**B1. What is inheritance?**

Ans. Inheritance is a key feature of object-oriented programming that enables the creation of new classes based on existing classes, inheriting their attributes and behaviours. This allows for the creation of more complex and sophisticated classes from simpler ones, promoting code reuse and reducing development time and effort. Inheritance also facilitates the creation of hierarchical relationships between classes, where derived classes inherit properties and methods from a parent or base class, thereby making the code more organized and easier to understand. Moreover, inheritance allows for the creation of more specialized classes, which can be customized to meet specific requirements while still retaining the functionality of their parent classes.

**B2. Which inheritance is not supported by Dart? Why?**

Ans. Dart does not support multiple inheritance. Multiple inheritance is a feature in some programming languages where a subclass can inherit properties and methods from more than one superclass.

**B3. What is advantage of inheritance?**

Ans. 1. Code Reuse: Inheritance allows subclasses to inherit properties and methods from a superclass, reducing the amount of code that needs to be written. This saves development time and improves code maintainability, since changes to the superclass automatically propagate to all subclasses.

2. Polymorphism: Inheritance enables the concept of polymorphism, which means that objects of different classes can be treated as if they were of the same class. This allows for greater flexibility in code design and implementation, since it allows for more generalized code that can work with different types of objects.

**B4. Difference between inheritance and encapsulation.**

* Inheritance and encapsulation are two fundamental concepts in object-oriented programming. The key differences between the two are as follows:
* Inheritance refers to the ability of a class to inherit properties and behaviours from a parent or superclass. This means that a subclass can extend or modify the functionality of the superclass. On the other hand, encapsulation refers to the practice of hiding internal details of an object from the outside world, by making its attributes and methods private and only allowing access through public interfaces.

**B5. Difference between inheritance and abstraction.**

Ans. Inheritance is a mechanism for creating a new class that is a modified version of an existing class, called the superclass. The new class, called the subclass, inherits all the properties and methods of the superclass, and may also have additional properties and methods of its own. Inheritance is used to promote code reuse, since the common properties and methods of the superclass are already defined, and the subclass only needs to define its own unique properties and methods. Inheritance creates an "is-a" relationship between the subclass and the superclass, indicating that the subclass is a specific type of the superclass.

Abstraction, on the other hand, is a mechanism for hiding the details of an implementation, and exposing only a simplified and high-level view of the functionality. Abstraction is used to manage complexity and increase modularity, by breaking down a system into smaller, more manageable components. In object-oriented programming, abstraction is implemented using abstract classes and interfaces, which define a contract that a class must conform to, but do not provide an implementation for all the methods. This allows different classes to implement the same interface or extend the same abstract class, but provide their own unique implementation for the methods.

**B6. Difference between inheritance and polymorphism.**

Ans. Inheritance is a mechanism for creating new classes that are derived from existing classes. The new class, called the subclass, inherits all the properties and methods of the existing class, called the superclass. Inheritance is used to promote code reuse and to create relationships between classes, such as "is-a" and "part-of" relationships.

Polymorphism, on the other hand, is a mechanism for using a single interface to represent multiple types. Polymorphism allows different types to be treated as if they were the same type, by using a common interface to interact with them. Polymorphism is achieved through inheritance, where a subclass can be used in place of its superclass, and through interfaces, where different classes can implement the same interface and be treated as if they were the same type.

**B7. Can we override static method in Dart?**

Ans. No, we cannot override a static method in Dart. This is because static methods belong to a class rather than an instance of a class, and therefore they cannot be overridden by a subclass. In Dart, when we declare a static method in a superclass, the same method is available to all subclasses, but it cannot be overridden. However, we can use a subclass to hide a static method in the superclass by declaring a static method with the same name in the subclass. In this case, the static method in the subclass will be called instead of the one in the superclass.

**B8. Can we overload static method in Dart?**

Ans. No, we cannot overload static methods in Dart. Overloading is a feature of object-oriented programming that allows multiple methods with the same name but different parameters to coexist in the same class. When a method is called, the appropriate version is selected based on the number and type of the arguments.

**B9. Can a class implement more than one interface?**

Ans. Yes, a class can implement more than one interface in Dart. Implementing an interface means that a class promises to implement all the methods and properties defined in the interface. By implementing multiple interfaces, a class can gain the abilities of all the interfaces it implements and be used in a variety of contexts.

To implement multiple interfaces in Dart, the class declaration must include a comma-separated list of the interface names after the implements keyword. For example:

class MyClass implements Interface1, Interface2 {

// class body

}

In this example, the class MyClass implements both Interface1 and Interface2. When a class implements multiple interfaces, it must provide implementations for all the methods and properties defined in each interface.

Implementing multiple interfaces can be useful when you need to provide different behaviours for different contexts or when you want to ensure that a class conforms to a specific set of rules or standards. It allows for greater flexibility and modularity in your code, and it can make your classes more reusable and adaptable.

**B10. Can a class extend more than one class in Dart?**

Ans. No, a class cannot extend more than one class in Dart. Dart does not support multiple inheritance, which is a feature of object-oriented programming that allows a class to inherit properties and behaviours from multiple classes. Multiple inheritance can lead to a complex class hierarchy and can make code harder to understand and maintain.

**B11. Can an interface extend more than one interface in Dart?**

Ans. Yes, an interface can extend more than one interface in Dart. In fact, this is a common practice in object-oriented programming, where one interface can build on the definitions of one or more other interfaces. Extending multiple interfaces allows an interface to inherit the properties and methods defined in those interfaces, and to define additional properties and methods of its own.

To extend multiple interfaces in Dart, an interface declaration includes a comma-separated list of the interface names that it extends, after the extends keyword.

For example:

abstract class MyInterface extends Interface1, Interface2 {

// interface body

}

In this example, the interface MyInterface extends both Interface1 and Interface2, inheriting their properties and methods.

Extending multiple interfaces is useful when you need to create a more specialized interface that combines the properties and methods of existing interfaces. This can help to promote code reuse, reduce redundancy, and improve the maintainability of your code. By breaking down functionality into smaller, modular interfaces, you can create a more flexible and extensible system that can be adapted to changing requirements over time.

**B12. What will happen if a class implements two interfaces and they both have a method with same name and signature?**

Ans. If a class implements two interfaces that both have a method with the same name and signature, the class must provide an implementation for that method that satisfies both interfaces. This is because interfaces define a contract that a class must conform to, and if two interfaces define a method with the same name and signature, the class must provide an implementation that satisfies both contracts.

If the method implementation in the class does not satisfy the requirements of both interfaces, a compile-time error will occur. If the method implementation satisfies the requirements of both interfaces, it can be called using a reference to either interface type, and the behaviour will be the same.

For example, suppose we have two interfaces, Interface1 and Interface2, each with a method named foo that takes an integer argument and returns a Boolean value. If a class MyClass implements both interfaces, it must provide an implementation for the foo method that satisfies both interfaces:

class MyClass implements Interface1, Interface2 {

bool foo(int x) {

// implementation that satisfies both interfaces

}

}

In this example, the MyClass implementation of the foo method satisfies the requirements of both Interface1 and Interface2, and it can be called using a reference to either interface type.

**B13. Can we pass an object of a subclass to a method expecting an object of the super class?**

Ans. Yes, you can pass an object of a subclass to a method that expects an object of the superclass. This is known as upcasting, which is a feature of object-oriented programming that allows a subclass instance to be treated as if it were an instance of its superclass.

Upcasting is possible because a subclass inherits all the properties and methods of its superclass, and may also have additional properties and methods of its own. This means that an object of the subclass can be treated as if it were an object of the superclass, since it has all the same properties and methods that the superclass expects.

For example, suppose we have a class Animal and a subclass Cat: class Animal {

void makeSound() {

print('Animal makes a sound.');

}

}

class Cat extends Animal {

void makeSound() {

print('Meow!');

}

}

In this example, the Cat class is a subclass of the Animal class, and it overrides the makeSound method with its own implementation.

**B14. Are static members inherited to sub classes?**

Ans. Static members in Dart are not inherited by subclasses. This means that a subclass cannot access the static members of its superclass directly. If a subclass needs to use a static member of its superclass, it must refer to the member using the name of the superclass.

For example, suppose we have a superclass Vehicle with a static member maxSpeed, and a subclass Car that inherits from Vehicle: class Vehicle {

static int maxSpeed = 100;

}

class Car extends Vehicle {

void drive() {

// Can't access Vehicle.maxSpeed directly

print('Driving at max speed ${Vehicle.maxSpeed} mph');

}

}

In this example, the Car class cannot access the maxSpeed static member of Vehicle directly. Instead, it refers to the member using the name of the superclass, Vehicle.maxSpeed.

Static members are associated with the class in which they are declared, not with instances of the class. This means that the value of a static member is the same for all instances of the class, and can be accessed without creating an instance of the class. However, static members are not inherited by subclasses, and must be referred to using the name of the class in which they are declared.

**B15. What happens if the parent and the child class have a field with same identifier?**

Ans. If both the parent and the child class have a field with the same identifier (name), then the child class field will hide (override) the parent class field. This is known as "field hiding" or "variable hiding".

When the child class refers to the field with that identifier, it will use its own field instead of the parent class field. This is because the child class overrides the field in the parent class with its own implementation.

**B16. Are constructors and initializers also inherited to sub classes?**

Ans. Constructors and initializers are not inherited to subclasses, but they are invoked by subclasses as part of the process of constructing an object.

When a subclass is created, it automatically calls the constructor of its superclass to initialize the inherited fields from the superclass. If the superclass has no default constructor, the subclass must call one of the superclass's constructors explicitly using the super keyword.

**B17. How do you restrict a member of a class from inheriting by its sub classes?**

Ans. In Java, you can use the final keyword to restrict a member of a class from being inherited by its subclasses. When a field, method, or class is marked as final, it cannot be overridden or inherited by any subclass.

**B18. How do you implement multiple inheritance in Dart?**

Ans. Dart does not support multiple inheritance, meaning a class can only inherit from a single superclass. However, Dart provides an alternative to multiple inheritance using interface.

Interface allow you to reuse a class's code in multiple class hierarchies. To use a mixins in Dart, you define a class that contains the code you want to reuse, and then use the with keyword to add that class's behaviour to another class. The syntax for using mixins in Dart is as follows:

class MyClass extends MySuperclass with interface1, interface2 {

// class implementation

}

**B19. Can a class extend by itself in Dart?**

Ans. No, a class cannot extend itself in Dart, as it would create a cyclic inheritance relationship that is not allowed.

Inheritance in Dart is a mechanism by which a class can inherit properties and methods from a superclass. The syntax for inheritance in Dart uses the extends keyword, and the name of the superclass that the new class is inheriting from.

**B20. How do you override a private method in Dart?**

Ans. In Dart, private methods are denoted by using an underscore (\_) at the

beginning of their names. By design, private methods are not intended to be overridden in child classes or accessed from outside the class where they are defined.

**B21. When to overload a method in Dart and when to override it?**

Ans. In Dart, method overloading and method overriding are two different concepts and are used in different scenarios.

Method Overloading:

Method overloading is when multiple methods in the same class have the same name but different parameters. In Dart, method overloading is not supported, so you cannot have multiple methods with the same name and different parameters in the same class. If you need to have similar functionality with different parameter types, you can create separate methods with different names.

Method Overriding:

Method overriding is when a subclass provides a specific implementation of a method that is already defined in its superclass. To override a method, you need to create a method with the same name and signature (parameter types and return type) in the subclass. This allows you to customize the behaviour of the method for the subclass without changing the implementation in the superclass. Method overriding is commonly used in object-oriented programming to implement polymorphism and inheritance.

**B22. What is the order of extends and implements keyword on Dart class declaration?**

Ans. In Dart, the order of the extends and implements keywords in a class declaration is as follows:

class MyClass extends MySuperClass implements MyInterface {

// class body

}

First, the extends keyword is used to specify the superclass that the class inherits from. This is followed by the implements keyword, which is used to specify the interface(s) that the class implements.

It is important to note that a class can only extend one superclass, but it can implement multiple interfaces by separating them with commas. Also, the order in which the implemented interfaces are listed does not matter.

**B23. How do you prevent overriding a Dart method without using the final modifier?**

Ans. In Dart, if you want to prevent a method from being overridden in a subclass, you can use the @sealed annotation provided by the meta package. The @sealed annotation marks a class, method, or field as sealed, which means that it cannot be extended or overridden.

To use the @sealed annotation, you need to add the meta package as a dependency in your pubspec.yaml file:

dependencies:

meta: ^1.7.0

Then, import the meta package in your Dart file and add the @sealed annotation before the method you want to seal:

import 'package:meta/meta.dart';

class MyClass {

@sealed

void myMethod() {

// method body

}

}

With this code, if a subclass tries to override myMethod(), a warning will be issued at compile time.

It's worth noting that the @sealed annotation is not a built-in feature of Dart and requires the meta package. Also, if you want to seal a class, you can add the @sealed annotation to the class declaration instead of a method.

**B24. What are the rules of method overriding in Dart?**

Ans. In Dart, method overriding allows a subclass to provide its own implementation of a method that is already defined in its superclass. The following are the rules of method overriding in Dart:

The method in the subclass must have the same name as the method in the superclass.

The method in the subclass must have the same or a more accessible (i.e., less restrictive) access modifier than the method in the superclass. For example, if the method in the superclass is declared as public, the method in the subclass can be declared as public or protected, but not private.

The method in the subclass must have the same return type as the method in the superclass, or a covariant return type. A covariant return type allows the subclass method to return a subtype of the return type of the superclass method.

**B25. Difference between method overriding and overloading in Dart.**

Ans. In Dart, method overriding and method overloading are two distinct concepts. The key differences between them are:

Method overriding allows a subclass to provide its own implementation of a method that is already defined in its superclass. Method overloading, on the other hand, allows a class to define multiple methods with the same name but different parameters.

Method overriding is used to implement runtime polymorphism, which allows a program to decide which implementation of a method to call at runtime, based on the type of the object on which the method is called. Method overloading, on the other hand, is used to provide different ways of calling a method with different combinations of arguments.

Method overriding requires that the method in the subclass has the same signature as the method in the superclass, with the same number and types of parameters. Method overloading, on the other hand, allows methods to have different signatures, with different numbers or types of parameters.

**B26. What happens when a class implements two interfaces and both declare field (variable) with same name?**

Ans. In Dart, when a class implements two interfaces that both declare a field (variable) with the same name, the class must provide an implementation of the field in order to resolve the conflict.

When a class implements an interface, it must provide an implementation of all the fields and methods declared in the interface. If two or more interfaces declare a field with the same name, the class must provide an implementation of that field that satisfies all the interface requirements.

The class can provide an implementation of the field by explicitly declaring the field in the class and initializing it with a value.

For example:

class MyClass implements Interface1, Interface2 {

int myField; // Explicitly declare the field in the class

// Provide an implementation for the fields declared in the interfaces

int get field1 => myField;

set field1(int value) => myField = value;

int get field2 => myField;

set field2(int value) => myField = value;

}

In the above example, the class MyClass implements two interfaces, Interface1 and Interface2, both of which declare a field named myField. The class provides an implementation for both fields by explicitly declaring the field in the class and initializing it with a value, and by providing getters and setters that access the field.

If the class does not provide an implementation of the field, it will result in a compilation error, as the compiler will not be able to determine which interface's field to use. Therefore, it is important to provide an implementation of the field in the class to avoid conflicts and to satisfy the requirements of both interfaces.

**B27. Can a subclass instance method override a superclass static method?**

Ans. No, a subclass instance method cannot override a superclass static method in Dart.

In Dart, instance methods and static methods are two different types of methods, and they cannot be overridden by each other. Instance methods are associated with an instance of a class and can be called on that instance, while static methods are associated with the class itself and can be called without creating an instance of the class.

When a subclass extends a superclass, it can override an instance method of the superclass with its own implementation, but it cannot override a static method of the superclass. Instead, the subclass can declare its own static method with the same name as the static method in the superclass.

**B28. Can a subclass static method hide superclass instance method?**

Ans. Yes, a subclass static method can hide a superclass instance method in Dart.

In Dart, static methods and instance methods are two different types of methods, and they can have the same name without causing a compile-time error. However, when a subclass defines a static method with the same name as an instance method in its superclass, the static method "hides" the instance method in the superclass, and the instance method becomes inaccessible through the subclass.

For example:

class SuperClass {

void myMethod() {

print('This is an instance method in SuperClass.');

}

}

class SubClass extends SuperClass {

static void myMethod() {

print('This is a static method in SubClass.');

}

}

void main() {

SubClass.myMethod(); // Output: This is a static method in SubClass.

SubClass mySubClass = SubClass();

// mySubClass.myMethod(); // This would result in a compile-time error.

}

**B29. Can a superclass access subclass member?**

Ans. No, a superclass cannot directly access subclass members in Dart.

In object-oriented programming, inheritance is a way for a subclass to inherit the properties and behaviours of its superclass. However, a superclass is not aware of any members that are specific to its subclasses, and it cannot directly access them.

Consider the following example:

class SuperClass {

void myMethod() {

print('This is a method in SuperClass.');

}

}

class SubClass extends SuperClass {

String myProperty = 'This is a property in SubClass.';

}

void main() {

SuperClass mySuperClass = SuperClass();

mySuperClass.myMethod();

SubClass mySubClass = SubClass();

mySubClass.myMethod();

// print(mySubClass.myProperty); // This would result in a compile-time error.

}

To access subclass members from a superclass, you can define a method in the superclass that takes a parameter of the subclass type and use the parameter to access the subclass members.

**B30. Difference between object oriented and object-based language.**

Ans. Object-oriented programming (OOP) and object-based programming (OBP) are two different approaches to programming.

Object-oriented programming is a programming paradigm that is based on the concept of "objects." In OOP, an object is an instance of a class, which encapsulates data and behaviour. OOP languages typically support inheritance, polymorphism, and encapsulation, allowing programmers to create reusable, modular code. Examples of object-oriented languages include Java, C++, and Python.

Object-based programming, on the other hand, is a programming approach that uses objects, but does not support all the features of OOP. Object-based languages allow the creation of objects and their manipulation, but do not support features such as inheritance or polymorphism. Examples of object-based languages include JavaScript and VBScript.

In summary, the main difference between object-oriented programming and object-based programming is that OOP is a more complete and sophisticated paradigm that supports more advanced features, such as inheritance and polymorphism, while OBP is a simpler form of programming that only supports basic object creation and manipulation.

**B31. Explain Diamond problem.**

Ans. The "Diamond Problem" is a name given to a specific ambiguity that can arise when a class hierarchy includes multiple inheritance and a method is invoked that is defined in two or more of the parent classes.

The problem is called the "Diamond Problem" because the class hierarchy, when diagrammed, forms a diamond shape. It occurs when a class inherits from two or more classes, which in turn inherit from a common base class. If the derived class calls a method that is defined in both of the base classes, then there is a conflict as to which implementation of the method should be used.

**B32. Why Dart does not support operator overloading?**

Ans.

**B33. What is Encapsulation in Dart?**

Ans. 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.

**B34. Which of the Dart OOPS feature promotes access protection or data hiding?**

Ans. In Dart, the access protection or data hiding is promoted by the concept of Encapsulation, which is one of the fundamental concepts of Object-Oriented Programming.