Linguagens de Programação
Roteiro de Laboratório 03 – 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 III
Integrando JS no Navegador

Nessa terceira parte vamos mostrar como é feita a integração de código JavaScript dentro dos navegadores. A seguir, vamos concluir o estudo inicial da linguagem construindo um pequeno jogo.

O conteúdo desse roteiro foi adaptado do **The Modern JavaScript Tutorial** (disponível em `https://javascript.info/`), e do tutorial disponível em `https://www.educative.io/blog/javascript-snake-game-tutorial`.

Atividade

Como já indicado nos roteiros anteriores, é importante que você tente reproduzir todos os exemplos do texto para ir pegando familiaridade com a linguagem e as suas ferramentas.

1 JS e HTML

Como já dito no Roteiro 01, o JavaScript foi criado como uma forma de prover mais funcionalidades para páginas *web*. Isto é possível porque *scripts* JS podem ser embutidos diretamente no código HTML das páginas, e estes são executados durante o carregamento do conteúdo no navegador.

1.1 O que pode e não pode ser feito em JS no navegador?

JavaScript no navegador pode fazer qualquer ação relacionada à manipulação de páginas *web*, interação com o usuário e com o servidor. Algumas das ações possíveis incluem:

- Adicionar código HTML a uma página, alterar um conteúdo existente, modificar estilos.
- Reagir a ações do usuário, como cliques do *mouse* e pressionamento de teclas.
- Enviar mensagens pela rede para servidores remotos, fazer *download* e *upload* de arquivos.
- Ler e escrever *cookies*, exibir mensagens, fazer perguntas, etc.

Por outro lado, JS foi pensada como uma LP "segura". Como ela foi criada inicialmente para ser executada somente dentro de navegadores, ela não provê nenhuma forma de acesso à memória ou à CPU, e somente um acesso bem restrito ao SO.

1.2 Incluindo um *script* em uma página

Vamos ver inicialmente como embutir um *script* em uma página HTML.

O *tag* `script`

Programas JavaScript podem ser inseridos em praticamente qualquer parte de um documento HTML usando-se o *tag* `<script>`. Segue um exemplo.

```
<!DOCTYPE HTML>
<html>

<body>

  <p>Before the script...</p>

  <script>
    alert( 'Hello, world!' );
  </script>

  <p>...After the script.</p>

</body>

</html>
```

Teste o exemplo acima salvando todo o código em um arquivo HTML e abrindo no navegador de sua preferência. O *tag* `<script>` contém código JS que é executado automaticamente quando o navegador processa o *tag*. Vamos falar mais sobre `alert` e outros comandos de I/O adiante.

Scripts externos

Um código como do exemplo anterior só é empregado quando o *script* JS é bastante pequeno. Na grande maioria dos casos, *scripts* ficam em arquivos separados e são anexados ao HTML com o atributo `src`.

```
<script src="/path/to/script.js"></script>
```

No código acima, `/path/to/script.js` é o caminho absoluto do *script* a partir da raiz do *site*. Também é possível se empregar um caminho relativo ou uma URL completa.

```
<script src=
"https://cdnjs.cloudflare.com/ajax/libs/lodash.js/4.17.11/lodash.js">
</script>
```

1.3 Console do desenvolvedor

Por padrão, usuários não veem erros no navegador. Para o desenvolvedor poder saber o que está acontecendo e consertar os problemas, os navegadores embutiram diversas ferramentas de desenvolvimento. Estas são bastante elaboradas, com muitas funcionalidades. Vamos ver somente o básico aqui.

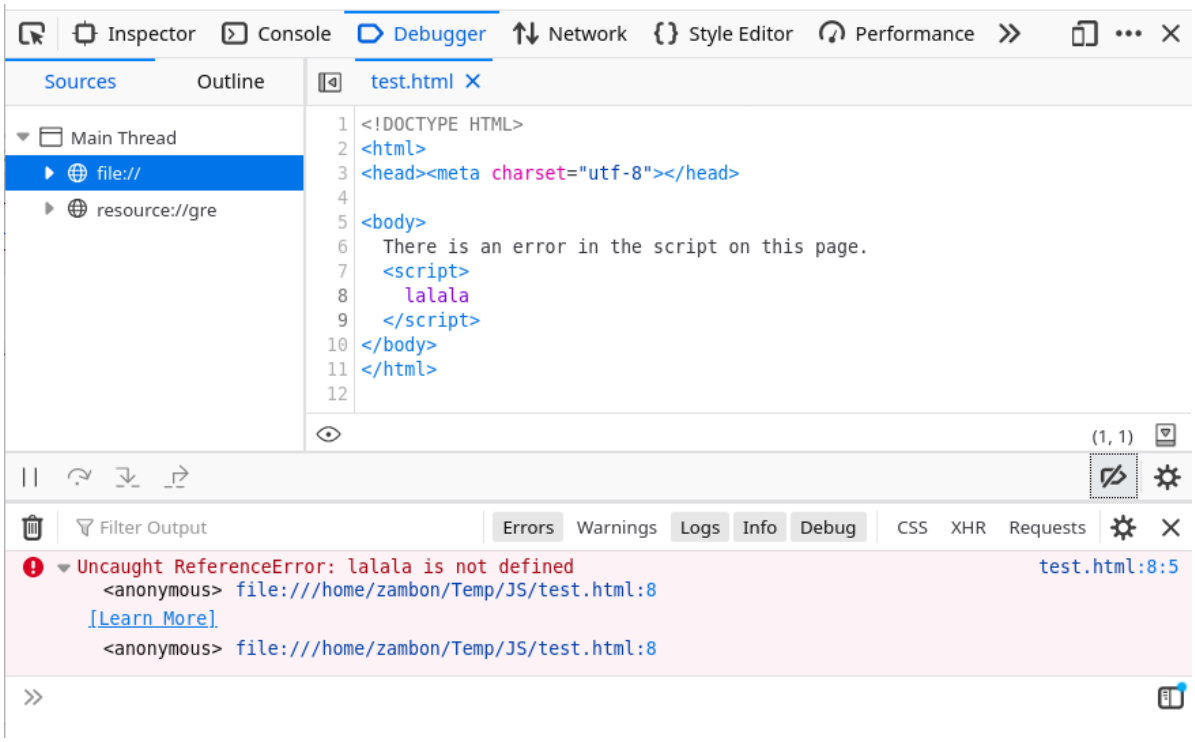
Firefox

Salve o código a seguir em um arquivo HTML e carregue-o no seu navegador preferido. Eu vou usar o Firefox como exemplo, mas outros navegadores (Chrome, etc) também vão funcionar de forma similar.

```
<!DOCTYPE HTML>
<html>
<head><meta charset="utf-8"></head>

<body>
  There is an error in the script on this page.
  <script>
    lalala
  </script>
</body>
</html>
```

Pressione `F12` para abrir as ferramentas de desenvolvimento. Uma janela como abaixo deve aparecer.



Na figura acima é possível ver o erro do interpretador, indicando que na linha 8 o comando `lalala` não é definido. Na parte de baixo da tela há o *prompt* do JS. Podemos digitar e executar comandos por ali.

1.4 Interação com o usuário

Vamos ver algumas funcionalidades do navegador para interação com o usuário.

`alert`

Este já foi visto anteriormente. O comando mostra uma mensagem e espera o usuário pressionar "OK". Segue um exemplo.

```
<script>
  alert("Hello");
</script>
```

A caixa de diálogo que aparece é exibida na forma *modal*. Isto quer dizer que o usuário não pode interagir com o restante da página até ter lidado com a caixa, neste caso, pressionando "OK".

Obs.: A partir de agora não vamos mais exibir o *tag* `script` nos códigos. Sempre que você ver código JS neste texto, lembre que ele precisa estar embutido em uma página HTML diretamente, ou incluído em um arquivo `.js`.

`prompt`

A função `prompt` serve para entrada de dados pelo usuário, ela aceita dois argumentos.

```
result = prompt(title, [default]);
```

O parâmetro `title` é o texto a ser exibido e `default` é um parâmetro opcional com o valor inicial do campo de entrada. A chamada retorna o texto digitado como uma *string* ou `null` se a entrada foi cancelada.

```
let age = prompt('How old are you?', 100);
alert(`You are ${age} years old!`);
```

Teste o código acima entrando com algo e apertando ESC para ver a diferença.

`confirm`

A sintaxe do comando é:

```
result = confirm(question);
```

A função mostra um diálogo modal com a pergunta em `question` e dois botões: "OK" e "Cancel". O resultado é `true` se "OK" for pressionado e `false` caso contrário. Eis um exemplo.

```
let isBoss = confirm("Are you the boss?");
alert( isBoss );
```

`console.log`

O comando `console.log` que vinha sendo usado até aqui continua funcionando no navegador, mas as mensagens agora são enviadas para o console de desenvolvimento.

2 Funcionalidades adicionais de JS

As construções discutidas nesta seção não são exclusivas para JS dentro de navegadores, podendo ser usadas em programas *stand-alone* também. Elas não foram vistas antes por falta de espaço nos roteiros anteriores.

2.1 Métodos JSON

Digamos que a gente tem um objeto complexo, e gostaríamos de convertê-lo para uma *string*, que pode depois ser enviada pela rede ou exibida no console. Obviamente, a *string* deve incluir todas as propriedades que a gente considerar relevantes. Um possível exemplo de implementação pode ser como abaixo.

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

  toString() {
    return `{name: "${this.name}", age: ${this.age}}`;
  }
};

alert(user); // {name: "John", age: 30}
```

O problema é que, durante o processo de desenvolvimento, as propriedades podem ser modificadas e o método `toString` ficar desatualizado. Uma alternativa seria iterar sobre as propriedades do objeto com um *loop*, mas isto é complexo para objetos com propriedades aninhadas. Felizmente este problema já foi resolvido.

JSON.stringify

O formato JSON (*JavaScript Object Notation*) é utilizado para representar valores e objetos, sendo descrito no padrão RFC 4627. Ele foi criado inicialmente para JavaScript mas várias outras linguagens também possuem bibliotecas para manipulá-lo. Assim, é fácil de se realizar uma troca de dados em JSON entre o cliente que usa JavaScript e o servidor que é escrito em Java/PHP/etc.

Os seguintes comandos JSON estão disponíveis em JavaScript:

- `JSON.stringify` converte objetos em JSON.
- `JSON.parse` converte JSON de volta em um objeto.

Por exemplo, eis um uso de `JSON.stringify` sobre um objeto `student`.

```
let student = {
  name: 'John',
  age: 30,
  isAdmin: false,
  courses: ['html', 'css', 'js']
};

let json = JSON.stringify(student);

alert(typeof json); // we've got a string!

alert(json);
/* JSON-encoded object:
{
  "name": "John",
  "age": 30,
  "isAdmin": false,
  "courses": ["html", "css", "js"]
}
*/
```

O método `JSON.stringify(student)` recebe um objeto e o converte para uma *string*. Esta *string* JSON resultante é chamada de objeto *JSON-encoded*, ou *serialized*, ou *stringified*, ou *marshalled*. Vale destacar as diferenças entre o objeto declarado no código JS e a sua codificação JSON.

- *Strings* sempre usam aspas duplas. Aspas simples e *backticks* não são permitidas em JSON.
- Os nomes das propriedades também sempre ficam entre aspas duplas.

JSON é uma especificação independente de LP, somente para dados. Assim, algumas propriedades específicas dos objetos de JS são ignoradas por `JSON.stringify`. Estas incluem:

- Propriedades de funções (métodos).
- Chaves simbólicas e seus valores.
- Propriedades que armazenam o valor `undefined`.

Por conta disto, o resultado final no código abaixo é uma *string* vazia.

```
let user = {
  sayHi() { // ignored
    alert("Hello");
  },
  [Symbol("id")]: 123, // ignored
  something: undefined // ignored
};

alert( JSON.stringify(user) ); // {} (empty object)
```

Objetos aninhados são suportados e convertidos automaticamente.

```
let meetup = {
  title: "Conference",
  room: {
    number: 23,
    participants: ["john", "ann"]
  }
};

alert( JSON.stringify(meetup) );
/* The whole structure is stringified:
{
  "title":"Conference",
  "room":{"number":23,"participants":["john","ann"]},
}
*/
```

A única limitação é que não podem haver referências circulares no(s) objeto(s).

Customizando JSON

Assim como o método `toString` de conversão para *strings*, um objeto também pode prover o método `toJSON` para sua conversão no formato JSON. O comando `JSON.stringify` sempre chama este método quando ele existe. Segue um exemplo.

```
let room = {
  number: 23,
  toJSON() {
    return this.number;
  }
};
```

```
let meetup = {
  title: "Conference",
  room
};

alert( JSON.stringify(room) ); // 23
alert( JSON.stringify(meetup) );
/*
  {
    "title":"Conference",
    "room": 23
  }
*/
```

Compare a saída JSON com a do exemplo anterior para ver a diferença na codificação de `room`.

Comando `JSON.parse`

Para se decodificar uma *string* JSON, utiliza-se o comando `JSON.parse`. A sintaxe é:

```
let value = JSON.parse(str, [reviver]);
```

aonde `str` é a *string* a ser decodificada, e `reviver` é uma função opcional que é chamada para cada par `(key, value)`, ficando encarregada de transformar o valor. Eis um exemplo simples.

```
let userData = '{ "name": "John", "age": 35, "friends": [0,1,2,3] }';
let user = JSON.parse(userData);
alert( user.friends[1] ); // 1
```

2.2 Escalonamento de execuções

Em JavaScript é possível se *escalonar a chamada* de uma função, de forma que ela seja executada após um certo tempo determinado. Há dois métodos para se fazer isto.

- `setTimeout` permite executar uma função uma única vez após um dado valor de tempo.
- `setInterval` permite executar uma função repetidamente em um dado intervalo de tempo.

Estes métodos não fazem parte da especificação do JavaScript mas são suportados por todos os navegadores e também alguns interpretadores *stand-alone*, como `Node.js` e `js78`.

Método `setTimeout`

A sintaxe do comando `setTimeout` é:

```
let timerId = setTimeout(func|code, [delay], [argA], [argB], ...)
```

Com os parâmetros:

- `func|code` – uma função ou *string* de código para ser executada. Geralmente se utilizam somente funções. Por razões históricas, uma *string* de código é aceita, mas seu uso não é mais recomendado em JS moderno.
- `delay` – o período de tempo antes da execução, em milissegundos. O valor padrão é 0.
- `argA, argB, ...` – os argumentos para a função.

Por exemplo, o código baixo executa `sayHi()` após um segundo.

```
function sayHi(phrase, who) {
  alert( phrase + ', ' + who );
}
setTimeout(sayHi, 1000, "Hello", "John"); // Hello, John
```

Método `setInterval`

A sintaxe de `setInterval` é a mesma de `setTimeout`, e todos os argumentos possuem o mesmo significado. A única diferença está na semântica: enquanto `setTimeout` executa a função uma única vez, `setInterval` executa regularmente entre um certo intervalo de tempo. Para terminar as chamadas, devemos executar `clearInterval(timerId)`. O exemplo abaixo mostra uma mensagem a cada 2 segundos. Após 5 segundos a saída é terminada.

```
// repeat with the interval of 2 seconds
let timerId = setInterval(() => alert('tick'), 2000);

// after 5 seconds stop
setTimeout(() => { clearInterval(timerId); alert('stop'); }, 5000);
```

2.3 Exceções

O tratamento de exceções em JavaScript possui a mesma sintaxe e semântica de Java. A sintaxe geral é dada abaixo, e provavelmente já deve ser familiar se você conhece Java.

```
try {
   ... attempt to execute the code ...
} catch(e) {
   ... handle errors ...
} finally {
   ... execute always ...
}
```

O código abaixo mostra um exemplo de uso de exceções para decodificação de *strings* JSON. Caso uma *string* `json` é mal-formada, o método `JSON.parse` gera um erro que "mata" o *script*. Assim, precisamos de um bloco de `try ... catch`.

```
let json = "{ bad json }";

try {
  let user = JSON.parse(json); // <-- when an error occurs...
  alert( user.name ); // doesn't work
} catch (e) {
  // ...the execution jumps here
  alert( "Our apologies, the data has errors..." );
  alert( e.name );
  alert( e.message );
}
```

Para maiores informações sobre o tratamento de exceções em JS, veja `https://javascript.info/try-catch`.

3 Estrutura do navegador

Vamos agora passar para a estrutura interna dos navegadores, e aprender como podemos manipular as páginas usando JavaScript.

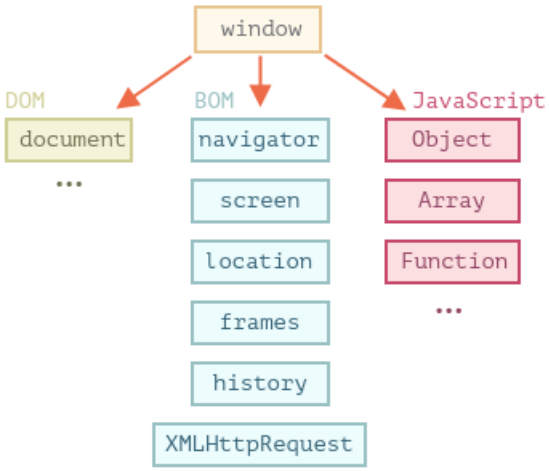
3.1 O ambiente do navegador

A linguagem JavaScript foi criada inicialmente para navegadores *web*. Desde então, ela tem evoluído e se tornou uma linguagem com vários casos de uso e plataformas.

Uma plataforma pode ser um navegador, ou um servidor *web*, ou um outro *host* na rede, ou qualquer outra máquina que consegue rodar JS. Cada uma destas provê funcionalidades que são específicas da plataforma (*platform-specific*). A especificação do JS chama isto de ambiente hospedeiro (*host environment*),

Um *host environment* provê os seus próprios objetos e funções além das já existentes no núcleo da linguagem. Navegadores fornecem meios para se controlar páginas *web*, `node.js` provê funcionalidades para servidores, etc.

A figura abaixo mostra uma visão da estrutura quando se roda JavaScript em um navegador.



Há um objeto raiz chamado `window`. Ele possui duas funções:

1. Servir como um objeto global para todo código JS.
2. Representar a janela do navegador, e prover métodos para controlá-la.

Por exemplo, no código abaixo, `window` é usada como o objeto global.

```
function sayHi() {
  alert("Hello");
}
window.sayHi(); // Global functions are methods of the global object.
```

E aqui ela é usada como a janela do navegador, para lermos a sua altura em pixels.

```
alert(window.innerHeight);
```

Há vários outros métodos e propriedades específicos de `window`. Vamos falar de mais alguns deles adiante.

DOM (*Document Object Model*)

O DOM (*Document Object Model*) representa todo o conteúdo de um página como objetos, para que eles possam ser manipulados.

O objeto `document` é o "ponto de entrada" para a página. Podemos modificar ou criar qualquer coisa na página usando este ponto. Por exemplo, o código abaixo muda a cor de fundo da janela.

```
// change the background color to red
document.body.style.background = "red";

// change it back after 1 second
setTimeout(() => document.body.style.background = "", 1000);
```

As várias propriedades e métodos do DOM estão descritas na especificação: **DOM Living Standard** (`https://dom.spec.whatwg.org/`).

BOM (*Browser Object Model*)

O BOM (*Browser Object Model*) representa os objetos adicionais fornecidos pelo navegador (*host environment*). Por exemplo:

- O objeto `navigator` provê informações adicionais sobre o navegador e o SO. Há várias propriedades, mas as duas mais usadas são: `navigator.userAgent`, para se identificar o navegador; e `navigator.platform`, para se diferenciar entre Windows/Linux/Mac, etc.
- O objeto `location` nos permite ler a URL atual e redirecionar o navegador para um novo local.

Eis um exemplo de uso do objeto `location`.

```
alert(location.href); // shows current URL
if (confirm("Go to Wikipedia?")) {
  location.href = "https://wikipedia.org"; // redirect to another URL
}
```

As funções `alert/confirm/prompt` também fazem parte do BOM, já que elas são métodos do navegador para comunicação com o usuário. A especificação do BOM está incluída no padrão do HTML, podendo ser vista em `https://html.spec.whatwg.org/`.

É importante gravar os *links* acima para referência, já que vamos ver somente as características mais básicas do seu conteúdo. Vamos começar agora a aprender sobre o DOM.

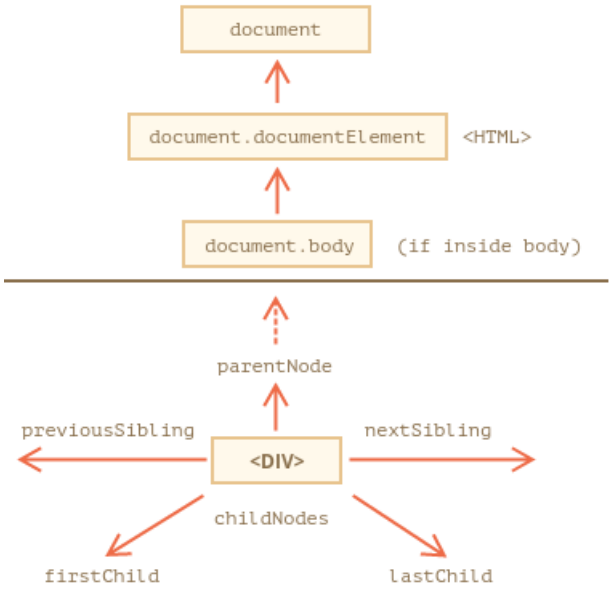
3.2 Árvore DOM

Um documento HTML é composto por várias *tags*. Segundo o DOM, toda *tag* HTML é um objeto. *Tags* aninhadas são filhas das *tags* externas, formando a árvore do documento (*DOM tree*). Todos estes objetos são acessíveis pelo JavaScript, e podemos usá-los para modificar a página.

Por exemplo, `document.body` é o objeto associado à *tag* `<body>`. Já foi mostrado um código que faz uma atribuição ao `style.background` de `document.body` para mudar a cor de fundo.

O DOM nos permite fazer qualquer coisa com elementos e os seus conteúdos, mas primeiro é necessário se acessar o objeto DOM correspondente.

Todas as operações no DOM começam com o objeto `document`, que é a raiz do modelo. A partir de `document` podemos acessar qualquer outro nó. Eis uma figura que mostra alguns dos *links* para acesso entre nós do DOM.



Vamos falar um pouco agora de alguns dos elementos desta figura.

Parte superior: `documentElement` e `body`

Os três nós superiores ficam disponíveis diretamente como propriedades de `document`:

- `document.documentElement` – nó DOM para o *tag* `<html>`.
- `document.body` – nó DOM para o *tag* `<body>`.
- `document.head` – nó DOM para o *tag* `<head>`.

Parte inferior: filhos

A coleção `childNodes` contém todos os nós filhos de um nó. O exemplo abaixo mostra os filhos de `document.body`.

```
<html>
<body>
  <div>Begin</div>

  <ul>
    <li>Information</li>
  </ul>

  <div>End</div>

  <script>
    for (let i = 0; i < document.body.childNodes.length; i++) {
      alert( document.body.childNodes[i] ); // Text, DIV, Text, UL, ...
    }
  </script>
  ...more stuff...
</body>
</html>
```

Note que o *loop* `for` apresenta mais coisas que os *tags* no documento HTML acima. Isto ocorre porque elementos adicionais, como quebras de linhas, comentários, etc, também entram no DOM. De fato, é possível se reproduzir o documento HTML inteiro pelo DOM.

As propriedades `firstChild` e `lastChild` são simplificações para o acesso ao primeiro e último filho de um nó, respectivamente. Também há a função `elem.hasChildNodes()` para testar se um nó possui filhos.

Coleções no DOM

No exemplo anterior, `childNodes` parece ser um *array*, mas na verdade é uma *coleção (collection)*. Uma consequência desta diferença é que métodos de *array* não funcionam diretamente.

```
alert(document.body.childNodes.filter); // undefined, no filter method!
```

Se a gente precisar de fato de um *array*, basta usar `Array.from` sobre uma coleção.

```
alert( Array.from(document.body.childNodes).filter ); // function
```

Uma similaridade entre coleções e *arrays* é que devemos sempre usar `for ... of` para iterar sobre ambos. Não use `for ... in`!

Coleções do DOM são somente para leitura

Todas as coleções do DOM e demais propriedades de navegação listadas aqui são somente para leitura. Isto quer dizer que não é possível se substituir um filho fazendo somente uma atribuição como `childNodes[i] = ...`. Vamos ver algumas formas de modificar o DOM adiante.

Nós irmãos e o nó pai

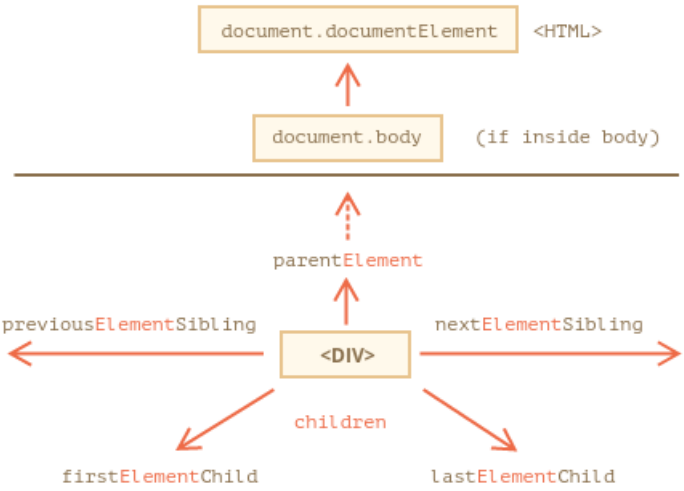
Nós irmãos (*siblings*) são filhos de um mesmo nó pai (*parent*). Por exemplo, no documento abaixo, `<head>` e `<body>` são irmãos.

```
<html>
  <head>...</head><body>...</body>
</html>
```

Nós irmãos são conectados como uma lista duplamente encadeada, com as propriedades `nextSibling` e `previousSibling` permitindo a navegação na lista. O nó pai é acessível pela propriedade `parentNode`.

Navegando somente pelos elementos relevantes

As propriedades de navegação referenciam todos os nós do DOM. Por exemplo, em `childNodes` podemos acessar tanto nós de elementos HTML (*tags*), como nós de texto, e até mesmo nós de comentários, se eles existirem. No entanto, para a maioria dos casos, estes últimos nós não são necessários. Para manipular somente os nós de elementos que formam a estrutura da página temos outros *links* de navegação como ilustrado na figura a seguir.



Alguns tipos de elementos do DOM (por exemplo, tabelas) fornecem propriedades e coleções adicionais para acesso ao seu conteúdo.

3.3 Buscando elementos no DOM

Além dos métodos de navegação já vistos, também é possível fazer uma busca por elementos arbitrários na página. Se um elemento HTML possui o atributo `id`, podemos acessar o seu objeto no DOM com o método `document.getElementById(id)`, como mostra o exemplo abaixo.

```
<div id="elem">
  <div id="elem-content">Element</div>
</div>

<script>
  // get the element
  let elem = document.getElementById('elem');

  // make its background red
  elem.style.background = 'red';
</script>
```

Além disso, há uma variável global com o nome de `id` que referencia o elemento.

```
<script>
// elem is a reference to DOM-element with id="elem"
elem.style.background = 'red';

//id="elem-content" has a hyphen inside, so it can't be a variable name
//...but we can access it using square brackets: window['elem-content']
</script>
```

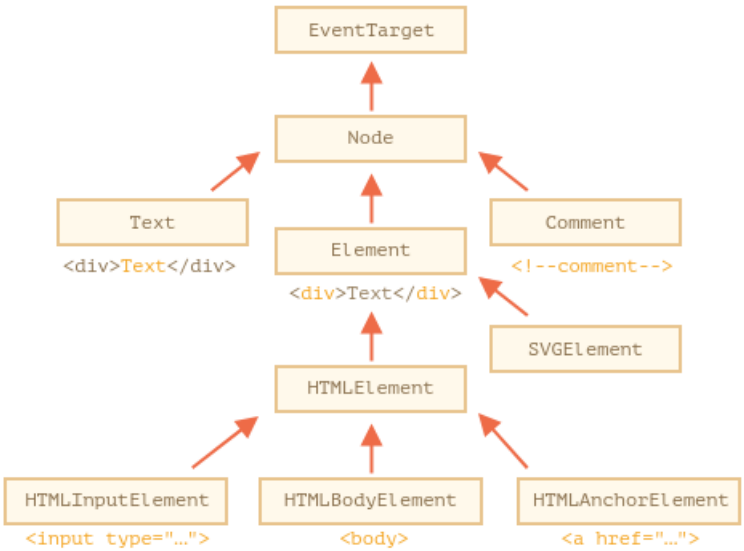
Um código como acima não é recomendável porque ele pode poluir o *namespace* e levar a conflitos de nomes. Além disso, quando alguém está lendo o código JS sem olhar o HTML, não fica óbvio de onde a variável está vindo. Por conta disto, usar `document.getElementById` é preferível.

3.4 Propriedades dos nós: tipo, *tag* e conteúdo

Nesta seção vamos falar de mais alguns detalhes dos nós do DOM, vendo como eles são implementados e as suas propriedades principais.

Classes de nós

Diferentes nós do DOM possuem diferentes propriedades, mas também podem compartilhar outras propriedades em comum. Sendo assim, a implementação dos nós em JavaScript utiliza OOP, com cada nó do DOM pertencendo a uma classe, como ilustra a figura abaixo.



Seguem alguns comentários sobre as classes da figura.

- `EventTarget` é a raiz da hierarquia de classes. Esta é uma classe abstrata, servindo para agrupar as funcionalidades de *eventos*, que serão vistos adiante.
- `Node` também é uma classe abstrata, servindo de base para os nós do DOM. Ela provê as funcionalidades básicas da árvore, como as propriedades de navegação `childNodes`, etc.
- `Element` é a classe base para todos os elementos do DOM, fornecendo funcionalidades de navegação e de busca. Como os navegadores também suportam os formatos XML e SVG além de HTML, esta classe é especializada em `SVGElement`, `HTMLElement`, etc.

Há várias formas de se descobrir a classe de um nó. O código abaixo ilustra três delas: usar a propriedade do `constructor`, ou converter o objeto com `toString`, ou ainda fazer testes com `instanceof`.

```
alert( document.body.constructor.name ); // HTMLBodyElement

alert( document.body ); // [object HTMLBodyElement]

alert( document.body instanceof HTMLBodyElement ); // true
alert( document.body instanceof HTMLElement ); // true
alert( document.body instanceof Element ); // true
alert( document.body instanceof Node ); // true
alert( document.body instanceof EventTarget ); // true
```

Conteúdo em `innerHTML`

A propriedade `innerHTML` nos permite acessar e modificar o HTML dentro de um elemento como uma *string*, sendo uma das formas mais diretas de se alterar a página. O exemplo a seguir exibe o conteúdo de `document.body` e a seguir o modifica por inteiro.

```
<body>
  <p>A paragraph</p>
  <div>A div</div>

  <script>
    alert( document.body.innerHTML ); // read the current contents
    document.body.innerHTML = 'The new BODY!'; // replace it
  </script>
</body>
```

Outras propriedades

Elementos do DOM possuem várias propriedades adicionais, geralmente dependentes da classe do elemento. A maioria dos atributos HTML ficam associados a uma propriedade no DOM. Consulte a especificação sempre que quiser ver a lista completa de propriedades suportadas para uma certa classe. Por exemplo, `HTMLElement` está documentada em `https://html.spec.whatwg.org/#htmlelement`.

3.5 Modificando o documento

Nesta seção vamos ver como criar novos elementos na página e modificar os já existentes.

Criando um elemento

Há duas formas para se criar nós no DOM, com a primeira sendo a mais comum.

```
// Creates a new element node with the given tag.
let div = document.createElement('div');

// Creates a new text node with the given text.
let textNode = document.createTextNode('Here I am');
```

Criando uma mensagem de destaque

Vamos ver como criar uma mensagem de destaque sem usar o comando `alert`. Os passos estão indicados nos comentários.

```
<body>
<!-- 0. Create the rendering information -->
<style>
.alert {
  padding: 15px;
  border: 1px solid #d6e9c6;
  border-radius: 4px;
  color: #3c763d;
  background-color: #dff0d8;
}
</style>

<script>
// 1. Create <div> element
let div = document.createElement('div');
```

```
// 2. Set its class to "alert"
div.className = "alert";

// 3. Fill it with the content
div.innerHTML = "<strong>Hi there!</strong> This is important!";

// 4. Insert it somewhere into document
document.body.append(div);
</script>
</body>
```

O resultado da página fica como na figura abaixo.

Remoção de nós

Para se remover um nó basta usar `node.remove()`. Inclua a linha abaixo no exemplo anterior para fazer a mensagem desaparecer após um segundo.

```
setTimeout(() => div.remove(), 1000);
```

3.6 Introdução aos eventos do navegador

Um *evento* é um sinal de que algo aconteceu. Todos os nós do DOM podem gerar estes sinais, mas eventos não estão restritos somente ao DOM.

Para informação, segue uma lista dos eventos do DOM mais úteis/comuns.

- **Eventos do mouse**:
  - `click` – gerado quando o mouse clica um elemento (também corresponde a um toque em um dispositivo *touch-screen*).
  - `contextmenu` – gerado quando há um clique com o botão direito em um elemento.
  - `mouseover` / `mouseout` – gerado quando o mouse entra ou sai de um elemento.
  - `mousedown` / `mouseup` – gerado quando o botão do mouse é pressionado ou solto sobre um elemento.
  - `mousemove` – gerado quando o mouse é movido.
- **Eventos do teclado**:
  - `keydown` / `keyup` – gerado quando uma tecla é pressionada ou solta.
- **Eventos de entrada**:
  - `submit` – gerado quando o usuário submete um `<form>`.
  - `focus` – gerado quando o usuário foca em um elemento de entrada, como `<input>`.

Para reagir a eventos, podemos criar tratadores (*handlers*) – funções que são executadas quando um evento ocorre. Há várias formas de se associar um *handler* a um evento.

Handlers em atributos HTML

Um *handler* pode ser definido diretamente em HTML com um atributo `on<event>`. Por exemplo, para atribuir a `click` um *handler* para um `input`, podemos usar `onclick`, como no exemplo abaixo.

```
<input value="Click me" onclick="alert('Click!')" type="button">
```

Quando um clique do mouse ocorre, a *string* com o código em `onclick` é executada. Um atributo HTML não é exatamente o lugar mais conveniente para se escrever código, assim geralmente se cria uma função em JS que é associada no *tag*.

```
<script>
  function countRabbits() {
    for(let i=1; i<=3; i++) {
      alert("Rabbit number " + i);
    }
  }
</script>

<input type="button" onclick="countRabbits()" value="Count rabbits!">
```

Handlers em propriedades do DOM

Também é possível se atribuir um *handler* usando alguma propriedade do DOM, como por exemplo, `onclick`.

```
<input id="elem" type="button" value="Click me">
<script>
  elem.onclick = function() {
    alert('Thank you');
  };
</script>
```

Comandos `addEventListener` e `removeEventListener`

O problema fundamental das formas anteriores para atribuição de *handlers* é a impossibilidade de se associar múltiplos *handlers* para um mesmo evento. Esta restrição é obviamente inviável em aplicações reais. Assim, existem os métodos `addEventListener` e `removeEventListener` para resolver este problema.

A sintaxe para se adicionar um *handler* é:

```
element.addEventListener(event, handler, [options]);
```

aonde `event` é o nome do evento para o qual queremos registrar a função `handler`. O parâmetro `options` indica algumas opções adicionais. Uma delas que vale destaque é `once`, que quando é verdadeira faz o *listener* ser removido após ser executado uma única vez.

A sintaxe de `removeEventListener` possui exatamente os mesmos parâmetros.

```
element.removeEventListener(event, handler, [options]);
```

O exemplo abaixo mostra como várias chamadas de `addEventListener` são usadas para adicionar múltiplos *handlers*.

```
<input id="elem" type="button" value="Click me"/>
<script>
  function handlerA() {
    alert('Thanks!');
  };
  function handlerB() {
    alert('Thanks again!');
  }
```

```
  elem.onclick = () => alert("Hello");
  elem.addEventListener("click", handlerA); // Thanks!
  elem.addEventListener("click", handlerB); // Thanks again!
</script>
```

Objetos de eventos

Para poder lidar adequadamente com um evento, geralmente são necessárias mais informações além de que simplesmente o evento ocorreu. Por exemplo, no caso do clique do mouse, quais foram as coordenadas? Se o teclado foi pressionado, qual foi a tecla?

Quando um evento ocorre, o navegador cria um *objeto do evento*, coloca os detalhes dentro dele e passa como argumento para o *handler*. Eis um exemplo que mostra como obter as coordenadas do cursor pelo objeto do evento.

```
<input type="button" value="Click me" id="elem">

<script>
  elem.onclick = function(event) {
    // show event type, element and coordinates of the click
    alert(event.type + " at " + event.currentTarget);
    alert("Coordinates: " + event.clientX + ":" + event.clientY);
  };
</script>
```

No código acima, `type` é o tipo do evento, no caso, `click`; `currentTarget` é o elemento que tratou o evento; e `clientX` e `clientY` são as coordenadas relativas do cursor na janela.

Diferentes tipos de eventos possuem diferentes propriedades. Vamos ver mais algumas delas para o mouse e o teclado adiante.

Ações padrão do navegador

Muitos eventos levam o navegador a realizar certas ações automaticamente. Por exemplo, um clique em um *link* inicia a navegação para a sua URL; pressionar e arrastar o mouse sobre um texto faz uma seleção, etc.

Quando estamos tratando um evento em JS, é possível que a gente não queria que o navegador realize a sua ação padrão. Há duas formas de se indicar isto.

- Através do objeto do evento, pelo método `event.preventDefault()`.
- Retornando `false`, se o *handler* foi atribuído usando `on<event>` (não por uma chamada a `addEventListener`).

No exemplo abaixo, os cliques nos *links* não ativam a navegação.

```
<a href="/" onclick="return false">Click here</a>
or
<a href="/" onclick="event.preventDefault()">here</a>
```

Eventos do mouse

Vamos ver mais alguns detalhes dos eventos do mouse. Por exemplo, todos eles incluem informações sobre as teclas modificadoras (Alt, Ctrl, Shift, etc) que foram pressionadas – as propriedades `altKey`, `ctrlKey` ou `shiftKey` são `true` neste caso.

O botão do código a seguir só funciona com `Alt+Shift+click`.

```
<button id="button">Alt+Shift+Click on me!</button>

<script>
  button.onclick = function(event) {
    if (event.altKey && event.shiftKey) {
      alert('Hooray!');
    }
  };
</script>
```

Para as coordenadas, todos os eventos do mouse fornecem dois tipos:

1. Relativas à janela: `clientX` e `clientY`.
2. Relativas ao documento: `pageX` e `pageY`.

As coordenadas (`pageX/Y`) são contadas a partir do canto superior esquerdo do documento, e não mudam quando a página é rolada (*scrolled*). Por outro lado, `clientX/Y` são contadas a partir do canto superior esquerdo da janela atual, e são afetadas por *scroll*.

Eventos do teclado

Eventos do teclado são utilizado quando queremos tratar o pressionamento de teclas, inclusive em teclados virtuais.

As duas propriedades principais dos objetos para eventos do teclado são `event.key`, que contém o caractere digitado; e `event.code`, que corresponde ao código físico da tecla. Vale destacar que `key` é afetada por teclas modificadoras mas `code` não. Por exemplo, a mesma tecla `Z` pode ser pressionada com ou sem `Shift`, o que gera dois caracteres diferentes em `event.key`: `z` ou `Z`. No entanto, em ambos os casos `event.code` é o mesmo: `KeyZ`, que em teclados QWERTY possui o valor 90.

Aprendendo mais sobre JS+HTML

Tudo o que foi discutido até aqui é só uma breve introdução sobre o uso de JavaScript em páginas *web*, mas este texto obviamente não é o local adequado para se entrar em maiores detalhes. Sendo assim, as demais partes serão deixadas para você buscar/estudar como preferir. Como o conteúdo é muito extenso, a melhor forma de se aprender programação *web* é por demanda: defina primeiro *o que você quer fazer* e depois busque *como fazer* na documentação. A segunda parte do tutorial em `https://javascript.info/` é um bom ponto de partida, bem como a documentação da linguagem em `https://developer.mozilla.org/en-US/docs/Web/JavaScript`.

4 Juntando tudo: o jogo *Snake*

A melhor (ou até mesmo a única) forma de se aprender uma nova LP é através de um projeto de programação. Nesta seção vamos tentar agrupar o conteúdo visto até aqui fazendo o projeto de um jogo simples. Vamos usar como exemplo o jogo Serpente (*Snake*), que ficou muito popular por conta dos celulares Nokia antigos. Veja mais informações sobre o jogo em `https://en.wikipedia.org/wiki/Snake_(video_game_genre)`.

4.1 Construindo os elementos gráficos

Vamos começar elaborando os elementos gráficos do jogo. Precisamos de um retângulo 2D para a área de navegação, aonde vamos desenhar a serpente e a comida. Além disso, queremos mostrar a

pontuação do jogador. Salve o código abaixo em um arquivo HTML e carregue-o no seu navegador preferido.

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="utf-8"/>
        <title>Snake Game</title>
    </head>

    <body>
        <div id="score">0</div>
        <canvas id="snakeboard" width="400" height="400"></canvas>
        <style>
            #snakeboard {
                position: absolute;
                top: 50%;
                left: 50%;
                transform: translate(-50%, -50%);
            }
            #score {
                text-align: center;
                font-size: 140px;
            }
        </style>
    </body>

    <script src="snake.js"></script>
</html>
```

Para criar o jogo, vamos usar um `<canvas>` em HTML, que serve para se desenhar gráficos em JavaScript. No código acima, nós criamos um *canvas* quadrado com 400x400 *pixels*, e damos a ele o `id` de `snakeboard`, que será usado no código JS depois. O *tag* `<style>` define algumas características de exibição dos elementos.

Ao carregar a página no navegador você só deve ver a pontuação do *tag* `<div>`. Isto ocorre porque o *canvas* ainda não foi inicializado. Vamos fazer isso em JS. Crie o arquivo `snake.js` para começar.

Criando o canvas

Para acessar o objeto do *canvas* no DOM, basta usar o `id` definido no HTML.

```
const board = document.getElementById("snakeboard");
const boardContext = board.getContext("2d");
```

A seguir, acessamos o contexto 2D do *canvas*, já que a tela do jogo será desenhada em um espaço 2D. Para começar, vamos desenhar as bordas do *canvas* com a função abaixo.

```
function clearBoard() {
    boardContext.fillStyle = "white";
    boardContext.strokestyle = "black";
    boardContext.fillRect(0, 0, board.width, board.height);
    boardContext.strokeRect(0, 0, board.width, board.height);
}
```

As duas primeiras linhas definem as cores de fundo e da borda, respectivamente. Os demais comandos desenham o *canvas*. Se você incluir uma chamada a esta função no seu arquivo `snake.js` e recarregar a página, deve aparecer um quadrado.

Criando a serpente e a comida

Vamos representar a serpente internamente como um *array* de coordenadas. O tamanho do *array* determina o comprimento da serpente. A declaração inicial fica como abaixo, aonde a primeira coordenada indica a cabeça, que começa no centro.

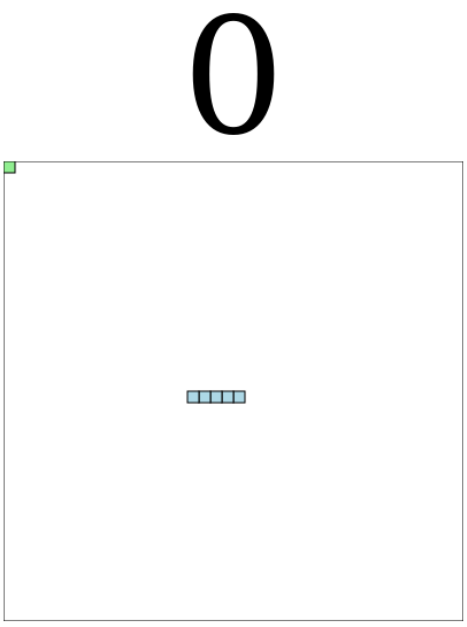
```
let snake = [
    {x: 200, y: 200},
    {x: 190, y: 200},
    {x: 180, y: 200},
    {x: 170, y: 200},
    {x: 160, y: 200},
];
```

Agora, para desenhar a serpente no *canvas*, podemos criar uma função que desenha um quadrado para cada par de coordenadas e depois construir a função `drawSnake()` com um *loop*.

```
function drawSnakePart(snakePart) {
    boardContext.fillStyle = "lightblue";
    boardContext.strokestyle = "darkblue";
    boardContext.fillRect(snakePart.x, snakePart.y, 10, 10);
    boardContext.strokeRect(snakePart.x, snakePart.y, 10, 10);
}

function drawSnake() {
    snake.forEach(drawSnakePart);
}
```

Uma função `drawFood()` para se desenhar a comida é praticamente idêntica a `drawSnakePart`. Tente criá-la, usando cores diferentes (por exemplo, verde). Use a variável `food` para guardar as coordenadas da comida. Depois, chame as funções `clearBoard()`, `drawFood()` e `drawSnake()`. Você deve ver algo como abaixo.



4.2 A lógica do jogo

A lógica de funcionamento do jogo começa pela função `run()`, que é bem simples.

```
// Start the game
function run() {
    // Register a listener for key presses.
    document.addEventListener("keydown", changeDirection);
    generateFood();
    mainLoop();
}
```

A primeira coisa a se fazer é registrar um *handler* para os eventos do teclado. Assim podemos controlar a serpente pelas setas. Isto é feito na função `changeDirection`, que será discutida adiante. A seguir, geramos uma comida em uma posição aleatória do *canvas* e iniciamos o *loop* principal do jogo. O código da função `generateFood()` é mostrado abaixo, juntamente com algumas funções auxiliares.

```
function sameCoords(one, two) {
    return (one.x === two.x) && (one.y === two.y);
}

function randomFoodCoord(min, max) {
    return Math.round((Math.random() * (max-min) + min) / 10) * 10;
}

function generateFood() {
    food.x = randomFoodCoord(0, board.width - 10);
    food.y = randomFoodCoord(0, board.height - 10);
    // Check if the new food location is where the snake currently is,
    // if so, generate a new food location.
    snake.forEach(
        part => {
            if (sameCoords(part, food)) generateFood();
    });
}
```

A função `mainLoop()` pode ser vista a seguir.

```
function mainLoop() {
    if (gameOver()) return;

    changingDirection = false;
    setTimeout(function onTick() {
                    clearBoard();
                    drawFood();
                    moveSnake();
                    drawSnake();
                    mainLoop(); // repeat
                }, 100);
}
```

O *loop* permanece rodando enquanto as condições do fim do jogo não forem sinalizadas pela função `gameOver()`. A variável `changingDirection` é utilizada pela função `moveSnake()`, que será discutida a seguir. O ponto chave do código acima é o uso de `setTimeout` para executar a função `onTick()` a cada 100 milissegundos. Note que ao final de cada chamada de `onTick()`, o *loop* volta a ser executado recursivamente.

Resta agora ver como a serpente se movimenta. O código de `moveSnake()` pode ser visto abaixo.

```
function moveSnake() {
    // Create the new snake head
    const head = {x: snake[0].x + velocity.dx,
                  y: snake[0].y + velocity.dy};
    // Add the new head to the beginning of snake body
    snake.unshift(head);
    if (sameCoords(head, food)) {
        // Increase score
        score += 10;
        // Display score on screen
        document.getElementById("score").innerHTML = score;
        // Generate new food location
        generateFood();
    } else {
        // Remove the last part of snake body
        snake.pop();
    }
}
```

Utilizamos a variável `velocity` para indicar a direção e velocidade do deslocamento que deve ser feito. Vamos trabalhar com uma velocidade constante de 10 *pixels*. Assim, se queremos que a serpente se movimente para a direita temos de definir `velocity.dx = 10`, e, de forma oposta, `velocity.dx = -10` para indicar uma movimentação para a esquerda. A mesma ideia se aplica no eixo y com `velocity.dy`.

Após criar uma nova cabeça da serpente na posição atualizada, ela é incluída no *array* `snake`. Se a cabeça não chegou na comida (bloco `else` no código acima), a última posição em `snake` é removida para evitar que a serpente cresça. Caso contrário (bloco `if`), a pontuação é atualizada e uma nova comida é gerada em outra posição.

O código visto até agora só garante que a serpente fica se movimentando automaticamente em uma única direção a cada 100 milissegundos. O controle do jogo é feito de forma assíncrona, pela função `changeDirection`, que é chamada toda vez que uma tecla é pressionada. O código desta função é dado abaixo, sendo razoavelmente simples: analisando o evento, podemos ver qual tecla foi pressionada e modificar a velocidade de acordo.

```
function changeDirection(event) {
    const LEFT_KEY = 37;
    const RIGHT_KEY = 39;
    const UP_KEY = 38;
    const DOWN_KEY = 40;

    // Prevent the snake from reversing
    if (changingDirection) return;
    changingDirection = true;

    const keyPressed = event.keyCode;
    const goingUp    = velocity.dy === -10;
    const goingDown  = velocity.dy === 10;
    const goingRight = velocity.dx === 10;
    const goingLeft  = velocity.dx === -10;

    if (keyPressed === LEFT_KEY && !goingRight) {
        velocity.dx = -10;
```

```
        velocity.dy = 0;
    } else if (keyPressed === UP_KEY && !goingDown) {
        velocity.dx = 0;
        velocity.dy = -10;
    } else if (keyPressed === RIGHT_KEY && !goingLeft) {
        velocity.dx = 10;
        velocity.dy = 0;
    } else if (keyPressed === DOWN_KEY && !goingUp) {
        velocity.dx = 0;
        velocity.dy = 10;
    }
}
```

Por fim, resta definir quando o jogo termina, o que ocorre quando a cabeça da serpente "bate" na borda do *canvas* ou em alguma outra parte do próprio corpo. Estas condições são codificadas na função `gameOver()`.

```
function gameOver() {
    let head = snake[0];
    // Check if head has hit its own body.
    for (let i = 4; i < snake.length; i++) {
        if (sameCoords(head, snake[i])) return true;
    }
    const hitLeftWall   = head.x < 0;
    const hitRightWall  = head.x > board.width - 10;
    const hitTopWall    = head.y < 0;
    const hitBottomWall = head.y > board.height - 10;
    return hitLeftWall || hitRightWall || hitTopWall || hitBottomWall;
}
```

Existem inúmeras melhorias que poderiam ser feitas a partir de agora, mas o código mostrado até aqui é suficiente para termos um jogo minimamente funcional. Copie e cole a lógica do jogo no seu arquivo `snake.js`, ou se preferir, veja o código completo disponibilizado junto com este documento. Certifique-se de que você entendeu todo o funcionamento e divirta-se. :)

Atividade

Implemente o jogo de adivinhar uma palavra, popularmente conhecido como *jogo da forca* (`https://en.wikipedia.org/wiki/Hangman_(game)`). Não é necessário fazer uma apresentação gráfica muito elaborada, um contador com o número de tentativas restantes basta. Você é livre para obter a palavra a ser adivinhada de qualquer lugar, por exemplo, de algum site na internet.