Boas práticas para legibilidade

Nesse texto serão abordadas dicas de boas práticas para tornar um código mais legível e de fácil entendimento. A clareza do código reduz a necessidade de comentários excessivos e torna o entendimento mais intuitivo.

> ⚠️ Um ponto importante é que se você está aprendendo programação agora e essas boas práticas pareçam barreiras a mais que estão sendo impostas para o seu aprendizado, então é justo que você ignore as dicas que achar pertinente, pois vale mais que estude e escreva códigos pouco legíveis com qualquer nome, código bilíngue e cheio de comentário, do que deixar de estudar programação. E futuramente, quando estiver mais maduro como programador, volte e aprenda as boas práticas.

▼ Nomenclatura de Variáveis

A escolha de nomes adequados para variáveis é essencial para a compreensão e manutenção do código. Um bom nome deve ser descritivo o suficiente para que seu significado seja claro sem a necessidade de consultas adicionais.

Evite nomes não descritivos e genéricos

1. Evitar nomes não descritivos

Exemplo ruim:

```
const users = ['Danylo', 'Emily', 'Pelé'];

const filtred = users.filter(u => {
```

```
    return u.startsWith('D');
});
```

Problemas:

- `filtred` não é descritivo. O nome não deixa claro quais dados ele armazena.
- `u` é um nome muito curto e não explicita seu significado.

Solução:

```
const users = ['Danylo', 'Emily', 'Pelé'];

const usersStartingWithLetterD = users.filter(user => {
    return user.startsWith('D');
});
```

Agora, a variável indica que está armazenando uma lista de usuários cujos nomes começam com "D".

---

2. Evitar nomes genéricos

Exemplo ruim:

```
function getUsers() {
    const data = getUsersFromDatabase();

    // lógica

    return data;
}
```

Problemas:

- `data` é genérico e não explica que tipo de dado está armazenando.

Solução:

```
function getUsers() {
   const users = getUsersFromDatabase();

   // lógica

   return users;
}
```

Agora fica claro que a variável armazena uma lista de usuários.

Exercício:

Desafio: Nomenclatura de Variáveis

Código do desafio

▼ Booleanos

Variáveis booleanas devem ser nomeadas como perguntas para melhorar a semântica do código.

Exemplo: Verificar se o usuário é maior de idade

Exemplo ruim:

```
const userOnMajority = true;

if (userOnMajority) {}
```

Problema:

O nome "userOnMajority" não parece uma pergunta e não é intuitivo.

semântica da condicional fica mais clara: "se o usuário maior de idade".

Solução:

```
const isUserOfLegalAge = true;
```

```
if (isUserOfLegalAge) {}
```

Agora a semântica da condicional fica mais clara: "se o usuário é maior de idade".

Exemplo: Verificar se o botão está desabilitado

Exemplo ruim:

```
const disabled = true;

if (disabled) {}
```

Problema:

O nome "disabled" é vago e pode se referir a qualquer coisa.

Solução:

```
const isButtonDisabled = true;

if (isButtonDisabled) {}
```

Agora a variável deixa claro que se refere ao estado do botão.

Exercício:

Desafio: Booleanos

Código do desafio

▼ Causa vs. Efeito

- **Efeito:** Refere-se ao estado final sem explicar a origem (Exemplo: `isFileProcessingComplete`).
- **Causa:** Explicita o motivo pelo qual o estado existe (Exemplo: `hasPendingTasks`).

Dar preferência a nomes baseados na causa pode tornar o código mais semântico e legível.

Exemplo: Processamento de arquivos

Nome refletindo o efeito

```
const isFileProcessingComplete = false; // efeito: processamento ainda não está completo

if (!isFileProcessingComplete) {
    console.log("Processando arquivo...");
}
```

Problema:

O nome apenas indica o estado, sem deixar claro o motivo pelo qual o arquivo ainda está sendo processado.

Nome refletindo a causa

```
const hasPendingTasks = true; // causa: ainda existem tarefas pendentes

if (hasPendingTasks) {
    console.log("Processando arquivo...");
}
```

Agora a condicional tem mais sentido: "se há tarefas pendentes, então processar arquivo".

Exemplo: Controle de Acesso em um Sistema

Nome refletindo o efeito

```
const isAccessDenied = True  // Efeito: acesso negado

if (isAccessDenied) {
```

```
    console.log("Acesso negado")
}
```

Problema:

O nome apenas indica o estado, sem deixar claro o motivo pelo qual o arquivo ainda está sendo processado.

Nome refletindo a causa

```
const hasPendingTasks = true; // causa: ainda existem tarefas pendente
s

if (hasPendingTasks) {
    console.log("Processando arquivo...");
}
```

Agora a condicional tem mais sentido: "se há tarefas pendentes, então processar arquivo".

Exercício:

Desafio: Causa vs. Efeito

Código do desafio

> https://github.com/rocketseat-education/ignite-clean-code-desafios/blob/main/desafios/03-causa-vs-efeito.tsx

---

▼ Código em inglês

inglês é o idioma internacional para código (e para documentações), as linguagens de programação tem suas palavras chaves em inglês e ao misturar palavras em português nesse código estamos criando um problema principalmente para pessoas que possuem alguma deficiência ou problema de visão que usam leitores de tela para poderem ler os códigos:

“Pronúncia Inconsistente: Leitores de tela geralmente usam um único idioma para interpretar o texto. Se uma variável está em inglês e o resto do código em português, a pronúncia pode ficar estranha ou incompreensível”

Devemos evitar misturar idiomas em nomes de variáveis, funções e comentários. Como:

- `ListProduto()`
- `const userEncontrado`

E utilizar apenas os termos em inglês:

- `ListProduct()`
- `const userFound`

Para auxiliar nisso podemos usar ferramentas de tradução ou então inteligências artificias como o chatGPT para tradução de termos.

Desafio:

Contexto:

Código:

> https://github.com/rocketseat-education/ignite-clean-code-desafios/blob/main/desafios/03-causa-vs-efeito.tsx

---

▼ Regras em condicionais

▼ Condicionais negativas:

As vezes é necessário unir algumas variáveis em estruturas condicionais como `if`

E essa união pode prejudicar a legibilidade do código caso seja necessário colocar negações

Exemplo - deseja saber se o aluno foi reprovado:

▼ Código ruim

```
const isGradeAboveSeven = True //tradução:  "A nota é maior que s
const isAttendanceAboveSeventyFive = True //tradução: "A presenç

if (!isGradeAboveSeven && !isAttendanceAboveSeventyFive){}
```

A leitura semântica dessa condição seria:

"Se a nota NÃO for maior que 7 E a presença NÃO for maior que 75%"

▼ Código bom

```
const isGradeBelowSeven = True //tradução:  "A nota é menor que s
const isAttendanceBelowSeventyFive = True //tradução: "A presenç

if (isGradeBelowSeven && isAttendanceBelowSeventyFive ){}
```

A leitura semântica dessa condição seria:

"Se a nota for menor que 7 e a presença menor que 75%"

▼ Early Return vs Else

Early Return e **Else** são duas abordagens para estruturar a lógica condicional em funções.

- **Early Return** → Na maioria das linguagens de programação, quando acontece um return dentro de uma função, o restante do código deixa de ser executado.
- **Else** → Agrupa a lógica dentro de um bloco condicional, o que pode gerar mais indentação.

▼ Usando **ELSE:**

```
function getStudentStatus(){
  const isGradeBelowSeven = true //tradução:  "A nota é menor que
  const isAttendanceBelowSeventyFive = true //tradução: "A presen
```

```
  if (isGradeBelowSeven && isAttendanceBelowSeventyFive ){
    return "Aluno reprovado"
  }else{
      return "Aluno aprovado"
  }
}
console.log(getStudentStatus())
```

Resultado: “Aluno reprovado”

O laço condicional não entraria no caso Else

▼ Usando **Early Return:**

```
function getStudentStatus(){
  const isGradeBelowSeven = true //tradução:  "A nota é menor que
  const isAttendanceBelowSeventyFive = true //tradução: "A presen

  if (isGradeBelowSeven && isAttendanceBelowSeventyFive ){
    return "Aluno reprovado"
  }

     return "Aluno aprovado"
}
console.log(getStudentStatus())
```

Resultado: “Aluno reprovado”

Como a condição do if é verdadeira, ao entrar nessa estrutura e receber um Return o restante da função não é executada, pois o computador volta imediatamente para a parte que instanciou a função.

▼ Condicionais aninhadas

Existe a possibilidade de serem feitos estrutura de condicionais dentro de outras, no entanto se possível evite aninhar as estruturas, prefira sempre fazer estruturas condicionais uma abaixo das outras para facilitar a compreensão do que está ocorrendo no código

Exemplo - se usuário pode acessar um sistema:

Função `canUserAccessSystem` recebe como parâmetro `role` (tipo de usuario) e `isActive` (Booleano se o usuário está ativo)

▼ Código ruim

```
function canUserAccessSystem(role, isActive) {
    if (isActive) { // Verifica se a conta está ativa
        if (role === "admin") {
            return "Acesso permitido: Administrador";
        } else if (role === "user") {
            return "Acesso permitido: Usuário padrão";
        } else {
            return "Acesso negado: Papel desconhecido";
        }
    } else {
        return "Acesso negado: Conta inativa";
    }
}
```

▼ Código bom

```
function canUserAccessSystem(role, isActive) {
    if (!isActive) {
        return "Acesso negado: Conta inativa";
    }

    if (role === "admin") {
      return "Acesso permitido: Administrador";
    }

    if (role === "user") {
      return "Acesso permitido: Usuário padrão";
    }

    return "Acesso negado: Papel desconhecido";
}
```

Desafio:

Contexto:

Código:

https://github.com/rocketseat-education/ignite-clean-code-desafios/blob/main/desafios/05-regras-em-condicionais.ts

Parâmetros e Desestruturação

Desestruturação

É comum na programação que dados sejam passados de uma função para outra

```
function createUserRoute(body){
  //validação

  createUserController(body)
}

function createUserController(data){
  usersRepository.create(data)
}

const usersRepository = {
  createUser(data){}
}
```

Mas em casos reais, essas funções costumam ficar em arquivos diferentes e se olharmos para algumas isoladamente é mais difícil entender o que é o dado sendo recebido e passado para outra função

No caso olhando apenas a função createUserController não é possível saber o que está contido dentro de data

```
function createUserController(data){
  usersRepository.create(data)
}
```

O Javascript tem uma solução para isso: a desestruturação, que significa separar os dados de dentro de um objeto recebido.

Código usando a reestruturação:

```
function createUserController(data){
   const {name, email, password} = data

   usersRepository.create({
      name,
      email,
      password,
   })
}
```

Agora é mais claro entender o que está sendo recebido nessa função e quais o dados estão sendo enviados para a criação do usuário.

▼ Receber Objetos invés de múltiplos parâmetros

Dentro do javascript também é preferível receber objetos ao invés de múltiplos parâmetros

Exemplo:

▼ Recebendo múltiplos parâmetros

Se na função `createUserRoute` além de receber o `body` fosse também recebido a variável `params`

```
function createUserRoute(body, params){
   //validação

   createUserController(body)
}
```

E em outro arquivo fosse sido criado as instâncias da função `createUserRoute`

Não é tão perceptível o que cada parte significa, principalmente qual o valor for null

```
createUserRoute({name, email, password}, {id: 3})
createUserRoute(null, {id: 3})
createUserRoute({name, email, password}, null)
```

▼ Recebendo objetos

Reescrevendo a função `createUserRoute` para receber objetos `body` e `params`

```
function createUserRoute({body, params}){
  //validação

  createUserController(body)
}
```

Criando as instancias da função em outro arquivo

```
createUserRoute({
  body: {name, email, password},
  params: {id: 3}
})

createUserRoute({
  body: null,
  params: {id: 3}
})

createUserRoute({
  params: null
})
```

▼ Devolver Objetos invés de variáveis

Assim como é melhor para a legibilidade e para manutenção receber parâmetros em forma de objetos, também é melhor retornar os dados em forma de objetos.

Olhando para a função `usersRepository` e imaginando que ela retorna um dado ao final de sua execução

Assim seria retornar apenas uma variável

```
const usersRepository = {
   createUser(data){
      const user = createUserOnDatabase()
      return user
   }
}
```

Porém, é preferível retornar um objeto e isso possibilitará até mesmo retornar mais itens sem quebrar o restante do código caso seja necessário no futuro.

Retornando em formato de objeto:

```
const usersRepository = {
   createUser(data){
      const user = createUserOnDatabase()
      return {
         user,
      }
   }
}
```

Retornando mais de um valor

```
const usersRepository = {
   createUser(data){
      const user = createUserOnDatabase()
      return {
         user,
         id,
      }
   }
}
```

Desafio:

Contexto:

Código:

https://github.com/rocketseat-education/ignite-clean-code-desafios/blob/main/desafios/06-parametros-e-desestruturacao.tsx

---

▼ Números mágicos

Números mágicos são números usados diretamente no código, sem o contexto sobre eles.

É necessário usar constantes ou variáveis que descrevam o que o valor significa e então utiliza-las no código

Vantagens de evitar números mágicos:

1. **Legibilidade**: Ao usar constantes ou variáveis com nomes explicativos, o código fica mais fácil de entender.
2. **Manutenibilidade**: Se for necessário alterar algum valor, fica mais fácil encontrar e modificar a constante sem ter que procurar por todos os números mágicos no código.
3. **Menor risco de erro**: A utilização de constantes evita a repetição de valores em diversos locais, o que diminui a chance de inconsistências.

▼ Exemplo desconto de preço:

No exemplo a função calcula descontos, mas não é possível entender o que seria o porque dos valores: 100 e 50 das estruturas condicionais

E nem os valores 0.2, 0.1 e 0.05 que são multiplicados pelo preço

```
function calculateDiscount(price) {

  if (price > 100) {
    return price* 0.2;
  }

  if (price> 50) {
```

```
    return price* 0.1;
  }

  return price* 0.05;
}
```

Uma solução é definir constantes que indiquem o que cada valor significa

```
const LIMITE_ALTO = 100;
const LIMITE_MEDIO = 50;
const DESCONTO_ALTO = 0.2;
const DESCONTO_MEDIO = 0.1;
const DESCONTO_BAIXO = 0.05;

function calculateDiscount(price) {

  if (price > LIMITE_ALTO) {
    return price* DESCONTO_ALTO;
  }

  if (price> LIMITE_MEDIO) {
    return price* DESCONTO_MEDIO;
  }

  return price* DESCONTO_BAIXO;
}
```

▼ Exemplo contador de tempo:

No javascript uma forma de contar um intervalo de tempo é o `setTimeout()` uma função que recebe uma função e um valor numérico em milisegundos

Passando valores numéricos diretamente é difícil ler e saber o que eles significam

```
  setTimeou(()⇒ {
  }, 1000 * 60 * 60 * 24 * 30)
```

Se for criado uma constante que indique o significado dos números melhora a legibilidade

```
const thirtyDays = 1000 * 60 * 60 * 24 * 30 // Tradução: trinta dias

setTimeou(()⇒ {
}, thirtyDays)
```

Desafio:

Contexto:

Código:

https://github.com/rocketseat-education/ignite-clean-code-desafios/blob/main/desafios/07-numeros-magicos.js

---

▼ Comentários vs Documentação

Tanto **comentários** quanto **documentação** são formas de descrever o código

A documentação é um conjunto mais formal e detalhado de explicações sobre o sistema, suas funcionalidades e como utilizá-lo.
Já os comentários devem ser usados como forma de avisos ou explicação de trechos complexos ou não óbvios do código, Justificação de uma abordagem específica.

Problemas dos comentários:

É um fato de que alguns programadores utilizam dos comentários para explicar seus código por que eles não tem uma boa legibilidade. Um código que segue boas práticas para ser mais legível dispensa a necessidade de comentários em excesso, deixando apenas os comentários que forem realmente necessários.

Exemplo de código com comentários em excessos:

```
// Esta função calcula a média das notas
function calc(t) {
  let s = 0; // Inicializa a soma das notas

  for (let i = 0; i < t.length; i++) {
    s += t[i]; // Soma todas as notas
  }

  return s / t.length; // Retorna a média das notas
}

let r = calc(3, [8, 9, 10]); // Calcula a média de 3 notas
console.log(r); // Exibe a média
```

Correção do código seguindo boas práticas:

traduções:

- gradeAverage = media de notas
- gradesList = lista de notas
- totalSumGrades = soma total das notas

```
function calculateGradeAverage({gradesList}) {
  let totalSumGrades = 0;

  for (let i = 0; i < gradesList.length; i++) {
    totalSumGrades += gradesList[i]; // Soma todas as notas
  }

  let gradeAverage = totalSumGrades / gradesList
  return {gradeAverage};
}

console.log(calculateGradeAverage({
  gradesList: [8, 9, 10],
}))
```

Desafio:

Contexto:

Código:

https://github.com/rocketseat-education/ignite-clean-code-desafios/blob/main/desafios/08-comentarios-vs-documentacao.js

---

▼ Syntatic Sugars

Syntactic sugars são recursos específicos de uma linguagem de programação e que não existem em outras linguagens.

Existem Syntactic sugars que trazem um bom benefício para o programador e para o código em geral, como por exemplo no Javascript tem a desestruturação, Arrow Functions, Template Literals.

Mas também existem casos que não representam um real ganho de produtividade e ainda prejudicam a legibilidade do código.

Exemplos:

```
const numberInString = "123456"
const numberConverted = +numberInString
const isNumberNotNull = !!numberConverted
```

Na linha 2 e a linha 3 acontecem transformações que são escritas de uma forma que é pouco intuitiva. Na verdade o que acontece é que na linha 2 a variável é convertida para um tipo numérico e na linha 3 é convertido para um tipo booleano.

Esse cenário poderia facilmente ser substituído por um recurso presente em todas as linguagens de programação: funções para conversão de tipos.

```
const numberInString = "123456"
const numberConverted = Number(numberInString)
const isNumberNotNull = Boolean(numberConverted)
```

Desafio:

Contexto:

Código:

https://github.com/rocketseat-education/ignite-clean-code-desafios/blob/main/desafios/09-syntatic-sugars.ts