Programming Languages Final Project

David Tang, Brian Dell, Jaclyn Wirth
CSIS-340 : Dr Jim Teresco
11/30/23

Dart in Flutter

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 such 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 decided to choose 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. In addition, the Flutter framework compatibility allows for an ease-of-use development process that helps us create a distinctive user interface. The Dart programming language using the Flutter framework to create a mobile

application is our goal for this project. To this, we came up with *To-Do,* an app that acts as a tool to remind the user of upcoming tasks and events, including features such as adding, marking off, and a calendar to showcase all existing reminders. Our goal for this project is to learn and understand more about Dart and Flutter.

How the language addresses list of language features studied

Dart for memory management uses its own garbage collector which allows the programmer to not have to allocate and deallocate memory manually. Dart does allow 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 object Ratio. With this overloading feature it 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 portabile. 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 showing 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.

If a programmer were to upgrade their Dart SDK version it may break their code if it relies on previous behavior because the sdk does not maintain backward compatibility. However when there is a new language version the Dart SDK maintains backward compatibility for existing code, the code can only potentially break when the programmer upgrades the language version of their code. “The Dart SDK maintains compatibility for deprecated code, with a warning. Deprecations are then completely removed in a subsequent release, breaking any code that relies on the previous behavior” (dart.dev/resources Breaking changes and deprecations). When there are experimental changes they are not stable and may not have breaking notices.

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. 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 is created and a set a variable is used 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 while variable declared and its “value is assigned at runtime” (Rose) 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 int x = 25;
const int y = 25;

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 for 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 w are letting Dart infer the type for x, which it would assign it as an int. So reassigning it as 4.0 would cause a compile error since its type has been assumed as 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 an uncommon in some programming languages such as null safety which helps to prevent errors that occur 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, 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 will be assigned to the value you 4 because it is null. You also can access the properties of a nullable object by using ?.

```
print(Object().name?.ToUpperCase())
```

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 or 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.

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

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

Dart also has many usages for constructors that can be used that separate it from other programming languages and 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.

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.