CLEAN CODE
Simplificado

```
const data = []
const results = []
const json = []
const filter = []
const active = []
```

TE ENSINO EM
5 MINUTOS

PARE COM ISSO !!!

10 DICAS DE CLEAN CODE

```ts
export function getOrderMessage(status: string): string {
  if (status === 'pending') {
    return 'Order is pending.'
  } else if (status === 'shipped') {
    return 'Order has been shipped.'
  } else if (status === 'delivered') {
    return 'Order has been delivered.'
  }
}
```

❌

✅

```ts
const orderMessages = {
  [OrderStatus.Pending]: 'Order is pending.',
  [OrderStatus.Shipped]: 'Order has been shipped.',
  [OrderStatus.Delivered]: 'Order has been delivered.',
} as const

function getOrderMessage(status: OrderStatus): OrderMessage {
  return orderMessages[status]
}
```

1 - ELIMINE NÚMEROS E PALAVRAS MAGICAS

❌

```
function fullDayElapsed(seconds){
  return seconds >= 86400;
}
```

✅

```
function fullDayElapsed(seconds){
  const SECONDS_IN_A_DAY = 86_400;
  return seconds >= SECONDS_IN_A_DAY;
}
```

2 - SUBSTITUA PARÂMETROS BOOLEANOS POR ENUMS

❌

```
function sendNotification(isUrgent) {
  console.log(isUrgent ?
  "Sending URGENT notification!"
  : "Sending regular notification.");
}
sendNotification(true);
```

✅

```
enum NotificationPriority {
  URGENT = "Urgent",
  REGULAR = "Regular",
}
function sendNotification(priority) {
  console.log(`Sending ${priority} notification.`);
}
sendNotification(NotificationPriority.URGENT);
```

3 - PARA FUNÇÕES COM MÚLTIPLOS PARÂMETROS USE OBJETOS

❌

```
function createUser(
  name: string,
  age: number,
  email: string,
  isAdmin: boolean
){}
createUser("Alice", 30, "alice@example.com", true)
```

✅

```
function createUser(params: CreateUserParams){
   const { name,age,email,role } = params
}
createUser({
  name: "Alice",
  age: 30,
  email: "alice@example.com",
  role: UserRole.ADMIN
})
```

4 - USE OBJETOS PARA REMOVER IFS

```ts
export function getOrderMessage(status: string): string {
  if (status === 'pending') {
    return 'Order is pending.'
  } else if (status === 'shipped') {
    return 'Order has been shipped.'
  } else if (status === 'delivered') {
    return 'Order has been delivered.'
  }
}
```

❌

✅

```ts
const orderMessages = {
  [OrderStatus.Pending]: 'Order is pending.',
  [OrderStatus.Shipped]: 'Order has been shipped.',
  [OrderStatus.Delivered]: 'Order has been delivered.',
} as const

function getOrderMessage(status: OrderStatus): OrderMessage {
  return orderMessages[status]
}
```

5 - USE EARLY RETURN PARA REDUZIR A INDENTAÇÃO DOS IFS

❌

```
function processUser(user) {
  if (user.isActive) {
    if (user.age > 18) {
      console.log("Processing user...");
    } else {
      console.log("User is underage.");
    }
  } else {
    console.log("User is inactive.");
  }
}
```

✅

```
function processUser(user) {
  if (!user.isActive) return console.log("User is inactive.");
  if (user.age <= 18) return console.log("User is underage.");
  console.log("Processing user...");
}
```

6 - USE CONST PARA TORNAR SEUS IF MAIS LEGÍVEIS

❌

```
function canAccessDashboard(user){
  return user.isActive &&
         user.age >= 18 &&
         user.subscriptionStatus === "active";
}
```

✅

```
function canAccessDashboard(user) {
  const isUserActive = user.isActive;
  const isUserOldEnough = user.age >= 18;
  const hasActiveSubscription =
        user.subscriptionStatus === "active";

  return isUserActive &&
         isUserOldEnough &&
         hasActiveSubscription;
}
```

7 - ESCREVA CÓDIGO AUTOEXPLICATIVO E ELIMINE COMENTÁRIOS

❌

```
// Função que checa se um número é válido para fazer algum cálculo
function checkIfValidNumber(num: number): boolean {
  if (num <= 0) return false
  if (num % 1 !== 0) return false;
  if (num > 1000) return false;
  return true;
```

✅

```
function isValidForCalculation(number: number){
  const isPositive = number > 0;
  const isInteger = number % 1 === 0;
  const isNotTooLarge = number <= 1000;
  return isPositive && isInteger && isNotTooLarge;
}
```

8 - UTILIZE HAS E IS PARA TORNAR SEUS BOOLEANOS MAIS DESCRITIVOS

❌

```
const permission = user => user.role === 'admin';
const age = age => age >= 18;
const active = user => user.subscriptionStatus === 'active';
```

✅

```
const hasPermission = user => user.role === 'admin';
const isAdult = age => age >= 18;
const hasActiveSubscription =
user => user.subscriptionStatus === 'active';
```

9 - CRIE UM PADRÃO DE NOMENCLATURA

❌

```
modifyUser()
editUser()
refreshUser()
changeUser()
adjustUser()
alterUser()
reviseUser()
```

■ - ação

■ - escopo

updateUser()
createUser()
deleteUser()
getUser()

✅

```
updateUser()
```

10 - UTILIZE MÉTODOS DO JAVASCRIPT PARA MELHORAR A LEGIBILIDADE

❌

```
function getActiveUsers(users){
  const activeUsers = [];
  for (let i = 0; i < users.length; i++) {
    if (users[i].isActive) {
      activeUsers.push(users[i].name);
    }
  }
  return activeUsers;
}
```

✅

```
function getActiveUsers(users){
  return users
    .filter(user => user.isActive)
    .map(user => user.name);
}
```

CLEAN CODE

NA PRATICA

1PT - NOMES SIGNIFICATIVOS

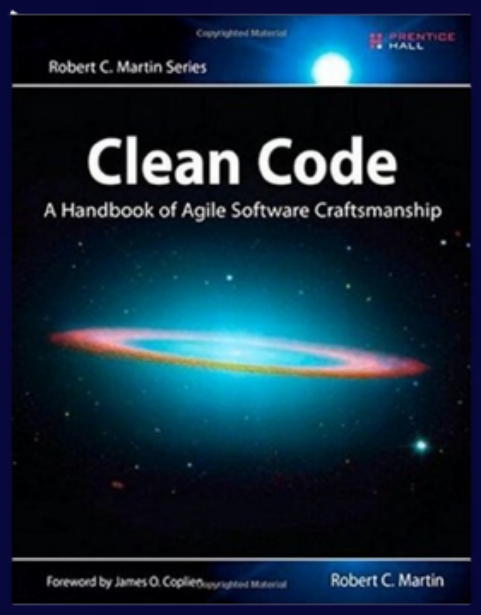
```
const data = []
const results = []
const json = []
const filter = []
const active = []
```

Quanto nome ruim em um só lugar 🤮

Isaac Gomes

TS

CLEAN CODE



Clean Code é um conceito que se refere à escrita de código de forma clara, organizada e legível, facilitando a manutenção, entendimento e evolução de sistemas de software. A ideia central é que o código deve ser simples e direto, evitando complexidade desnecessária e favorecendo a legibilidade para outros desenvolvedores.

Isaac Gomes

Capitulo 1

NOMES SIGNIFICATIVOS

Nomes que revelam seus Propósitos

pensemos um cenário onde um usuário vai ter acesso a um recurso específico no premium da categoria dele... qual seria um bom nome para a const que guarda o user???

Isaac Gomes

Capitulo 1

NOMES SIGNIFICATIVOS

Nomes que revelam seus Propósitos

```
const user = {}
```

❌ - *pouco descritivo*

```
// user represents a premium user
// with access to special features
const user = {}
```

❌ - *comentario para se explicar*

```
const premiumUserWithAccessToXFeature = {}
```

✅ - *autoexplicativo*

Isaac Gomes

TS

Capitulo 1

NOMES SIGNIFICATIVOS

Evite Informações Erradas

pensemos em um cenário onde temos que filtrar usuários banidos pense em um nome que pode te induzir ao erro

Isaac Gomes

Capitulo 1

NOMES SIGNIFICATIVOS

Evite Informações Erradas

```
function filterActiveUsers(users){
    return users.filter(user => user.isBanned);
}
```

❌ *- nome informa errado*

```
function filterBannedUsers(users){
    return users.filter(user => user.isBanned);
}
```

✅ *- nome que informa de maneira correta*

Isaac Gomes

Capitulo 1

NOMES SIGNIFICATIVOS

Distinções significativas

evite usar termos similares
para informações distintas
deixe o mais legível possível

Isaac Gomes

Capitulo 1

NOMES SIGNIFICATIVOS

Distinções significativas

```
const user1 = {}
const user2 = {}
```

❌ ***- nomes similares***

```
const adminUser = {}
const guestUser = {}
```

✅ ***- nomes distintos***

Isaac Gomes

Capitulo 1

NOMES SIGNIFICATIVOS

nomes pronunciáveis

use nomes pronunciáveis,
evite abreviações que podem
gerar confusões

Isaac Gomes

Capitulo 1

NOMES SIGNIFICATIVOS

nomes pronunciáveis

```
const t3rmlntrs: number; = 10
```

❌ *- contem abreviações confusas*

```
const terminalCount: number; = 10
```

✅ *- sem abreviações e claro*

Isaac Gomes

Capitulo 1

NOMES SIGNIFICATIVOS

nomes passíveis de busca

use nomes descritivos que podem ser pesquisados com facilidade

Isaac Gomes

Capitulo 1

NOMES SIGNIFICATIVOS

nomes passíveis de busca

```
const x: number = 25
```

❌ ***- nome sem busca clara***

```
const temperatureInCelsius: number = 25
```

✅ ***- nome passivel de busca***

Isaac Gomes

Capitulo 1

NOMES SIGNIFICATIVOS

Não de uma de espertinho

Não de uma de espertinho evite nomes genéricos ou vagos... mesmo que tenha contexto da criação escreva de maneira que qualquer um possa ler

Isaac Gomes

Capitulo 1

NOMES SIGNIFICATIVOS

Não de uma de espertinho

```
class DataProcessor {
    process(data) {}
}
```

❌ *- nome generico*

```
class UserProfileProcessor {
    process(userProfile) {}
}
```

✅ *- nome explicativo*

Isaac Gomes

TS

Capitulo 1

NOMES SIGNIFICATIVOS

Não faça trocadilhos

Não coloque nomes engraçadinhos ou com mais de uma interpretação possível seja objetivo e claro

Isaac Gomes

Capitulo 1

NOMES SIGNIFICATIVOS

Não faça trocadilhos

```
class DateShifter {
    shiftDate(date, days) {
        return new Date(date.getTime() + days * 86400000);
    }
}
const timeTraveler = new DateShifter();
const futureDate = timeTraveler.shiftDate(new Date(), 3);
```

❌ ***- O nome DateShifter parece engraçado ou criativo, mas o termo "shifter" pode ser confundido com outras coisas (por exemplo, manipulação de bits ou mudanças de estado) e não comunica diretamente que a classe lida com cálculos de datas.***

Isaac Gomes

TS

Capitulo 1

NOMES SIGNIFICATIVOS

Não faça trocadilhos

```
class DateCalculator {
    addDaysToDate(date, days) {
        return new Date(date.getTime() + days * 86400000)
    }
}
const dateCalculator = new DateCalculator();
const futureDate = dateCalculator.addDaysToDate(new Date(), 3);
```

✅ - O nome DateCalculator comunica claramente que a classe realiza cálculos com datas. Além disso, o nome do método addDaysToDate explica explicitamente o que o método faz, tornando o código mais fácil de entender para todos que o leem.

Isaac Gomes

Capitulo 1

NOMES SIGNIFICATIVOS

Não faça trocadilhos

```
class DateCalculator {
    addDaysToDate(date, days) {
        return new Date(date.getTime() + days * 86400000)
    }
}
const dateCalculator = new DateCalculator();
const futureDate = dateCalculator.addDaysToDate(new Date(), 3);
```

✅ - O nome DateCalculator comunica claramente que a classe realiza cálculos com datas. Além disso, o nome do método addDaysToDate explica explicitamente o que o método faz, tornando o código mais fácil de entender para todos que o leem.

Isaac Gomes

COMÓ CRIAR NOMES CONSISTENTES NOS SEUS PROJETOS

```
createUser()
deleteUser()
updateUser()
getUser()
```

UMA DAS COISAS MAIS DIFÍCEIS EM PROGRAMAÇÃO É DAR BONS NOMES

até pq temos muitos nomes possíveis para a mesma coisa

```
updateUser()
modifyUser()
editUser()
refreshUser()
changeUser()
adjustUser()
alterUser()
```

para resolver isso podemos criar padrões que facilitem essa escrita

POST - CREATE

PUT - UPDATE

DELETE - DELETE

GET - GET

- ação

- escopo

updateUser()

createUser()

deleteUser()

getUser()

PUT - UPDATE

❌

```
modifyUser()
editUser()
refreshUser()
changeUser()
adjustUser()
alterUser()
reviseUser()
```

✅

```
updateUser()
```

POST - CREATE

❌

`registerUser()`

`addUser()`

`newUser()`

`saveUser()`

`insertUser()`

✅

`createUser()`

DELETE - DELETE

❌

```
removeUser()
discardUser()
eraseUser()
destroyUser()
deactivateUser()
purgeUser()
eliminateUser()
```

✅

```
deleteUser()
```

GET - GET

❌

```
fetchUser()
retrieveUser()
loadUser()
findUser()
```

✅

```
getUser()
```

ESCREVA CÓDIGOS QUE SE DOCUMENTEM

```
// ❌ ERRADO
// Formats a user object with
// full name and adult status
function processData(user) {}


// ✅ CORRETO
function formatUserForDisplay(user) {}
```

```
// ❌ ERRADO
// Formats a user object with
// full name and adult status
function processData(user) {}
```

JÁ SE DEPAROU COM ESSE TIPO DE PROBLEMA?

- **PROCESSA O QUÊ EXATAMENTE?**
- **QUAL É O ESCOPO?**
- **QUAL É A FINALIDADE?**

MAS RELAXA, TEM UM COMENTARIO EXPLICANDO

```
// Formats a user object with
// full name and adult status
```

Ao chamar a função, verei o comentário?

```
// Formats a user object with
// full name and adult status
```

Não, mas você pode abrir o arquivo para ver.

VAMOS CRIAR CÓDIGOS QUE SE DOCUMENTEM POR SI SÓ

```
function formatUserForDisplay(user) {}
```

o que faz?

Formata

```
formatUserForDisplay(user) {}
```

Formata o Usuário

```
formatUserForDisplay(user) {}
```

Formata o Usuario para onde ?

exibição

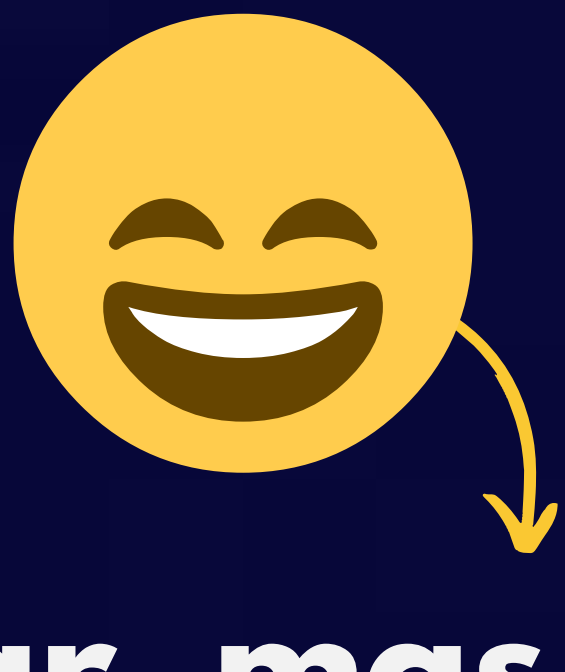
Que lindo cara!

```
// Formats a user object with
// full name and adult status
function processData(user) {}
```

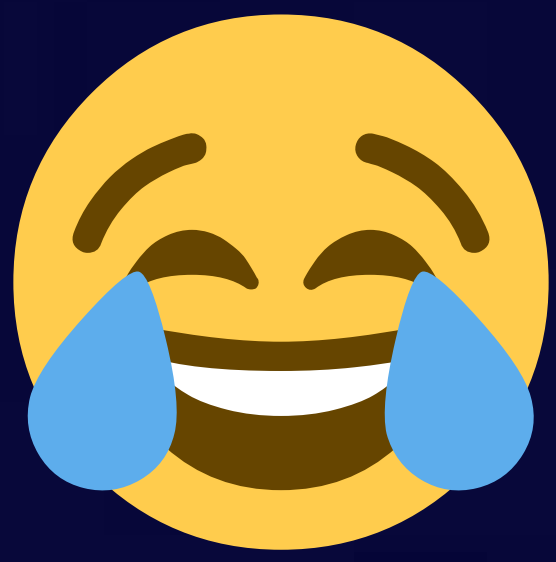
```
function formatUserForDisplay(user) {}
```

Agora, ao chamar, fica claro o que faz

então não posso usar comentários?



Pode usar, mas apenas quando houver uma necessidade real.



Agora se vc faz isso... melhore por favor

```
/* Criar usuário */
function createUser() {}
```

PARE DE ESCREVER ESSE TIPO DE FUNÇÃO

```
//❌ ERRADO
handleSubmit(newUser, false)
```

```
//✅ CORRETO
handleUserAction({
    action: UserActionType.CREATE
    data: user,
})
```

```
//❌ ERRADO
handleSubmit(newUser, false)
```

Vamos analisar os problemas dessa função

1 - parâmetros não nomeados

2 - Passar parâmetros booleanos pode dificultar a compreensão do código, pois não deixa claro o propósito do valor.

```
//✅ CORRETO
handleUserAction({
    action: UserActionType.CREATE
    data: user,
})
```

Resolvendo

1 -Vamos organizar nossos parâmetros usando um objeto com nomes descritivos.

2 - usaremos um enum para tornar o código ainda mais descritivo e fácil de entender.

```
//❌ ERRADO
const arrayUsers =
```

Redundância

A tipagem já mostra que é um array.

Semântica ambígua

O nome não indica claramente o propósito

```
= fetchingUsers()
```

Ao utilizar algo como **fetchingUsers**, você acaba abrindo espaço para a criação de funções como...

```
retrieveProducts()
fetchPurchaseHistory()
```

```
getUsers()
retrieveProducts()
fetchPurchaseHistory()
```

Observe que todos realizam buscas, porém não há um padrão definido para a nomenclatura.

✅

```
getUsers()
getProducts()
getPurchaseHistory()
```

Agora, alteramos tudo para usar o prefixo do verbo "get", o que resultou em um código mais padronizado e claro

✅

```
const { users }
const { products }
const { userPurchaseHistory }
```

Agora, vamos alterar o retorno para um objeto, utilizando nomes que facilitem o seu uso e eliminem redundâncias como "users", "products", etc.

```
function getUsers() {}
export default getUsers

import fetchUser from './';
```

Para evitar a necessidade de renomeação, evite o uso de export default.

✅

```ts
export function getUsers() {}
import { getUsers } from './';
```

```ts
const { users } = getUsers()
```

Agora, além de utilizarmos o export, obtemos um retorno mais claro e objetivo.

PARE DE ESCREVER IFS DESSA MANEIRA

```
function processOrder(order) {
  if (order) {
    if (order.user.isActive) {
      if (order.status === 'pending') {
        if (order.amount > 0) {
          return ''
        }
      }
    }
  }
}
```

hadouken

use early return


```
function processOrder(order){
  if (!order) {
    return 'Invalid order.'
  }

  if (!order.user.isActive) {
    return 'User is not active.'
  }

  if (order.status !== 'pending') {
    return 'Order status is not pending.'
  }

  if (order.amount <= 0) {
    return 'Order amount must be greater than zero.'
  }

  return 'Order is valid and being processed.'
}
```

MELHORANDO OS NOMES DAS SUAS VARIÁVEIS

```
//❌ -errado
function checkAccess(userRole){}
```

```
//✅ -correto
function isUserAdmin(userRole) {}
```

UM **ERRO** COMUM É NOMEAR **FUNÇÕES** DE FORMA QUE NAO DEIXEM CLARO QUE RETORNAM UM **BOOLEAN**.

```
//❌-errado
function checkAccess(userRole){}
```

"VERIFICAR ACESSO" PODE PARECER UM BOM **NOME**, MAS NÃO DESCREVE CLARAMENTE A **FUNCIONALIDADE QUE ELE REALIZA**.

TEM UM JEITO MELHOR ???

```
//✅ -correto
function isUserAdmin(userRole) {}
```

"E ADMINISTRADOR" TORNA A FUNÇAO INTUITIVA E SEU RETORNO **BOOLEANO**, TORNANDO O COMPORTAMENTO **PREVISÍVEL**.

SE O BOOLEANO ESTIVER RELACIONADO A POSSE OU EXISTÊNCIA, UTILIZE "HAS"

```
//❌-errado
function checkPermission(userPermissions, requiredPermission) {
  const permission =
  userPermissions.includes(requiredPermission)
  return permission;
}


//✅-certo
function hasPermission(userPermissions, requiredPermission){
  const hasRequiredPermission =
  userPermissions.includes(requiredPermission)
  return hasRequiredPermission;
}
```

OBSERVE COMO O USO DE "IS" E "HAS" TRAZ MAIS CLAREZA AOS NOSSOS BOOLEANOS, TORNANDO O CÓDIGO MAIS SIMPLES E FÁCIL DE LER.

Gostou?

Curta

Compartilhe

Salve

Ative as notificações