Introdução ao Dart - Conteúdo Prático

1. Sintaxe Básica

A sintaxe do Dart é simples e direta, semelhante a outras linguagens como JavaScript e Java.

Exemplos práticos:

```
// Declaração de uma função principal
void main() {
 print('Hello, Dart!');  // Imprime uma mensagem no console
}
```

2. Tipos de Dados e Variáveis

1. Tipos Numéricos

- **int**:
    - Armazena números inteiros, positivos ou negativos, sem partes decimais.
    - Ex.: `int idade = 30;`
    - Range: Depende do sistema, mas normalmente pode armazenar valores de -2^53 a 2^53 - 1.
- **double**:
    - Armazena números de ponto flutuante (decimais).
    - Ex.: `double altura = 1.85;`
    - Útil para valores que precisam de precisão decimal, como medições.

- **num**:
    - Supertipo de `int` e `double`. Pode armazenar tanto inteiros quanto números de ponto flutuante.
    - Ex.: `num preco = 5.99;`
    - Usado quando não há certeza se o valor será inteiro ou decimal.

2. Tipos de Texto

- **String**:
    - Armazena sequências de caracteres (texto).
    - Ex.: `String nome = 'Keite';`
    - Pode usar aspas simples `'` ou duplas `"` para definir valores.
    - Permite interpolação com `$variavel` ou `${expressao}`.
    - Ex.: `String mensagem = 'Olá, $nome!';`

3. Tipos Booleanos

- **bool**:
    - Armazena valores verdadeiros ou falsos (`true` ou `false`).
    - Ex.: `bool isAtivo = true;`
    - Usado em condições e controle de fluxo (if, while).

4. Tipos de Coleções

- **List**:
    - Armazena uma lista ordenada de valores, que podem ser de qualquer tipo.
    - Pode ser tipada, como `List<int>` ou não tipada, `List`.
    - Ex.: `List<int> numeros = [1, 2, 3];`
    - Suporta métodos para adicionar, remover e acessar elementos.

- **Set**:
  - Armazena uma coleção de valores únicos, não ordenados.
  - Ex.: `Set<String> cores = {'vermelho', 'azul', 'verde'};`
  - Útil quando a unicidade dos elementos é importante.
- **Map**:
  - Armazena pares de chave-valor, similar a um dicionário.
  - Ex.: `Map<String, String> capitais = {'Moçambique': 'Maputo', 'Portugal': 'Lisboa'};`
  - As chaves são únicas e podem ser de qualquer tipo.

5. Tipos Dinâmicos

- **var**:
  - Determina automaticamente o tipo da variável com base no valor atribuído.
  - Uma vez atribuído, o tipo não pode ser alterado.
  - Ex.: `var idade = 25;` (inferido como `int`).
- **dynamic**:
  - Pode armazenar valores de qualquer tipo e pode mudar durante a execução.
  - Ex.: `dynamic qualquerCoisa = 'texto'; qualquerCoisa = 42;`
  - Útil em situações onde o tipo pode variar, mas deve ser usado com cuidado para evitar erros.
- **Object**:
  - Tipo base de todos os tipos em Dart.

- Pode armazenar qualquer valor, mas ao contrário de `dynamic`, não permite chamadas de métodos que não são comuns a todos os objetos sem antes fazer um *casting*.
- Ex.: `Object objeto = 'Uma string';`

6. Tipos Especiais

- **Null**:
  - Representa um valor nulo ou ausência de valor.
  - Em Dart, desde o *null-safety*, é necessário declarar variáveis como nulas usando `?`.
  - Ex.: `String? nome;` (indica que `nome` pode ser `null`).

1. Operadores Aritméticos

- Usados para realizar operações matemáticas básicas.
  - `+` (Adição): Soma dois valores.
    - Ex.: `int soma = 5 + 3;` // Resultado: `8`
  - `-` (Subtração): Subtrai um valor do outro.
    - Ex.: `int diferenca = 5 - 3;` // Resultado: `2`
  - `*` (Multiplicação): Multiplica dois valores.
    - Ex.: `int produto = 5 * 3;` // Resultado: `15`
  - `/` (Divisão): Divide um valor pelo outro e retorna um `double`.
    - Ex.: `double divisao = 5 / 2;` // Resultado: `2.5`
  - `~/` (Divisão Inteira): Divide e retorna apenas a parte inteira.
    - Ex.: `int divisaoInteira = 5 ~/ 2;` // Resultado: `2`
  - `%` (Módulo): Retorna o resto da divisão.
    - Ex.: `int resto = 5 % 2;` // Resultado: `1`

2. Operadores de Atribuição

- Usados para atribuir valores a variáveis.
  - `=` (Atribuição): Atribui um valor à variável.
    - Ex.: `int a = 5;`
  - `+=` (Adição e Atribuição): Soma e atribui.
    - Ex.: `a += 2;` // `a` se torna `7`.
  - `-=` (Subtração e Atribuição): Subtrai e atribui.
    - Ex.: `a -= 2;` // `a` se torna `3`.
  - `*=` (Multiplicação e Atribuição): Multiplica e atribui.
    - Ex.: `a *= 2;` // `a` se torna `10`.
  - `/=` (Divisão e Atribuição): Divide e atribui.
    - Ex.: `a /= 2;` // `a` se torna `2.5`.

3. Operadores de Comparação

- Usados para comparar valores e retornam `bool` (`true` ou `false`).
  - `==` (Igualdade): Verifica se dois valores são iguais.
    - Ex.: `bool isIgual = (5 == 5);` // `true`.
  - `!=` (Diferença): Verifica se dois valores são diferentes.
    - Ex.: `bool isDiferente = (5 != 3);` // `true`.
  - `>` (Maior que): Verifica se um valor é maior que o outro.
    - Ex.: `bool isMaior = (5 > 3);` // `true`.
  - `<` (Menor que): Verifica se um valor é menor que o outro.
    - Ex.: `bool isMenor = (5 < 3);` // `false`.
  - `>=` (Maior ou Igual): Verifica se um valor é maior ou igual ao outro.
    - Ex.: `bool isMaiorOuIgual = (5 >= 5);` // `true`.

- `<=` (Menor ou Igual): Verifica se um valor é menor ou igual ao outro.
  - Ex.: `bool isMenorOuIgual = (3 <= 5);` // `true`.

4. Operadores Lógicos

- Usados para combinar expressões booleanas.
  - `&&` (E lógico): Retorna `true` se ambas as expressões forem verdadeiras.
    - Ex.: `bool resultado = (5 > 3) && (3 < 4);` // `true`.
  - `||` (Ou lógico): Retorna `true` se pelo menos uma das expressões for verdadeira.
    - Ex.: `bool resultado = (5 > 3) || (3 > 4);` // `true`.
  - `!` (Negação lógica): Inverte o valor booleano.
    - Ex.: `bool resultado = !(5 > 3);` // `false`.

5. Operadores de Incremento e Decremento

- Usados para incrementar ou decrementar valores de variáveis.
  - `++` (Incremento): Adiciona 1 ao valor da variável.
    - Ex.: `int a = 5; a++;` // `a` se torna `6`.
  - `--` (Decremento): Subtrai 1 do valor da variável.
    - Ex.: `int b = 5; b--;` // `b` se torna `4`.

6. Operadores de Condicional

- Usados para executar expressões condicionais.
  - `?:` (Operador Ternário): Retorna um valor baseado em uma condição.

        - Ex.: `int a = (5 > 3) ? 10 : 20;` // `a` se torna `10`.
    - `??` (Operador de Coalescência Nula): Retorna o valor da esquerda se ele não for `null`, caso contrário, retorna o valor da direita.
        - Ex.: `String nome = null; String resultado = nome ?? 'Sem Nome';` // `resultado` se torna `'Sem Nome'`.

3. Controle de Fluxo

1. Condicional `if`, `else if`, e `else`

- O `if` é usado para executar um bloco de código se uma condição for verdadeira. O `else if` é usado para testar outra condição caso a primeira seja falsa, e o `else` é executado se nenhuma das condições anteriores for verdadeira.

Sintaxe:
dart
Copy code

```
if (condicao1) {
  // Executa se condicao1 for verdadeira
} else if (condicao2) {
  // Executa se condicao2 for verdadeira
} else {
  // Executa se nenhuma das condições acima for
verdadeira
}
```

-

Exemplo:
dart
Copy code

```
int idade = 18;

if (idade < 18) {
  print('Menor de idade');
} else if (idade == 18) {
  print('Tem exatamente 18 anos');
} else {
  print('Maior de idade');
}
```

-
- **Uso**: Neste exemplo, o código verifica a idade e imprime uma mensagem correspondente.

2. Operador Ternário

- Um operador ternário é uma forma compacta de um `if-else` para atribuições simples.

Sintaxe:
dart
Copy code

```
variavel = condicao ? valorSeVerdadeiro : valorSeFalso;
```

-

Exemplo:
dart

Copy code

```
int numero = 5;
String resultado = numero % 2 == 0 ? 'Par' : 'Ímpar';
print(resultado); // Saída: Ímpar
```

-
- **Uso**: O operador ternário avalia a expressão e atribui 'Par' se o número for par e 'Ímpar' se for ímpar.

3. `switch` e `case`

- O `switch` é usado quando há várias condições baseadas em uma única variável e cada caso é verificado para encontrar uma correspondência.

Sintaxe:
dart
Copy code

```
switch (variavel) {
  case valor1:
    // Código para valor1
    break;
  case valor2:
    // Código para valor2
    break;
  default:
    // Código se nenhum caso for verdadeiro
}
```

-

Exemplo:

dart

Copy code

```
String dia = 'segunda';

switch (dia) {
  case 'segunda':
    print('Começo da semana');
    break;
  case 'sexta':
    print('Fim da semana de trabalho');
    break;
  default:
    print('Dia comum');
}
```

-
- **Uso**: O `switch` compara o valor de `dia` com os `cases` e executa o código correspondente. O `default` é executado se nenhum `case` for correspondido.

4. Laço de Repetição `for`

- O `for` é usado para repetir um bloco de código um número específico de vezes.

Sintaxe:

dart

Copy code

```
for (inicializacao; condicao; incremento) {
```

```
  // Código a ser executado
}
```

-

Exemplo:
dart
Copy code

```
for (int i = 0; i < 5; i++) {
  print('Contagem: $i');
}
```

-
- **Uso**: Neste exemplo, o laço `for` executa o código 5 vezes, imprimindo os valores de `i` de 0 a 4.

5. Laço `for-in`

- Usado para iterar sobre cada elemento em uma coleção, como uma lista.

Sintaxe:
dart
Copy code

```
for (var elemento in colecao) {
  // Código a ser executado para cada elemento
}
```

-

Exemplo:
dart

Copy code

```
List<String> nomes = ['Ana', 'Carlos', 'Beatriz'];

for (var nome in nomes) {
  print('Olá, $nome');
}
```

-
- **Uso**: O `for-in` percorre cada elemento da lista `nomes` e imprime uma saudação.

6. Laço `while`

- O `while` executa um bloco de código enquanto uma condição for verdadeira.

Sintaxe:
dart
Copy code

```
while (condicao) {
  // Código a ser executado enquanto a condição for verdadeira
}
```

-

Exemplo:
dart
Copy code

```
int contador = 0;
```

```
while (contador < 5) {
  print('Contador: $contador');
  contador++;
}
```

-
- **Uso**: O `while` continua executando enquanto `contador` for menor que 5.

7. Laço `do-while`

- Similar ao `while`, mas garante que o código seja executado pelo menos uma vez antes de verificar a condição.

Sintaxe:
dart
Copy code
```
do {
  // Código a ser executado
} while (condicao);
```

-

Exemplo:
dart
Copy code
```
int numero = 0;

do {
  print('Número: $numero');
  numero++;
```

```
} while (numero < 3);
```

-
- **Uso**: O `do-while` imprime os números de 0 a 2, mesmo que a condição seja testada após cada execução.

8. Controle de Laços com `break` e `continue`

- **`break`**: Interrompe a execução de um laço ou um `switch` e sai dele.

Exemplo:
dart
Copy code

```
for (int i = 0; i < 5; i++) {
  if (i == 3) {
    break; // Sai do laço quando i é igual a 3
  }
  print(i);
}
// Saída: 0, 1, 2
```

  -

- **`continue`**: Pula a iteração atual e vai para a próxima.

Exemplo:
dart
Copy code

```
for (int i = 0; i < 5; i++) {
  if (i == 2) {
    continue; // Pula quando i é igual a 2
  }
```

```
  print(i);
}
// Saída: 0, 1, 3, 4
```

-

9. Assert (Verificação de Condições)

- O `assert` é usado para verificar condições durante a fase de desenvolvimento, parando a execução se a condição for falsa.

Sintaxe:
dart
Copy code

```
assert(condicao, 'Mensagem de erro opcional');
```

-

Exemplo:
dart
Copy code

```
int idade = 15;
assert(idade >= 18, 'A idade deve ser maior ou igual a 18.');
```

-
- **Uso**: O `assert` ajuda a encontrar erros lógicos em tempo de desenvolvimento.

Resumo

- **Controle condicional (`if`, `else if`, `else`)**: Decide qual bloco de código executar com base em condições.

- **`switch` e `case`**: Útil para múltiplas condições de um único valor.
- **Laços de repetição (`for`, `while`, `do-while`)**: Executam repetidamente um bloco de código.
- **Controle de laços (`break` e `continue`)**: Controla a execução de laços, permitindo saídas antecipadas ou saltos de iterações.
- **Operador ternário**: Simplifica atribuições condicionais.
- **`assert`**: Verifica condições para detectar problemas durante o desenvolvimento.

1. Declaração Básica de Função

- A forma mais simples de uma função em Dart possui um tipo de retorno, um nome e parâmetros (que podem ser opcionais).

Sintaxe:
dart
Copy code

```
tipoDeRetorno nomeDaFuncao(parametros) {
  // Corpo da função
  return algumValor; // se o tipo de retorno não for
void
}
```

-

Exemplo:
dart
Copy code

```
int somar(int a, int b) {
  return a + b;
}
```

-

Uso: Esta função `somar` recebe dois parâmetros do tipo `int`, `a` e `b`, soma-os e retorna o resultado. Você pode chamá-la assim:
dart
Copy code
```
int resultado = somar(2, 3); // resultado será 5
```

-
- **Tipo de Retorno**: Você pode especificar um tipo de retorno (como `int`, `String` ou `void`). Se a função não retornar um valor, use `void`.

2. Funções Void

- Estas funções não retornam nenhum valor. Elas apenas executam uma ação.

Exemplo:
dart
Copy code
```
void saudar() {
  print('Olá, Mundo!');
}
```

-

Uso: A função `saudar` não recebe nenhum parâmetro e imprime uma mensagem quando chamada:
dart
Copy code
```
saudar(); // Saída: Olá, Mundo!
```

-

3. Parâmetros Opcionais Posicionais

- Os parâmetros podem ser tornados opcionais usando colchetes `[]`. Se não forem fornecidos, assumem um valor `null` por padrão.

Sintaxe:
dart
Copy code

```
void exibirMensagem(String mensagem, [int? numero]) {
  print(mensagem);
  if (numero != null) {
    print('Número: $numero');
  }
}
```

-

Exemplo de Uso:
dart
Copy code

```
exibirMensagem('Bem-vindo'); // Saída: Bem-vindo
exibirMensagem('Você ganhou', 10); // Saída: Você ganhou / Número: 10
```

-

4. Parâmetros Nomeados

- Parâmetros nomeados são úteis para melhorar a legibilidade do código e podem ser opcionais. São definidos usando chaves `{}`.

Sintaxe:

dart

Copy code

```
void criarUsuario({required String nome, int idade =
18}) {
  print('Nome: $nome, Idade: $idade');
}
```

-

Uso:

dart

Copy code

```
criarUsuario(nome: 'Carlos'); // Saída: Nome: Carlos,
Idade: 18
criarUsuario(nome: 'Ana', idade: 25); // Saída: Nome:
Ana, Idade: 25
```

-
- **`required`**: Indica que um parâmetro nomeado é obrigatório.

5. Funções Anônimas (Closures)

- Funções que não têm um nome e são frequentemente usadas em funções de ordem superior, como `forEach`, `map` e `filter`.

Exemplo:

dart

Copy code

```
List<int> numeros = [1, 2, 3];
numeros.forEach((numero) {
```

```
  print(numero * 2);
});
// Saída: 2, 4, 6
```

-

6. Funções de Uma Linha (Arrow Functions)

- Quando a função tem apenas uma expressão, você pode usar a "arrow syntax" (`=>`) para torná-la mais concisa.

Exemplo:
dart
Copy code

```
int multiplicar(int a, int b) => a * b;
print(multiplicar(2, 3)); // Saída: 6
```

-

7. Funções Assíncronas (`async` e `await`)

- Funções assíncronas permitem operações que demoram a ser concluídas sem bloquear o restante do código, como chamadas de API ou leituras de arquivos.

Sintaxe:
dart
Copy code

```
Future<void> buscarDados() async {
  print('Iniciando busca...');
  await Future.delayed(Duration(seconds: 2));
  print('Dados obtidos!');
```

```
}
```

-

Uso:
dart
Copy code
```
buscarDados(); // Saída: Iniciando busca... (após 2 segundos) Dados obtidos!
```

-
- `await` é usado para esperar que uma operação assíncrona seja concluída antes de continuar.

8. Funções de Ordem Superior

- São funções que podem receber outras funções como parâmetros ou retorná-las.

Exemplo:
dart
Copy code
```
void executarOperacao(int a, int b, Function operacao)
{
  print(operacao(a, b));
}

int soma(int x, int y) => x + y;
int multiplicacao(int x, int y) => x * y;

executarOperacao(2, 3, soma); // Saída: 5
```

```
executarOperacao(2, 3, multiplicacao); // Saída: 6
```

-

9. Funções com Retorno

- Funções podem retornar valores de qualquer tipo, como `int`, `String`, `List`, entre outros.

Exemplo:
dart
Copy code

```
double calcularAreaCirculo(double raio) {
  return 3.14 * raio * raio;
}

double area = calcularAreaCirculo(5);
print(area); // Saída: 78.5
```

-
- Aqui, a função `calcularAreaCirculo` retorna um `double` representando a área de um círculo.

10. Funções Recursivas

- Funções que chamam a si mesmas para resolver um problema.

Exemplo (Cálculo do Fatorial):
dart
Copy code

```
int fatorial(int n) {
  if (n <= 1) {
```

```
    return 1;
  } else {
    return n * fatorial(n - 1);
  }
}

print(fatorial(5)); // Saída: 120
```

-
- A função `fatorial` calcula o fatorial de um número, multiplicando-o por todos os números inteiros menores que ele até 1.

Resumo

- **Funções em Dart** são blocos de código que realizam uma tarefa específica e podem retornar um valor.
- **Funções com parâmetros nomeados e opcionais** aumentam a flexibilidade no fornecimento de argumentos.
- **Funções assíncronas** permitem lidar com operações demoradas sem bloquear a execução.
- **Closures e funções de ordem superior** são úteis para manipular listas e coleções.
- **Funções recursivas** são importantes para algoritmos que precisam se repetir até atingir uma condição base.