

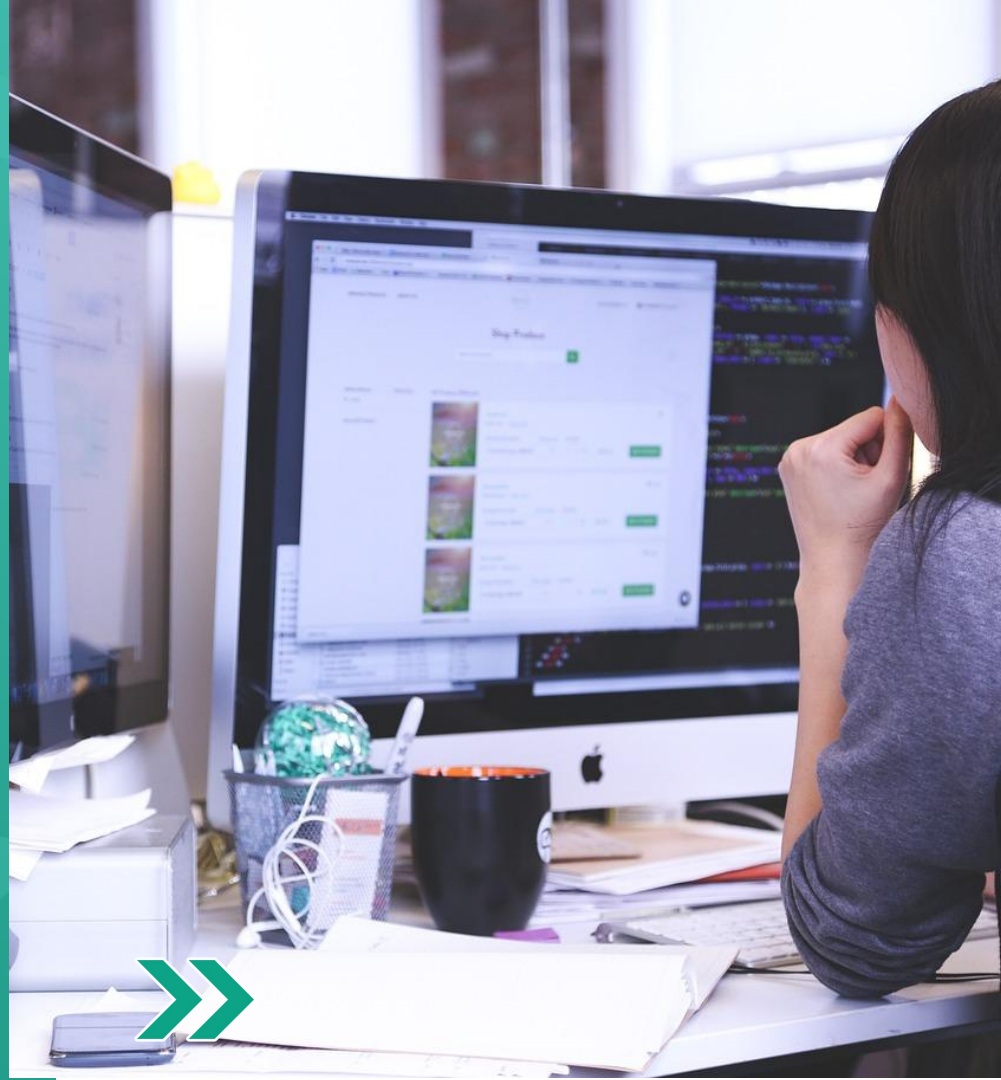


escola  
britânica de  
artes criativas  
& tecnologia

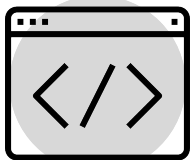
# Profissão: Engenheiro Front-End



# BOAS PRÁTICAS



# Recursos do ES6+



Confira boas práticas da comunidade de Front-End por assunto relacionado às aulas.

- **Conheça o Babel**
- **Métodos de arrays**
- **Conheça a arrow function**
- **Operadores de Spread e Rest**
- **Conheça as estruturas Map e Set**
- **Programação Assíncrona**
- **Orientações a Objetos com ES6**



# Conheça o Babel



## Babel

O Babel também oferece suporte a plugins e presets que permitem personalizar e estender suas funcionalidades. Os plugins permitem adicionar recursos e transformações específicas ao código, enquanto os presets são conjuntos pré-configurados de plugins que podem ser usados para transpilar para uma determinada versão do ECMAScript.



# Conheça o Babel



## ● Preset

O Babel oferece vários presets prontos para uso, como:

- **@babel/preset-env:** Este é o preset mais comumente usado. Ele permite que você especifique um conjunto de ambientes de destino (como navegadores específicos ou versões do Node.js) e, em seguida, o Babel escolhe os plugins necessários com base nesses ambientes de destino. Isso permite que você escreva código moderno do ECMAScript e o transpile para uma versão compatível com os ambientes especificados.
- **@babel/preset-react:** Este preset é projetado para transpilar código JSX usado no desenvolvimento de aplicativos React. Ele inclui os plugins necessários para transformar o JSX em JavaScript válido.
- **@babel/preset-typescript:** Este preset é usado para transpilar código TypeScript para JavaScript. Ele inclui os plugins necessários para lidar com as diferenças de sintaxe entre TypeScript e JavaScript.

# Métodos de arrays

Acompanhe algumas dicas para usar os métodos arrays:



- **Considere a utilização de métodos encadeados:** Os métodos de arrays podem ser encadeados para realizar várias transformações em um único fluxo de código. Isso pode ajudar a melhorar a legibilidade e a manutenibilidade do seu código.
- **Esteja ciente do impacto no desempenho:** Embora os métodos de arrays sejam úteis e expressivos, é importante ter em mente que eles podem ter um impacto no desempenho, especialmente ao lidar com arrays muito grandes. Em alguns casos, loops tradicionais podem ser mais eficientes. Avalie o contexto e o tamanho do array em questão para tomar a decisão adequada.
- **Mantenha-se atualizado com as novidades do ECMAScript:** O JavaScript está em constante evolução, com novas versões do ECMAScript trazendo recursos e melhorias. Fique atualizado com as novidades do ECMAScript para aproveitar as funcionalidades mais recentes e utilizar o Babel para transpilá-las em ambientes que ainda não oferecem suporte nativo.

# Conheça a arrow function

## Arrow function

Acompanhe algumas dicas que podem ser úteis ao usar a arrow function:



- Verifique a compatibilidade com o ambiente de destino:** Antes de usar arrow functions, verifique se o ambiente de destino do seu código é compatível com as versões do ECMAScript (ES) que as suportam. Arrow functions foram introduzidas no ES6 (ES2015) e são amplamente suportadas em navegadores e ambientes modernos. No entanto, se você estiver transpilando seu código para versões mais antigas do ECMAScript, como o ES5, o Babel garantirá que as arrow functions sejam transpiladas para uma sintaxe equivalente suportada pelos ambientes de destino.
- Use arrow functions quando adequado:** As arrow functions podem ser úteis quando você deseja uma sintaxe mais concisa para funções, especialmente em casos de funções curtas e expressivas. No entanto, lembre-se de que elas não são adequadas para todos os cenários. Em particular, as arrow functions não possuem seus próprios valores de this e não são adequadas para serem usadas como métodos de objeto. Nestes casos, é preferível o uso de funções tradicionais.

# Conheça a arrow function

## Arrow function

Acompanhe algumas dicas que podem ser úteis ao usar a arrow function:



- **Mantenha a legibilidade do código:** Embora as arrow functions possam tornar o código mais conciso, é importante manter a legibilidade. Evite aninhar arrow functions excessivamente ou escrever expressões muito longas em uma única linha. Separe o corpo da função em várias linhas, se necessário, para melhorar a clareza do código.
- **Teste o código transpilado:** Certifique-se de testar o código transpilado pelo Babel, especialmente em diferentes navegadores e ambientes de destino. Isso ajudará a garantir que as arrow functions e outras funcionalidades transpiladas estejam funcionando corretamente em todos os ambientes desejados.



# Conheça a arrow function

## This

Quando se trata do contexto this em arrow functions, aqui estão algumas dicas importantes a serem consideradas:



- Herança do contexto léxico:** Diferentemente das funções tradicionais, as arrow functions herdam o valor de this do contexto léxico em que estão definidas. Isso significa que o valor de this dentro de uma arrow function será o mesmo valor de this fora dela.
- Evite usar arrow functions como métodos de objeto:** Devido à herança do contexto léxico, as arrow functions não são adequadas para serem usadas como métodos de objeto, onde geralmente se espera que this se refira ao próprio objeto. Em vez disso, use funções tradicionais para esses casos.



# Conheça a arrow function

## This

Quando se trata do contexto `this` em arrow functions, aqui estão algumas dicas importantes a serem consideradas:

- 
**Use arrow functions em callbacks:** Arrow functions são particularmente úteis em situações de callbacks, como em métodos de array (`map`, `filter`, `forEach`) ou ao lidar com eventos em bibliotecas ou frameworks. Nesses casos, o contexto do `this` é frequentemente capturado corretamente pela arrow function, evitando a necessidade de usar `.bind()` ou criar uma variável para armazenar o contexto.
- 
**Se necessário, use `.bind()` ou crie uma variável para o contexto:** Em situações em que você precisa especificar explicitamente um contexto diferente para a arrow function, pode-se usar o método `.bind()` ou criar uma variável para armazenar o contexto desejado.



# Operadores de Spread e Rest

## Arguments

Arguments não é um array real, mas possui algumas características de um array, como a propriedade "length". No entanto, ele não possui métodos de array como "forEach", "map", entre outros. Se você precisar usar métodos de array, pode convertê-lo em um array real usando técnicas como o spread operator ou o método "Array.from()".

Embora a variável "arguments" seja útil em certos cenários, o uso excessivo dela pode tornar o código menos legível e dificultar a manutenção. Em muitos casos, é preferível usar parâmetros de função explícitos para tornar o código mais claro e previsível.



# Operadores de Spread e Rest

## Rest

Acompanhe algumas dicas para usar o operador rest:



- **Coleta de argumentos variáveis:** O operador rest é especialmente útil quando você deseja criar funções que aceitem um número variável de argumentos. Ele permite que você colete todos os argumentos passados para a função em um único parâmetro como um array. Isso é útil quando você não sabe quantos argumentos serão fornecidos.

**Acesso aos argumentos individuais:** Com o operador rest, você pode acessar cada argumento individualmente usando índices ou métodos de array. Isso permite que você trabalhe com os argumentos de maneira flexível e fácil.

**Combinação com parâmetros regulares:** O operador rest pode ser combinado com parâmetros regulares em uma função. Os parâmetros regulares são definidos antes do parâmetro rest e podem ser usados normalmente.

**Uso em desestruturação:** O operador rest pode ser usado em conjunto com a desestruturação de arrays ou objetos para extrair valores específicos ou restantes.

# Operadores de Spread e Rest

## Spread

Acompanhe algumas dicas para usar o operador spread:

- **Criação de cópias de arrays e objetos:** O operador spread é ótimo para criar cópias superficiais (shallow copies) de arrays e objetos, permitindo que você trabalhe com uma nova referência sem modificar o original.
- **Combinação de arrays:** O operador spread permite combinar vários arrays em um único array.
- **Mesclagem de objetos:** O operador spread também pode ser usado para mesclar múltiplos objetos em um único objeto.
- **Passagem de argumentos para funções:** O operador spread pode ser usado para passar argumentos individuais para uma função que espera múltiplos argumentos.
- **Criação de arrays ou objetos dinamicamente:** O operador spread permite adicionar elementos extras a um array ou propriedades adicionais a um objeto de forma dinâmica.



# Operadores de Spread e Rest

## Desestruturação

Acompanhe algumas dicas sobre o uso dessa funcionalidade:



- **Atribuição de valores padrão:** Ao desestruturar um objeto ou array, você pode atribuir valores padrão a variáveis caso as propriedades ou elementos não estejam presentes. Isso ajuda a evitar erros quando os dados são incompletos ou ausentes.
- **Alias para propriedades:** Você pode usar a desestruturação para atribuir valores a variáveis com nomes diferentes dos das propriedades originais, criando aliases (apelidos).
- **Desestruturação aninhada:** Você pode desestruturar objetos ou arrays aninhados, permitindo acessar propriedades ou elementos internos com facilidade.
- **Ignorar valores:** Você pode usar a sintaxe de vírgula para ignorar valores que não são necessários na desestruturação. Isso é útil quando você deseja extrair apenas alguns elementos de um array ou propriedades específicas de um objeto.
- **Parâmetros de função:** A desestruturação pode ser usada para extrair valores de objetos diretamente nos parâmetros de uma função, facilitando a passagem de valores.

# Conheça as estruturas Map e Set

## Map

Acompanhe algumas dicas para o uso do conjunto map:



- **Criação de um Map:** Para criar um novo Map vazio, você pode usar a sintaxe `new Map()`
- **Adição de valores:** Você pode adicionar valores ao Map usando o método `set(chave, valor)`
- **Recuperação de valores:** Para recuperar o valor associado a uma chave específica, use o método `get(chave)`
- **Verificação de existência de uma chave:** Para verificar se uma chave existe no Map, use o método `has(chave)`
- **Remoção de um par chave-valor:** Para remover um par chave-valor do Map, use o método `delete(chave)`.
- **Iteração sobre um Map:** O Map oferece várias formas de iterar sobre seus pares chave-valor, como os métodos `forEach`, `keys`, `values` e `entries`.
- **Tamanho do Map:** Você pode obter o número de pares chave-valor em um Map usando a propriedade `size`.

# Conheça as estruturas Map e Set

## Set

Acompanhe algumas dicas para o uso do conjunto set:



- **Criação de um Set:** Para criar um novo Set vazio, você pode usar a sintaxe `new Set()`
- **Adição de valores:** Você pode adicionar valores ao Set usando o método `add(valor)`
- **Verificação de existência de um valor:** Para verificar se um valor existe no Set, use o método `has(valor)`
- **Remoção de um valor:** Para remover um valor do Set, use o método `delete(valor)`
- **Iteração sobre um Set:** O Set oferece várias formas de iterar sobre seus valores, como o método `forEach` e o uso de `for...of`.
- **Tamanho do Set:** Você pode obter o número de valores únicos em um Set usando a propriedade `size`.
- **Conversão do Set em um Array:** Se precisar converter um Set em um Array, você pode usar o operador `spread ...` ou o método `Array.from()`.



# Programação Assíncrona

## Async

Acompanhe algumas dicas para  
usar a função `async`:



- **Utilize o `async` em torno da função:** Ao definir uma função assíncrona, coloque a palavra-chave `async` antes da declaração da função. Isso indica que a função será assíncrona e retornará uma Promise.
- **Use o `await` dentro da função assíncrona:** O `await` é usado dentro de uma função assíncrona para pausar a execução e esperar a resolução de uma Promise. Certifique-se de usar o `await` apenas dentro de funções assíncronas. Isso permite escrever código assíncrono de forma síncrona, evitando callbacks e mantendo a legibilidade do código.
- **Trate erros com `try...catch`:** Ao usar `async` e `await`, é recomendado envolver o código com um bloco `try...catch`. Isso permite capturar erros lançados por Promises rejeitadas dentro da função assíncrona. Utilize o bloco `try` para envolver o código que contém o `await` e o bloco `catch` para tratar os erros capturados.

# Programação Assíncrona

## Async

Acompanhe algumas dicas para usar a função `async`:



- Lide com múltiplas Promises:** Se você estiver trabalhando com várias Promises dentro de uma função assíncrona, pode usar o `await` em cada uma delas individualmente para aguardar a resolução. Alternativamente, você pode usar `Promise.all()` para aguardar a resolução de várias Promises ao mesmo tempo.
- Evite o uso excessivo de `async` em cadeias de chamadas:** Embora o `async` possa ser útil em certos pontos do código, tenha cuidado ao usá-lo em excesso em cadeias de chamadas de funções. Isso pode levar a um desempenho inferior, pois cada função assíncrona terá uma camada adicional de Promise envolvendo a execução.

# Programação Assíncrona

## Async

Acompanhe algumas dicas para  
usar a função `async`:

- **Teste o código assíncrono:** Ao escrever código assíncrono com `async` e `await`, é importante testar cuidadosamente os cenários assíncronos. Certifique-se de testar diferentes caminhos de execução, como Promises resolvidas, rejeitadas e tempos limite (timeout), para garantir que seu código esteja lidando corretamente com essas situações.
- **Esteja atento à compatibilidade:** Lembre-se de verificar a compatibilidade do `async` e `await` com as versões do JavaScript que você está utilizando.



# Programação Assíncrona

## Promise

Acompanhe algumas dicas para usar Promises de forma eficaz:



- Encadeie Promises com then():** Aproveite o encadeamento de métodos `then()` para executar ações sequenciais com base no resultado de Promises anteriores. Isso permite criar um fluxo lógico mais limpo e legível.
- Utilize catch() para capturar erros:** Utilize o método `catch()` para capturar erros nas Promises encadeadas. Isso permite tratar falhas e erros de forma centralizada e evitar que o código pare de executar em caso de exceções.
- Evite o "callback hell":** As Promises foram introduzidas como uma alternativa ao "callback hell" (aninhamento excessivo de callbacks). Utilize Promises para lidar com operações assíncronas em cascata de forma mais limpa e estruturada.

# Programação Assíncrona

## Promise

Acompanhe algumas dicas para usar Promises de forma eficaz:

- **Crie funções utilitárias reutilizáveis:** Se você estiver trabalhando com operações assíncronas repetitivas, considere criar funções utilitárias para encapsular a lógica da Promise. Isso promove a reutilização de código e mantém o código mais limpo e legível.
- **Conheça os métodos auxiliares:** Além do `then()` e `catch()`, familiarize-se com outros métodos auxiliares de Promises, como `finally()`, `Promise.resolve()` e `Promise.reject()`. Esses métodos podem ser úteis em diferentes situações e podem aprimorar seu código assíncrono.



# Programação Assíncrona

## • Await

O uso do `await` está diretamente relacionado ao conceito de Promises e funções assíncronas em JavaScript. Ele permite que você trabalhe com código assíncrono de forma síncrona, tornando-o mais legível e fácil de entender. No entanto, é importante utilizá-lo adequadamente e compreender as implicações do uso de funções assíncronas no fluxo de execução do código. Acompanhe algumas dicas para usá-lo de forma eficaz:

- Use o `await` somente dentro de funções assíncronas: O `await` só pode ser usado dentro de funções declaradas com a palavra-chave `async`. Certifique-se de que a função em que você está usando o `await` seja marcada como `async`.



# Programação Assíncrona

**Use o `await` antes de uma expressão `Promise`:** O `await` é usado antes de uma expressão `Promise` para pausar a execução da função assíncrona até que a `Promise` seja resolvida. Isso permite que você aguarde o resultado de uma operação assíncrona antes de prosseguir.

**Armazene o resultado do `await` em uma variável:** Ao utilizar o `await`, você pode atribuir o resultado da `Promise` a uma variável. Isso permite que você trabalhe com o resultado retornado pela `Promise` no restante da função.

**Lide com erros utilizando `try...catch`:** Ao utilizar o `await`, é uma boa prática envolver o bloco de código que contém o `await` em um bloco `try...catch`. Isso permite que você capture erros caso a `Promise` seja rejeitada e trate-os de maneira adequada.



# Programação Assíncrona



**Não bloqueie o código desnecessariamente:** Lembre-se de que o `await` pausa a execução do código assíncrono até que a `Promise` seja resolvida. Certifique-se de utilizar o `await` apenas quando for necessário aguardar a conclusão de uma operação assíncrona. Caso contrário, você pode bloquear desnecessariamente a execução do código.

**Use o `await` em sequência:** O `await` pode ser usado em sequência para aguardar a resolução de várias `Promises` em ordem. Isso permite que você execute tarefas assíncronas em uma determinada ordem e trabalhe com os resultados conforme eles estão disponíveis.

**Esteja ciente do tempo limite (`timeout`):** Ao usar o `await`, especialmente em chamadas de API, considere definir um tempo limite para evitar que o código fique bloqueado indefinidamente caso a `Promise` não seja resolvida em um período razoável de tempo. Você pode fazer isso usando `Promise.race()` com uma `Promise` de tempo limite.



# Orientações a Objetos com ES6

## Extends

Essa palavra reservada é uma parte essencial da herança em JavaScript, permitindo a criação de hierarquias de classes e a reutilização de código. Acompanhe algumas dicas para usá-la de forma eficaz:



**Planeje a hierarquia de classes:** Ao usar o `extends`, é necessário planejar adequadamente a hierarquia de classes. Considere como as classes estão relacionadas entre si e se a relação de herança é apropriada para o problema que você está resolvendo.

**Defina uma classe pai adequada:** Escolha uma classe adequada para servir como a classe pai na hierarquia de classes. A classe pai deve conter as características e comportamentos comuns que podem ser compartilhados pelas classes filhas.

**Aproveite os métodos e propriedades herdados:** Ao usar o `extends`, você pode acessar os métodos e propriedades da classe pai por meio da instância da classe filha usando a palavra-chave `super`. Isso permite que você reutilize e estenda o comportamento da classe pai.

# Orientações a Objetos com ES6

## Extends

Essa palavra reservada é uma parte essencial da herança em JavaScript, permitindo a criação de hierarquias de classes e a reutilização de código. Acompanhe algumas dicas para usá-la de forma eficaz:



### **Adicione métodos e propriedades específicos da classe filha:**

Aproveite a capacidade de adicionar métodos e propriedades específicos da classe filha para personalizar seu comportamento. Isso permite que você estenda e especialize a funcionalidade herdada da classe pai.

**Evite heranças profundas e complexas:** Tenha cuidado ao criar hierarquias de classes profundas e complexas. Muitos níveis de herança podem levar a um código mais difícil de entender e manter. Considere utilizar outros princípios de design, como composição, quando a herança se tornar muito complexa.

# Orientações a Objetos com ES6

## Super

O `super` é usado para garantir que a funcionalidade da classe pai seja reutilizada ou estendida na classe filha, fornecendo uma maneira de manter uma relação de especialização entre as classes. Acompanhe algumas dicas para usá-lo de forma eficaz:



**Chame o construtor da classe pai:** Ao usar `super()` no construtor da classe filha, lembre-se de chamar o construtor da classe pai. Isso é importante para garantir que as propriedades da classe pai sejam inicializadas corretamente antes de adicionar as propriedades da classe filha.

**Use `super` para acessar métodos da classe pai:** A palavra-chave `super` permite acessar os métodos da classe pai a partir da classe filha. Use `super.nomeDoMetodo()` para chamar o método da classe pai. Isso é útil quando você quer estender o comportamento da classe pai na classe filha.

**Aproveite `super` para acessar propriedades da classe pai:** Você pode usar `super` para acessar as propriedades da classe pai na classe filha. Isso permite que você acesse ou sobrescreva propriedades específicas da classe pai dentro da classe filha.

# Orientações a Objetos com ES6

## Super

O `super` é usado para garantir que a funcionalidade da classe pai seja reutilizada ou estendida na classe filha, fornecendo uma maneira de manter uma relação de especialização entre as classes. Acompanhe algumas dicas para usá-lo de forma eficaz:



**Utilize `super` no contexto apropriado:** Lembre-se de usar `super` apenas dentro da classe filha, pois ele não terá significado fora do contexto da herança. Tenha cuidado para não usar `super` onde não é necessário ou onde não faz sentido.

**Aplique o `super` em sobrescrições de métodos:** Ao sobrescrever um método da classe pai na classe filha, você pode usar `super` para chamar o método original da classe pai. Isso permite que você estenda o comportamento da classe pai sem perder a funcionalidade original.

# Orientações a Objetos com ES6

## Super

O `super` é usado para garantir que a funcionalidade da classe pai seja reutilizada ou estendida na classe filha, fornecendo uma maneira de manter uma relação de especialização entre as classes. Acompanhe algumas dicas para usá-lo de forma eficaz:



**Utilize `super` com cautela:** Embora `super` seja uma ferramenta poderosa para acessar membros da classe pai, é importante usá-lo com cautela. Considere se a herança é a abordagem correta para o problema que você está resolvendo e avalie se a relação de herança é apropriada para o caso em questão.

**Evite referências circulares:** Evite criar referências circulares entre classes pai e filha, onde a classe filha também é uma classe pai de outra classe. Isso pode levar a um código confuso e difícil de manter. Em vez disso, procure seguir uma hierarquia de herança clara e evite dependências complexas.

# Bons estudos!

