**Module-2**

**Flutter-Dart Programing**

**21. What is inheritance?**

Inheritance in programming refers to the concept of creating a new class that is a modified version of an existing class. The new class, known as the derived or child class, inheritance attributes and behaviors from the existing class, called the base or parent class. This allows for code reuse and the creating of a hierarchy of classes.

Imagine the parent class as the blueprint, and the child class as a specific instance that inherits characteristics from that blueprint.

It’s like passing on traits from one generation to the next, but in the world of programming.

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

Dart does not support multiple inheritance. Multiple inheritance occurs when a class inheritance from more than one class. Dart uses single inheritance, meaning a class can only inherit from one superclass.

This decision simplifies the language and reduces complexity. Multiple inheritance can lead to various issues such as the diamond problem, Whare conflicts arise if a class inherits from two classes that have a common ancestor. Dart avoids these complications by sticking to single inheritance and promoting the use of mixins to achieve code reuse in a more controlled way.

**22(B3). What is advantage of inheritance?**

Inheritance in programming offers several advantages:

**1. Code reusability:** Inheritance allows a class to reuse the code of an existing class. The attributes and behaviours of the parent class can be inherited by the child class, reducing redundancy and promotion a more modular and maintainable codebase.

**2. Polymorphism:** Inheritance enable polymorphism, where object of the derived class can be treated as objects of the base class. This flexibility allows for more generic and adaptable code, as different classes can share a common interface though inheritance.

**3. Extensibility:** New classes can be created by extending existing classes, adding or modifying functionalities. This makes it easier to extend the capabilities of a system without altering the existing code, promotion a more modular and scalable design.

**4.Encapsulation:** Inheritance support encapsulation by allowing the bundling of data and methods within a class. The derived class can access the public and protect members of the base class, contribution to the organization and encapsulation of code.

**5.Hierarchy and Organization:** Inheritance provides and natural way to model and represent hierarchical relationships between classes. This makes the code more intuitive and reflective of real-world relationships, enhancing the overall organization and understanding of the system.

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

|  |  |  |
| --- | --- | --- |
| Feature | Inheritance | Encapsulation |
| Definition | Mechanism allowing a class to inherit properties and behaviours from another class (base/parent class). | Bundling of data (attributes) and methods within a class, hiding the internal details from external access. |
| Syntax | Often involves keywords like extends or inherits in programming languages | Utilizes access modifiers (public, private, protected) to control visibility and encapsulate data. |
| Purpose | Promotes code reuse by establishing a relationship between classes, allowing a derived class to inherit attributes and methods from a base class. | Organizes and structures code by controlling access to the internal state of an object, protecting the integrity of data. |
| Relationship | Establishes an "is-a" relationship between classes, indicating that a derived class is a specialized form of the base class. | Focuses on a "has-a" relationship, emphasizing the containment of data and methods within a class. |
| Access Control | Controls visibility through access modifiers like public, protected and private. Derived classes may access public and protected members. | Controls access to data and methods by specifying visibility (e.g., private attributes), limiting direct external access. |
| Code Reuse | Facilitates code reuse by inheriting common functionality from a base class, reducing redundancy. | Promotes modular code by encapsulating data and methods within a class, making it easier to reuse and maintain. |
| Example | Base class "Animal" with properties like "name" and "age," and a derived class "Dog" inheriting these properties. | Base class "Animal" with properties like "name" and "age," and a derived class "Dog" inheriting these properties. |

**23(B5). Difference between inheritance and abstraction.**

|  |  |  |
| --- | --- | --- |
| Feature | Inheritance | Abstraction |
| Definition | Mechanism allowing a class to inherit properties and behaviours from another class (base/parent class). | Process of hiding the implementation details and showing only the essential features of an object. |
| Syntax | Often involves keywords like extends or inherits in programming languages | Involves abstract classes or interfaces with abstract methods |
| Purpose | Promotes code reuse by establishing a relationship between classes, allowing a derived class to inherit attributes and methods from a base class. | Simplifies complex systems by providing a clear and abstract view, focusing on what an object does rather than how it achieves it. |
| Relationship | Establishes an "is-a" relationship between classes, indicating that a derived class is a specialized form of the base class. | Establishes a "has-a" relationship, indicating that a class has certain essential features without revealing their implementation details. |
| Code Reuse | Facilitates code reuse by inheriting common functionality from a base class, reducing redundancy. | Encourages code reuse by providing a common interface or abstraction that multiple classes can implement |
| Example | Base class "Animal" with properties like "name" and "age," and a derived class "Dog" inheriting these properties. | Class "Shape" with abstract methods like calculate allowing specific shapes like Circle and Rectangle to implement their own versions. |

**24. Difference between Inheritance and Polymorphism.**

|  |  |  |
| --- | --- | --- |
| Feature | Inheritance | Polymorphism |
| Definition | Mechanism allowing a class to inherit properties and behaviours from another class (base/parent class). | The ability of objects of different types to be treated as objects of a common type, often achieved through method overriding. |
| Syntax | Often involves keywords like extends or inherits in programming languages | Implemented through method overriding, where a derived class provides a specific implementation for a method declared in the base class. |
| Purpose | Promotes code reuse by establishing a relationship between classes, allowing a derived class to inherit attributes and methods from a base class. | Enhances flexibility and extensibility by allowing objects to take on multiple forms and be manipulated through a common interface. |
| Relationship | Establishes an "is-a" relationship between classes, indicating that a derived class is a specialized form of the base class. | Allows objects of different types to be treated uniformly under a common interface, regardless of their specific implementations. |
| Types | Includes single inheritance and multiple inheritance (where supported), defining a hierarchy of classes. | Can be achieved through compile-time (method overloading, static polymorphism) and runtime (method overriding, dynamic polymorphism) mechanisms. |
| Compile-time | Mostly determined at compile-time, as the type of the object is known during compilation. | Resolves at runtime, allowing the same method name to behave differently based on the actual type of the object being referenced. |
| Example | Base class "Animal" with properties like "name" and "age," and a derived class "Dog" inheriting these properties. | Polymorphism allows a method like display to be defined in a base class and overridden in derived classes |

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

In dart, static methods cannot be overridden. They are associated with the class itself rather than instances of the class.

When a subclass declares a static method with the same name as a static method in its superclass, it is simply hiding the superclass method rather than overriding it.

If you want to achieve polymorphic behavior with methods in Dart, you should use instance methods and inheritance.

Static methods are more about the class itself and don’t participate in the inheritance hierarchy in the same way instance methods do.

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

No, Dart does not support method overloading in the traditional sense, whether the methods are static or instance methods.

Method overloading typically refers to defining multiple methods in the same class with same name but different parameter lists.

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

Yes, in Dart, a class can implement multiple interfaces. Dart supports multiple inheritance through the use of interfaces.

When a class implements multiple interfaces, it means that the class promises to provide implementations for all the methods declared by those interfaces.

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

No, in Dart, a class cannot extend more than one class. Dart supports single inheritance, meaning a class can only have one direct superclass.

However, dart does support mixins, which allow a class to reuse the code of multiple classes.

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

Yes, in Dart, an interface can extend more than one interface. Dart supports multiple inheritance for interfaces, allowing you to create complex hierarchies of interfaces.

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

If a class implements two interfaces, and both interfaces have a method with the same name and signature, dart will not raise any conflict or error.

The class is simply required to provide an implementation for that method, and it will be treated as a single method in the implementing class.

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

Yes, in most object-oriented programming languages, including dart, you can pass an object of a method expecting an object of the superclass.

This is a fundamental concept known as polymorphism. If a subclass inherits from a superclass, you can treat an instance of the subclass as an instance of the superclass.

**30(B14). Are static members inherited to sub classes?**

No, static members are not inherited by subclasses in dart. Static members belong to the class itself and are not part of the instance-level inheritance hierarchy.

When a subclass is created, it does not inherit the static members from its superclass.

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

In object-oriented programming, if both the parent and child classes have a field with the same identifier, it can lead to a situation known as ‘field hiding’ or ‘field shadowing’.

In such cases, the field in the child class will ‘hide’ or ‘shadow’ the field in the parent class.

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

In dart, constructor are not inherited by default, but they are implicitly called in a specific order during the creating of an instance of a subclass.

When you create an instance of a subclass. When you create an instance of a subclass, the constructor of the superclass is called before the constructor of the subclass.

If the superclass has a default constructor (a constructor with no parameter), and you don’t explicitly define a constructor in the subclass, the default constructor od the superclass will be called.

However, if the superclass has a parameterized constructor, and you want to use that constructor in the subclass, you need to call it explicitly using the ‘super’ keyword in the constructor of the subclass.

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

In dart, by default, all members (field and methods) of a class are accessible to its subclasses. However, if you want to restrict a member from being inheritance by its subclasses, you can use the ‘final’ keyword or make it private by starting its name with an underscore (‘\_’).

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

Dart does not support multiple inheritance in the traditional sense, where a class directly inherits from more than one class. However, Dart provides a way to achieve similar functionality using mixins.

Mixins allow you to reuse a class’s code in multiple class hierarchies. A mixin is a way to reuse a class’s code in multiple class hierarchies. You can use the ‘with’ keyword to include the functionality of a mixins in a class.

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

In Dart, a class cannot directly extend itself. Attempting to create a direct cyclic inheritance, where a class extends itself, will result in a compile-time error.

In Dart class inheritance is intended to form a hierarchical structure, and a class cannot be its own superclass. If you need to share behaviour or functionality between instances of the same class, you typically use methods, fields, or static members within the class rather than attempting to create a circular inheritance relationship.

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

In Dart, private methods (those with names starting with an underscore ‘\_’) are not directly accessible or visible outside of the library where they are defined. Therefore, overriding a private method from a superclass is not possible in the usual way.

Remember that private members, including private methods, are not visible to subclasses outside of the defining library. If you need to provider a similar behaviour that can be overridden, consider using a protected method or making the method public, depending on your design considerations.

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

In Dart, overloading and overriding are two different concepts related to methods in object-oriented programming.

**Method Overloading:**

-Method overloading involves defining multiple methods in the same class with the same name but different parameter lists. Dart does not support traditional method overloading based on parameter types or number of parameters. In dart, you can achieve a form of method overloading using optional and named parameters.

**Method Overriding:**

-Method overriding occurs when a subclass provides a specific implementation for a method that is already defined in its superclass. The overriding method in the subclass must have the same signature (name and parameter) as the method in the superclass.

**When to Overload:**

-Overload methods when you want to provide different ways to call a method with varying parameters in the same class.

**When to Override:**

- Override methods when you want to provide a specialized implementation of a method in a subclass, refining or extending the behaviour defined in the superclass.

In summary, method overloading is about providing multiple methods with the same name in the same class, while method overriding is about providing a specific implementation of a method in a subclass that is already defined in its superclass. The choice between overloading and overriding depending depends on the requirements of your application and the behaviour you want to achieve.

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

In Dart, when declaring a class, the ‘extends’ keyword is used to specify the superclass (the class that new class is inheriting from), and the ‘implements’ keyword is used to specify the interfaces that the class is implementing.

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

In Dart, if you want to prevent method from being overridden in subclasses without using the ‘final’ modifier, you can use the ‘@sealed’ annotation. The ‘@sealed’ annotation is not a built-in feature in Dart, but it’s a convention that some developers follow.

**39. What are the rules of method overriding in Dart?**   
 In Dart, method overriding is the process of providing a specific implementation for a method that is already defined in a superclass. To successfully override a method in Dart, you should adhere to the following rules

**1.Signature Match:** The overriding method in the subclass must have the same name, return type, and parameters (including their types) as the method in the superclass.

**2.Use @overide Annotation:** While not strictly required, it's good practice to use the @override annotation before the method in the subclass to explicitly indicate that you are intending to override a method.

**3. Inheritance Relationship:** The superclass and subclass must have an inheritance relationship. The subclass should extend the superclass.

**4.Visibility and Access Levels:** The overriding method in the subclass should not reduce the visibility or change the access level of the method in the superclass. For example, you cannot override a private method in the superclass with a public method in the subclass.

**5.Covarient Parameter types (Optional):**  If the overriding method has covariant parameter types, it should use the covariant keyword to indicate the intent. This is optional but can help improve type safety.

**6.Covarient Return Types (Dart 2.14 and later):** In Dart 2.14 and later, you can use the covariant keyword for return types as well, providing better support for overriding methods with covariant return types.

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

|  |  |  |
| --- | --- | --- |
| Feature | Method Overriding | Method Overloading |
| Definition | Redefining a method in the subclass that is already defined in the superclass. | Declaring multiple methods in the same class with the same name but different parameters. |
| Inheritance Requirement | Overriding occurs in a subclass, inheriting from a superclass. | Overloading can occur within the same class or in subclasses. It is not dependent on inheritance. |
| Signature | The overridden method in the subclass must have the same name, return type, and parameters as the method in the superclass. | Overloaded methods must have the same name but differ in the number or types of parameters. Return type alone is not sufficient for differentiation. |
| Keyword usage | The @overide annotation is used to indicate that a method is intended to override a superclass method. | No specific keyword is used for overloading; it is determined by the method signatures. |
| Execution Context | The method implementation in the subclass is executed when the method is called on an instance of the subclass. | The appropriate overloaded method is selected based on the number and types of arguments during the method call. |
| Example | dart class Superclass { void display() { print('Superclass display'); } } class Subclass extends Superclass { @override void display() { print('Subclass display'); } } | dart class Example { void showInfo(String name) { print('Name: $name'); } void showInfo(String name, int age) { print('Name: $name, Age: $age'); } } |

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

In Dart, when a class implements two interfaces, and both interfaces declare a field (variable) with the same name, the implementing class will need to provide an implementation for that field.

Dart allows a class to implement multiple interfaces, but it does not support multiple inheritance for fields.

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

In Dart, it is not possible for a subclass instance method to override a superclass static method. Method overriding in Dart is typically associated with instance methods and polymorphism, and it does not apply to static methods.

When you declare a static method in a Dart superclass, it cannot be overridden by a subclass instance method. Static methods are associated with the class itself rather than instances of the class. Subclasses can have their own static methods with the same name, but this is not considered method overriding in the polymorphic sense.

**43. Can a subclass static method hide superclass instance method?**   
Yes, in Dart, it is possible for a subclass static method to hide a superclass instance method. This is known as method hiding, and it occurs when a subclass declares a static method with the same name as an instance method in its superclass.

Method hiding is not considered true method overriding, as it involves static methods and does not exhibit polymorphic behaviour.

**44. Can a superclass access subclass member?**   
In Dart, a superclass cannot directly access the members (fields or methods) of its subclass. The relationship between a superclass and a subclass is one of inheritance, and it is designed to allow subclasses to inherit and extend the behaviour of the superclass.

However, the superclass itself does not have direct access to the members that are specific to its subclasses.

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

|  |  |  |
| --- | --- | --- |
| Feature | Object-Oriented Languages | Object-Based Languages |
| Definition | Support both objects and classes. | Primarily support objects but may not support classes or inheritance. |
| Class support | Classes and inheritance are fundamental concepts. | May not support classes or inheritance. |
| inheritance | Supports inheritance for code reuse and polymorphism. | May not support inheritance. |
| Polymorphism | Supports polymorphism through inheritance and interfaces. | May support polymorphism but not necessarily through inheritance. |
| Encapsulation | Emphasizes encapsulation to hide implementation details. | May support encapsulation but with less emphasis on hiding details. |
| abstraction | Provides abstraction through classes and interfaces. | May support abstraction but not necessarily through classes or interfaces. |
| Example languages | ava, C++, C#, Python. | JavaScript, VBScript (in classic ASP), MATLAB. |
| Use cases | Suitable for complex, large-scale applications. | Suitable for smaller projects and scripting. |
| example | java class Dog extends Animal { // ... } | Java script var dog = {// ... }; |

**46. Create a program using List**

void main() {

// Creating a list of integers

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

// Print the original list

print("Original List: $numbers");

// Add an element to the list

numbers.add(6);

// Print the modified list

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

// Accessing elements in the list

print("Element at index 2: ${numbers[2]}");

// Updating an element in the list

numbers[3] = 9;

print("Modified List (after updating element at index 3): $numbers");

// Removing an element from the list

numbers.remove(4);

print("Modified List (after removing element 4): $numbers");

// Iterating through the list

print("Iterating through the list:");

for (int number in numbers) {

print(number);

}

}

**47. Create a program using Set**

void main() {

// Creating a set of integers

Set<int> numbers = {1, 2, 3, 4, 5};

// Print the original set

print("Original Set: $numbers")

// Adding elements to the set

numbers.add(6);

numbers.add(7);

// Print the modified set

print("Modified Set (after adding 6 and 7): $numbers");

// Removing an element from the set

numbers.remove(4);

print("Modified Set (after removing element 4): $numbers");

// Checking if an element is in the set

print("Is 5 in the set? ${numbers.contains(5)}");

// Iterating through the set

print("Iterating through the set:");

for (int number in numbers) {

print(number);

}

}

**48. Create a program using Map**

void main() {

// Creating a map of string keys and integer values

Map<String, int> studentScores = {

'Alice': 85,

'Bob': 92,

'Charlie': 78,

'David': 95,

};

// Print the original map

print("Original Map: $studentScores");

// Adding a new entry to the map

studentScores['Eva'] = 89;

print("Map after adding Eva: $studentScores");

// Accessing values using keys

print("Bob's score: ${studentScores['Bob']}");

// Updating a value in the map

studentScores['Charlie'] = 80;

print("Map after updating Charlie's score: $studentScores");

// Removing an entry from the map

studentScores.remove('Alice');

print("Map after removing Alice: $studentScores");

// Iterating through the map

print("Iterating through the map:");

studentScores.forEach((key, value) {

print("$key: $value");

});

}