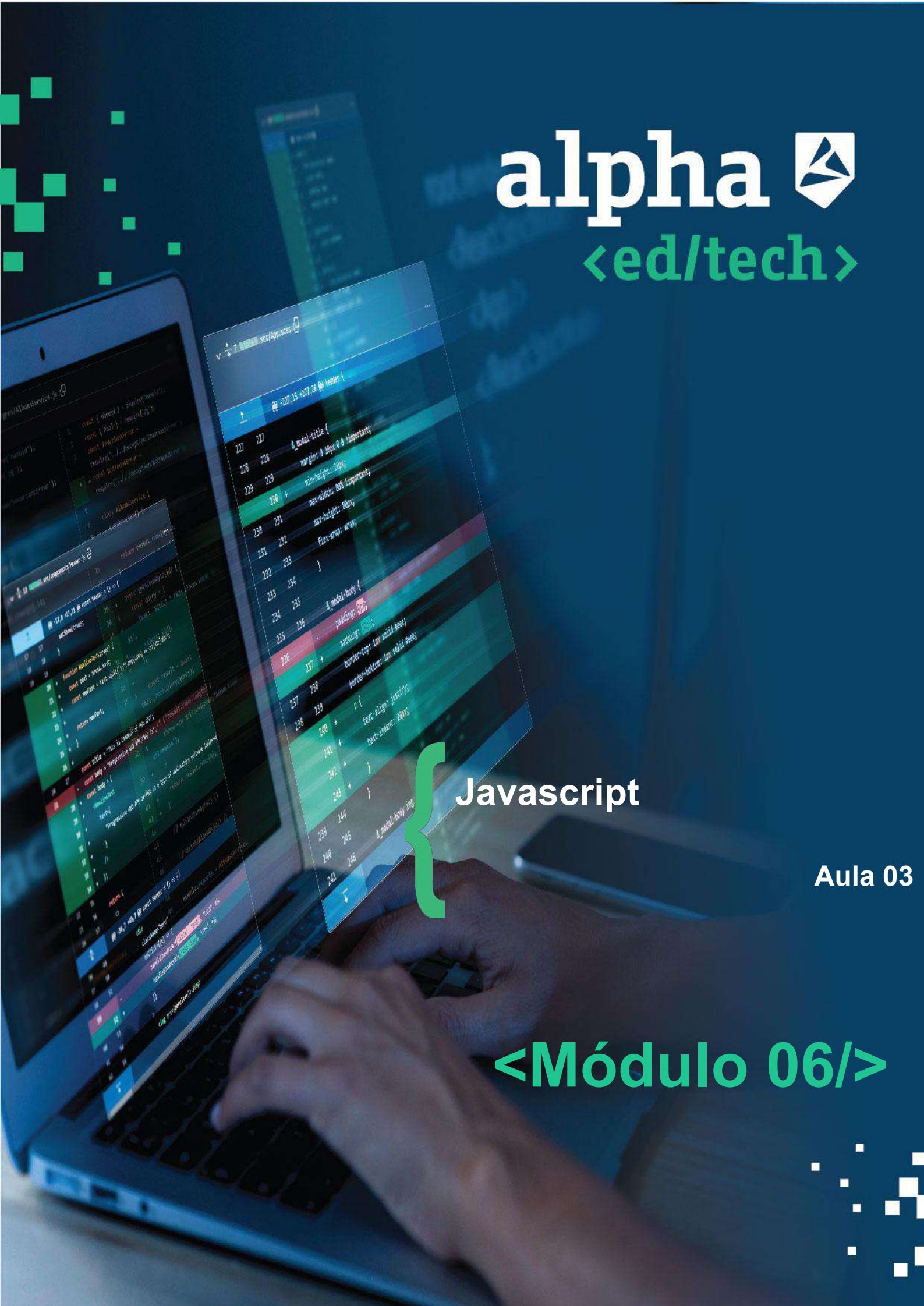

alpha
<ed/tech>
Javascript
Aula 03
<Módulo 06/>

Local Storage, JSON e try-catch

Sumário

Local Storage

Introdução: problema do formulário

As páginas web que conseguimos codificar até agora possuem uma limitação:
Ao fechar a página, tudo que estávamos fazendo é perdido.
Por exemplo, suponha que sua página tem um formulário muito longo:

```
<form>
  Nome: <input id="name-input" />
  <br />
  Idade: <input id="age-input" />
  <br />
  Endereço
  <br />
  Rua: <input id="street-input" />
  <br />
  Bairro:
  <input id="district-input" />
  <br />
  Cidade: <input id="city-input" />
  <br />
  Estado:
  <select id="state-input">
    <option value="SP">São Paulo</option>
    <option value="RJ">Rio de Janeiro</option>
    <option value="MG">Minas Gerais</option>
    <option value="PR">Paraná</option>
    <option value="other">Outro</option>
  </select>
</form>
```

Se o usuário preencher parte dos dados e fechar a página, perderá o que foi preenchido.
Ao reabrir a página, os inputs estarão em branco de novo.
Veremos agora como salvar os dados mesmo após a página fechar.

Local Storage

O que é Local Storage

Os navegadores web fornecem para cada site um espaço de armazenamento local.

"Local" significa que são dados guardados no disco rígido do computador do usuário, de maneira permanente.

O javascript do site tem capacidade de acessar esse armazenamento, e assim cada site pode salvar dados de maneira permanente no computador do usuário.

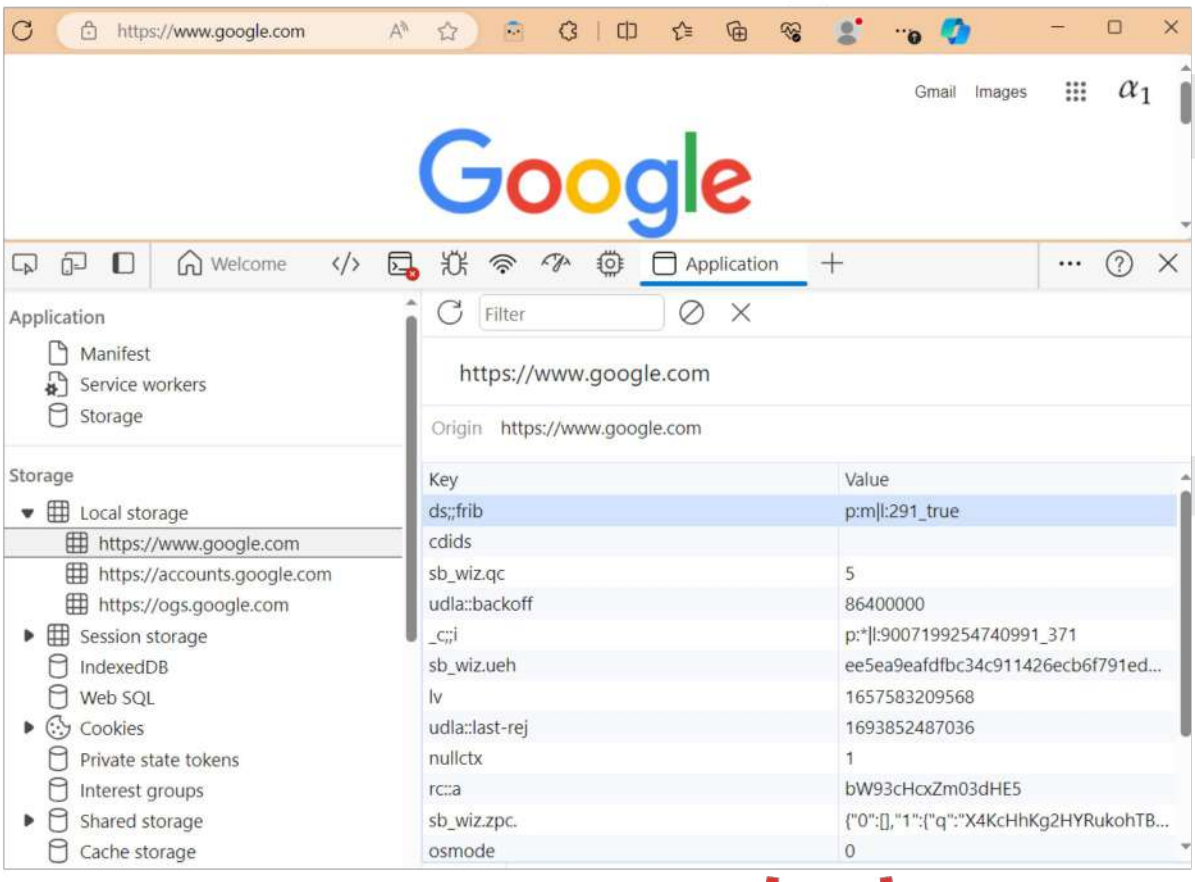
E cada site tem seu "espaço de armazenamento" próprio.

Significa que um site não pode acessar os dados de outro site.

Vejamos como visualizar o Local Storage:

- Abra um site, por exemplo o google.
- Abra as ferramentas de desenvolvedor e procure a guia Application.
- Clique em Local Storage

Você verá o seguinte:



Local Storage do domínio (site) www.google.com

Pares chave-valor armazenados pelo Google

Local Storage

À esquerda, aparecem três domínios (sites), o www.google.com está selecionado.
À direita aparece uma tabela, que é o Local Storage gerenciado por esse domínio.
A estrutura dessa tabela é uma lista de pares chave-valor.
Por exemplo:

- Na primeira linha a chave é a string `"ds;;frib"` e o valor é a string `"p:m|l:291_true"`
- Na segunda linha a chave é a string `"cdids"` e o valor é uma string vazia
- Na terceira linha a chave é a string `"sb_wiz.qc"` e o valor é a string `"5"`

Note que não há chaves repetidas (valor pode repetir) e que todas as chaves e valores são strings (mesmo números são guardados como strings).
Quem colocou essas chaves e valores na tabela foi o próprio javascript do site do google, então o significado desses dados depende desse javascript (provavelmente o código armazena os dados criptografados/embaralhados, por isso não faz sentido para nós).

Como você pode ver do lado esquerdo, existem vários espaços de armazenamento diferentes além do Local Storage.
Por exemplo Session Storage, IndexedDB, Cookies e outros.
Não vamos abordar essas outras formas de armazenamento aqui.

Os dados do Local Storage são permanentes, exceto se:

- O usuário manualmente deletar os dados (pelas configurações do navegador)
- O próprio javascript do site deletar seus dados (afinal o javascript pode manipular como quiser o Local Storage do domínio)

Local Storage

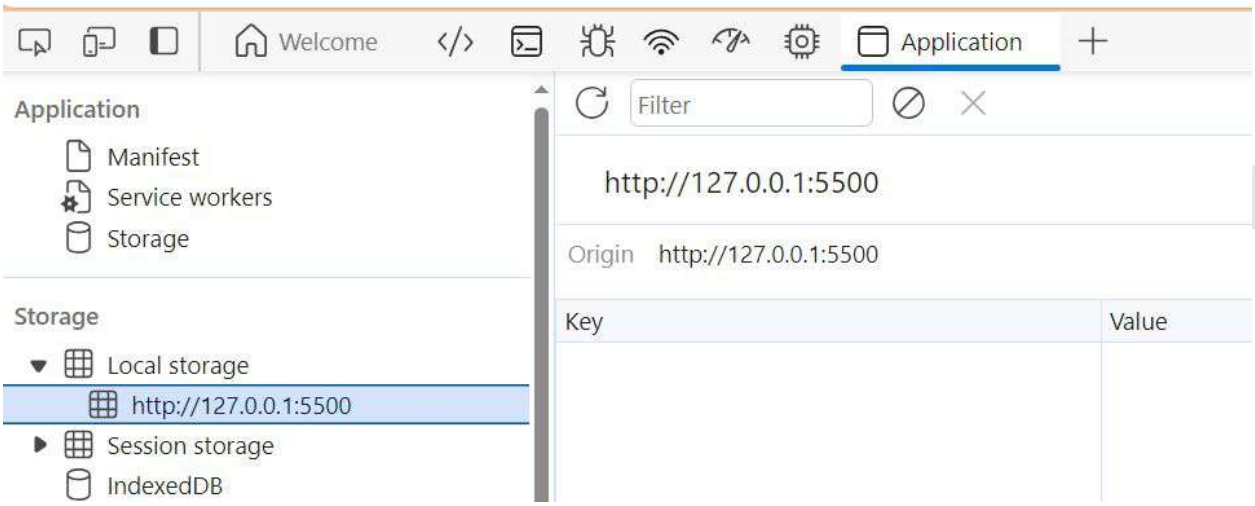
Como acessar o Local Storage no javascript

Para vermos como o javascript pode acessar o Local Storage, vamos criar uma página web e abrí-la no navegador, para não mexermos nos dados do Google.

Crie um arquivo html (pode ser vazio mesmo) e abra no navegador.

Acesse o Local Storage nas ferramentas de desenvolvedor.

Você verá o seguinte:



O domínio é 127.0.0.1:5500 (isso depende do servidor local que você usou para abrir a página).

E a tabela está vazia, por nunca usei o Local Storage desse domínio.

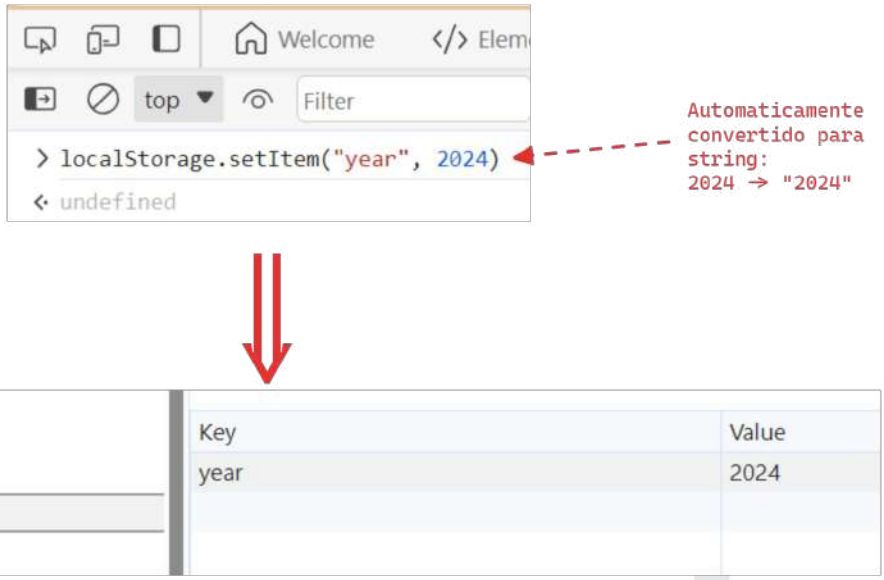
Ainda nessa página, pelo Console podemos acessar o Local Storage usando um objeto pré-definido chamado `localStorage`.

Entre seus métodos, dois deles nos serão úteis imediatamente:

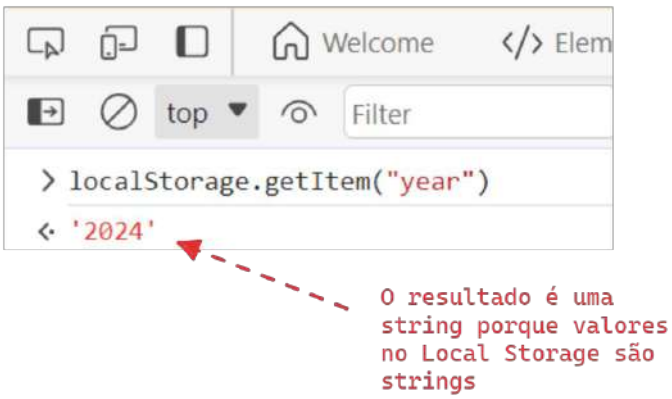
- `localStorage.setItem(«chave», «valor»)`
  Armazena um novo par chave-valor.
  A chave e o valor são convertidos para strings se já não forem strings.
  Se a chave já existia no Local Storage, seu antigo valor será deletado e substituído pelo novo valor.
- `localStorage.getItem(«chave»)`
  Retorna o valor (string) associado à chave solicitada.
  Se a chave não existe no Local Storage, retorna `null`

Local Storage

Por exemplo para guardar uma chave `"year"` com valor `"2024"`:



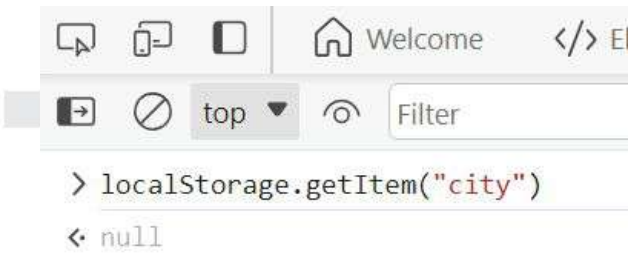
Agora o javascript pode consultar o valor dessa chave:



Local Storage

Essa consulta funciona mesmo se você fechar a página e abrir de novo (porque o Local Storage é permanente).

Por último, ao consultar uma chave que não existe, o `getItem` retorna **`null`**:



Esse espaço de armazenamento é exclusivo para o domínio 127.0.0.1:5500.

Se em outro momento você abrir o HTML com outro endereço, por exemplo 127.0.0.1:8000, a tabela do Local Storage estará vazia !

Não quer dizer que os dados foram perdidos.

Quer dizer que o endereço 127.0.0.1:8000 tem *outro espaço de armazenamento* isolado do espaço de armazenamento do endereço 127.0.0.1:5500.

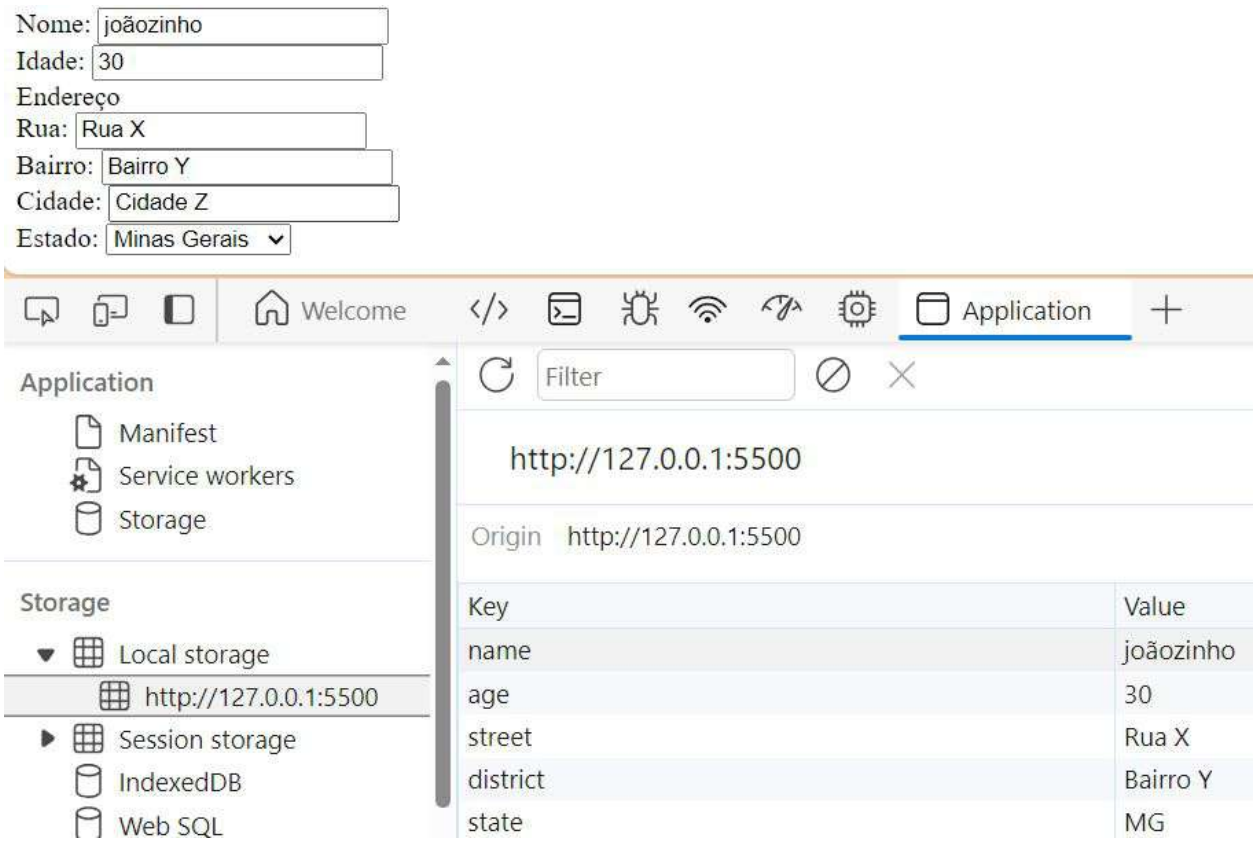
Local Storage

Solução do formulário (parte 1): salvar os dados continuamente

Voltando ao problema do formulário, para que os valores digitados pelo usuário não sejam perdidos caso feche a página, podemos adicionar ouvintes de evento "change" para atualizar o Local Storage:

```
document.getElementById("name-input").addEventListener("change", function () {
  localStorage.setItem("name", this.value);
});

document.getElementById("age-input").addEventListener("change", function () {
  localStorage.setItem("age", this.value);
});


... etc ...

document.getElementById("state-input").addEventListener("change", function () {
  localStorage.setItem("state", this.value);
});
```

Assim o Local Storage recebe o dado de um input sempre que o usuário o altera de valor:



Local Storage

O código ficou longo por termos que adicionar vários ouvintes de evento.

Por enquanto não é possível fazer melhor que isso, pois precisaríamos de um comando de repetição (ainda veremos).

Solução do formulário (parte 2): carregar os dados do Local Storage quando a página iniciar

Essa foi metade da solução.

Mas ao recarregar a página, os inputs estarão em branco.

Precisamos carregar do Local Storage o que estiver guardado nele e colocar nos inputs, no momento que a página abre:

```
// código global
// (para executar imediatamente quando a página abre)

const name = localStorage.getItem("name");
document.getElementById("name-input").value = name;

const age  = localStorage.getItem("age");
document.getElementById("age-input").value = age;

... etc ...

const state = localStorage.getItem("state");
document.getElementById("state-input").value = state;
```

Note que esse código é "duvidoso" se for a primeira vez que o usuário acessa a página.

O código está assumindo que há dados guardados no Local Storage, mas na primeira vez não tem.

Então no primeiro acesso à página, todos os `getItem` retornarão **`null`**.

Felizmente, isso não causa erro nenhum, porque o passo a passo da execução fica assim (exemplo para o nome):

- `document.getElementById("name-input").value = name;`
- ⇒ `document.getElementById("name-input").value = null;`

Então o input recebe o valor **`null`**, que (peculiaridade do DOM) tem o mesmo efeito que `""`, portanto o input permanece vazio, como queremos.

Local Storage

Casos de uso e limitações do Local Storage

Casos de Uso do `localStorage`:

- **Manter o Estado da Aplicação**

  Útil para salvar o estado de uma aplicação web, como preferências do usuário ou dados de uma sessão.

  Por exemplo, se um usuário ajusta o tema de um site para modo escuro, essa preferência pode ser salva no `localStorage` para manter a consistência em visitas futuras.

- **Caching de Dados Leves**

  Para armazenar informações que não precisam ser atualizadas com frequência, como um pequeno catálogo de produtos ou uma lista de postagens de blog.

- **Salvar Progresso em Jogos ou Formulários**

  Em aplicações como jogos ou formulários longos, o `localStorage` pode ser usado para salvar o progresso do usuário, permitindo que ele retome de onde parou mesmo após fechar o navegador

Limitações do `localStorage` :

- **Capacidade de Armazenamento Limitada**

  Geralmente limitado a cerca de 5MB por domínio, o que pode ser insuficiente para algumas aplicações mais robustas.

  Mesmo assim 5MB é muita coisa.

- **Segurança**

  Dados armazenados no `localStorage` são suscetíveis a ataques de Cross-Site Scripting (XSS), pois qualquer script injetado na página pode acessá-los.

- **Armazenamento Apenas no Lado do Cliente**

  Não é uma solução para compartilhar dados entre diferentes usuários ou dispositivos, já que os dados são armazenados localmente no navegador do usuário.

- **Sem Suporte para Dados Complexos**

  O `localStorage` armazena dados somente como strings, o que significa que objetos precisam ser convertidos (usualmente via JSON, veremos em seguida) para serem armazenados e lidos.

JSON

Introdução

JSON significa "JavaScript Object Notation".

É uma forma de armazenamento de objetos javascript no formato de string.

Um uso extremamente comum disso é para transmitir dados pela internet: o remetente converte o objeto em string, transmite a string, e o destinatário reconverte a string de volta em objeto.

No nosso caso, vamos usar JSON para armazenar os dados do formulário no Local Storage.

A ideia é ter uma única variável para todos os inputs, em vez de uma variável para cada input.

Seria assim:

```
const data = {
  name: "",
  age: "",
  address: {
    street: "",
    district: "",
    city: "",
    state: "",
  },
};
```

E no Local Storage não teremos mais várias chaves, teremos uma única chave `"data"` armazenando o objeto inteiro.

Mas como o Local Storage só pode armazenar strings, precisamos "converter" o objeto numa string.

O JSON é justamente isso, um formato textual que pode ser convertido de volta em objeto (e vice-versa).

Método `JSON.stringify`

Em cada ouvinte de evento, devemos atualizar uma das propriedades do objeto `data`, e então guardar o objeto no Local Storage.

Uma tentativa seria assim:

```
// exemplo para o nome:
document.getElementById("name-input").addEventListener("change", function () {
  // atualiza a propriedade
  data.name = this.value;

  // salva no Local Storage
  localStorage.setItem("data", data);
});
```

Mas essa abordagem *não funciona*.

Se você olhar como ficou salvo no Local Storage, verá o seguinte:

JSON

Key	Value
data	[object Object]

Ficou salvo como `"[object Object]"`.

Isso aconteceu porque a conversão implícita de objeto (`data`) para string transforma qualquer objeto nessa string acima.

Não é útil, perdeu totalmente as informações do objeto.

Por isso precisamos do método `JSON.stringify`.

Ele transforma um objeto numa string com formato JSON, que é um formato sem perda de dados.

Vamos fazer um teste no Console:

```
> JSON.stringify({
      name: "João",
      age: 30,
      address: {
          street: "Rua X",
          district: "Bairro Y",
          city: "Cidade Z",
          state: "SP"
      }
  })
< '{"name":"João","age":30,"address":{"street":"Rua X","district":"Bairro Y","city":"Cidade Z","state":"SP"}}'
```

Veja que a string retornada pelo `JSON.stringify` é uma representação fiel ao objeto, que preserva todas as informações dele.

Então o código corrigido do ouvinte de evento é:

```
// exemplo para o nome:
document.getElementById("name-input").addEventListener("change", function () {
  // atualiza a propriedade
  data.name = this.value;

  // salva no Local Storage
  localStorage.setItem("data", JSON.stringify(data));
});
```

JSON

Método `JSON.parse`

O `JSON.stringify` foi metade da solução: ele permite guardar o objeto como string sem perda da informação.

Agora precisamos pensar na outra metade: ao recarregar a página, o objeto deve ser recuperado do Local Storage e usado para preencher os inputs.

Uma tentativa seria a seguinte:

```
// código global

// 1. ler o objeto do Local Storage
const data = localStorage.getItem("data");

// 2. preencher todos os inputs
document.getElementById("name-input").value = data.name;
document.getElementById("age-input").value = data.age;
... etc ...
```

Mas isso vai falhar porque o `getItem` retorna sempre uma string, não faz a conversão da string em objeto.

Em outras palavras, `data` receberá a string `'{"name":"João","age":30,"address":{"street":"Rua X","district":"Bairro Y","city":"Cidade Z","state":"SP"}}'` que estava guardada no Local Storage.

Nossa intenção é reconverter essa string em objeto.

Para isso precisamos do método `JSON.parse`, que faz o oposto do `JSON.stringify`:

- `JSON.stringify`: objeto ⇒ string
- `JSON.parse`: string ⇒ objeto

Vamos fazer um teste no Console:

```
> JSON.parse('{"name":"João","age":30,"address":{"street":"Rua X","district":"Bairro Y","city":"Cidade Z","state":"SP"}}')
< ▼ {name: 'João', age: 30, address: {…}} i
    ▶ address: {street: 'Rua X', district: 'Bairro Y', city: 'Cidade Z', state: 'SP'}
      age: 30
      name: "João"
```

Assim, o código correto seria:

```
// código global

// 1. ler o objeto do Local Storage
const dataString = localStorage.getItem("data");
const data = JSON.parse(dataString);
// ou fazer tudo de uma vez: const data = JSON.parse(localStorage.getItem("data"));

// 2. preencher todos os inputs
document.getElementById("name-input").value = data.name;
document.getElementById("age-input").value = data.age;
... etc ...
```

JSON

O formato JSON é um formato textual para representar objetos, não tem relação direta com o Local Storage.

Estamos usando as duas coisas juntas porque o Local Storage só consegue armazenar strings, logo usar strings JSON é uma solução possível para armazenar objetos.

Cuidados com o primeiro acesso à página

O código global que carrega o objeto do Local Storage é:

```
const dataString = localStorage.getItem("data");
const data = JSON.parse(dataString);
```

Esse código ainda tem um bug, que acontece na primeira vez que o usuário abre a página (quando o Local Storage não tem nenhum dado).

Nessa situação, `localStorage.getItem("data")` retorna `null`, então `dataString` fica com valor `null`.

Em seguida, `JSON.parse(null)` também retorna `null` (faça o teste aí), logo `data` obtém o valor `null`.

Aí o erro acontece ao tentar acessar `data.name`:

Como `data` vale `null`, o comando se torna `null.name`, e isso gera um erro de execução (como você já sabe).

A solução é:

- Se `dataString` vale `null`, significa que o Local Storage não tinha informação.
- Em outras palavras, o usuário não tinha preenchido nada nos inputs ainda.
- Nesse caso, a variável `data` deve ser inicializada com um objeto que reflete essa situação (valores vazios nos inputs).

JSON

Podemos usar um `if/else` ou um operador ternário para isso. Uma solução possível:

```
// código global

// 1. ler o objeto do Local Storage
const dataString = localStorage.getItem("data");
const data =
  dataString !== null
    ? JSON.parse(dataString)
    : {
        name: "",
        age: "",
        address: {
          street: "",
          district: "",
          city: "",
          state: "",
        },
      };

// 2. preencher todos os inputs
document.getElementById("name-input").value = data.name;
document.getElementById("age-input").value = data.age;
... etc ...
```

JSON

Limitações do JSON

O JSON tem suporte para os seguintes tipos de dados:

- objetos "avulsos" (`{name: ..., age: ...}`)
- números
- strings
- booleanos
- null
- arrays (ainda estudaremos)

Objetos cujas propriedades contenham esses tipos de dados podem ser convertidos numa string JSON sem perda de informação.

Por outro lado, propriedades que contenham outros tipos de dados, como funções, `Date` e **`undefined`**, podem ser perdidas.

Por exemplo:

```
> JSON.stringify({
      id: "a9323v",
      score: 120,
      isAdmin: true,
      database: null,
      date: new Date(),
      nickname: undefined
  })
<· '{"id":"a9323v","score":120,"isAdmin":true,"database":null,"date":"2024-01-16T13:37:18.416Z"}'
```

As 4 primeiras propriedades são convertidas normalmente (porque os tipos string, número, booleano e null são suportados).

A 5ª chave tem um objeto `Date`, ele é convertido em string no formato ISO.

E a 6ª chave tem valor **`undefined`**, ela é perdida totalmente.

Note que não ocorre nenhum erro ao tentar converter o objeto, somente algumas propriedades (aquelas cujo tipo de dados não é suportado) podem ser alteradas ou perdidas (dependendo do tipo de dados).

Na prática, essa limitação não é relevante, porque os tipos de dados suportados pelo JSON são mais que suficientes para a maioria dos propósitos.

Captura de erros

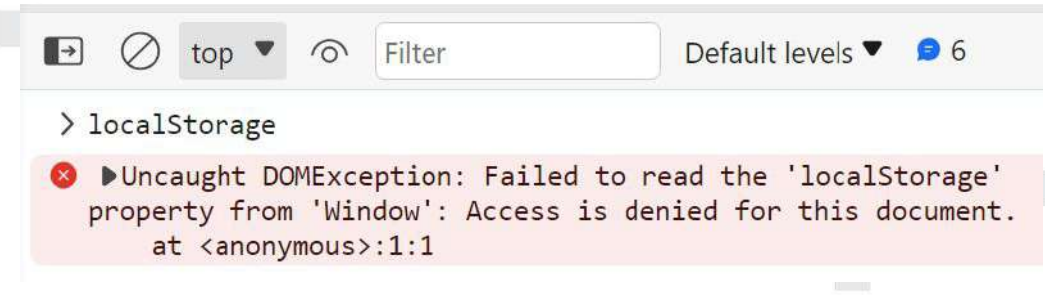
Introdução: Local Storage pode estar indisponível

Os navegadores têm configurações para desativar o armazenamento de um site (normalmente desativar os "cookies" desativa também o Local Storage, apesar de serem coisas diferentes).

Então ao escrever um javascript que faz uso do `localStorage`, é necessário que o código, antes de usar o `localStorage`, teste se ele está realmente disponível.

Então se o usuário o desativou, o javascript do seu site não poderá fazer uso do `localStorage`.

Por exemplo desativei o Local Storage no meu navegador e tentei acessá-lo pelo Console:



A simples referência ao objeto `localStorage` produziu um erro de execução.

Para piorar, mesmo que o usuário não tenha desativado esse recurso, especificamente no navegador Safari (Apple) em modo de "Navegação Anônima", o `localStorage` está disponível mas o comando `localStoragem.setItem` gera um erro (em outros navegadores funcionaria sem problemas).

Nosso site não pode simplesmente "quebrar" se o `localStorage` estiver bloqueado ou não-funcional.

O javascript precisa detectar isso e contornar o problema sem abortar com um erro de execução.

No nosso exemplo do formulário, se o `localStorage` não estiver funcional, podemos colocar uma mensagem no HTML dizendo "suas alterações não serão salvas" mas manter o formulário como está (porém sem os ouvintes de evento, já que não é possível armazenar os dados preenchidos).

Captura de erros

Comando try-catch

Como o javascript pode checar se o `localStorage` está disponível de maneira "segura" ? (sem abortar por causa de um erro de execução ?)

A única maneira de testar o `localStorage` é tentar usá-lo:

```
// colocar e depois retirar um par chave-valor qualquer,
// para ver se vai dar certo ou abortar com erro de execução
localStorage.setItem("test key", "test value");
localStorage.removeItem("test key");

// se o código não abortar nas linhas acima, quer dizer
// que o localStorage está funcional.
// Então o código daqui para baixo não precisa se
// preocupar com isso mais, porque se chegou até aqui
// está tudo certo.
// restante do código: carregar do localStorage,
// preencher inputs, adicionar ouvintes de evento
...
```

Mas não pode ser só isso, senão quando o `localStorage` não estiver funcional, o código vai abortar na primeira linha e não teremos chance de fazer mais nada.

Nesse cenário, existe o comando **`try-catch`**.

Vamos exemplificar com um código simples (ainda não é o código para a página de formulário, mas depois adaptamos):

```
console.log("início do programa");

try {
  console.log("testando localStorage");
  localStorage.setItem("test key", "test value");
  localStorage.removeItem("test key");
  console.log("teste bem sucedido");
} catch (error) {
  console.log("local storage não está disponível");
}

console.log("fim do programa");
```

Captura de erros

O funcionamento é o seguinte:

- Primeiro imprime `"início do programa"`
- Agora o navegador web encontra um comando try-catch.

  O **`try`** e o **`catch`** precisam estar ambos presentes, não é possível usar somente o **`try`** ou somente o **`catch`**.
- O **`try`** significa "tentar". O navegador vai tentar executar os comandos dentro do bloco do **`try`**.

  "Tentar executar" significa executar observando se o comando vai gerar um erro de execução ou não.
- Agora há duas opções:
  - Pode ser que os comandos do **`try`** não tenham produzido erro de execução.

    Nesse caso, o bloco **`catch`** será *pulado* como se não existisse.

    E então vai chegar à última linha do código imprimindo `"fim do programa"`
  - Mas pode ser que algum comando do **`try`** tenha produzido um erro de execução.

    Nesse caso, as linhas seguintes dentro do **`try`** são abortadas. Por exemplo, se deu erro no segundo comando do **`try`**, o 3º e o 4º comandos não serão executados.

    Em seguida o navegador vai executar o bloco **`catch`** ("capturar").

    E finalmente a última linha de código.

Ou seja, em forma de diagrama:
Se não houver erro de execução em nenhum comando dentro do **`try`**:

```
console.log("início do programa");                    ✔ OK

try {
  console.log("testando localStorage");               ✔ OK
  localStorage.setItem("test key", "test value");     ✔ OK
  localStorage.removeItem("test key");                ✔ OK
  console.log("teste bem sucedido");                  ✔ OK
} catch (error) {
  console.log("local storage não está disponível");   ⏩ pulado
}

console.log("fim do programa");                       ✔ OK
```

Captura de erros

Se houver erro de execução em algum comando dentro do `try`:

```
console.log("início do programa");                    OK

try {
  console.log("testando localStorage");               OK
  localStorage.setItem("test key", "test value");     Erro de Execução
  localStorage.removeItem("test key");                pulado
  console.log("teste bem sucedido");                  pulado
} catch (error) {
  console.log("local storage não está disponível");   OK
}

console.log("fim do programa");                       OK
```

Nós usamos o try-catch aqui porque ele consegue *sobreviver* a um erro de execução.

Sem ele, a ocorrência de um erro de execução aborta o código onde está (fica uma execução incompleta).

Com ele, a ocorrência de um erro de execução não aborta o código (na verdade aborta somente o código dentro do `try`, como visto).

Em vez de abortar totalmente, o código continua no `catch` e no restante do código após ele.

Portanto podemos colocar o código "perigoso" (que pode dar erro) dentro do `try`, e assim garantir que o restante do programa não vai ser abortado mesmo que aconteça erro dentro do `try`.

Dentro do `try {...}` e dentro do `catch(error) {...}` você pode colocar quaisquer comandos.

Por exemplo pode ter um `if-else` dentro.

Captura de erros

Variável `error`

Faltou falar sobre aquela variável `error` no meio dos parênteses do **`catch`**.

Se o **`catch`** for executado, é porque houve erro dentro do **`try`**.

E no javascript, um erro de execução é um objeto do tipo `Error`.

A variável `error` faz a "captura" do erro, significa que o valor dela é o objeto `Error` que ocorreu dentro do **`try`**.

Assim, dentro do **`catch`** você tem acesso ao objeto de erro, por isso pode fazer alguma coisa com ele.

Por exemplo mostrar ao usuário a mensagem de erro: propriedade `error.message`, que é uma string.

Ou então imprimir o erro com `console.log(error)`.

O comando `catch` não é uma função

O parênteses em **`catch`** `(error)` não são os parênteses de uma função, são somente delimitadores que fazem parte do comando **`catch`**.

O **`catch`** não é uma função.

Não importa o nome da variável de erro

Pode ser `error` como mostramos, mas também é comum usar `err` para encurtar.

E o nome não tem significado especial, pode ser qualquer nome de variável.

Captura de erros

Escopo do try-catch

A região dentro do `try { ... }` constitui um escopo próprio.
Significa que variáveis declaradas dentro dele só existem lá dentro.
E a região dentro do `catch (err) { ... }` constitui outro escopo próprio.
Além disso, a variável `err` pertence ao escopo do `catch` (também só pode ser usada dentro dele).
Exemplo sem sentido prático, só para demonstrar os escopos:

```
let x = 5; // x existe em qualquer lugar

try {
  let y = 10;
  console.log(x); // funciona
  console.log(y); // funciona
  // console.log(z); erro porque z existe dentro do catch
} catch (err) {
  let z = 20;
  console.log(err); // funciona
  console.log(x); // funciona
  // console.log(y); erro porque y existe dentro do try
  console.log(z); // funciona
}

console.log(x); // funciona
// console.log(y); erro porque y existe dentro do try
// console.log(z); erro porque z existe dentro do catch
// console.log(err); erro porque err existe dentro do catch
```

Captura de erros

Aplicação do try-catch ao problema do formulário

Usando try-catch, podemos consertar a página do formulário para que ela:

- Exiba uma mensagem que "as respostas não serão salvas" caso o `localStorage` não esteja disponível.
- Opere normalmente caso o `localStorage` esteja disponível.

  Ou seja, carrega do armazenamento as respostas prévias, e adiciona ouvintes de evento para guardar as novas respostas.

Segue uma solução possível:

```
// local storage disponível ? true ou false
let localStorageAvailable;

try {
  localStorage.setItem("test key", "test value");
  localStorage.removeItem("test key");
  localStorageAvailable = true;
} catch (err) {
  localStorageAvailable = false;
}

if (!localStorageAvailable) {
  alert("Suas respostas não serão salvas");
} else {
  // Mesmo código antigo:

  // 1. ler o objeto do Local Storage
  const dataString = localStorage.getItem("data");
  const data =
    dataString !== null
      ? JSON.parse(dataString)
      : {
          name: "",
          age: "",
          address: {
            street: "",
            district: "",
            city: "",
            state: "",
          },
        };

  // 2. preencher todos os inputs
  document.getElementById("name-input").value = data.name;
  document.getElementById("age-input").value = data.age;
  // ... etc outros inputs ...

  // 3. adicionar ouvintes de evento
  document.getElementById("name-input").addEventListener("change", function () {
    data.name = this.value;
    localStorage.setItem("data", JSON.stringify(data));
  });
  // ... etc outros inputs ...
}
```

Captura de erros

Quando usar o try-catch ?

Uma boa regra de bolso é a seguinte:

- Se o cenário de erro pode ser previsto pelo código, prefira `if-else` para evitar o erro (não deixar que aconteça).

  Por exemplo para validar input de usuário: o código pode fazer os testes se o input está vazio, se é numérico, se tem o formato correto, etc.

- Se o cenário de erro não pode ser previsto pelo código, o erro não pode ser evitado.

  A única saída possível é o try-catch, se você não quiser que um erro de execução aborte o código no meio (incompleto).

  Esse foi o caso do `localStorage`.

  Não é possível fazer um teste para evitar o erro, ou seja, algo como `if (《localStorage está disponível》)`.

  A única maneira de testar a disponibilidade é executar código que pode dar erro: `localStorage.setItem(...)`

  Aí se deu erro, concluímos que o `localStorage` não está disponível. Senão, concluímos que está disponível.

Introdução a arrays, comando for-of e querySelectorAll

Sumário

Operações básicas em arrays

Introdução: galeria de imagens

Vamos fazer uma galeria de imagens !

Funciona assim:

- Um `<input>` para o usuário preencher o link da imagem, e um `<button>` escrito "Adicionar".
- Embaixo disso, uma `<img>` e dois `<button>` "⬅️" e "➡️".

  Ao adicionar a primeira imagem, ela é exibida na `<img>` .

  Ao adicionar uma segunda imagem, ela é exibida na `<img>` no lugar da primeira.

  E assim por diante.

  Mas as imagens anteriores não são perdidas:

  Digamos que já foram adicionadas 3 imagens e a 3ª está sendo exibida. Ao clicar no botão "⬅️", a 2ª imagem volta a ser exibida no lugar da 3ª. Clicando de novo, a 1ª imagem é exibida no lugar da 2ª. Clicando de novo, volta a 3ª (ou seja, "dá a volta" quando chega na 1ª.)

  E o contrário para o botão "➡️": se está sendo exibida a 1ª da três imagens, ele troca para a 2ª. Depois para a 3ª. Depois para a 1ª.

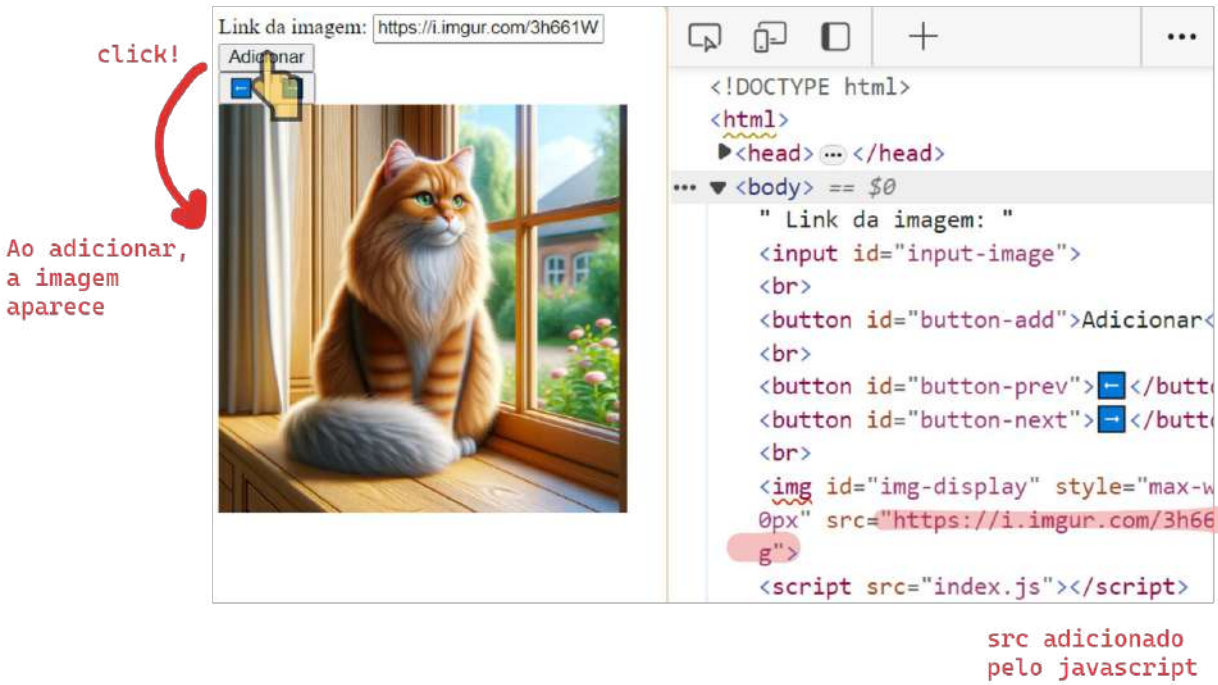
Para começar, ainda sem pensar nos botões "⬅️" e "➡️", vamos fazer uma primeira versão onde, ao adicionar uma nova imagem, a imagem prévia é perdida:

```
<!-- index.html -->
<html>
  <head></head>
  <body>
    Link da imagem: <input id="input-image" />
    <br />
    <button id="button-add">Adicionar</button>

    <br />
    <button id="button-prev">⬅️</button>
    <button id="button-next">➡️</button>
    <br />
    <img id="img-display" style="max-width: 300px" />

    <script src="index.js"></script>
  </body>
</html>
```

Operações básicas em arrays

```
// index.js
const input = document.getElementById("input-image");
const buttonAdd = document.getElementById("button-add");
const buttonPrev = document.getElementById("button-prev");
const buttonNext = document.getElementById("button-next");
const img = document.getElementById("img-display");

buttonAdd.addEventListener("click", function () {
  // obter o link digitado no input
  const imageLink = input.value;
  // colocar esse link como `src` da imagem
  img.src = imageLink;
});
```

Ao digitar um link e clicar em "Adicionar", o `src` da `<img>` é trocado e a imagem aparece:



Os botões "⬅" e "➡" não estão fazendo nada por enquanto (note que estão sem ouvinte de evento).

Nessa versão inicial, se outro link for digitado no input, ao clicar em "Adicionar" vai substituir o `src`, portanto o link antigo é perdido e substituído pelo novo link (nova imagem).

Não é isso que queremos ! Precisamos guardar todos os links para poder implementar os botões "⬅" e "➡", não podemos perder o link atual ao adicionar o novo link.

Isso pode ser feito usando **Arrays** no javascript, que são um tipo especial de objeto.

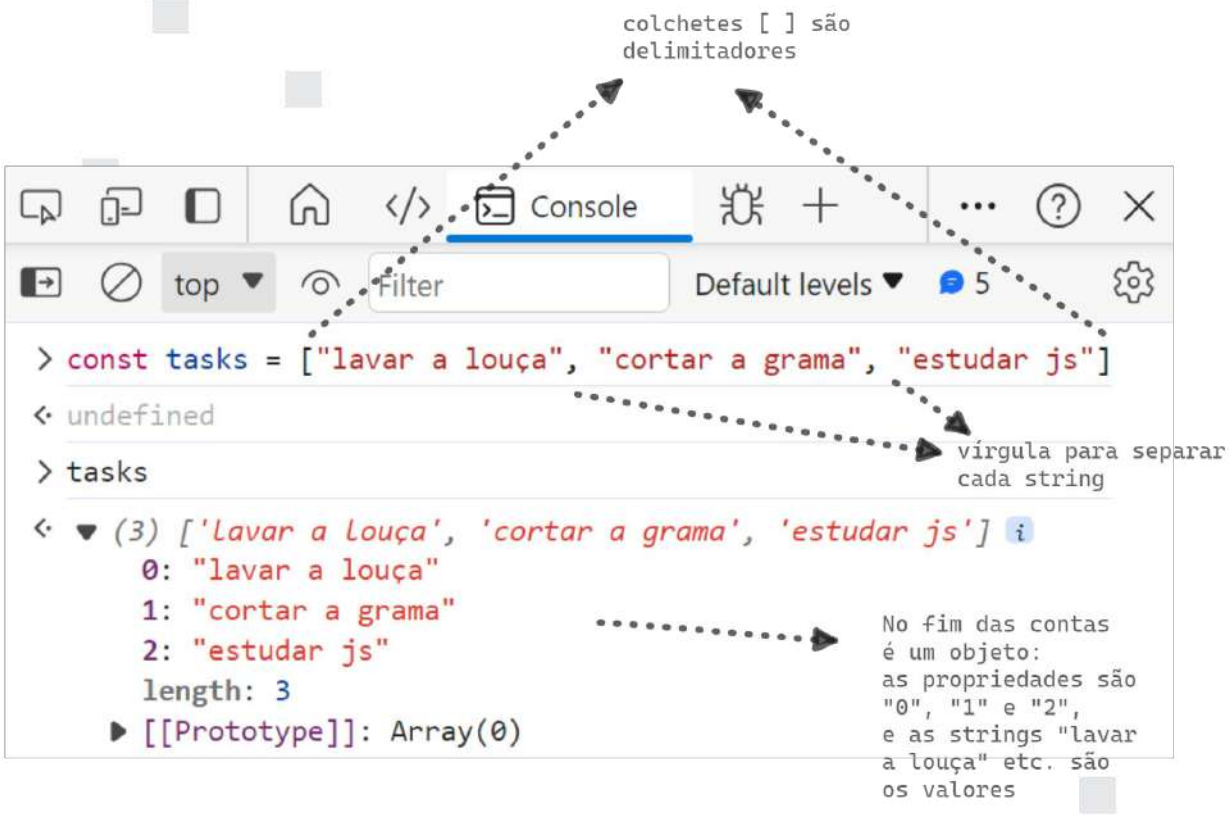
Um array é uma lista ordenada, ou seja, um espaço onde é possível armazenar vários valores (várias strings, números, etc.) e cada valor tem uma posição dentro da lista (primeiro, segundo, terceiro, ...)

Vejamos primeiro como funciona um array, depois o aplicaremos para resolver o problema da galeria de imagens.

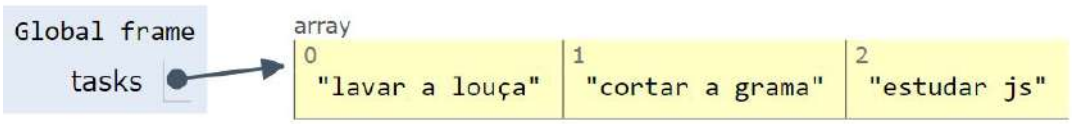
Operações básicas em arrays

Sintaxe de criação de array

Deixando de lado a galeria de imagens por enquanto, vamos exemplificar como criar um array (lista) contendo três strings, e guardar esse array dentro de uma variável:



Você pode pensar num array da seguinte forma:



Ou seja, o array é uma lista contendo três strings.

Não necessariamente precisa ser strings, você pode criar uma lista (array) de números, por exemplo um sorteio da loteria:

- `const lottery = [30, 35, 12, 54, 59, 60]`

E não necessariamente o array precisa ter somente números ou strings, pode misturar números, strings e qualquer outro tipo de dados:

- `const list = ["joão", 30, true, undefined, null, "maria"]`

Mas isso é incomum.

Operações básicas em arrays

Elemento

Cada valor guardado no array é chamado de **Elemento**.

Então no array `tasks`, o primeiro elemento é a string `"lavar a louça"`, o segundo é `"cortar a grama"` e o terceiro é `"estudar js"`.

Já no array `lottery`, o primeiro elemento é o número `30`, o segundo elemento é o número `35`, etc.

E no array `list`, o primeiro elemento é a string `"joão"`, o segundo é o número `30`, o terceiro é o booleano `true`, etc.

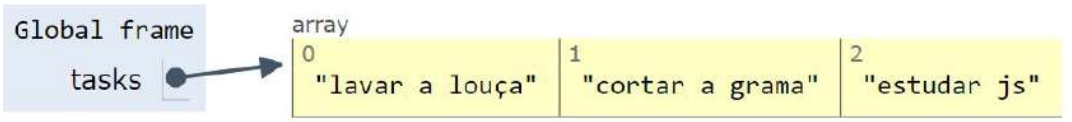
Índice

A posição onde um elemento se encontra dentro do array é chamada de **Índice**, começando pelo zero.

Então no array `tasks`, a string `"lavar a louça"` está no índice `0`, a string `"cortar a grama"` está no índice `1` e a string `"estudar js"` está no índice `2`.

Acesso e modificação

Como dito, você pode pensar no array simplificadamente como no diagrama abaixo:

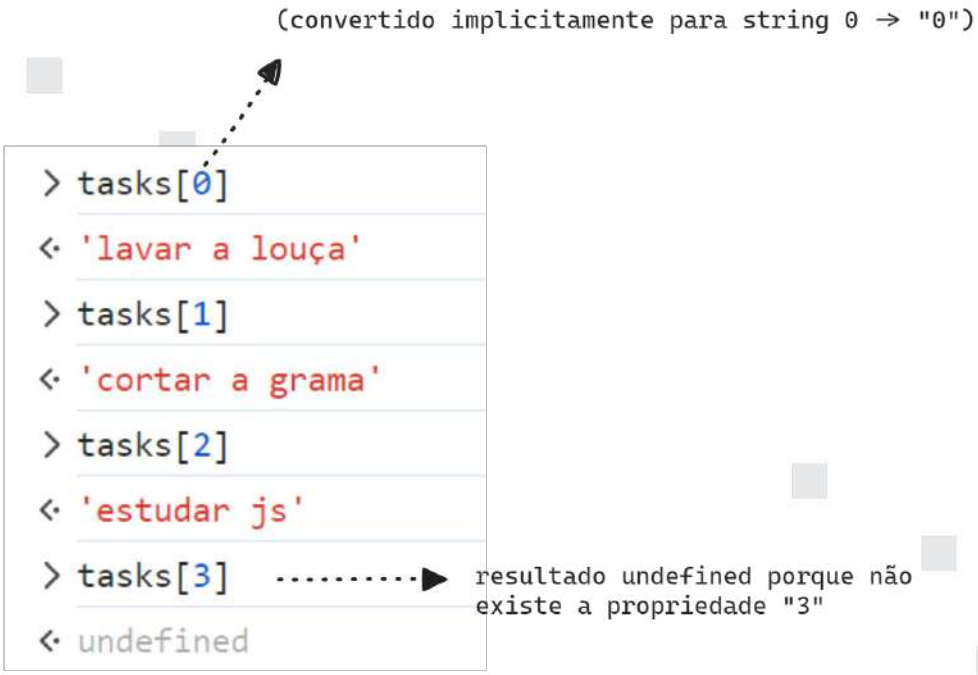


Mas na realidade o array é tecnicamente um objeto:

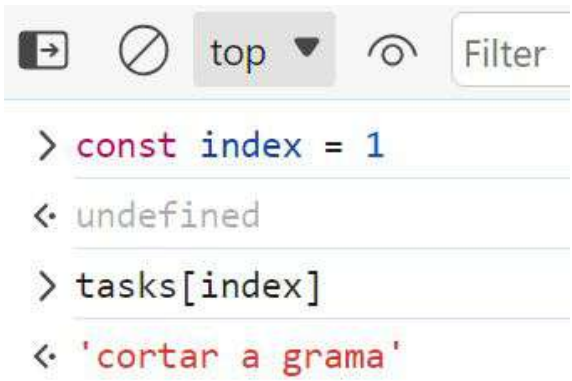
```
> tasks
<· ▼ (3) ['lavar a louça', 'cortar a grama', 'estudar js']
       0: "lavar a louça"
       1: "cortar a grama"
       2: "estudar js"
       length: 3
```

Operações básicas em arrays

Sendo um objeto, você já sabe como acessar cada um dos seus valores, usando a *notação de colchete* já vista:



Lembrando também que qualquer expressão pode ser usada dentro dos colchetes:



O passo a passo é:

- `tasks[index]` ⇒ `tasks[1]` ⇒ `"cortar a grama"`

Operações básicas em arrays

E você também já sabe como substituir um valor, usando a notação de colchete:

```
> tasks
< ▼ (3) ['lavar a louça', 'cortar a grama', 'estudar js']
    0: "lavar a louça"
    1: "cortar a grama"
    2: "estudar js"
    length: 3
  ▶ [[Prototype]]: Array(0)
> tasks[1] = "fazer nada"
< 'fazer nada'
> tasks
< ▼ (3) ['lavar a louça', 'fazer nada', 'estudar js']
    0: "lavar a louça"
    1: "fazer nada"
    2: "estudar js"
    length: 3
  ▶ [[Prototype]]: Array(0)
```

Operações básicas em arrays

Propriedade length

Nos prints do Console, você deve ter percebido que existia uma outra propriedade além dos índices (0, 1, 2), que é a propriedade `length`.

Essa é uma propriedade mágica que existe em arrays, o valor dela é sempre a quantidade de elementos que existe no array.

Como `tasks` tem 3 elementos, `tasks.length` vale 3.

No exemplo da loteria, `lottery.length` vale 6:

```
> const lottery = [30, 35, 12, 54, 59, 60]
<· undefined
> lottery
<· ▼ (6) [30, 35, 12, 54, 59, 60] i
      0: 30
      1: 35
      2: 12
      3: 54
      4: 59
      5: 60
      length: 6
```

Estratégia: índice do último elemento

Suponha que você tem um array, esse array pode ter qualquer quantidade positiva de elementos.

Como fazer para acessar o último elemento do array ?

Pense na seguinte lógica:

- Se o array tem 1 elemento, o último tem índice 0
- Se o array tem 2 elementos, o último tem índice 1
- Se o array tem 3 elementos, o último tem índice 2
- etc.

Percebe qual o padrão ? O último elemento tem índice `tasks.length - 1`.

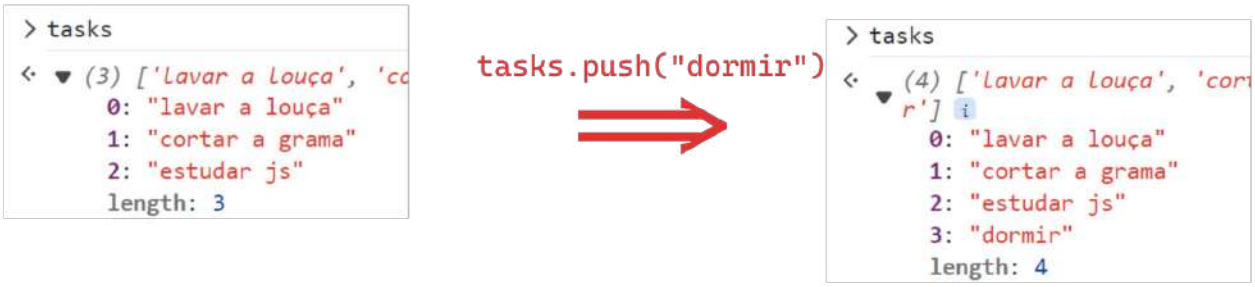
Então, não importando a quantidade de elementos do array, um código que sempre funciona para acessar o último elemento é: `tasks[tasks.length - 1]`.

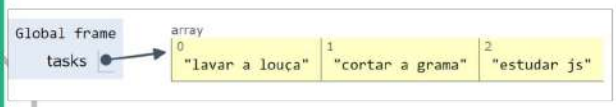
Obs: se o array tem tamanho 0 (ou seja, está vazio), note que esse código tentará acessar `tasks[-1]`, que vai resultar **`undefined`**. Se no contexto do seu código o array pode estar vazio, pode ser uma boa ideia ter um `if (tasks.length !== 0)`.

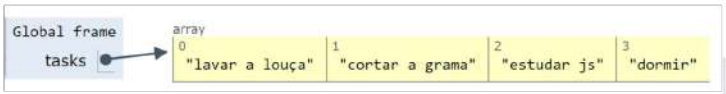
Operações básicas em arrays

Adicionar um elemento no final: método push()

Se quisermos adicionar um novo elemento no final da lista, podemos usar `tasks.push("dormir")` :







Note que a propriedade `length` se atualiza automaticamente, refletindo a nova quantidade de elementos.

Remover um elemento do final: método pop()

Faz o contrário do `push`, é usado assim: `tasks.pop()`

A propriedade `length` se atualiza automaticamente, refletindo a nova quantidade de elementos (um a menos).

Operações básicas em arrays

Aplicação de array: solução da galeria de imagens

Na galeria de imagens, o problema da nossa solução inicial (seção Introdução) era que, ao adicionar uma nova imagem, a imagem prévia era perdida.

E os botões "⬅️" e "➡️" precisavam ser implementados.

Vamos resolver a primeira parte (guardar a imagem prévia).

A estratégia será ter uma variável global do tipo array:

- `const images = []; // array vazio (i.e. sem nenhum elemento)`

Quando o usuário adicionar um novo link, nós o colocaremos no final do array (com push).

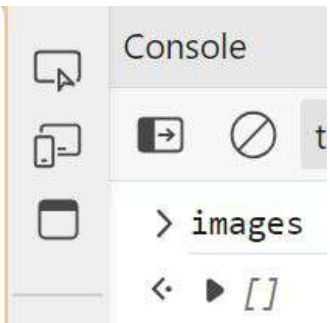
O código fica assim:

```
const input = document.getElementById("input-image");
const buttonAdd = document.getElementById("button-add");
const buttonPrev = document.getElementById("button-prev");
const buttonNext = document.getElementById("button-next");
const img = document.getElementById("img-display");

const images = []; // código novo: array global

buttonAdd.addEventListener("click", function () {
  const imageLink = input.value;
  img.src = imageLink;

  images.push(imageLink); // código novo: adicionar o link no final do array
});
```

Com isso, o array vai acumulando todos os links já adicionados.

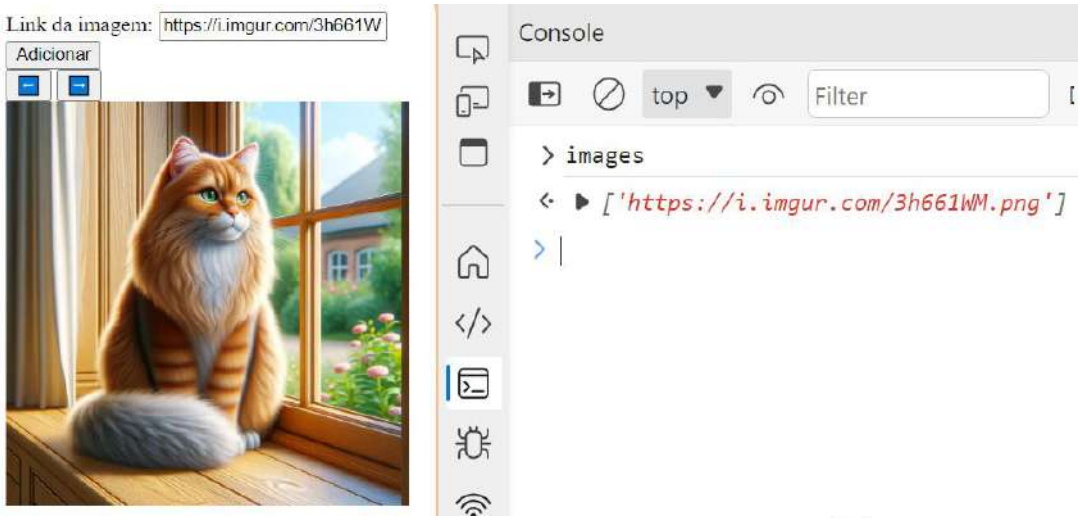
Veja:

Antes de adicionar nenhum link:

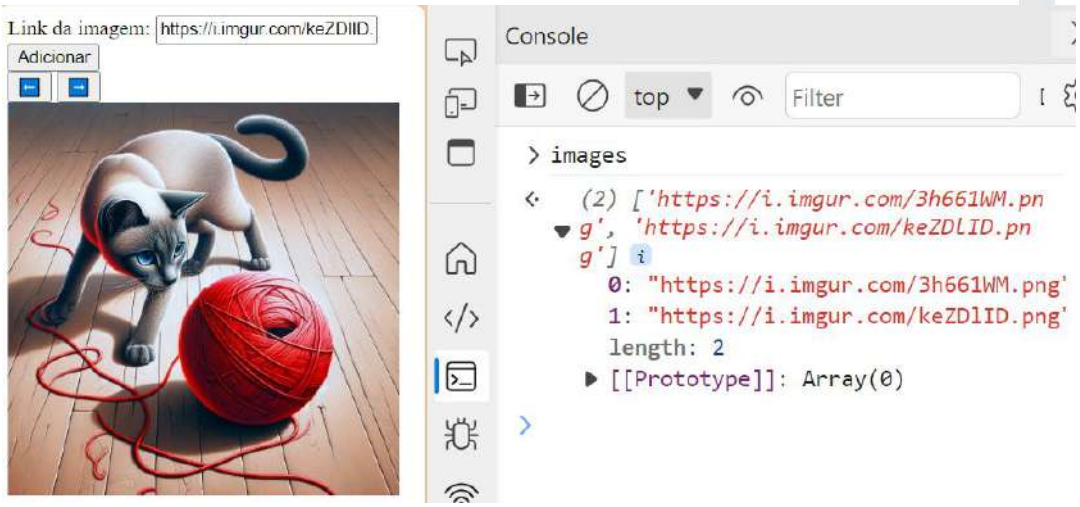


Operações básicas em arrays

Logo após adicionar um link:

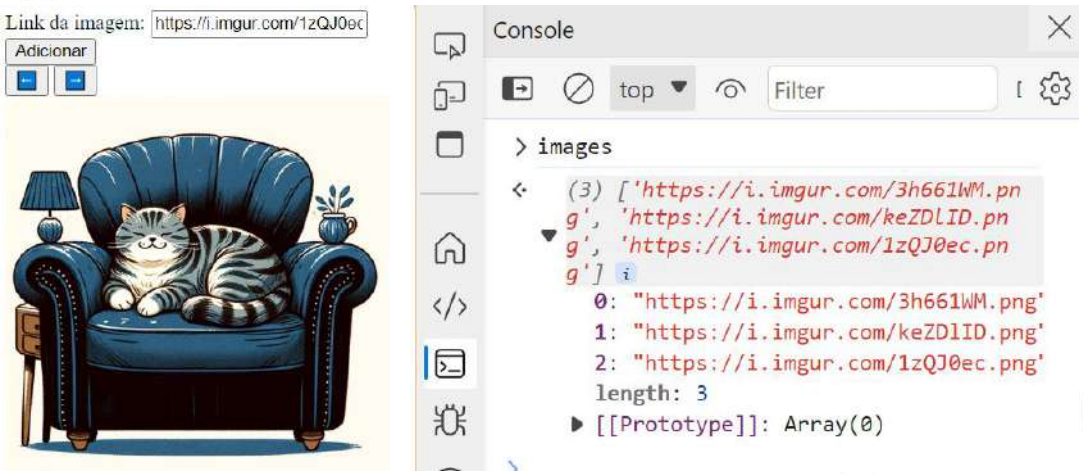


Logo após adicionar mais um link:



Operações básicas em arrays

Logo após adicionar um terceiro link:



Como visto, a `<img>` está sempre exibindo o link recém-adicionado, mas o array contém todos os links (o último link sendo a imagem mais recente).

Agora vamos fazer os botões "⬅️" e "➡️" funcionarem, começando pelo "⬅️".

Tendo adicionado as 3 imagens, ao clicar nesse botão, a segunda imagem deve ser exibida. Ao clicar de novo, a primeira imagem deve ser exibida.

Em outras palavras:

- Estamos vendo atualmente a terceira imagem, que é o índice `2` do array.
- Ao clicar no "⬅️", veremos a imagem do índice `1` do array.
- Ao clicar de novo no "⬅️", veremos a imagem do índice `0` do array.

Então o javascript precisa saber *"qual índice estamos vendo agora ?"* para que, ao clicar no botão, ele diminua esse índice em 1 unidade e pegue a imagem correspondente.

Para isso, vamos criar uma variável global `currentIndex` ("índice atual").

Dentro da função de adicionar imagem, vamos atribuir a essa variável o valor `array.length - 1`:

```
...

let currentIndex; // código novo

buttonAdd.addEventListener("click", function () {
  const imageLink = input.value;
  img.src = imageLink;

  images.push(imageLink);

  currentIndex = images.length - 1; // código novo
});
```

Operações básicas em arrays

Isso garante que o javascript sabe o índice em que estamos:

- Quando adicionamos a primeira imagem, o array `images` tinha tamanho 1.
  Logo `currentIndex = 0`, significa que estamos vendo a imagem do índice `0`.

- Quando adicionamos a segunda imagem, o array `images` ficou com tamanho 2.
  Logo `currentIndex = 1`, significa que estamos vendo a imagem do índice `1`.

- Finalmente, quando adicionamos a terceira imagem, o array `images` ficou com tamanho 3.
  Logo `currentIndex = 2`, significa que estamos vendo a imagem do índice `2`.

E agora o botão "⬅️" é fácil de fazer, fica assim:

```
...

buttonPrev.addEventListener("click", function () {
  currentIndex--; // diminuir em 1 unidade

  const imageLink = images[currentIndex]; // pegar o link da imagem

  img.src = imageLink; // colocar o link no <img src="...">
});
```

Assim:

- Se já adicionamos 3 imagens e estamos vendo a 3ª, o `currentIndex` vale 2.
- Ao clicar no botão "⬅️", o `currentIndex` se tornará `1`, e vai pegar a imagem do índice `1`.
- Ao clicar no botão "⬅️" de novo, o `currentIndex` se tornará `0`, e vai pegar a imagem do índice `0`.

Porém existe um bug: agora que o `currentIndex` vale `0`, ao clicar no botão "⬅️", o `currentIndex` vai se tornar `-1`.

Isso não faz sentido, pois ele vai em seguida tentar acessar `images[-1]`, que vai dar resultado **`undefined`**.

O comportamento desejado é: se estamos no índice `0`, devemos "dar a volta", ou seja, exibir a última imagem da lista.

Isso pode ser corrigido com um condicional, o código do botão fica assim:

Operações básicas em arrays

```
...

buttonPrev.addEventListener("click", function () {
  if (currentIndex === 0) {
    currentIndex = images.length - 1; // último índice da lista
  }
  else {
    currentIndex--;
  }

  const imageLink = images[currentIndex];

  img.src = imageLink;
});
```

E agora esse botão funciona perfeitamente.
Por fim, seguindo uma lógica parecida, podemos implementar o outro botão "➡️":

```
...

buttonNext.addEventListener("click", function () {
  // se estamos no último índice da lista,
  // vamos dar a volta (voltar pro 0)
  if (currentIndex === images.length - 1) {
    currentIndex = 0;
  }
  // senão vamos avançar em 1 unidade
  else {
    currentIndex++;
  }

  const imageLink = images[currentIndex];

  img.src = imageLink;
});
```

Operações básicas em arrays

Para que você possa testar mais facilmente, segue o código completo:

```
const input = document.getElementById("input-image");
const buttonAdd = document.getElementById("button-add");
const buttonPrev = document.getElementById("button-prev");
const buttonNext = document.getElementById("button-next");
const img = document.getElementById("img-display");

let currentIndex;
const images = [];

buttonAdd.addEventListener("click", function () {
  const imageLink = input.value;
  img.src = imageLink;

  images.push(imageLink);

  currentIndex = images.length - 1;
});

buttonPrev.addEventListener("click", function () {
  if (currentIndex === 0) {
    currentIndex = images.length - 1;
  } else {
    currentIndex--;
  }

  const imageLink = images[currentIndex];

  img.src = imageLink;
});

buttonNext.addEventListener("click", function () {
  if (currentIndex === images.length - 1) {
    currentIndex = 0;
  } else {
    currentIndex++;
  }

  const imageLink = images[currentIndex];

  img.src = imageLink;
});
```

Na verdade, ainda tem um bug

Esse código funciona se você adicionar pelo menos 1 imagem antes de começar a usar os botões "⬅️" e "➡️".

Mas se, sem adicionar imagem nenhuma, você clicar nos botões, o comportamento sai fora do esperado.

O que exatamente acontece, e como resolver, vamos deixar como exercício.

Mais operações em arrays

Introdução

Vamos continuar a adicionar funcionalidades na aplicação da galeria de imagens, partindo de onde paramos no final da última seção.

Deletar imagem da galeria: método `splice()`

No meio dos botões "⬅️" e "➡️", vamos adicionar um botão de deletar com ícone de uma lixeira: "🗑️".

Ao ser clicado, esse botão deleta a imagem que está sendo exibida agora.

Por exemplo, suponha que já adicionamos três imagens A, B e C, depois clicamos no botão "⬅️" e portanto está sendo exibida a imagem B. Se clicarmos no botão de deletar, a imagem B será deletada e passaremos a ver a imagem seguinte (C) no seu lugar.

Como sobraram somente A e C, os botões "⬅️" e "➡️" navegam entre elas, pois a imagem B não existe mais no array:

- `[A, B, C]` ⇒ `[A, C]`

Para deletar um elemento de um array, usamos o método `splice(《início》, 《quantidade》)`.

Onde `《início》` é o índice que queremos deletar (no exemplo, é o índice `1`).

E `《quantidade》` é a quantidade de elementos a serem deletados, pois o `splice` permite deletar vários começando no índice solicitado. No nosso caso, queremos deletar somente `1` elemento.

Então ficaria: `images.splice(1, 1)`

Mas é claro que o índice que queremos deletar depende de qual imagem estamos vendo, se estivéssemos vendo a imagem A (das 3 imagens A, B e C), deveríamos deletar o índice `0`, e ficariam sobrando as imagens B e C.

Lembrando que o índice da imagem exibida é armazenado na variável `currentIndex`, o botão de deletar deverá executar o código `images.splice(currentIndex, 1)`.

Veja que os índices se reorganizam ao fazer a exclusão, os elementos que estavam à direita do elemento deletado se "deslocam" para a esquerda:

Mais operações em arrays

Uma primeira versão do código desse botão fica assim (supondo que adicionamos um `<button id="button-trash">🗑️</button>` ao HTML):

```
...

const buttonTrash = document.getElementById("button-trash");

buttonTrash.addEventListener("click", function () {
  // por exemplo, suponha que o array tem 3 imagens: [A, B, C].
  // E o currentIndex vale 1 (i.e. estamos vendo a imagem B).
  // O splice vai deletar o índice 1, com isso o array fica [A, C]
  images.splice(currentIndex, 1);

  // pegar a imagem que foi deslocada para o índice,
  // no exemplo acima é a imagem C
  const newImage = images[currentIndex];

  img.src = newImage;
});

...
```

Mas esse código ainda tem bug:

O que acontece se estamos vendo a última imagem da lista e clicamos em deletar ?

Por exemplo a lista tem 3 imagens `[A, B, C]`, estamos vendo a imagem C, e clicamos em deletar:

- O `currentIndex` vale 2.
- O `splice` vai deletar o índice 2 (que é a imagem C).
- O array fica `[A, B]`, e o `currentIndex` continua em 2, claro.
- Mas agora o código tenta acessar `images[currentIndex]`, só que `images[2]` é **`undefined`** porque não sobrou nenhuma imagem no índice 2 já que ele era o último índice.

Então o código precisa tomar cuidado:

- Se ele deleta um índice que *não é o último* da lista, a imagem seguinte passa a ocupar aquele índice, então o código funciona.
- Mas se ele deleta o *último índice* da lista, aquele índice fica sem imagem.

  Nesse caso precisamos decidir o que fazer: podemos mostrar a última imagem que sobrou no array, ou então mostrar a primeira imagem do array.

  A escolha é arbitrária (depende do que queremos para nossa aplicação), e aqui vamos escolher mostrar a primeira imagem.

Mais operações em arrays

O código fica assim:

```
buttonTrash.addEventListener("click", function () {
  images.splice(currentIndex, 1);

  // A lógica desse if é: suponha que tínhamos 3
  // imagens [A, B, C] e currentIndex vale 2.
  // Então deletamos a imagem, o array fica [A, B]
  // e currentIndex continua valendo 2, que é o tamanho
  // do array [A, B].
  // Essa lógica vale para mais imagens também:
  // se o array tinha 5 imagens e estávamos na última
  // (currentIndex vale 4), ao deletá-la o array
  // fica com 4 elementos.
  if (currentIndex === images.length) {
    currentIndex = 0; // voltar para a primeira imagem
  }

  const newImage = images[currentIndex];

  img.src = newImage;
});
```

Salvar a galeria no `localStorage` com JSON

Da maneira como o código está até agora, ao recarregar a página, voltamos da estaca zero (imagens são perdidas).

Se quisermos salvar as informações para recuperar quando a página recarregar, você deve concordar que essa aplicação tem duas informações a serem guardadas:

- variável `currentIndex`, que é um número
- variável `images`, que é um array

Uma possibilidade é salvar cada uma separadamente no `localStorage`.

Para o `currentIndex`, podemos fazer assim:

- Para salvar: `localStorage.setItem("currentIndex", currentIndex)`, o valor numérico será convertido implicitamente em string (pois, como você já sabe, o `localStorage` só armazena strings)
- Para recuperar: `let currentIndex = Number(localStorage.getItem("currentIndex"))`, precisamos converter para número porque vai vir como string do `localStorage`

Mais operações em arrays

Já o array `images` requer uso de JSON, porque é um objeto (afinal array é um tipo de objeto):

- Para salvar: `localStorage.setItem("images", JSON.stringify(images))` , veja:

Javascript

```
> const images = ["A", "B", "C"]
< undefined
> localStorage.setItem("images", JSON.stringify(images))
< undefined
```

Local Storage

- Para recuperar: `const images = JSON.parse(localStorage.getItem("images"))`

Aí precisamos cuidar no código do carregamento e salvamento:

- No carregamento, tem aquele já conhecido problema: se o `localStorage` não tiver nada armazenado, virá como `null`, precisamos checar com `if`.
- Sobre o salvamento, precisamos salvar sempre que algo mudar: o `currentIndex` mudar, uma imagem for adicionada no array, ou for deletada do array.

Finalmente, note também que, ao iniciar a página, logo após carregar as informações do `localStorage`, precisamos exibir a imagem ! Senão as informações serão carregadas para as variáveis do javascript mas a `<img>` no HTML não vai exibir nada.

Com isso, o código completo fica assim (desconsiderando o potencial problema de o `localStorage` estar indisponível):

```
const input = document.getElementById("input-image");
const buttonAdd = document.getElementById("button-add");
const buttonPrev = document.getElementById("button-prev");
const buttonNext = document.getElementById("button-next");
const buttonTrash = document.getElementById("button-trash");
const img = document.getElementById("img-display");

// carregar currentIndex do localStorage
const currentIndexLocalStorage = localStorage.getItem("currentIndex");
let currentIndex =
  currentIndexLocalStorage === null ? 0 : Number(currentIndexLocalStorage);

// carregar images do localStorage
const imagesLocalStorage = localStorage.getItem("images");
const images =
  imagesLocalStorage === null ? [] : JSON.parse(imagesLocalStorage);

// se o array carregado tem alguma imagem, colocá-la na tela
if (images.length !== 0) {
  img.src = images[currentIndex];
}
```

(código continua na próxima página)

Mais operações em arrays

(continuação do código da página anterior)

```
buttonAdd.addEventListener("click", function () {
  const imageLink = input.value;
  img.src = imageLink;

  images.push(imageLink);

  currentIndex = images.length - 1;

  // salvar images e currentIndex
  localStorage.setItem("currentIndex", currentIndex);
  localStorage.setItem("images", JSON.stringify(images));
});

buttonPrev.addEventListener("click", function () {
  if (currentIndex === 0) {
    currentIndex = images.length - 1;
  } else {
    currentIndex--;
  }

  const imageLink = images[currentIndex];

  img.src = imageLink;

  // só precisamos salvar currentIndex porque images não mudou
  localStorage.setItem("currentIndex", currentIndex);
});

buttonNext.addEventListener("click", function () {
  if (currentIndex === images.length - 1) {
    currentIndex = 0;
  } else {
    currentIndex++;
  }

  const imageLink = images[currentIndex];

  img.src = imageLink;

  // só precisamos salvar currentIndex porque images não mudou
  localStorage.setItem("currentIndex", currentIndex);
});
```

(código continua na próxima página)

Mais operações em arrays

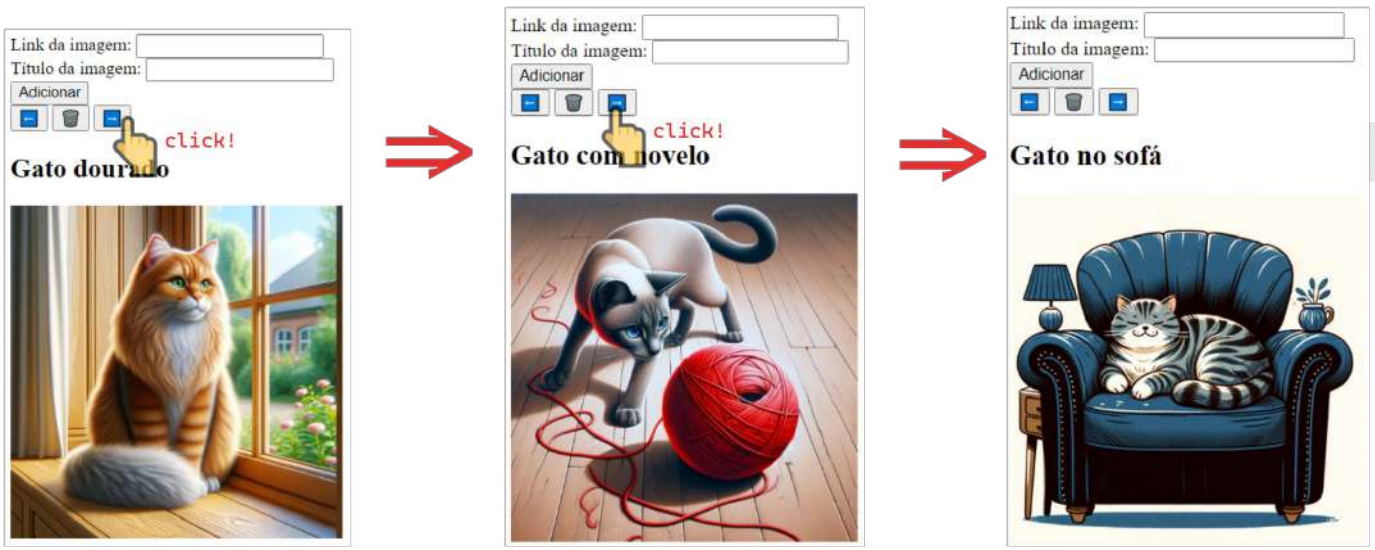
(continuação do código da página anterior)

```
buttonTrash.addEventListener("click", function () {
  images.splice(currentIndex, 1);

  if (currentIndex === images.length) {
    currentIndex = 0;
  }

  const newImage = images[currentIndex];

  img.src = newImage;

  // salvar images e currentIndex
  localStorage.setItem("currentIndex", currentIndex);
  localStorage.setItem("images", JSON.stringify(images));
});
```

Imagens com título: array de objetos

Vamos adicionar mais uma funcionalidade na página: o formulário de adição de imagem terá, além do input para o link da imagem, mais um input para o título da imagem.

Quando exibidas, as imagens aparecerão com o título em cima (num `<h2>`), ou seja, no final ficará assim:



A maneira mais fácil de fazer isso, mantendo todas as funcionalidades anteriores (deletar imagens e salvamento automático) é fazer um *array de objetos*.

Ou seja, quando o usuário clicar no botão "Adicionar", vamos criar um objeto contendo as duas informações da imagem, por exemplo:

```
{
  link: "https://i.imgur.com/3h661WM.png",
  title: "Gato dourado"
}
```

Mais operações em arrays

E aí, em vez de adicionar somente o link da imagem no array `images`, vamos adicionar o objeto.
Ao adicionar as 3 imagens da figura acima, por exemplo, o array vai ficar assim:

```
[
  {
    link: "https://i.imgur.com/3h661WM.png",
    title: "Gato dourado",
  },
  {
    link: "https://i.imgur.com/keZDlID.png",
    title: "Gato com novelo",
  },
  {
    link: "https://i.imgur.com/1zQJ0ec.png",
    title: "Gato no sofá",
  },
];
```

Para acessar os elementos desse array, não tem nada novo: cada elemento é um objeto, por exemplo `array[0]` é o primeiro objeto.
Então `array[0].link` é o link da primeira imagem, e `array[0].title` é o título. Veja:

```
> images[0]
◂ ▶ {link: 'https://i.imgur.com/3h661WM.png', title: 'Gato dourado'}
> images[0].link
◂ 'https://i.imgur.com/3h661WM.png'
> images[0].title
◂ 'Gato dourado'
```

O código javascript completo fica assim (comentários explicando as novidades):

```
const inputLink = document.getElementById("input-image");
// 1: o HTML tem um input para o título agora
const inputTitle = document.getElementById("input-title");
const buttonAdd = document.getElementById("button-add");
const buttonPrev = document.getElementById("button-prev");
const buttonNext = document.getElementById("button-next");
const buttonTrash = document.getElementById("button-trash");
// 2: o HTML tem um h2 em cima da imagem agora
const h2 = document.getElementById("image-title");
const img = document.getElementById("img-display");
```

(código continua na próxima página)

Mais operações em arrays

(continuação do código da página anterior)

```
const currentIndexLocalStorage = localStorage.getItem("currentIndex");
let currentIndex =
  currentIndexLocalStorage === null ? 0 : Number(currentIndexLocalStorage);

const imagesLocalStorage = localStorage.getItem("images");
const images =
  imagesLocalStorage === null ? [] : JSON.parse(imagesLocalStorage);

if (images.length !== 0) {
  // 6: pegar um elemento do array. Agora
  // esse elemento é um objeto
  const imageObj = images[currentIndex];

  // 7: acessar as propriedades `link` e `title` do objeto,
  // colocar no HTML
  img.src = imageObj.link;
  h2.innerText = imageObj.title;
}

buttonAdd.addEventListener("click", function () {
  const imageLink = inputLink.value;
  // 3: ler o valor do input de título
  const imageTitle = inputTitle.value;

  img.src = imageLink;
  // 4: colocar o título da nova imagem no h2
  h2.innerText = imageTitle;

  // 5: adicionar um novo elemento no array.
  // esse elemento é um objeto {link ..., title: ...}
  images.push({ link: imageLink, title: imageTitle });

  currentIndex = images.length - 1;

  localStorage.setItem("currentIndex", currentIndex);
  localStorage.setItem("images", JSON.stringify(images));
});
```

(código continua na próxima página)

Mais operações em arrays

(continuação do código da página anterior)

```
buttonPrev.addEventListener("click", function () {
  if (currentIndex === 0) {
    currentIndex = images.length - 1;
  } else {
    currentIndex--;
  }

  // 6: pegar um elemento do array. Agora
  // esse elemento é um objeto
  const imageObj = images[currentIndex];

  // 7: acessar as propriedades `link` e `title` do objeto,
  // colocar no HTML
  img.src = imageObj.link;
  h2.innerText = imageObj.title;

  localStorage.setItem("currentIndex", currentIndex);
});

buttonNext.addEventListener("click", function () {
  if (currentIndex === images.length - 1) {
    currentIndex = 0;
  } else {
    currentIndex++;
  }

  // 6: pegar um elemento do array. Agora
  // esse elemento é um objeto
  const imageObj = images[currentIndex];

  // 7: acessar as propriedades `link` e `title` do objeto,
  // colocar no HTML
  img.src = imageObj.link;
  h2.innerText = imageObj.title;

  localStorage.setItem("currentIndex", currentIndex);
});
```

(código continua na próxima página)

Mais operações em arrays

(continuação do código da página anterior)

```
buttonTrash.addEventListener("click", function () {
  images.splice(currentIndex, 1);

  if (currentIndex === images.length) {
    currentIndex = 0;
  }

  // 6: pegar um elemento do array. Agora
  // esse elemento é um objeto
  const imageObj = images[currentIndex];

  // 7: acessar as propriedades `link` e `title` do objeto,
  // colocar no HTML
  img.src = imageObj.link;
  h2.innerText = imageObj.title;

  localStorage.setItem("currentIndex", currentIndex);
  localStorage.setItem("images", JSON.stringify(images));
});
```

Note especialmente que as grandes mudanças foram:

- O código `images.push(...)`, pois agora adicionamos um objeto no array, não uma string (link da imagem) como antes.
- Os códigos de acesso ao array `images[currentIndex]` agora recuperam um objeto `{link: ..., title: ...}`, não uma string como era antes (link da imagem).

Fora isso, não precisamos alterar:

- A lógica de deletar imagem, pois continua deletando um dos elementos do array (agora um elemento é um objeto)
- A lógica de salvar no Local Storage, pois continua salvando o array inteiro (com seus elementos do tipo objeto), veja como ele fica no Local Storage agora:

Key	Value
currentIndex	5
images	[{"link":"https://i.imgur.com/3h661WM.png","title":"Gato dourado"},{"link":"https://i.imgur.com/keZDlID.p...

Fundamentos sobre loops

Introdução

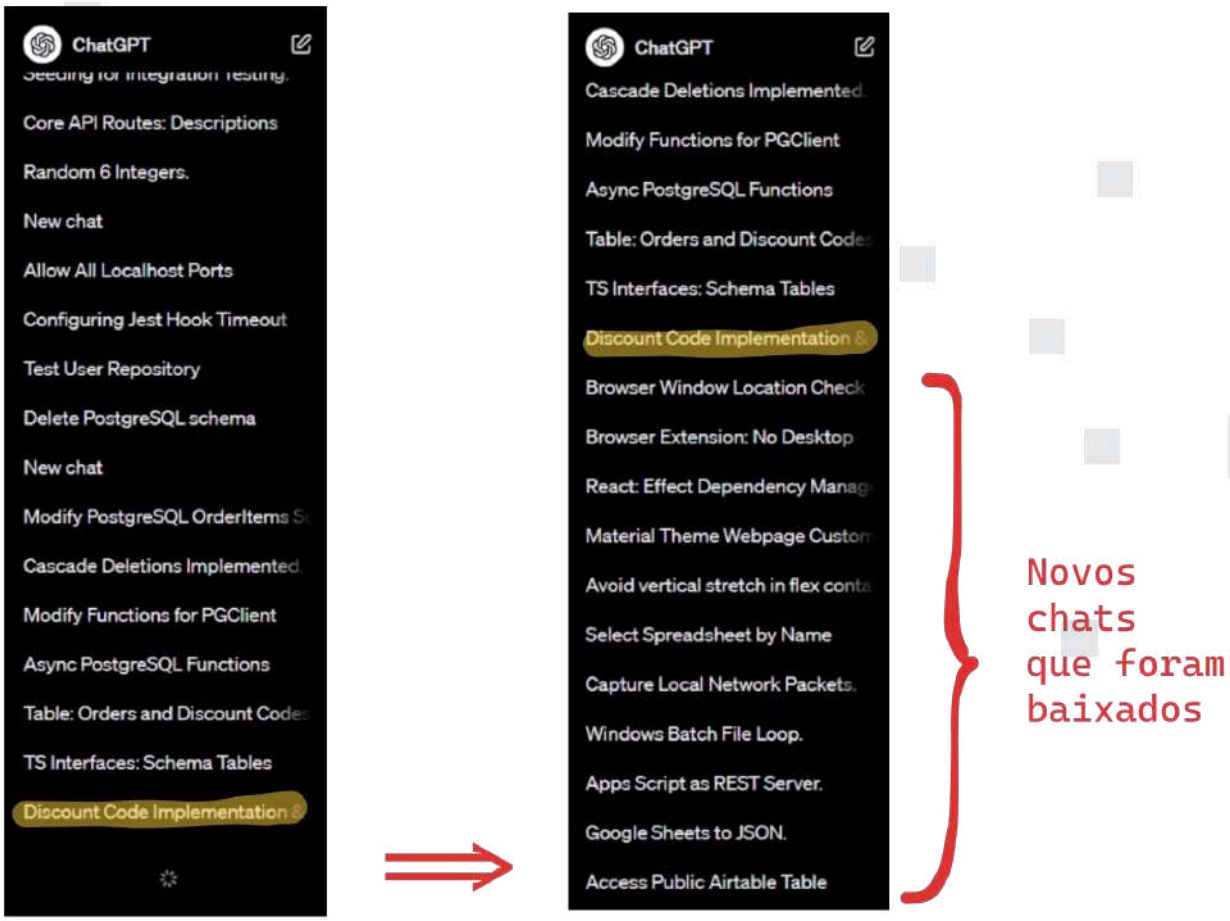
Veremos agora nosso primeiro comando de repetição (do inglês *loop*: "laço", "volta").

Ele serve para executar certas linhas do código *várias vezes* em vez de *uma vez só* como é o normal.

Isso é uma necessidade extremamente comum em websites. Vejamos um exemplo:

No site do Chat GPT, quando você chega ao final da sua lista de chats, aparece um ícone de carregamento que se parece com uma engrenagem ⚙️.

Instantes depois, os chats são carregados e aparecem na lista:



O que está acontecendo é:

- Primeiro os novos chats são baixados da internet (do servidor do Chat GPT).
  Não sabemos fazer esse tipo de download ainda, nem é o objetivo agora.
- Uma vez baixados, o javascript tem um array contendo os novos chats.
  Algo como: `["Browser Window Location Check", "Browser Extension", "React: Effect Dependency", ...]`
- Esse array é uma variável no javascript, no HTML ainda não aparecem os novos chats.
  Agora o código precisa inserir esses novos chats na lista de chats do HTML.
  Isso pode ser feito com um `createElement()` e `appendChild()` para cada um dos novos chats.
- E é aqui que está a necessidade de um comando de repetição:
  O array pode ter vários elementos (vários chats), então o `createElement()` e `appendChild()` precisam ser repetidos várias vezes, para adicionar um a um dos novos chats.

Fundamentos sobre loops

Motivação: lista de tarefas com salvamento automático

Para tornar essa ideia mais palpável, vamos começar com o já conhecido aplicativo de lista de tarefas.
Uma primeira versão básica dele seria assim:

Um `<input>` para o texto da tarefa, um `<button>` para adicionar, e embaixo disso uma `<ul>` onde cada tarefa adicionada aparece como uma `<li>`.
O código para isso é:

```html
<!-- body da página -->
<body>
  Nova tarefa: <input id="task-input" />
  <button id="add-button">Adicionar</button>

  <ul id="tasklist"></ul>
  <script src="index.js"></script>
</body>
```

```javascript
// index.js

const taskInput = document.getElementById("task-input"); // <input>
const addButton = document.getElementById("add-button"); // <button>
const tasklist = document.getElementById("tasklist"); // <ul>

addButton.addEventListener("click", addTask);

function addTask() {
  const taskText = taskInput.value;

  const newLi = document.createElement("li");
  newLi.innerText = taskText;

  tasklist.appendChild(newLi);
}
```

Fundamentos sobre loops

Agora suponha que queremos salvar as tarefas no Local Storage, para que elas não sejam perdidas ao fechar a página.

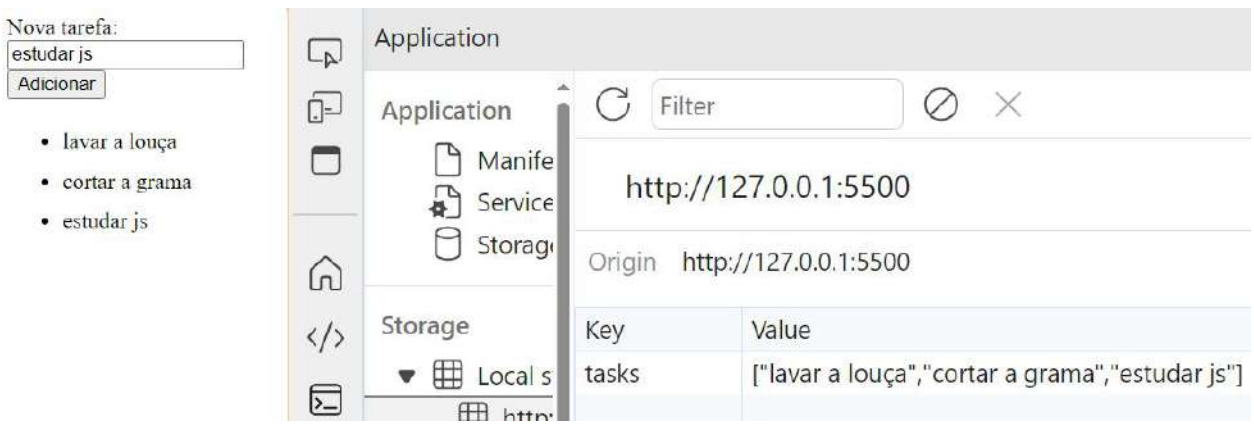
Você já sabe (do exemplo da galeria de imagens) que podemos usar um array global para guardar todas as tarefas adicionadas, e salvar esse array no Local Storage.

Pensando no salvamento, o código fica assim (comentários numerados indicam as mudanças):

```
const taskInput = document.getElementById("task-input"); // <input>
const addButton = document.getElementById("add-button"); // <button>
const tasklist = document.getElementById("tasklist"); // <ul>

addButton.addEventListener("click", addTask);

// 1: array global, vai guardar todas
// as tarefas, exemplo ["lavar a louça", "cortar a grama", ...]
const tasks = [];

function addTask() {
  const taskText = taskInput.value;

  const newLi = document.createElement("li");
  newLi.innerText = taskText;

  tasklist.appendChild(newLi);

  // 2: adicionar a tarefa (string) no final do array
  tasks.push(taskText);

  // 3: salvar o array no Local Storage
  localStorage.setItem("tasks", JSON.stringify(tasks));
}
```

Isso guarda todas as tarefas num array, e salva o array inteiro no Local Storage sempre que uma nova tarefa é adicionada.

Veja como fica depois de adicionar três tarefas:



Fundamentos sobre loops

Mas esse foi só metade do problema: salvar.

Tem ainda a outra metade: ao reabrir a página, fazer o carregamento.

Parte desse problema você já sabe resolver, mudando a inicialização da variável `tasks` (comentário numerado indica a mudança):

```
const taskInput = document.getElementById("task-input"); // <input>
const addButton = document.getElementById("add-button"); // <button>
const tasklist = document.getElementById("tasklist"); // <ul>

addButton.addEventListener("click", addTask);

// 1: alterado para carregar do Local Storage quando abrir a página
const tasksLocalStorage = localStorage.getItem("tasks");
const tasks = tasksLocalStorage === null ? [] : JSON.parse(tasksLocalStorage);

function addTask() {
  const taskText = taskInput.value;

  const newLi = document.createElement("li");
  newLi.innerText = taskText;

  tasklist.appendChild(newLi);

  tasks.push(taskText);

  localStorage.setItem("tasks", JSON.stringify(tasks));
}
```

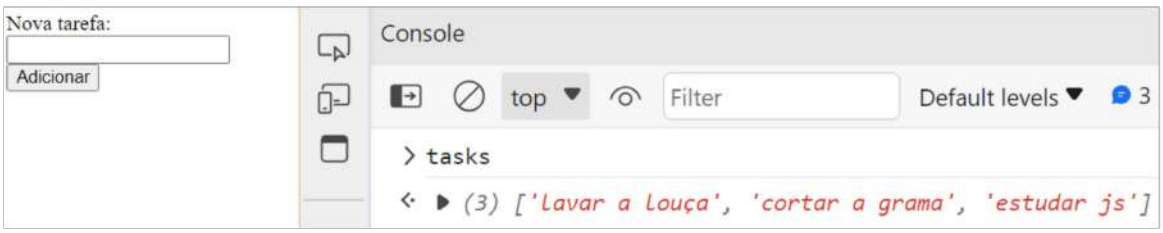
Com isso, se houver dados no Local Storage, a variável `tasks` terá o array de dados (e se o Local Storage não tiver nada, a `tasks` terá um array vazio `[]`).

Não sei se você percebeu, mas isso não é tudo que precisa ser feito no momento do carregamento das tarefas.

Vamos fazer o teste para ver o que falta:

- Lembre que já adicionamos 3 tarefas e elas estão no Local Storage.
- Agora salve o novo código javascript e recarregue a página.

O resultado será o seguinte:



✗ 2: Mas o HTML não tem tarefa nenhuma

✓ 1: A variável `tasks` contém as tarefas carregadas do Local Storage (funcionou!)

Fundamentos sobre loops

É evidente que o HTML não mostra tarefa nenhuma ao recarregar a página.

Afinal o código javascript carregou as tarefas na variável `tasks` *mas não as adicionou ao HTML*.

Isso está errado, quando a página inicia, ela deve carregar do Local Storage mas também mostrar no HTML as tarefas carregadas.

Para corrigir isso, precisamos inserir um novo código no escopo global, logo após o código de carregamento do Local Storage.

O novo código será responsável por adicionar as tarefas no HTML, usando a mesma estratégia da função `addTask`, ou seja, usando os comandos `createElement()` e `appendChild()`.

Mas tem um problema: precisamos adicionar no HTML *todas as tarefas* do array `tasks`, então de quantos `createElement("li")` e quantos `appendChild` vamos precisar ?

Não tem como saber a princípio, porque o array de tarefas pode ter muitas tarefas.

Entra como solução o comando de repetição **`for of`**.

Fundamentos sobre loops

Comando `for of`

Vamos apresentar a solução de uma vez, o código de que precisamos é o seguinte (comentário numerado indica a mudança):

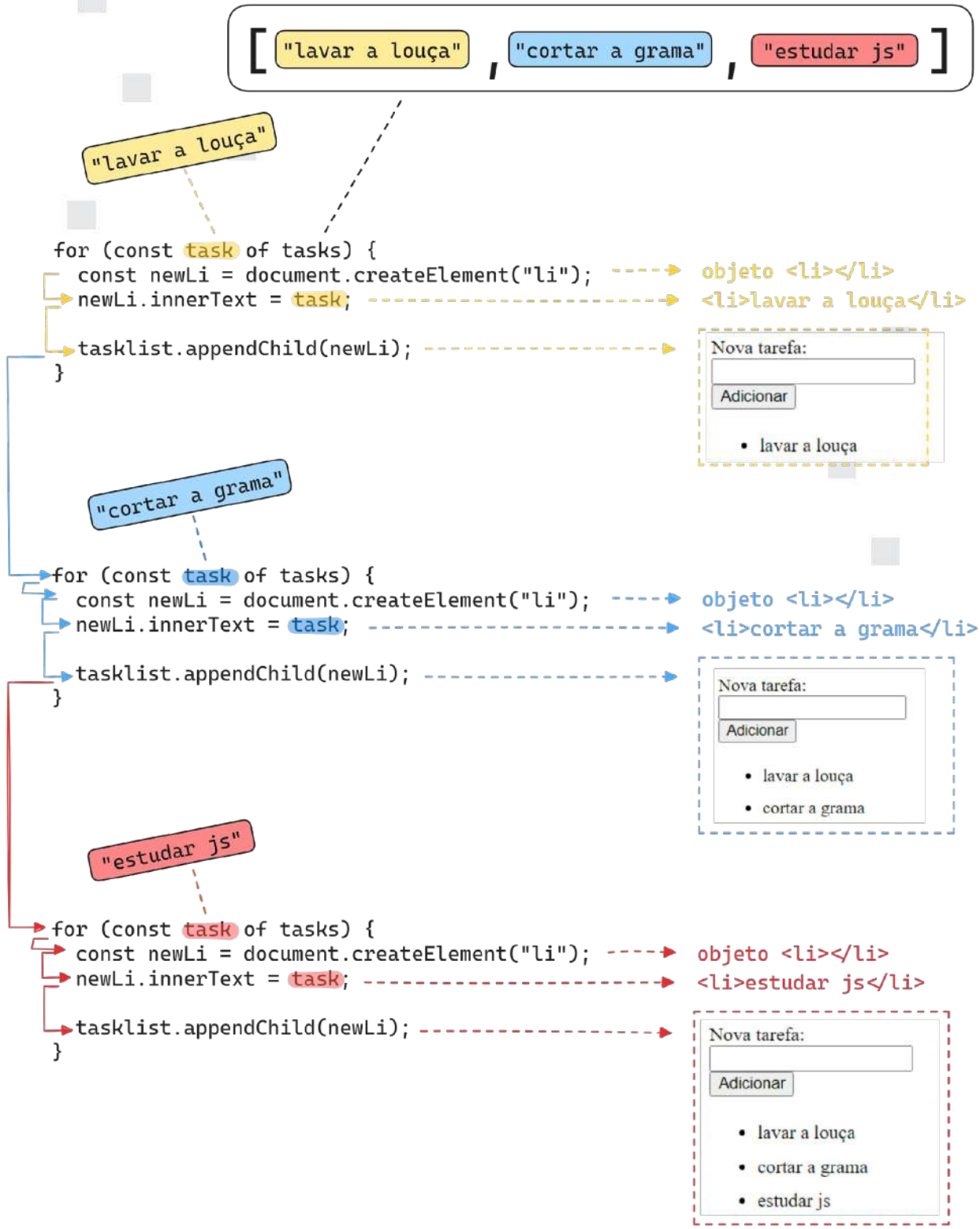
```
const taskInput = document.getElementById("task-input"); // <input>
const addButton = document.getElementById("add-button"); // <button>
const tasklist = document.getElementById("tasklist"); // <ul>

addButton.addEventListener("click", addTask);

const tasksLocalStorage = localStorage.getItem("tasks");
const tasks = tasksLocalStorage === null ? [] : JSON.parse(tasksLocalStorage);

// 1: para cada tarefa carregada, adicioná-la no HTML
for (const task of tasks) {
  const newLi = document.createElement("li");
  newLi.innerText = task;

  tasklist.appendChild(newLi);
}

function addTask() {
  const taskText = taskInput.value;

  const newLi = document.createElement("li");
  newLi.innerText = taskText;

  tasklist.appendChild(newLi);

  tasks.push(taskText);

  localStorage.setItem("tasks", JSON.stringify(tasks));
}
```

A sintaxe desse novo comando é a seguinte:

```
for (《declaração de variável》 of 《array》) {
  《 comando 1 》;
  《 comando 2 》;
  ...
}
```

Fundamentos sobre loops

No exemplo da lista de tarefas, ele funciona conforme o diagrama abaixo:



Fundamentos sobre loops

Ou seja, os comandos que estão dentro das chaves `{ }` são repetidos várias vezes, uma vez para cada elemento da lista.

Em cada repetição, a variável `task` contém um dos elementos (strings) do array `tasks`.

É comum chamar cada repetição de **Iteração** (sinônimo de repetição).

Então esse loop fez 3 iterações (3 repetições). Se o array tivesse mais elementos, seriam mais iterações, uma para cada elemento.

E se o array estivesse vazio (nenhuma tarefa), nenhuma iteração seria feita, como se o `for of` não existisse, o que dá o efeito correto: nenhuma tarefa seria adicionada no HTML, já que nenhuma tarefa está presente no array `tasks`.

A variável `task` poderia ter qualquer outro nome, as iterações funcionariam do mesmo jeito.

E qualquer comando pode ser usado dentro das chaves do `for of`, poderia por exemplo ter um `if-else` lá dentro, etc.

Fundamentos sobre loops

Escopo do `for of`

Você não achou estranho `const task` ?

Se ela é `const`, como ela pode se alterar de uma iteração para outra ? (lembrando que essa variável é uma string diferente em cada iteração)

A resposta é que não se trata de *uma única* variável `task`, mas sim *três* variáveis `task` diferentes.

Pense que cada iteração do loop é como se fosse a execução de uma função, e as variáveis `task` e `newLi` (esta última declarada dentro das chaves) são como "variáveis locais".

Em cada iteração, são criadas *uma nova* variável `task` e *uma nova* variável `newLi`, que só estão disponíveis durante aquela iteração.

Elas são chamadas de **Variáveis de Loop**: a variável que é declarada nos parênteses do `for of` (no caso a `task`) e as variáveis que forem declaradas *dentro* das chaves do loop (no caso `newLi`) são *variáveis locais* do loop, recriadas a cada iteração.

Então cada iteração tem variáveis únicas e independentes das variáveis das outras iterações.

E após o comando `for of`, não é possível utilizar mais essas variáveis, ou seja, a última linha do código abaixo vai dar um erro dizendo que `task` não está definida:

```
for (const task of tasks) {
  const newLi = ...
  ...
}

// Dá erro: variável `task` não existe fora do loop.
// Daria erro também console.log(newLi);
console.log(task);
```

Dizer que "cada iteração é como a execução de uma função" é uma metáfora.

Ela é útil porque, se você pensar assim, chegará nas conclusões corretas sobre o funcionamento das variáveis de loop.

Porém tecnicamente o comando `for of` *não é uma função*, os parênteses em `for (....) { ... }` não são os parênteses de uma função, são simplesmente parte da sintaxe do comando.

Fundamentos sobre loops

Outras táticas com loop

Mostramos o uso básico do comando `for of`, mas é possível pensar em variações.

Exemplo 1: loop sobre um array de objetos

Vamos voltar na galeria de imagens:

Naquela aplicação, cada imagem era um objeto `{link: ..., title: ...}` e todos os objetos ficavam guardados dentro de um array global `images`.

Na ocasião, estávamos exibindo somente uma imagem por vez , mas poderíamos desejar exibir todas ao mesmo tempo (por exemplo uma ao lado da outra).

Para isso, após fazer o carregamento do Local Storage quando a página é iniciada, faríamos em seguida um loop `for (const image of images)`.

Mas note que a variável `image` será um objeto nesse caso, já que `images` é um array de objetos.

Exemplo 2: loop com condicional dentro

Já dissemos que dentro do loop qualquer comando pode ser usado.

Vamos ver um exemplo onde um condicional `if` seria útil dentro de um loop:

Suponha que você tem uma página de calendário com um formulário onde o usuário preenche o nome do evento e a data, aí clicar em Adicionar.

Os eventos ficam guardados todos dentro de um array, cada evento é um objeto como `{description: "Reunião da turma", date: "2024-04-25"}`.

O usuário pode então selecionar uma data específica e pedir para a aplicação mostrar somente os eventos daquela data.

Para fazer isso, a aplicação poderia ter um código assim:

```
// supondo que `events` é o array de objetos,
// e que `selectedDate` é uma variável
// string contendo a data que o usuário
// quer ver
for (const event of events) {
  if (event.date === selectedDate) {
    // código para exibir o evento na página
  }
}
```

A lógica é:

- Fazemos um loop por todos os elementos (objetos) do array
- Verificamos se a data do evento é igual à data solicitada pelo usuário.
- Se sim, o código do `if` será executado e o evento vai aparecer na página (provavelmente dentro desse `if` tem um `createElement()` e um `appendChild()`)
- Mas se as datas não baterem, o código do `if` não vai executar, aí esta iteração do loop vai terminar sem ter feito nada.

  Portanto o evento não vai aparecer (como desejado).

Método querySelectorAll()

Introdução

Você deve se lembrar que o comando `querySelector` retorna *somente um* objeto do DOM.

Então, por exemplo, se uma página HTML tem vários `<p>` e você executa `document.querySelector("p")`, ele vai retornar o primeiro `<p>` que encontrar.

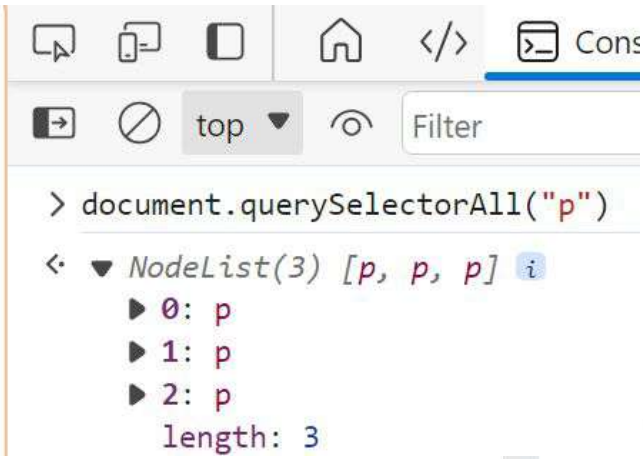
O `querySelectorAll` é parecido, mas em vez de retornar o primeiro `<p>`, ele retorna um *array com todos os objetos* `<p>`.

Veja um teste numa página que tem 3 `<p>`:

Parágrafo 1

Parágrafo 2

Parágrafo 3



É um array onde cada elemento é um objeto DOM (um dos `<p>`), na ordem em que eles aparecem na página.

Na verdade, o resultado do `querySelectorAll` não é *exatamente* um array, mas sim um *NodeList* (veja que aparece esse nome no print acima).

Na prática, é quase a mesma coisa que um array normal, só que um NodeList é um pouco mais "deficiente", ou seja, ele tem menos funcionalidades do que um array propriamente dito.

Entre todas as funcionalidades de array que já vimos, as únicas que o NodeList não possui são:

- `push()`
- `pop()`
- `splice()`

O resto funciona (acesso por índice, propriedade `length`, loop `for of`).

Método querySelectorAll()

Exemplo de uso: sumário automático

Suponha que você está escrevendo um documento (talvez um relatório) em HTML.
E ele é dividido em seções (`<section>`), cada uma delas com um título (`<h1>`):

```
<section>
  <h1>Introdução</h1>
  <p>blablabla</p>
  ...
</section>

<section>
  <h1>Materiais e métodos</h1>
  <p>blablabla</p>
  ...
</section>

<section>
  <h1>Resultado experimental</h1>
  <p>blablabla</p>
  ...
</section>

... etc ...
```

Tem muitas seções, você queria um código javascript para fazer um sumário automático.
Ou seja, você coloca um `<ul id="summary"></ul>` antes de todas as seções, e o javascript por conta própria preenche esse `<ul>` com várias `<li>`, cada uma sendo o título de uma das seções do documento:

```
<ul>
  <li>Introdução</li>
  <li>Materiais e métodos</li>
  <li>Resultado experimental</li>
  ...
</ul>
```

Método querySelectorAll()

Gerar o sumário automaticamente é legal porque é tedioso de fazer na mão, e se você alterar a ordem ou os títulos das seções, teria que consertar manualmente o sumário também.

O código javascript para fazer isso pode ser o seguinte:

```
// ul onde colocar os ítens do sumário
const ul = document.getElementById("summary");

// array com todas as section
const sectionList = document.querySelectorAll("section");

// para cada objeto section:
for (const section of sectionList) {
  // criar um objeto li
  const li = document.createElement("li");

  // encontrar o h1 que está dentro da section
  // (note que é section.querySelector para procurar
  // somente dentro da section, pois document.querySelector
  // procuraria no documento inteiro e acabaria pegando
  // o h1 errado)
  const h1 = section.querySelector("h1");

  // colocar no texto da li o que está no texto do h1
  // (i.e. o título da seção)
  li.innerText = h1.innerText;

  // pendurar o li dentro da ul
  ul.appendChild(li);
}
```

Pode testar ! Se você tiver um documento como descrito (com uma `<ul id="summary">` vazia no topo), basta inserir esse script no final da página e abrir no navegador.

Ao abrir, o sumário vai aparecer automaticamente.

Outros comandos e táticas com loop

Sumário

Comando for clássico

Introdução: problema das linhas da planilha

O comando `for of` é muito prático para fazer repetições (iterações) sobre um array.

Porém, há situações em que você quer fazer repetições, mas não tem nenhum array envolvido.

Por exemplo, suponha que você está criando uma aplicação de planilha, como o Google Sheets ou o Microsoft Excel.

Uma funcionalidade básica da aplicação é ter várias linhas (as linhas da planilha, já supondo que você conhece como funciona uma planilha).

Digamos que serão 1000 linhas.

O seu código javascript precisa, ao abrir a página, criar 1000 linhas na planilha. Para simplificar, vamos simular isso com 1000 `<li>` dentro de uma `<ul>`.

Ou seja, queremos criar o seguinte:

```
<!-- id="rows" para facilitar a seleção no javascript -->
<ul id="rows">
  <li>1</li>
  <li>2</li>
  <li>3</li>
  ...
  <li>999</li>
  <li>1000</li>
</ul>
```

Não vamos escrever 1000 `<li>` manualmente... quando a página abrir, a `<ul>` estará vazia.

Não é prático escrever 1000 `<li>` manualmente, por isso quem vai criar as 1000 `<li>` será o código javascript.

Mas note que não tem nenhum array envolvido nessa história, então não tem como usar `for of`.

A página simplesmente inicia, e o código precisa dar algum jeito de criar as 1000 `<li>`. Como ?

Solução com for clássico

A solução que provavelmente 99% dos desenvolvedores iria pensar para esse problema é o comando `for` *clássico*, que não é o mesmo que o `for of`.

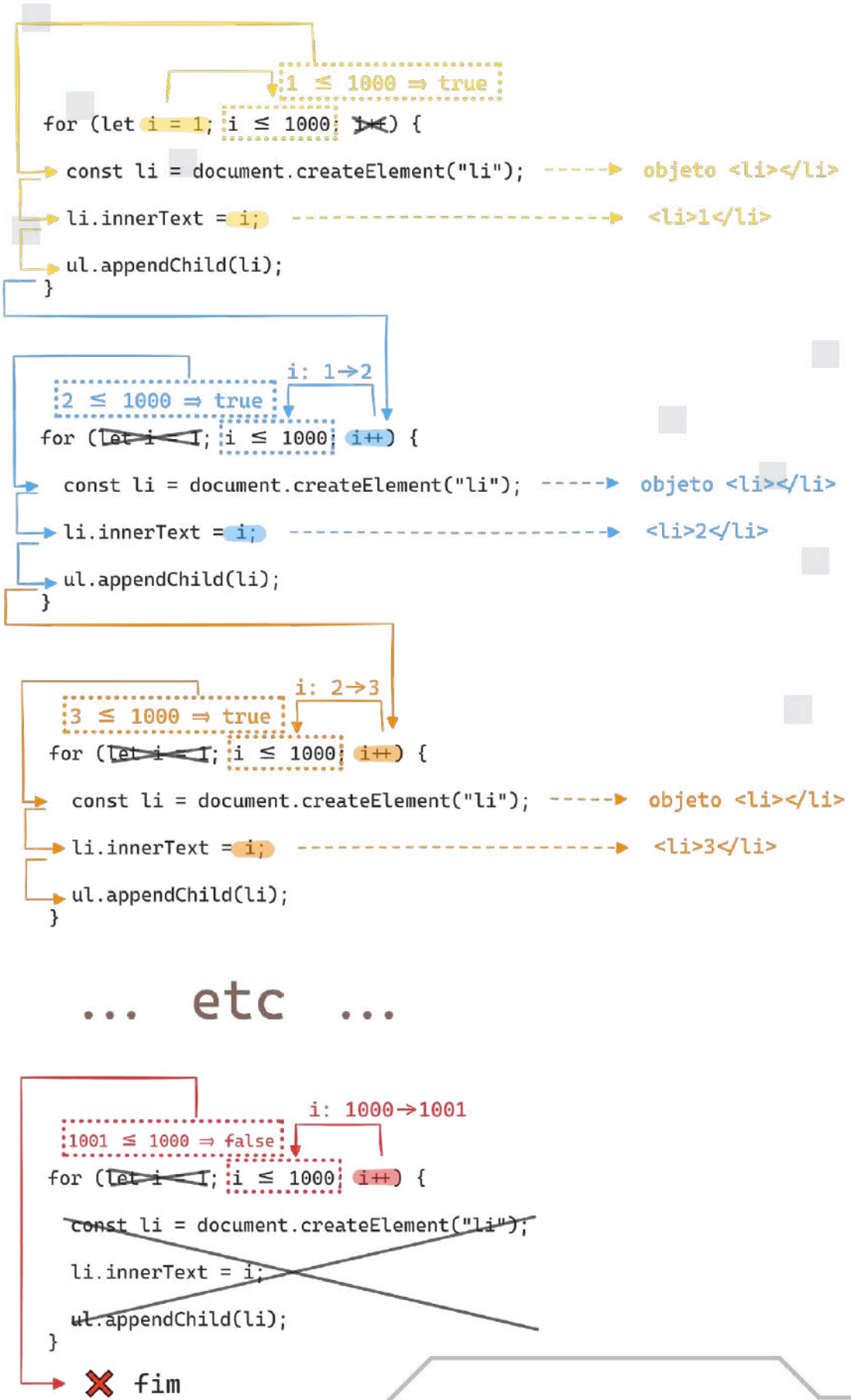
Estamos chamando de *clássico* porque esse comando é muito antigo, e a maioria das linguagens de programação possui esse *mesmo* comando com a *mesma* sintaxe (forma de escrever), funcionando da *mesma* maneira em todas essas linguagens.

Já começando pela solução, o código que resolve nosso problema é o seguinte (colocado no escopo global para executar quando a página inicia):

```
const ul = document.getElementById("rows");

for (let i = 1; i <= 1000; i++) {
  const li = document.createElement("li");

  li.innerText = i;

  ul.appendChild(li);
}
```

Comando for clássico

O comando `for` clássico funciona conforme o diagrama abaixo:



Comando for clássico

Dizendo de outra forma:

- Na primeira iteração, cria a variável `i = 1`.

  Depois verifica se `1 <= 1000`.

  Como deu **`true`**, executa os comandos dentro das chaves `{ }`.

- Na segunda iteração, primeiro faz `i++` : muda de 1 para 2.

  Depois verifica se `2 <= 1000`.

  Como deu **`true`**, executa os comandos dentro das chaves `{ }`.

- Na terceira iteração, primeiro faz `i++` : muda de 2 para 3.

  Depois verifica se `3 <= 1000`.

  Como deu **`true`**, executa os comandos dentro das chaves `{ }`.

- Etc.

- Na última iteração válida (que não está desenhada no diagrama, seria uma iteração antes da vermelha): muda `i` de 999 para 1000.

  Depois verifica se `1000 <= 1000`.

  Como deu **`true`**, executa os comandos dentro das chaves `{ }`

- Finalmente, na última vez (em vermelho no diagrama): muda `i` de 1000 para 1001.

  Depois verifica se `1001 <= 1000`.

  Dessa vez deu **`false`**.

  Então *não executa* o comando das chaves, mas sim *desiste* do loop.

Com esse mecanismo, em cada iteração o valor de `i` é uma unidade a mais que a iteração anterior.

A condição `i <= 1000` serve como um *guarda de proteção* para verificar se a iteração deve realmente acontecer:

Quando essa condição resulta em **`true`**, significa que a iteração *deve* acontecer.

E quando ela resulta em **`false`**, significa que a iteração *não deve* acontecer, e o loop deve ser interrompido (é interrompido na primeira vez que dá **`false`**).

Comando for clássico

Detalhes sobre o `for` clássico

1: Sintaxe

A sintaxe geral desse comando é a seguinte:

```
for (《declaração de variável》; 《condição》; 《atribuição》) {
  《comando 1 》;
  《comando 2 》;
  ...
}
```

Os parênteses são parte intrínseca do comando, mas *não quer dizer que é uma função*.

O comando `for` clássico não é uma função.

2: Variável contadora

A variável `i` é chamada *variável contadora* (porque ela serve para contar: 1, 2, 3 ... 1000).

Ela poderia ter qualquer outro nome de variável, o nome `i` é arbitrário.

É comum chamar de `i`, porque o nome é pequeno (fácil de escrever).

Como visto no diagrama, essa variável é declarada na primeira iteração.

3: Condição de parada

A condição `i <= 1000` é geralmente chamada de condição de parada, e poderia ser qualquer condição que você conhece (qualquer operador de comparação, e possivelmente `||` e `&&`).

Mas esse nome é um tanto estranho.

Em cada iteração, o `for` primeiro verifica se essa condição vale `true` e, em caso afirmativo, ele executa os comandos internos.

Então seria melhor chamar essa condição de *condição de continuidade* em vez de condição de parada.

Como visto no diagrama, essa condição vale `true` nas primeiras 1000 iterações.

Na iteração 1001, a condição dá `false` pela primeira vez.

Isso faz a iteração ser abortada (não executa os comandos internos) e o loop como um todo é abortado também (não vai tentar fazer outra iteração depois dessa).

4: Incremento

O comando `i++` é chamado de *incremento* ou *atualização*.

Ele acontece logo no início de cada iteração, antes de testar a condição.

5: Comandos dentro das chaves

Assim como no `for of`, qualquer comando pode ser colocado dentro das chaves `{ }`, por exemplo poderia ter um `if-else` lá dentro.

Comando for clássico

6: Regras de escopo

O comando `for` clássico tem as mesmas regras de escopo do `for of` que você já conhece.

Ou seja, a variável `i` e as variáveis declaradas dentro das chaves (nesse caso `const li`) são **variáveis de loop**.

Como você já sabe, significa que só podem ser usadas dentro das chaves `{ }`.

Outras variações do for clássico

Podemos mudar os 3 comandos dentro dos parênteses do `for` para fazer coisas diferentes com ele.

Não necessariamente são variações muito úteis, raramente você precisaria delas, mas vale saber que existem.

Variação 1: contagem decrescente

Para contar de 1000 até 1 (em vez de 1 até 1000), faríamos o seguinte:

```
for (let i = 1000; i >= 1; i--) {
  // comandos ...
}
```

Tente visualizar mentalmente as iterações disso.

Variação 2: contagem de 2 em 2 (ou 3 em 3, etc.)

Podemos contar de 1 até 1000 de 2 em 2, ou seja: 1, 3, 5, 7...

Seria assim:

```
for (let i = 1; i <= 1000; i += 2) {
  // comandos ...
}
```

Consegue perceber que o último número será 999 ? Depois dele seria 1001, mas para 1001 a condição será `false` e abortará o loop.

Variação 3: contagem de qualquer número até qualquer número

É possível alterar onde a contagem começa e onde termina facilmente.

Por exemplo, podemos pedir para o usuário definir esses dois limites:

```
const min = Number(prompt("Digite o número inicial"));
const max = Number(prompt("Digite o número final"));

// vai imprimir min, min+1, min+2, ..., max
for (let i = min; i <= max; i++) {
  console.log(i);
}
```

Comando for clássico

For clássico para iterar sobre array

É possível usar o `for` clássico para fazer a mesma coisa que o `for of`: iterar por todos os elementos de um array.

A ideia é que a variável `i` do `for` clássico vai ser o *índice* do array: começado em 0, depois 1, 2, etc.

Como o primeiro índice de um array é 0, a variável contadora do `for` será `let i = 0` para começar no zero.

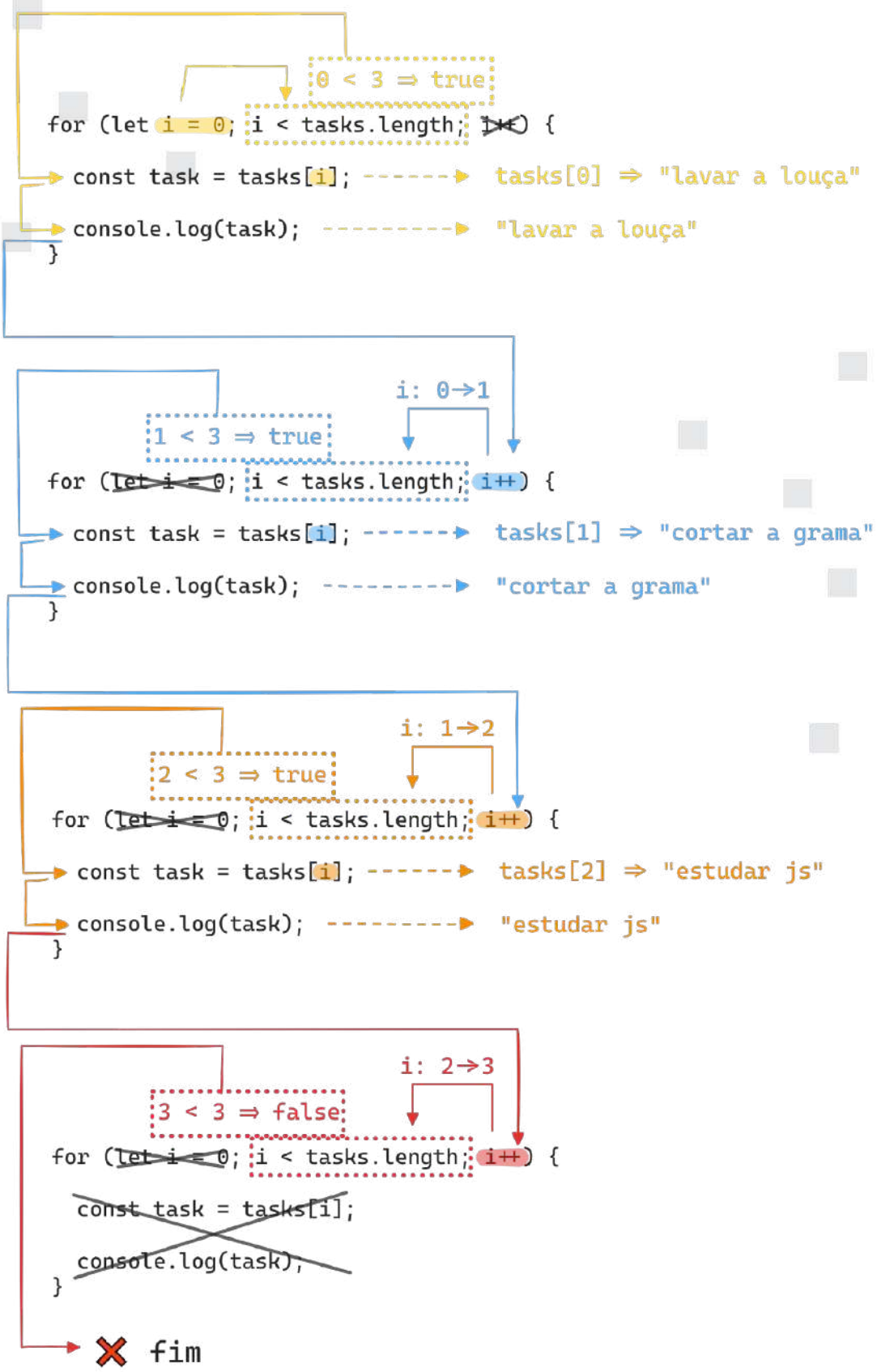
E como o índice do último elemento de um array é `array.length - 1`, a condição do `for` será `i < array.length`.

Então fica assim (poderia ser outros comandos no lugar do `console.log`, é só para ilustrar):

```
const tasks = ["lavar a louça", "cortar a grama", "estudar js"];

for (let i = 0; i < tasks.length; i++) {
  const task = tasks[i];
  console.log(task);
}
```

Veja como fica o diagrama de execução:

Comando for clássico



Comando for clássico

Note que ele passa por todos os índices do array (nesse caso 0, 1 e 2) e aborta quando chega no 3 (tamanho do array).

Se o array fosse maior ou menor, também daria certo. Por exemplo com tamanho 10: passaria pelos índices 0, 1, 2, ... 9. E abortaria ao chegar no 10.

O `for` clássico pode ser mais "difícil" do que o `for of` porque tem mais detalhes para entender, mas para percorrer arrays é até *melhor* que o `for of`.

Porque no `for of` só sabemos *qual é o elemento*, e no `for` clássico sabemos *qual é o índice* (variável `i`) e *qual é o elemento* (variável `task`).

No final desta aula mostraremos um exemplo (lista de tarefas) onde saber o índice será fundamental.

For clássico para buscar um elemento do array

Há situações em que temos em mãos um elemento do array, mas não sabemos o índice dele.

Por exemplo temos o array `const tasks = ["lavar a louça", "cortar a grama", "estudar js", "dormir"]` , e queremos saber qual o índice do elemento `"estudar js"`.

Isso pode ser resolvido usando um `for` clássico:

```
const tasks = ["lavar a louça", "cortar a grama", "estudar js"];

// elemento cujo índice queremos descobrir
const target = "estudar js";

// variável que terá a resposta no final (o índice)
let index;

for (let i = 0; i < tasks.length; i++) {
  const task = tasks[i];

  if (task === target) {
    index = i;
  }
}

console.log("O índice é " + index);
```

Funciona assim:

- Na primeira iteração, `i` vale `0` e `task` vale `"lavar a louça"`.

  Portanto a condição do `if` é `false`.

- Na segunda iteração, `i` vale `1` e `task` vale `"cortar a grama"`.

  Portanto a condição do `if` é `false`.

- Na terceira iteração, `i` vale `2` e `task` vale `"estudar js"`.

  Dessa vez a condição do `if` é `true`, então a variável `index` recebe o valor `2` (que é o índice procurado).

- Na quarta iteração, `i` vale `3` e `task` vale `"dormir"`.

  Portanto a condição do `if` é `false`.

- E acaba o loop.

Comando for clássico

Feito todo o loop, somente 1 iteração executou o comando `if`, atribuindo o valor de `2` para a variável `index`, que é a resposta correta !

Por isso quando o loop acaba, a variável `index` contém a resposta que buscávamos (o índice do elemento `"estudar js"`).

No final desta aula mostraremos um exemplo (lista de tarefas) onde precisaremos usar esse algoritmo de busca.

Loop infinito

Ao usar o `for` clássico, é preciso tomar cuidado para não fazer um loop infinito.

Loop infinito é uma repetição que nunca acaba: ela faz infinitas iterações.

Um exemplo:

```
for (let i = 0; i >= 0; i++) {
  console.log(i);
}
```

Simule mentalmente a execução desse `for` e perceberá que ele vai fazer iterações sem parar nunca.

- Primeira iteração, `i` vale 0, a condição (`0 >= 0`) é `true`.
- Segunda iteração, `i` vale 1, a condição (`1 >= 0`) é `true`.

E assim por diante, o `i` vai aumentar, então em todas as iterações ele será um número positivo, por isso a condição `i >= 0` será `true` em toda iteração.

Como o loop só é abortado quando chega uma iteração onde a condição dá `false`, este loop nunca vai parar.

Claramente isso é um bug, não é intencional fazer um loop infinito.

Geralmente acontece quando você se confunde ao escrever o código.

É um Erro Lógico.

Quando isso acontecer com você, o sintoma é fácil de perceber: a página web ficará travada, porque o navegador está "ocupado" executando o loop infinito para sempre.

Isso significa que nenhuma interação com a página vai funcionar: nenhum evento, nem clicar em nada.

Você terá que fechar a página e abrir de novo (depois de corrigir o código).

Se você não tem ideia de onde ou como está acontecendo um loop infinito no seu código, sugerimos usar o debugger.

Comandos while e do-while

Introdução

Os comandos `for of` e `for` clássico costumeiramente são usados nas seguintes situações:

- Você tem um array e quer iterar por ele: `for of` ou `for` clássico.
- Você tem um intervalo de números e quer iterar por ele: `for` clássico

(O que é 90% dos casos)

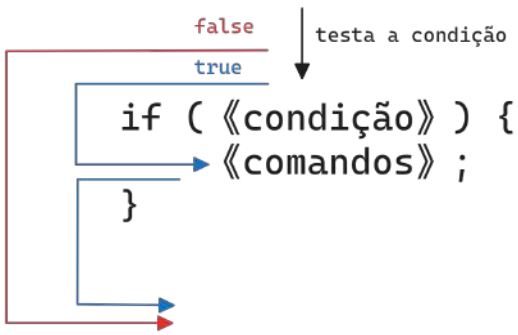
Para qualquer outro cenário de repetição, é comum usar um dos dois outros comandos de repetição do javascript: o `while` ou o `do-while`.

Comando `while`

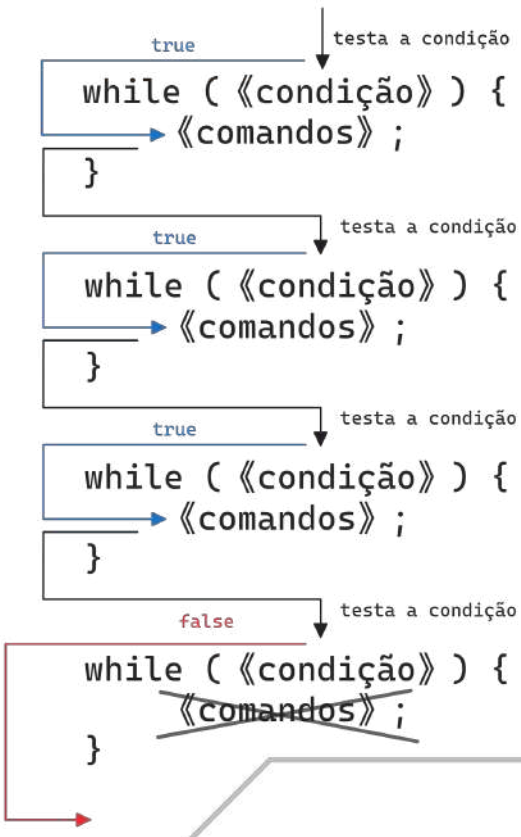
O `while` (tradução: "enquanto") é uma versão simplificada do `for` clássico que só tem a condição.

Ele é como um `if` repetido.

No caso do comando `if`, ele testa a condição e (se deu `true`) executa os comandos, depois segue adiante:



Já no `while`, ele testa a condição, executa os comandos, depois testa a condição de novo, executa os comandos de novo, e assim por diante:



Comandos while e do-while

Ele só interrompe quando, ao testar a condição, o resultado dá `false` pela primeira vez. Aí ele não faz essa iteração, e aborta o loop.

Exemplo de uso do `while`

Suponha que você está fazendo um programa que pede para o usuário digitar um número, com um `prompt()`.

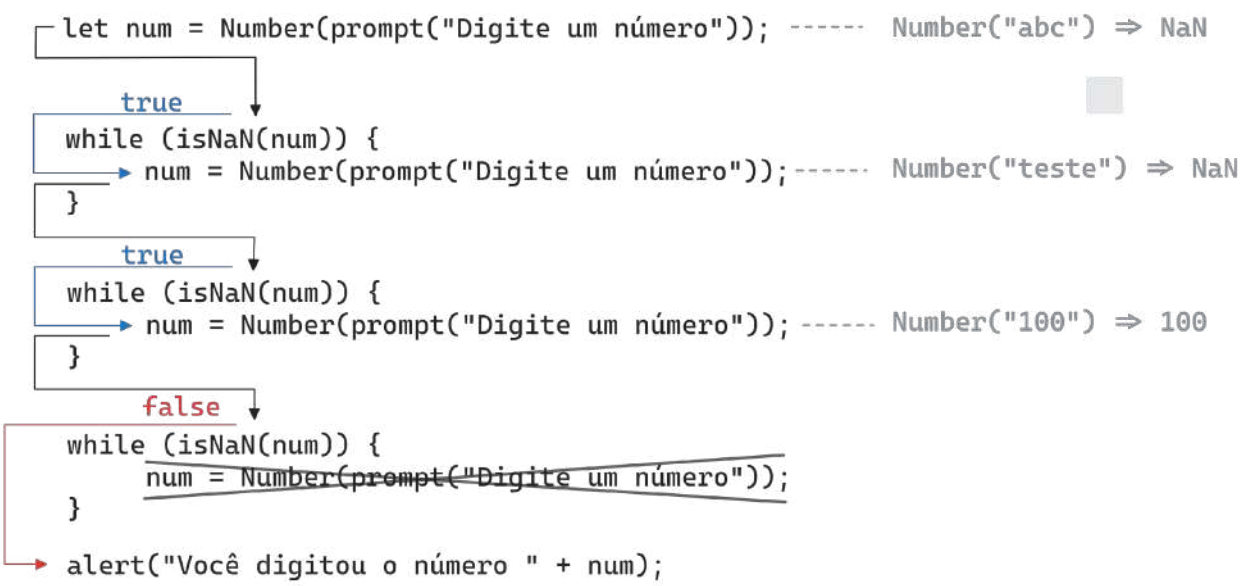
O usuário pode digitar algo que não é um número, e você não quer permitir isso.

Se for digitado algo que não é um número, você quer repetir o prompt. E continuar repetindo até que finalmente digite um número corretamente.

Podemos fazer isso com `while`:

```
let num = Number(prompt("Digite um número"));

while (isNaN(num)) {
  num = Number(prompt("Digite um número"));
}

alert("Você digitou o número " + num);
```

Imaginando que o usuário tentou digitar "abc" na primeira vez, depois "teste" na segunda vez, e finalmente digitou "100" na terceira vez, o diagrama da execução fica assim:



Note que declaramos a variável `num` *fora* das chaves do loop, porque dentro das chaves é um escopo próprio (igual aos comandos `for`).

Comandos while e do-while

Comando do-while

O `do-while` (tradução: "faça enquanto") é parecido com o `while`, mas ele executa os comandos primeiro, e depois testa se deve executar de novo ou abortar o loop.

Podemos refazer o exemplo anterior usando `do-while`:

```
let num;

do {
  num = Number(prompt("Digite um número"));
} while (isNaN(num));
```

Note que precisa do ponto-e-vírgula após o `while(isNaN(num))`.

Aplicação de loops e arrays: lista de tarefas

Introdução

Na aula passada, fizemos uma lista de tarefas, onde o usuário podia adicionar uma nova tarefa, e todas ficavam guardadas no Local Storage e eram recuperadas ao reabrir a página.

O grande segredo para ser capaz de salvar e recuperar as tarefas de maneira organizada foi tê-las todas num array global do javascript.

Para relembrar, o código HTML da página era:

```
<!-- body da página -->
<body>
  Nova tarefa: <input id="task-input" />
  <button id="add-button">Adicionar</button>

  <ul id="tasklist"></ul>
  <script src="index.js"></script>
</body>
```

Aplicação de loops e arrays: lista de tarefas

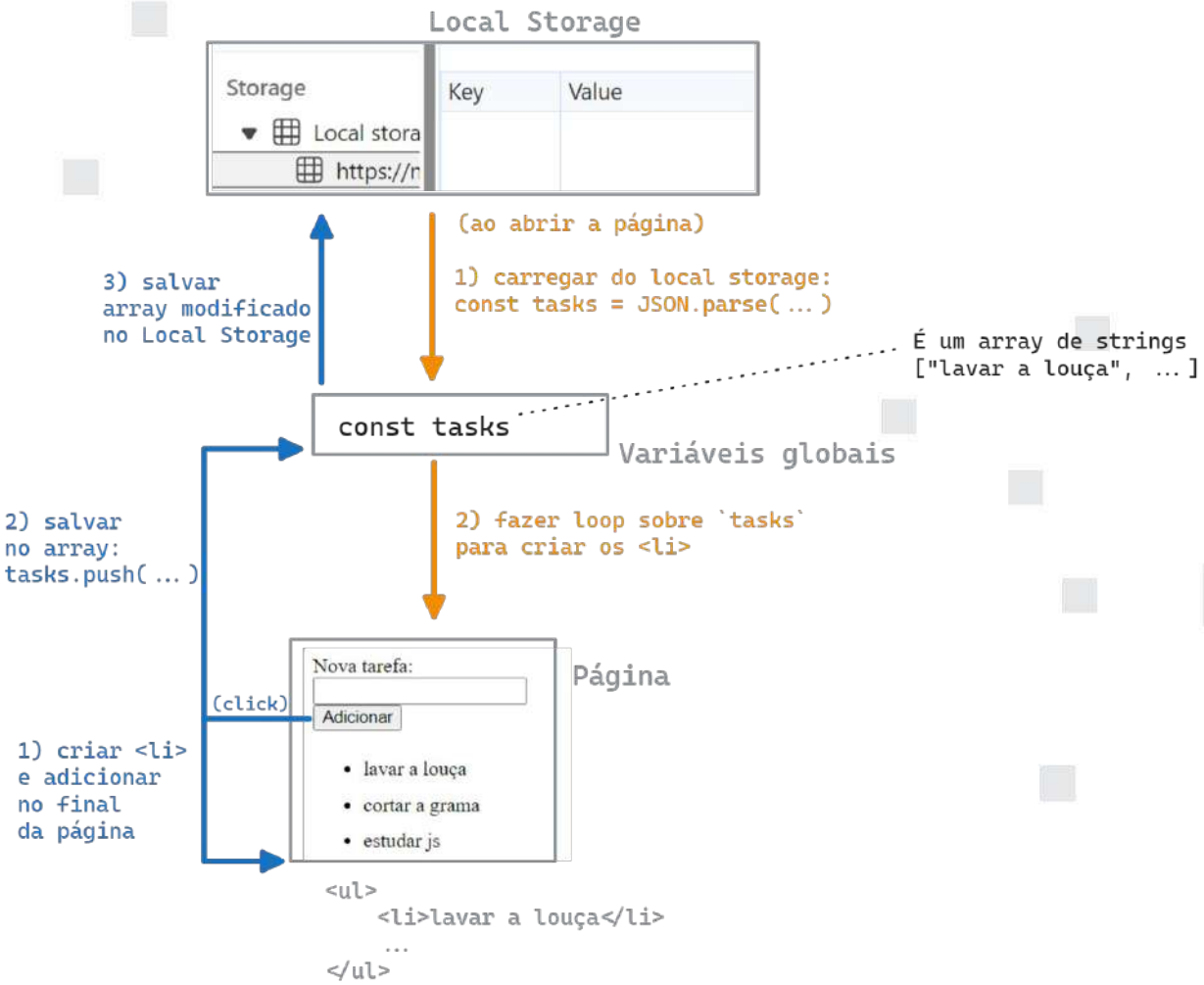
E o javascript era:

```
// index.js

const taskInput = document.getElementById("task-input"); // <input>
const addButton = document.getElementById("add-button"); // <button>
const tasklist = document.getElementById("tasklist"); // <ul>

addButton.addEventListener("click", addTask);

// carregar tarefas do Local Storage para o array
// global `tasks`, ou inicializar com array vazio []
// caso o LocalStorage não tenha dados.
// É array de strings: ["lavar a louça", ...]
const tasksLocalStorage = localStorage.getItem("tasks");
const tasks = tasksLocalStorage === null ? [] : JSON.parse(tasksLocalStorage);

// para cada tarefa carregada, adicioná-la no HTML.
// Se não tem nenhuma tarefa (i.e. array vazio), este
// `for` não executa iteração nenhuma (corretamente)
for (const task of tasks) {
  const newLi = document.createElement("li");
  newLi.innerText = task;

  tasklist.appendChild(newLi);
}

// botão que cria nova tarefa
function addTask() {
  const taskText = taskInput.value;

  const newLi = document.createElement("li");
  newLi.innerText = taskText;

  // colocar a tarefa no HTML (<li>)
  tasklist.appendChild(newLi);

  // colocar a tarefa no array global (string)
  tasks.push(taskText);

  // o array foi modificado (tarefa adicionada), portanto
  // devemos salvar no LocalStorage para não perder
  localStorage.setItem("tasks", JSON.stringify(tasks));
}
```

Aplicação de loops e arrays: lista de tarefas

Como o código fica cheio de detalhes e pode ser difícil acompanhar todos os detalhes e ao mesmo tempo se lembrar de como as coisas funcionam no seu todo, vale a pena pensar na página como um sistema.

O diagrama abaixo tenta enxergar esse sistema, explicação em seguida:



O diagrama enxerga o sistema por dois pontos de vista:

1. Depósitos de informação: estão em cor cinza, são os lugares onde informação fica guardada.
   São o Local Storage, as variáveis globais e a página (HTML) em si.

2. Módulos de código: indicados pelas setas coloridas, são o código global e os ouvintes de evento da página (só tem um).
   Em laranja, o código global de inicialização.
   Em azul, o código do ouvinte de evento do botão "Adicionar".
   Estamos denominando de "módulos" porque cada código tem um papel específico e é executado num momento específico.
   Quando o código de um dos módulos é executado, as setas indicam o fluxo de informação: de onde o módulo lê informações e para onde a informação é direcionada/guardada.

Observe de novo o diagrama e veja como ele corresponde fielmente ao código da página.

Aplicação de loops e arrays: lista de tarefas

Tarefas com prioridade (array de objetos e dataset)

Vamos implementar uma nova funcionalidade.

Cada tarefa terá uma prioridade, de duas prioridades possíveis: normal e alta.

Ao adicionar uma tarefa, ela tem prioridade normal.

Ao clicar em uma tarefa existente, ela muda para prioridade alta, o que é indicado visualmente com um plano de fundo amarelo no texto da tarefa.

Se clicar numa tarefa de alta prioridade, ela volta para prioridade normal.

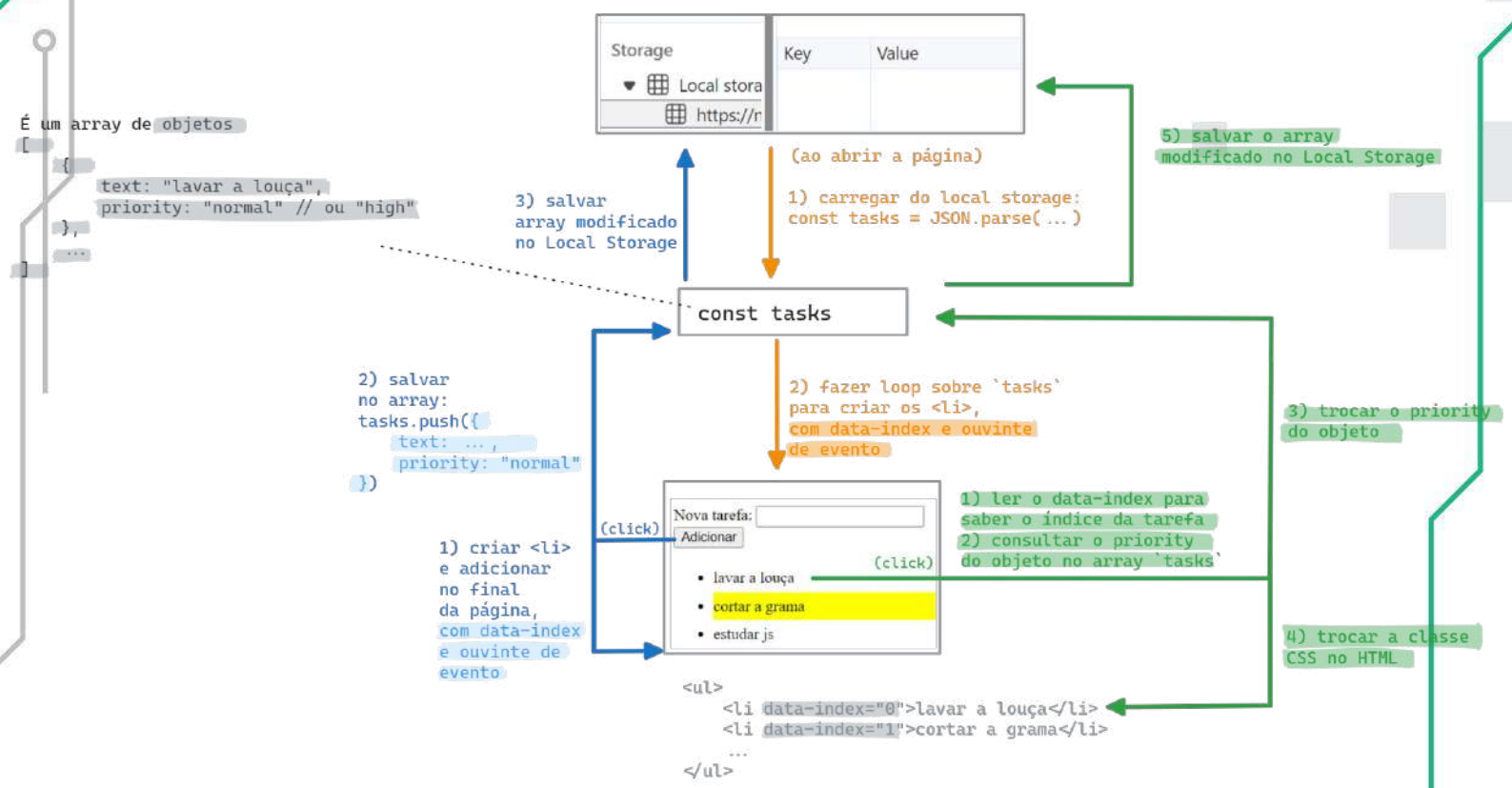
Ao recarregar a página, as prioridades devem ser preservadas (tarefas de alta prioridade devem voltar com fundo amarelo).

Para a estilização, podemos usar uma classe CSS "priority-high" para tarefas de alta prioridade, com a seguinte regra:

```css
.priority-high {
  background-color: yellow;
}
```

Para implementar isso no sistema, vamos pensar primeiro no diagrama de sistema, depois transformamos em código.

O diagrama vai ficar assim (mudanças grifadas, explicação em seguida):



Aplicação de loops e arrays: lista de tarefas

As mudanças:

- Em preto: o array `tasks` agora é um array de objetos porque precisamos salvar a prioridade da tarefa.
- Em laranja e em azul: o código global de inicialização e o ouvinte de evento do botão "Adicionar", ao criarem as `<li>` no HTML, precisam colocar nelas um ouvinte de evento de clique e uma propriedade customizada `data-index` para que, quando a tarefa for clicada, termos condições de saber *qual o índice da tarefa*.
- Em azul: ao clicar no botão Adicionar, ele tem que criar um objeto para a nova tarefa, com prioridade normal, e colocar o objeto no final do array.
- Em verde: novo ouvinte de evento de clique na tarefa.

  Esse código sabe qual o índice da tarefa por meio do `this.dataset.index` (lembrando que `this` é o objeto `<li>` que foi clicado).

  Ele então verifica no array se a tarefa com esse índice tem prioridade normal ou alta.

  E então faz a troca da prioridade no array (e salva no Local Storage) e estiliza o CSS com a classe "priority-high" se necessário.

Agora podemos transformar isso em código (comentários indicam as mudanças):

```
const taskInput = document.getElementById("task-input");
const addButton = document.getElementById("add-button");
const tasklist = document.getElementById("tasklist");

addButton.addEventListener("click", addTask);

const tasksLocalStorage = localStorage.getItem("tasks");
const tasks = tasksLocalStorage === null ? [] : JSON.parse(tasksLocalStorage);

// mudamos para `for` clássico porque precisamos
// saber o índice
for (let i = 0; i < tasks.length; i++) {
  const task = tasks[i];
  const newLi = document.createElement("li");

  // cuidado: task é um objeto {text: ..., priority: ...}
  newLi.innerText = task.text;

  // aplicar a estilização da prioridade
  if (task.priority === "high") {
    newLi.classList.add("priority-high");
  }

  // adicionar data-index informando o índice
  newLi.dataset.index = i;

  // adicionar ouvinte de evento
  newLi.addEventListener("click", changeTaskPriority);

  tasklist.appendChild(newLi);
}
```

(código continua na próxima página)

Aplicação de loops e arrays: lista de tarefas

(continuação do código da página anterior)

```
function addTask() {
  const taskText = taskInput.value;

  const newLi = document.createElement("li");
  newLi.innerText = taskText;

  // adicionar data-index ao <li> informando
  // o índice da nova tarefa.
  // O índice da nova tarefa é `tasks.length`, veja só:
  // se a lista tem 3 tarefas e esta será a 4ª,
  // o índice será 3 (tamanho atual do array).
  // Isso também dá certo se há outra quantidade
  // de tarefas (3, 4, 5, ...)
  newLi.dataset.index = tasks.length;

  // adicionar ouvinte de evento
  newLi.addEventListener("click", changeTaskPriority);

  tasklist.appendChild(newLi);

  // adicionar um objeto no final do array
  tasks.push({
    text: taskText,
    priority: "normal",
  });

  localStorage.setItem("tasks", JSON.stringify(tasks));
}

// ouvinte de evento novo: quando a tarefa é clicada
function changeTaskPriority() {
  // this é o objeto `<li>`
  const index = Number(this.dataset.index);

  // calcular qual a nova prioridade
  const newPriority = tasks[index].priority === "normal" ? "high" : "normal";

  // alterar a prioridade do objeto dentro do array
  tasks[index].priority = newPriority;

  // salvar porque o array foi alterado
  localStorage.setItem("tasks", JSON.stringify(tasks));

  // atualizar o estilo no CSS
  if (newPriority === "high") {
    this.classList.add("priority-high");
  } else {
    this.classList.remove("priority-high");
  }
}
```

Aplicação de loops e arrays: lista de tarefas

Botão de deletar, parte 1: e <button>

Vamos adicionar mais uma funcionalidade:

Cada tarefa terá, ao lado do texto, um botão "Deletar", que deverá fazer a tarefa sumir da página e do array.

Para começar, o `<li>` precisará de uma `<span>` para o texto e um `<button>`.

Então antes de qualquer outra coisa, vamos implementar somente isso (o `<button>` não vai fazer nada por enquanto).

O código fica assim (comentários indicam as mudanças):

```
const taskInput = document.getElementById("task-input");
const addButton = document.getElementById("add-button");
const tasklist = document.getElementById("tasklist");

addButton.addEventListener("click", addTask);

const tasksLocalStorage = localStorage.getItem("tasks");
const tasks = tasksLocalStorage === null ? [] : JSON.parse(tasksLocalStorage);

for (let i = 0; i < tasks.length; i++) {
  const task = tasks[i];
  const newLi = document.createElement("li");

  // <span> a ser colocada dentro da <li>
  const span = document.createElement("span");

  // o texto da tarefa é colocado dentro da <span>
  span.innerText = task.text;

  // aplicar a estilização da prioridade na <span>
  if (task.priority === "high") {
    span.classList.add("priority-high");
  }

  // adicionar ouvinte de evento na <span>
  span.addEventListener("click", changeTaskPriority);

  // pendurar a <span> dentro da <li>
  newLi.appendChild(span);

  // criar o botão e pendurar dentro da <li>
  // (botão não tem ouvinte de evento por enquanto)
  const button = document.createElement("button");
  button.innerText = "Deletar";

  newLi.appendChild(button);

  newLi.dataset.index = i;

  tasklist.appendChild(newLi);
}
```

(código continua na próxima página)

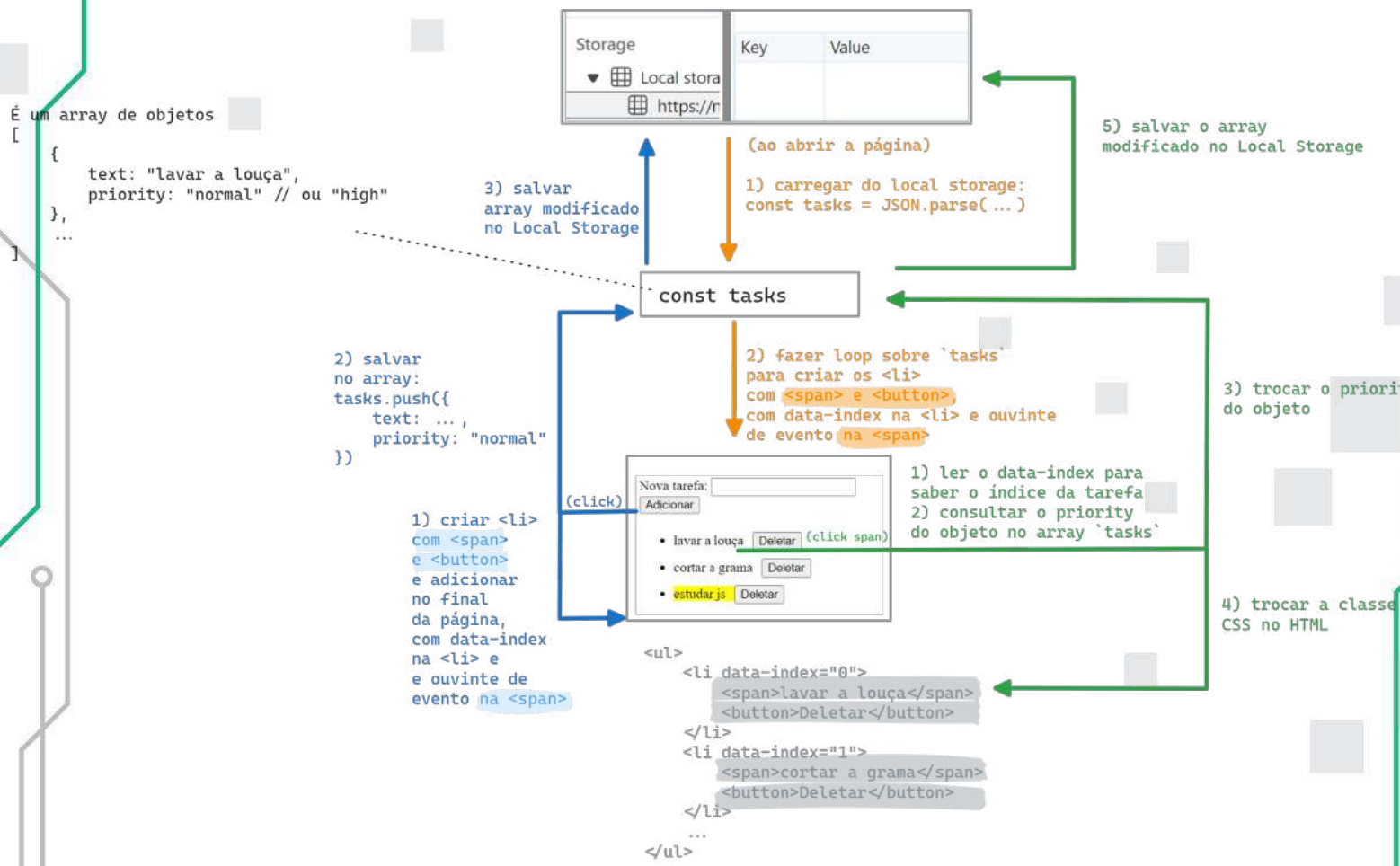
Aplicação de loops e arrays: lista de tarefas

```
function addTask() {
  const taskText = taskInput.value;

  const newLi = document.createElement("li");

  // <span> a ser colocada dentro da <li>
  const span = document.createElement("span");

  // o texto da tarefa é colocado dentro da <span>
  span.innerText = taskText;

  // adicionar ouvinte de evento na <span>
  span.addEventListener("click", changeTaskPriority);

  // pendurar a <span> dentro da <li>
  newLi.appendChild(span);

  // criar o botão e pendurar dentro da <li>
  // (botão não tem ouvinte de evento por enquanto)
  const button = document.createElement("button");
  button.innerText = "Deletar";

  newLi.appendChild(button);

  newLi.dataset.index = tasks.length;

  tasks.push({
    text: taskText,
    priority: "normal",
  });

  localStorage.setItem("tasks", JSON.stringify(tasks));
}

function changeTaskPriority() {
  // this é o objeto <span>.
  // A <span> fica dentro do <li data-index="...">.
  // Então this.parentElement é o objeto <li>.
  // E nesse objeto tem o dataset.
  const index = Number(this.parentElement.dataset.index);

  const newPriority = tasks[index].priority === "normal" ? "high" : "normal";

  tasks[index].priority = newPriority;

  localStorage.setItem("tasks", JSON.stringify(tasks));

  if (newPriority === "high") {
    this.classList.add("priority-high");
  } else {
    this.classList.remove("priority-high");
  }
}
```

Aplicação de loops e arrays: lista de tarefas

Você pode testar que esse código continua funcionando (as prioridades continuam funcionando) e o botão está "inofensivo" (existe mas não faz nada).

Em termos do diagrama de sistema, tudo que mudou foi a criação de `<span>` e `<button>` (mudanças grifadas):



Aplicação de loops e arrays: lista de tarefas

Botão de deletar, parte 2: problema dos índices

Como próximo passo, se você está pensando que basta colocar um ouvinte de evento no botão da tarefa, e esse ouvinte vai deletar a tarefa do array (usando o método `splice()`) e do HTML (usando o método `remove()`), está enganado(a).

Para ver por que não é só isso, imagine que temos algumas tarefas:

```
<ul>
  <!-- span e button omitidos por simplicidade -->
  <li data-index="0">A</li>
  <li data-index="1">B</li>
  <li data-index="2">C</li>
  <li data-index="3">D</li>
  <li data-index="4">E</li>
</ul>
```

Portanto o array `tasks` está assim (simplificadamente ! Cada letra é um objeto): `[A, B, C, D , E]`.

Se você deletar a tarefa B simplesmente removendo-a do HTML e do array, o HTML ficará assim:

```
<ul>
  <li data-index="0">A</li>
  <li data-index="2">C</li>
  <li data-index="3">D</li>
  <li data-index="4">E</li>
</ul>
```

E o array assim: `[A, C, D, E]`.

Percebeu o problema ? Os `data-index` dos `<li>` *posteriores ao que foi removido* ficaram incorretos.

Por exemplo, ao clicar na span da tarefa D no HTML, o `data-index` vale 3 mas o índice dessa tarefa no array agora é 2.

Essa discordância aconteceu porque (como você sabe) ao deletar um elemento do array, os elementos posteriores se "deslocam" para a esquerda, diminuindo seus índices em 1 unidade:

- D tinha índice 3, agora é 2.
- E tinha índice 4, agora é 3.

Mas os `data-index` obviamente não se atualizam sozinhos só porque um dos `<li>` foi removido.

Nós *poderíamos* consertar esse problema fazendo com que o ouvinte de evento do botão "Deletar" também conserte os `data-index` dos `<li>` posteriores ao deletado.

Esse ouvinte teria que deletar a tarefa (do array e do HTML) e corrigir os `data-index` dos `<li>` seguintes.

É uma solução que funciona, mas não é a prática mais comum.

Pelo menos identificamos que existe esse problema, e a causa raiz dele é o fato de que, ao deletar um elemento do array, *os índices dos elementos seguintes se alteram*.

Vamos ver a maneira mais comum de resolver isso.

Aplicação de loops e arrays: lista de tarefas

Botão de deletar, parte 3: IDs únicos com timestamp

O *índice* dos elementos no array não é necessariamente fixo, pois pode mudar quando algum outro elemento é deletado.

Isso nos deu problemas, queremos algo que seja fixo, que não mude mesmo ao deletar elementos do array.

Então vamos inventar alguma outra coisa que seja fixa.

Essa "outra coisa" é uma solução clássica no desenvolvimento web: **IDs únicos**.

Funciona assim:

Sempre que uma tarefa for criada, vamos criar para ela um ID (identificador) único, um número que nunca vai se repetir para nenhuma outra tarefa.

Uma maneira fácil é o código `new Date().getTime()`, que (como você sabe) calcula o timestamp atual em milissegundos.

Dentro da função `addTask` (ouvinte de evento do botão de Adicionar Tarefa), vamos calcular o timestamp atual e usar isso como um ID para a tarefa, que ficará guardado como uma propriedade da tarefa:

```
// nova estrutura do objeto tarefa:
{
  text: "lavar a louça",
  priority: "normal",
  id: new Date().getTime() // um número calculado na hora, por exemplo 1706135204780
}
```

A cada vez que uma tarefa for adicionada, o timestamp calculado será diferente, e assim os IDs nunca se repetem, cada tarefa terá um número diferente.

Ao inserir o `<li>` de uma tarefa no HTML, não vamos mais usar `<li data-index="...">` porque o índice no array não é fixo (problema dos índices).

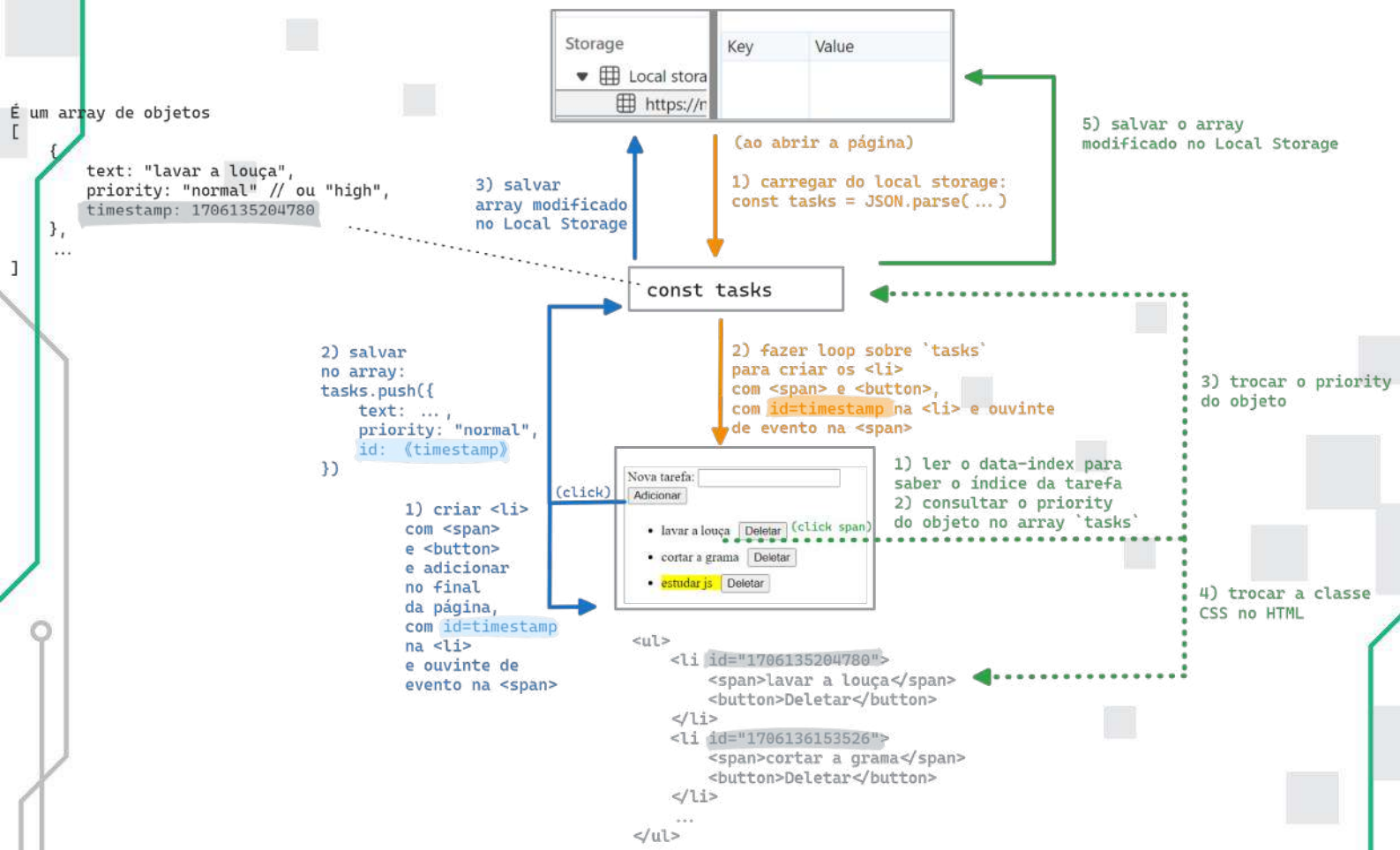
Em vez disso, podemos usar o atributo `id` (o atributo oficial do HTML), colocando nele o timestamp da tarefa, por exemplo `<li id="1706135204780">`.

Assim cada `<li>` terá um `id` que é o timestamp da tarefa.

Por enquanto vamos parar por aqui e implementar somente essa mudança (a função `changeTaskPriority` vai ficar quebrada porque não tem mais `data-index`).

Aplicação de loops e arrays: lista de tarefas

Do ponto de vista do diagrama de sistema, vai ficar assim (mudanças grifadas, ouvinte de evento da span está tracejado para indicar que está quebrado porque não existe mais data-index):



Aplicação de loops e arrays: lista de tarefas

E o código fica assim (comentários indicam as mudanças):

```
const taskInput = document.getElementById("task-input");
const addButton = document.getElementById("add-button");
const tasklist = document.getElementById("tasklist");

addButton.addEventListener("click", addTask);

const tasksLocalStorage = localStorage.getItem("tasks");
const tasks = tasksLocalStorage === null ? [] : JSON.parse(tasksLocalStorage);

for (let i = 0; i < tasks.length; i++) {
  const task = tasks[i];
  const newLi = document.createElement("li");

  const span = document.createElement("span");

  span.innerText = task.text;

  if (task.priority === "high") {
    span.classList.add("priority-high");
  }

  span.addEventListener("click", changeTaskPriority);

  newLi.appendChild(span);

  const button = document.createElement("button");
  button.innerText = "Deletar";

  newLi.appendChild(button);

  // colocar a propriedade `id` da tarefa como
  // valor do atributo <li id="...">.
  // Não confunda as duas coisas ! Um é o `id`
  // que é propridade do objeto, outro é o
  // atributo `id` do HTML
  newLi.id = task.id;

  tasklist.appendChild(newLi);
}
```

(código continua na próxima página)

Aplicação de loops e arrays: lista de tarefas

(continuação do código da página anterior)

```
function addTask() {
  const taskText = taskInput.value;

  const newLi = document.createElement("li");

  const span = document.createElement("span");

  span.innerText = taskText;

  span.addEventListener("click", changeTaskPriority);

  newLi.appendChild(span);

  const button = document.createElement("button");
  button.innerText = "Deletar";

  newLi.appendChild(button);

  // criar um id para a tarefa (timestamp do instante atual)
  const taskId = new Date().getTime();

  // colocar o timestamp acima como
  // valor do atributo <li id="...">.
  newLi.id = taskId;

  tasks.push({
    text: taskText,
    priority: "normal",
    // guardar no objeto o timestamp gerado
    id: taskId,
  });

  localStorage.setItem("tasks", JSON.stringify(tasks));
}

function changeTaskPriority() {
  const index = Number(this.parentElement.dataset.index);

  const newPriority = tasks[index].priority === "normal" ? "high" : "normal";

  tasks[index].priority = newPriority;

  localStorage.setItem("tasks", JSON.stringify(tasks));

  if (newPriority === "high") {
    this.classList.add("priority-high");
  } else {
    this.classList.remove("priority-high");
  }
}
```

Aplicação de loops e arrays: lista de tarefas

Botão de deletar, parte 4: consertando o ouvinte de evento da span

Como dito, a última mudança (implementação do timestamp no lugar do `data-index`) quebrou o código do ouvinte de evento da span.

Porque ele dependia do `data-index` para saber *qual o índice* da tarefa que foi clicada (releia o código/diagrama para lembrar porque ele precisava do índice da tarefa).

Atualmente, o atributo `<li id="1706136153526">` está no lugar do `<li data-index="0">`, ou seja, nós não sabemos mais o *índice* da tarefa no array, mas sabemos o `id` da tarefa.

Você já viu isso antes ? Nós temos uma informação sobre a tarefa (id), mas precisamos encontrar o *índice* da tarefa no array...

Precisamos fazer uma busca ! (se não se lembra, releia a seção *"For clássico para buscar um elemento do array"* desta aula)

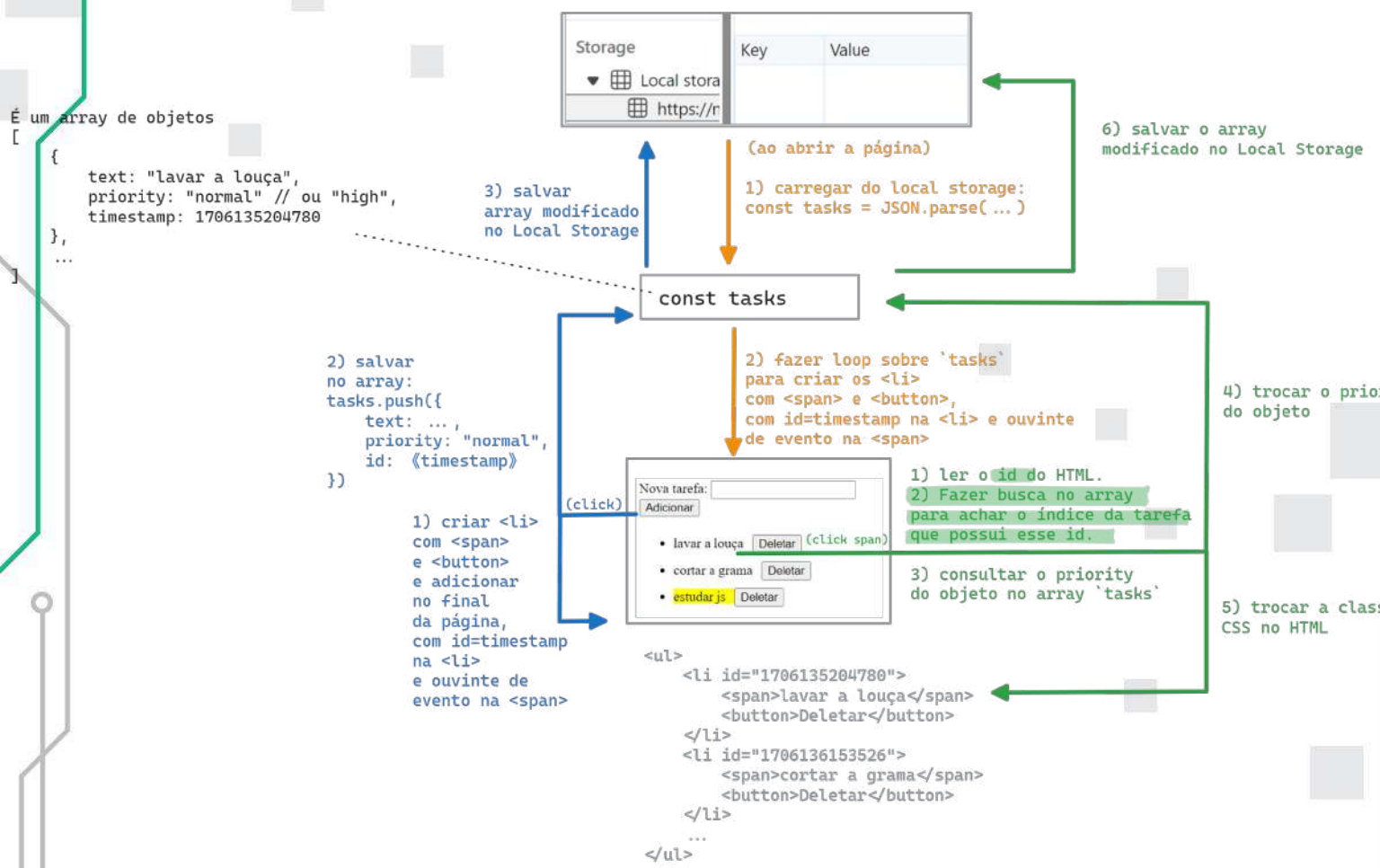
Como só precisamos mudar o código do ouvinte de evento da span, segue somente essa parte do código corrigida (comentários indicam as mudanças):

```
function changeTaskPriority() {
  // this é o <span>.
  // this.parentElement é o "pai" do <span>, que é
  // o <li id="《timestamp》">
  const taskId = Number(this.parentElement.id);

  // precisamos do índice da tarefa no array.
  // Só temos o id da tarefa (taskId extraído do HTML).
  // Vamos fazer uma busca no array para encontrar
  // a tarefa com id igual ao `taskId` do HTML.
  let index;

  // buscar no array uma tarefa cujo `task.id`
  // seja igual ao `taskId` do HTML.
  // Quando encontrar, salvar o índice (i)
  // na variável `index`
  for (let i = 0; i < tasks.length; i++) {
    const task = tasks[i];

    if (task.id === taskId) {
      index = i;
    }
  }

  const newPriority = tasks[index].priority === "normal" ? "high" : "normal";

  tasks[index].priority = newPriority;

  localStorage.setItem("tasks", JSON.stringify(tasks));

  if (newPriority === "high") {
    this.classList.add("priority-high");
  } else {
    this.classList.remove("priority-high");
  }
}
```

Aplicação de loops e arrays: lista de tarefas

Pode testar ! Combine esse código com o da seção anterior (implementação dos IDs) e verá que tudo está funcionando de novo, incluindo a marcação de prioridade !

O diagrama está assim (mudanças grifadas):

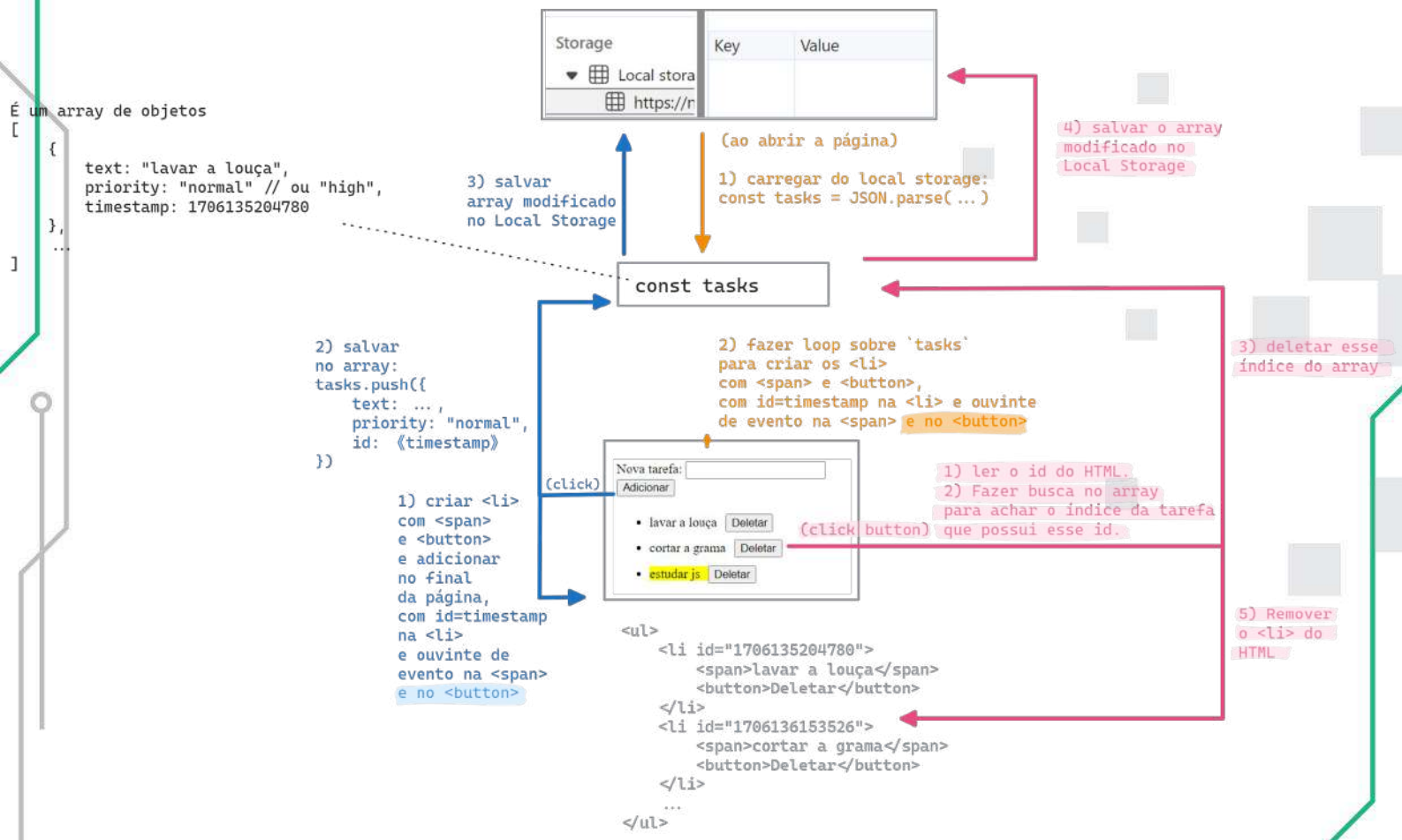


Aplicação de loops e arrays: lista de tarefas

Botão de deletar, parte 5: implementando o ouvinte de evento do botão Deletar

Finalmente, depois de toda essa volta para implementar IDs únicos (que obrigou a fazer uma busca no array quando precisamos achar o índice no array a partir do ID da tarefa), podemos agora implementar o botão de deletar com facilidade.

O diagrama fica assim (o novo código em rosa é o ouvinte de evento do botão Deletar, e omitimos o ouvinte verde do span para não poluir a figura e porque não muda nada nele):



Aplicação de loops e arrays: lista de tarefas

O código final completo fica assim (comentários indicam as mudanças):

```
const taskInput = document.getElementById("task-input");
const addButton = document.getElementById("add-button");
const tasklist = document.getElementById("tasklist");

addButton.addEventListener("click", addTask);

const tasksLocalStorage = localStorage.getItem("tasks");
const tasks = tasksLocalStorage === null ? [] : JSON.parse(tasksLocalStorage);

for (let i = 0; i < tasks.length; i++) {
  const task = tasks[i];
  const newLi = document.createElement("li");

  const span = document.createElement("span");

  span.innerText = task.text;

  if (task.priority === "high") {
    span.classList.add("priority-high");
  }

  span.addEventListener("click", changeTaskPriority);

  newLi.appendChild(span);

  const button = document.createElement("button");
  button.innerText = "Deletar";

  // colocar o ouvinte de clique no botão
  button.addEventListener("click", deleteTask);

  newLi.appendChild(button);

  newLi.id = task.id;

  tasklist.appendChild(newLi);
}
```

(código continua na próxima página)

Aplicação de loops e arrays: lista de tarefas

(continuação do código da página anterior)

```
function addTask() {
  const taskText = taskInput.value;

  const newLi = document.createElement("li");

  const span = document.createElement("span");

  span.innerText = taskText;

  span.addEventListener("click", changeTaskPriority);

  newLi.appendChild(span);

  const button = document.createElement("button");
  button.innerText = "Deletar";

  // colocar o ouvinte de clique no botão
  button.addEventListener("click", deleteTask);

  newLi.appendChild(button);

  const taskId = new Date().getTime();

  newLi.id = taskId;

  tasks.push({
    text: taskText,
    priority: "normal",
    id: taskId,
  });

  localStorage.setItem("tasks", JSON.stringify(tasks));
}
```

(código continua na próxima página)

Aplicação de loops e arrays: lista de tarefas

(continuação do código da página anterior)

```
function changeTaskPriority() {
  const taskId = Number(this.parentElement.id);

  let index;

  for (let i = 0; i < tasks.length; i++) {
    const task = tasks[i];

    if (task.id === taskId) {
      index = i;
    }
  }

  const newPriority = tasks[index].priority === "normal" ? "high" : "normal";

  tasks[index].priority = newPriority;

  localStorage.setItem("tasks", JSON.stringify(tasks));

  if (newPriority === "high") {
    this.classList.add("priority-high");
  } else {
    this.classList.remove("priority-high");
  }
}

// novo ouvinte de evento
function deleteTask() {
  // ler o ID do html
  const taskId = Number(this.parentElement.id);

  // fazer uma busca no array, procurando qual o
  // índice da tarefa (objeto) que possui o ID
  // igual ao ID lido do HTML
  let index;

  for (let i = 0; i < tasks.length; i++) {
    const task = tasks[i];

    if (task.id === taskId) {
      index = i;
    }
  }

  // remover esse índice do array
  tasks.splice(index, 1);

  // salvar o array modificado no Local Storage
  localStorage.setItem("tasks", JSON.stringify(tasks));

  // remover o <li> do HTML.
  // Lembrando que `this` é o <button> que foi clicado,
  // então this.parentElement é o pai do <button>, que
  // é o <li>.
  this.parentElement.remove();
}
```