</ Dart programming Languages />

Part 2

constructor :Dr.Vahidi
Presenter :Armita Kamari

Fall 1403_1404

1 0 1 1 0 1 1 0 1 1 0 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 1 0 1 1 0 1 1 1 1 1 0 1

</Topics

Collections	Lists , sets , maps
Error Handling in Dart	Try , catch , on typeexception catch , finally
Working with Files in Dart	Reading , writing , appending , delete
Concurrency in Dart	Await , async , future

1 0 1 1 0 1 1 0 1 0 1 1 0 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 1 0 1 1 0 1 1 1 0 1 1 0 1

</ collections

{01}
Lists

{02}
sets

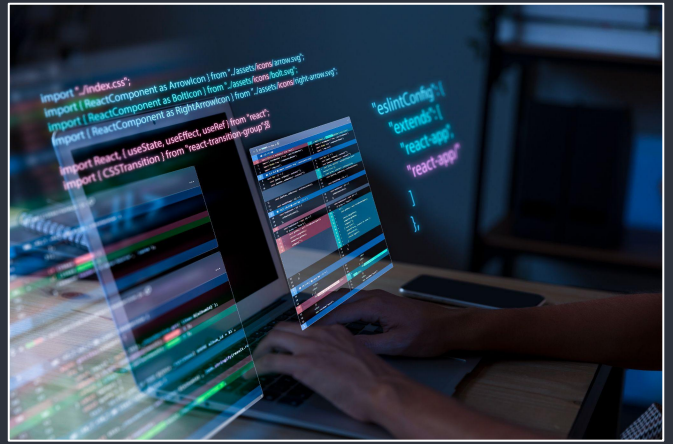
{03}
maps

1011 011 01 1011001 10 11011 011 01 110110 110111 1101

</ what are the collections ? />

In Dart, **collections** are a type of data structure that allow you to group and manage multiple related elements

dart>



1 0 1 1 0 1 1 0 1 1 0 1 1 0 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 1 0 1 1 0 1 1 0 1 1 1 1 1 0 1

Collections

list

A **List** is an ordered collection of elements, where each element can be accessed by its index. Lists can contain duplicate elements and can be either fixed-length or growable.

Set

A **Set** is an unordered collection of unique elements, meaning it does not allow duplicate values. Sets are used when the order doesn't matter, and uniqueness is essential.

Map

A **Map** is a collection of key-value pairs where each key is unique, and it maps to exactly one value. This is useful for storing pairs of data, like a dictionary or database.

1 0 1 1 0 1 1 0 1 1 0 1 1 0 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 1 0 1 1 0 1 1 1 1 1 0 1

</ Collections and Methods />

Lists

- add(element): Adds an element to the list.
- remove(element): Removes the first occurrence of the element.

Maps

- putIfAbsent(key, value): Adds a key-value pair if the key is not present.
- remove(key): Removes the key-value pair.
- containsKey(key): Checks if a key exists.

Sets

- add(element): Adds an element to the set.
- contains(element): Checks if an element exists in the set.
- remove(element): Removes an element from the set.

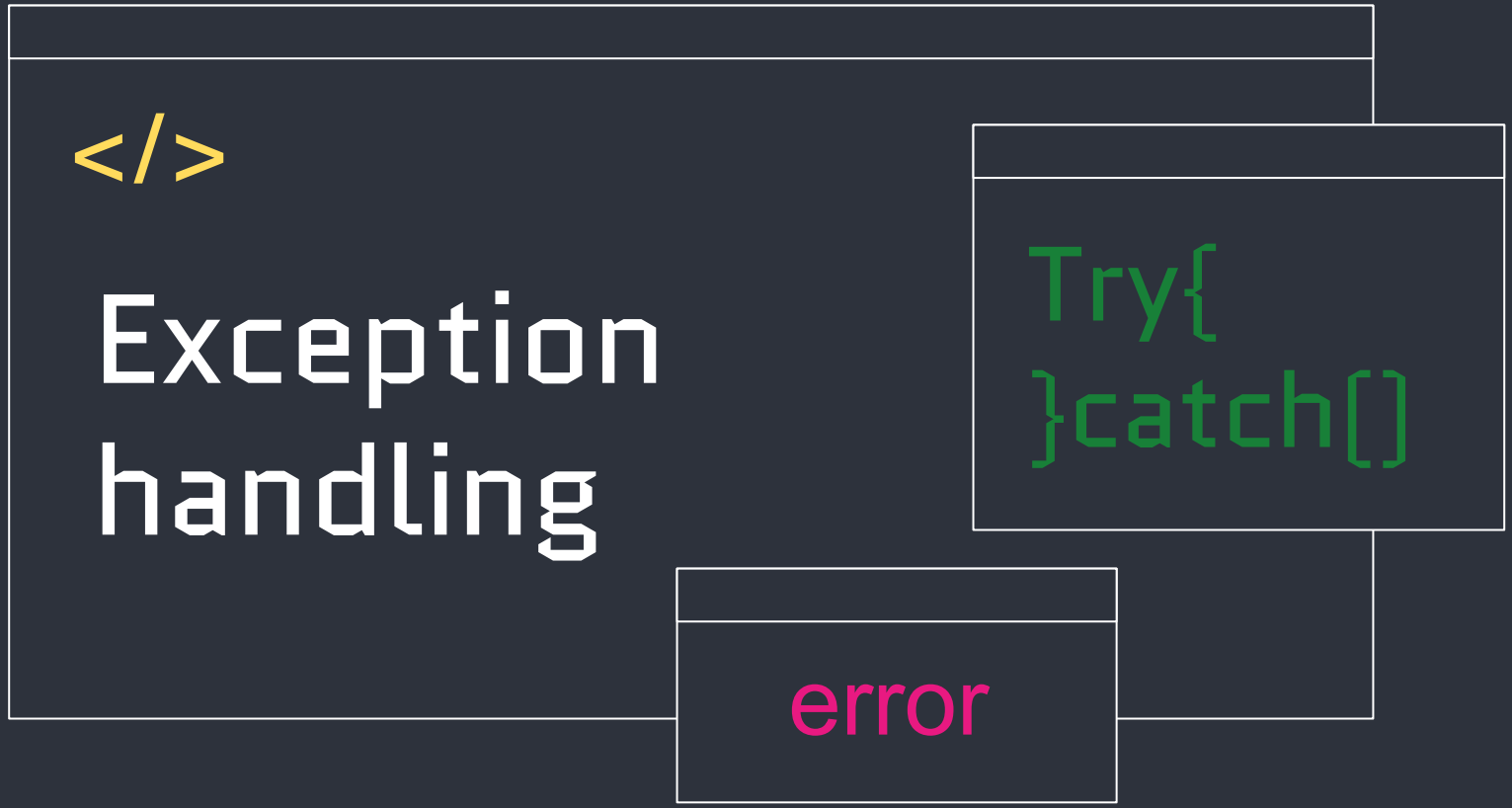
} /> [

1 0 1 1 0 1 1 0 1 1 0 1 1 0 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 1 0 1 1 0 1 1 0 1 1 1 1 1 0 1

</Code/>

```
// Lists
List<String> fruits = ["Apple", "Banana", "Cherry"];
fruits.add("Date"); // Adding an item
print(fruits); // Output: [Apple, Banana, Cherry, Date]
// Maps
Map<String, int> ages = {
  "Alice": 30,
  "Bob": 25,
  "Charlie": 28,
};
ages["Dave"] = 22; // Adding a key-value pair
print(ages); // Output: {Alice: 30, Bob: 25, Charlie: 28, Dave: 22}
// Sets
Set<String> uniqueFruits = {"Apple", "Banana", "Apple"};
uniqueFruits.add("Cherry"); // Adding an item
print(uniqueFruits); // Output: {Apple, Banana, Cherry}
```

1 0 1 1 0 1 1 0 1 1 0 1 1 0 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 1 0 1 1 0 1 1 0 1 1 1 1 1 0 1



1 0 1 1 0 1 1 0 1 0 1 1 0 0 1 1 0 1 1 0 1 1 0 1 1 1 0 1 1 0 1 1 1 0 1 1 0 1

</ Exception handling

. try-catch Block

A **`try-catch`** block is used to handle exceptions. Code that might throw an exception is placed inside the `try` block, and any exceptions that occur can be caught in the `catch` block.

Finally Block

The **`finally`** block is always executed, whether an exception occurs or not. It's used for cleanup tasks, like closing a file or releasing resources.

. on-Catch Block

The **on** clause can be used to catch a specific type of exception. For example, you can catch an `IntegerDivisionByZeroException` if you want to handle only that specific case.

Multiple Catch Blocks

You can also have multiple `catch` blocks if you want to handle different exceptions in different ways

1 0 1 1 0 1 1 0 1 1 0 1 1 0 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 1 0 1 1 0 1 1 0 1 1 1 1 1 0 1

</ Exception handling

Throwing Exceptions

In Dart, you can also throw your own exceptions using the `throw` keyword.

An exception is an error that occurs during the execution of a program. For example, dividing a number by zero or accessing a non-existent file can cause exceptions. In Dart, exceptions are thrown and can be caught to handle errors.

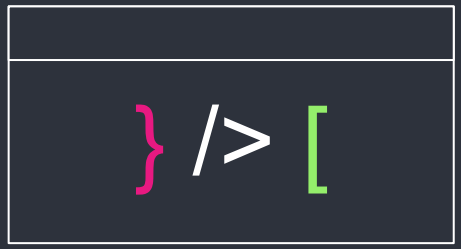
1 0 1 1 0 1 1 0 1 1 0 1 1 0 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 1 0 1 1 0 1 1 0 1 1 1 1 1 0 1

</ Code />

```
try {
  int result = 10 ~/ 0; // Division by zero
  print(result);
} on IntegerDivisionByZeroException catch (e) {
  print("Caught an integer division by zero: $e"); // on-catch block
} catch (e) {
  print("Caught an exception: $e"); // General catch block
} finally {
  print("This will always run."); // Finally block
}
```

1 0 1 1 0 1 1 0 1 1 0 1 1 0 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 1 0 1 1 0 1 1 0 1 1 1 1 1 0 1

File Handling in Dart



Working with files in Dart is done using the dart:io library. Key operations include reading, writing, appending, and deleting files

/>

1 0 1 1 0 1 1 0 1 1 0 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 1 0 1 1 0 1 1 1 0 1

Reading Files:

Uses `readAsString` to read file contents.
Asynchronous operation with `await`.

Writing to Files:

Uses `writeAsString` to write content to a file.
Asynchronous operation with `await`.

Appending to Files:

Uses `writeAsString` with `FileMode.append` to add text to the end of a file.
Asynchronous operation with `await`.

Deleting Files:

Uses `delete` to remove a file.
Asynchronous operation with `await`.

1 0 1 1 0 1 1 0 1 1 0 1 1 0 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 1 0 1 1 0 1 1 0 1 1 1 1 1 0 1

</ Code />

```dart
import 'dart:io';
void main() async {
  // Writing to a file
  File file = File('example.txt');
  await file.writeAsString('Hello, Dart!'); // Write text to file

  // Appending to a file
  await file.writeAsString('\nWelcome to file handling!', mode: FileMode.append);

  // Reading from a file
  String contents = await file.readAsString();
  print('File contents:\n$contents'); // Output: Hello, Dart! Welcome to file handling!

  // Deleting the file
  await file.delete();
  print('File deleted: ${file.path}'); // Confirm deletion
}
```

1 0 1 1 0 1 1 0 1 1 0 1 1 0 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 1 0 1 1 0 1 1 0 1 1 1 1 1 0 1

Concurrency in Dart

</

/>

async

await

Concurrency in Dart allows for performing multiple operations simultaneously. Key concepts include Future, async, and await.

1 0 1 1 0 1 1 0 1 1 0 1 1 0 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 1 0 1 1 0 1 1 1 1 1 0 1

Concurrency

future

A `Future` represents a value that will be available at some point in the future..

Explanation:

- The `Future` is used for operations that are performed asynchronously.

Using async and await

`async` keyword marks a function as asynchronous.

`await` keyword pauses execution until the `Future` completes.

Explanation:

- The `main` function is marked as `async`.
- `await` waits for `fetchData` to complete before printing the result.

1 0 1 1 0 1 1 0 1 1 0 1 1 0 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 1 0 1 1 0 1 1 0 1 1 1 1 1 0 1

Concurrency

Error Handling with Futures

Handle errors in asynchronous operations using try, catch, and finally

try block executes the code.

catch block handles any exceptions.

finally block runs regardless of success or failure

Running Multiple Futures Concurrently

Use Future.wait to run multiple Futures concurrently and wait for all of them to complete.

Future.wait waits for both future1 and future2 to complete.

Returns a list of results.

1 0 1 1 0 1 1 0 1 1 0 1 1 0 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 1 0 1 1 0 1 1 0 1 1 1 1 1 0 1

Concurrency

Isolates for Parallel Execution

Isolates allow running Dart code in parallel with its own memory and event loop

`Isolate.spawn` creates a new isolate.

`ReceivePort` receives messages from the isolate

1 0 1 1 0 1 1 0 1 1 0 1 1 0 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 1 0 1 1 0 1 1 0 1 1 1 1 1 0 1

</ Code />

```dart
import 'dart:async';
Future<String> fetchData() async {
  await Future.delayed(Duration(seconds: 2));
  return "Data fetched";
}

Future<void> main() async {
  try {
    String data = await fetchData(); // Using async and await
    print(data); // Output: Data fetched

    // Running multiple futures concurrently
    var results = await Future.wait([fetchData(), fetchData()]);
    print(results); // Output: [Data fetched, Data fetched]
  } catch (e) {
    print("Error: $e"); // Error handling
  }
}
```

1 0 1 1 0 1 1 0 1 1 0 1 1 0 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 1 0 1 1 0 1 1 0 1 1 1 1 1 0 1

</ Resources

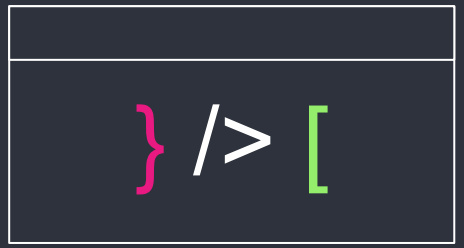
Did you like the resources in this template?
Get them for free at our other websites:

Vectors

- Gradient artificial intelligence youtube thumbnail

Photos

- Programming background with person working with codes on computer



/>

1 0 1 1 0 1 1 0 1 1 0 1 1 0 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 1 0 1 1 0 1 1 0 1 1 1 1 1 0 1

</ Thanks! />

} /> [

Do you have any questions?

armytakmry464@gmail.com

1 0 1 1 0 1 1 0 1 0 1 1 0 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 1 0 1 1 0 1 1 1 1 1 0 1