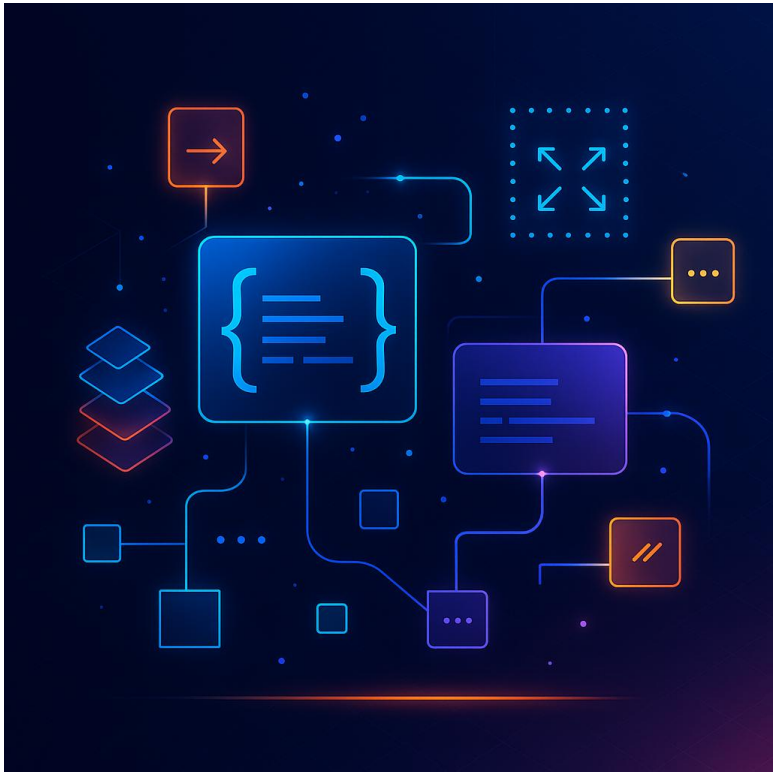
Recursos Modernos de Dart

Resumo dos elementos explorados



Construtores e Parâmetros

- Inicialização formal atribui argumentos diretamente às propriedades.
- Parâmetros nomeados podem ser opcionais e ter valores padrão.
- Use 'required' para forçar parâmetros obrigatórios.

```
class Personagem {

final String nome;

final int nivel;

final int? forca;

Personagem(this.nome, { this.nivel = 1, this.forca });
}
final p = Personagem('Aria', nivel: 3);
```

[1] [2]

Null Safety e Operadores

- Tipos nulos são declarados com '?' (ex.: int?).
- '??' fornece um valor padrão quando o operando é null.
- '??=' atribui valor somente se a variável for null.
- '?.' acessa propriedades de forma segura sem lançar exceção.

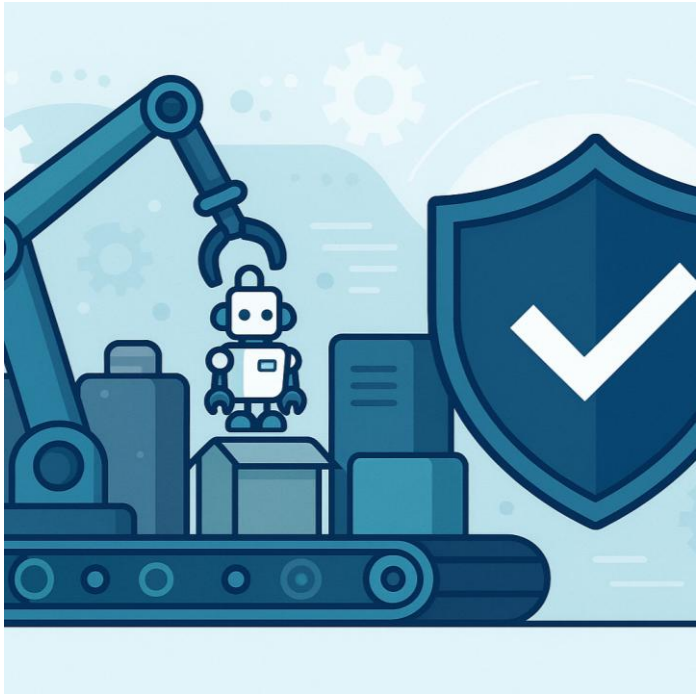
```
int? forca;
int
poder(int nivel) => (forca ?? 10) * nivel;
void init() { forca ??= 5; }
String? apelido;
String descricao(String nome) => 'Olá, ${apelido ?? nome}';
```

[3] [4]

Cascata e Coleções

- Operador '..' encadeia múltiplas chamadas no mesmo objeto.
- Coleções suportam 'if' e 'for' para construir listas dinamicamente.
- '...' insere todos os elementos de uma lista em outra; '...?': ignora se null.

```
var list = [1, 2, 3];
var nova = [0, ...list, if (promoAtivo) 42];
var extra = personagem..marcarConcluida()..descricao ??= 'Treino';
List<int>? talvez;
var outra = [0, ...?talvez];
```



[5] [6]

Factory e Imutabilidade

- Construtores 'factory' podem retornar instâncias existentes ou customizadas.
- Útil para inicializar objetos a partir de mapas ou JSON.
- 'UnmodifiableListView' fornece uma visão somente leitura de uma lista.

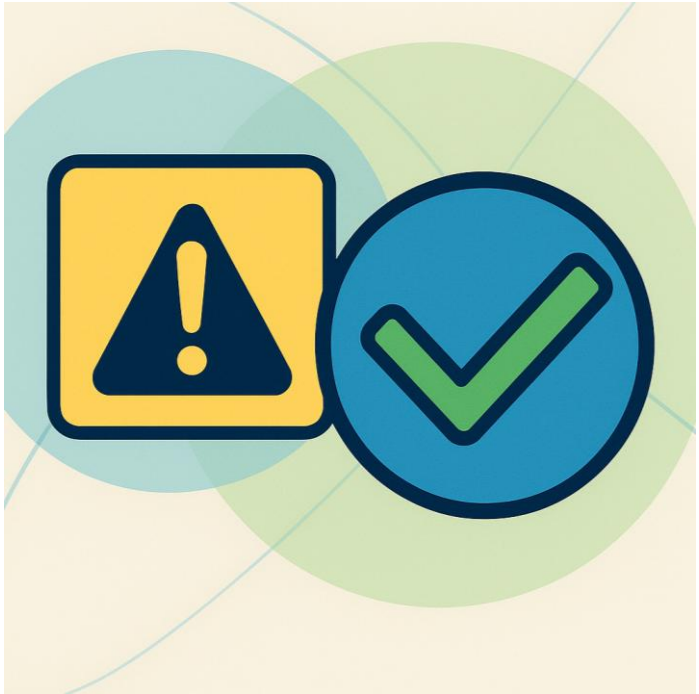
```
class Missao {

factory Missao.fromMap(Map m) =>

Missao(id: m['id'], titulo: m['titulo'], concluida: m['concluida'] ?? false);
}
class Carrinho {

final List<Item> _itens;

UnmodifiableListView<Item> get itens => UnmodifiableListView(_itens);
}
```

[7] [8]

Asserts e Validação

- 'assert' verifica condições em desenvolvimento e é ignorado em produção.
- Aceita apenas expressões booleanas; use comparações explícitas.
- Ideal para validar argumentos em construtores ou funções.

```
class Item {

Item({required double preco, int qtd = 1})

: assert(preco >= 0, 'preço negativo'),

assert(qtd > 0, 'qtd deve ser > 0');
}
```

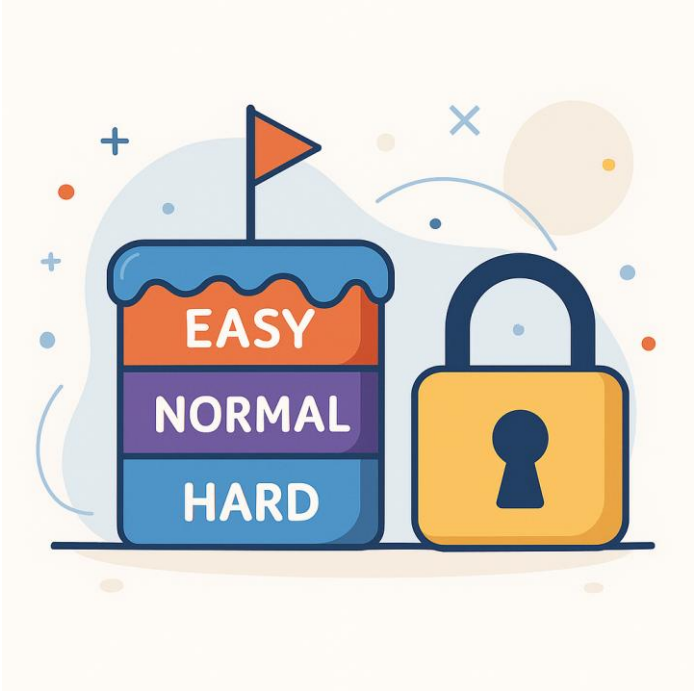


[9] [10]

Enums & Classes Seladas

- Enums podem ter campos, métodos e construtores const.
- Classes 'sealed' definem um conjunto fechado de subtipos e permitem switches exaustivos.
- Pattern matching extrai propriedades diretamente no 'switch'

```
enum Dificuldade {

facil(0.8, '😊'), normal(1.0, '⚔️'), dificil(1.2, '💀');

const Dificuldade(this.mult, this.emoji);

String info() => '$mult ${emoji}';
}
sealed
class Acao {}
class Atacar extends Acao { const Atacar(this.danoBase); final int
danoBase; }
switch (acao) {

case Atacar(:final danoBase): /*...*/

case Defender(): /*...*/
}
```



[11] [12] [13]

Mixins e Typedefs

- Mixins compartilham métodos e propriedades usando 'with'.
- Defina mixins com 'mixin' e reutilize em várias classes.
- 'typedef' cria apelidos para tipos complexos ou funções.

```
mixin LogMixin {

void log(String msg) => print('[LOG] ${msg}');
}
typedef ModDano = int Function(int base, Dificuldade diff);
class Simulador with LogMixin {

final ModDano mod;

Simulador(this.mod);
}
```

[14] [15] [16]

Records, Extension Types e Callable

- Records agrupam valores heterogêneos com campos posicionais ou nomeados.
- Pattern matching facilita a desestruturação de records.
- Extension types envolvem tipos primitivos e controlam acessos.
- Classes chamáveis implementam 'call', agindo como funções.

```
final jogadas = <(String, int, String?)>[

('alice', 40, 'normal'),

('bob', 50, null),
];
extension type PlayerId(String value) {
  bool
get isValido => value.length >= 3;
}
class BonusCalc {
  int
call({required int base, String? tipo, required int streak}) => base +
streak;
}
```

[17] [18] [19]

Formatação de Strings

- Interpolação insere variáveis ou expressões em strings com '$' ou '\${...}'.
- Strings podem ser concatenadas com '+' ou literais adjacentes.
- Use ''' ou """ para textos multi-linha.
- 'r' antes da string cria uma string bruta (sem escapes).
- Métodos como toStringAsFixed() formatam números para impressão.

```
var s = 'Dart';
print('Dart tem $s, que é muito útil.');
print('Isso merece caps: ${s.toUpperCase()}!');
var pi = 3.14159;
print('PI com 2 casas: ${pi.toStringAsFixed(2)}');
var texto = '''
Texto
multi-linha
'''
;
```

[20] [21]