Dart

Bite Ye

Introduction

What is Dart?

- Dart is an object-oriented, class-based language with C-style syntax.
- Dart is a client-optimized language for developing fast apps on any platform.

Why Dart?

- Dart is optimized for UI, making it a great choice for mobile and web applications.
- It supports both Just-In-Time (JIT) and Ahead-Of-Time (AOT) compilation, making it flexible.
- Dart can be transpiled into JavaScript.

Names, Binding, and Scopes

- Names are identifiers used to refer to variables, functions, classes, or types
  - UpperCamelCase for types, lowerCamelCase for variable and function names
  - Reserved words cannot be used as identifiers. (abstract, import, etc.)
- Binding
  - Dart is statically typed.
  - Variables must be declared using var, final, or const keyword before they are used.
- Scopes
  - Dart uses lexical scoping, which means the scope of variables is determined statically, simply by the layout of the code.
  - Local Scope: Variables are only available within the block they are declared.
  - Global Scope: Variables are available throughout the entire code after they are declared.

Data Types

- Basic Data Types:
  - Number: `int` and `double`
  - `String`
  - `Boolean`
  - `List` and `Map`
- When declaring a variable without specifying its type, Dart uses the `var` keyword and infers the type based on the initial value.
- `final` and `const` keywords: Both are used to declare constants. `final` at runtime and `const` at compile time
- Supports `Null` Type

Expressions and Assignment Statements

- **Expressions**: In Dart, an expression is a piece of code that produces a value when evaluated.
  - Arithmetic expressions: +, -, *, /, %
  - Logical expressions: &&, ||, !
  - Comparison expressions: ==, !=, <, >, <=, >=
  - String expressions: + and += for concatenation
- **Assignment Statements:** Dart uses the = operator for assignments. The result of an assignment expression is the value assigned.
  - Standard assignment: var a = 10;
  - Compound assignment: a += 10; // equivalent to a = a + 10;

Expressions and Assignment Statements

- **Null-aware Operators:** Dart includes operators that help to reduce the amount of code needed to work with potentially null objects.
  - Null-aware assignment: ??= assigns a value to a variable if it is null.
  - Null-aware access: ?. calls a method/getter on an object if it is not null.
- **Conditional Expressions:** Dart supports conditional (ternary) expressions.
  - Syntax: condition ? expr1 : expr2;

OOP

- **Classes and Objects:** Dart uses classes and objects to support object-oriented programming.
- **Inheritance:** Dart supports single inheritance by using `extend`.
- **Encapsulation:** Dart uses instance variables and methods to encapsulate data and functionality. Use the underscore _ to mark identifiers as private.
- **Polymorphism:** Dart supports polymorphism, allowing a child class to override or implement the methods of its parent class. Use override keyword for method overriding.
- **Abstract Classes and Interfaces:** Using `abstract` keyword.

```dart
class Stack<T> {
  List<T> _items = [];

  void push(T item) {
    _items.add(item);
  }

  T pop() {
    if (isEmpty) {
      throw Exception('Stack is empty');
    }

    return _items.removeLast();
  }

  bool get isEmpty => _items.isEmpty;

  T get top {
    if (isEmpty) {
      throw Exception('Stack is empty');
    }

    return _items.last;
  }
}
```

```dart
void main() {
  var stack = Stack<int>();

  stack.push(1);
  stack.push(2);
  stack.push(3);

  print(stack.top);  // Prints: 3

  print(stack.pop());  // Prints: 3
  print(stack.pop());  // Prints: 2
  print(stack.pop());  // Prints: 1
}
```

Concurrency

- **Single-Threaded Model:** By default, Dart follows a single-threaded model of execution.
- **Event Loop:** Dart uses an event loop for handling asynchronous operations.
- **Futures and async/await:** Dart uses `Future` objects to represent asynchronous operations that produce a result in the future.
- **Isolates:** To achieve concurrency, Dart uses `isolates`, which are independent workers that do not share memory, but instead use message passing to communicate. This prevents issues like race conditions and deadlocks.
- **Streams:** Dart uses `Stream` objects for handling sequences of asynchronous events.

```dart
class BufferMessage {
  final bool isRemove;
  final int? value;

  BufferMessage.remove()
      : isRemove = true,
        value = null;
  BufferMessage.insert(this.value) : isRemove = false;
}

void worker(SendPort sendPort) async {
  var receivePort = ReceivePort();
  var random = Random();
  sendPort.send(receivePort.sendPort);

  await for (var message in receivePort) {
    if (message is SendPort) {
      while (true) {
        if (random.nextInt(100) % 2 == 0) {
          message.send(BufferMessage.remove());
        } else {
          message.send(BufferMessage.insert(random.nextInt(10)));
        }
        await Future.delayed(
            Duration(milliseconds: random.nextInt(4000) + 1000));
      }
    }
  }
}
```

```dart
void main() async {
  var buffer = <int>[];

  var receivePort1 = ReceivePort();
  var isolate1 = await Isolate.spawn(worker, receivePort1.sendPort);
  var sendPort1 = await receivePort1.first;

  var receivePort2 = ReceivePort();
  var isolate2 = await Isolate.spawn(worker, receivePort2.sendPort);
  var sendPort2 = await receivePort2.first;

  var monitorReceivePort = ReceivePort();
  sendPort1.send(monitorReceivePort.sendPort);
  sendPort2.send(monitorReceivePort.sendPort);

  await for (var message in monitorReceivePort) {
    if (message is BufferMessage) {
      if (message.isRemove) {
        if (buffer.isNotEmpty) {
          print('Removed ${buffer.removeAt(0)}');
        } else {
          print('Buffer is empty');
        }
      } else {
        buffer.add(message.value!);
        print('Inserted ${message.value}');
        print('Current buffer ${buffer.toString()}');
      }
    } else {
      break;
    }
  }

  isolate1.kill();
  isolate2.kill();
}
```

Exception Handling and Event Handling

- **Exception Handling:** Dart provides `try-catch` block to handle exceptions.
- **Event Handling:** Dart provides event-driven programming where you can generate and respond to events.
  - **Event generation:** Events can be generated by UI interactions, I/O operations, timers, etc.
  - **Event listening:** Dart uses `Stream` to represent a sequence of events. Use listen() method to set up an event listener on a Stream. Use listen() method to set up an event listener on a Stream.
  - **Event classes:** Dart provides event classes (like MouseEvent, KeyboardEvent) that encapsulate details about the event.
- **Asynchronous Exception Handling:** Dart allows handling exceptions asynchronously using Future.catchError() and Stream.handleError().

Functional Programming

- **First-Class Functions:** Dart treats functions as first-class objects.
- **Anonymous Functions:** Dart supports anonymous functions, which are unnamed functions used for short, localized operations within your code.
- **Closures:** Dart supports closures, which are special functions that can capture and carry around state from their lexical scope.
- **Map, Reduce, and Filter:** Dart provides built-in functions of those.
- **Immutability:** Dart encourages using immutable data structures and values. The final and const keywords help define immutable variables.
- **Functional Constructs:** Dart includes constructs such as if and for in expressions, allowing for more concise and expressive code.

```
void main() {
  var double = (int x) => x * 2;
  var increment = (int x) => x + 1;

  Function compose(Function f, Function g) {
    return (int x) {
      return f(g(x));
    };
  }

  var doubleThenIncrement = compose(increment, double);
  print(doubleThenIncrement(5)); // Output: 11

  var incrementThenDouble = compose(double, increment);
  print(incrementThenDouble(5)); // Output: 12
}
```

Questions?

Thank you for your time.