Dart Programming Essentials

Welcome to the very first step on your journey to mastering Flutter! Before we can build beautiful and performant mobile, web, and desktop applications with Flutter, we need to get comfortable with Dart, the programming language that powers Flutter. Dart is a modern, client-optimized language developed by Google, designed to be approachable, powerful, and highly productive for building user interfaces.

If you have prior experience with object-oriented languages like Java, C#, or dynamically-typed languages like JavaScript, you'll find Dart's syntax familiar and intuitive. Even if you're new to programming, Dart's clear structure makes it a great language to start with.

In this chapter, we'll lay a solid foundation in Dart. We won't be building a full Flutter app just yet. Instead, we'll focus on understanding the core building blocks of the Dart language through detailed explanations, practical use cases, and hands-on coding exercises. Think of this as meticulously learning the notes and scales before composing a symphony. This thorough understanding will be invaluable as we progress to more complex Flutter development.

Learning Objectives:

Upon completing this chapter, you will be able to:

- Understand and confidently use Dart's fundamental syntax, including how to declare variables with appropriate keywords (`var`, `final`, `const`) and work with a variety of built-in data types.
- Implement robust control flow in your Dart programs using conditional statements (if/else, switch) for decision-making and various loop constructs (for, while, for-in) for iteration.
- Define and call functions with different types of parameters (required, optional, named) and return types, and leverage Dart's concise arrow syntax for simple functions.
- Grasp the basic principles of Object-Oriented Programming (OOP) in Dart, including defining classes, creating objects (instances), and utilizing constructors, methods, properties, and basic inheritance to model real-world entities.
- Understand the importance and mechanics of Dart's sound null safety features, enabling you to write more robust, predictable, and error-free code by effectively managing null values.

Concepts and Technologies Covered:

This chapter will provide a comprehensive introduction to the following core Dart concepts:

1. Introduction to Dart:

- **What is Dart? Why was it created?**
  - Dart is an open-source, general-purpose programming language initially developed by Google. It was designed to be a more structured and scalable alternative to JavaScript for web development, but has since evolved to be a versatile language for mobile (Flutter), web (AngularDart, Flutter Web), server-side (Dart Frog), and even desktop applications.
- **Key Features:**
  - **Client-Optimized:** Dart is specifically optimized for building user interfaces, with features that support fast development cycles and high-performance applications.
  - **JIT (Just-In-Time) and AOT (Ahead-Of-Time) Compilation:**
    - **JIT Compilation:** During development, Dart uses a JIT compiler. This allows for features like "hot reload" in Flutter, where code changes can be injected into a running application almost instantaneously, drastically speeding up the development and iteration process.
    - **AOT Compilation:** For production releases, Dart can be AOT compiled into fast, predictable, native machine code. This results in applications that start quickly and run smoothly.
  - **Strongly Typed with Type Inference:** Dart is strongly typed, meaning every variable has a type that is known at compile time. This helps catch many errors early. However, Dart also features powerful type inference (`var` keyword), so you don't always have to explicitly write out the types, making the code less verbose.

    Generated dart

    ```dart
    var name = 'Alice'; // Type String is inferred
    int age = 30;       // Type int is explicitly declared
    // name = 123;        // This would cause a compile-time error
    ```
  - **Sound Null Safety:** One of Dart's standout features. By default, types are non-nullable, meaning variables cannot hold `null` unless explicitly declared. This eliminates a vast category of runtime errors (null pointer exceptions) at compile time. We'll cover this in detail later in the chapter.
  - **Object-Oriented:** Dart is a true object-oriented language. Everything is an object, including numbers, functions, and even `null`. All objects inherit from the `Object` class.

- **Setting up a Dart Practice Environment:**
  - **DartPad (dartpad.dev):** The easiest way to start. It's a free, online editor that lets you write and run Dart code directly in your browser. Perfect for the exercises in this chapter.
  - **Local Dart SDK:** For more serious development, you'll install the Dart SDK. This provides the `dart` command-line tool for running Dart scripts, compiling code, and managing packages. You can download it from dart.dev/get-dart.
  - **IDE (Integrated Development Environment):** Tools like Visual Studio Code (VS Code) or IntelliJ IDEA (with the Dart plugin) offer excellent Dart support, including code completion, debugging, and refactoring tools.

---

2. Variables and Data Types:

Variables are named storage locations for data.

- **Declaring Variables:**
  - `var`: The most common way. The compiler infers the type. Once inferred, the type cannot change.
    ```
    var greeting = "Hello, Dart!"; // Inferred as String
    var year = 2024;               // Inferred as int
    // greeting = 100; // Error: A value of type 'int' can't be
    assigned to a variable of type 'String'.
    ```
  - `final`: Used for single-assignment variables. Once a `final` variable is set, its value cannot be changed. The value is determined at runtime.
    ```
    final String appName = "MyCoolApp";
    // appName = "AnotherApp"; // Error: The final variable 'appName'
    can only be set once.
    final currentTime = DateTime.now(); // Value set at runtime, this
    is valid.
    ```
  - `const`: Used for compile-time constants. The value must be known at compile-time. `const` variables are implicitly `final`.
    ```
    const double pi = 3.14159;
    const appVersion = "1.0.0";
    // const buildTime = DateTime.now(); // Error: Const variables
    must be initialized with a constant value.
    ```
  - **Best Practice:** Prefer `const` for values that are true constants. Prefer `final` for values that are set only once. Use `var` (or an explicit type) only when a variable's value needs to change over time.

- **Built-in Data Types:**
  - **Numbers:**
    - `int`: Integer values (whole numbers).
    - `double`: Floating-point numbers (with a decimal point).
    - `num`: A type that can be either an `int` or a `double`.

```
int score = 100;
int temperature = -5;
double price = 19.99;
double gravity = 9.81;

num anyNumber = 5;       // It's an int
print(anyNumber);
anyNumber = 3.14;    // Now it's a double
print(anyNumber);
```

  - **Strings:** Sequences of UTF-16 code units (characters).

```
String singleQuote = 'Single quotes work.';
String doubleQuote = "Double quotes work too.";
String multiLine = """
This is a
multi-line string.
""";

// String Interpolation
String name = "Dave";
int points = 75;
String userInfo = "User: $name, Points: $points"; // Output:
User: Dave, Points: 75
String calculation = "2 + 2 = ${2 + 2}";          // Output: 2 +
2 = 4

// Common Methods
String yelling = "hello world".toUpperCase(); // "HELLO WORLD"
String whispering = "I AM WHISPERING".toLowerCase(); // "i am
whispering"
String messy = "  some text   ";
String clean = messy.trim(); // "some text"
```

  - **Booleans:** Either `true` or `false`.

```
bool isLoggedIn = true;
bool hasPermission = false;
var isEmpty = "".isEmpty; // true
```

- **Lists:** An ordered collection of objects (like an array).

```
List<String> fruits = ["Apple", "Banana", "Orange"];
print(fruits[0]); // Output: Apple

fruits.add("Mango");       // Add an item
fruits.insert(1, "Kiwi"); // Insert at an index
print(fruits); // [Apple, Kiwi, Banana, Orange, Mango]

fruits.remove("Banana");
print(fruits.length); // 4
print(fruits.first);  // Apple
print(fruits.last);   // Mango
```

- **Sets:** An unordered collection of *unique* items.

```
Set<String> uniqueColors = {"Red", "Green", "Blue"};
var numbersSet = {1, 2, 3, 1, 2}; // Duplicates are ignored

print(numbersSet); // Output: {1, 2, 3}
uniqueColors.add("Yellow");
uniqueColors.add("Red"); // "Red" is already there, no change
print(uniqueColors.contains("Green")); // Output: true
```

- **Maps:** A collection of key-value pairs (like a dictionary or hash map). Keys are unique.

```
Map<String, String> capitals = {
  "USA": "Washington D.C.",
  "UK": "London",
  "Japan": "Tokyo"
};

// Accessing a value
print(capitals["UK"]); // Output: London

// Adding a new entry
capitals["Germany"] = "Berlin";

// Checking for keys and values
print(capitals.containsKey("France")); // false
print(capitals.containsValue("Tokyo")); // true
```

3. Operators:

Special symbols that perform operations on operands.

- **Arithmetic Operators:** `+`, `-`, `*`, `/` (always returns a double), `~/` (integer division), `%` (modulo).
- **Relational and Type Test Operators:** `==`, `!=`, `>`, `<`, `>=`, `<=`, `is`, `is!`.

```
dynamic value = "hello";
print(value is String); // true
print(value is! int);   // true

// Using `is` for type promotion is safe
if (value is String) {
  // The compiler now knows `value` is a String here
  print(value.toUpperCase());
}
```

- **Assignment Operators:** `=`, `+=`, `-=`, `*=`, `??=` (null-aware assignment).

```
String? name; // Nullable string
name ??= "Guest"; // Assigns "Guest" because name is null
print(name);      // Output: Guest

name ??= "User";  // Does not assign "User" because name is already
"Guest"
print(name);      // Output: Guest
```

- **Logical Operators:** `&&` (AND), `||` (OR), `!` (NOT).
- **Conditional Expressions:**
  - `condition ? expr1 : expr2` (ternary operator)
  - `expr1 ?? expr2` (if-null or null-coalescing operator)

```
String? preferredName; // null
String displayName = preferredName ?? "Default User";
print(displayName); // Output: Default User
```

- **Cascade Notation (`..`):** Allows you to make a sequence of operations on the same object.

```
// Without cascade
var list = [];
list.add(1);
list.add(2);
list.add(3);
```

```dart
// With cascade - more fluent and readable
var list2 = []
  ..add(1)
  ..add(2)
  ..add(3);

// Very useful for configuring objects, e.g., in Flutter
// var paint = Paint()
//   ..color = Colors.blue
//   ..strokeWidth = 5.0;
```

---

4. Control Flow Statements:

- **Conditional Statements:**
  - `if`, `else if`, `else`: Execute blocks of code based on boolean conditions.
  - `switch`: Selects a code block based on the value of an expression.

```dart
String grade = 'B';
switch (grade) {
  case 'A':
    print('Excellent!');
    break;
  case 'B':
    print('Good job!');
    break;
  case 'C':
    print('Fair.');
    break;
  default:
    print('Needs improvement.');
}
// Output: Good job!
```

- **Loops:**
  - `for loop`: Traditional loop with initializer, condition, and increment.

```dart
for (int i = 1; i <= 3; i++) {
  print("Number: $i");
}
```

  - `for-in loop`: Iterates over the elements of an iterable collection.

```dart
var fruits = ["Apple", "Banana", "Orange"];
for (var fruit in fruits) {
  print("I like to eat $fruit");
}
```

- `while loop`: Repeats a block as long as a condition is true (checked *before* iteration).
- `do-while loop`: Repeats a block as long as a condition is true (checked *after* iteration, so it always runs at least once).
- `break` and `continue`: `break` exits the loop; `continue` skips to the next iteration.

---

5. Functions:

Reusable blocks of code that perform a specific task.

- **Defining and Calling:**

```
// Function with a return type and parameters
String greet(String name) {
  return "Hello, $name!";
}

// Calling the function
String myGreeting = greet("World"); // "Hello, World!"
```

- **Parameters:**
  - **Required Positional:** Must be provided in order. `int add(int a, int b)`
  - **Optional Positional `[]`:** Can be omitted; defaults to `null` if no default value is

```
String describe(String name, [String? title]) {

  return title == null ? name : "$title. $name";
}
print(describe("Smith")); // "Smith"
print(describe("Smith", "Dr")); // "Dr. Smith"
```

  - **Optional Named `{}`:** Identified by name, can be in any order. Use `required` for mandatory named parameters.

    Generated dart

```
void printUserDetails({required String name, int? age, String
city = "Unknown"}) {
  print("Name: $name");
  if (age != null) {
    print("Age: $age");
  }
```

```
          print("City: $city");
        }

        printUserDetails(name: "Grace", city: "London");
```

- **Arrow Syntax (=>):** A shorthand for functions with a single expression.

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

  // Arrow syntax
  int multiplyShort(int a, int b) => a * b;
  ```
- **Anonymous Functions (Closures/Lambdas):** Functions without a name.

  Generated dart

  ```
  var numbers = [1, 2, 3];
  // Using an anonymous function with forEach
  numbers.forEach((number) {
    print("Number is $number");
  });

  // Using an anonymous function with .map to transform a list
  var squares = numbers.map((n) => n * n).toList();
  print(squares); // Output: [1, 4, 9]
  ```

6. Introduction to Object-Oriented Programming (OOP):

OOP is a paradigm based on "objects", which bundle data (properties) and behavior (methods) together. Think of it as creating custom blueprints for the data structures in your application.

- **Classes and Objects: The Blueprint and the Product**
  - A **`class`** is a blueprint. For example, a `Product` class defines what every product in an e-commerce app *must* have: a name, a price, and an inventory count. It also defines what every product *can do*: display its information or apply a discount.
  - An **`object` (or instance)** is the real thing created from the blueprint. If `Product` is the blueprint, then a specific "Laptop" or "T-Shirt" in your store are the objects. Each object has its own values for the properties defined in the class (e.g., the laptop has its own price, the t-shirt has its own).
- **Let's build a `Product` class:**

```
class Product {
  // 1. Properties (Instance Variables): Data for the object.
  // The underscore `_` makes a property private to its own file.
  String name;
  double price;
  int _inventoryCount;

  // 2. Constructor: A special method to create and initialize objects.
  // This is a shorthand constructor. It automatically assigns the
  // parameters to the properties with the same name.
  Product(this.name, this.price, this._inventoryCount);

  // 3. Named Constructor: An alternative way to create an object.
  // Useful for creating objects from different data sources, like a
Map (e.g. from JSON).
  Product.fromMap(Map<String, dynamic> map)
    : name = map['name'] ?? 'No Name',
      price = map['price'] ?? 0.0,
      _inventoryCount = map['stock'] ?? 0;

  // 4. Methods: Behavior for the object. Functions inside a class.
  void displayInfo() {
    print("Product: $name");
    print("Price: \$${price.toStringAsFixed(2)}");
    print("In Stock: ${_inventoryCount > 0}");
  }

  // 5. Getter: A special method to provide read-access to a property.
  // Allows us to expose a computed value or a private property safely.
  int get stock => _inventoryCount;

  // 6. Setter: A special method to provide write-access to a property.
  // Allows us to add validation logic before changing a value.
  set stock(int value) {
    if (value >= 0) {
      _inventoryCount = value;
    } else {
      print("Inventory count cannot be negative.");
    }
  }
}
```

- **Using our `Product` class:**

  ```
  // Create an object using the main constructor
  var laptop = Product("Super Laptop", 999.99, 15);
  laptop.displayInfo();
  // Output:
  // Product: Super Laptop
  // Price: $999.99
  // In Stock: true

  // Accessing a getter
  print("Items in stock: ${laptop.stock}"); // Output: 15

  // Using the setter
  laptop.stock = 10; // Sets the stock to 10
  laptop.stock = -5; // Prints "Inventory count cannot be negative."
  print("Updated items in stock: ${laptop.stock}"); // Output: 10
  ```

- **Inheritance: Building on Existing Blueprints**
  Inheritance allows a class (the child) to inherit properties and methods from another class (the parent). This promotes code reuse. A `Book` is a *type of* `Product`, so it can inherit from `Product` and add its own specific properties.

  ```
  // Book is the child, Product is the parent.
  class Book extends Product {
  ```

```
  String author;

  // The Book constructor takes its own parameters plus the ones needed
by Product.
  // `super` calls the parent class's constructor.
  Book(String name, double price, int stock, this.author)
      : super(name, price, stock);

  // `@override` indicates that we are intentionally replacing a method
from the parent.
  @override
  void displayInfo() {
    super.displayInfo(); // Call the parent's method first to avoid
repetition.
    print("Author: $author"); // Then add the child-specific
information.
  }
}
```

- **Using the Book class:**

```
var myBook = Book("The Dart Apprentice", 29.99, 50, "Ray Wenderlich
Team");
myBook.displayInfo();
// Output:
// Product: The Dart Apprentice
// Price: $29.99
// In Stock: true
// Author: Ray Wenderlich Team
```

7. Null Safety:

A cornerstone of modern Dart, designed to prevent null reference errors.

- **Understanding the Problem:** Historically, `null` could be assigned to any variable, leading to runtime crashes if code tried to use a `null` variable as if it held a real object.
- **Dart's Sound Null Safety:**
  - **Non-Nullable Types by Default:** Types are non-nullable unless you explicitly say they can be `null`.

    ```
    int myAge = 30;
    // myAge = null; // Error: A value of type 'Null' can't be
    assigned to type 'int'.
    ```
  - **Nullable Types (`?`):** To declare that a variable *can* hold `null`, append a `?` to its type.

    ```
    String? middleName; // Can be a String or null. Defaults to null.
    print(middleName);   // Output: null
    middleName = "William";
    ```

    content_copydownload

- **Working with Nullable Types:**
  - **Null-aware Access Operator (`?.`):** Access a property/method only if the object is not `null`. If it is `null`, the expression evaluates to `null`.

    ```
    String? name; // null
    print(name?.length); // Output: null (no error)
    name = "Alice";
    print(name?.length); // Output: 5
    ```
  - **If-Null Operator (`??`):** Provides a default value if an expression is `null`.

    Generated dart

    ```
    String? userName;
    String displayName = userName ?? "Guest"; // If userName is null,
    use "Guest"
    print(displayName); // Output: Guest
    ```
  - **Assertion Operator (`!`):** (Use with extreme caution!) Tells the compiler you are *certain* an expression is not `null`. If it is `null` at runtime, your app will crash. Only use it when you have already performed a null check.

```
String? message;
message = "Data loaded";
if (message != null) {
  // Inside this block, Dart knows `message` is not null.
  print(message.toUpperCase());
  // Using `!` here is safe because of the check above.
  print(message!.length);
}
```

- `late` **Keyword:** Used for non-nullable variables that are initialized *after* their declaration but *before* they are first used. Dart trusts you to initialize it. If you access a `late` variable before initialization, a runtime error occurs.

```
class MyService {
  late String _data; // Declared now, will be initialized later.

  void fetchData() {
    // Simulate a network call
    _data = "Fetched important data";
  }

  String processData() {
    // If fetchData() hasn't been called, accessing _data here
would crash.
    return "Processing: $_data";
  }
}
```

Project Description: Dart Fundamentals Practice Exercises

In this chapter, we won't build a single, cohesive application. Instead, you will engage in a series of focused coding exercises designed to solidify your understanding of each Dart concept introduced. These exercises will be small, self-contained problems that you can solve directly in **DartPad (dartpad.dev)** or your local Dart environment (VS Code or IntelliJ with Dart SDK).

How to Approach:

1. Read the explanation for a concept (e.g., Variables and Data Types).
2. Study the provided code examples.
3. Try to replicate the examples yourself in DartPad.
4. Modify the examples: change values, add print statements, try to break them to see the error messages. This is a great way to learn!

5. Attempt the suggested exercises below.

Suggested Exercises (To be done as you learn each section):

- **Variables & Data Types:**
    1. Declare `final` variables for your first name and last name, and a `var` for your age. Print a sentence combining these.
    2. Create a `List<String>` of your top 3 favorite movies. Print the second movie. Add a fourth movie and print the whole list.
    3. Create a `Map<String, dynamic>` to store a product's details: name (String), price (double), inStock (bool). Print the product's name and price.
    4. Declare a `const` for the number of days in a week. Try to reassign it and observe the error.
- **Operators:**
    1. Write a program that takes a temperature in Celsius (double) and converts it to Fahrenheit (F = C * 9/5 + 32).
    2. Given two numbers, print `true` if their sum is greater than 100, `false` otherwise.
    3. Use the ternary operator: given a variable `isRaining` (bool), assign "Take an umbrella" or "Enjoy the sunshine" to a String variable.
    4. Declare a nullable String `username`. Use `??=` to assign "Guest" if it's null. Print it. Then assign a name to `username` and try `??=` again.
- **Control Flow:**
    1. Write a program that prints numbers from 10 down to 1 using a `for` loop.
    2. Using a `for-in` loop, iterate through your list of favorite movies and print each one prefixed with "Movie: ".
    3. Write a `switch` statement that takes a day of the week (String, e.g., "Monday") and prints "Workday" or "Weekend".
    4. Write a `while` loop that keeps asking for (simulated) input until the user types "exit". (For DartPad, you can simulate by having a list of inputs and iterating through it).
- **Functions:**
    1. Write a function `calculateArea(double width, double height)` that returns the area of a rectangle.
    2. Write a function `greetUser({required String name, String greeting = "Hello"})` that prints a personalized greeting. Call it with and without the optional greeting.
    3. Convert the `calculateArea` function to use arrow syntax.
    4. Create a list of numbers. Use the `forEach` method with an anonymous function to print the cube of each number.
- **OOP Basics:**
    1. Define a `Book` class with properties: `title` (String), `author` (String), `pages` (int).

2. Add a constructor to the `Book` class to initialize these properties.
3. Add a method `displayInfo()` to the `Book` class that prints "Title: [title], Author: [author], Pages: [pages]".
4. Create two `Book` objects and call their `displayInfo()` methods.
5. Add a named constructor `Book.shortStory(String title, String author)` that sets `pages` to a default value (e.g., 50).

- **Null Safety:**
  1. Declare a nullable `String? description`.
  2. Use the null-aware access operator `?.` to print its length if it's not null.
  3. Use the if-null operator `??` to provide a default "No description available" if `description` is null.
  4. Create a function that takes a `String?` parameter. Inside the function, if the parameter is not null, print its uppercase version. If it is null, print "Input was null". Test with both null and non-null values.
  5. Declare a `late String settingsData;`. Write a separate function `initializeSettings()` that assigns a value to `settingsData`. Call this function and then print `settingsData`.