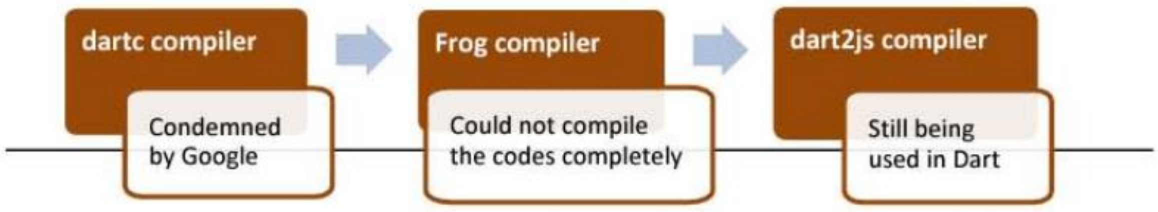
compilers used by Dart:



They are described as follows:

Just in Time (JIT)

JIT compiler converts program source code into native machine code just before the execution of a program. The JIT compilers can be used to improve speed and application runtime. They also try to predict the instructions that will be executed next so that the code can be compiled in advance.

Ahead of Time (AOT)

AOT compiler compiles the code before it is delivered to the runtime environment that runs the code. It usually is utilized when an application is ready for deployment either on the server, App Store, or the Play Store. It also compiles the Dart code into JavaScript code before the browser downloads and runs the code.

2.6 Map

The Map data type is utilized to represent data in key and value form. The items stored in this particular type can only be accessed using the associated key provided for the Map object.

A Map variable which is inside curly braces `{}` has each key-value pair separated from one another using commas. In Dart, Map can be declared using Map constructors or Map literals.

2.10.4 Create a Variable Using Dynamic Keyword

Variable declared using `dynamic` keywords do not have any predefined data type.

Code Snippet 6 shows how to declare a variable with the keyword `dynamic` and to assign several values to the declared variables.

2.10.5 Create a Variable Using Final Keyword

The `final` keyword is utilized for the creation of immutable objects in Dart. The variable with the `final` keyword is utilized when the user wants to always store the same value and does not want it to change.

Syntax:

```
final variable name = variable value;
```

2.10.6 Create a Variable Using const Keyword

The `const` keyword is utilized to let the user know that the value stored in the variable is constant and cannot change during program execution. If value for the said variable is known at compile-time, then `'const'` can be used over

`'final'`. The only difference between the `final` and `const` keyword is that `final` is a runtime-constant, which in turn means that its value can be assigned at runtime instead of the compile-time.

Instance Variable	Class Variable
Instance Variable is basically, a class variable shared by all class instances.	Class Variable is a static variable declared inside a class.
It can have different values for each object.	It can have only one value and is shared among all objects.
The value is retained as long as the object is active.	The value is retained until the program terminates.

Example: `class NewClass{`
`  int counter;`
`}`
	Example: `class NewClass{`
	`  static int counter;`
	`}`

Class Methods (Static Methods)

For declaring a class method, also called as a static method, the `static` keyword is used. A static method cannot be accessed through an object or instance. This avoids the necessity to create a new object each time for using the method, thus optimizing code. A static method is declared by using `static` keyword, method name, and return type.

Syntax:

```
static Returntype methodname()
{
    //statement
}
```

Method	Description	Syntax
Getter	Used to retrieve the value of a particular class field. It is defined using the `get` keyword. Data can only be read and not modified using the `get` method.	`returntype get fieldname{ }`
Setter	Used to set or override the value of a variable fetched using the `get` method. Defined using the `set` keyword.	`set fieldname(value) { }`

Instance Methods

Instance Methods are those methods which can be accessed only by using an object of the class. Instance methods can be accessed within a class too, through `this` keyword, which implies an implicit unnamed object.

A method will have a definition, a body, and a method call. An instance method is defined with a name, a return type, and a list of parameters. This is called its **signature**.

Syntax of creating an instance method with signature and body is as follows:

Syntax:

```
Returntype methodname(arguments)
{
   //statement body
}
```

An instance method can be 'called' or 'invoked' by an object anywhere in the program, provided the object has been defined.

4.6.1 Inheritance

The concept of a class inheriting properties and methods of another class is called Inheritance. The main class which has the properties and methods is called **parent** class and the one which inherits from the parent class is called child class. The keyword `extends` is used to inherit the properties from parent class to child class.

There are two types of inheritance in Dart:

Single level inheritance:

In single level inheritance, one child class is inherited by one parent class.

Multi-level inheritance:

In multi-level inheritance, a child class is inherited from another child class.

Syntax of Inheritance:

```
class ParentClass{
...
}
//single level inheritance
class ChildClass1 extends ParentClass
{
...
}
// multi level inheritance
class ChildClass2 extends ChildClass1{
```

4.7 Constructors in Dart

Constructors are special methods which have the same name as a class and are used to initialize an object. Whenever an object is created in a program, it automatically calls the constructor.

Syntax of Constructor:

```
class ClassName {
   ClassName () {
   }
}
```

Types of Constructors in Dart

There are three types of constructors which include:

- Default Constructor
- Parameterized Constructor
- Named Constructors

Parameterized Constructor

Parameterized constructors are those constructors which have parameters through which they take in some variables as arguments. These will help to decide which constructor will be called.

An example of a parameterized constructor is shown in Code Snippet 5.

Code Snippet 5:

```
 class Student{
   Student(String name){
     print("Student name is: ${name}");
   }
}
void main() {
   Student std = new Student("Ray");
```

Named Constructor

A named Constructor is used to create multiple constructors with different names in the same class.

An example of named constructor is shown in Code Snippet 6.

Code Snippet 6:

```
void main() {
  Student std1 = new Student();
  Student std2 = new Student.namedConst("Computer Science");
}
class Student{
  Student(){
    print("Default constructor");
  }
  Student.namedConst(String branch){
    print("Branch name is: ${branch}");
  }
}
```

this keyword

this keyword is included when parameters and attributes of the same name as class attributes are used. It refers to the current class object and this keyword removes ambiguity which is generated while declaring the same name of parameters and attributes.

An example of this keyword is shown in Code Snippet 7.

Code Snippet 7:

```
void main()
{
  Student s1 = new Student('101');
}
class Student
{
  var stid;
  Student(var stid)
  {
    this.stid = stid;
    print("Dart this keyword Example");
    print("Student ID is : ${stid}");
  }
}
```

super keyword

super keyword is used to refer to the parent class object. Methods and properties of parent class can be called with the help of super keyword. It also removes ambiguity between parent class method and child class that have the same method name and are called through objects.

An example of using the super keyword is shown in Code Snippet 8.

Code Snippet 8:

```
class ParentClass {
  String subject = "Example of Super Keyword";
}
class SubClass extends ParentClass {
  String subject = "Science";
  void showMessage(){
    print(super.subject);
    print("$subject has ${subject.length} letters.");
  }
}
void main(){
  SubClass myClass = new SubClass();
```

6.2.1 Asynchronous Programming with `Future` Keyword

`Future` keyword is used for getting a value sometimes in future. The asynchronous function returns Future, which contains the result.
Future objects are a mechanism to represent values returned by an expression whose execution will be complete at a later point in time.

b. Future error
Future error creates a future, which completes with an error. Code Snippet 4 demonstrates an example of Future error.

Code Snippet 4:

```
import 'dart:async';
void main(){

Future<int> getFuture() {
    return Future.error("This is an error");
  }
 getFuture();
}
```

c. Future delayed
Future delayed creates a future, which runs its computation after a delay. Future delay always works with a certain amount of duration. Computation will be executed after a given duration has passed. Code Snippet 5 is an example of Future delay.

Code Snippet 5:

```
import 'dart:async';
void main(){
Future.delayed(Duration(milliseconds: 10000), () {
print("This is a delayed future");
});
}
```

6.2.2 Asynchronous Programming with Async keyword

The `async` keyword is used when declaring a function as asynchronous. The `async` keyword is added after the function name. When an `async` function is called, Future is returned and the function is executed.

Syntax:

```
Function_name() async {
}
```

6.2.3 Asynchronous Programming with Await keyword

The `await` keyword is used only on the asynchronous function. The `await` keyword holds the currently running function until the result is ready. If the result is ready, then it continues execution from the next line of code. Code Snippet 6 demonstrates an example of `await` keyword.

Code Snippet 6: