Dart and Flutter

David Tang, Brian Dell, Jaclyn Wirth
CSIS-340: Dr. Jim Teresco
Programming Languages Final Project
11/30/23

ABSTRACT:

This research paper explores the Dart programming language, which is known for its efficiency in mobile app development. To demonstrate this characteristic of Dart, we developed a simple dynamic to-do list application. This application will show a practical demonstration of Darts capabilities. Dart was a programming language developed by Google. Dart's integration with Flutter allows for a single codebase deployable across multiple platforms. Flutter is an open-source framework developed by Google which allows programmers to build front-end and back-end applications using Dart as well as other languages, including Java, C, C++, and etc. Furthermore, we explore key features of this language such as its null safety mechanisms, the ability to have dynamic type variables, and its capability to compile native ARM or x64 code, and JavaScript for web applications.

Introducing the language

Dart is an open-source language developed by Google with the aim of allowing developers to use an object-oriented language with static type analysis. This language is unique in that it was designed with the goal of making the development process as comfortable and fast as possible for developers. Dart comes with a fairly extensive set of built-in tools, such as its own package manager, various compilers / transpilers, and a parser and formatter. We chose this language because we were interested in mobile app development. We discovered that Dart works well for high performance applications, is an overall popular language and its unique built-in features along with the Flutter framework. Flutter is used for building cross platform apps, such as javascript for web, native mobile, and desktop apps. In addition, the Flutter framework

compatibility allows for an ease-of-use development process that helps us create a distinctive user interface. Using Dart programming language with the Flutter framework to create a mobile application is our goal for this project. To this end, we are developing *To-Do,* an app that acts as a tool to remind the user of upcoming tasks and events, including features such as adding, and marking off reminders as done. Our primary goal for this project is to learn and understand more about Dart and Flutter.

We will explore how Dart addresses a list of language features studied in class such as memory management, readability, reliability, writability, and portability, dynamic and static binding. We also will showcase a few highlighting features of Dart like null safety, type safety, type inference,and its approach on object oriented programming. From there we will explain how our application we design works and on our experience working with Dart to build it.

How the language addresses list of language features studied

In this section we will examine how Dart as a programming language addresses multiple language features studied during this semester. Dart's memory management uses garbage collection which frees the programmer from having to allocate and deallocate memory manually. Dart allows for overloading operators. For example:

```
class Ratio{
  final int numerator, denominator;
  Ratio(this.numerator, this.denominator);

  Ratio operator +(Ratio r) => Ratio(numerator + r.numerator,
denominator + r.denominator);
  Ratio operator -(Ratio r) =>  Ratio(numerator - r.numerator,
denominator - r.denominator);

  @override
  bool operator ==(Object other) =>
      other is Ratio && numerator == other.numerator && denominator
== other.denominator;
}
```

In this example we were able to overload the +, - and == operators for the class Ratio. Overloading features will allow for programmers to have more writable and readable code. This will reduce the need to have equality methods, arithmetic methods, compared to methods for new

objects. This language for the most part is readable, writable, reliable and portable. For instance, because Dart is statically typed it clearly specifies the data types in the code which allows programs to easily understand type-related reasoning and data structures within the code. It also allows for code to be checked at compile time, exposing common errors before running the program, and reduces errors overall. Also, since developers do not need to allocate memory as mentioned before this also mitigates memory issues and helps make the code more reliable. While learning the unique features of Dart might be a little difficult depending on the programmer features like null safety, type inference , and Dart's syntax helps programmers to write code with relative ease. Also the framework Flutter, which is primarily used for Dart, lets developers create applications for desktop, mobile, web, in a single codebase making this language highly portable. However, there are still situations where a specific platform is needed for optimizations. Programmers must be aware of when upgrading their SDK because Dart is not backwards compatible and could potentially break their code.

Dart is able to provide both dynamic binding and static binding. For instance a programmer can use the modifier `late` to declare a variable that is not null and will only be initialized after it is declared. This is useful in cases where a programmer creates a variable but it may not be needed and initializing this variable is costly for the program. For example:

```
// This is the program's only call to readWholeBook().
late String book= readWholeBook(); // Lazily initialized.
```

In the above example `book` is never used then the function `readWholeBook()` is never called, this is called lazy initialization. This also fixes instances when analysis fails and Dart is unable to determine whether top level variables and instance variables are set. For example if a programmer created and set a variable and tries to use it but Dart does not see the variable was declared the programmer can fix it by modifying the variable as `late`. By declaring a variable with the `late` keyword modifier it creates a level of dynamic behavior because it allows the programmer to assign the value later in the program. An example of this is shown below:

```
late String test;
void main() {
  test ="Hello!";
  print(test);
}
```

Dart also provides two other keywords `final` and `const` which provide a static type binding. By using these keywords the programmer is sure that he will never change a variable. The difference between these two is that a variable declared final can only be set once and it's "value is assigned at runtime" (Rose, Ch1, Section8), while a const is implicitly final and is a compile time constant. An example of creating a final variable is shown below.

```
final test = "a";
final String testTwo = "B";

test = "C"; //Will throw error because final variable can only be set
once
```

A final object cannot be modified however its fields can be changed as shown below.

```
final student = new Student("David");
student.setName("Brian");
```

A programmer would use a const variable when they want a variable that cannot be changed at any point in the program after compile time. For example

```
const int monthsInYear = 12;
print('There are $daysInYear days in a year');
monthsInYear = 10; //would throw an error at compile time saying an
assignment cannot be made due because monthsInYear is designated as a
const
```

An interesting feature with const is if there are multiple instances of the same value for const Dart will create one object for the repeating value and it will be shared for all the instances. For example

```
const Text x = Text("test");
const Test y = Text("test");

void main() {
  print(identical(x, y));// will return true since x and y share the
same object
}
```

Dart is able to do this because const is checked at compile-time and is not able to change during run time so it will then optimize memory usage.

Highlighting features, and how they compare to other languages

The features of a programming language are crucial in determining the effectiveness and usability of that language. After taking a closer examination of Dart it has multiple characteristics that stand out and reveals how unique and useful this language is. A common feature that Dart shares with other languages is type safety. “It uses a combination of static type checking and runtime checks to ensure that a variable’s value always matches the variable’s static type, sometimes referred to as sound typing” (dart.dev/language The Dart type system). This allows the compiler to check for type errors at compile time instead of run time. While Dart requires types, the annotations for types are actually optional. For example, the code below shows var x being defined and initialized. However, there is no int annotation in front of x. This is because Dart uses type inference, which means the analyzer can reason a type for a field, variables, and methods. In this case, by using var, we let Dart infer the type for x, which would assign x as an int. So, reassigning it as 4.0 would cause a compile error since its type has been assumed to be an int.

```
var x = 3; // x is inferred as an int at compile time
x = 4.0;
```

Another way to cause Dart to infer the type is by using the final or dynamic to declare a variable. While it is less efficient to use dynamic to declare variables, it does allow for variables to change their type. For instance, if we were to change the previous example to:

```
dynamic x = 3; // x is inferred as an int at run time.
x = 4.0; //x is reassigned as a double at run time
```

It will no longer throw an error at compile time because x can now change its type.

Dart also implements a feature uncommon in most programming languages, such as null safety, which helps prevent errors from variables that are unintentionally set to null.. Its analyzer and compiler will flag if a variable that is not nullable “has either: not been assigned to null, been assigned a null value” (dart.dev/null-safety Sound null safety). Dart follows two main principles with their design to null safety. The first is that variables are non-nullable by default, and the second is that they are fully sound, meaning that null-related errors cannot happen at runtime. For instance, “If the type system determines that something isn’t null, then that thing can never be null” (dart.dev/null-safety Sound null safety). To show that a variable could have a null value, you add a ? to its type. For example :

```
int? x = null;
String? nullableString = "Hello";  // Variable can be null
String nonNullableString = "World"; // Variable cannot be null
```

You could also assign a value to a variable if its value is null by using ??

```
int? t;
t = null;
t ??= 4; //If t is null it will be assigned the value 4
```

In this example, t is a nullable int and is assigned to null, and then it will be assigned to the value four because it is null. You also can access the properties of a nullable object by using ?. as shown below.

```
print(Object().name?.toUpperCase());
```

If `name` in this instance is null, toUpperCase() will not be called, null will be printed, and no null exception error will be thrown.

Object Oriented Programming

Like most modern programming languages, Dart is an object-oriented programming language. According to the official Dart documentation, “Dart is an object-oriented language with classes and mixin-based inheritance” (dart.dev). This means that although every class has exactly one superclass, (except for Object which is at the top), a class body can be reused in multiple hierarchies. Dart has many advanced features surrounding object-oriented programming that separates it from other programming languages.

Mixins are a great example of this. “Mixins are a way of defining code that can be reused in multiple class hierarchies. They are intended to provide member implementations en masse” (dart.dev). Mixins are useful because they allow your class to adapt the methods and fields of any mixin it uses. This can allow developers to create shared functionalities and methods across multiple classes while reducing the need to rewrite the code each time. Although basic inheritance can accomplish the same task, mixins can accomplish this without needing to extend or implement a superclass. Additionally, a class that uses a mixin is not considered a subtype of said mixin. This is beneficial to developers because Dart only supports single inheritance, but a class can have multiple mixins. Similar to the syntax for extending a class, developers can use mixins by using the with keyword, followed by one or more mixin names in the class declaration.

```
class Human extends Mammal with Athlete {
  // ...
}

class Student extends Human with Athlete, Gamer, Chef{
  Student(String StudentName) {
    name = StudentName;
  }
}
```

The syntax for mixins requires that it has neither an extends clause nor any generative constructors. If one wants to restrict the types that can use a mixin, this can be done using the on keyword, followed by the type the mixin is restricted to. Lastly, in the rare case when you need to create both a mixin and class, you can create a mixin class. This means that it can function like both a mixin or a class of its own. Like normal mixins, mixin classes cannot have their own extends clauses. Additionally, because classes can't have an on clause, neither can mixin classes, however, the functionality using the on clause can be achieved by making your mixin class abstract and declaring an abstract method in your mixin class. This ensures that every class using this mixin implements this method. Mixins are a step up from the approaches to inheritance seen by languages like Java because you can inherit methods from more than one source. Mixins solve the problem of single inheritance, without needing to use multiple inheritance. Java attempts to solve this problem via interfaces, however the problem remains that the programmer must still implement each method of the interface that they override.

```
mixin Athlete on Human {
  int timesWon = 0;
  int yearsPlayed = 0;

  void incrementYearsPlayed() {//...}
  void incrementTimesWon() {//...}
}
```

Dart has numerous ways to utilize its constructors that makes it stand out compared to other programming languages. It provides a more convenient developer experience, for example, Dart includes a shorthand syntax for assigning instance variables that can be done by typing "this.variableName" as a parameter when declaring the constructor. When this constructor is

called, your instance variables will automatically be assigned before the body of your constructor runs. This syntax eliminates the need to manually assign your variables inside the constructor. Using this syntax also allows your variables to be null safe. Because your instance variables are initialized before the body of the constructor, they are technically assigned while the object is being created, and thus, their values are never null. If we were to assign these values in the body of the constructor, we would need to add a "?" to the declaration of our variables to make it nullable because until the line that assigns the variable executes, its value is null. In addition to the shorthand syntax, Dart provides another way to initialize variables before the body of the program runs via initializer lists. Instance variables initialized using an initializer list are initialized after the formal parameters from the shorthand syntax.

Additionally, when overloading constructors, Dart can create named constructors rather than having all of them be the exact same name. This greatly improves the readability of your program because it will make the purpose of a constructor obvious, rather than having to look at documentation. Factory constructors are another useful feature Dart provides. They are created by putting the factory keyword before a constructor. Factory constructors don't always return an instance of its class, it could also "return an instance from a cache, or it might return an instance of a subtype. Another use case for factory constructors is initializing a final variable using logic that can't be handled in the initializer list" (dart.dev/language/constructors Constructors). Because they don't always return an instance, factory constructors have no reference. Lastly, Dart provides const constructors, by putting the const keyword before a constructor, and making all the fields of a class final, you can create your class a compile time constant in your program, greatly increasing the efficiency of your program.

```
class Ratio {
  double numerator;
  double denominator;

 Ratio(this.numerator, this.denominator);
  // Sets the instance variables before the constructor body runs.
 // Using initializer list to set values
  Ratio({double numerator = 0.0, double denominator = 1.0})
      : numerator = numerator,
```

```
        denominator = denominator {
  }

 // named constructor
 Ratio.wholeNumber(this.numerator, {this.denominator = 1.0});

 // factory constructor
  factory Ratio(int numerator, int denominator) {
    return Ratio(numerator, denominator);
  }
}
```

When it comes to concurrency, Dart supports concurrency in a similar way to languages like javascript with keywords like async and await, which can execute code concurrently, but is still single threaded. Dart's implementation of this is slightly different from javascript in the sense that Dart uses an object called a Future, as supposed to Promises with javascript. There exist minor differences between the two, however the main functionalities of these two things are very similar. What makes Dart unique in comparison to other languages are isolates, which are completely independent units of execution that run concurrently. Each isolate has its own memory space, and each isolate communicates with another through message passing. Isolates allow your Dart program to take advantage of multi-core processors.

To-Do Application

Our to-do list application called To-do is programmed in Flutter, and simulates a reminders application where users can create tasks for their to-do list, and keep track of what tasks are completed using a checkbox. When a user presses the checkbox, the task goes to the done tab, where it can then be unchecked to go back to the doing tab.

Widgets are the backbone of the Flutter framework. Any item or component that is rendered by Flutter is a subtype of a widget. There are two main types of Widgets, stateless widgets and stateful widgets. Widgets are immutable, which is significant, because this means they can be declared using `const`, making the compile time constants. This means that if I have multiple of the same widget, with the same fields in my application, all of these widgets are the same object which is memory efficient.

When creating a widget, developers have the option to either create stateless widgets, or stateful widgets. Developer tools in VSCode make it really easy to create your own widget. Simply typing stateless, or stateful inside a Dart file will prompt VSCode to autocomplete a stub for a stateless widget or stateful widget (depending on your choice). Every widget has a build method that they must override, java programmers can think of this method as their repaint method, every time the widget is rendered, this method is called. Stateless widgets are simple, because they don't change, they simply display the information that you want them to. Stateful widgets are used when you want to display constantly changing information. A stateful widget has a mutable state object, which is what the widget displays. The state object then contains a build method that renders the widget. To update the state of your application, you can call the `setState()` method, which takes a void callback function as a parameter. The concept of a state will be very easily comprehended by React programmers, as the logic from this was taken as inspiration from the React framework. For example, a widget that displays a counter that increments each time I press a button would be a stateful widget, but the button itself is a stateless widget. The example code below shows an example for both a stateless and stateful widget. In addition the Flutter framework provides a multitude of prebuilt widgets for developers to program with.

```
class HelloWorld extends StatelessWidget {
  const HelloWorld({Key? key}) : super(key: key);

  @override
  Widget build(BuildContext context) {
    return const MaterialApp(
      home: Center(child: Text('Hello World')),
    );
  }
}
```

```
class Home extends StatefulWidget {
  const Home({super.key});

  @override
```

```
  State<Home> createState() => _HomeState();
}

class _HomeState extends State<Home> {
  int count = 0;
  @override
  Widget build(BuildContext context) {
    return GestureDetector(
        onTap: () {
          setState(() {
            count++;
          });
        },
        child: Center(child: Text("Count: $count")));
  }
}
```

When creating our to-do list app, we leveraged both stateful and stateless widgets to create a functional user interface. Our most significant challenge when programming was state management. In our app, variables such as the list of tasks that we are displaying need to be accessed from many different widgets throughout our large widget tree. Additionally, calls to `setState` need to be made from many different widgets as well. One way to solve this is to pass in all of your variables, and setState methods via parameters to your widgets, but that becomes unmaintainable with a very large amount of parameters. To solve this we used InheritedWidget, a pre-built Flutter widget that allows you to define variables that will be accessible from any widget that is below it in the widget tree. Despite the existence of InheritedWidget, state management remains the largest obstacle that developers face when programming with not just Flutter, but other frameworks that use a similar approach like react. To combat this, there are many packages that Flutter provides, each of which have their own abstraction to solve the issue of state management. Some of these packages include provider and bloc. We did not utilize these for our application because it is on the smaller side, however for a large, full scale application. It will be really challenging to build an app without the use of one of these packages.

Again, one of the biggest selling points of the Flutter framework is its efficiency, caused by its ability to compile to the web, as well as native machine code for any platform. This efficiency then is increased by the unique way that Flutter renders widgets. In a Flutter application, there are three different trees that run your application behind the scenes: The widget tree, the elements tree, and the render tree. As programmers, we only have direct access to the widget tree. The widget tree is simple, all of your widgets of your application, starting from the root widget, all the way down to the deepest child widget of your application form the widget tree. The element tree on the other hand contains the actual logical structure of the user interface. In other words, all of the complex details about your widgets are abstracted away by the element tree. Simply put, the widget tree contains a template for the element tree. Finally The render tree actually contains the objects that contain the information needed to render the app on your screen. The separation of the logical structure of the user interface to the widget tree and element tree adds to the efficiency of Flutter because of how lightweight widgets are. For example the widget tree of a verbose Flutter application may contain hundreds of widgets all of which are being refreshed constantly with calls to `setState`. To re-create all of the data for each widget in this tree each time would be inefficient. Instead only the widget tree is rebuilt, which is not a taxing process due to how lightweight widgets are. After the widget tree is rebuilt, the changes of the widget tree are compared with that of the element tree, then Flutter makes the necessary changes in the element tree to reflect the new widget tree.

The back-end of this application was programmed in Firebase, which integrates well with Flutter to form a cohesive tech stack. Firebase is a back-end app development platform that allows users to develop comprehensive and complex applications without the need of any code. Also developed by Google, Flutter provides packages that allow Firebase to integrate beautifully with Flutter. Firebase uses a NoSql database to store information called FirebaseFirestore. In our app, we both read and write to Firestore to retrieve tasks from the database, add a task to the database, and update the status of a task when a user marks it as done. To constantly update our list widget to display our tasks according to the database, we used Futures to return our data from the database asynchronously, then display the data when it is complete.

In conclusion, the Dart programming language and Flutter framework are modern solutions to a wide range of problems, entailing a detailed range of unique features such as null safety, type inference, mixins, static and dynamic binding, type safety, object oriented

programming and much more. Through the analysis of Dart, we were fully able to understand these features and applied it to *To-Do* application. However, we weren't fully able to grasp the entirety of the language due to the time constraints, some features we were not able to explore and implement. For example, we wanted to implement authentication with Firebase, which would have allowed us to include a login feature within the application. Moreover, there could have been different users and then we could have curated profiles specifically for each user. This is significant such that each user has their own task and only visible if they are logged in on their specific login. Another implementation we did not get too was a Calender utility. We were seeking to add a visual aspect allowing users to visibly see when their tasks are upcoming and due. Unfortunately, we also did not have enough time to complete this.

Citations

dart.dev/language The Dart type system, https://dart.dev/language/type-system#type-inference

dart.dev/language Constructors, https://dart.dev/language/constructors

dart.dev/null-safety Sound null safety, https://dart.dev/null-safety

dart.dev/resources, Breaking changes and deprecations, https://dart.dev/resources/breaking-changes

Rose, Richard, *Flutter and Dart Cookbook*. (2022). O'Reilly Media, Inc.