**21.What is inheritance?**

inheritance in Dart, like in many other object-oriented programming languages, is a mechanism that allows one class to inherit properties and behavior from another class. It's a fundamental concept in object-oriented programming and is used to create a hierarchy of classes where a subclass (derived class or child class) can inherit attributes and methods from a superclass (base class or parent class).

**22.Which inheritance is not supported by Dart? Why? B3. What is advantage of inheritance?**

Dart supports single inheritance, which means that a class in Dart can inherit from only one superclass at a time. This is also known as "single inheritance" or "single class inheritance." Dart does not support multiple inheritance, where a class can inherit from multiple super classes.

The reason Dart doesn't support multiple inheritance is to simplify the language and reduce complexity. Multiple inheritance can lead to a variety of issues and complications, including the diamond problem. The diamond problem occurs when a class inherits from two classes that have a common base class, creating ambiguity in the inheritance hierarchy.

Advantages of Inheritance in Object-Oriented Programming:

1. Code Reusability: Inheritance allows you to reuse code from an existing class (the superclass) in a new class (the subclass). This promotes code reuse and reduces duplication.

2. Extensibility: You can extend the functionality of a class by creating a subclass and adding new methods or properties while inheriting existing ones. This allows for easy extension and modification of code.

3. Abstraction: Inheritance supports the creation of abstract base classes, which define a common interface or set of behaviors for a group of related classes. Subclasses can provide specific implementations.

4. Polymorphism: Inheritance is closely tied to polymorphism, where objects of different classes can be treated as objects of their common superclass. This enables more generic and flexible code.

5.Organization and Hierarchy: Inheritance helps create a hierarchical structure for classes, making it easier to understand and organize code. It models real-world relationships and hierarchies effectively.

6. Maintenance: Changes or updates made to the superclass automatically affect all its subclasses. This makes maintenance and updates more efficient.

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

| Aspect | Inheritance | Encapsulation |
| --- | --- | --- |
| Purpose | Allows subclass to inherit properties and behavior from a superclass. | Focuses on restricting access to object components and hiding internal details. |
| Code Reusability | Promotes code reusability by reducing duplication through subclassing. | Doesn't directly promote code reusability; ensures data protection and object integrity. |
| Access Control | Doesn't directly control access to class members; inherited members retain their original visibility. | Enforces access control through access modifiers, specifying member visibility. |
| Relationship | Used to model "is-a" relationships in class hierarchies. | Used to protect the internal state and behavior of objects. |

| Aspect | Inheritance | Abstraction |
| --- | --- | --- |
| Purpose | Establishes a hierarchical relationship between classes, allowing a subclass to inherit properties and behaviors from a superclass. | Defines a common interface or set of behaviors that multiple classes can share without specifying the actual implementation. |
| Code Reusability | Promotes code reusability by allowing a subclass to inherit attributes and methods from a superclass, reducing code duplication. | Promotes code reusability by providing a common interface or contract that multiple classes can implement, ensuring a consistent behavior. |
| Implementation | Inheritance involves the concrete implementation of classes and the extension of functionality through subclassing. | Abstraction deals with defining abstract classes, interfaces, or protocols that declare a contract without providing concrete implementation details. |
| Flexibility | Can lead to a rigid class hierarchy, which may limit flexibility in certain cases. | Allows for more flexible designs, as multiple classes can implement the same abstract interface without being constrained by a strict hierarchy. |
| Relationship | Establishes an "is-a" relationship, indicating that a subclass is a specific type of the superclass. | Establishes a "has-a" or "uses" relationship, where multiple classes can use a common interface or abstract class without being direct subclasses. |

| Aspect | Inheritance | Polymorphism |
| --- | --- | --- |
| Purpose | Establishes a hierarchical relationship between classes, allowing a subclass to inherit properties and behaviors from a superclass. | Enables objects of different classes to be treated as objects of their common superclass, promoting flexibility and extensibility. |
| Code Reusability | Promotes code reusability by allowing a subclass to inherit attributes and methods from a superclass, reducing code duplication. | Promotes flexibility and extensibility by allowing for more generic and flexible code where different objects can be used interchangeably. |
| Implementation | Inheritance involves the concrete implementation of classes and the extension of functionality through subclassing. | Polymorphism involves the dynamic behavior of objects at runtime and allows for different classes to respond differently to the same method call (method overriding). |
| Relationship | Establishes an "is-a" relationship, indicating that a subclass is a specific type of the superclass. | Establishes a relationship where objects of different classes share a common interface (method signature) or base class. |
| Type of Behavior | Inheritance focuses on the behavior and attributes inherited from the superclass and allows for customization and extension. | Polymorphism focuses on the ability of objects to exhibit different behaviors based on their specific class or type. |
| Method Overriding | Inheritance allows subclasses to override methods from the superclass to provide their own implementations. | Polymorphism relies on method overriding to enable different objects to respond differently to the same method call based on their specific class's implementation. |

25. Can we override static method in Dart?

[In Dart, it is not possible to override a static method. but i can't understand output of this code](https://stackoverflow.com/questions/73034328/in-dart-it-is-not-possible-to-override-a-static-method-but-i-cant-understand)

**Ex:**

class Student {

static void getDetails() {

print('Get details method of Student class');

}

}

class DartStudent extends Student {

static void getDetails() {

print('Get details method of DartStudent class');

}

}

void main() {

DartStudent.getDetails();

}

o/p: Output : Get details method of DartStudent class

Expected : Error. static method cannot be overriden.. or something wrong..

what's wrong with me?

getDetails() in DartStudent class is not overriding parent class's method?

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

**Implementing Multiple Interfaces:** Yes, a class can implement more than one interface. Dart supports multiple interface implementation, which means that a class can declare that it adheres to the contract (method signatures) of multiple interfaces. To implement multiple interfaces, you list them after the implements keyword, separated by commas.

class MyClass implements Interface1, Interface2, Interface3 {

// Class implementation

}

**Extending Multiple Classes:** No, a class cannot extend more than one class in Dart. Dart supports single inheritance, which means that a class can have only one superclass. This is unlike some other languages that allow multiple inheritance, where a class can inherit from multiple superclasses.

class Subclass extends Superclass {

// Class implementation

}

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

In Dart, an interface can extend more than one interface by using the extends keyword. This allows you to create new interfaces that inherit the method signatures and requirements from multiple parent interfaces. Here's the syntax for defining an interface that extends multiple interfaces:

abstract class Interface1 {

void method1();

}

abstract class Interface2 {

void method2();

}

abstract class Interface3 {

void method3();

}

abstract class CombinedInterface implements Interface1, Interface2, Interface3 {

// This interface inherits the method signatures from Interface1, Interface2, and Interface3

}

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

In Dart, if a class implements two interfaces, and both interfaces have a method with the same name and signature, the class is still required to provide only one implementation of that method. The Dart language does not allow method overloading (creating multiple methods with the same name but different parameters), and when a class implements multiple interfaces with a method of the same name and signature, Dart treats those methods as a single method.

Ex:

abstract class Interface1 {

void commonMethod();

}

abstract class Interface2 {

void commonMethod();

}

class MyClass implements Interface1, Interface2 {

@override

void commonMethod() {

print('Implemented in MyClass');

}

}

**30. Can we pass an object of a subclass to a method expecting an object of the super class? B14. Are static members inherited to sub classes?**

Yes, in object-oriented programming, you can pass an object of a subclass to a method that expects an object of the superclass. This concept is known as "polymorphism" and is a fundamental feature of many object-oriented programming languages, including Dart. When you pass a subclass object to a method that expects a superclass object, it's referred to as "upcasting."

Ex:

class Animal {

void speak() {

print('Animal speaks');

}

}

class Dog extends Animal {

@override

void speak() {

print('Dog barks');

}

}

void myMethod(Animal animal) {

animal.speak();

}

void main() {

Animal myAnimal = Dog(); // Upcasting a Dog object to an Animal reference

myMethod(myAnimal); // Calls the Dog's implementation of speak

}

**31. What happens if the parent and the child class have a field with same identifier? B16. Are constructors and initializers also inherited to sub classes?**

When a parent class and a child class have a field with the same identifier, the child class's field will shadow or hide the parent class's field. This means that the child class's field will be used in place of the parent class's field when referencing the field from the child class. This behavior is sometimes referred to as "variable shadowing."

Ex:

class Parent {

String sharedField = 'Parent Field';

}

class Child extends Parent {

String sharedField = 'Child Field'; // Shadows the Parent's sharedField

}

void main() {

Parent parent = Parent();

Child child = Child();

print(parent.sharedField); // Output: Parent Field

print(child.sharedField); // Output: Child Field

}

**Are constructors and initializers also inherited by subclasses?**

Constructors are not inherited by subclasses in Dart, but subclasses can call constructors of the superclass as part of their own constructors using the **super** keyword. When a subclass is created, it implicitly calls the constructor of its superclass, and you can choose to call a specific constructor of the superclass by using **super**.

Ex:

class Parent {

String name;

Parent(this.name);

}

class Child extends Parent {

int age;

Child(String name, this.age) : super(name);

}

void main() {

var child = Child('Alice', 10);

print('Name: ${child.name}, Age: ${child.age}');

}

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

In Dart, you can restrict a member of a class from being inherited by its subclasses by making that member **private**. Dart uses underscore (**\_**) as a prefix to mark a member (variable, method, etc.) as private.

**Private Variables:** To make a variable private, prefix it with an underscore. Private variables can only be accessed within the same class, not by its subclasses.

Ex: class MyClass {

int \_privateVar = 42; // Private variable

}

**Private Methods:** To make a method private, prefix it with an underscore. Private methods can only be accessed within the same class, not by its subclasses.

Ex:

class MyClass {

void \_privateMethod() {

// Private method

}

}

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

Dart does not support multiple inheritance in the traditional sense where a class can inherit from multiple superclasses. Dart uses single inheritance, which means a class can have only one superclass. This design choice simplifies the language and avoids complications such as the "diamond problem" that can arise in languages with multiple inheritance.

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

In Dart, a class cannot directly extend itself. Attempting to define a class that extends itself will result in a compilation error. Dart enforces a rule that disallows circular class inheritance.

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

In Dart, you cannot override a private method in a subclass because private methods are not accessible outside the class where they are declared, including in subclasses. The idea behind private methods (or variables) is to restrict access to the class where they are defined. Private methods are indicated by a prefix of an underscore (**\_**).

**Ex:**

class Parent {

void \_privateMethod() {

print('Private method in Parent');

}

}

class Child extends Parent {

// Attempting to override the private method will result in an error.

// @override

void \_privateMethod() {

print('This will cause a compilation error');

}

}

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

* Use method overloading when you want to provide multiple versions of a method with different parameter sets to enhance usability and flexibility.
* Use method overriding when you want to customize or extend the behavior of a method inherited from a superclass to meet specific requirements in a subclass.

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

In Dart, when declaring a class, you can use both the extends and implements keywords in the same class declaration. The order of these keywords depends on whether the class is extending another class (using extends) and implementing one or more interfaces (using implements).

Ex:

class MyClass extends SuperClass implements Interface1, Interface2 {

// Class members and implementation

}

* The **extends** keyword is used when you are creating a subclass that inherits from a superclass (i.e., creating a class that extends another class).
* The **implements** keyword is used when you want to specify that your class adheres to one or more interfaces.

The superclass is specified using **extends**, and the implemented interfaces are listed after **implements**. You can have both **extends** and **implements** in the same class declaration if your class is extending a superclass and implementing one or more interfaces.

Ex:

class Animal {

void speak() {

print('Animal speaks');

}

}

class Flyable {

void fly() {

print('Can fly');

}

}

class Bird extends Animal implements Flyable {

// Bird is a subclass of Animal and implements the Flyable interface

@override

void speak() {

print('Bird chirps');

}

}

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

In Dart, you can prevent a method from being overridden by subclasses without using the final modifier by using a combination of private methods and an @protected annotation. This approach restricts direct access to the method by subclasses while still allowing the class itself to use it.

Ex:

class MyClass {

@protected

void \_nonOverridableMethod() {

// This method is non-overridable by subclasses

}

void publicMethod() {

// You can use \_nonOverridableMethod within the class

\_nonOverridableMethod();

}

}

class MySubclass extends MyClass {

@override

void publicMethod() {

// You can override publicMethod, but \_nonOverridableMethod remains inaccessible

}

}

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

1. **Method Signature Match:**
   * The overriding method in the subclass must have the same name, return type, and parameter list (including types and order) as the method in the superclass. This is known as a method signature match.
2. **@override Annotation:**
   * It is a good practice to annotate the overriding method with @override. While not strictly required, this annotation helps make your intention clear and can help catch errors at compile time.
3. **Visibility:**
   * The overriding method in the subclass cannot have reduced visibility compared to the method it is overriding. In other words, you cannot override a public method with a private method in the subclass. However, you can override a protected method with a public method in the subclass.
4. **Covariant Parameters:**
   * In Dart 2.13 and later, you can use the covariant keyword before a parameter type to indicate that the parameter type in the subclass is allowed to be a subtype of the parameter type in the superclass. This is useful for more flexible method signatures.
5. **Superclass Invocation:**
   * When you override a method in a subclass, you can use the super keyword to call the same method in the superclass from within the overriding method. This allows you to extend the behavior of the superclass method.
6. **Call to Superclass Constructor:**
   * In the constructor of a subclass, you must call a constructor of the superclass using super(...). This ensures that the superclass is properly initialized.

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

| **Aspect** | **Method Overriding** | **Method Overloading** |
| --- | --- | --- |
| **Purpose** | Customizing or extending behavior inherited from a superclass in a subclass. | Providing multiple ways to call a method with different parameter sets. |
| **Method Signature** | The overriding method must have the same name, return type, and parameter list as the method in the superclass (same method signature). | Overloaded methods have different parameter lists, including different types, numbers, or order of parameters (distinct method signatures). |
| **Use Case** | Customizing inherited behavior to meet specific requirements in a subclass. | Enhancing usability and flexibility by allowing a method to accept different types and numbers of arguments. |
| **Example** | **dart class Dog extends Animal { void makeSound() { super.makeSound(); print('Dog barks'); } }** | **dart class Calculator { int add(int a, int b) { return a + b; } double add(double a, double b) { return a + b; } }** |

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

When a class implements two interfaces, and both interfaces declare a field (variable) with the same name, it's generally not an issue because fields in interfaces are implicitly **static**, and they don't conflict with each other when implemented in a class. In Dart,

interface fields are automatically **static**, which means they belong to the class, not to instances of the class.

Ex:

class Interface1 {

static int commonField = 42;

}

class Interface2 {

static int commonField = 99;

}

class MyClass implements Interface1, Interface2 {

// Both Interface1 and Interface2 have a static field named commonField

// These fields don't conflict with each other

}

void main() {

print(MyClass.commonField); // Accesses the commonField from Interface1

}

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

No, in Dart, a subclass instance method cannot override a superclass static method. Method overriding and method hiding are two distinct concepts in object-oriented programming, and they apply to different types of methods.

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

No, in Dart, a subclass static method cannot hide a superclass instance method. Method hiding and method overriding are two distinct concepts, and they apply to different types of methods.

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

No, a superclass cannot directly access members (fields, methods) of its subclass in Dart. The inheritance relationship works in one direction, where a subclass inherits members from its superclass, but not the other way around. Members in a superclass are encapsulated and do not have direct access to members in subclasses.

**45.Difference between object oriented and object based language.**

| **Aspect** | **Object-Oriented Programming (OOP)** | **Object-Based Programming** |
| --- | --- | --- |
| **Classes and Objects** | Central concept with classes defining blueprints for objects. | Objects may exist without class definitions. |
| **Inheritance** | Supports class-based inheritance, allowing one class to inherit from another. | Limited or no support for inheritance. |
| **Polymorphism** | Emphasizes polymorphism, including method overriding and overloading. | May have limited or no support for polymorphism. |
| **Encapsulation** | Strong emphasis on encapsulation with access modifiers (public, private, protected). | Less emphasis on encapsulation, data and methods are part of objects. |
| **Examples of Languages** | Java, C++, Python, C#, Dart, etc. | JavaScript (considered object-based), Self, etc. |

**46. Create a program using List**

void main() {

// Create a List of integers

List<int> numbers = [1, 2, 3, 4, 5];

// Print the original list

print("Original List: $numbers");

// Accessing elements by index

int secondElement = numbers[1];

print("Second element: $secondElement");

// Modifying elements

numbers[3] = 10;

print("List after modifying: $numbers");

// Adding elements to the end of the list

numbers.add(6);

print("List after adding 6: $numbers");

// Removing elements from the list

numbers.remove(3);

print("List after removing 3: $numbers");

// Finding the length of the list

int length = numbers.length;

print("Length of the list: $length");

// Iterating through the list

print("Iterating through the list:");

for (int number in numbers) {

print(number);

}

}

**47. Create a program using Set**

void main() {

// Create a Set of strings to store unique names

Set<String> names = Set();

// Adding elements to the set

names.add("Alice");

names.add("Bob");

names.add("Charlie");

names.add("Alice"); // Duplicate value, won't be added

// Print the original set

print("Original Set: $names");

// Checking if an element is in the set

if (names.contains("Alice")) {

print("Alice is in the set.");

} else {

print("Alice is not in the set.");

}

// Removing an element from the set

names.remove("Bob");

// Print the set after removal

print("Set after removing Bob: $names");

// Finding the size of the set

int size = names.length;

print("Size of the set: $size");

// Iterating through the set

print("Iterating through the set:");

for (String name in names) {

print(name);

}

}

**48. Create a program using Map**

void main() {

// Create a Map to store key-value pairs of names and ages

Map<String, int> ages = {

"Alice": 30,

"Bob": 25,

"Charlie": 35,

};

// Print the original map

print("Original Map: $ages");

// Accessing values by key

int aliceAge = ages["Alice"];

print("Alice's age: $aliceAge");

// Modifying values

ages["Bob"] = 26;

print("Map after modifying Bob's age: $ages");

// Adding new key-value pairs

ages["David"] = 28;

print("Map after adding David's age: $ages");

// Removing a key-value pair

ages.remove("Charlie");

print("Map after removing Charlie: $ages");

// Checking if a key exists in the map

if (ages.containsKey("Bob")) {

print("Bob's age is in the map.");

} else {

print("Bob's age is not in the map.");

}

// Finding the number of key-value pairs in the map

int size = ages.length;

print("Size of the map: $size");

// Iterating through the map

print("Iterating through the map:");

ages.forEach((name, age) {

print("$name is $age years old.");

});

}