1. Abstract Classes:

- an abstract class is a class that cannot be instantiated and is meant to serve as a base for other classes.
- It may contain abstract methods (methods without an implementation) that must be overridden by the concrete subclasses.
- Abstract classes are used to define a common interface and provide common behavior for related classes.
- Concrete classes derived from an abstract class inherit its properties and methods while providing their own implementations.

To define a class that doesn't require a full, concrete implementation of its entire interface, use the abstract modifier.

An Abstract class in Dart is defined as those classes which contain one or more than one abstract method (methods without implementation) in them. Whereas, to declare an abstract class we make use of the abstract keyword. So, it must be noted that a class declared abstract may or may not include abstract methods but if it includes an abstract method then it must be an abstract class.

Features of Abstract Class:

1. Instantiation: Abstract classes cannot be instantiated directly, meaning you cannot create objects of an abstract class using the new keyword. On the other hand, normal classes can be instantiated to create objects.

2. Implementation: Abstract classes can contain abstract methods, which are methods without a body. Subclasses inheriting from an abstract class must implement these abstract methods. Normal classes, however, do not require their methods to be implemented by subclasses.

3. Inheritance: Abstract classes are designed to be extended by subclasses. Subclasses inherit the properties and methods of the abstract class and can provide their own implementations. Normal classes can also be inherited, but it's not mandatory for the subclasses to provide their own implementations of methods.

4. Usage: Abstract classes are often used when you want to define a common interface or behavior that multiple classes should adhere to. They provide a way to ensure consistency and enforce certain methods to be implemented by subclasses. They are commonly used in design patterns like the Factory pattern or as base classes for creating polymorphic relationships.

2. static:

is a keyword that can be used to define class members (variables and methods) that belong to the class itself rather than to instances of the class.

1- Static Variables:

Static variables are declared using the “static” keyword within a class. They are shared among all instances of the class. Static variables are initialized only once and retain their values across multiple instances. They can be accessed using the class name itself, without creating an instance.

2- Static Methods:
Static methods belong to the class itself, rather than the instances of the class. They can be called without creating an instance of the class. Static methods are useful for performing operations or computations that are not specific to any particular instance.

3- Utility Classes:
Utility classes are classes that provide helper functions or constants that can be accessed globally without the need for creating instances. They often have private constructors to prevent instantiation.

4- Sharing Data:
Static variables can be used to share data between different parts of an application, such as screens or widgets. They provide a convenient way to store and access data that needs to be shared without passing it through constructors or function parameters.

5- Performance Considerations:
While static variables and methods can be beneficial in certain scenarios, their excessive use can lead to tight coupling and make code harder to maintain and test. Additionally, static methods cannot be overridden in subclasses, which may limit flexibility in some cases. It’s important to use static members judiciously.

6- Class-level State:
Static variables can be used to maintain class-level state, meaning that the variable retains its value across multiple instances of the class. This can be useful when you want to keep track of information that is relevant to the entire class rather than individual instances.

7- Initialization Order:
Static variables are initialized before any instance of the class is created. This means that you can access and use static variables even without creating an instance of the class.

3. Encapsulation:

is a mechanism for hiding important and sensitive data from users. To use encapsulation, you make the field private and use the public getters and setters to access and set the value of that field. In Dart, encapsulation is done at the library level, not at the class level. To provide default getters and setters, you can get and set values directly using field names. We can say encapsulation is building code into a single unit where you can determine the scope of each piece of data.

In Dart, Encapsulation means hiding data within a library, preventing it from outside factors. It helps you control your program and prevent it from becoming too complicated.

Encapsulation can be achieved by:

Declaring the class properties as private by using underscore(_).
Providing public getter and setter methods to access and update the value of private property.

the importance of Encapsulation :
1- Data Hiding: Encapsulation hides the data from the outside world. It prevents the data from being accessed by the code outside the class. This is known as data hiding.

2- Testability: Encapsulation allows you to test the class in isolation. It will enable you to test the class without testing the code outside the class.

3- Flexibility: Encapsulation allows you to change the implementation of the class without affecting the code outside the class.

4- Security: Encapsulation allows you to restrict access to the class members. It will enable you to limit access to the class members from the code outside the library

4. Polymorphism:

in Dart refers to the ability of objects of different classes to be treated as objects of a common superclass. It allows you to write code that can work with objects of different types, as long as they share a common interface or superclass.

polymorphism in Dart is a powerful concept that enhances code flexibility, extensibility, and indirectly promotes code reusability when used effectively. It allows you to work with diverse classes through a common interface or superclass, making your code more adaptable and easier to maintain.

There are two main forms of polymorphism in Dart:

1- Static Polymorphism (Compile-Time Polymorphism):
In Dart, static polymorphism is not achieved through traditional method overloading, as seen in some other programming languages like Java or C++. Dart does not support defining multiple methods with the same name in a class that only differs in their parameter lists. Instead, Dart encourages you to either use different names for methods or utilize optional named or unnamed parameters to achieve similar functionality.

2- Dynamic Polymorphism (Runtime Polymorphism):
Dart supports dynamic polymorphism, primarily through method overriding. Method overriding allows a subclass to provide its own implementation of a method defined in its superclass. The appropriate method to execute is determined at runtime based on the actual type of the object.

Advantages of Polymorphism:
Polymorphism offers several advantages in Dart:

1- Code Flexibility:

You can write more generic and flexible code that can work with different types of objects, promoting modularity and reducing dependencies on specific class implementations.

2- Extensibility:
Polymorphism enables you to extend existing classes by creating new subclasses that can override or extend the behavior of the superclass.

3- Code Reusability:
While polymorphism primarily provides code flexibility and extensibility, it indirectly promotes code reusability by allowing you to use common interfaces or superclasses in your codebase. This can lead to more maintainable and scalable software