Dart Programming Guide

This guide provides a comprehensive introduction to Dart programming, covering fundamental concepts, syntax, and best practices. It is designed for beginners and those looking to deepen their understanding of Dart, the language behind Flutter.

Table of Contents

Introduction to Dart & Setup

What is Dart? History, philosophy, and ecosystem

Dart is a client-optimized programming language for fast apps on any platform. It is developed by Google and is used to build mobile, desktop, web, and backend applications. Dart is the language behind Flutter, Google's UI toolkit for building natively compiled applications for mobile, web, and desktop from a single codebase.

History

Dart was first unveiled at the GOTO conference in Aarhus, Denmark, on October 10, 2011. It was initially designed to be a replacement for JavaScript as the primary language for web development. However, it gained significant traction with the advent of Flutter, which was released in 2017. Since then, Dart's popularity has grown steadily, primarily due to Flutter's success in cross-platform mobile development.

Philosophy

Dart's philosophy revolves around several key principles:

- **Client-optimized:** Dart is designed to be highly productive for building user interfaces, with features like hot reload and a rich set of UI widgets.
- **Productive:** Dart aims to provide a highly productive development experience with features like type safety, null safety, and a comprehensive standard library.
- **Fast:** Dart compiles to native code, allowing for high performance on various platforms. It also supports Just-In-Time (JIT) compilation for fast development cycles and Ahead-Of-Time (AOT) compilation for optimized production builds.
- **Portable:** Dart code can run on multiple platforms, including mobile (iOS, Android), web, desktop (Windows, macOS, Linux), and backend servers.

Ecosystem

Dart's ecosystem is rapidly expanding, with Flutter being its most prominent component. Other notable aspects of the ecosystem include:

- **Pub:** Dart's package manager, similar to npm for Node.js or Maven for Java. It hosts thousands of packages and libraries that extend Dart's functionality.
- **DartPad:** An online editor where you can write and run Dart code directly in your browser, without any local setup.
- **Dart DevTools:** A suite of debugging and performance tools for Dart and Flutter applications.
- **Community:** A growing and active community of developers contributing to Dart and Flutter projects, sharing knowledge, and providing support.

Installing Dart SDK

The Dart SDK (Software Development Kit) includes the Dart VM (Virtual Machine), `dart` command-line tool, and core libraries. Here's how to install it on different operating systems:

Windows

1. **Using Chocolatey (Recommended):** If you have Chocolatey installed, open an administrative PowerShell or Command Prompt and run: `bash choco install dart-sdk`
2. **Manual Installation:**
    - Download the Dart SDK from the official Dart website: https://dart.dev/get-dart
    - Extract the downloaded ZIP file to a desired location (e.g., `C:\src\dart`).
    - Add the `bin` directory of the Dart SDK to your system's `PATH` environment variable.

macOS

1. **Using Homebrew (Recommended):** Open Terminal and run: `bash brew tap dart-lang/dart brew install dart`
2. **Manual Installation:**
    - Download the Dart SDK from the official Dart website: https://dart.dev/get-dart
    - Extract the downloaded archive to a desired location (e.g., `/usr/local/opt/dart`).
    - Add the `bin` directory of the Dart SDK to your system's `PATH` environment variable.

Linux

1. **Using `apt` (Debian/Ubuntu):** `bash sudo apt update sudo apt install apt-transport-https sudo sh -c 'wget -qO- https://dl-ssl.google.com/linux/linux_signing_key.pub | apt-key add -' sudo sh -c 'wget -qO- https://storage.googleapis.com/download.dartlang.org/linux/debian/dart_stable.list > /etc/apt/sources.list.d/dart_stable.list' sudo apt update sudo apt install dart`
2. **Manual Installation:**
    - Download the Dart SDK from the official Dart website: https://dart.dev/get-dart
    - Extract the downloaded archive to a desired location (e.g., `/opt/dart`).
    - Add the `bin` directory of the Dart SDK to your system's `PATH` environment variable.

Verify Installation

After installation, open a new terminal or command prompt and run:

```
dart --version
```

This command should display the installed Dart SDK version, confirming a successful installation.

Setting up IDE/editor (VS Code, IntelliJ)

While you can write Dart code in any text editor, using an IDE (Integrated Development Environment) or a code editor with Dart extensions significantly enhances the development experience by providing features like syntax highlighting, code completion, debugging, and refactoring.

Visual Studio Code (VS Code)

VS Code is a popular, lightweight, and highly customizable code editor that is widely used for Dart and Flutter development.

1. **Install VS Code:** If you don't have it, download and install VS Code from the official website: https://code.visualstudio.com/
2. **Install Dart and Flutter Extensions:**
    - Open VS Code.
    - Go to the Extensions view by clicking on the Extensions icon in the Activity Bar on the side of the window (or press `Ctrl+Shift+X`).
    - Search for "Dart" and "Flutter" and install the official extensions provided by Dart Code and Google, respectively. The Flutter extension usually includes the Dart extension as a dependency.

IntelliJ IDEA / Android Studio

IntelliJ IDEA is a powerful IDE from JetBrains, and Android Studio is based on IntelliJ IDEA, making both excellent choices for Dart and Flutter development, especially for larger projects.

1. **Install IntelliJ IDEA / Android Studio:**
    - Download and install IntelliJ IDEA Community Edition (free) or Ultimate Edition from https://www.jetbrains.com/idea/
    - Download and install Android Studio from https://developer.android.com/studio
2. **Install Dart and Flutter Plugins:**
    - Open IntelliJ IDEA or Android Studio.
    - Go to `File > Settings > Plugins` (on macOS, `IntelliJ IDEA > Preferences > Plugins`).
    - Search for "Dart" and "Flutter" in the Marketplace tab and install them. Restart the IDE after installation.

Dart extensions and plugins

Beyond the core Dart and Flutter extensions, several other plugins can further improve your development workflow:

- **Awesome Flutter Snippets:** Provides a collection of useful Flutter and Dart code snippets.
- **Bracket Pair Colorizer:** Helps in identifying matching brackets by coloring them.
- **Material Icon Theme:** Adds material design icons to your file explorer, making it easier to distinguish file types.
- **Pubspec Assist:** Helps in adding, removing, and upgrading dependencies in your `pubspec.yaml` file.

Creating and running your first Dart program

Let's create a simple Dart program. This will be a classic "Hello, World!" example.

1. **Create a new directory for your project:** `bash mkdir my_first_dart_app cd my_first_dart_app`
2. **Create a new Dart file:** Inside the `my_first_dart_app` directory, create a file named `main.dart`.
3. **Write the Dart code:** Open `main.dart` in your chosen IDE/editor and add the following code: `dart // main.dart void main() { print('Hello, World!'); }`
    - `void main()`: This is the entry point of every Dart program. The `main` function is where the program execution begins. `void` indicates that the function does not return any value.
    - `print('Hello, World!');`: This line uses the built-in `print` function to display the string "Hello, World!" to the console. The semicolon `;` at the end of the statement is mandatory in Dart.
4. **Run the Dart program:** Open your terminal or command prompt, navigate to the `my_first_dart_app` directory, and run the following command: `bash dart run main.dart` You should see the output: `Hello, World!`

    Alternatively, if you are using VS Code, you can click the "Run" button in the top right corner or use the "Run and Debug" view. In IntelliJ IDEA/Android Studio, you can right-click on the `main.dart` file and select "Run 'main.dart'".

Dart compilation modes: JIT vs AOT

Dart supports two primary compilation modes: Just-In-Time (JIT) and Ahead-Of-Time (AOT) compilation. Understanding these modes is crucial for optimizing your Dart applications for development and production.

Just-In-Time (JIT) Compilation

JIT compilation occurs during runtime, meaning the code is compiled into machine code just before it is executed. This mode is primarily used during development due to its benefits:

- **Fast Development Cycles:** JIT compilation enables features like "hot reload" and "hot restart" in Flutter. When you make changes to your code, the JIT compiler can quickly inject the updated code into the running application without losing its state, leading to extremely fast iteration times.

- **Dynamic Features:** JIT allows for more dynamic features, such as dynamic code loading and reflection, which can be useful during development for rapid prototyping and experimentation.
- **Optimizations at Runtime:** The JIT compiler can perform optimizations based on the actual runtime behavior of the program, potentially leading to highly optimized code for specific execution paths.

When to use JIT: JIT compilation is ideal for the development phase of your application, where rapid iteration and quick feedback are paramount. It's the default compilation mode when you run `dart run` or `flutter run` in debug mode.

Ahead-Of-Time (AOT) Compilation

AOT compilation occurs before the application is run, typically during the build process for deployment. The entire Dart code is compiled into native machine code, which can then be directly executed by the operating system. This mode is designed for production environments due to its advantages:

- **Fast Startup Performance:** Since the code is already compiled to native machine code, AOT compiled applications start much faster as there's no runtime compilation overhead.
- **Predictable Performance:** AOT compilation results in more consistent and predictable performance because all optimizations are applied during the build process, and there are no pauses for JIT compilation during runtime.
- **Smaller Code Size:** AOT compilation can lead to smaller executable sizes because the compiler can remove unused code (tree shaking) and optimize the generated machine code more aggressively.
- **Enhanced Security:** AOT compiled applications are generally more secure as they don't include a JIT compiler or dynamic code loading capabilities, making them less susceptible to certain types of attacks.

When to use AOT: AOT compilation is essential for deploying your Dart applications to production. When you build a Flutter app for release (e.g., `flutter build apk`, `flutter build ios`), it is AOT compiled. Similarly, when you compile a standalone Dart application for deployment, you would use AOT compilation.

Summary of JIT vs AOT

Feature	Just-In-Time (JIT) Compilation	Ahead-Of-Time (AOT) Compilation
Timing	During runtime	Before runtime (build time)
Primary Use	Development, rapid iteration	Production, deployment
Startup Speed	Slower (due to runtime compilation)	Faster (pre-compiled)
Performance	Dynamic, can vary	Consistent, predictable
Code Size	Larger (includes compiler)	Smaller (optimized, tree-shaken)
Hot Reload/Restart	Supported	Not supported
Optimizations	Runtime-based	Build-time based
Security	Less secure (dynamic code)	More secure (static code)

Understanding the differences between JIT and AOT compilation allows developers to leverage Dart's strengths for both efficient development and high-performance production applications.

Variables and Data Types

In Dart, variables are used to store data values. Before you can use a variable, you must declare it. Dart is a type-safe language, meaning that variables have a static type that is determined at compile time. This helps prevent errors and makes your code more robust.

Variable declaration (var, final, const, late)

Dart provides several keywords for declaring variables, each with specific characteristics regarding mutability and initialization.

`var`

The `var` keyword is used for type inference. When you declare a variable with `var`, Dart infers its type based on the initial value assigned to it. Once the type is inferred, it cannot be changed.

```dart
void main() {
  var name = "Alice"; // Dart infers String type
  print(name);

  // name = 123; // This would cause a compile-time error because name is already a String

  var age = 30; // Dart infers int type
  print(age);

  var price = 19.99; // Dart infers double type
  print(price);
}
```

In the example above, `name` is inferred as a `String`, `age` as an `int`, and `price` as a `double`. While `var` provides flexibility during declaration, the variable's type becomes fixed after initialization.

`final`

The `final` keyword is used to declare a variable that can only be set once. Its value cannot be changed after it has been initialized. `final` variables are initialized at runtime, meaning their value can be determined based on calculations or external factors that are not known at compile time.

```dart
void main() {
  final String firstName = "Bob";
  // firstName = "Robert"; // Error: A final variable can only be set once.
  print(firstName);

  final DateTime now = DateTime.now(); // Value determined at runtime
  print(now);

  // Example with type inference
  final lastName = "Smith"; // Inferred as String
  print(lastName);
}
```

`final` is useful for values that are constant within the lifetime of an object or function, but might vary between different runs of the program or different instances of an object.

`const`

The `const` keyword is used to declare a compile-time constant. This means the value of a `const` variable must be known at compile time and cannot change ever. `const` variables are implicitly `final`.

```dart
void main() {
  const double PI = 3.14159;
  // PI = 3.0; // Error: Constant variables can't be assigned a value.
  print(PI);

  const String GREETING = "Hello, Dart!";
  print(GREETING);

  // const DateTime compileTime = DateTime.now(); // Error: Not a compile-time constant
}
```

`const` is ideal for values that are truly immutable and known before the program even starts executing, such as mathematical constants or fixed configuration values. Using `const` can also lead to performance optimizations as the compiler can inline these values.

`late`

The `late` keyword was introduced in Dart 2.12 to address two main use cases:

1. **Declaring a non-nullable variable that's initialized after its declaration:** This is useful when a variable's value cannot be known at the point of declaration but will definitely be assigned before it's used.
2. **Lazy initialization:** A `late` variable's initializer runs only when the variable is accessed for the first time. This can be beneficial for performance if the initialization is expensive and the variable might not always be used.

```
late String description; // Declared but not initialized

void main() {
  description = "This is a late-initialized string."; // Initialized later
  print(description);

  late int expensiveCalculation = _performExpensiveCalculation();
  print("Accessing expensiveCalculation...");
  print(expensiveCalculation); // _performExpensiveCalculation() is called here
}

int _performExpensiveCalculation() {
  print("Performing expensive calculation...");
  // Simulate a time-consuming operation
  return 42;
}
```

In the example, `description` is initialized after its declaration. `expensiveCalculation` demonstrates lazy initialization; the `_performExpensiveCalculation()` function is only executed when `expensiveCalculation` is first accessed.

Summary of Variable Declaration Keywords

Keyword	Mutability	Initialization Timing	Use Case
`var`	Mutable	At declaration	General-purpose, type inferred
`final`	Immutable	At runtime	Value set once, cannot be changed
`const`	Immutable	At compile time	Compile-time constants, truly immutable
`late`	Mutable	After declaration / Lazy	Non-nullable variables initialized later, expensive initialization

Built-in data types (int, double, String, bool)

Dart comes with a set of built-in data types to represent common kinds of values.

`int`

Represents integer numbers (whole numbers) up to 64 bits on platforms that support it, or 32 bits otherwise. Dart integers are arbitrary-precision on the web.

```
void main() {
  int age = 25;
  int temperature = -10;
  print("Age: $age");
  print("Temperature: $temperature");
}
```

`double`

Represents floating-point numbers (numbers with decimal points). `double` is a 64-bit (double-precision) floating-point number.

```
void main() {
  double price = 99.99;
  double gravity = 9.81;
  print("Price: $price");
  print("Gravity: $gravity");
}
```

`num`

`num` is an abstract class that is the supertype of both `int` and `double`. You can use `num` if you want a variable to hold either an integer or a double.

```
void main() {
  num quantity = 10; // Can be int
  print("Quantity: $quantity");

  quantity = 15.5; // Can be double
  print("Quantity: $quantity");
}
```

`String`

Represents a sequence of characters (text). Dart strings are UTF-16 code units. You can use single or double quotes to create string literals.

```
void main() {
  String greeting = "Hello, Dart!";
  String message = 'Welcome to the world of Flutter.';
  print(greeting);
  print(message);

  // Multi-line strings
  String multiLineString = """
This is a multi-line string.
It can span multiple lines
without using newline characters.\n
  """;
  print(multiLineString);

  String rawString = r"C:\Program Files\Dart"; // Raw string, ignores escape sequences
  print(rawString);
}
```

`bool`

Represents boolean values: `true` or `false`. Dart is strict about boolean types; only the boolean values `true` and `false` are treated as such. Unlike some other languages, `0`, `1`, empty strings, or `null` are not implicitly converted to boolean values.

```
void main() {
  bool isActive = true;
  bool hasPermission = false;
  print("Is active: $isActive");
  print("Has permission: $hasPermission");

  if (isActive) {
    print("User is active.");
  }
}
```

Type inference and explicit typing

Dart supports both type inference and explicit typing, giving developers flexibility in how they declare variables.

Type Inference

As seen with the `var` keyword, Dart can infer the type of a variable based on the value assigned to it during initialization. This reduces verbosity and can make code cleaner, especially when the type is obvious from the context.

```dart
void main() {
  var count = 100; // Inferred as int
  var name = "Dart"; // Inferred as String
  var isReady = true; // Inferred as bool
  var temperature = 25.5; // Inferred as double

  print("Count: ${count.runtimeType}");
  print("Name: ${name.runtimeType}");
  print("Is Ready: ${isReady.runtimeType}");
  print("Temperature: ${temperature.runtimeType}");
}
```

Once a type is inferred, it is fixed for the lifetime of that variable. You cannot assign a value of a different type to it.

Explicit Typing

Explicit typing involves explicitly stating the data type of a variable during its declaration. This can improve code readability, especially for complex types or when the initial value doesn't fully convey the intended type.

```dart
void main() {
  int quantity = 50;
  String productName = "Laptop";
  bool isValid = false;
  double discount = 0.15;

  print("Quantity: $quantity");
  print("Product Name: $productName");
  print("Is Valid: $isValid");
  print("Discount: $discount");
}
```

When to use which?

- **Use `var` (type inference) when:**
    - The type is obvious from the initializer (e.g., `var count = 0;`).
    - It makes the code more concise without sacrificing readability.
    - You are dealing with local variables where the type is less critical for understanding the overall program flow.
- **Use explicit typing when:**
    - The type is not immediately obvious from the initializer.
    - You want to make the code's intent clearer, especially for API boundaries (function parameters, return types) or public fields.
    - You are declaring variables without an initial value.

```dart
void main() {
  // Explicit typing for clarity, especially for complex types or APIs
  List<String> names = ["Alice", "Bob"];
  Map<String, int> scores = {"Alice": 95, "Bob": 88};

  // Type inference for local variables where type is clear
  var totalScore = 0;
  for (var score in scores.values) {
    totalScore += score;
  }
  print("Total Score: $totalScore");
}
```

String interpolation and manipulation

Dart provides powerful and convenient ways to work with strings, including interpolation for embedding expressions and various methods for manipulation.

String Interpolation

String interpolation allows you to embed the value of an expression inside a string literal. This is done using `$ or ``$variableName`.

```dart
void main() {
  String name = "World";
  int year = 2025;
  double temperature = 23.5;

  // Interpolating a variable
  print("Hello, $name!");

  // Interpolating an expression
  print("The current year is ${year + 1}.");

  // Interpolating a method call
  print("Temperature: ${temperature.toStringAsFixed(1)}°C");

  String city = "New York";
  String country = "USA";
  print("I live in $`city, `$country.");
}
```

String Manipulation Methods

Dart's `String` class provides a rich set of methods for common string operations.

- **Concatenation:** Joining strings together.

  ```dart void main() { String firstName = "John"; String lastName = "Doe"; String fullName = firstName + " " + lastName; print(fullName);

  // Using adjacent string literals (for constants) String greeting = "Good" " Morning"; print(greeting); } ```

- **Length:** Getting the number of characters in a string.
  ```

```
dart void main() { String text = "Dart Programming"; print("Length: ${text.length}"); }
```

- **`toUpperCase()` and `toLowerCase()`:** Converting case.

  ```
  dart void main() { String message = "Hello Dart"; print(message.toUpperCase()); print(message.toLowerCase()); }
  ```

- **`contains()`:** Checking if a string contains a substring.

  ```
  dart void main() { String sentence = "Dart is a powerful language."; print(sentence.contains("Dart")); // true print(sentence.contains("Java")); // false }
  ```

- **`startsWith()` and `endsWith()`:** Checking prefixes and suffixes.

  ```
  dart void main() { String fileName = "document.pdf"; print(fileName.startsWith("doc")); // true print(fileName.endsWith(".pdf")); // true }
  ```

- **`substring()`:** Extracting a portion of a string.

  ```
  dart void main() { String email = "user@example.com"; String domain = email.substring(email.indexOf('@') + 1); print(domain); // example.com }
  ```

- **`replaceFirst()`, `replaceAll()`:** Replacing parts of a string.

  ```
  dart void main() { String text = "Hello world, hello Dart."; print(text.replaceFirst("hello", "hi")); // Hi world, hello Dart. print(text.replaceAll("hello", "hi")); // Hi world, hi Dart. }
  ```

- **`trim()`, `trimLeft()`, `trimRight()`:** Removing leading/trailing whitespace.

  ```
  dart void main() { String paddedText = " Some text with spaces "; print("'"+ paddedText.trim() + "'"); // 'Some text with spaces' print("'"+ paddedText.trimLeft() + "'"); // 'Some text with spaces ' print("'"+ paddedText.trimRight() + "'"); // ' Some text with spaces' }
  ```

- **`split()`:** Splitting a string into a list of substrings.

  ```
  dart void main() { String csvData = "apple,banana,orange"; List<String> fruits = csvData.split(","); print(fruits); // [apple, banana, orange] }
  ```

Numeric type conversion

Dart provides straightforward ways to convert between numeric types (`int` and `double`) and to/from strings.

`int` to `double`

An `int` can be directly assigned to a `double` variable, as `double` can represent all integer values. This is an implicit conversion.

```
void main() {
  int integerValue = 10;
  double doubleValue = integerValue.toDouble(); // Explicit conversion
  print("Integer: $`integerValue, Double: `$doubleValue"); // Output: Integer: 10, Double:
10.0

  double anotherDouble = integerValue; // Implicit conversion
  print("Another Double: $anotherDouble"); // Output: Another Double: 10.0
}
```

`double` to `int`

Converting a `double` to an `int` requires explicit conversion because it might involve loss of precision (the decimal part). Dart provides methods like `toInt()`, `round()`, `floor()`, and `ceil()`.

```
void main() {
  double decimalValue = 15.7;

  int intValue = decimalValue.toInt(); // Truncates the decimal part
  print("toInt(): $intValue"); // Output: toInt(): 15

  int roundedValue = decimalValue.round(); // Rounds to the nearest integer
  print("round(): $roundedValue"); // Output: round(): 16

  int flooredValue = decimalValue.floor(); // Rounds down to the nearest integer
  print("floor(): $flooredValue"); // Output: floor(): 15

  int ceiledValue = decimalValue.ceil(); // Rounds up to the nearest integer
  print("ceil(): $ceiledValue"); // Output: ceil(): 16

  double negativeDecimal = -12.3;
  print("toInt() negative: ${negativeDecimal.toInt()}"); // Output: toInt() negative: -12
  print("round() negative: ${negativeDecimal.round()}"); // Output: round() negative: -12
  print("floor() negative: ${negativeDecimal.floor()}"); // Output: floor() negative: -13
  print("ceil() negative: ${negativeDecimal.ceil()}"); // Output: ceil() negative: -12
}
```

String to Numeric

To convert a `String` to an `int` or `double`, you can use the `parse()` method available on `int` and `double` classes. These methods throw a `FormatException` if the string cannot be parsed.

```dart
void main() {
  String intString = "123";
  String doubleString = "45.67";
  String invalidString = "abc";

  try {
    int parsedInt = int.parse(intString);
    print("Parsed int: $parsedInt"); // Output: Parsed int: 123

    double parsedDouble = double.parse(doubleString);
    print("Parsed double: $parsedDouble"); // Output: Parsed double: 45.67

    // Using tryParse() for safer conversion (returns null on failure)
    int? safeParsedInt = int.tryParse(invalidString);
    print("Safe parsed int: $safeParsedInt"); // Output: Safe parsed int: null

    // This would throw a FormatException
    // int.parse(invalidString);
  } catch (e) {
    print("Error parsing string: $e");
  }
}
```

Numeric to String

To convert an `int` or `double` to a `String`, you can use the `toString()` method.

```dart
void main() {
  int number = 100;
  double pi = 3.14159;

  String numberString = number.toString();
  print("Number as string: $numberString"); // Output: Number as string: 100
  print("Type of numberString: ${numberString.runtimeType}");

  String piString = pi.toString();
  print("Pi as string: $piString"); // Output: Pi as string: 3.14159

  // Formatting double to string with specific decimal places
  String formattedPi = pi.toStringAsFixed(2);
  print("Formatted Pi: $formattedPi"); // Output: Formatted Pi: 3.14
}
```

These conversion methods are essential for handling user input, data serialization, and interoperability with other systems that might represent data in different formats.

Operators in Dart

Operators are special symbols that perform operations on one, two, or three operands. Dart provides a rich set of operators that are similar to those in other C-style languages.

Arithmetic operators (+, -, *, /, %, ~/)

These operators perform basic mathematical calculations.

- `+` (Addition): Adds two operands.
- `-` (Subtraction): Subtracts the second operand from the first.
- `*` (Multiplication): Multiplies two operands.
- `/` (Division): Divides the first operand by the second, resulting in a `double`.
- `%` (Modulo): Returns the remainder of an integer division.
- `~/` (Integer Division): Divides two operands and returns an integer result.

```
void main() {
  int a = 10;
  int b = 3;

  print("a + b = ${a + b}");       // 13
  print("a - b = ${a - b}");       // 7
  print("a * b = ${a * b}");       // 30
  print("a / b = ${a / b}");       // 3.333...
  print("a % b = ${a % b}");       // 1
  print("a ~/ b = ${a ~/ b}");     // 3
}
```

Logical operators (&&, ||, !)

Logical operators are used to combine or invert boolean expressions.

- `&&` (Logical AND): Returns `true` if both operands are `true`.
- `||` (Logical OR): Returns `true` if at least one of the operands is `true`.
- `!` (Logical NOT): Inverts the boolean value of its operand.

```
void main() {
  bool isSunny = true;
  bool isWarm = false;

  print("isSunny && isWarm: ${isSunny && isWarm}"); // false
  print("isSunny || isWarm: ${isSunny || isWarm}"); // true
  print("!isSunny: ${!isSunny}");                   // false
}
```

Comparison operators (==, !=, <, >, <=, >=)

These operators compare two operands and return a boolean value.

- `==` (Equal to): Returns `true` if the operands are equal.
- `!=` (Not equal to): Returns `true` if the operands are not equal.
- `<` (Less than): Returns `true` if the first operand is less than the second.
- `>` (Greater than): Returns `true` if the first operand is greater than the second.
- `<=` (Less than or equal to): Returns `true` if the first operand is less than or equal to the second.

- `>=` (Greater than or equal to): Returns `true` if the first operand is greater than or equal to the second.

```
void main() {
  int x = 5;
  int y = 10;

  print("x == y: ${x == y}");   // false
  print("x != y: ${x != y}");   // true
  print("x < y: ${x < y}");    // true
  print("x > y: ${x > y}");    // false
  print("x <= 5: ${x <= 5}"); // true
  print("y >= 10: ${y >= 10}"); // true
}
```

Assignment operators (=, +=, -=, *=, /=, etc.)

Assignment operators are used to assign values to variables. The compound assignment operators provide a shorthand way to perform an operation and assign the result.

- `=` (Assignment): Assigns the value of the right operand to the left operand.
- `+=` (Add and assign): Adds the right operand to the left operand and assigns the result to the left operand.
- `-=` (Subtract and assign): Subtracts the right operand from the left operand and assigns the result to the left operand.
- `*=` (Multiply and assign): Multiplies the left operand by the right operand and assigns the result to the left operand.
- `/=` (Divide and assign): Divides the left operand by the right operand and assigns the result to the left operand.

```
void main() {
  int num = 10;
  print("Initial value: $num");

  num += 5; // num = num + 5
  print("After += 5: $num");

  num -= 3; // num = num - 3
  print("After -= 3: $num");

  num *= 2; // num = num * 2
  print("After *= 2: $num");

  // Note: /= results in a double
  double doubleNum = num.toDouble();
  doubleNum /= 4; // doubleNum = doubleNum / 4
  print("After /= 4: $doubleNum");
}
```

Operator precedence and associativity

Operator precedence determines the order in which operators are evaluated in an expression. Associativity determines the order in which operators with the same precedence are evaluated.

For example, `*` and `/` have higher precedence than `+` and `-`. Most operators in Dart are left-associative.

```
void main() {
  int result = 10 + 5 * 2; // 5 * 2 is evaluated first
  print("10 + 5 * 2 = $result"); // 20

  int anotherResult = (10 + 5) * 2; // Use parentheses to override precedence
  print("(10 + 5) * 2 = $anotherResult"); // 30
}
```

Conditional Statements in Dart

Conditional statements allow you to execute different blocks of code based on certain conditions.

if, else if, else statements

The `if` statement executes a block of code if a specified condition is true. The `else if` statement allows you to check for multiple conditions, and the `else` statement provides a default block of code to execute if none of the preceding conditions are true.

```
void main() {
  int score = 85;

  if (score >= 90) {
    print("Grade: A");
  } else if (score >= 80) {
    print("Grade: B");
  } else if (score >= 70) {
    print("Grade: C");
  } else {
    print("Grade: D");
  }
}
```

Ternary operator (condition ? expr1 : expr2)

The ternary operator is a concise way to write an `if-else` statement. If the condition is true, `expr1` is evaluated; otherwise, `expr2` is evaluated.

```dart
void main() {
  int age = 20;
  String message = age >= 18 ? "You are an adult." : "You are a minor.";
  print(message);
}
```

Conditional expressions (?? and ??=)

Dart provides special operators for handling null values.

- `??` (Null-aware operator): If the expression on the left is not null, it returns its value; otherwise, it evaluates and returns the value of the expression on the right.
- `??=` (Null-aware assignment operator): Assigns a value to a variable only if that variable is currently null.

```dart
void main() {
  String? name; // Nullable string
  String displayName = name ?? "Guest";
  print("Display Name: $displayName"); // Guest

  name = "Alice";
  displayName = name ?? "Guest";
  print("Display Name: $displayName"); // Alice

  int? a;
  a ??= 10;
  print("a: $a"); // 10

  a ??= 20;
  print("a: $a"); // 10 (not reassigned)
}
```

Switch statements and pattern matching

`switch` statements provide a way to test a variable against a series of values. Dart has enhanced `switch` statements with pattern matching, making them more powerful and expressive.

```dart
void main() {
  String command = "OPEN";

  switch (command) {
    case "OPEN":
      print("Opening the file...");
      break;
    case "CLOSE":
      print("Closing the file...");
      break;
    default:
      print("Unknown command.");
  }

  // Switch with pattern matching (Dart 3+)
  var shape = ("circle", 5);
  switch (shape) {
    case ("circle", var radius):
      print("Circle with radius $radius");
      break;
    case ("square", var side):
      print("Square with side $side");
      break;
    default:
      print("Unknown shape");
  }
}
```

```
# Loops and Loop Control in Dart

Loops are fundamental control flow statements that allow you to execute a block of code
repeatedly. Dart provides several types of loops to handle different iteration scenarios.

## for loops (traditional, for-in, forEach)

### Traditional `for` loop

The traditional `for` loop is used when you know exactly how many times you want to iterate.
It consists of an initialization, a condition, and an increment/decrement statement.

```dart
void main() {
  // Print numbers from 1 to 5
  for (int i = 1; i <= 5; i++) {
    print("Count: $i");
  }

  // Iterate over a list using index
  List<String> fruits = ["Apple", "Banana", "Cherry"];
  for (int i = 0; i < fruits.length; i++) {
    print("Fruit at index $`i: `${fruits[i]}");
  }
}
```

`for-in` loop

The `for-in` loop (also known as a enhanced for loop or foreach loop in other languages) is used to iterate over collections like lists, sets, and maps.
```

```
void main() {
  List<String> colors = ["Red", "Green", "Blue"];
  for (String color in colors) {
    print("Color: $color");
  }

  Map<String, int> scores = {"Alice": 90, "Bob": 85, "Charlie": 92};
  for (var entry in scores.entries) {
    print("$`{entry.key}: `${entry.value}");
  }
}
```

`forEach` loop

The `forEach` method is available on iterable objects (like `List`, `Set`, `Map`) and provides a functional way to iterate over elements. It takes a function as an argument, which is executed for each element.

```
void main() {
  List<int> numbers = [1, 2, 3, 4, 5];
  numbers.forEach((number) {
    print("Number: $number");
  });

  Set<String> uniqueItems = {"Pen", "Book", "Eraser"};
  uniqueItems.forEach((item) => print("Item: $item")); // Using arrow function
}
```

while loops

The `while` loop executes a block of code as long as a specified condition is true. The condition is evaluated before each iteration.

```
void main() {
  int i = 0;
  while (i < 5) {
    print("While count: $i");
    i++;
  }
}
```

do-while loops

The `do-while` loop is similar to the `while` loop, but it guarantees that the loop body is executed at least once, because the condition is evaluated after the first iteration.

```dart
void main() {
  int i = 0;
  do {
    print("Do-While count: $i");
    i++;
  } while (i < 5);

  // Example where condition is initially false
  int j = 10;
  do {
    print("This will print once: $j");
    j++;
  } while (j < 5);
}
```

break and continue statements

`break` statement

The `break` statement is used to terminate the innermost loop immediately. Execution continues with the statement immediately following the loop.

```dart
void main() {
  for (int i = 0; i < 10; i++) {
    if (i == 5) {
      break; // Exit the loop when i is 5
    }
    print("Break example: $i");
  }
  print("Loop finished (break).");
}
```

`continue` statement

The `continue` statement is used to skip the current iteration of a loop and proceed to the next iteration. It effectively bypasses the remaining code in the loop body for the current iteration.

```dart
void main() {
  for (int i = 0; i < 10; i++) {
    if (i % 2 == 0) {
      continue; // Skip even numbers
    }
    print("Continue example (odd): $i");
  }
}
```

Loop control with labels

Dart allows you to label loops, which can be useful when you have nested loops and want to `break` or `continue` an outer loop from within an inner loop.

```dart
void main() {
  outerLoop:
  for (int i = 1; i <= 3; i++) {
    innerLoop:
    for (int j = 1; j <= 3; j++) {
      if (i == 2 && j == 2) {
        break outerLoop; // Breaks out of the outerLoop
      }
      print("i: $`i, j: `$j");
    }
  }
  print("Exited outer loop.");

  print("\n--- Continue with label ---");

  anotherOuterLoop:
  for (int i = 1; i <= 3; i++) {
    for (int j = 1; j <= 3; j++) {
      if (i == 2 && j == 1) {
        continue anotherOuterLoop; // Skips to the next iteration of anotherOuterLoop
      }
      print("i: $`i, j: `$j");
    }
  }
}
```

Functions in Dart

Functions are blocks of code that perform a specific task. They help organize code, make it reusable, and improve readability. In Dart, everything is an object, including functions.

Function declaration and syntax

Functions in Dart are declared using the following syntax:

```dart
returnType functionName(parameter1Type parameter1, parameter2Type parameter2, ...) {
  // Function body
  return someValue;
}
```

- `returnType` : The data type of the value that the function will return. If the function does not return a value, use `void` .
- `functionName` : The name of the function.
- `parameters` : A comma-separated list of input parameters, each with its type and name.

```dart
void main() {
  // Calling a function that returns void
  greet("Alice");

  // Calling a function that returns a value
  int sumResult = add(10, 20);
  print("Sum: $sumResult");

  String message = createMessage("Dart");
  print(message);
}

// Function that returns void (does not return a value)
void greet(String name) {
  print("Hello, $name!");
}

// Function that returns an int
int add(int a, int b) {
  return a + b;
}

// Function that returns a String
String createMessage(String topic) {
  return "Learning $topic is fun!";
}
```

Arrow functions

Dart supports a shorthand syntax for functions that contain only a single expression. These are called arrow functions (or fat arrow syntax) and use `=>`.

```dart
void main() {
  print("Multiply: ${multiply(5, 4)}");
  print("Is Even: ${isEven(7)}");
}

// Traditional function
int multiply(int a, int b) {
  return a * b;
}

// Arrow function equivalent
// int multiply(int a, int b) => a * b;

bool isEven(int number) => number % 2 == 0;
```

Arrow functions are particularly useful for callbacks and when defining simple, single-line operations.

Anonymous functions

Anonymous functions (also called lambda functions or closures) are functions without a name. They are often used as arguments to other functions (callbacks) or when you need a function for a short period.

```dart
void main() {
  // Assigning an anonymous function to a variable
  var sayHello = (String name) {
    print("Hello, $name from anonymous function!");
  };
  sayHello("Bob");

  // Using an anonymous function with forEach
  List<int> numbers = [1, 2, 3];
  numbers.forEach((number) {
    print("Number squared: ${number * number}");
  });

  // Anonymous function with arrow syntax
  numbers.forEach((number) => print("Number cubed: ${number * number * number}"));
}
```

Higher-order functions and callbacks

Higher-order functions are functions that can take other functions as arguments or return functions as results. Callbacks are functions passed as arguments to other functions, to be executed later.

```dart
void main() {
  // Example of a higher-order function
  void executeOperation(int a, int b, Function operation) {
    print("Result: ${operation(a, b)}");
  }

  // Passing an anonymous function as a callback
  executeOperation(10, 5, (x, y) => x + y); // Addition
  executeOperation(10, 5, (x, y) => x - y); // Subtraction

  // Another example: filter a list
  List<String> names = ["Alice", "Bob", "Anna", "Charlie"];
  List<String> filteredNames = filterList(names, (name) => name.startsWith("A"));
  print("Names starting with A: $filteredNames");
}

// A higher-order function that filters a list based on a predicate function
List<T> filterList<T>(List<T> list, bool Function(T) predicate) {
  List<T> result = [];
  for (var item in list) {
    if (predicate(item)) {
      result.add(item);
    }
  }
  return result;
}
```

Higher-order functions and callbacks are fundamental to functional programming paradigms and are extensively used in Flutter for handling events, asynchronous operations, and UI updates.

Function Parameters and Scope in Dart

Understanding how to define and use function parameters, as well as how variable scope works, is crucial for writing well-structured and maintainable Dart code.

Positional vs named parameters

Dart functions can have two types of parameters: positional and named.

Positional Parameters

Positional parameters are the most common type. They are defined in the order they appear in the function signature, and arguments must be passed to them in the same order.

```
void main() {
  // All arguments are positional
  greetUser("Alice", 30);
  displayProduct("Laptop", 1200.0, 10);
}

void greetUser(String name, int age) {
  print("Hello, $`name! You are `$age years old.");
}

void displayProduct(String name, double price, int quantity) {
  print("Product: $name, Price: ```math
price, Quantity: $quantity");
}
```

Named Parameters

Named parameters are enclosed in curly braces `{}` in the function signature. When calling the function, you specify the parameter name along with its value. This makes function calls more readable, especially when a function has many parameters.

By default, named parameters are optional. To make a named parameter required, you can use the `required` keyword (introduced in Dart 2.12).

```
void main() {
  // Calling with named parameters
  printUserDetails(name: "Bob", age: 25, city: "New York");
  printUserDetails(age: 40, name: "Charlie"); // Order doesn't matter

  // Calling with required named parameter
  createOrder(product: "Book", quantity: 2);
}

void printUserDetails({String? name, int? age, String? city}) {
  print("Name: $`{name ?? 'N/A'}, Age: `${age ?? 'N/A'}, City: ${city ?? 'N/A'}");
}

void createOrder({required String product, required int quantity, double? discount}) {
  print("Order created for: $`product, Quantity: `$quantity");
  if (discount != null) {
    print("Discount applied: ${discount * 100}%");
  }
}
```

Named parameters are particularly useful in Flutter widgets, where widgets often have many optional configuration properties.

Optional parameters and default values

Dart allows you to define optional parameters, which can be either positional or named. You can also provide default values for these parameters.

Optional Positional Parameters

Optional positional parameters are enclosed in square brackets `[]`. They must appear after any required positional parameters.

```
void main() {
  sayGreeting("Hello");
  sayGreeting("Hi", "Alice");
  sayGreeting("Good morning", "Bob", "How are you?");
}

void sayGreeting(String greeting, [String? name, String? message]) {
  String result = greeting;
  if (name != null) {
    result += ", $name";
  }
  if (message != null) {
    result += ". $message";
  }
  print(result);
}
```

Optional Named Parameters with Default Values

Named parameters can have default values. If a value is not provided during the function call, the default value is used.

```
void main() {
  configureSettings();
  configureSettings(theme: "dark");
  configureSettings(fontSize: 16.0, theme: "light");
}

void configureSettings({
  String theme = "light",
  double fontSize = 14.0,
  bool notificationsEnabled = true,
}) {
  print("Settings: Theme: $`theme, Font Size: `$fontSize, Notifications:
$notificationsEnabled");
}
```

Function scope (local, global)

Scope refers to the region of a program where a variable or function can be accessed. Dart has lexical scoping, meaning the scope of a variable is determined by its location within the source code.

Local Scope

Variables declared inside a function or a block (like an `if` statement or a loop) have local scope. They can only be accessed within that function or block.

```
void main() {
  String globalMessage = "This is a global message."; // Global to main, but not truly
global

  void myFunction() {
    String localVariable = "I am local to myFunction.";
    print(localVariable);
    print(globalMessage); // Can access variables from outer scope
  }

  myFunction();
  // print(localVariable); // Error: Undefined name 'localVariable'
}

// Top-level variables have global scope within the file
String appName = "My Dart App";

void anotherFunction() {
  print("Accessing global variable: $appName");
}
```

In Dart, variables declared at the top level of a file (outside any function or class) have file-level scope, which is often referred to as global scope within the context of that file. They can be accessed from anywhere within that file.

Global Scope (File-level Scope)

```
// This is a top-level variable, accessible throughout this file
String applicationName = "AwesomeApp";

void printApplicationName() {
  print("Application Name: $applicationName");
}

void main() {
  printApplicationName();
  print("From main: $applicationName");

  // Variables declared inside main are local to main
  String mainLocalVar = "Local to main";
  print(mainLocalVar);
}

// print(mainLocalVar); // Error: Undefined name 'mainLocalVar'
```

Closures and variable capturing

A closure is a function that has access to the parent scope, even after the parent function has closed. In Dart, all functions are closures because they can access variables defined in their lexical scope.

```dart
void main() {
  // Example 1: Counter closure
  Function counter = createCounter();
  print(counter()); // 1
  print(counter()); // 2
  print(counter()); // 3

  // Example 2: Multiplier closure
  Function multiplier = createMultiplier(5);
  print(multiplier(2)); // 10
  print(multiplier(3)); // 15
}

// This function returns another function (a closure)
Function createCounter() {
  int count = 0; // This variable is 'captured' by the returned function
  return () {
    count++;
    return count;
  };
}

// This function returns a closure that multiplies by a given factor
Function createMultiplier(int factor) {
  return (int number) => number * factor;
}
```

In the `createCounter` example, the anonymous function returned by `createCounter` retains access to the `count` variable, even after `createCounter` has finished executing. Each time the returned function is called, it increments and returns its own `count`.

Similarly, in `createMultiplier`, the `factor` variable is captured by the returned anonymous function, allowing it to be used in subsequent calls to the `multiplier` function.

Closures are a powerful feature in Dart, enabling patterns like encapsulation, state management, and functional programming constructs. They are widely used in Flutter for managing widget states, handling events, and more.

Collections (Lists and Sets)

Collections are a way to group related objects. Dart provides core collection types like Lists, Sets, and Maps, which are highly optimized and flexible.

List creation, manipulation, and methods

A `List` is an ordered collection of values, also known as an array in other programming languages. Lists in Dart are zero-indexed, meaning the first element is at index 0.

List Creation

```
void main() {
  // Literal way to create a List (most common)
  List<int> numbers = [1, 2, 3, 4, 5];
  print("Numbers: $numbers");

  // Creating an empty List
  List<String> names = [];
  print("Empty names list: $names");

  // Using the List constructor (less common for simple lists)
  List<double> temperatures = List.empty(growable: true);
  temperatures.add(25.5);
  temperatures.add(28.0);
  print("Temperatures: $temperatures");

  // List.filled: Creates a list of a specified length with a default value
  List<int> fixedList = List.filled(3, 0);
  print("Fixed list: $fixedList"); // [0, 0, 0]

  // List.generate: Creates a list where each element is generated by a function
  List<String> greetings = List.generate(3, (index) => "Hello ${index + 1}");
  print("Greetings: $greetings"); // [Hello 1, Hello 2, Hello 3]
}
```

List Manipulation and Methods

Dart's `List` class provides a rich set of methods for adding, removing, updating, and querying elements.

- **`add(element)`:** Adds an element to the end of the list.
- **`addAll(iterable)`:** Adds all elements from an iterable to the end of the list.
- **`insert(index, element)`:** Inserts an element at a specific index.
- **`remove(element)`:** Removes the first occurrence of the specified element.
- **`removeAt(index)`:** Removes the element at the specified index.
- **`removeLast()`:** Removes and returns the last element.
- **`clear()`:** Removes all elements from the list.
- **`length`:** Returns the number of elements in the list.
- **`isEmpty`, `isNotEmpty`:** Checks if the list is empty.
- **`contains(element)`:** Checks if the list contains the specified element.
- **`indexOf(element)`:** Returns the index of the first occurrence of the element.
- **`elementAt(index)`:** Returns the element at the specified index (same as `list[index]`).
- **`first`, `last`:** Returns the first and last elements.
- **`sublist(start, [end])`:** Returns a new list containing elements from `start` to `end` (exclusive).
- **`sort([compare])`:** Sorts the list. Optionally takes a comparison function.

- **`reversed`:** Returns an iterable of the elements in reverse order.

```
void main() {
  List<String> fruits = ["Apple", "Banana", "Cherry"];
  print("Initial fruits: $fruits");

  // Add elements
  fruits.add("Date");
  print("After add: $fruits"); // [Apple, Banana, Cherry, Date]

  fruits.addAll(["Elderberry", "Fig"]);
  print("After addAll: $fruits"); // [Apple, Banana, Cherry, Date, Elderberry, Fig]

  fruits.insert(1, "Grape");
  print("After insert: $fruits"); // [Apple, Grape, Banana, Cherry, Date, Elderberry, Fig]

  // Remove elements
  fruits.remove("Banana");
  print("After remove: $fruits"); // [Apple, Grape, Cherry, Date, Elderberry, Fig]

  fruits.removeAt(0);
  print("After removeAt(0): $fruits"); // [Grape, Cherry, Date, Elderberry, Fig]

  String lastFruit = fruits.removeLast();
  print("Removed last: $`lastFruit, Remaining: `$fruits"); // Removed last: Fig, Remaining:
[Grape, Cherry, Date, Elderberry]

  // Querying
  print("Length: ${fruits.length}"); // 4
  print("Is empty: ${fruits.isEmpty}"); // false
  print("Contains Cherry: ${fruits.contains("Cherry")}"); // true
  print("Index of Date: ${fruits.indexOf("Date")}"); // 2

  // Sorting
  fruits.sort();
  print("Sorted: $fruits"); // [Cherry, Date, Elderberry, Grape]

  // Reversing
  print("Reversed: ${fruits.reversed.toList()}"); // [Grape, Elderberry, Date, Cherry]

  // Sublist
  List<String> sub = fruits.sublist(1, 3);
  print("Sublist (index 1 to 2): $sub"); // [Date, Elderberry]

  // Clear
  fruits.clear();
  print("After clear: $fruits"); // []
}
```

Set operations and uniqueness constraints

A `Set` is an unordered collection of unique items. This means a Set cannot contain duplicate elements. Sets are useful when you need to ensure that each element in a collection is distinct.

Set Creation

```
void main() {
  // Literal way to create a Set
  Set<int> uniqueNumbers = {1, 2, 3, 2, 1}; // Duplicates are ignored
  print("Unique numbers: $uniqueNumbers"); // {1, 2, 3}

  // Creating an empty Set
  Set<String> names = {}; // Use {} for empty set, not {} for empty map
  print("Empty names set: $names");

  // Using the Set constructor from a List
  List<int> numbersWithDuplicates = [1, 2, 2, 3, 4, 4, 5];
  Set<int> distinctNumbers = Set<int>.from(numbersWithDuplicates);
  print("Distinct numbers: $distinctNumbers"); // {1, 2, 3, 4, 5}
}
```

Set Operations

Sets support mathematical set operations like union, intersection, and difference.

- **`add(element)`:** Adds an element to the set. Returns `true` if the element was added (i.e., it was not already present), `false` otherwise.
- **`remove(element)`:** Removes the specified element from the set. Returns `true` if the element was present, `false` otherwise.
- **`contains(element)`:** Checks if the set contains the specified element.
- **`union(other)`:** Returns a new set containing all elements from both sets.
- **`intersection(other)`:** Returns a new set containing only the elements common to both sets.
- **`difference(other)`:** Returns a new set containing elements present in this set but not in the `other` set.
- **`clear()`:** Removes all elements from the set.

```
void main() {
  Set<String> setA = {"Apple", "Banana", "Cherry"};
  Set<String> setB = {"Banana", "Date", "Apple"};

  print("Set A: $setA");
  print("Set B: $setB");

  // Add element
  setA.add("Grape");
  print("Set A after adding Grape: $setA"); // {Apple, Banana, Cherry, Grape}

  // Union
  Set<String> unionSet = setA.union(setB);
  print("Union (A U B): $unionSet"); // {Apple, Banana, Cherry, Grape, Date}

  // Intersection
  Set<String> intersectionSet = setA.intersection(setB);
  print("Intersection (A \u2229 B): $intersectionSet"); // {Apple, Banana}

  // Difference
  Set<String> differenceSet = setA.difference(setB);
  print("Difference (A - B): $differenceSet"); // {Cherry, Grape}

  // Check containment
  print("Set A contains Cherry: ${setA.contains("Cherry")}"); // true

  // Remove element
  setA.remove("Cherry");
  print("Set A after removing Cherry: $setA"); // {Apple, Banana, Grape}
}
```

Collection literals and constructors

Dart provides convenient ways to create collections using literals and various constructors.

List Literals

List literals are defined using square brackets `[]`.

```
void main() {
  var numbers = [1, 2, 3]; // List<int>
  var names = <String>["Alice", "Bob"]; // Explicit type annotation
  var mixed = [1, "hello", true]; // List<Object>
  print(numbers);
  print(names);
  print(mixed);
}
```

Set Literals

Set literals are defined using curly braces `{}`. If you create an empty set, you must specify the type, otherwise, Dart will infer it as a `Map`.

```dart
void main() {
  var uniqueIds = {101, 102, 103}; // Set<int>
  var fruits = <String>{"Apple", "Banana"}; // Explicit type annotation
  Set<int> emptySet = {}; // Correct way to create an empty Set
  // var emptyMap = {}; // This creates an empty Map<dynamic, dynamic>
  print(uniqueIds);
  print(fruits);
  print(emptySet);
}
```

Map Literals

Map literals are also defined using curly braces `{}`, but they contain key-value pairs.

```dart
void main() {
  var scores = {
    "Alice": 95,
    "Bob": 88,
  }; // Map<String, int>
  var userInfo = <String, dynamic>{
    "name": "Charlie",
    "age": 30,
    "isActive": true,
  }; // Explicit type annotation
  print(scores);
  print(userInfo);
}
```

Constructors

Collections can also be created using their constructors, which offer more control over initialization.

```dart
void main() {
  // List constructors
  List<int> listFromIterable = List<int>.from([1, 2, 3]);
  List<String> fixedSizeList = List.filled(2, "default");
  List<double> generatedList = List.generate(3, (i) => i * 1.0);
  print("List from iterable: $listFromIterable");
  print("Fixed size list: $fixedSizeList");
  print("Generated list: $generatedList");

  // Set constructors
  Set<int> setFromList = Set<int>.from([1, 2, 2, 3]);
  print("Set from list: $setFromList");

  // Map constructors
  Map<String, int> mapFromEntries = Map.fromEntries([MapEntry("a", 1), MapEntry("b", 2)]);
  print("Map from entries: $mapFromEntries");
}
```

Spread operator and collection-if/for

Dart 2.3 introduced powerful features for creating collections conditionally and with dynamic content: the spread operator (`...` and `...?`) and collection `if` / `for` .

Spread Operator (`...` and `...?`)

The spread operator allows you to insert all elements of a collection into another collection. The null-aware spread operator `...?` can be used if the collection you are spreading might be null.

```
void main() {
  List<int> list1 = [1, 2, 3];
  List<int> list2 = [4, 5, 6];

  // Combine lists using spread operator
  List<int> combinedList = [0, ...list1, ...list2, 7];
  print("Combined List: $combinedList"); // [0, 1, 2, 3, 4, 5, 6, 7]

  // Using spread operator with Sets
  Set<String> set1 = {"A", "B"};
  Set<String> set2 = {"B", "C"};
  Set<String> combinedSet = {"Z", ...set1, ...set2};
  print("Combined Set: $combinedSet"); // {Z, A, B, C}

  // Null-aware spread operator
  List<int>? nullableList;
  List<int> safeCombinedList = [10, ...?nullableList, 20];
  print("Safe Combined List: $safeCombinedList"); // [10, 20]

  nullableList = [100, 200];
  safeCombinedList = [10, ...?nullableList, 20];
  print("Safe Combined List (with value): $safeCombinedList"); // [10, 100, 200, 20]
}
```

Collection `if`

Collection `if` allows you to conditionally include elements or collections in a new collection.

```
void main() {
  bool showExtra = true;
  List<String> items = [
    "Item 1",
    "Item 2",
    if (showExtra) "Extra Item", // Conditionally include "Extra Item"
    "Item 3",
  ];
  print("Items: $items"); // [Item 1, Item 2, Extra Item, Item 3]

  bool isAdmin = false;
  Set<String> permissions = {
    "read",
    "write",
    if (isAdmin) "delete", // "delete" is only included if isAdmin is true
  };
  print("Permissions: $permissions"); // {read, write}
}
```

Collection `for`

Collection `for` allows you to build a collection by iterating over another collection.

```dart
void main() {
  List<int> numbers = [1, 2, 3];

  // Create a new list by transforming elements from another list
  List<int> squaredNumbers = [
    for (var number in numbers) number * number,
  ];
  print("Squared Numbers: $squaredNumbers"); // [1, 4, 9]

  // Create a Set of strings based on a list of objects
  List<Map<String, dynamic>> users = [
    {"name": "Alice", "age": 30},
    {"name": "Bob", "age": 25},
    {"name": "Charlie", "age": 35},
  ];

  Set<String> userNames = {
    for (var user in users) user["name"] as String,
  };
  print("User Names: $userNames"); // {Alice, Bob, Charlie}
}
```

These features make collection creation in Dart very expressive and concise, reducing the need for imperative loops and conditional statements when building new collections.

Maps and Iterable Methods in Dart

Map key-value operations and iteration

A `Map` is a dynamic collection that represents a set of key-value pairs. Keys must be unique, but values can be duplicated. Maps are highly optimized for retrieving a value given its key.

Map Creation

```
void main() {
  // Literal way to create a Map (most common)
  Map<String, int> ages = {
    "Alice": 30,
    "Bob": 25,
    "Charlie": 35,
  };
  print("Ages: $ages");

  // Creating an empty Map
  Map<String, String> capitals = {}; // Explicitly define key and value types
  print("Empty capitals map: $capitals");

  // Using the Map constructor
  Map<int, String> numbersMap = Map();
  numbersMap[1] = "One";
  numbersMap[2] = "Two";
  print("Numbers Map: $numbersMap");

  // Map.from: Creates a new map from an existing map
  Map<String, int> agesCopy = Map.from(ages);
  print("Ages Copy: $agesCopy");

  // Map.fromIterable: Creates a map from an iterable of keys and values
  List<String> names = ["Alice", "Bob", "Charlie"];
  Map<String, int> nameLengths = Map.fromIterable(names, key: (name) => name, value: (name)
=> name.length);
  print("Name Lengths: $nameLengths");
}
```

Map Operations

Dart's `Map` class provides various methods for manipulating key-value pairs.

- **`[key] = value`:** Adds or updates a key-value pair.
- **`[key]`:** Retrieves the value associated with a key. Returns `null` if the key is not found.
- **`putIfAbsent(key, ifAbsent)`:** If `key` is not in the map, adds the key-value pair where the value is the result of calling `ifAbsent`.
- **`remove(key)`:** Removes the key-value pair associated with the key.
- **`containsKey(key)`:** Checks if the map contains the specified key.
- **`containsValue(value)`:** Checks if the map contains the specified value.
- **`length`:** Returns the number of key-value pairs.
- **`isEmpty`, `isNotEmpty`:** Checks if the map is empty.
- **`keys`:** Returns an `Iterable` of all keys.
- **`values`:** Returns an `Iterable` of all values.
- **`clear()`:** Removes all key-value pairs from the map.

```dart
void main() {
  Map<String, String> countries = {
    "USA": "Washington D.C.",
    "France": "Paris",
    "Japan": "Tokyo",
  };
  print("Initial Countries: $countries");

  // Add/Update elements
  countries["Germany"] = "Berlin";
  print("After adding Germany: $countries");

  countries["USA"] = "New York"; // Update existing key
  print("After updating USA: $countries");

  // Retrieve value
  print("Capital of France: ${countries["France"]}"); // Paris
  print("Capital of Italy: ${countries["Italy"]}");   // null

  // putIfAbsent
  countries.putIfAbsent("Spain", () => "Madrid");
  print("After putIfAbsent Spain: $countries");
  countries.putIfAbsent("USA", () => "Some other city"); // Not added, USA already exists
  print("After putIfAbsent USA: $countries");

  // Remove element
  countries.remove("Japan");
  print("After removing Japan: $countries");

  // Querying
  print("Contains key France: ${countries.containsKey("France")}"); // true
  print("Contains value Berlin: ${countries.containsValue("Berlin")}"); // true
  print("Length: ${countries.length}"); // 4

  // Iterating over keys, values, or entries
  print("\nKeys:");
  for (var key in countries.keys) {
    print(key);
  }

  print("\nValues:");
  for (var value in countries.values) {
    print(value);
  }

  print("\nEntries:");
  countries.forEach((key, value) {
    print("$`key: `$value");
  });

  // Clear
  countries.clear();
  print("After clear: $countries"); // {}
}
```

Iterable methods (map, filter, reduce, etc.)

Dart's `Iterable` class (which `List` and `Set` implement) provides a rich set of higher-order methods for transforming, filtering, and aggregating collections in a functional style. These methods are powerful for data manipulation and often lead to more concise and readable code than traditional loops.

`map()`

The `map()` method transforms each element of an iterable into a new form, returning a new `Iterable` containing the transformed elements.

```
void main() {
  List<int> numbers = [1, 2, 3, 4, 5];

  // Square each number
  Iterable<int> squaredNumbers = numbers.map((number) => number * number);
  print("Squared Numbers: ${squaredNumbers.toList()}"); // [1, 4, 9, 16, 25]

  // Convert numbers to strings with a prefix
  Iterable<String> prefixedNumbers = numbers.map((number) => "Num_$number");
  print("Prefixed Numbers: ${prefixedNumbers.toList()}"); // [Num_1, Num_2, Num_3, Num_4,
Num_5]

  List<String> names = ["alice", "bob", "charlie"];
  Iterable<String> capitalizedNames = names.map((name) => name.toUpperCase());
  print("Capitalized Names: ${capitalizedNames.toList()}"); // [ALICE, BOB, CHARLIE]
}
```

`where()` (Filter)

The `where()` method (often referred to as `filter` in other languages) returns a new `Iterable` containing only the elements that satisfy a given condition (predicate function).

```
void main() {
  List<int> numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];

  // Get even numbers
  Iterable<int> evenNumbers = numbers.where((number) => number % 2 == 0);
  print("Even Numbers: ${evenNumbers.toList()}"); // [2, 4, 6, 8, 10]

  List<String> fruits = ["Apple", "Banana", "Cherry", "Avocado"];
  // Get fruits starting with 'A'
  Iterable<String> aFruits = fruits.where((fruit) => fruit.startsWith("A"));
  print("Fruits starting with A: ${aFruits.toList()}"); // [Apple, Avocado]
}
```

`fold()` (Reduce)

The `fold()` method (often referred to as `reduce` in other languages) combines the elements of an iterable into a single value. It takes an initial value and a combining function.

```dart
void main() {
  List<int> numbers = [1, 2, 3, 4, 5];

  // Calculate the sum of all numbers
  int sum = numbers.fold(0, (previousValue, element) => previousValue + element);
  print("Sum: $sum"); // 15

  List<String> words = ["Dart", "is", "fun"];
  // Concatenate words into a single sentence
  String sentence = words.fold("", (previousValue, element) => "$`previousValue
`$element").trim();
  print("Sentence: \"$sentence\""); // "Dart is fun"

  // Calculate product
  int product = numbers.fold(1, (prev, element) => prev * element);
  print("Product: $product"); // 120
}
```

`reduce()`

The `reduce()` method is similar to `fold()`, but it doesn't take an initial value. It combines the elements of an iterable into a single value, starting with the first two elements.

```dart
void main() {
  List<int> numbers = [1, 2, 3, 4, 5];

  // Calculate the sum of all numbers
  int sum = numbers.reduce((value, element) => value + element);
  print("Sum (reduce): $sum"); // 15

  List<String> words = ["Dart", "is", "awesome"];
  String combined = words.reduce((value, element) => "$`value `$element");
  print("Combined words: $combined"); // Dart is awesome

  // Note: reduce will throw an error on an empty list
  // List<int> emptyList = [];
  // emptyList.reduce((value, element) => value + element); // Throws StateError
}
```

Other useful Iterable methods

- **`firstWhere(test, {orElse})`:** Returns the first element that satisfies the given predicate. Throws `StateError` if no element is found and `orElse` is not provided.
- **`lastWhere(test, {orElse})`:** Returns the last element that satisfies the given predicate.
- **`singleWhere(test, {orElse})`:** Returns the single element that satisfies the given predicate. Throws `StateError` if no or more than one element is found.
- **`any(test)`:** Returns `true` if at least one element satisfies the predicate.
- **`every(test)`:** Returns `true` if all elements satisfy the predicate.
- **`take(count)`:** Returns a new iterable containing the first `count` elements.
- **`skip(count)`:** Returns a new iterable skipping the first `count` elements.
- **`toList()`, `toSet()`:** Converts the iterable to a `List` or `Set`.

```dart
void main() {
  List<int> numbers = [10, 20, 30, 40, 50];

  // firstWhere
  int firstOver30 = numbers.firstWhere((n) => n > 30);
  print("First number over 30: $firstOver30"); // 40

  // firstWhere with orElse
  int firstOver100 = numbers.firstWhere((n) => n > 100, orElse: () => -1);
  print("First number over 100: $firstOver100"); // -1

  // any
  bool hasEven = numbers.any((n) => n % 2 == 0);
  print("Has any even number: $hasEven"); // true

  // every
  bool allEven = numbers.every((n) => n % 2 == 0);
  print("All numbers are even: $allEven"); // false

  // take
  List<int> firstThree = numbers.take(3).toList();
  print("First three: $firstThree"); // [10, 20, 30]

  // skip
  List<int> skipTwo = numbers.skip(2).toList();
  print("Skip first two: $skipTwo"); // [30, 40, 50]
}
```

These iterable methods promote a more declarative and functional programming style, making code often more readable and less prone to errors compared to traditional loop-based approaches. They are heavily used in Flutter for efficient UI updates and data processing.

Object-Oriented Programming in Dart

Object-Oriented Programming (OOP) is a programming paradigm based on the concept of "objects", which can contain data and code: data in the form of fields (often known as attributes or properties), and code in the form of procedures (often known as methods). Dart is an object-oriented language, and almost everything in Dart is an object, including numbers, functions, and `null`.

Class definition and instantiation

A **class** is a blueprint for creating objects (a particular data structure), providing initial values for state (member variables or attributes), and implementations of behavior (member functions or methods).

Instantiation is the process of creating an object (an instance) of a class.

Class Definition

To define a class in Dart, you use the `class` keyword, followed by the class name (conventionally PascalCase).

```
// Define a simple class named \'Dog\'
class Dog {
  // Instance variables (properties)
  String name;
  String breed;
  int age;

  // Constructor (more on this later)
  Dog(this.name, this.breed, this.age);

  // Instance method (behavior)
  void bark() {
    print("$name says Woof!");
  }

  void describe() {
    print("$`name is a `$age-year-old $breed.");
  }
}
```

In this `Dog` class: * `name`, `breed`, and `age` are **instance variables** (or fields/properties). Each `Dog` object will have its own set of these values. * `Dog(this.name, this.breed, this.age)` is a **constructor**, a special method used to create new instances of the class and initialize their properties. * `bark()` and `describe()` are **instance methods**. These methods define the behaviors that `Dog` objects can perform.

Instantiation

To create an object (an instance) of a class, you use the `new` keyword (which is optional in Dart 2.0 and later) followed by the class name and constructor arguments.

```
void main() {
  // Instantiating objects of the Dog class
  Dog myDog = Dog("Buddy", "Golden Retriever", 3);
  Dog anotherDog = Dog("Lucy", "Labrador", 5);

  // Accessing instance variables
  print("My dog\'s name: ${myDog.name}");
  print("Another dog\'s breed: ${anotherDog.breed}");

  // Calling instance methods
  myDog.bark();
  anotherDog.describe();

  // You can also omit \'new\' keyword
  var thirdDog = Dog("Max", "German Shepherd", 2);
  thirdDog.bark();
}
```

Each object (`myDog`, `anotherDog`, `thirdDog`) is an independent instance of the `Dog` class, with its own unique set of `name`, `breed`, and `age` values.

Constructors (default, named, factory)

Constructors are special methods used to create and initialize objects of a class. Dart provides several types of constructors to suit different initialization needs.

Default Constructor

If you don\'t declare a constructor, Dart provides a default (unnamed, no-argument) constructor. This default constructor calls the no-argument constructor in the superclass.

```
class Point {
  double x, y;

  // If you don\'t define a constructor, Dart provides a default one:
  // Point() : x = 0.0, y = 0.0;

  Point(this.x, this.y);
}

void main() {
  var p1 = Point(10.0, 20.0); // Using the defined constructor
  print("Point 1: ($`{p1.x}, `${p1.y})");

  // If no constructor was defined, you could do:
  // var p2 = Point(); // Calls the default constructor
}
```

Named Constructors

Named constructors allow you to create multiple constructors for a class, each with a descriptive name. This is useful when you want to provide different ways to create an object.

```
class Person {
  String name;
  int age;

  // Default constructor
  Person(this.name, this.age);

  // Named constructor for creating a Person from a birth year
  Person.fromBirthYear(String name, int birthYear) :
    this(name, DateTime.now().year - birthYear);

  // Named constructor for creating an anonymous Person
  Person.anonymous() : this("Anonymous", 0);

  void introduce() {
    print("Hello, my name is $`name and I am `$age years old.");
  }
}

void main() {
  var person1 = Person("Alice", 30);
  person1.introduce();

  var person2 = Person.fromBirthYear("Bob", 1995);
  person2.introduce();

  var person3 = Person.anonymous();
  person3.introduce();
}
```

Named constructors are very common in Flutter, where widgets often have named constructors for different use cases (e.g., `Text.rich`, `Container.builder`).

Factory Constructors

Factory constructors are used when you want a constructor to return an instance of a class that is not necessarily a new instance of its own class. This can be useful for:

- Returning an existing instance from a cache.
- Returning an instance of a subclass.
- Complex initialization logic that cannot be handled by a regular constructor.

Factory constructors cannot access `this`.

```
class Logger {
  final String name;
  static final Map<String, Logger> _cache = <String, Logger>{};

  // Factory constructor
  factory Logger(String name) {
    if (_cache.containsKey(name)) {
      return _cache[name]!;
    } else {
      final logger = Logger._internal(name);
      _cache[name] = logger;
      return logger;
    }
  }

  // Private named constructor for internal use
  Logger._internal(this.name);

  void log(String message) {
    print("[" + name + "] " + message);
  }
}

void main() {
  var logger1 = Logger("MyApp");
  logger1.log("First message from MyApp.");

  var logger2 = Logger("MyApp"); // Returns the same instance as logger1
  logger2.log("Second message from MyApp.");

  print(identical(logger1, logger2)); // true

  var logger3 = Logger("Auth");
  logger3.log("Message from Auth.");
  print(identical(logger1, logger3)); // false
}
```

In this example, the `Logger` factory constructor ensures that only one `Logger` instance exists for a given name, effectively implementing a singleton pattern for loggers.

Instance variables and methods

Instance Variables (Fields/Properties)

Instance variables hold the state of an object. Each instance of a class has its own copy of these variables.

```dart
class Car {
  String make;    // Instance variable
  String model;   // Instance variable
  int year;       // Instance variable
  double _speed = 0; // Private instance variable (conventionally prefixed with _)

  Car(this.make, this.model, this.year);

  // Getter for _speed
  double get speed => _speed;

  // Setter for _speed
  set speed(double newSpeed) {
    if (newSpeed >= 0) {
      _speed = newSpeed;
    } else {
      print("Speed cannot be negative.");
    }
  }

  void accelerate(double amount) {
    _speed += amount;
    print("Accelerating. Current speed: $_speed");
  }
}

void main() {
  var myCar = Car("Toyota", "Camry", 2020);
  print("My car: $`{myCar.make} `${myCar.model} (${myCar.year})");

  myCar.accelerate(50);
  myCar.speed = 100; // Using the setter
  print("My car\'s speed: ${myCar.speed}"); // Using the getter

  myCar.speed = -10; // Will print error message
}
```

Dart automatically generates default getters and setters for all public instance variables. You can define custom getters and setters to control how properties are accessed and modified, as shown with `_speed`.

Instance Methods

Instance methods define the behaviors that objects of a class can perform. They operate on the instance variables of the object.

```dart
class Calculator {
  double _result = 0.0;

  double get result => _result;

  void add(double a, double b) {
    _result = a + b;
  }

  void subtract(double a, double b) {
    _result = a - b;
  }

  void multiply(double a, double b) {
    _result = a * b;
  }

  void divide(double a, double b) {
    if (b != 0) {
      _result = a / b;
    } else {
      print("Error: Division by zero.");
      _result = double.nan; // Not a Number
    }
  }

  void clear() {
    _result = 0.0;
  }
}

void main() {
  var calc = Calculator();
  calc.add(10, 5);
  print("Add result: ${calc.result}");

  calc.subtract(calc.result, 3);
  print("Subtract result: ${calc.result}");

  calc.multiply(calc.result, 2);
  print("Multiply result: ${calc.result}");

  calc.divide(calc.result, 0);
  print("Divide result: ${calc.result}");

  calc.clear();
  print("Cleared result: ${calc.result}");
}
```

Static members and class-level functionality

Static members (variables and methods) belong to the class itself, not to any specific instance of the class. You access them directly on the class name.

Static Variables

Static variables are shared across all instances of a class. They are initialized once, when the class is first accessed.

```
class MathUtils {
  static const double PI = 3.1415926535;
  static int calculationCount = 0;

  static double circleArea(double radius) {
    calculationCount++;
    return PI * radius * radius;
  }

  static double circleCircumference(double radius) {
    calculationCount++;
    return 2 * PI * radius;
  }
}

void main() {
  // Accessing static variables directly on the class
  print("Value of PI: ${MathUtils.PI}");
  print("Initial calculation count: ${MathUtils.calculationCount}");

  // Calling static methods
  double area = MathUtils.circleArea(5.0);
  print("Area of circle: $area");

  double circumference = MathUtils.circleCircumference(10.0);
  print("Circumference of circle: $circumference");

  // The calculationCount is shared and increments with each static method call
  print("Total calculations: ${MathUtils.calculationCount}");

  // You cannot access static members on an instance
  // var utils = MathUtils(); // Error: The class \'MathUtils\' doesn\'t have a default
constructor.
  // print(utils.PI); // Error
}
```

Static variables are often used for constants that are relevant to the class as a whole (like `PI` in `MathUtils`) or for shared state that doesn\'t depend on individual object instances.

Static Methods

Static methods can be called directly on the class without creating an instance of the class. They cannot access instance variables or `this` because they don\'t operate on a specific object instance.

```
class StringUtils {
  static String capitalize(String text) {
    if (text.isEmpty) return text;
    return text[0].toUpperCase() + text.substring(1).toLowerCase();
  }

  static bool isPalindrome(String text) {
    String cleanedText = text.replaceAll(RegExp(r\'[^a-zA-Z0-9]\'), \'\').toLowerCase();
    return cleanedText == cleanedText.split(\'\').reversed.join(\'\');
  }
}

void main() {
  print("Capitalized: ${StringUtils.capitalize("hello world")}"); // Hello world
  print("Is \'madam\' palindrome: ${StringUtils.isPalindrome("madam")}"); // true
  print("Is \'A man, a plan, a canal: Panama\' palindrome: ${StringUtils.isPalindrome("A
man, a plan, a canal: Panama")}"); // true
}
```

Static methods are ideal for utility functions that perform operations related to the class but don\'t require access to instance-specific data. They are often used for helper functions or factory methods that don\'t need an object\'s state.

Encapsulation and Polymorphism

Encapsulation and Polymorphism are two fundamental pillars of Object-Oriented Programming (OOP) that contribute significantly to code organization, maintainability, and flexibility.

Encapsulation and access modifiers

Encapsulation is the bundling of data (attributes) and methods (functions) that operate on the data into a single unit, or class. It also involves restricting direct access to some of an object\'s components, which means that the internal representation of an object is hidden from the outside world. This is typically achieved through access modifiers.

In Dart, access control is simpler than in some other languages like Java or C++. Dart does not have keywords like `public`, `private`, or `protected`. Instead, it uses a convention based on identifiers:

- **Public (default):** All declarations (classes, variables, functions, methods) are public by default. They can be accessed from anywhere.
- **Private (library-level):** To make a declaration private to its library (which is typically a single `.dart` file), you prefix its identifier with an underscore (`_`). This means a private member is accessible within the same `.dart` file where it\'s declared, but not from other files.

Example of Encapsulation

Consider a `BankAccount` class. We want to ensure that the `_balance` can only be modified through controlled methods like `deposit` and `withdraw`, preventing direct external manipulation.

```
class BankAccount {
  // Private instance variable (library-private)
  double _balance;

  // Constructor
  BankAccount(this._balance);

  // Public getter to read the balance
  double get balance => _balance;

  // Public method to deposit money
  void deposit(double amount) {
    if (amount > 0) {
      _balance += amount;
      print("Deposited: $`amount. New balance: `$_balance");
    } else {
      print("Deposit amount must be positive.");
    }
  }

  // Public method to withdraw money
  void withdraw(double amount) {
    if (amount > 0 && _balance >= amount) {
      _balance -= amount;
      print("Withdrew: $`amount. New balance: `$_balance");
    } else if (amount <= 0) {
      print("Withdrawal amount must be positive.");
    } else {
      print("Insufficient balance. Current balance: $_balance");
    }
  }

  // A private method (only accessible within this file)
  void _logTransaction(String type, double amount) {
    print("Logging: $`type of `$amount");
  }
}

void main() {
  var account = BankAccount(1000.0);

  print("Initial balance: ${account.balance}");

  account.deposit(500.0);
  account.withdraw(200.0);
  account.withdraw(1500.0); // Insufficient balance

  // Direct access to _balance is prevented from outside this file
  // account._balance = 5000.0; // Error: The setter \'_balance=\' isn\'t defined.

  // Calling a private method from outside is not allowed
  // account._logTransaction("Test", 100.0); // Error: The method \'_logTransaction\' isn\'t
defined.
}
```

In this example, `_balance` is encapsulated. External code can only interact with the balance through the `deposit` and `withdraw` methods, which enforce business rules (e.g., positive amounts, sufficient funds). The `_logTransaction` method is also private, indicating it\'s an internal helper function.

Benefits of Encapsulation:

- **Data Hiding:** Protects an object\'s internal state from unauthorized access or modification.

- **Maintainability:** Changes to the internal implementation of a class do not affect external code as long as the public interface remains the same.
- **Flexibility:** Allows for controlled modification of data through methods, enabling validation and other logic.
- **Modularity:** Makes classes self-contained and easier to understand and manage.

Method overriding and polymorphism

Polymorphism (meaning "many forms") is the ability of an object to take on many forms. In OOP, it allows objects of different classes to be treated as objects of a common type. This is often achieved through **method overriding**.

Method Overriding occurs when a subclass provides its own implementation of a method that is already defined in its superclass. The overridden method in the subclass must have the same name, return type, and parameter list as the method in the superclass.

Example of Polymorphism and Method Overriding

Let\'s consider a hierarchy of `Shape` classes.

```
// Base class
class Shape {
  String color;

  Shape(this.color);

  // Method to be overridden by subclasses
  void draw() {
    print("Drawing a $color shape.");
  }

  double calculateArea() {
    print("Area calculation not implemented for generic shape.");
    return 0.0;
  }
}

// Subclass Circle
class Circle extends Shape {
  double radius;

  Circle(String color, this.radius) : super(color);

  @override // Annotation to indicate method overriding
  void draw() {
    print("Drawing a $`color circle with radius `$radius.");
  }

  @override
  double calculateArea() {
    return 3.14159 * radius * radius;
  }
}

// Subclass Rectangle
class Rectangle extends Shape {
  double width;
  double height;

  Rectangle(String color, this.width, this.height) : super(color);

  @override
  void draw() {
    print("Drawing a $`color rectangle with width `$width and height $height.");
  }

  @override
  double calculateArea() {
    return width * height;
  }
}

void main() {
  // Creating objects of different shape types
  Shape genericShape = Shape("transparent");
  Circle redCircle = Circle("red", 5.0);
  Rectangle blueRectangle = Rectangle("blue", 4.0, 6.0);

  // Polymorphism in action: treating different objects as their common base type
  List<Shape> shapes = [genericShape, redCircle, blueRectangle];

  for (var shape in shapes) {
    shape.draw(); // Calls the overridden draw() method for each specific shape
    print("Area: ${shape.calculateArea().toStringAsFixed(2)}\n\n");
  }

  // You can also assign a subclass instance to a superclass variable
  Shape myShape = Circle("green", 7.0);
  myShape.draw(); // Drawing a green circle with radius 7.0.
```

```
  print("Area: ${myShape.calculateArea().toStringAsFixed(2)}");
}
```

In this example:

1. `Shape` is the base class with `draw()` and `calculateArea()` methods.
2. `Circle` and `Rectangle` are subclasses that `extend` `Shape`.
3. Both `Circle` and `Rectangle` **override** the `draw()` and `calculateArea()` methods to provide their specific implementations.
4. In `main()`, a `List<Shape>` is created. This demonstrates polymorphism: objects of `Circle` and `Rectangle` types are treated as `Shape` objects.
5. When `shape.draw()` or `shape.calculateArea()` is called within the loop, Dart\'s runtime determines the actual type of the object and executes the appropriate overridden method. This is known as **dynamic dispatch**.

Benefits of Polymorphism:

- **Flexibility and Extensibility:** New classes can be added to the system without modifying existing code, as long as they adhere to the common interface (base class).
- **Code Reusability:** Common behavior can be defined in a base class and specialized in subclasses.
- **Simplified Code:** Allows you to write generic code that works with objects of different types, reducing complexity and improving readability.
- **Decoupling:** Reduces dependencies between different parts of the system, making it easier to manage and test.

Encapsulation and polymorphism, along with inheritance and abstraction, are cornerstones of OOP that enable the creation of robust, scalable, and maintainable software systems.

Inheritance, Abstract classes

Inheritance is a fundamental concept in Object-Oriented Programming (OOP) that allows a class to inherit properties and behaviors from another class. This promotes code reusability and establishes a hierarchical relationship between classes. Abstract classes, on the other hand, provide a blueprint for other classes and cannot be instantiated directly.

Single inheritance and extends keyword

Dart supports single inheritance, meaning a class can inherit from only one direct superclass (parent class). The `extends` keyword is used to establish this inheritance relationship.

When a class `extends` another class, it inherits all the public and protected members (methods and variables) of the superclass. The subclass can then add its own new members or override inherited members to provide specialized implementations.

Example of Single Inheritance

Let's define a base class `Animal` and then a subclass `Dog` that extends `Animal`.

```
// Superclass (Parent Class)
class Animal {
  String name;
  int age;

  Animal(this.name, this.age);

  void eat() {
    print("$name is eating.");
  }

  void sleep() {
    print("$name is sleeping.");
  }
}

// Subclass (Child Class) extending Animal
class Dog extends Animal {
  String breed;

  // Constructor for Dog. It calls the superclass constructor using super().
  Dog(String name, int age, this.breed) : super(name, age);

  // New method specific to Dog
  void bark() {
    print("$`name (`${breed}) says Woof!");
  }

  // Overriding an inherited method
  @override
  void eat() {
    print("$name is happily munching on dog food.");
  }
}

void main() {
  var myAnimal = Animal("Leo", 5);
  myAnimal.eat();
  myAnimal.sleep();

  print("\n");

  var myDog = Dog("Buddy", 3, "Golden Retriever");
  myDog.eat();    // Calls the overridden eat() method in Dog
  myDog.sleep();  // Calls the inherited sleep() method from Animal
  myDog.bark();   // Calls the new method in Dog

  // Accessing inherited properties
  print("Dog's name: ${myDog.name}");
  print("Dog's age: ${myDog.age}");
}
```

In this example: * `Animal` is the superclass, defining common properties (`name`, `age`) and behaviors (`eat`, `sleep`). * `Dog` is the subclass, inheriting from `Animal`. It adds its own property (`breed`) and method (`bark`). * The `Dog` constructor uses `super(name, age)` to pass `name` and `age` to the

`Animal` class's constructor. * The `eat()` method is overridden in `Dog` to provide a more specific behavior for dogs.

Benefits of Inheritance: * **Code Reusability:** Common code can be placed in a superclass and reused by multiple subclasses. * **Maintainability:** Changes to common behavior only need to be made in one place (the superclass). * **Extensibility:** New subclasses can be easily added without modifying existing code. * **Polymorphism:** Allows objects of different subclasses to be treated as objects of their common superclass.

Abstract classes and abstract methods

An **abstract class** is a class that cannot be instantiated directly. It serves as a blueprint for other classes and often contains abstract methods. Abstract classes are declared using the `abstract` keyword.

An **abstract method** is a method declared in an abstract class without an implementation. Subclasses that extend an abstract class must provide an implementation for all its abstract methods, unless the subclass itself is abstract.

Characteristics of Abstract Classes:

- Cannot be instantiated using `new`.
- Can contain both abstract and non-abstract (concrete) methods.
- Can have instance variables, constructors, and static members.
- Subclasses must implement all abstract methods, or they must also be declared abstract.

Example of Abstract Class and Methods

Let's create an abstract `Shape` class with an abstract `calculateArea()` method.

```
// Abstract Superclass
abstract class Shape {
  String color;

  Shape(this.color);

  // Abstract method (no implementation)
  double calculateArea();

  // Concrete method (with implementation)
  void describe() {
    print("This is a $color shape.");
  }
}

// Concrete Subclass Circle
class Circle extends Shape {
  double radius;

  Circle(String color, this.radius) : super(color);

  @override
  double calculateArea() {
    return 3.14159 * radius * radius;
  }

  void draw() {
    print("Drawing a $`color circle with radius `$radius.");
  }
}

// Concrete Subclass Rectangle
class Rectangle extends Shape {
  double width;
  double height;

  Rectangle(String color, this.width, this.height) : super(color);

  @override
  double calculateArea() {
    return width * height;
  }

  void draw() {
    print("Drawing a $`color rectangle with width `$width and height $height.");
  }
}

void main() {
  // Shape myShape = Shape("green"); // Error: Abstract classes can't be instantiated.

  var circle = Circle("red", 5.0);
  circle.describe();
  print("Circle Area: ${circle.calculateArea().toStringAsFixed(2)}");
  circle.draw();

  print("\n");

  var rectangle = Rectangle("blue", 4.0, 6.0);
  rectangle.describe();
  print("Rectangle Area: ${rectangle.calculateArea().toStringAsFixed(2)}");
  rectangle.draw();

  // Polymorphism with abstract class
  List<Shape> shapes = [circle, rectangle];
  for (var s in shapes) {
    s.describe();
    print("Area: ${s.calculateArea().toStringAsFixed(2)}");
  }
}
```

In this example: * `Shape` is an `abstract` class with an abstract method `calculateArea()` and a concrete method `describe()`. * `Circle` and `Rectangle` `extend` `Shape` and are forced to implement `calculateArea()`. * You cannot create an instance of `Shape` directly.

When to use Abstract Classes: * When you want to define a common interface and some common behavior for a group of related classes, but you don't want to allow direct instantiation of the base class. * When you want to enforce that certain methods must be implemented by subclasses. * To provide a partial implementation for a set of related classes.

Super keyword and parent class access

The `super` keyword in Dart is used to refer to the immediate parent class (superclass). It is primarily used in two contexts:

1. **Calling the superclass constructor:** In a subclass constructor, `super()` is used to invoke the constructor of the superclass. This must be the first operation in the initializer list of the subclass constructor.
2. **Accessing superclass members:** To call a method or access a property defined in the superclass that might have been overridden in the subclass.

Calling Superclass Constructor

```
class Vehicle {
  String brand;

  Vehicle(this.brand) {
    print("Vehicle constructor called for brand: $brand");
  }

  void drive() {
    print("Vehicle is driving.");
  }
}

class Car extends Vehicle {
  String model;

  Car(String brand, this.model) : super(brand) {
    print("Car constructor called for model: $model");
  }

  @override
  void drive() {
    super.drive(); // Call the superclass's drive method
    print("Car is driving on the road.");
  }
}

void main() {
  var myCar = Car("Toyota", "Camry");
  // Output:
  // Vehicle constructor called for brand: Toyota
  // Car constructor called for model: Camry

  myCar.drive();
  // Output:
  // Vehicle is driving.
  // Car is driving on the road.
}
```

In the `Car` constructor, `super(brand)` ensures that the `Vehicle` class's constructor is called and its `brand` property is initialized before the `Car` class's own initialization.

Accessing Superclass Members

When a subclass overrides a method from its superclass, you can still call the superclass's version of that method using `super.methodName()`.

```dart
class Person {
  String name;

  Person(this.name);

  void greet() {
    print("Hello, my name is $name.");
  }
}

class Student extends Person {
  String studentId;

  Student(String name, this.studentId) : super(name);

  @override
  void greet() {
    super.greet(); // Call the parent's greet method
    print("I am a student with ID: $studentId.");
  }

  void study() {
    print("$name is studying.");
  }
}

void main() {
  var student = Student("Alice", "S12345");
  student.greet();
  student.study();
}
```

Here, `super.greet()` in the `Student` class's `greet` method allows the `Person` class's greeting to be printed before the `Student`'s specific message. This is a common pattern for extending behavior rather than completely replacing it.

Using `super` effectively is key to building robust and well-structured class hierarchies in Dart, allowing for proper initialization and extension of inherited functionality.

Interfaces, Mixins, and Method Overriding

Dart, unlike some other object-oriented languages, does not have a dedicated `interface` keyword. Instead, every class implicitly defines an interface. This means any class can be used as an interface. Mixins provide a way to reuse code across class hierarchies, offering a form of multiple inheritance of implementation. Method overriding, as discussed briefly before, is a core concept for achieving polymorphism.

Interfaces and implements keyword

In Dart, a class can implement one or more interfaces. When a class `implements` an interface, it must provide a concrete implementation for all the methods and instance variables defined in that interface. This ensures that the implementing class adheres to a specific contract.

Example of Interfaces

Let's define a `Walkable` interface (implicitly, by defining a class) and then have `Dog` and `Person` classes implement it.

```
// Implicit Interface: Any class can be an interface
class Walkable {
  void walk() {
    // This method must be implemented by classes that implement Walkable
    print("Default walk behavior (should be overridden).");
  }
}

class Dog implements Walkable {
  String name;

  Dog(this.name);

  @override
  void walk() {
    print("$name is walking on four legs.");
  }

  void bark() {
    print("$name says Woof!");
  }
}

class Person implements Walkable {
  String name;

  Person(this.name);

  @override
  void walk() {
    print("$name is walking on two legs.");
  }

  void greet() {
    print("Hello, I am $name.");
  }
}

void main() {
  Walkable myDog = Dog("Buddy");
  Walkable myPerson = Person("Alice");

  myDog.walk();
  myPerson.walk();

  // You can only call methods defined in the interface when using the interface type
  // myDog.bark(); // Error: The method \'bark\' isn\'t defined for the type \'Walkable\'.

  // To access specific methods, you can cast or use \'is\' and \'as\'
  if (myDog is Dog) {
    myDog.bark();
  }
  (myPerson as Person).greet();

  List<Walkable> walkers = [Dog("Max"), Person("Bob")];
  for (var walker in walkers) {
    walker.walk();
  }
}
```

In this example: * `Walkable` is a regular class, but it serves as an interface. Any class that `implements` `Walkable` must provide an implementation for the `walk()` method. * `Dog` and `Person` both `implement` `Walkable`, providing their own specific `walk()` behavior. * Polymorphism is demonstrated by treating `Dog` and `Person` objects as `Walkable` types in the `walkers` list.

When to use Interfaces: * To define a contract that multiple unrelated classes must adhere to. * To achieve polymorphism when classes don't share a common inheritance hierarchy. * To enable dependency inversion principle (DIP) by programming to an interface rather than an implementation.

Mixins and multiple inheritance patterns

Dart does not support multiple inheritance of classes (a class can only `extend` one superclass). However, it provides **mixins** as a way to reuse a class's code in multiple class hierarchies. A mixin is a regular class that is used with the `with` keyword to add its capabilities to another class.

Characteristics of Mixins:

- A mixin cannot be instantiated directly.
- A mixin cannot have a constructor (prior to Dart 2.1, but now allowed if it's a `mixin` declaration).
- A mixin can contain instance variables, methods, and even abstract methods.
- A class can use multiple mixins.

Example of Mixins

Let's create some mixins for `Flyable` and `Swimmable` behaviors.

```dart
// Define a mixin for flying behavior
mixin Flyable {
  void fly() {
    print("I can fly!");
  }
}

// Define a mixin for swimming behavior
mixin Swimmable {
  void swim() {
    print("I can swim!");
  }
}

class Bird with Flyable {
  String name;
  Bird(this.name);

  void chirp() {
    print("$name is chirping.");
  }
}

class Duck with Flyable, Swimmable {
  String name;
  Duck(this.name);

  void quack() {
    print("$name is quacking.");
  }
}

class Fish with Swimmable {
  String name;
  Fish(this.name);

  void bubble() {
    print("$name is blowing bubbles.");
  }
}

void main() {
  var eagle = Bird("Eagle");
  eagle.fly();
  eagle.chirp();

  var donald = Duck("Donald");
  donald.fly();
  donald.swim();
  donald.quack();

  var nemo = Fish("Nemo");
  nemo.swim();
  nemo.bubble();
}
```

In this example: * `Flyable` and `Swimmable` are defined as `mixin`s. * `Bird` uses `with Flyable` to gain flying capabilities. * `Duck` uses `with Flyable, Swimmable` to gain both flying and swimming capabilities. * `Fish` uses `with Swimmable` to gain swimming capabilities.

Mixins allow you to compose classes by adding predefined behaviors without being constrained by a strict single-inheritance hierarchy. This is particularly powerful in Flutter, where mixins are used extensively for adding functionalities to widgets (e.g., `SingleTickerProviderStateMixin`).

When to use Mixins: * To share code between unrelated classes that need similar functionalities. * To add capabilities to a class without extending it (which would limit its ability to extend another class). * To implement cross-cutting concerns.

Override annotations and method signatures

As seen in previous sections, **method overriding** is when a subclass provides its own implementation of a method that is already defined in its superclass or an interface it implements. The `@override` annotation is not strictly required but is highly recommended.

`@override` Annotation

The `@override` annotation is a metadata annotation that tells the Dart compiler that the annotated method is intended to override a method from a superclass or an interface. It provides several benefits:

- **Compile-time checking:** If you misspell the method name, or if the method signature (return type, parameter types, number of parameters) doesn't match the superclass method, the compiler will issue an error. This helps catch bugs early.
- **Readability:** It clearly indicates to other developers (and your future self) that the method is overriding an inherited one, improving code clarity.

Method Signatures

For a method to successfully override another, their signatures must match. A method signature includes:

- **Method Name:** Must be identical.
- **Return Type:** Must be the same or a subtype (covariant return type, more advanced topic).
- **Parameter List:**
  - **Number of parameters:** Must be the same.
  - **Types of parameters:** Must be the same or a supertype (contravariant parameter type, more advanced topic).
  - **Names of parameters:** Can be different for positional parameters, but must match for named parameters.

```dart
class Animal {
  void makeSound() {
    print("Animal makes a sound.");
  }

  void eat(String food) {
    print("Animal eats $food.");
  }
}

class Cat extends Animal {
  @override
  void makeSound() {
    print("Meow!");
  }

  @override
  void eat(String food) {
    print("Cat eats $food with grace.");
  }

  // This would cause a compile-time error if @override was used:
  // @override
  // void eat(String food, int quantity) { // Incorrect number of parameters
  //   print("Cat eats $`quantity of `$food.");
  // }

  // This would also cause an error:
  // @override
  // String makeSound() { // Incorrect return type
  //   return "Meow!";
  // }
}

void main() {
  Animal myAnimal = Animal();
  myAnimal.makeSound(); // Animal makes a sound.
  myAnimal.eat("grass"); // Animal eats grass.

  Cat myCat = Cat();
  myCat.makeSound(); // Meow!
  myCat.eat("fish"); // Cat eats fish with grace.

  Animal polymorphicAnimal = Cat();
  polymorphicAnimal.makeSound(); // Meow! (runtime polymorphism)
}
```

In this example, `Cat` correctly overrides `makeSound()` and `eat()` from `Animal`. The commented-out examples show how the `@override` annotation helps catch signature mismatches at compile time, preventing subtle bugs that might otherwise only appear at runtime.

Always use the `@override` annotation when you intend to override a member. It's a good practice that improves code quality and helps prevent errors.

Null Safety & Type System

Dart introduced null safety in version 2.12, making the language soundly null safe. This means that variables, by default, cannot contain `null` unless you explicitly declare them as nullable. Null safety

helps you prevent null reference errors (also known as null pointer exceptions), a common source of bugs in many programming languages.

Understanding null safety fundamentals

At its core, null safety in Dart aims to eliminate null reference errors at compile time. This is achieved by making the type system aware of nullability. When a variable is declared, the compiler knows whether it can hold a `null` value or not.

Sound Null Safety

Dart's null safety is **sound**. This means:

- **Compile-time guarantees:** If the static analysis (compiler) determines that something is not null, then it can never be null at runtime. This is a strong guarantee that prevents null errors.
- **Optimized code:** The compiler can produce smaller and faster code because it doesn't need to add runtime checks for nullability where it knows a value cannot be null.

Nullable vs non-nullable types

By default, all types in Dart are **non-nullable**. This means a variable of a non-nullable type must always contain a valid value and cannot be `null`.

To make a type **nullable**, you append a question mark (`?`) to the type name.

```
void main() {
  // Non-nullable types (default)
  String name = "Alice";
  int age = 30;
  bool isActive = true;

  // name = null; // Compile-time error: A value of type \'Null\' can\'t be assigned to a
variable of type \'String\'.

  // Nullable types (explicitly declared with \'?\')
  String? middleName; // Can be null
  int? numberOfChildren = 2; // Can be null, currently 2
  double? temperature = null; // Can be null, currently null

  middleName = "Marie"; // Assign a non-null value
  print("Middle Name: $middleName");

  middleName = null; // Assign null back
  print("Middle Name: $middleName");

  // You must handle nullable types before using them as non-nullable
  // print(middleName.length); // Compile-time error: The property \'length\' can\'t be
unconditionally accessed because the receiver can be \'null\'.
}
```

Null-aware operators (?, ??, ??=, ?.)

Dart provides several null-aware operators to safely work with nullable types and handle potential `null` values concisely.

`?` (Nullable type declaration)

As seen above, `?` is used to declare a type as nullable.

`??` (Null-coalescing operator)

The null-coalescing operator `??` provides a default value if an expression evaluates to `null`.

```
void main() {
  String? firstName;
  String lastName = "Doe";

  String fullName = (firstName ?? "John") + " " + lastName;
  print("Full Name: $fullName"); // John Doe

  firstName = "Jane";
  fullName = (firstName ?? "John") + " " + lastName;
  print("Full Name: $fullName"); // Jane Doe

  int? value;
  int result = value ?? 0; // If value is null, result is 0
  print("Result: $result"); // 0
}
```

`??=` (Null-coalescing assignment operator)

The null-coalescing assignment operator `??=` assigns a value to a variable only if that variable is currently `null`.

```
void main() {
  String? message;

  message ??= "Default message";
  print("Message: $message"); // Default message

  message ??= "Another message"; // Will not assign, as message is not null
  print("Message: $message"); // Default message

  int? count;
  count ??= 10;
  print("Count: $count"); // 10
}
```

`?.` (Null-aware access operator)

The null-aware access operator `?.` allows you to safely access properties or call methods on an object that might be `null`. If the object is `null`, the expression short-circuits and evaluates to `null`.

```dart
class User {
  String? name;
  User(this.name);
}

void main() {
  User? user1 = User("Alice");
  User? user2; // This user is null

  print("User1 name length: ${user1?.name?.length}"); // Accesses length, prints 5
  print("User2 name length: ${user2?.name?.length}"); // Short-circuits, prints null

  String? userName1 = user1?.name; // userName1 is \'Alice\'
  String? userName2 = user2?.name; // userName2 is null
  print("User Name 1: $userName1");
  print("User Name 2: $userName2");

  // Using it with method calls
  user1?.name = "Bob"; // Safely assigns if user1 is not null
  user2?.name = "Charlie"; // Does nothing if user2 is null
  print("User1 name after assignment: ${user1?.name}");
  print("User2 name after assignment: ${user2?.name}");
}
```

`!` (Null assertion operator)

The null assertion operator `!` tells the compiler that you are certain that a nullable expression is not `null` at runtime. If the expression *is* `null` at runtime, a `NullThrownError` will be thrown. Use this operator with caution and only when you are absolutely sure the value won't be null.

```dart
void main() {
  String? nullableString = "Hello";
  String nonNullableString = nullableString!; // Asserting it\'s not null
  print(nonNullableString); // Hello

  String? anotherNullableString; // This is null
  try {
    String anotherNonNullableString = anotherNullableString!; // Throws NullThrownError
    print(anotherNonNullableString);
  } catch (e) {
    print("Caught error: $e"); // Caught error: Null check operator used on a null value
  }
}
```

Late initialization and lazy evaluation

The `late` keyword, introduced with null safety, allows you to declare a non-nullable variable that is initialized later. This is particularly useful for:

1. **Variables that will be initialized before use:** When a variable cannot be initialized at its declaration point (e.g., in a constructor body or `initState` in Flutter), but you guarantee it will be non-null before its first access.
2. **Lazy initialization:** When the initialization of a variable is expensive, and you want to defer it until the variable is actually used.

```dart
// Case 1: Initialized before use
late String description;

void setupDescription() {
  description = "This description is set up later.";
}

// Case 2: Lazy initialization
late final String expensiveData = _fetchExpensiveData();

String _fetchExpensiveData() {
  print("Fetching expensive data...");
  // Simulate a time-consuming operation
  return "Data from a remote server";
}

void main() {
  setupDescription();
  print(description); // Accesses description after it\'s initialized

  print("\nBefore accessing expensiveData...");
  print(expensiveData); // _fetchExpensiveData() is called here for the first time
  print(expensiveData); // No re-call, value is cached
}
```

Type promotion and flow analysis

Dart's compiler performs **flow analysis** to intelligently determine the nullability of variables based on control flow. This allows for **type promotion**, where a nullable type can be treated as a non-nullable type within a certain scope if the compiler can prove it's not `null`.

```dart
void main() {
  String? name = "Alice";

  // Type promotion: Inside this if block, \'name\' is promoted from String? to String
  if (name != null) {
    print("Name length: ${name.length}"); // No error, \'name\' is treated as non-nullable
  }

  // After the if block, \'name\' reverts to String?
  // print(name.length); // Error: The property \'length\' can\'t be unconditionally
accessed.

  // Another example with \'is\' check
  Object value = "Hello Dart";
  if (value is String) {
    print("Value length: ${value.length}"); // \'value\' is promoted to String
  }

  // Using \'assert\' for type promotion (debug mode only)
  String? message = getMessage();
  assert(message != null); // In debug mode, this asserts non-nullability
  print("Message: ${message.toUpperCase()}"); // \'message\' is promoted to String
}

String? getMessage() {
  // In a real app, this might return null sometimes
  return "A non-null message";
}
```

Migration strategies for legacy code

Migrating existing Dart projects to null safety can be a significant undertaking, especially for large codebases. Dart provides tools and strategies to help with this process.

`dart migrate` tool

The `dart migrate` command-line tool helps you interactively migrate your project to null safety. It analyzes your code, suggests changes, and allows you to apply them.

1. **Run the tool:** `bash cd your_project_directory dart migrate`
2. **Review suggestions:** The tool will open a web interface in your browser, showing suggested changes. You can review and accept/reject them.
3. **Apply changes:** Once satisfied, you can apply the changes, and the tool will modify your source code.

Incremental Migration

Dart supports incremental migration, meaning you don't have to migrate your entire project at once. You can migrate libraries (packages or files) one by one. Code that is not yet migrated runs in a legacy (non-null-safe) mode, and Dart ensures interoperability between null-safe and legacy code.

- **Opt-in basis:** Null safety is opt-in. New projects created with recent Dart SDKs are null-safe by default. Existing projects need to explicitly opt-in.
- **Interoperability:** Null-safe code can call legacy code, and vice-versa. The compiler adds runtime checks at the boundaries to ensure safety.

Best Practices for Migration:

- **Start small:** Begin by migrating leaf libraries (libraries that don't depend on other unmigrated libraries).
- **Use `late` and `!` sparingly:** While useful, over-reliance on `late` and `!` can hide potential nullability issues. Prefer making types nullable (`?`) and using null-aware operators (`??` , `?.`) where appropriate.
- **Test thoroughly:** After migration, rigorously test your application to ensure that null safety has been correctly applied and no new bugs have been introduced.
- **Understand your data:** Analyze your data flows to determine where `null` values are genuinely possible and where they are not.

Null safety is a powerful feature that makes Dart applications more robust and easier to maintain by eliminating an entire class of common programming errors. Embracing it leads to more reliable and predictable code.

Error Handling & Exceptions

Error handling is a crucial aspect of robust software development. It allows your program to gracefully manage unexpected situations, preventing crashes and providing a better user experience. Dart uses an exception-based error handling mechanism, similar to many other modern languages.

Try-catch-finally blocks

The `try-catch-finally` construct is the primary way to handle exceptions in Dart.

- **`try` block:** Contains the code that might throw an exception.
- **`catch` block:** Catches and handles exceptions thrown within the `try` block. You can specify the type of exception to catch.
- **`on` clause:** Similar to `catch`, but allows you to specify the exception type without binding the exception object to a variable.
- **`finally` block:** Contains code that is guaranteed to execute whether an exception was thrown or not, and whether it was caught or not. It\"s typically used for cleanup operations.

```
void main() {
  // Example 1: Catching a specific exception
  try {
    int result = 10 ~/ 0; // Integer division by zero
    print("Result: $result");
  } on IntegerDivisionByZeroException {
    print("Error: Cannot divide by zero!");
  } catch (e) {
    print("An unexpected error occurred: $e");
  } finally {
    print("This code always runs (Example 1).");
  }

  print("\n");

  // Example 2: Catching any exception with stack trace
  try {
    List<int> numbers = [1, 2, 3];
    print(numbers[5]); // Accessing out-of-bounds index
  } catch (e, s) { // e for exception object, s for stack trace
    print("Caught exception: $e");
    print("Stack trace: $s");
  } finally {
    print("This code always runs (Example 2).");
  }

  print("\n");

  // Example 3: No exception thrown
  try {
    print("No exception here.");
  } catch (e) {
    print("This catch block will not execute.");
  } finally {
    print("This code always runs (Example 3).");
  }
}
```

Throwing custom exceptions

You can define and throw your own custom exceptions to represent specific error conditions in your application. Custom exceptions are typically classes that `implement` or `extend` `Exception` or `Error`.

- **`Exception`:** Represents an error that the program can reasonably recover from.
- **`Error`:** Represents a program error that indicates a defect in the code and is usually not recoverable.

It\"s generally recommended to `implement` `Exception` for custom exceptions that you expect to be caught and handled.

```
// Define a custom exception class
class InvalidAmountException implements Exception {
  final String message;

  InvalidAmountException([this.message = "Amount must be positive."]);

  @override
  String toString() => "InvalidAmountException: $message";
}

void processPayment(double amount) {
  if (amount <= 0) {
    throw InvalidAmountException("Payment amount cannot be zero or negative.");
  }
  print("Processing payment of
```amount...");
  // Simulate payment processing
  print("Payment successful!");
}

void main() {
  try {
    processPayment(100.0);
    processPayment(-50.0);
  } on InvalidAmountException catch (e) {
    print("Caught custom exception: ${e.message}");
  } catch (e) {
    print("Caught unexpected exception: $e");
  }
}
```

Built-in exception types

Dart provides several built-in exception types for common error scenarios:

- `FormatException`: Thrown when an input string is not in a valid format (e.g., `int.parse(\"abc\")`).
- `ArgumentError`: Thrown when an invalid argument is passed to a function.
- `StateError`: Thrown when an object is in an invalid state for the operation (e.g., calling `first` on an empty list).
- `RangeError`: Thrown when a value is outside a specified valid range (e.g., accessing a list index out of bounds).
- `UnsupportedError`: Thrown when an operation is not supported.
- `NullThrownError`: Thrown when a `null` value is used with the null assertion operator (`!`) and the value is actually `null`.
- `IntegerDivisionByZeroException`: Thrown when attempting integer division by zero.
```

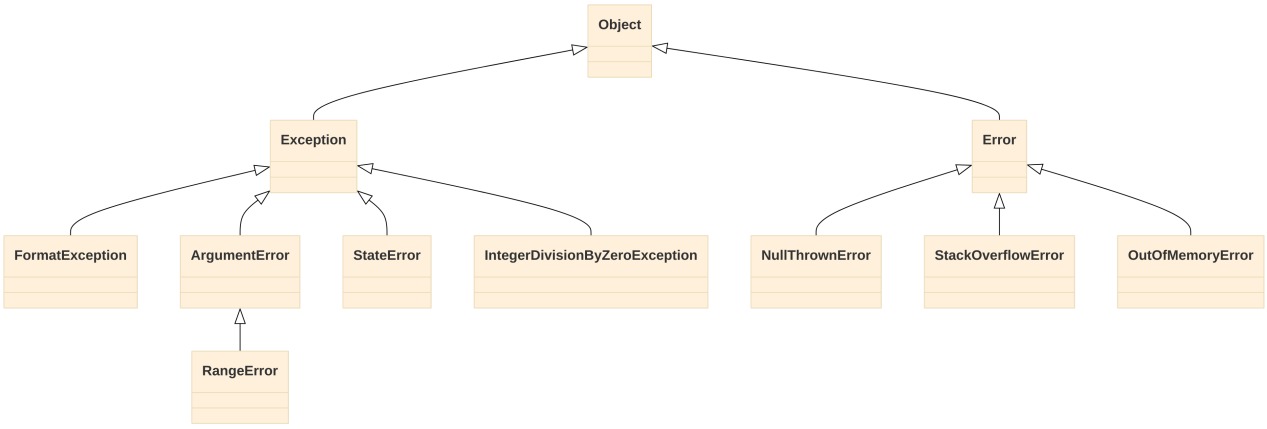
```dart
void main() {
  // FormatException
  try {
    int.parse("hello");
  } on FormatException catch (e) {
    print("FormatException: ${e.message}");
  }

  // ArgumentError
  try {
    List<int> list = [];
    list.removeAt(-1); // Invalid index
  } on RangeError catch (e) { // RangeError is a subclass of ArgumentError
    print("RangeError: ${e.message}");
  }

  // StateError
  try {
    List<int> emptyList = [];
    emptyList.first; // No element
  } on StateError catch (e) {
    print("StateError: ${e.message}");
  }
}
```

Exception hierarchy and handling strategies

Dart\"s exception hierarchy is rooted in `Object`. The two main branches are `Exception` and `Error`.



Handling Strategies:

- **Catch specific exceptions first:** Always try to catch the most specific exception types first, followed by more general ones. This allows for precise error handling.
- **Don\"t swallow exceptions:** Avoid empty `catch` blocks (`catch (e) {}`) as they can hide critical issues. At a minimum, log the exception.
- **Re-throw when necessary:** If a `catch` block cannot fully handle an exception, it can re-throw it using `rethrow` to allow higher-level handlers to deal with it.
- **Use `finally` for cleanup:** Ensure resources (like file handles or network connections) are properly closed or released in a `finally` block.

- **Errors vs. Exceptions:**
  - **Exceptions** are for predictable, recoverable problems (e.g., invalid user input, file not found). Your code should anticipate and handle these.
  - **Errors** are for programming bugs or system failures (e.g., out of memory, stack overflow). These usually indicate a flaw in your code that needs to be fixed, rather than handled at runtime.

```
void readFile(String path) {
  try {
    // Simulate file reading that might throw an exception
    if (path.isEmpty) {
      throw ArgumentError("File path cannot be empty.");
    } else if (path == "non_existent.txt") {
      throw Exception("File not found: $path");
    } else if (path == "corrupt.txt") {
      throw FormatException("Corrupt file data.");
    } else {
      print("Reading file: $path");
    }
  } on ArgumentError catch (e) {
    print("Specific error: ${e.message}");
    // Log the error, maybe show a user-friendly message
  } on FormatException catch (e) {
    print("Data format error: ${e.message}");
    // Prompt user to fix file format
  } catch (e) { // Catch any other Exception or Error
    print("An unexpected error occurred: $e");
    rethrow; // Re-throw to propagate the error if not fully handled
  } finally {
    print("File operation attempted for: $path");
    // Close file handle or release resources here
  }
}

void main() {
  readFile("data.txt");
  readFile("");
  readFile("non_existent.txt");
  readFile("corrupt.txt");
}
```

Assert statements for debugging

`assert` statements are used for debugging purposes. They take a boolean condition and an optional message. If the condition is `false`, an `AssertionError` is thrown. `assert` statements are only active in debug mode and are ignored in production (release) builds.

```
void main() {
  int age = 15;

  // Assert that age is positive
  assert(age >= 0, "Age cannot be negative.");
  print("Age is valid: $age");

  age = -5;
  // This assert will fail in debug mode, throwing an AssertionError
  // In production, this line is effectively removed.
  assert(age >= 0, "Age cannot be negative. Found: $age");
  print("This line will not be reached if assert fails in debug mode.");
}
```

`assert` is a powerful tool for validating assumptions in your code during development, helping you catch logical errors early.

Stack traces and error debugging

A **stack trace** is a list of the active stack frames at a certain point in time during the execution of a program. It shows the sequence of method calls that led to the current point, typically where an exception was thrown. Stack traces are invaluable for debugging, as they pinpoint the exact location and path of execution that caused an error.

When an uncaught exception occurs, Dart automatically prints the stack trace to the console. You can also explicitly get the stack trace in a `catch` block.

```
void functionC() {
  throw Exception("Error from Function C");
}

void functionB() {
  functionC();
}

void functionA() {
  functionB();
}

void main() {
  try {
    functionA();
  } catch (e, s) {
    print("Caught exception in main: $e");
    print("\n--- Stack Trace ---");
    print(s);
    print("--------------------");
  }
}
```

Interpreting a Stack Trace:

- The stack trace reads from bottom to top, with the most recent function call at the top.
- Each line typically shows the file name, line number, and column number where the call was made.

- Look for your own code files first to identify where the error originated or where an unexpected state was created.

Effective error handling and debugging with stack traces are essential skills for any Dart developer to build reliable applications.

Asynchronous Programming (Future, async/await, Streams)

Asynchronous programming is fundamental for building responsive and efficient applications, especially in environments like Flutter where UI must remain fluid while long-running operations (like network requests or file I/O) are performed. Dart is single-threaded, meaning it executes one operation at a time. Asynchronous operations allow your program to perform tasks without blocking the main execution thread.

Understanding asynchronous vs synchronous execution

- **Synchronous execution:** Operations run one after another, in sequence. Each operation must complete before the next one starts. If a synchronous operation takes a long time, it will block the entire program, making it unresponsive.

  ```dart void synchronousExample() { print("Start synchronous operation"); // Simulate a long-running task for (int i = 0; i < 1000000000; i++) {} print("End synchronous operation"); print("This line runs after the loop completes"); }

  void main() { synchronousExample(); print("Main function finished"); } `` In this example, "Main function finished" will only be printed after the synchronousExample` function, including its long loop, has fully completed.

- **Asynchronous execution:** Operations can start and run in the background without blocking the main thread. The program can continue executing other tasks while waiting for the asynchronous operation to complete. When the asynchronous operation finishes, it notifies the program, and its result can be processed.

  ```dart import \'dart:async\';

  Future asynchronousExample() async { print("Start asynchronous operation"); await Future.delayed(Duration(seconds: 2), () { print("Asynchronous task completed after 2 seconds"); }); print("End asynchronous operation"); }

  void main() { asynchronousExample(); print("Main function continues immediately"); // Other non-blocking operations can run here } `` In this example, "Main function continues

`immediately" is printed almost instantly, even though the` asynchronousExample `function takes 2 seconds to complete its delayed task. This is because` Future.delayed` is an asynchronous operation.

## Future basics and completion handling

A `Future` object represents a potential value or error that will be available at some time in the future. It\"s a way to handle a value that isn\"t available yet but will be eventually.

### Creating and Completing Futures

- **`Future.value(value)`:** Creates a Future that immediately completes with the given value.
- **`Future.error(error)`:** Creates a Future that immediately completes with the given error.
- **`Future.delayed(duration, [computation])`:** Creates a Future that completes after a specified duration. The `computation` callback is executed after the duration.

### Handling Future Completion

- **`then()`:** Registers a callback to be executed when the Future completes successfully with a value.
- **`catchError()`:** Registers a callback to be executed if the Future completes with an error.
- **`whenComplete()`:** Registers a callback to be executed when the Future completes, whether successfully or with an error. Useful for cleanup.

```
void main() {
  print("Fetching user data...");

  fetchUserData().then((data) {
    print("User data fetched: $data");
  }).catchError((error) {
    print("Error fetching user data: $error");
  }).whenComplete(() {
    print("User data fetch operation completed.");
  });

  print("Program continues while fetching...");

  // Example with Future.delayed
  Future.delayed(Duration(seconds: 1), () => "Delayed message")
      .then((msg) => print(msg));

  // Example with error
  Future.error(Exception("Something went wrong!"))
      .catchError((e) => print("Caught immediate error: $e"));
}

Future<String> fetchUserData() {
  return Future.delayed(Duration(seconds: 3), () {
    // Simulate a network error 20% of the time
    if (DateTime.now().second % 5 == 0) {
      throw Exception("Network error: Failed to connect.");
    }
    return "John Doe";
  });
}
```

Async/await syntax and error handling

The `async` and `await` keywords provide a more synchronous-looking way to write asynchronous code, making it much easier to read and manage than chained `then()` calls. They are syntactic sugar over `Future`s.

- **`async`:** Marks a function as asynchronous. An `async` function always returns a `Future`.
- **`await`:** Can only be used inside an `async` function. It pauses the execution of the `async` function until the `Future` it\"s waiting on completes. The result of the `Future` is then returned.

Using `async` / `await`

```
Future<String> fetchProductDetails() async {
  print("Fetching product details...");
  await Future.delayed(Duration(seconds: 2)); // Simulate network delay
  return "Laptop - High Performance";
}

Future<double> calculateShippingCost() async {
  print("Calculating shipping cost...");
  await Future.delayed(Duration(seconds: 1)); // Simulate calculation delay
  return 15.50;
}

void main() async {
  print("Starting order process...");

  try {
    String product = await fetchProductDetails();
    double shipping = await calculateShippingCost();

    print("Order Summary: $product, Shipping: $${shipping.toStringAsFixed(2)}");
  } catch (e) {
    print("An error occurred during order processing: $e");
  } finally {
    print("Order process completed.");
  }

  print("Main function continues after async operations.");
}
```

Error Handling with `async` / `await`

Errors thrown within an `async` function (or by an `await`ed `Future`) can be caught using standard `try-catch` blocks, just like synchronous code. This is a major advantage for readability.

```dart
Future<String> loginUser(String username, String password) async {
  print("Attempting to log in $username...");
  await Future.delayed(Duration(seconds: 2));

  if (username == "test" && password == "password") {
    return "Login successful!";
  } else if (username == "errorUser") {
    throw Exception("User account locked.");
  } else {
    throw Exception("Invalid credentials.");
  }
}

void main() async {
  // Successful login
  try {
    String result = await loginUser("test", "password");
    print(result);
  } catch (e) {
    print("Login failed: $e");
  }

  print("\n");

  // Failed login - invalid credentials
  try {
    String result = await loginUser("wrong", "pass");
    print(result);
  } catch (e) {
    print("Login failed: $e");
  }

  print("\n");

  // Failed login - user account locked
  try {
    String result = await loginUser("errorUser", "anypass");
    print(result);
  } catch (e) {
    print("Login failed: $e");
  }
}
```

Stream creation and subscription

A `Stream` is a sequence of asynchronous events. It\"s like a `Future` that can deliver multiple values (or errors) over time, rather than just one. Streams are commonly used for handling user input events, file I/O, network data, or real-time updates.

Creating Streams

- **`Stream.fromIterable()`:** Creates a stream from an existing `Iterable`.
- **`Stream.periodic()`:** Creates a stream that emits events at regular intervals.
- **`StreamController`:** Provides a way to imperatively add data, errors, and completion events to a stream.

```dart
import \'dart:async\';

void main() async {
  // 1. Stream from Iterable
  Stream<int> numberStream = Stream.fromIterable([1, 2, 3, 4, 5]);
  print("\nSubscribing to numberStream:");
  numberStream.listen((data) {
    print("Number: $data");
  }, onDone: () => print("Number stream done."));

  // 2. Stream.periodic
  Stream<int> countdownStream = Stream.periodic(Duration(seconds: 1), (count) => 5 - count)
      .take(6); // Take only 6 events (5, 4, 3, 2, 1, 0)
  print("\nSubscribing to countdownStream:");
  countdownStream.listen((data) {
    print("Countdown: $data");
  }, onDone: () => print("Countdown stream done."));

  // 3. StreamController (for more control)
  final controller = StreamController<String>();

  controller.stream.listen(
    (data) => print("Controller Data: $data"),
    onError: (error) => print("Controller Error: $error"),
    onDone: () => print("Controller stream done."),
    cancelOnError: true, // If an error occurs, cancel the subscription
  );

  controller.add("Hello");
  await Future.delayed(Duration(milliseconds: 500));
  controller.add("World");
  await Future.delayed(Duration(milliseconds: 500));
  controller.addError("Oops, something went wrong!");
  await Future.delayed(Duration(milliseconds: 500));
  controller.add("This will not be printed if cancelOnError is true");
  controller.close(); // Important: close the controller when done
}
```

Subscribing to Streams

- **`listen()`:** The most common way to consume a stream. It takes callbacks for data, error, and completion events.
- **`await for`:** A convenient way to iterate over a stream in an `async` function, treating each event as if it were a synchronous loop iteration.

```dart
import \'dart:async\';

Stream<String> getMessages() async* {
  yield "Message 1";
  await Future.delayed(Duration(seconds: 1));
  yield "Message 2";
  await Future.delayed(Duration(seconds: 1));
  yield "Message 3";
}

void main() async {
  print("\nUsing await for with messages stream:");
  await for (var message in getMessages()) {
    print("Received: $message");
  }
  print("Messages stream finished.");
}
```

Stream transformations and operations

Streams can be transformed and manipulated using various methods, similar to `Iterable` methods, to filter, map, or combine events.

- **`map()`:** Transforms each event into a new event.
- **`where()`:** Filters events based on a condition.
- **`take()`:** Takes a specified number of events and then closes the stream.
- **`skip()`:** Skips a specified number of events.
- **`distinct()`:** Emits only events that are different from the previous one.
- **`reduce()`** / **`fold()`:** Aggregates stream events into a single value.
- **`asyncMap()`:** Asynchronously transforms each event.
- **`expand()`:** Transforms each event into zero or more new events.

```dart
import \'dart:async\';

void main() async {
  Stream<int> originalStream = Stream.fromIterable([1, 2, 3, 4, 5, 6, 7, 8, 9, 10]);

  // Map: Square each number
  Stream<int> squaredStream = originalStream.map((n) => n * n);
  print("\nSquared numbers:");
  await for (var n in squaredStream) {
    print(n);
  }

  // Where: Filter even numbers
  Stream<int> evenStream = originalStream.where((n) => n % 2 == 0);
  print("\nEven numbers:");
  await for (var n in evenStream) {
    print(n);
  }

  // Take: Get first 3 numbers
  Stream<int> firstThreeStream = originalStream.take(3);
  print("\nFirst three numbers:");
  await for (var n in firstThreeStream) {
    print(n);
  }

  // Reduce: Sum all numbers
  int sum = await originalStream.reduce((a, b) => a + b);
  print("\nSum of numbers: $sum");

  // Combine multiple streams (e.g., using StreamZip from rxdart or similar logic)
  // For simplicity, showing a basic example of combining values from two streams
  Stream<String> names = Stream.fromIterable(["Alice", "Bob", "Charlie"]);
  Stream<int> ages = Stream.fromIterable([25, 30, 35]);

  // This is a simplified conceptual example; real-world combining often uses rxdart or
custom logic
  // For a proper zip, you\"d use something like Rx.zip2 from rxdart package.
  // Here, we\"ll just show how to process them sequentially.
  print("\nNames and Ages (sequential processing):");
  List<String> collectedNames = [];
  List<int> collectedAges = [];

  await for (var name in names) {
    collectedNames.add(name);
  }
  await for (var age in ages) {
    collectedAges.add(age);
  }

  for (int i = 0; i < collectedNames.length; i++) {
    print("$`{collectedNames[i]} is `${collectedAges[i]} years old.");
  }
}
```

Isolates for concurrent programming

Dart is single-threaded, but it can achieve concurrency using **Isolates**. Isolates are independent workers that run in their own memory space, meaning they don\"t share memory with the main thread or other isolates. This prevents shared-state concurrency issues like race conditions and deadlocks.

When you need to perform CPU-intensive computations that would block the main UI thread, you can offload them to an Isolate.

How Isolates Work

- Each Isolate has its own event loop and memory.
- Communication between Isolates happens via **messages** (ports). Data is copied, not shared.
- The `compute` function from `package:flutter/foundation.dart` (or `dart:isolate` directly) is a convenient way to run a function in a separate isolate.

```
 import \'dart:isolate\';

// A function that performs a heavy computation
// This function must be a top-level function or a static method
int heavyComputation(int count) {
  int sum = 0;
  for (int i = 0; i < count; i++) {
    sum += i;
  }
  return sum;
}

void main() async {
  print("Main thread: Starting heavy computation...");

  // Create a ReceivePort to listen for messages from the new isolate
  final receivePort = ReceivePort();

  // Spawn a new isolate
  final isolate = await Isolate.spawn(
    heavyComputation, // The function to run in the new isolate
    1000000000,       // The argument to pass to the function
    onExit: receivePort.sendPort, // Send a message when isolate exits
    onError: receivePort.sendPort, // Send error messages
  );

  // Listen for messages from the isolate
  receivePort.listen((message) {
    if (message is int) {
      print("Main thread: Computation result: $message");
    } else if (message is List) {
      // Error message format: [error, stackTrace]
      print("Main thread: Isolate error: ${message[0]}");
      print("Main thread: Stack trace: ${message[1]}");
    } else if (message is SendPort) {
      // This is the onExit message, indicating the isolate has exited
      print("Main thread: Isolate exited.");
    } else {
      print("Main thread: Received unknown message: $message");
    }
    receivePort.close(); // Close the port when done
    isolate.kill(); // Terminate the isolate
  });

  print("Main thread: Continuing with other tasks...");
  // The UI remains responsive here
}
```

In this example, `heavyComputation` runs in a separate isolate, preventing the main thread from freezing. The result is sent back to the main thread via a `SendPort` and `ReceivePort` mechanism.

When to use Isolates: * For CPU-bound tasks (e.g., complex calculations, image processing, JSON parsing of very large files). * When you need to ensure the UI remains smooth and responsive.

Asynchronous programming with Futures, `async` / `await`, Streams, and Isolates are powerful tools in Dart for building high-performance, non-blocking applications, especially crucial for Flutter development.

Generics in Dart

Generics are a powerful feature in Dart (and many other languages) that allow you to write code that works with different types while maintaining type safety. They enable you to create classes, methods, and functions that can operate on objects of various types without losing type information or resorting to `Object` and casting, which can lead to runtime errors.

Generic classes and type parameters

Generic classes are classes that are parameterized by type. This means you can define a class that works with a placeholder type, and then specify the actual type when you create an instance of that class.

Example of Generic Class

Consider a simple `Box` class that can hold a single item. Without generics, you might store an `Object`, which loses type information.

```
// Non-generic Box (stores Object)
class ObjectBox {
  Object content;
  ObjectBox(this.content);
}

// Generic Box (stores a specific type T)
class Box<T> {
  T content;
  Box(this.content);

  T getContent() {
    return content;
  }

  void setContent(T newContent) {
    content = newContent;
  }
}

void main() {
  // Using non-generic box (type safety lost)
  var intBox = ObjectBox(123);
  // var value = intBox.content as int; // Requires casting, prone to runtime errors
  // print("Int Box Content: $value");

  // Using generic box (type safety maintained)
  var stringBox = Box<String>("Hello Generics");
  String message = stringBox.getContent(); // No casting needed
  print("String Box Content: $message");

  var numberBox = Box<int>(42);
  int number = numberBox.getContent();
  print("Number Box Content: $number");

  // Compile-time error: A value of type \'double\' can\'t be assigned to a variable of type
\'int\'.
  // numberBox.setContent(3.14);

  // You can also use type inference
  var inferredBox = Box(true); // Inferred as Box<bool>
  bool booleanValue = inferredBox.getContent();
  print("Inferred Box Content: $booleanValue");
}
```

In `Box<T>`, `T` is a **type parameter**. When you create `Box<String>`, `T` becomes `String` for that instance, and the compiler ensures that only `String` values can be stored or retrieved, providing compile-time type safety.

Generic methods and functions

Just like classes, methods and functions can also be generic. This allows them to operate on arguments of various types while maintaining type safety.

Example of Generic Method/Function

```
// Generic function to print an item of any type
void printItem<T>(T item) {
  print("Item: $`item (Type: `${item.runtimeType})");
}

// Generic method within a class
class Util {
  static T? getFirstElement<T>(List<T> list) {
    if (list.isEmpty) {
      return null;
    }
    return list[0];
  }
}

void main() {
  printItem<String>("Dart");
  printItem<int>(123);
  printItem(3.14); // Type inference: printItem<double>

  List<String> names = ["Alice", "Bob", "Charlie"];
  String? firstPerson = Util.getFirstElement<String>(names);
  print("First person: $firstPerson");

  List<double> temperatures = [25.5, 28.1, 22.0];
  double? firstTemp = Util.getFirstElement(temperatures); // Type inference
  print("First temperature: $firstTemp");

  List<int> emptyList = [];
  int? firstInt = Util.getFirstElement(emptyList);
  print("First int in empty list: $firstInt");
}
```

Type constraints and bounded generics

Sometimes, you want to restrict the types that can be used as type arguments for a generic class or method. This is done using **type constraints** (also known as **bounded generics**).

You can specify that a type parameter `T` must be a subtype of a certain type `S` using the `extends` keyword: `class MyClass<T extends S>` or `void myMethod<T extends S>(T arg)`.

Example of Bounded Generics

Suppose you want a `NumberBox` that can only hold numbers (integers or doubles).

```dart
// A generic class that only accepts types that extend num
class NumberBox<T extends num> {
  T value;
  NumberBox(this.value);

  double get doubleValue => value.toDouble();

  void add(T other) {
    // Operations like + are available because T is guaranteed to be a num
    value = (value + other) as T; // Cast needed because Dart doesn\'t know the exact
subtype of num
  }
}

// A generic function that only accepts types that extend Comparable
int findMax<T extends Comparable>(T a, T b) {
  if (a.compareTo(b) > 0) {
    return 1; // a is greater
  } else if (a.compareTo(b) < 0) {
    return -1; // b is greater
  } else {
    return 0; // they are equal
  }
}

void main() {
  var intBox = NumberBox<int>(10);
  print("Int Box value: ${intBox.value}");
  intBox.add(5);
  print("Int Box after add: ${intBox.value}");

  var doubleBox = NumberBox<double>(3.14);
  print("Double Box value: ${doubleBox.value}");
  doubleBox.add(2.5);
  print("Double Box after add: ${doubleBox.value}");

  // This would be a compile-time error because String does not extend num
  // var stringBox = NumberBox<String>("hello");

  print("\nComparing numbers:");
  print("Max of 10 and 20: ${findMax(10, 20)}"); // -1 (20 is greater)
  print("Max of 5.5 and 3.2: ${findMax(5.5, 3.2)}"); // 1 (5.5 is greater)

  print("\nComparing strings:");
  print("Max of \"apple\" and \"banana\": ${findMax("apple", "banana")}"); // -1 (banana is
greater alphabetically)
}
```

By using `T extends num`, we ensure that `T` will always have methods like `toDouble()` and arithmetic operators available, preventing runtime errors. Similarly, `T extends Comparable` ensures that the `compareTo` method is available.

Generic collections and type safety

Dart\'s built-in collection classes (`List`, `Set`, `Map`) are generic. This is a primary reason why generics are so important: they provide type safety for collections.

```dart
void main() {
  // List<String> - ensures only strings can be added
  List<String> names = [];
  names.add("Alice");
  names.add("Bob");
  // names.add(123); // Compile-time error

  // Set<int> - ensures only unique integers
  Set<int> uniqueNumbers = {1, 2, 3, 2};
  print("Unique Numbers: $uniqueNumbers"); // {1, 2, 3}
  // uniqueNumbers.add(\"four\"); // Compile-time error

  // Map<String, double> - ensures string keys and double values
  Map<String, double> productPrices = {
    "Laptop": 1200.0,
    "Mouse": 25.50,
  };
  // productPrices[\"Keyboard\"] = \"50\"; // Compile-time error

  // Iterating over generic collections
  for (String name in names) {
    print("Name: $name");
  }

  // Benefits of type safety
  String first = names[0]; // No casting needed
  // int firstInt = names[0]; // Compile-time error
}
```

Without generics, collections would typically store `Object`, requiring manual casting and making it easy to introduce `ClassCastException` errors at runtime.

Variance and covariance concepts

Variance describes how subtyping between complex types (like generic types) relates to subtyping between their component types. Dart\'s generics are generally **covariant** for return types and **contravariant** for parameter types, but for simplicity, Dart treats most generic types as **invariant** by default to ensure sound null safety and prevent runtime errors, unless explicitly opted into covariance.

- **Covariance:** If `A` is a subtype of `B`, then `List<A>` is a subtype of `List<B>`. (e.g., `List<Dog>` is a `List<Animal>`). This is generally true for return types.
- **Contravariance:** If `A` is a subtype of `B`, then `List<B>` is a subtype of `List<A>`. (e.g., `Function(Animal)` is a subtype of `Function(Dog)`). This is generally true for parameter types.
- **Invariance:** `List<A>` is neither a subtype nor a supertype of `List<B>` unless `A` and `B` are the same type. This is the default for most generic types in Dart to ensure type safety.

Dart collections are **invariant** by default to prevent runtime errors. However, you can use the `covariant` keyword on a parameter to allow a more specific type in a subclass method, but this comes with a runtime check.

```dart
class Animal {
  void eat(Food food) {
    print("Animal eats $food");
  }
}

class Dog extends Animal {
  @override
  void eat(covariant DogFood food) { // Using covariant keyword
    print("Dog eats $food");
  }
}

class Food {}
class DogFood extends Food {}

void main() {
  Animal animal = Dog();
  animal.eat(Food()); // This will work at compile time due to covariance on `eat` parameter
                      // but will throw a runtime error if `food` is not `DogFood`
                      // because `Dog.eat` expects `DogFood`.

  Dog dog = Dog();
  dog.eat(DogFood()); // This is fine
}
```

This is an advanced topic and often not something you need to explicitly manage unless dealing with complex type hierarchies and method overriding. The `covariant` keyword essentially tells the compiler to relax the type check at compile time and instead perform a runtime check.

Type erasure and runtime behavior

Dart, like Java and C#, uses **type erasure** for generics at runtime. This means that the generic type information (e.g., `<String>` in `List<String>`) is not available at runtime. At runtime, `List<String>` and `List<int>` both appear as just `List<dynamic>`.

```dart
void main() {
  List<String> stringList = ["a", "b"];
  List<int> intList = [1, 2];

  print(stringList.runtimeType); // Output: List<String>
  print(intList.runtimeType);    // Output: List<int>

  // However, at a deeper level, the generic type is erased.
  // You cannot check the generic type at runtime using `is`
  print(stringList is List<dynamic>); // true
  print(intList is List<dynamic>);    // true

  // This is why you can do this (though it\'s bad practice and leads to runtime errors):
  List<dynamic> dynamicList = [1, 2, 3];
  // dynamicList.add(\"hello\"); // No compile-time error

  // If you then try to treat it as a List<int>:
  // List<int> anotherIntList = dynamicList; // Compile-time error due to invariance

  // But if you force it with `as`:
  // List<int> forcedIntList = dynamicList as List<int>; // Runtime error if elements are
not int
}
```

While `runtimeType` might show the generic type, this is primarily for debugging and is not a reliable way to perform type checks on generic parameters at runtime. The compiler uses the generic information during static analysis to provide type safety, but this information is largely erased for execution efficiency.

Generics are a cornerstone of writing flexible, reusable, and type-safe code in Dart, especially when working with collections and building robust APIs.

Dart Packages & Pub.dev

Dart, like many modern programming languages, relies heavily on a robust package ecosystem to extend its functionality. **Pub.dev** is the official package repository for Dart and Flutter, hosting thousands of open-source packages that developers can use to build applications more efficiently. The **pub tool** (part of the Dart SDK) is used to manage these packages.

Understanding pubspec.yaml configuration

The `pubspec.yaml` file is the heart of any Dart or Flutter project. It\'s a YAML (YAML Ain\'t Markup Language) file that defines the project\'s metadata, dependencies, and other configurations.

Key Sections of `pubspec.yaml`

- **`name`**: The name of your package. This should be a valid Dart identifier (lowercase, `snake_case`).
- **`description`**: A brief, human-readable description of your project.
- **`version`**: The current version of your package, following semantic versioning guidelines (e.g., `1.0.0`, `0.5.1+2`).
- **`environment`**: Specifies the Dart SDK version constraints that your package supports. This ensures compatibility. `yaml environment: sdk: ">=3.0.0 <4.0.0"`
- **`dependencies`**: Lists the packages your project needs to run in production. These are typically packages from Pub.dev.
- **`dev_dependencies`**: Lists packages used only during development, testing, or building (e.g., testing frameworks, linters).
- **`dependency_overrides`**: (Rarely used) Allows you to temporarily override a dependency\'s version, useful for testing local changes or resolving conflicts.
- **`flutter`**: (For Flutter projects) Contains Flutter-specific configurations, such as asset declarations, font declarations, and plugin registrations.

Example `pubspec.yaml`

```yaml
name: my_cli_tool
description: A simple command-line tool for managing tasks.
version: 1.0.0
homepage: https://example.com/my_cli_tool
repository: https://github.com/myuser/my_cli_tool

environment:
  sdk: ">=3.0.0 <4.0.0"

dependencies:
  http: ^1.2.0 # For making HTTP requests
  args: ^2.4.2 # For command-line argument parsing

dev_dependencies:
  lints: ^3.0.0 # For Dart linting rules
  test: ^1.24.0 # For writing unit tests

# For Flutter projects, you\'d also have a flutter section:
# flutter:
#   uses-material-design: true
#   assets:
#     - assets/images/
#     - assets/data.json
```

Adding and managing dependencies

Adding and managing dependencies in Dart is straightforward using the `pub` command-line tool.

Adding a Dependency

To add a new dependency, you can manually edit `pubspec.yaml` under `dependencies:` or `dev_dependencies:`, or use the `dart pub add` command.

Manual Edit:

```yaml
dependencies:
  some_package: ^1.0.0
```

Using `dart pub add` (recommended):

```
dart pub add some_package
# To add a dev dependency:
dart pub add --dev some_dev_package
```

This command automatically adds the package to your `pubspec.yaml` with a recommended version constraint and then runs `dart pub get`.

Getting Dependencies

After modifying `pubspec.yaml` (e.g., adding or removing dependencies), you need to fetch them. This downloads the packages and creates a `pubspec.lock` file, which precisely locks the versions of

all direct and transitive dependencies.

```
dart pub get
```

Upgrading Dependencies

To upgrade your dependencies to the latest compatible versions, use `dart pub upgrade`.

```
dart pub upgrade
```

This command respects the version constraints defined in `pubspec.yaml`. If you want to upgrade to the latest possible version ignoring constraints (use with caution!), you can use `dart pub upgrade --major-versions`.

Removing Dependencies

To remove a dependency, simply delete its entry from `pubspec.yaml` and then run `dart pub get`.

```
# After removing from pubspec.yaml
dart pub get
```

Alternatively, use `dart pub remove`:

```
dart pub remove some_package
```

Publishing packages to pub.dev

If you develop a useful Dart or Flutter package, you can share it with the community by publishing it to Pub.dev. Before publishing, ensure your package meets the publishing guidelines.

Steps to Publish

1. **Prepare your package:**
    - Ensure `pubspec.yaml` has `name`, `description`, `version`, `homepage` (or `repository`), and `environment` fields correctly set.
    - Add a `README.md` file explaining how to use your package.
    - Add a `CHANGELOG.md` file documenting changes for each version.
    - Add a `LICENSE` file.
    - Ensure your code is well-documented with Dartdoc comments.

2. **Run `dart pub publish --dry-run`:** This command simulates the publishing process without actually uploading the package. It checks for common errors and warnings.

   `bash dart pub publish --dry-run`

3. **Publish the package:** If the dry run is successful, you can proceed with the actual publish command.

   `bash dart pub publish` The first time you publish, you\'ll be asked to log in to your Google account (the same one used for Pub.dev). Follow the instructions in your browser.

Once published, your package will be available on Pub.dev for others to use.

Package versioning and semantic versioning

Dart packages follow **Semantic Versioning (SemVer)**, a widely adopted standard for version numbers. A version number is typically in the format `MAJOR.MINOR.PATCH` (e.g., `1.2.3`).

- **`MAJOR` version (e.g., `1.0.0` to `2.0.0`):** Incremented for incompatible API changes. Breaking changes.
- **`MINOR` version (e.g., `1.0.0` to `1.1.0`):** Incremented for adding functionality in a backward-compatible manner. New features.
- **`PATCH` version (e.g., `1.0.0` to `1.0.1`):** Incremented for backward-compatible bug fixes.

Version Constraints in `pubspec.yaml`

When specifying dependencies, you use version constraints to indicate which versions of a package your project is compatible with.

- **`any`**: Allows any version. (Not recommended for production). `yaml some_package: any`
- **`^` (Caret syntax - recommended):** Allows versions greater than or equal to the specified version, and less than the next major version. This is the most common and recommended way.
  - `^1.2.3` means `>=1.2.3 <2.0.0`
  - `^0.1.2` means `>=0.1.2 <0.2.0` (for versions less than 1.0.0, it only allows patch and minor updates) `yaml some_package: ^1.2.3`
- **`~>` (Tilde-caret syntax):** Allows versions greater than or equal to the specified version, and less than the next minor version. (Less common in Dart).
  - `~>1.2.3` means `>=1.2.3 <1.3.0` `yaml some_package: ~>1.2.3`
- **Exact version:** Only allows the specified version. (Generally discouraged as it prevents bug fixes). `yaml some_package: 1.2.3`

- **Range:** Specifies a custom range. `yaml some_package: ">=1.0.0 <2.0.0"`

Understanding SemVer and using appropriate version constraints is crucial for maintaining stable and compatible dependencies in your projects.

Local package development and path dependencies

Sometimes, you might be developing a package locally alongside an application that uses it, or you might want to use a package that hasn\'t been published to Pub.dev yet. In such cases, you can use **path dependencies**.

Path dependencies allow you to link to a package located on your local file system.

Example of Path Dependency

Suppose you have a project structure like this:

```
my_app/
  pubspec.yaml
  lib/
src/
  my_local_package/
    pubspec.yaml
    lib/
```

In `my_app/pubspec.yaml`, you would reference `my_local_package` like this:

```yaml
# my_app/pubspec.yaml
name: my_app
version: 1.0.0
environment:
  sdk: ">=3.0.0 <4.0.0"

dependencies:
  my_local_package:
    path: ../src/my_local_package # Relative path to the local package

dev_dependencies:
  lints: ^3.0.0
  test: ^1.24.0
```

After adding the path dependency, run `dart pub get` in `my_app/` to link the local package. Any changes you make in `my_local_package` will be reflected immediately in `my_app` without needing to publish.

When to use Path Dependencies: * During active development of a package and an application that consumes it. * For internal packages that are not intended for public distribution on Pub.dev. * For testing unreleased versions of your own packages.

Popular Dart packages and ecosystem overview

Dart\'s ecosystem is rich and constantly growing, largely driven by Flutter\'s popularity. Here\'s an overview of some popular and essential packages:

Core Utility Packages

- **`http`**: A composable, multi-platform, Future-based API for HTTP requests. Essential for network communication.
- **`path`**: A robust, cross-platform path manipulation library. Useful for working with file paths consistently across different operating systems.
- **`intl`**: Provides internationalization and localization facilities, including date/time formatting, number formatting, and message translation.
- **`collection`**: Additional utilities for working with collections (lists, sets, maps) that are not in `dart:collection`.
- **`uuid`**: Generates universally unique identifiers (UUIDs).

Data Handling and Serialization

- **`json_serializable`** / **`json_annotation`** / **`build_runner`**: A powerful set of packages for automatic JSON serialization/deserialization code generation. Reduces boilerplate for data models.
- **`protobuf`** / **`grpc`**: For working with Protocol Buffers and gRPC services, enabling efficient inter-service communication.
- **`xml`**: For parsing and generating XML.

State Management (Flutter-centric, but concepts apply)

- **`provider`**: A simple, scalable, and widely used state management solution for Flutter.
- **`bloc`** / **`flutter_bloc`**: A predictable state management library that helps implement the BLoC (Business Logic Component) pattern.
- **`riverpod`**: A reactive caching and data-binding framework, often seen as an alternative to Provider.
- **`getx`**: A fast, reactive, and opinionated state management solution, also offering dependency injection and route management.

Database and Persistence

- **`sqflite`**: SQLite plugin for Flutter. Allows you to interact with local SQLite databases.
- **`hive`**: A lightweight and blazing fast key-value database for Flutter and Dart.

- `shared_preferences`: For simple key-value storage of primitive data.
- `drift` **(formerly** `moor`**):** A reactive persistence library for Flutter and Dart, built on top of SQLite.

Testing

- `test`: The official Dart testing framework for unit and integration tests.
- `mockito`: A mocking framework for Dart, useful for isolating units of code during testing.

Development Tools

- `lints`: Recommended set of lint rules for Dart and Flutter projects.
- `build_runner`: A code generation tool used by many packages (like `json_serializable`) to generate boilerplate code.

This rich ecosystem significantly accelerates development by providing ready-to-use solutions for common programming tasks, allowing developers to focus on application-specific logic.

Mini Project: CLI Tool or Simple API Consumer

This section outlines a mini-project to consolidate your understanding of Dart programming concepts by building a practical command-line interface (CLI) tool or a simple API consumer. This project will cover planning, structure, argument parsing, file I/O, HTTP requests, JSON parsing, error handling, testing, and distribution.

Project planning and structure setup

Before writing any code, it\"s crucial to plan your project. For this mini-project, let\"s choose to build a simple CLI tool that fetches and displays random quotes from a public API.

Project Goal: Create a Dart CLI tool that fetches a random quote from an API and prints it to the console. It should also allow saving quotes to a local file.

Core Features: 1. Fetch a random quote from an API. 2. Display the quote and its author. 3. Optionally save the fetched quote to a local text file. 4. Handle network errors and file I/O errors gracefully.

Project Structure

First, create a new Dart project:

```
dart create console --force quote_cli_tool
cd quote_cli_tool
```

This will create a basic project structure:

```
quote_cli_tool/
├── .dart_tool/
├── bin/
│   └── quote_cli_tool.dart # Main executable file
├── lib/
│   └── quote_cli_tool.dart # Library code (where most logic will go)
├── pubspec.yaml
├── README.md
└── analysis_options.yaml
```

`pubspec.yaml` Setup

We\"ll need some external packages for HTTP requests and argument parsing. Open `pubspec.yaml` and add the following dependencies:

```
# pubspec.yaml
name: quote_cli_tool
description: A simple CLI tool to fetch and save random quotes.
version: 1.0.0

environment:
  sdk: ">=3.0.0 <4.0.0"

dependencies:
  http: ^1.2.0 # For making HTTP requests
  args: ^2.4.2 # For command-line argument parsing
  path: ^1.9.0 # For path manipulation (optional, but good practice)

dev_dependencies:
  lints: ^3.0.0
  test: ^1.24.0
```

After modifying `pubspec.yaml`, run `dart pub get` in your terminal to download the new dependencies:

```
dart pub get
```

Command-line argument parsing

The `args` package helps you define and parse command-line arguments. We\"ll use it to allow users to specify whether to save the quote and where.

Modify `bin/quote_cli_tool.dart`:

```
// bin/quote_cli_tool.dart
import \'package:args/args.dart\';
import \'package:quote_cli_tool/quote_cli_tool.dart\' as quote_cli_tool;

void main(List<String> arguments) async {
  final parser = ArgParser()
    ..addFlag(
      \'help\',
      abbr: \'h\',
      negatable: false,
      help: \'Print this usage information.\',
    )
    ..addFlag(
      \'save\',
      abbr: \'s\',
      negatable: false,
      help: \'Save the fetched quote to a file.\',
    )
    ..addOption(
      \'output\',
      abbr: \'o\',
      defaultsTo: \'quotes.txt\',
      help: \'Specify the output file name for saving quotes.\',
    );

  ArgResults argResults = parser.parse(arguments);

  if (argResults[\'help\']) {
    print(\'Usage: dart run quote_cli_tool [options]\');
    print(parser.usage);
    return;
  }

  final bool shouldSave = argResults[\'save\'] as bool;
  final String outputFile = argResults[\'output\'] as String;

  await quote_cli_tool.run(shouldSave, outputFile);
}
```

HTTP requests and JSON parsing

We\"ll use the `http` package to fetch data from a public API. A good choice for random quotes is `https://api.quotable.io/random`.

Create a new file `lib/quote_service.dart`:

```dart
// lib/quote_service.dart
import \'dart:convert\';
import \'package:http/http.dart\' as http;

class Quote {
  final String content;
  final String author;

  Quote({required this.content, required this.author});

  factory Quote.fromJson(Map<String, dynamic> json) {
    return Quote(
      content: json[\'content\'] as String,
      author: json[\'author\'] as String,
    );
  }

  @override
  String toString() {
    return \'"$`content" - `$author\';
  }
}

Future<Quote> fetchRandomQuote() async {
  final response = await http.get(Uri.parse(\'https://api.quotable.io/random\'));

  if (response.statusCode == 200) {
    // If the server returns a 200 OK response, parse the JSON.
    return Quote.fromJson(jsonDecode(response.body) as Map<String, dynamic>);
  } else {
    // If the server did not return a 200 OK response, throw an exception.
    throw Exception(\'Failed to load quote: ${response.statusCode}\');
  }
}
```

Now, modify `lib/quote_cli_tool.dart` to use this service:

```dart
// lib/quote_cli_tool.dart
import \'dart:io\';
import \'package:path/path.dart\' as p;
import \'package:quote_cli_tool/quote_service.dart\';

Future<void> run(bool shouldSave, String outputFile) async {
  try {
    final quote = await fetchRandomQuote();
    print(\'\n${quote.toString()}\n\');

    if (shouldSave) {
      await saveQuoteToFile(quote, outputFile);
      print(\'Quote saved to $outputFile\');
    }
  } catch (e) {
    print(\'Error: $e\');
  }
}

Future<void> saveQuoteToFile(Quote quote, String filename) async {
  final file = File(p.join(Directory.current.path, filename));
  await file.writeAsString(\'${quote.toString()}\n\', mode: FileMode.append);
}
```

File I/O operations and data persistence

The `dart:io` library provides classes for file and directory operations. We\"ve already integrated `File` and `FileMode.append` in the `saveQuoteToFile` function above to append the quote to a file.

Error handling in real applications

We\"ve already implemented basic `try-catch` blocks in `run` and `fetchRandomQuote` to handle exceptions. For a more robust application, you might want to:

- **Specific Exception Handling:** Catch more specific exceptions (e.g., `SocketException` for network issues, `FileSystemException` for file errors) and provide user-friendly messages.
- **Retry Mechanisms:** For transient network errors, implement a retry logic.
- **Logging:** Use a logging package (e.g., `logging`) to log errors for debugging.

```
// Example of more specific error handling in lib/quote_cli_tool.dart
import \'dart:io\';
import \'package:path/path.dart\' as p;
import \'package:http/http.dart\' as http;
import \'package:quote_cli_tool/quote_service.dart\';

Future<void> run(bool shouldSave, String outputFile) async {
  try {
    final quote = await fetchRandomQuote();
    print(\'\n${quote.toString()}\n\');

    if (shouldSave) {
      await saveQuoteToFile(quote, outputFile);
      print(\'Quote saved to $outputFile\');
    }
  } on http.ClientException catch (e) {
    print(\'Network Error: Could not connect to the quote API. Please check your internet
connection. ($e)\');
  } on FileSystemException catch (e) {
    print(\'File Error: Could not save the quote to $`outputFile. (`$e)\');
  } on Exception catch (e) {
    print(\'An unexpected error occurred: $e\');
  }
}

// ... rest of the code ...
```

Testing and documentation best practices

Testing

Unit testing is crucial for ensuring the correctness of your code. The `test` package is Dart\"s official testing framework.

Create `test/quote_service_test.dart` :

```
// test/quote_service_test.dart
import \'package:test/test.dart\';
import \'package:quote_cli_tool/quote_service.dart\';

void main() {
  group(\'Quote\', () {
    test(\'fromJson creates a Quote object correctly\', () {
      final json = {\'content\': \'Test Quote\', \'author\': \'Test Author\'};
      final quote = Quote.fromJson(json);
      expect(quote.content, \'Test Quote\');
      expect(quote.author, \'Test Author\');
    });

    test(\'toString returns formatted string\', () {
      final quote = Quote(content: \'Hello World\', author: \'Anonymous\');
      expect(quote.toString(), \'"Hello World" - Anonymous\');
    });
  });

  // Note: Testing fetchRandomQuote() directly would require mocking HTTP requests.
  // For simplicity, we skip direct API call testing here, but in a real app,
  // you\"d use a mocking library like `mockito`.
}
```

Documentation

Good documentation makes your code understandable and usable. Dart has a built-in documentation generator (`dart doc`).

- **Dartdoc Comments:** Use `///` for single-line documentation comments and `/** ... */` for multi-line comments. These comments support Markdown.

  ```dart /// Represents a quote with content and author. class Quote { /// The main text content of the quote. final String content;

  /// The author of the quote. final String author;

  /// Creates a [Quote] instance. /// /// [content] is the text of the quote. /// [author] is the person who said or wrote the quote. Quote({required this.content, required this.author});

  // ... rest of the class ... } ```

- **README.md:** Provide a comprehensive `README.md` in your project root, explaining what your tool does, how to install it, how to use it, and any known issues.

Generate documentation:

```
dart doc
```

This will generate HTML documentation in the `doc/api` directory.

## Packaging and distribution strategies

For a CLI tool, the primary distribution method is often through Dart\"s native executables.
```

Compiling to Native Executables

Dart can compile your CLI tool into a standalone executable for various platforms (Windows, macOS, Linux) using `dart compile exe`.

```
# Compile for your current platform
dart compile exe bin/quote_cli_tool.dart -o quote_cli_tool

# For a specific platform (e.g., Linux ARM64)
# dart compile exe bin/quote_cli_tool.dart -o quote_cli_tool_linux_arm64 --target-os linux --target-arch arm64
```

This creates a single executable file that can be run without installing the Dart SDK on the target machine.

Distributing via Pub.dev

If your CLI tool is intended to be installed globally by other Dart developers, you can publish it to Pub.dev. Users can then install it using `dart pub global activate`.

```
dart pub global activate quote_cli_tool
```

This mini-project provides a practical application of many Dart concepts, from basic syntax to asynchronous programming, error handling, and package management. Building such a tool helps solidify your understanding and prepares you for more complex Dart and Flutter development.