todas as informações de uma propriedade pode ser consultado.

```
js> let user = {
  name: "John"
};
js> let descriptor = Object.getOwnPropertyDescriptor(user, "name");
js> descriptor
({value:"John", writable:true, enumerable:true, configurable:true})
```

O valor retornado pelo método `getOwnPropertyDescriptor` é um objeto chamado "descritor de propriedade" (*property descriptor*), ele contém o valor e todas as suas *flags*. (O exemplo acima foi feito no *shell* somente porque objetos são mais fáceis de serem exibidos no interpretador.)

Para se alterar as *flags*, utiliza-se o método `defineProperty`.

```
let user = {
  name: "John"
};

Object.defineProperty(user, "name", {
  writable: false
});

user.name = "Pete"; // TypeError: "name" is read-only
```

Um ponto para ficar atento: erros como acima somente são exibidos em modo estrito. Em modo não-estrito, a operação de escrita ainda falha mas nenhum erro é emitido. Esse é só mais um motivo para sempre se usar o modo estrito em todos os *scripts*.

A *flag* de não-configurável (`configurable:false`) costuma ser utilizada para objetos *built-in* e outras constantes globais.

```
js> let descriptor = Object.getOwnPropertyDescriptor(Math, 'PI');
js> descriptor
({value:3.141592653589793, writable:false, enumerable:false,
configurable:false})
```

Cuidado ao tornar uma propriedade não-configurável já que essa operação não pode ser desfeita: a propriedade se torna uma constante "selada para sempre".

3.2 *Getters* e *setters* de propriedades

Existem dois tipos de propriedades de objetos. O primeiro tipo são as propriedades de dados (*data properties*); que correspondem a todas as propriedades vistas até aqui.

Por outro lado, o segundo tipo de propriedade é algo novo, chamadas de propriedades de acesso (*accessor properties*). Essencialmente, elas são funções (métodos) que são executadas ao se buscar ou atribuir um valor, mas que parecem propriedades normais para um código externo.

Getters e *setters*

Propriedades de acesso são representadas por métodos *getters* e *setters*. Em uma declaração de um objeto elas são indicadas pelas palavras reservadas `get` e `set`.

```
let obj = {
  get propName() {
    // getter, the code executed on getting obj.propName
  },
```

```
  set propName(value) {
    // setter, the code executed on setting obj.propName = value
  }
};
```

Como é de se esperar em um cenário OO, o *getter* é invocado quando `obj.propName` for lida, e o *setter* quando ela for atribuída. Segue um exemplo completo.

```
let user = {
  name: "John",
  surname: "Smith",

  get fullName() {
    return `${this.name} ${this.surname}`;
  },

  set fullName(value) {
    [this.name, this.surname] = value.split(" ");
  }
};

console.log(user.fullName); // John Smith
user.fullName = "Alice Cooper";
console.log(user.name, user.surname); // Alice Cooper
```

Como resultado, nós temos agora a propriedade "virtual" `fullName`. Ela pode ser lida e escrita mas seu acesso sempre é feito pela invocação implícita do *getter* e *setter*. Por conta disso, *getters* e *setters* podem ser usados como *wrappers* sobre os valores "reais" das propriedades, na tentativa de se obter mais controle sobre as operações realizadas sobre eles. Por exemplo, podemos fazer a higienização de algum valor passado para um *setter*.

```
let user = {
  get name() {
    return this._name;
  },

  set name(value) {
    if (value.length < 4) {
      console.log("Name is too short!");
      return;
    }
    this._name = value;
  }
};
```

Neste caso, o nome do usuário é armazenado na variável `_name`, e o acesso é feito pelo *getter* e *setter*. Na prática, um código externo pode acessar o nome diretamente usando `user._name`, mas há uma convenção geral de que as propriedades começando com *underscore* são internas e não devem ser tocadas por fora do objeto. Essa convenção também é aplicada em Python e outras linguagens de *script*. Novamente vemos uma grande diferença de filosofia entre LPs, com linguagens como Java oferecendo mecanismos sintáticos muito mais robustos para garantir encapsulamento de objetos. Apesar da maioria dos programadores terem uma certa preferência por uma ou outra filosofia, como sempre o argumento vai depender de uma série de fatores, muitos deles fora do controle do desenvolvedor. Para encerrar esta discussão, vale citar Larry Wall, criador do Perl: *"Perl doesn't have an infatuation with enforced privacy. It would prefer that you stayed out of its living room because you weren't invited, not because it has a shotgun."*

4 Reutilizando uma implementação (vulgo herança)

Em programação OO um dos conceitos fundamentais é o de **herança** (*inheritance*), que permite reutilizar a implementação de entidades similares, com a intenção de evitar duplicação de código (dentre outras). Diferentes LPs empregam diferentes estilos de herança, com os dois tipos principais sendo:

- Herança baseada em **classes** (*class-based programming*, ou *class-orientation*).
- Herança baseada em **protótipos** (*prototype-based programming*, ou *prototype-orientation*).

Em *class-based programming*, a herança ocorre via a definição de *classes* de objetos, ao contrário da herança ocorrendo diretamente via objetos, como é o caso em *prototype-based programming*.

O modelo baseado em classes é o mais popular e desenvolvido de programação OO. Neste modelo, objetos são entidades que combinam *estado* (isto é, dados), *comportamento* (i.e., procedimentos ou métodos) e *identidade* (existência única entre todos os outros objetos) . A estrutura e comportamento de um objeto são definidos por uma *classe*, que serve de "projeto" ou "forma" para todos os objetos de um certo tipo. Um objeto precisa ser criado explicitamente baseando-se em uma classe, e, sendo assim, o objeto é considerando uma *instância* da classe. Linguagens como C++, C#, Java e PHP (dentre várias outras) utilizam o modelo de *class-based programming*.

Por outro lado, no estilo de *prototype-based programming*, a herança é realizada via um processo de reutilização de objetos que servem como *protótipos*. Este modelo também é chamado de programação *classless* ou *instance-based* (dentre outros).

Prototype-based programming utiliza o conceito de objetos generalizados, os quais podem ser então clonados e estendidos. Por exemplo, um objeto generalizado "fruta" pode representar as propriedades e funcionalidades de frutas em geral. Um objeto generalizado "banana" pode então ser clonado do objeto generalizado "fruta" e propriedades gerais de bananas podem ser adicionadas. A partir daí, cada objeto individual "banana" é clonado do objeto generalizado "banana". Compare isso com o modelo *class-oriented*, aonde uma *classe* "fruta" seria estendida por uma *classe* "banana".

A primeira LP com herança baseada em protótipos foi Self, criada durante a década de 1980. A partir do final da década de 1990 este modelo veio crescendo em popularidade. Atualmente, pode-se argumentar que ele é bastante utilizado, já que é o modelo empregado por JavaScript. Uma outra linguagem que também o emprega é Lua.

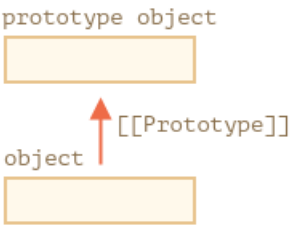
Como já era de se esperar, existe uma certa divergência entre os defensores de cada modelo. Advogados da programação baseada em classes criticam o modelo baseado em protótipos com argumentos similares aos da batalha de tipagem estática vs. dinâmica: da mesma forma que tipagem estática permitiria o desenvolvimento de código mais robusto, o modelo de classes também garantiria um código mais sólido e livre de surpresas em tempo de execução, já que o sistema de tipos do compilador pode detectar estaticamente chamadas de métodos inválidas. Advogados do outro lado argumentam que o modelo de protótipos, assim como tipagem dinâmica, é mais flexível e simples para o programador. Como sempre, a menos que você seja extremista, não há uma resposta absoluta para esta questão, ela vai depender da aplicação e preferência de cada um.

4.1 Herança por protótipos

Como já dito há pouco, JavaScript implementa herança através de protótipos.

Propriedade `[[Prototype]]`

Em JS, objetos possuem a propriedade especial oculta `[[Prototype]]`, que ou é `null` ou é uma referência para um outro objeto, o chamado *objeto protótipo*, como ilustrado na figura a seguir.



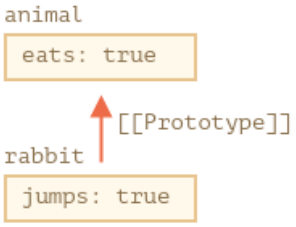
A propriedade `[[Prototype]]` é interna e oculta ao objeto, mas há várias formas distintas de se defini-la. Uma delas é usando o nome `__proto__`, como abaixo.

```
let animal = {
  eats: true
};

let rabbit = {
  jumps: true
};

rabbit.__proto__ = animal; // sets rabbit.[[Prototype]] = animal
```

Agora, se a gente tentar ler uma propriedade faltante de `rabbit`, o interpretador JS vai tomá-la diretamente de `animal`. Isto vale tanto para campos quanto para métodos.

```
console.log( rabbit.eats );  // true
console.log( rabbit.jumps ); // true
```

Neste caso, dizemos que "`animal` é o protótipo de `rabbit`" ou que "`rabbit` *herda prototipicamente* de `animal`". Visualmente, temos a seguinte situação.



É importante destacar que esta herança ocorre diretamente sobre os objetos e não sobre definições de classes. Inclusive, aqui é possível modificar a herança dinamicamente: basta trocar o protótipo do objeto a qualquer hora. Isto claramente não é possível em um linguagem com herança baseada em classes, já que neste caso as heranças são amarradas estaticamente pelo compilador. Certifique-se que você entendeu bem essa diferença, já que até agora você só deve ter visto linguagens com herança baseadas em classes (Java, etc).

Sobre `__proto__`

A propriedade `__proto__` é um *getter/setter* para `[[Prototype]]`. Ela existe por razões históricas mas está um pouco desatualizada. A documentação do JavaScript moderno sugere que sejam usadas as funções `Object.getPrototypeOf` e `Object.setPrototypeOf` para se consultar e definir o protótipo de um objeto, respectivamente. Vamos falar dessas funções mais adiante.

Segundo a especificação, `__proto__` só precisa ser suportada dentro dos navegadores. No entanto, todos os motores JS *stand-alone* também aceitam essa propriedade. Assim, é bastante seguro usá-la em qualquer lugar. Como a notação de `__proto__` é mais concisa e simples, vamos continuar utilizando-a nos exemplos seguintes.

Escrita não utiliza o protótipo

O protótipo é utilizado somente para leitura de propriedades. As operações de escrita e remoção sempre operam diretamente no objeto. No exemplo abaixo, um método `walk` é atribuído diretamente a `rabbit`.

```
let animal = {
  eats: true,
  walk() {
    console.log("Animal walk.");
  }
};

let rabbit = {
  __proto__: animal
};

rabbit.walk = function() {
  console.log("Rabbit! Bounce-bounce!");
};

rabbit.walk(); // Rabbit! Bounce-bounce!
```

Como a chamada `rabbit.walk()` encontra o método diretamente no objeto, este é executado sem usar o protótipo. Neste caso, dizemos que o método `walk()` de `animal` foi sobrescrito (*overwritten*) em `rabbit`.

Uma exceção são as propriedades de acesso, já que uma atribuição é feita por um *setter*. Assim, escrever tal propriedade é equivalente a se chamar uma função. Esta é a razão do código abaixo funcionar.

```
let user = {
  name: "John",
  surname: "Smith",

  set fullName(value) {
    [this.name, this.surname] = value.split(" ");
  },

  get fullName() {
    return `${this.name} ${this.surname}`;
  }
};

let admin = {
  __proto__: user,
  isAdmin: true
};

console.log(admin.fullName); // John Smith - getter triggers!
admin.fullName = "Alice Cooper"; // setter triggers!

console.log(admin.fullName); // Alice Cooper, state of admin modified
console.log(user.fullName); // John Smith, state of user protected
```

O código acima ilustra um outro ponto muito importante: quando os métodos de `user` são executados, os campos do objeto não são alterados, apesar do *setter* usar `this`. Lembra que a

gente disse anteriormente que `this` é amarrado dinamicamente em JS? Pois é, isto quer dizer que `this` não é afetado por protótipos. A regra para se lembrar é a seguinte: **não interessa aonde o método se encontra (em um objeto ou em um protótipo), em uma chamada de método, `this` é sempre o objeto antes do ponto.** Assim, a chamada do *setter* em `admin.fullName=` usa `admin` como `this`, e não `user`. Esse comportamento é essencial em uma linguagem *classless*, já que os protótipos são usados como "armazéns" de métodos. Desta forma, os métodos são compartilhados mas os estados dos objetos não.

O *loop* `for...in`

O *loop* `for...in` itera tanto sobre as propriedades próprias de um objeto quanto as herdadas. Todos os outros métodos para acesso a `key/value` operam somente sobre o objeto.

```
let animal = {
  eats: true
};

let rabbit = {
  jumps: true,
  __proto__: animal
};

// Object.keys only returns own keys
console.log(Object.keys(rabbit)); // jumps

// for...in loops over both own and inherited keys
for(let prop in rabbit) console.log(prop); // jumps, then eats
```

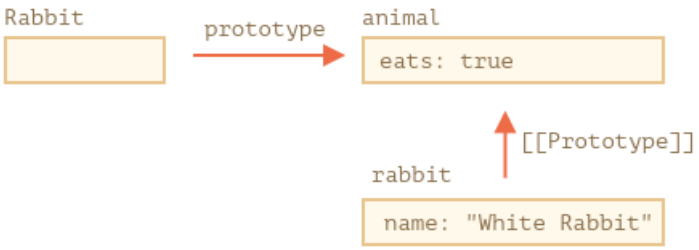
4.2 Propriedade `F.prototype`

Nós já vimos que novos objetos podem ser criados por uma função construtora, como em `new F()`. Se `F` possui a propriedade `F.prototype` definida e referenciando um objeto, então o operador `new` a utiliza para atribuir o `[[Prototype]]` do novo objeto.

Note que `F.prototype` aqui representa uma propriedade comum de `F` com nome `prototype`. Eis um exemplo.

```
let animal = {
  eats: true
};

function Rabbit(name) {
  this.name = name;
}

Rabbit.prototype = animal;

let rabbit = new Rabbit("White Rabbit"); // rabbit.__proto__ == animal

console.log( rabbit.eats ); // true
```

O comando `Rabbit.prototype = animal` faz a seguinte indicação: "quando um `new Rabbit` é criado, atribua ao seu `[[Prototype]]` o objeto `animal`. Visualmente, o resultado fica como ilustrado na figura a seguir.



`F.prototype` padrão

Qualquer função possui a propriedade `prototype`, mesmo quando nós não a indicamos diretamente. O `prototype` padrão é um objeto com uma única propriedade `constructor` que aponta de volta para o própria função.

O código abaixo verifica esta configuração.

```
function Rabbit() {}
// by default:
// Rabbit.prototype = { constructor: Rabbit }

console.log( Rabbit.prototype.constructor === Rabbit ); // true
```

A gente pode usar a propriedade `constructor` para criar um novo objeto usando o mesmo construtor do objeto já existente.

```
function Rabbit(name) {
  this.name = name;
  console.log(name);
}

let rabbitA = new Rabbit("White Rabbit");
let rabbitB = new rabbitA.constructor("Black Rabbit");
```

Isto é útil quando a gente tem um objeto cujo construtor é desconhecido (por exemplo, porque o código está em uma biblioteca), e queremos criar outro objeto do mesmo tipo.

4.3 Protótipos nativos

A propriedade `prototype` é amplamente utilizada pelo núcleo do JavaScript. Todos os construtores `built-in` a utilizam.

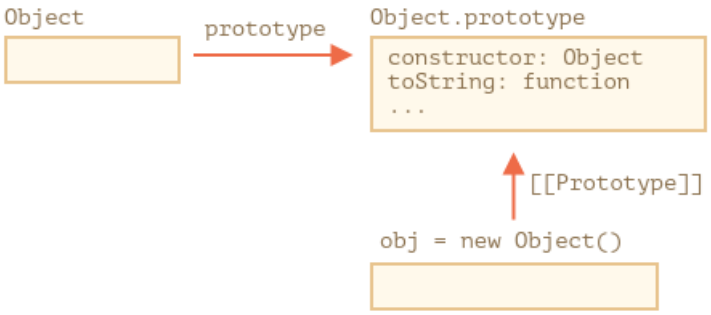
`Object.prototype`

Vamos criar e exibir um objeto vazio.

```
let obj = {};
console.log( obj ); // "[object Object]" ?
```

Aonde está o código do método *built-in* `toString` que retorna `[object Object]`? Ele está no protótipo de `Object`. Vamos ver os detalhes.

A notação `obj = {}` é equivalente a `obj = new Object()`, onde `Object` é um construtor *built-in*, com seu `prototype` referenciando um grande objeto que contém vários métodos, dentre eles o `toString`. Assim, quando o construtor é chamado, o `[[Prototype]]` é definido como `Object.prototype`, conforme a regra já discutida. Visualmente, temos a seguinte configuração.



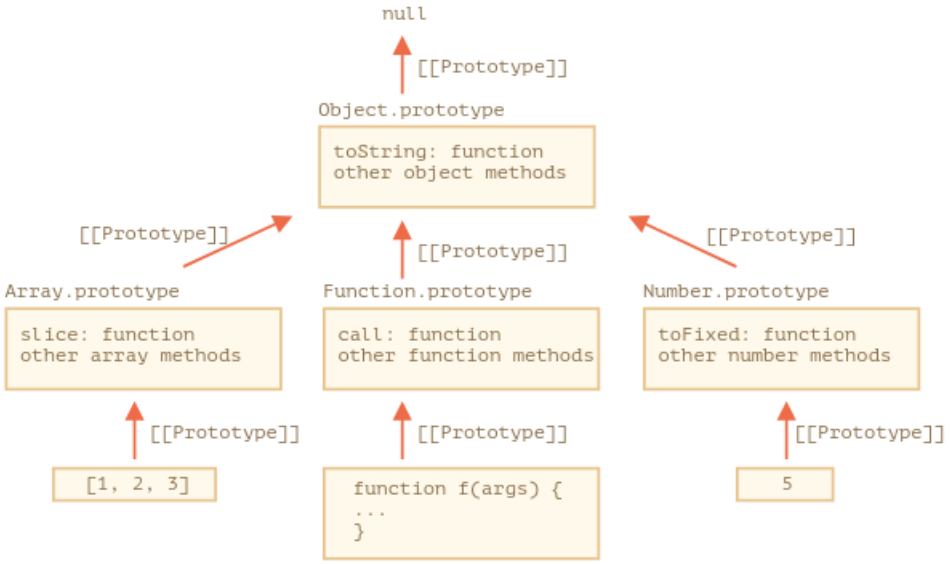
Então, quando `obj.toString()` é chamado, o método usado vem de `Object.prototype`. O código abaixo verifica isto.

```
let obj = {};

console.log(obj.__proto__ === Object.prototype);           // true
console.log(obj.toString  === obj.__proto__.toString);    // true
console.log(obj.toString  === Object.prototype.toString); // true
```

Outros protótipos *built-in*

Outros objetos *built-in* como `Array`, `Function`, etc, também armazenam os métodos nos seus protótipos. Por exemplo, quando criamos um *array* `[1, 2, 3]`, o construtor padrão `new Array()` é usado internamente. Assim, `Array.prototype` vira o protótipo do *array* e provê os seus métodos.

Segundo a especificação, todos os protótipos *built-in* possuem `Object.prototype` no topo da hierarquia de herança. É por isso que se diz que "tudo herda de objeto". Eis uma figura para os *built-ins* `Array`, `Function` e `Number`.



Tipos primitivos

Como já visto na Seção 2.1, os tipos primitivos `string`, `number`, etc, podem ser utilizados como objetos porque *wrappers* são criados sempre que necessário. Os métodos destes objetos *wrappers* também ficam em protótipos, disponíveis em `String.prototype`, `Number.prototype`, etc. Por conta disso, os protótipos destes tipos também podem ser modificados. Por exemplo, no código abaixo nós adicionamos um método ao `String.prototype` que fica disponível para todas as *strings*.

```
String.prototype.show = function() {
  console.log(this);
};

"BOOM!".show(); // BOOM!
```

Obviamente é uma completa insanidade fazer este tipo de coisa no seu código, mas a linguagem te deixa livre para tal. Outras linguagens como Java são bem mais "engessadas" neste sentido, proibindo modificações como acima. Por exemplo, em Java a classe `String` é declarada como `final` para impedir que ela seja herdada por outra classe e modificada indiretamente com sobrescrita de métodos.

Em JavaScript há uma única situação em que se modificar os protótipos de tipos primitivos é considerado adequado: *polyfilling*. Este é o termo usado para se denotar a criação de um método substituto que já existe na especificação de JS, mas que ainda não é suportado por um dado motor da linguagem. *Polyfilling* serve então para se uniformizar o código entre diferentes motores. Este é um aspecto bem mais avançado de uso em JS, então não vamos mais falar disto aqui.

Métodos de protótipos

Foi mencionado anteriormente que `__proto__` é considerado desatualizado e que há outros métodos mais modernos para se definir um protótipo. Estes métodos foram introduzidos no padrão em 2015. Eles são:

- `Object.create(proto, [descriptors])` – cria um objeto vazio com `proto` como `[[Prototype]]` e os demais descritores adicionais.
- `Object.getPrototypeOf(obj)` – retorna o `[[Prototype]]` de `obj`.
- `Object.setPrototypeOf(obj, proto)` – define o `[[Prototype]]` de `obj` como `proto`.

5 Classes

Classes foram introduzidas em JavaScript no padrão de 2015. Apesar do nome, isto não quer dizer que o sistema de herança da linguagem foi alterado para ser baseado em classes. Pelo contrário, ele continua sendo baseado em protótipos, como acabamos de discutir. A declaração de classes de JS pode ser considerada um tipo de **açúcar sintático** (*syntatic sugar* – `https://en.wikipedia.org/wiki/Syntactic_sugar`) da linguagem, incluído para tentar facilitar a vida do programador. Nós vamos esclarecer melhor este ponto adiante.

5.1 Sintaxe básica de classes

A sintaxe básica para se criar uma classe é dada a seguir.

```
class MyClass {
  // class methods
```

```
  constructor() { ... }
  methodA() { ... }
  methodB() { ... }
  methodC() { ... }
  ...
}
```

A seguir basta usar `new MyClass()` para se criar um novo objeto com todos os campos e métodos da classe. O método `constructor()` é chamado *automaticamente* por `new`, então costuma-se inicializar o objeto neste método. Veja um exemplo.

```
class User {
  constructor(name) {
    this.name = name;
  }
  sayHi() {
    console.log(this.name);
  }
}

let user = new User("John");
user.sayHi(); // John
```

Note que, ao contrário da declaração de um objeto literal, na declaração de classe os métodos não ficam separados por vírgula.

O que é uma classe em JS?

Vamos agora tentar entender como as classes em JavaScript funcionam. Como ilustra o código abaixo, uma classe nada mais é do que uma função.

```
class User {
  constructor(name) { this.name = name; }
  sayHi() { console.log(this.name); }
}

console.log(typeof User); // function
```

O construto `class User {...}` faz o motor JS realizar os seguintes passos:

1. Criar uma função `User`, que vira o resultado da declaração da classe. O código da função vem do método `constructor`, considerado vazio se ele não foi explicitamente declarado.
2. Armazenar os métodos da classe, como `sayHi` em `User.prototype`.

Depois que um objeto `new User` é criado, o seu método `sayHi` vem do protótipo, exatamente como já foi explicado anteriormente. O resultado da declaração `class User` fica como abaixo.

Pela figura acima, já deve ter ficado claro a ideia de que a sintaxe de `class` é só uma forma mais conveniente para se criar objetos e seus protótipos. Para reforçar a ideia de que JS continua baseada em protótipos, mesmo após a inclusão do construto `class`, veja o código abaixo que é basicamente equivalente à declaração de classe anterior.

```
// Rewriting class User in pure functions

// 1. Create constructor function
function User(name) {
  this.name = name;
}
// a function prototype has "constructor" property by default,
// so we don't need to create it

// 2. Add the method to prototype
User.prototype.sayHi = function() {
  console.log(this.name);
};

// Usage:
let user = new User("John");
user.sayHi();  // John
```

Apesar do código acima ser equivalente, não é totalmente correto se dizer que o uso de classes é só açúcar sintático em JS, já que há algumas diferenças sutis entre as duas construções acima. No entanto, na maioria dos casos estas diferenças não importam. Nós vamos usar a sintaxe de classes sempre que possível, já que esta é a forma mais moderna recomendada pela especificação da linguagem. Mesmo assim, também é importante conhecer o outro formato, porque muito código JS antigo ainda o utiliza. Mesmo correndo o risco de ser repetitivo: **lembre sempre que mesmo usando `class` a linguagem continua baseada em protótipos**.

5.2 Herança com classes

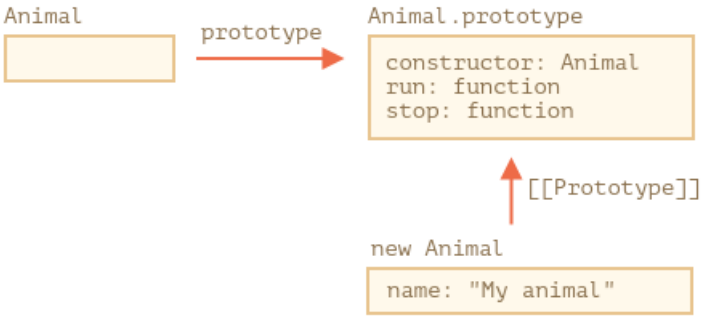
A sintaxe de herança com classes em JavaScript é exatamente igual à sintaxe de Java.

Palavra-chave `extends`

Suponha que nós temos uma classe `Animal`.

```
class Animal {
  constructor(name) {
    this.speed = 0;
    this.name = name;
  }
  run(speed) {
    this.speed = speed;
    console.log(`${this.name} runs with speed ${this.speed}.`);
  }
  stop() {
    this.speed = 0;
    console.log(`${this.name} stands still.`);
  }
}

let animal = new Animal("My animal");
```

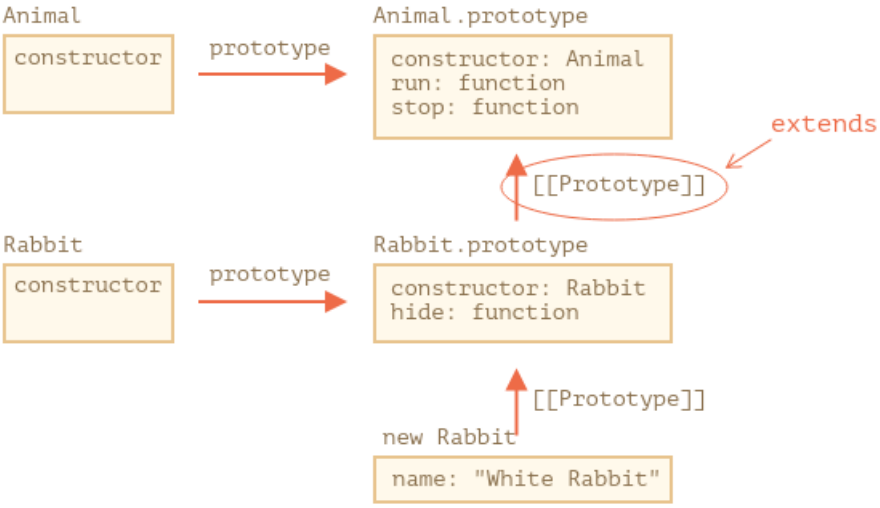
A configuração criada pelo código acima pode ser ilustrada como a seguir.



Como um coelho é um animal, podemos usar herança para criar a classe `Rabbit`.

```
class Rabbit extends Animal {
  hide() {
    console.log(`${this.name} hides!`);
  }
}

let rabbit = new Rabbit("White Rabbit");

rabbit.run(5); // White Rabbit runs with speed 5.
rabbit.hide(); // White Rabbit hides!
```

Internamente, a palavra-chave `extends` funciona com a velha mecânica dos protótipos. Ela define `Rabbit.prototype.[[Prototype]]` como `Animal.prototype`. Assim, se um método não é encontrado em `Rabbit.prototype`, o motor JS procura a implementação em `Animal.prototype`. Visualmente, temos a seguinte configuração.



Sobrescrevendo métodos

Como já visto anteriormente, se a subclasse redeclara um método já existente na superclasse, este é sobrescrito. No entanto, ainda é possível se chamar o método original, através do comando `super`, exatamente como em Java. Vamos modificar o exemplo anterior para que o coelho se esconda sempre que estiver parado.

```
class Rabbit extends Animal {
  hide() { console.log(`${this.name} hides!`); }
```

```
  stop() {
    super.stop(); // call parent stop
    this.hide(); // and then hide
  }
}

let rabbit = new Rabbit("White Rabbit");

rabbit.run(5); // White Rabbit runs with speed 5.
rabbit.stop(); // White Rabbit stands still. White Rabbit hides!
```

Sobrescrevendo construtores

Até agora, `Rabbit` não tem o seu próprio construtor. Segundo a especificação de JS, se uma classe estende outra sem declarar explicitamente um construtor, o seguinte `constructor` vazio é gerado.

```
class Rabbit extends Animal {
  constructor(...args) {
    super(...args);
  }
}
```

Como podemos ver, o código simplesmente chama o construtor da superclasse passando todos os argumentos. Ao criar um construtor explicitamente, você devem sempre lembrar de chamar `super` antes de usar `this` no seu construtor. Note que este comportamento é diferente de Java, que já chama `super()` implicitamente para você.

O exemplo abaixo cria um construtor explícito para `Rabbit` com um campo adicional.

```
class Rabbit extends Animal {

  constructor(name, earLength) {
    super(name);
    this.earLength = earLength;
  }

  // ...
}

let rabbit = new Rabbit("White Rabbit", 10);
console.log(rabbit.name); // White Rabbit
console.log(rabbit.earLength); // 10
```

5.3 Propriedades e métodos estáticos

Também é possível se atribuir um método à própria função da classe, ao invés do seu protótipo. Tais métodos são chamados de estáticos (*static*). Geralmente, métodos estáticos são usados para se implementar funções que são mais caracterizadas pela classe, e não fazem sentido no contexto de objetos individuais. Esta construção é exatamente igual ao Java. Segue um exemplo de uma função de comparação que foi declarada como estática.

```
class Article {
  constructor(title, date) {
    this.title = title;
```

```
    this.date = date;
  }

  static compare(articleA, articleB) {
    return articleA.date - articleB.date;
  }
}

let articles = [
  new Article("HTML", new Date(2019, 2, 1)),
  new Article("CSS", new Date(2019, 1, 1)),
  new Article("JavaScript", new Date(2019, 11, 1))
];

articles.sort(Article.compare);
console.log( articles[0].title ); // CSS
```

Também é possível se criar propriedades estáticas da classe. A decisão de tornar um método ou propriedade estática pode ser subjetiva, dependendo um pouco da aplicação e da preferência de quem está programando.

5.4 Operador `instanceof`

O operador `instanceof` permite testar se um dado objeto pertence a uma certa classe. O seu funcionamento é idêntico ao operador de mesmo nome em Java, respeitando a hierarquia de herança dos protótipos, como mostra o exemplo seguinte.

```
let arr = [1, 2, 3];
console.log( arr instanceof Array ); // true
console.log( arr instanceof Object ); // true
```

5.5 Mixins

Conforme você já tenha inferido pela sintaxe de herança de classes, JavaScript só permite a definição de herança simples, como em Java. No entanto, herança múltipla pode ser simulada através de *mixins*: uma classe que contém métodos que podem ser utilizados por outras classes sem a necessidade de herança. Em outras palavras, um *mixin* provê métodos que implementam algum comportamento, mas o *mixin* não é usado isoladamente, e sim para incluir este comportamento em outros objetos.

A forma mais simples de se implementar um *mixin* em JS é se criar um objeto com os métodos desejados, de forma que estes possam ser fundidos com o protótipo de qualquer classe. Por exemplo, o `sayHiMixin` provê uma forma de "comunicação" para `User`.

```
let sayHiMixin = {
  sayHi() {
    console.log(`Hello ${this.name}!`);
  },
  sayBye() {
    console.log(`Bye ${this.name}`);
  }
};

class User {
  constructor(name) {
```

```
    this.name = name;
  }
}

// copy the methods
Object.assign(User.prototype, sayHiMixin);
// now User can say hi
new User("Dude").sayHi(); // Hello Dude!
```

Note que não há herança, somente cópia de métodos. Assim, `User` pode herdar de outra classe e ainda incluir o *mixin* para trazer os métodos adicionais.

```
class User extends Person {
  // ...
}

Object.assign(User.prototype, sayHiMixin);
```

Um cenário como este pode ser considerado um ponto a favor no argumento para se usar herança baseada em protótipos. Linguagens como Java, que são baseadas em classes mas não possuem herança múltipla, precisam fazer vários malabarismos para simular o comportamento de um *mixin*, através da declaração de interfaces e uso de contêineres. De fato, programadores em LPs sem herança múltipla podem ficar tão preocupados em "queimar a herança", que acabam criando uma ordenação de tipos quase que totalmente plana, usando somente contêineres (objetos que contém outros objetos, ao invés de herdarem deles).

Vale destacar que, ao contrário do que você talvez já tenha ouvido por aí, herança múltipla não é necessariamente algo maligno que deve ser exorcizado de todo código. Pelo contrário, em várias situações é muito mais simples e adequado modelar as classes com herança múltipla. Se não fosse assim, linguagens como C++, OCaml e Python (dentre outras) não implementariam herança múltipla. Como praticamente tudo na vida, se usada com parcimônia e consciência, herança múltipla é muito útil para a vida de um desenvolvedor.

6 Conclusão

Neste segundo roteiro você aprendeu alguns conceitos de OO em JavaScript. No próximo roteiro, vamos ver o funcionamento básico do motor JavaScript dentro de um navegador.

Para você ir se familiarizando com JS, não deixe de desenvolver os programas pedidos no bloco de Atividade da próxima página, utilizando os elementos da linguagem vistos aqui. Todos os exercícios pedidos são triviais para quem já tem um mínimo de conhecimento de programação. Mesmo que você consiga resolver as questões praticamente de cabeça; sente, escreva as suas soluções e teste. Essa é a única forma de se ganhar experiência com uma nova linguagem.

Atividade

Resolva os exercícios abaixo em JavaScript, executando-os no terminal.

1. Desenvolva e teste uma função que recebe como entrada uma matriz com duas dimensões de números e normaliza os seus valores, dividindo-os pelo maior valor da matriz.

2. Utilizando uma representação OO, crie um "novo tipo" `Complex` para armazenar números complexos. Defina as operações de soma e multiplicação destes números, criando tanto métodos normais quanto estáticos. Defina *getters* para os campos e um método `toString` adequado. Além disso, crie valores estáticos para representar os números 0, 1 e i. Teste adequadamente a sua implementação.

3. Utilizando a nova notação de classes de JavaScript, crie uma classe `ContaCorrente` com um campo que armazena o saldo. Utilize um par *getter/setter* para proteger este campo e impedir que o saldo fique negativo. A seguir crie outra classe `ContaEspecial` herdando da anterior. Uma conta especial permite que o saldo seja negativo mas impõe uma taxa de $\%1$ sobre o valor total abaixo de zero, a cada vez que o saldo é alterado para um valor negativo. Modifique o *setter* da subclasse para refletir esta diferença. Teste a sua implementação adequadamente.

4. Crie a seguinte hierarquia de classes:
   - Uma "interface" para representar qualquer forma geométrica, definindo métodos para cálculo do perímetro e cálculo da área da forma.
   - Uma classe "abstrata" para representar quadriláteros. Seu construtor deve receber os tamanhos dos 4 lados, e o método de cálculo do perímetro já pode ser implementado.
   - Classes para representar retângulos e quadrados. A primeira deve receber o tamanho da base e da altura no construtor, enquanto a segunda deve receber apenas o tamanho do lado.
   - Uma classe para representar um círculo. Seu construtor deve receber o tamanho do raio.

   No programa principal, crie um *array* de formas e armazene quadrados, retângulos e círculos neste vetor. A seguir, para cada forma, exiba as seguintes informações:
   (a) os dados (lados ou raio) da forma;
   (b) o perímetro da forma; e
   (c) a área da forma.

   Sempre que possível, tire vantagem de sobrescrita de métodos e polimorfismo, evitando o uso de `instanceof` e *downcast*.

   Obs.: JavaScript não possui uma definição de interfaces e classes abstratas como em Java. Você consegue pensar em alguma forma de simular o comportamento destas estruturas em JS, evitando a criação de instâncias para interfaces e classes abstratas?