DCC917A – TÓPICOS ESPECIAIS III: DESENVOLVIMENTO DE APLICATIVOS MÓVEIS

AULA 04

Carlos Bruno Oliveira Lopes
Engenheiro de Computação
Mestre em Ciência da Computação

Linguagem de programação Dart

Mixins

- É uma forma de reutilizar código de classes em múltiplas hierarquias de classes.
- Você pode “misturar ou reutilizar os códigos de duas formas” usando a palavra chave `with`, ou `mixin` e `with` juntas
- Uma classe mixin pode ser definida pela sintaxe:

```
mixin identifier {}
```

Linguagem de programação Dart

Mixins

Ex.:

```
class Morder {
  void morde() {
    print("anh anh");
  }
}

class Nadar {
  void nada() {
    print("split splot
plash ploshe");
  }
}
```

```
class Cachorro with Nadar,
Morder {
  void somDeCachorro() {
    nada();
    morde();
  }
}

void main() {
  var dog = Cachorro();
  dog.somDeCachorro();
  dog.morde();
  dog.nada();
}
```

Linguagem de programação Dart

Mixins (Exemplo)

```
class Person {
  String name = 'unknown';
  Person(this.name);
}

mixin Avenger {
  bool wieldsMjolnir = false;
  bool hasArmor = false;
  bool canShrink = true;

  void whichAvenger() {
    if (wieldsMjolnir) {
      print("I'm Thor");
    } else if (hasArmor) {
      print("I'm Iron Man");
    } else {
      print("I'm Ant Man");
    }
  }
}
```

```
class Superhero extends Person with Avenger {
  Superhero(wieldsMjolnir, hasArmor, canShrink,
        String name) : super(name) {
    this.wieldsMjolnir = wieldsMjolnir;
    this.hasArmor = hasArmor;
    this.canShrink = canShrink;
  }

  String get name => "${super.name}";
}

void main() {
  Superhero s = new Superhero(true, false, false, "Thunder");
  s.whichAvenger();
  print (s.name);
}
```

Linguagem de programação Dart

Visibilidade

- Em Dart tudo é publico a não ser que comece com um underscore, que marca o elemento como sendo privado de biblioteca, ou classe.

Linguagem de programação Dart

Funções

- Em Dart funções são objetos e tem um tipo, Function.
- Elas podem ser atribuídas a variáveis ou podem ser passadas com argumentos de outras funções;
- Podemos omitir o tipo de uma função;

Linguagem de programação Dart

Funções (Exemplos)

```
int fibonacci(int n) {
  if (n == 0 || n == 1) return n;
  return fibonacci(n - 1) + fibonacci(n -
2);
}

void main() {
  var result = fibonacci(5);
  print(result);
}
```

Linguagem de programação Dart

Funções

- A notação => são usadas para funções com uma única expressão.
  - Sintaxe: `name => expr` [abreviação para `{return expr;}`]

  Ex.:

```
int soma2num (x,y) => x + y;

void main() {
  var result = soma2num(5,3);
  print(result);
}
```

Linguagem de programação Dart

Funções (Parâmetros)

- Uma função pode ter os seguintes parâmetros “especiais”:
  - **Nomeados**.
  - **Posicionais opcionais**.
  - **Com valores padrão**.

Linguagem de programação Dart

Funções (Parâmetros)

Nomeados.

- Eles são opcionais, a menos que sejam especificamente marcados como `required` (obrigatórios);
- Eles podem ser definidas como usando a sintaxe `{param1, param2, …}` para especificar os parâmetros;
- Quando a função é invocada os parâmetros conseguem ser especificados usando a sintaxe `paramName: value`;
- Para definir que um parâmetro é obrigatório podemos usar a sintaxe `required` que indica que tal parâmetro deve ser fornecido pelo usuário;

Linguagem de programação Dart

Funções (Parâmetros)

Nomeados. (exemplos)

```
num soma2num({required num x, num y = 0}) {
  if (y == 0) {
    return x;
  } else {
    return x + y;
  }
}

void main() {
  var res0 = soma2num(x: 5, y: 3);
  var res1 = soma2num(x: -5);
  print("res0 = $res0; res1 = $res1");
}
```

Linguagem de programação Dart

Funções (Parâmetros)

Posicionais opcionais.

– Em uma função com um conjunto de parâmetros, podemos indicar quais parâmetros são opcionais usando o `[]`.

Ex.:

```
String say(String from, String msg, [String? device]) {
  var result = '$from says $msg';
  if (device != null) {
    result = '$result with a $device';
  }
  return result;
}
```

Linguagem de programação Dart

Funções (Parâmetros)

Com valores padrão.

– Pode definir valores padrões de parâmetros usando o operador de atribuição =.

Ex.:

```
num soma2num({num x = 0, num y = 0}) => x + y;

void main() {
  var res0 = soma2num(x: 2, y: 3);
  var res1 = soma2num();
  print(res0);
  print(res1);
}
```

Linguagem de programação Dart

Funções (Como objetos)

- Função são objetos da classe `Function`.
- Dessa forma podemos passar uma função como parâmetro de outra;

Ex.:

```
void printElement(int element) {
  print(element);
}

void main() {
  var list = [1, 2, 3];
  // Pass printElement as a parameter.
  list.forEach(printElement);
}
```

Linguagem de programação Dart

Funções (Como objetos)

Ex.:

```
void greet(String name) {
  print("Hello, $name");
}

class MyClass {
  void greetAgain({required Function f, String n = "human"}) {
    f(n);
  }
}

void main() {
  MyClass mc = new MyClass();
  greet("Frank");
  mc.greetAgain(f: greet, n: "Traci");
  mc.greetAgain(f: greet);
}
```

Linguagem de programação Dart

Funções (Anônimas)

- É uma função sem nome conhecida como anônima ou lambda;
- Corpo da função (sintaxe):

```
(Type param1, …) {
  codeBlock;
};
```

Linguagem de programação Dart

Funções (Anônimas)

Ex.:

```
void main() {
  const list = ['apples', 'bananas', 'oranges'];
  list.forEach((item) {
    print('${list.indexOf(item)}: $item');
  });
}
```

Linguagem de programação Dart

Funções Assíncronas

- São aquelas que retornam após a configuração de uma operação possivelmente demorada, sem esperar que a operação seja concluída;
- Ou seja, elas são retornados antes de a operação terminar, permitindo que o programa espere o resultado enquanto faz outras coisas, e então continue de onde parou quando o resultado for fornecido.
- Geralmente elas são aplicadas em funções que retornam objetos do tipo:
  - **Future**. Usado para representar um valor potencial, ou erro, que estará disponível em algum momento no futuro.
  - **Stream**. Fornece uma maneira de receber uma sequência de eventos. Cada evento é um evento de dados, chamado de elemento do fluxo, ou um evento de erro, que é uma notificação de que algo falhou.
- As palavras chaves `async` e `await` oferecem suporte a programação assíncrona permitindo que se possa escrever códigos assíncronos que se parecem com síncronos.

Linguagem de programação Dart

Funções Assíncronas

Manipulação de Futuros

- Quando há a necessidade de um “Futuro concluído”, há duas opções:
  - Uso do `async` e `await`.
  - Uso da API Future.
- Exemplo de código que usa o `await` para esperar o resultado de uma função assíncrona:

```
await lookUpVersion();
```

- Para o uso do `await`, a função deve ser assíncrona. Isso significa, que a função deve ser marcada com `async`:

```
Future<void> checkVersion() async {
  var version = await lookUpVersion();
  // Do something with version
}
```

Linguagem de programação Dart

Funções Assíncronas

Manipulação de Futuros

– Pode se usar `try`, `catch` and `finally` para manipular erros e limpezas no código que use `await`:

```
try {
  version = await lookUpVersion();
} catch (e) {
  // React to inability to look up the version
}
```

Linguagem de programação Dart

Funções Assíncronas

Manipulação de Futuros

- Na sintaxe `await expression`, o valor da `expression` é geralmente um `Future`; se não for, o valor é automaticamente empacotado em um Future.
  - O objeto `Future` indica uma promessa de retorno de um objeto.
    - O `await expression` pausa a execução até que o objeto esteja disponível;

Linguagem de programação Dart

Funções Assíncronas (Declaração)

- Uma função `async` é uma função cujo o corpo é marcado com o modificador `async`.
  - Ao adicionar a palavra-chave `async` a função, isso faz com que ela retorne um objeto `Future`.

    Ex.: `Future<String> lookUpVersion() async => '1.0.0';`
    - » Observe que o corpo não necessita que seja usado a API Future.
    - » O Dart criará o objeto Future se necessário.
- Para executar um função `async` no função principal o corpo dela deve ser `async`.

Linguagem de programação Dart

Funções Assíncronas (Declaração)

Ex.:

```
Future fazAlgo() async {

  Future.delayed(const Duration(seconds: 5));
  print("Exemplos de função assincrona!");
}

void main() async {
  fazAlgo();
}
```

Linguagem de programação Dart

Funções Assíncronas

Manipulação de Streams

- Quando há necessidade de receber valores de um Stream (fluxo), há duas opções:
  - Uso do `async` e um repetição assincrona usando for (`await for`).
  - Uso da API Stream.
- Sintaxe de assincronismo para repetição:

```
await for (varOrType identifier in expression) {
  // Executes each time the stream emits a value.
}
```

  - O valor da `expression` deve ser um Stream.
  - A execução prossegue da seguinte forma:
    1. Espera até o stream emitir um valor;
    2. Executa o corpo do *loop*, com a variável definida para aquele valor emitido.
    3. Repita 1 e 2 até que o stream seja fechado.
    - Para parar a escuta no stream, pode se usar uma instrução de `break` ou `return`, que sai do loop for e cancela a inscrição do fluxo

Linguagem de programação Dart

Funções Assíncronas

Manipulação de Streams

Ex.:

```
Future<void> main() async {
  // ...
  await for (var request in requestServer) {
    handleRequest(request);
  }
  // ...
}
```

Linguagem de programação Dart

Bibliotecas

- Uma biblioteca fornece uma API externa para outros códigos que desejam usa-la
- Ela serve como um método de isolamento para que qualquer identificador de uma biblioteca que comece com um caractere underscore
  - A mesma só será visto dentro dessa biblioteca.
- Todo aplicativo Dart é automaticamente uma biblioteca
- As bibliotecas podem ser empacotadas e distribuídas para outras pessoas com o uso da ferramenta `pub` do Dart SDK (gerenciador de pacotes e assets)

Linguagem de programação Dart

Bibliotecas

- Para usarmos uma biblioteca, temos de empregar a palavra-chave `import`:

```
import "dart:html";
```

- Se a biblioteca importada vier de um pacote fazemos:

```
import "package:someLib.dart";
```

- Se biblioteca fizer parte de seu código, ou se for uma copia do codebase, a URI será um caminho relativo do sistema de arquivos:

```
import "../libs/myLibrary.dart";
```

- Se o importe de duas bibliotecas for conflituoso podemos usar a palavra-chave `as` para renomear a biblioteca:

```
import "libs/lib1.dart";
import "libs/lib2.dart" as lib2;
```

- Podemos importar parte dos recursos da biblioteca usando o comando `show` e `hide`:

```
import "package:lib1.dart" show Account;
import "package:lib2.dart" hide Account;
```

Linguagem de programação Dart

Manipulação de exceções

- Exceções são erros que indicam que algo inesperado aconteceu
- Manipular exceções no Dart é semelhante a como é feito em Java ou JavaScript
- Todas as exceções de Dart são exceções não verificadas. Portanto,
  - os métodos não declaram quais exceções eles podem lançar;
  - e não o obrigada a capturar nenhuma exceção.
- O Dart fornece tipos de exceção e erro, bem como vários subtipos predefinidos
- Ele permite o desenvolvedor definir suas próprias exceções
- E permite o desenvolvedor lançar qualquer coisa como exceção.

Linguagem de programação Dart

Manipulação de exceções

Throw

Ex.:

```
throw FormatException('Expected at least 1 section’);
throw 'Out of llamas!’;
```

– Uma exceção é uma expressão que pode ser aplicada em instruções usando =>:

```
void distanceTo(Point other) => throw UnimplementedError();
```

Linguagem de programação Dart

Manipulação de exceções

Catch

- Capturar uma exceção interrompe a propagação da exceção.
- A captura de uma exceção permite que o desenvolvedor possa lidar com ela:

```
try {
  breedMoreLlamas();
} on OutOfLlamasException {
  buyMoreLlamas();
}
```

Linguagem de programação Dart

Manipulação de exceções

Catch

– Para lidar com um código que pode lançar mais de um tipo de exceção, pode-se especificar várias cláusulas catch:

```
try {
  breedMoreLlamas();
} on OutOfLlamasException {
  // A specific exception
  buyMoreLlamas();
} on Exception catch (e) {
  // Anything else that is an exception
  print('Unknown exception: $e');
} catch (e) {
  // No specified type, handles all
  print('Something really unknown: $e');
}
```

Como mostra o código atual e anterior, pode-se usar **on** ou **catch** ou ambos.

- Use `on` quando precisar especificar o tipo de exceção.
- Use `catch` quando seu manipulador de exceção precisar do objeto de exceção.

Linguagem de programação Dart

Manipulação de exceções

Catch

- Para garantir que algum código seja executado, independentemente de uma exceção ser lançada ou não, use uma cláusula `finally`.
  - Se nenhuma cláusula catch corresponder à exceção, a exceção será propagada após a execução da cláusula finally:

```
try {
  breedMoreLlamas();
} finally {
  // Always clean up, even if an exception is thrown.
  cleanLlamaStalls();
}
```

Linguagem de programação Dart

Manipulação de exceções

Catch

– A cláusula finally é executada após qualquer cláusula catch correspondente:

```
try {
  breedMoreLlamas();
} catch (e) {
  print('Error: $e'); // Handle the exception first.
} finally {
  cleanLlamaStalls(); // Then clean up.
}
```

Linguagem de programação Dart

Geradores

- Quando há a necessidade de produzir vagarosamente uma sequência de valores, podemos usar um função geradora.
- O Dart possui suporte a dois tipos de funções geradoras:
  - Gerador síncrono: que retorna um objeto Iterable.
  - Gerador assíncrono: que retorna um objeto Stream.

Linguagem de programação Dart

Geradores

- Para implementar uma **função geradora síncrona**, marque o corpo da função como `sync*`, e use a instrução `yield` para entregar o valor:

```
Iterable<int> naturalsTo(int n) sync* {
  int k = 0;
  while (k < n) yield k++;
}
```

```
main() {
  Iterable it = naturalsTo(5);
  Iterator i = it.iterator;
  while (i.moveNext()){
    print(i.current);
  }
}
```

Linguagem de programação Dart

```
Iterable<int> naturalsTo(int n) sync* {
  int k = 0;
  while (k < n) yield k++;
}

void main() {
  Iterable it = naturalsTo(5);
  Iterator i = it.iterator;
  while (i.moveNext()){
    print(i.current);
  }
}
```

Geradores

- `sync*` indica que é uma função geradora;
- `yield` retorna o valor dentro do gerador;
- `naturalsTo()` retorna um objeto iterável;
  - O código pode então extrair um iterador para começar a percorrer a lista de resultados;
- `naturalsTo()` não é executada até o código que a está chamando extrair o iterador e chamar `moveNext()`;
  - Quando isso ocorre, `naturalsTo()` é executada até chegar à instrução `yield`, onde, a expressão `i++` é avaliada e retornada.
    - Em seguida, `naturalsTo()` é suspensa e `moveNext()` retorna `true` para seu chamador.
    - A função `naturalsTo()` voltará a ser executada na próxima vez que `moveNext()` for chamada.

Linguagem de programação Dart

Geradores

- Para implementar uma **função geradora assíncrona**, marque o corpo da função como `async*`, e use a instrução `yield` para entregar o valor:

```
Stream<int> asynchronousNaturalsTo(int n) async* {
  int k = 0;
  while (k < n) yield k++;
}
```

```
void main() async {
  Stream s = asynchronousNaturalsTo(5);
  await for (int i in s) {
    print(i);
  }
}
```

Linguagem de programação Dart

```
Stream<int> asynchronousNaturalsTo(int n) async* {
  int k = 0;
  while (k < n) yield k++;
}

void main() async {
  Stream s = asynchronousNaturalsTo(5);
  await for (int i in s) {
    print(i);
  }
}
```

Geradores

- `async*` indica que é uma função geradora;
- `yield` retorna o valor dentro do gerador;
- `asynchronousNaturalsTo()` retorna um `Stream`;
- `await for` é loop usado em operações assíncronas. Ele espera que a função `asynchronousNaturalsTo()` "empurre" o valor por intermédio do objeto Stream retornado.

Linguagem de programação Dart

Geradores

– Se o gerador é recursivo, pode-se melhorar a execução usando `yield*`:

```
Iterable<int> naturalsDownFrom(int n) sync* {
  if (n > 0) {
    yield n;
    yield* naturalsDownFrom(n-1);
  }
}
```

Linguagem de programação Dart

Metadados

- O Dart dá suporte ao conceito de metadados embutidos no código.
- Eles são usado para dá informação adicional ao código.
- Três anotações são disponíveis para todo os códigos em Dart:
  - `@deprecated`, usado para indicar que provavelmente aquele recurso não será mais usado e que em algum momento esse elemento pode ser removido da biblioteca.
  - `@Deprecated`, o mesmo que `@deprecated`, mas com uma informação a mais para indicar que mudança deve ser feito.
  - `@override`, usada para indicar que uma classe está intencionalmente sobrepondo um membro de sua superclasse.

Linguagem de programação Dart

Metadados

Exemplos de `@Deprecated` :

```
class Television {
  /// Use [turnOn] to turn the power on instead.
  @Deprecated('Use turnOn instead')
  void activate() {
    turnOn();
  }

  /// Turns the TV's power on.
  void turnOn() {...}
}
```

Linguagem de programação Dart

Metadados

- É possível criar as próprias anotações. Elas são apenas classes:

```
class MyAnnotation {
  final String note;
  const MyAnnotation(this.note);
}

@MyAnnotation("This is my function")
void myFunction() {
  print("Faz algo");
}

void main () {
  myFunction();
}
```

Linguagem de programação Dart

Genéricos

- São usados para permitir pluralidade de tipagem na codificação de uma classe, método ou variável;
  - ou seja, ele permite a construção de códigos que permitam informa o tipo da estrutura de dados no momento em que ela for utilizada de acordo com necessidade de tipo do desenvolvedor.
- Resumidamente, eles são usados para informar o tipo de algo no momento oportuno:

```
var ls = List<String>();
```

  - O Dart saberá que lista `ls` só pode conter String.

Linguagem de programação Dart

Genéricos

- Vantagens:
  - A especificação adequada de tipos genéricos resulta em um melhor código gerado.
  - Usar genéricos reduz a duplicação de código.

Linguagem de programação Dart

Genéricos

- Se a necessidade de uma lista conter apenas String, basta declara como `List<String>`:

```
var names = <String>[];
names.addAll(['Seth', 'Kathy', 'Lars']);
names.add(42); // Error
```

Linguagem de programação Dart

Genéricos

– Se deseja reduzir a duplicidade:

```
abstract class ObjectCache {
  Object getByKey(String key);
  void setByKey(String key, Object value);
}

abstract class StringCache {
  String getByKey(String key);
  void setByKey(String key, String value);
}
```

```
abstract class Cache<T> {
  T getByKey(String key);
  void setByKey(String key, T value);
}
```

Linguagem de programação Dart

Genéricos

- Usando como literal (`List`, `Set` e `Map`):

```
var names = <String>['Seth', 'Kathy', 'Lars'];
var uniqueNames = <String>{'Seth', 'Kathy', 'Lars'};
var pages = <String, String>{
  'index.html': 'Homepage',
  'robots.txt': 'Hints for web robots',
  'humans.txt': 'We are people, not machines'
};
```

Linguagem de programação Dart

Genéricos

- Usando tipo parametrizado com construtor:

```
var nameSet = Set<String>.from(names);
var views = Map<int, View>();
```

- Coleção genérica e os tipos que elas contêm:

```
var names = <String>[];
names.addAll(['Seth', 'Kathy', 'Lars']);
print(names is List<String>); // true
```

Linguagem de programação Dart

Genéricos

- Usando métodos genéricos:

```
T first<T>(List<T> ts) {
  // Do some initial work or error checking, then...
  T tmp = ts[0];
  // Do some additional checking or processing...
  return tmp;
}
```