Tipos de Variáveis no Dart: Flexibilidade com Segurança

Fortemente Tipada

Dart é uma linguagem **fortemente tipada**, o que significa que cada variável tem um tipo definido. Isso garante maior segurança e menos erros em tempo de execução.

Inferência de Tipo

Apesar de fortemente tipada, Dart oferece **flexibilidade com inferência de tipo** (usando `var`), permitindo que o compilador determine o tipo da variável automaticamente.

Primitivos e Coleções

Suporta uma vasta gama de **tipos primitivos** (números, texto, booleanos) e **coleções** (listas, mapas), essenciais para estruturar dados.

Segurança em Tempo de Compilação

A tipagem no Dart contribui para a **segurança em tempo de compilação**, detectando incompatibilidades de tipo antes mesmo de o código ser executado.

Produzido por MARCIO SCHOENFELDER

Variáveis Numéricas e de Texto

Números Inteiros (int)

Representam números inteiros, sem casas decimais. No Dart, um `int` pode armazenar valores de **-9.223.372.036.854.775.808** a **9.223.372.036.854.775.807**.

```
int idade = 30; // Exemplo de uso
```

Números de Ponto Flutuante (double)

Utilizados para números com casas decimais. São implementados como números de **64 bits** de precisão dupla, seguindo o padrão

```
double altura = 1.75; // Exemplo de uso
```

Sequência de Caracteres (String)

Representam texto, que é uma **sequência de caracteres Unicode**. Podem ser declaradas com aspas simples ou duplas.

```
String nome = 'Maria'; // Exemplo de uso
```

```
String saudacao = "Olá, mundo!";
```

Booleano, Listas e Mapas: Outros Tipos Essenciais

Booleano (bool)

Representa valores lógicos de **verdadeiro** ou **falso**. Fundamental para controle de fluxo em programas.

```
bool ativo = true;
```

Listas (List)

Coleção **ordenada** de objetos, similar a arrays em outras linguagens. Permitem armazenar múltiplos valores do mesmo tipo ou de tipos variados.

```
List<int> nums = [1, 2, 3];
```

Mapas (Map)

Coleção de **pares chave-valor**, semelhante a dicionários. Cada chave é única e está associada a um valor.

```
Map<String, String> paises =
{'BR': 'Brasil', 'US':
'Estados Unidos'};
```

Declaração de Variáveis: `var`, `final` e `const`

`var`: Inferência Inteligente

A palavra-chave `var` permite que o tipo da variável seja **inferido dinamicamente** pela primeira atribuição de valor. Uma vez inferido, o tipo não pode ser alterado.

```
var preco = 9.99; // preco é inferido como double
```

`final`: Valor Único em Tempo de Execução

Com `final`, o valor da variável pode ser **atribuído apenas uma vez**, em tempo de execução. Ideal para valores que não mudarão após a inicialização.

```
final PI = 3.14159; // PI não pode ser reatribuído
```

`const`: Constante em Tempo de Compilação

Utilize `const` para declarar variáveis com **valores constantes que já são conhecidos em tempo de compilação**. Isso otimiza a performance, pois o valor é fixo antes do programa rodar.

```
const ANO_ATUAL = 2024;
```

Entrada de Dados no Dart: Interagindo com o Usuário via Console

Para ler dados do usuário via console em Dart, utilizamos a biblioteca `dart:io`, que fornece funcionalidades de entrada e saída de dados.

A função principal para entrada de texto é `stdin.readLineSync()`. Ela lê uma linha de texto do console até que o usuário pressione Enter.

É importante notar que `stdin.readLineSync()` retorna uma `String?` (string anulável), o que significa que o valor pode ser nulo caso não haja entrada. A manipulação de nulos é crucial aqui.

```
import 'dart:io';
```

```
String? input = stdin.readLineSync();
```

Processando a Entrada de Dados: Convertendo e Validando

Conversão de Tipos

Frequentemente, a entrada do usuário (sempre uma `String`) precisa ser convertida para outros tipos de dados, como números. Para inteiros, usamos `int.parse()`, e para decimais, `double.parse()`.

Tratamento de Erros

A conversão pode falhar se a string não estiver no formato esperado. É fundamental usar **blocos** `try-catch` para tratar essas exceções e garantir a robustez do programa, evitando falhas inesperadas.

Exemplo Prático

O exemplo abaixo demonstra como ler uma entrada, convertê-la para um inteiro e lidar com o caso de entrada nula ou inválida.

O operador `!` aqui significa que você está dizendo que `input` não é null ➔

```
try {
  int numero = int.parse(input!);
  print('Você digitou: $numero’);
} catch (e) {
    print('Entrada inválida.’);
}
```

Processando a Entrada de Dados: Convertendo e Validando

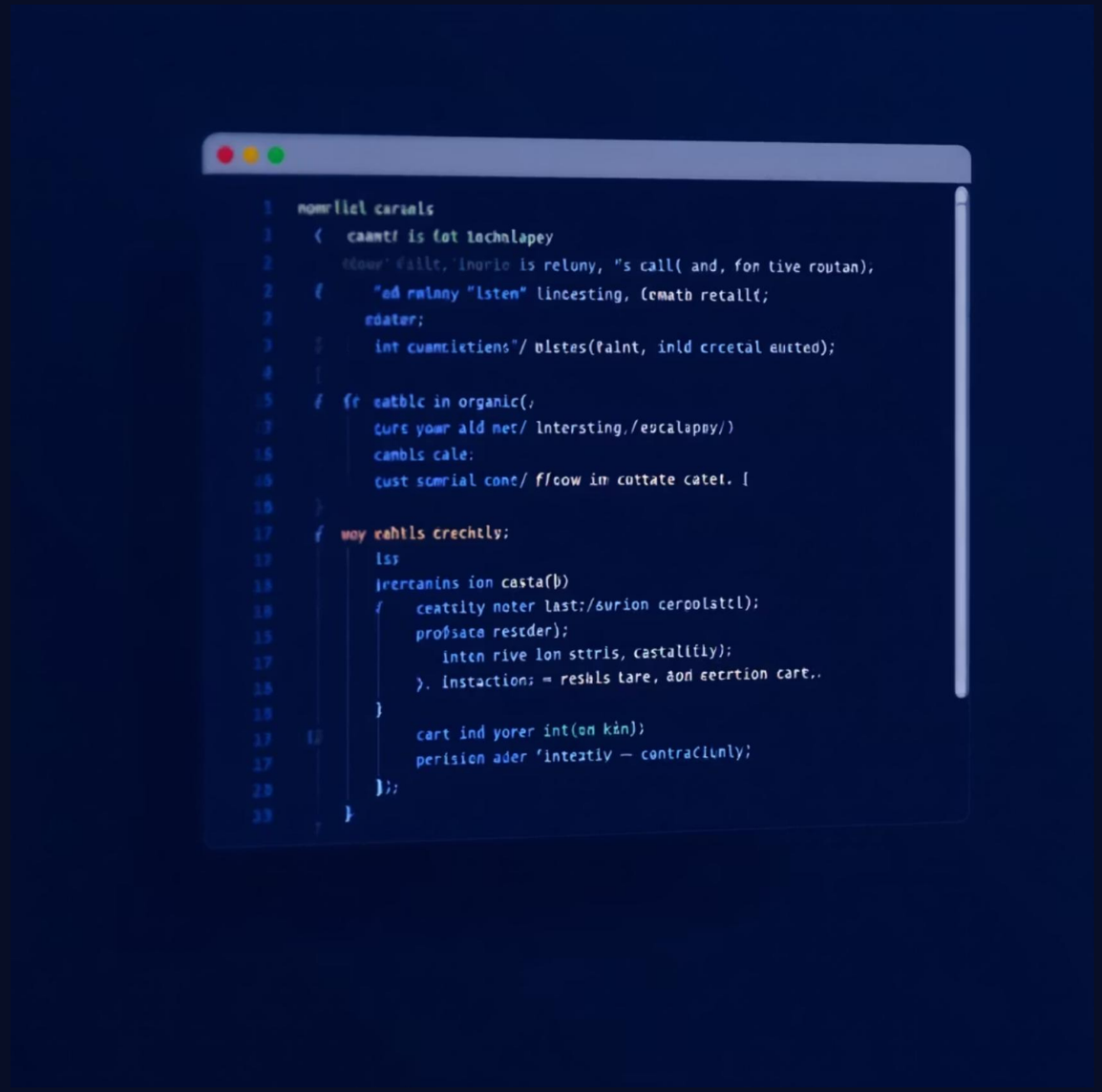
O que acontece:

input! : O ! diz ao compilador "Eu tenho certeza de que input não é null". Caso input seja null, o programa lance um erro de execução.

int.parse(input!) : Aqui, o Dart tentará converter o valor de input para um número inteiro. Se input não for uma string válida que pode ser convertida para um número, o Dart lançará uma exceção que será capturada pelo bloco catch.

Tratamento do erro : Se a entrada do usuário for inválida (como uma string não numérica), o catch captura a exceção e exibe uma mensagem de erro.

Saída de Dados no Dart: A Função `print()`



A função `print()` é a maneira mais comum e simples de **exibir informações no console** em Dart. Ela é extremamente versátil, aceitando qualquer tipo de objeto como argumento.

Para combinar texto e variáveis de forma elegante, utilizamos a **interpolação de string**. Basta usar o prefixo `$` antes do nome da variável ou `${}` para expressões mais complexas dentro de uma string literal.

```
String nome = 'Maria';
```

```
int idade = 30;
```

```
print('Olá, $nome! Sua idade é $idade.');
```

```
print('Em 5 anos, ${idade + 5} anos.');
```

Conclusão e Próximos Passos no Dart

1

Base Sólida

As **variáveis** são os blocos de construção fundamentais do Dart, permitindo armazenar e manipular dados. A **entrada e saída de dados** possibilitam a interação de seus programas com os usuários.

2

Caminho para o Sucesso

Compreender esses conceitos é crucial para desenvolver programas mais complexos e dinâmicos. Eles são a espinha dorsal para qualquer aplicação Dart, incluindo o desenvolvimento de interfaces com Flutter.

3

Desafie-se!

Agora que dominamos os fundamentos, o próximo passo é explorar as **estruturas de controle** (condicionais, laços de repetição) e as **funções**, que permitirão organizar e reutilizar seu código de forma eficiente.