Important Topics:

Basics of Dart: Variables, data types, functions, control flow statements (if-else, loops), and operators.

Classes and Objects: Defining classes, creating objects, constructors, inheritance, and polymorphism.

Functions and Methods: Declaring functions, named parameters, optional parameters, anonymous functions, and higher-order functions.

Asynchronous Programming: Using async and await keywords, Future and Stream objects, handling asynchronous operations, and working with futures and streams in Flutter.

Collections: Working with lists, maps, and sets, iterating over collections, manipulating data in collections, and using collection operators.

Error Handling: Handling exceptions and errors using try-catch blocks, throwing exceptions, and using the Dart exception hierarchy.

Libraries and Packages: Importing and using libraries and packages, managing dependencies, and using popular Dart packages in Flutter development.

Generics: Using generic classes, functions, and methods to write reusable code.

Mixins: Defining and using mixins to reuse code across multiple classes.

Dart Development Tools: Familiarize yourself with Dart development tools such as Dart SDK, Dart Pad, Dart DevTools, and Dart Analyzer.

Metadata and Callable classes:

Dart Date and Time:

Tools:

1. Dart Pad
2. Visual Studio Code
3. Android Studio

Final and Constant keywords:

Mutability:

final: A final variable can be assigned a value only once and is mutable until its value is set. Once the value is assigned, it cannot be changed or reassigned.

```
e.g. void main(){

final a = 10;

print(a);

}
```

Now in this program variable ‘a’ will not take memory if we initialize ‘a’ with the final keyword. It will take memory only if we print ‘a’ via the print method as “print(a)”.

const: A const variable is implicitly final, meaning it also cannot be reassigned once a value is assigned to it. However, unlike final variables, const variables must have their value known at compile-time and are immutable throughout the program's execution.

e.g.

```
void main(){

const a = 20;

print(a);

}
```

Now in the case of “const” if we only initialize ‘a’ it will take space in memory whether we print ‘a’ or not.

Bitwise Operator:

Shift Left

Code:

```
void main() {

  var a = 25;

  var b = 20;

  var c = 0;


  // Binary left shift Operator

  c = a << 2;


  print("c << 2 = ${c}");

}
```

Solution:

The logic of the program:

1. Variables a, b, and c are declared and assigned initial values.
   - a is assigned the value 25.
   - b is assigned the value 20.
   - c is assigned the value 0.
2. The line c = a << 2; performs a left shift operation on a by 2 positions.
   - The binary representation of a is 00000000000000000000000000011001.

- The left shift operator (<<) shifts the bits of a to the left by the specified number of positions.
- Shifting by 2 positions results in 00000000000000000000000001100100.

3. The result of the left shift operation is assigned to the variable c.

- c will hold the decimal value corresponding to the binary number 00000000000000000000000001100100, which is 100.

4. The line print ("c << 2 = ${c}"); prints the value of c to the console.

The output will be: c << 2 = 100.

Number Methods:

compareTo() method:

Compares the Unicode value of two characters or strings.

Code:

```
void main() {
 String a = "apple";
 String b = "banana";

 int comparisonResult = a.compareTo(b);

 if (comparisonResult < 0) {
  print("$a comes before $b");
 } else if (comparisonResult > 0) {
  print("$a comes after $b");
 } else {
  print("$a is equal to $b");
 }
}
```

Explanation:

- In this example, the compareTo() method is called on the string a and passed b as an argument. The comparison is performed character by character:
- The first character of a ('a') has a lower Unicode value than the first character of b ('b'), so it is considered "less than" b.
- Therefore, it will print "apple comes before banana".

Steps:

- Here are the rules for string comparison using compareTo():
- If the first character of the calling string (the string on which compareTo() is invoked) has a lower Unicode value than the first character of the argument string, it is considered "less than" the argument string.
- If the first character of the calling string has a higher Unicode value than the first character of the argument string, it is considered "greater than" the argument string.
- If the first characters of both strings have the same Unicode value, the comparison moves to the next character, and the process repeats until a difference is found or the end of either string is reached.
- If all characters of both strings are the same, the strings are considered equal.

UTF 16-code:

UTF-16 (16-bit Unicode Transformation Format) is a character encoding scheme that represents Unicode characters using 16-bit code units.

Symbols:

Symbols are used to use or extract the metadata (information or data) stored inside any class, method, object, variable, identifier, and library.

Metadata:

Metadata in programming refers to extra information or annotations that can be added to classes, functions, variables, or other elements in the code to provide additional context or instructions. In Dart and other programming languages, metadata is represented using special syntax and is typically enclosed in square brackets [].

Function:

Actual parameters = arguments e.g., mul(2,3);

Formal parameters e.g., void main(int a , int b);

Assert ():

In Dart, assert is a built-in language feature used as a debugging aid to check certain conditions and ensure that they are true during a program's development and testing phase. It is used to catch logical errors early and identify issues in the code. The assert statement takes a Boolean expression as its argument and throws an Assertion Error if the expression evaluates to false.

Lexical Scope:

- When we say that variable scope is decided at compile time, it means that the scope of a variable in a programming language, such as Dart, is determined during the compilation phase of the program, rather than at runtime.

- During compilation, the code is analysed by the compiler to determine the visibility and accessibility of variables. The compiler identifies where variables are declared and enforce the rules that govern their scope.
- The scope of a variable refers to the region or part of the program where the variable is visible and can be accessed. It is the portion of the code where the variable is valid and meaningful.

In languages with compile-time scoping, the compiler statically determines the scope of each variable based on the structure of the code, its location in the source code, and the rules defined by the language. Once the scope of a variable is established during compilation, it remains fixed throughout the execution of the program.

Compile-time scoping offers several advantages:

- Predictability: The scope of variables is known before the program runs, making it easier to reason about the behavior of the code.
- Efficient Memory Management: The compiler can optimize memory usage based on variable scope, allowing it to allocate and deallocate memory efficiently.
- Static Analysis: The compiler can perform static analysis to detect errors related to variable access and scoping before the program is executed.

Here's an example in Dart to illustrate compile-time scoping:

```
void main() {
int x = 10; // Variable 'x' has scope within the 'main' function.
if (x > 5) {
 int y = 20; // Variable 'y' has scope within the 'if' block.
print(x); // Variable 'x' is accessible here.
 print(y); // Variable 'y' is accessible here.
}
// print(y); // Uncommenting this line would cause a compilation error since 'y' is not in scope here.
}
```

- In this example, the scope of variable x includes the entire main function, while the scope of variable y is limited to the if block. Attempting to access y outside of the if block would result in a compilation error because y is not in scope at that point. By deciding variable scope at compile time, Dart provides predictability, clarity, and better opportunities for code optimization.

This Keyword:

This.modelname = modelname;

In the given Dart code, the line this.modelname = modelname; is a constructor assignment statement inside the Mobile class. It is used to assign the value of the constructor parameter modelname to the instance variable modelname of the class.

Let's break it down:

- this.modelname: this is a keyword that refers to the current instance of the class. When used with a dot notation (e.g., this.modelname), it represents the instance variable modelname of the class. It helps differentiate between the instance variable and the constructor parameter that have the same name.
- modelname: This is the parameter of the constructor. It represents the value passed to the constructor when creating an object of the class. It is a local variable within the constructor's scope.
- =: The assignment operator is used to assign the value of modelname (the parameter) to this.modelname (the instance variable).

When an object of the Mobile class is created using the constructor with specific arguments, the constructor will take the value provided for the modelname parameter and assign it to the instance variable modelname.

<u>Code:</u>

```
class Mobile {
  String modelname;


  Mobile(String modelname) {
    this.modelname = modelname; // Assign the value of 'modelname' parameter to the
instance variable.
  }
}


void main() {
  Mobile mob = Mobile("IPhone 11");
  print(mob.modelname); // Output: IPhone 11
}
```

<u>Var and (String, int, double):</u>

If you use "var" while variable declaration then it will not give an error because "var" will take space according to the initializing value, whenever you initialize it in another place in code. e.g.

```
var name;

var age;
```

But if you declare a variable with data type like int, String etc., you also have to initialize it while its declaration.

e.g.

String name = “Kashif”;

Int age = 25;

Normal Methods:

A "normal method" is a general term used to refer to any regular method that performs some action or computation within a class. It is the most common type of method in object-oriented programming. Normal methods can have any access modifier (public, private, protected) and can return values or be void (returning nothing). They can also take parameters or have no parameters.

Concrete Methods:

A "concrete method" is a specific type of normal method that provides a concrete (actual) implementation. In other words, a concrete method is a method that has a complete body with executable code, and it provides the actual behavior of the method. Concrete methods are opposite to abstract methods, which are defined in abstract classes or interfaces but lack implementation details.

Typedef function

Aliases:

In programming, an alias refers to a different name or identifier that is used to refer to the same entity. The primary purpose of creating an alias is to provide an alternative or more meaningful name for an existing entity, making the code easier to read, understand, and maintain

Synchronous Generator:

- Here's a simplified explanation step by step:
- A synchronous generator is a function that can generate a sequence of values.
- You use the sync* keyword before the function body to declare it as a synchronous generator.
- Instead of using return, you use yield inside the function to produce values.
- Each time the yield keyword is encountered, the generator produces a value and pauses temporarily.
- You can use the returned iterable object to retrieve these values one by one in a synchronized manner.

Isolates:

First code explanation:

- Here's a step-by-step explanation of what the code does:
- The code starts by importing the dart:isolate library, which provides support for working with isolates in Dart.
- The sayhii function takes a parameter msg and simply prints the message provided as an argument.
- In the main function, you use Isolate.spawn three times to create three new isolates.
- Each Isolate.spawn call creates a new isolate and runs the sayhii function with a different message as an argument. The messages passed are 'Hello!!', 'Whats up!!', and 'Welcome!!'.
- After spawning the isolates, the main function continues to execute without waiting for the isolates to finish their execution.
- The code then prints three messages: 'execution from main1', 'execution from main2', and 'execution from main3'.

Concurrency:

In Dart, we can use the concurrency mechanism with the help of isolates. Isolates are the blocks of code running isolated on the processor. You may find isolates relatable to Java's thread. If you don't know about the thread, read this article on the Java thread. The main difference between thread and isolates is that thread runs on the same shared memory while the isolates have their own individual memory, they do not have shared variables. The only way they can communicate is through messages that are sent at the time of creating the isolates.

Async and await keywords:

1. **Async and Await Keywords**: To handle this challenge, Dart provides two special keywords: async and await.
2. **Async Function:** When we declare a function as "async" using the async keyword, it means that this function can perform asynchronous tasks. Asynchronous tasks are tasks that might take some time to complete, like reading a file or making an HTTP request.
3. **Await Keyword**: When we call an asynchronous function inside another function, we can use the await keyword. This tells Dart to pause the execution of the current function and wait for the result of the asynchronous function before continuing. It helps us avoid having to deal with complex callbacks or chaining asynchronous calls manually.
4. **Future Type:** When a function uses the await keyword, it becomes a "Future." A Future is like a box that holds a value that will be available in the future. It allows us to deal with the result of an asynchronous operation once it completes.
5. The async and await keywords are allowed to implement asynchronous programming without using the Future API.

Date and Time in Dart:

UTC:

UTC stands for Coordinated Universal Time. It is the primary time standard by which the world regulates clocks and time. UTC is not subject to Daylight Saving Time adjustments and provides a consistent time reference globally. It is often used in scientific, technological, and international contexts.

Local Time Zone refers to the time standard used within a specific geographic region or area. Different parts of the world have their own local time zones, which are usually based on the Earth's rotation and the position of the sun in the sky. Local time zones can vary from UTC by a certain number of hours and minutes. They can also be influenced by factors like Daylight Saving Time (DST), where clocks are adjusted forward or backward by one hour to make better use of daylight during certain periods of the year.

For example, if UTC is 12:00 PM and the local time zone is UTC+2, then the local time would be 2:00 PM. However, during DST, the local time might become UTC+3, and the local time would then be 3:00 PM.

It's important to be aware of both UTC and local time zones, especially when dealing with international communication, travel, or any activity that involves coordinating events across different regions with varying time differences.

Epoch:

In computing, the "epoch" refers to a specific point in time that serves as a reference. In many systems, including Dart, the epoch is often set to January 1, 1970 (UTC), which is commonly used as the starting point for measuring time.