

TypeScript na Prática

Explorando recursos para o dia a dia







TypeScript???

Se alguém ainda não conhece

- Superconjunto do JavaScript
- Uma linguagem de programação fortemente tipada
- Superset de JavaScript

Afinal, por que TypeScript?

Se JavaScript já roda em todo lugar, por que colocar “tipos” no meio do caminho?

- Visualização de erros em tempo de desenvolvimento
- Melhor legibilidade de código
- Autocomplete
- ...

```
// TypeScript (te puxa pela orelha)
function formatBRL(amountCents: number) {
  return (amountCents / 100).toFixed(2);
}
formatBRL("1000"); // ❌ Argument of type 'string' is not as
```



Colocando a mão na massa

Como o Typescript pode nos ajudar no dia a dia



Props para componentes

Podemos utilizar para substituir no bom e velho PropTypes

```
type AvatarProps = {  
  name: string  
  imageUrl?: string // opcional  
}  
  
export function Avatar({ name, imageUrl }: AvatarProps) {  
  return (  
    <div>  
      <img src={imageUrl ?? "/default.png"} alt={name} />  
      <p>{name}</p>  
    </div>  
  )  
}
```

enum

Garante constantes únicas e tipadas evitando typos em strings soltas

```
export enum DiscountSettlementSteps {  
  SIMULATE = 'simulacao',  
  PROPOSAL = 'proposta',  
  PIN = 'pin',  
  PAYMENT_METHOD = 'forma-de-pagamento',  
  REVIEW = 'revisao'  
}  
  
const steps: { key: DiscountSettlementSteps; render: () => JSX.Element }[] = [  
  {  
    key: DiscountSettlementSteps.PROPOSAL,  
    render: () => <RenegotiationDiscountAmountSimulation userEmail={userEmail} mode={mode} />  
  },  
  {  
    key: DiscountSettlementSteps.PIN,  
    render: () => <RenegotiationChallenge saveSpotOffer={setSpotOffer} />  
  }  
]
```

União de literais de string

Define que a variável só pode assumir um entre valores exatos. Ótimo para evitar typos, ganhar autocomplete e usar como discriminador em switch

```
type UserType = 'Customer' | 'Operator' | 'Leadership';

function canApprove(u: UserType) {
  return u === 'Leadership';
}

let u: UserType = 'Operator'; // ok
// u = 'Operater';           // ✗ erro: valor fora da união
```


Interseção

`&` em TypeScript é o operador de interseção: o valor precisa satisfazer todos os tipos ao mesmo tempo.

```
type Identifiable = { id: string };
type Timestamped = { createdAt: string; updatedAt: string };

type Payment = Identifiable & Timestamped & {
  amountCents: number;
  status: 'authorized' | 'captured' | 'refused';
};

// precisa ter todos os campos
const p: Payment = {
  id: 'p1',
  createdAt: '2025-08-01',
  updatedAt: '2025-08-10',
  amountCents: 1299,
  status: 'captured',
};
```

Interfaces aninhadas

Você modela um objeto complexo compondo interfaces menores (reutilizáveis). Isso melhora coesão, reuso e evolução independente dos “blocos” do domínio.

```
export interface InstallmentPlan {
  amount: number;
  count: number;
  totalSum: number;
  paymentReferenceDate: string;
  gracePeriod?: number;
  daysToDownPayment?: number;
}

export interface Renegotiation {
  securities: SecurityRenegotiation[];
  installmentPlan: InstallmentPlan;
}

export interface RenegotiationProposal {
  id: string;
  offerId: string;
  type: string;
  status: string;
  expirationDate: string;
  creationDate: string;
}
```

Tipando respostas da API

Podemos utilizar para substituir no bom e velho PropTypes

```
export interface ApiResponse {
  success: boolean
  message?: string
  data?: {
    id: string
    amountCents: number
    status: 'authorized' | 'captured' | 'refused'
    createdAt: string
  }
}

export async function getPayment(id: string): Promise<APApiResponseI> {
  const res = await http.get<ApiResponse>(`/payments/${id}`);
  return res.data.data;
}
```

type x interface

Podemos utilizar para substituir no bom e velho PropTypes

- Use `interface` para APIs de objeto que podem evoluir (ex.: modelos, props de componentes, contratos públicos, Window, “augment” de libs).
- Use `type` para unions, tuples, funções, mapped types e utilitários; também é ótimo para compor tipos rapidamente com `&` e para aliases simples.

Na prática:

- Props de componente → tanto faz, mas interface costuma ficar mais legível e extensível em libs.
- Tipos de domínio (Money, IDs, DTOs) → qualquer um. Se precisar de utilitários/mapeados, prefira type.
- Chaves i18n/rotas/eventos (template literal/union) → type.

Generics

Aqui as coisas começam a ficar interessantes

Generics são uma forma de criar **funções, classes ou tipos** que funcionam com vários tipos diferentes, mas sem perder a **segurança de tipo**.

É como se você quisesse dizer:

“Não sei ainda que tipo vou receber, mas quando eu souber, vou trabalhar com ele corretamente.”

Generics

```
function identidade(valor: any): any {  
    return valor  
}
```

Esse código funciona perfeitamente, porém, não temos segurança nenhuma. Se você passar uma string, pode receber um número e o Typescript não vai reclamar de nada.

```
function identidade<T>(valor: T): T {  
    return valor  
}  
  
const a = identidade<string>("Olá") // a é do tipo string  
const b = identidade<number>(10)    // b é do tipo number  
const idade = identidade(25)        // TypeScript vai infe
```

Nesse exemplo, `T` é um tipo genérico (que pode ser qualquer tipo: string, number, object, etc.).

Por que eu deveria usar Generics?

- Reutilização: Você escreve uma função que serve para vários tipos.
- Segurança: mantém o tipo correto durante a execução do código.
- Clareza: facilita a legibilidade e entendimento do que esperar da função e do seu retorno.

Com arrays

```
function primeiroElemento<T>(array: T[]): T {  
    return array[0]  
}  
  
const num = primeiroElemento([1, 2, 3])    // number  
const str = primeiroElemento(["a", "b"])    // string
```

Com interfaces

```
interface Caixa<T> {  
    valor: T  
}  
  
const caixaDeString: Caixa<string> = { valor: "Texto" }  
const caixaDeNumero: Caixa<number> = { valor: 123 }
```

Constraints

Constraints são limites/regras em genéricos (via `extends`) que dizem quais tipos são aceitos por um parâmetro de tipo.

```
function getName<T extends { name: string }>(obj: T): string {  
    return obj.name;  
}
```

```
getName({ nome: "João", idade: 30 }) // ✓
```

```
getName({ idade: 30 }) // ✗ erro: falta o nome
```

- `T extends { name: string }` é uma **constraint** do genérico `T` : ela diz que qualquer tipo usado como `T` precisa ser atribuível a `{ name: string }`.
- `T` deve ter pelo menos a propriedade `name` do tipo `string`. Ele pode ter campos extras, mas não pode faltar `name`.
- Como consequência, dentro da função, `obj.name` é seguro e o retorno pode ser `string` sem `undefined`.

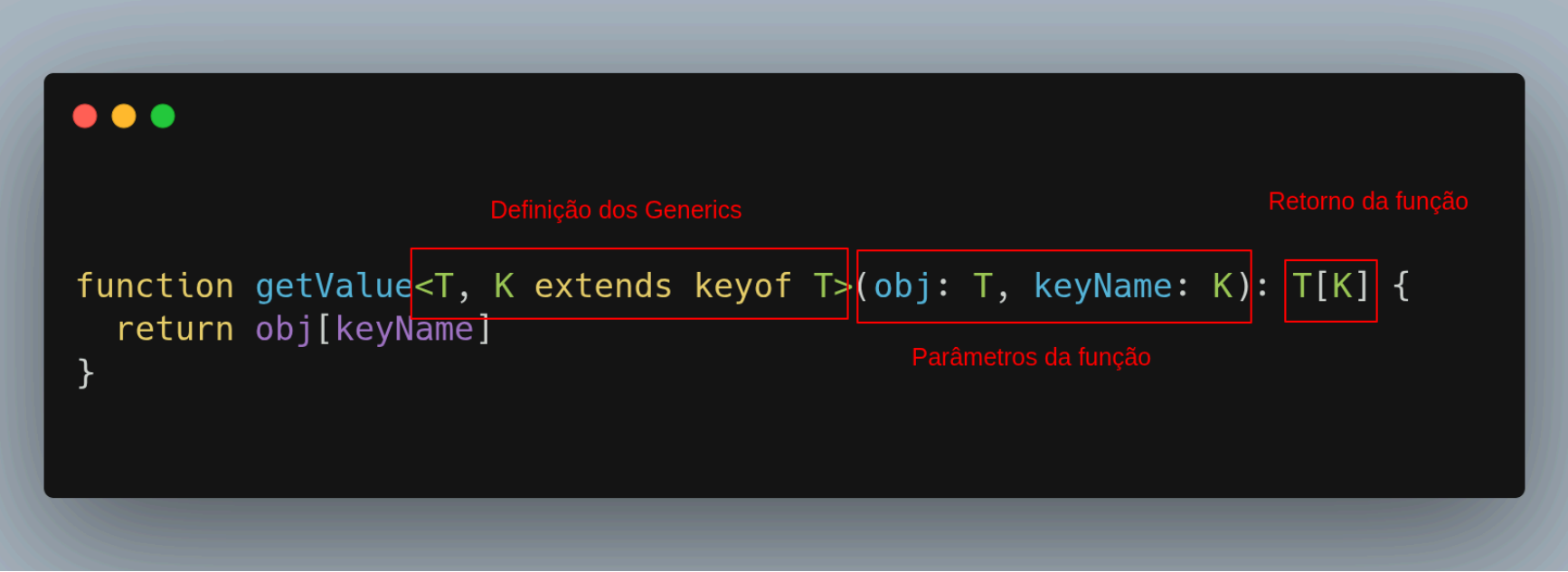
keyof com Generics

Você pode criar funções que acessam propriedades dinamicamente

```
function getValue<T, K extends keyof T>(obj: T, keyName: K): T[K] {  
    return obj[keyName]  
}  
  
const pessoa = { nome: "Ana", idade: 25 }  
  
const nome = getValue(pessoa, "nome") // tipo: string  
const idade = getValue(pessoa, "idade") // tipo: number  
  
getValue(pessoa, "altura") // ✗ erro: 'altura' não existe em 'pessoa'
```

E aqui as coisas podem ficar bem confusas

Então, vamos por partes



```
function getValue<T, K extends keyof T>(obj: T, keyName: K): T[K] {  
    return obj[keyName]  
}
```

Definição dos Generics

Parâmetros da função

Retorno da função

Conditional Types (if/else para tipos)

- Decidem qual shape usar em tempo de tipo com `T extends U ? X : Y`.
- É como um if do TypeScript: adapta interfaces/campos conforme condição (status, método, flag), com autocomplete e bloqueio de combinações inválidas.

```
type Choose<T> = T extends string ? 'é string' : 'não é string';
```

```
interface ApiState<Loading extends boolean> {  
  loading: Loading;  
  data: Loading extends true ? null : { id: string };  
  error: Loading extends true ? null : string | null;  
}
```

Conditional Types Exemplo

```
type Method = 'pix' | 'boleto';

interface PaymentCommon {
  id: string;
  amountCents: number;
}

interface PixFields {
  method: 'pix';
  pixCode: string;    // E2E do PIX
  payerKey: string;    // chave PIX (telefone, email, EV)
}

interface BoletoFields {
  method: 'boleto';
  barcode: string;    // código de barras
  dueDate: string;    // ISO date
}
```

```
// Conditional type escolhe o shape certo conforme M
type Payment<M extends Method> =
  M extends 'pix'
    ? PaymentCommon & PixFields
    : PaymentCommon & BoletoFields;

/** Exemplos de uso */
const p1: Payment<'pix'> = {
  id: 'p1',
  amountCents: 1299,
  method: 'pix',
  pixCode: 'E123 ... ',
  payerKey: 'email@dominio.com',
};

const p2: Payment<'boleto'> = {
  id: 'p2',
  amountCents: 2590,
  method: 'boleto',
  barcode: '23793 ... ',
  dueDate: '2025-09-01',
};
```

?

