Pitch Deck

20xx

</ Dart programming language

Dart

/>

Instructor:Dr.Vahidi
Presenter:Armita Kamari

Fall 1403-1404

1 0 1 1 0 1 1 0 1 1 0 1 1 0 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 1 0 1 1 0 1 1 0 1 1 1 1 1 0 1

General topics

1 0 1 1 0 1 1 0 1 1 0 1 1 0 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 1 0 1 1 0 1 1 0 1 1 1 1 1 0 1

Introduction to Dart

._**Dart** is a modern, object-oriented programming language designed for building fast, scalable, and reliable applications.

._**Created by Google**, first announced in **October 2011**.

._The first stable version, **Dart 1.0**, was released in **November 2013**.

._**Dart 2.0** was released in **August 2018**, with enhancements for modern client development.

1 0 1 1   0 1 1   0 1   1 0 1 1 0 0 1   1 0   1 1 0 1 1   0 1 1   0 1   1 1 0 1 1 0   1 1 0 1 1 1   1 1 0 1

Use Cases of Dart

- **Web Development**: Dart is primarily used for building **client-side** web applications with frameworks like **Flutter**.
- **Mobile Development**: Dart is the backbone of **Flutter**, one of the most popular frameworks for creating **cross-platform mobile applications** for both Android and iOS.
- **Server-Side Development**: Dart can be used on the server side as well, thanks to its fast runtime and ability to handle concurrent tasks efficiently.
- **Desktop Applications**: With **Flutter**, Dart is also capable of creating native desktop applications for Windows, macOS, and Linux.
- **Embedded Systems**: Dart can be used in **IoT** (Internet of Things) projects for building applications that interact with hardware.

1 0 1 1   0 1 1   0 1   1 0 1 1 0 0 1   1 0   1 1 0 1 1   0 1 1   0 1   1 1 0 1 1 0   1 1 0 1 1 1   1 1 0 1

Why Choose Dart

Strong typing: Dart supports both **strong** and **flexible** typing, making it suitable for small scripts or large-scale applications.

High performance: Dart compiles to **native machine code**, ensuring apps run quickly and efficiently.

Fast development: Dart offers **hot-reload**, allowing developers to see code changes instantly without restarting the app.

1 0 1 1   0 1 1   0 1   1 0 1 1 0 0 1   1 0   1 1 0 1 1   0 1 1   0 1   1 1 0 1 1 0   1 1 0 1 1 1   1 1 0 1

</ Types of variables

{01}
var

{02}
dynamic

{03}
String

{04}
Int , double

{05}
bool

{06}
List ,Map

1 0 1 1   0 1 1   0 1   1 0 1 1 0 0 1   1 0   1 1 0 1 1   0 1 1   0 1   1 1 0 1 1 0   1 1 0 1 1 1   1 1 0 1

Variables in dart

1. var

- **Type inference**: When you use `var`, Dart infers the type based on the assigned value. Once a type is inferred, it cannot be changed.
- **Cannot reassign a different type**: You cannot assign a different type to a variable once its type has been inferred.

2. dynamic

- **Type flexibility**: The `dynamic` keyword allows a variable’s type to be reassigned at runtime. This makes `dynamic` suitable for cases where the type is not known in advance.
- **Risks of dynamic**: While `dynamic` provides flexibility, it also introduces risks because type errors might only appear at runtime.

1 0 1 1 0 1 1 0 1 1 0 1 1 0 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 1 0 1 1 0 1 1 0 1 1 1 1 1 0 1

Variables in dart

3. Object

- **Superclass of all types**: `Object` is the base class for all Dart objects, including both built-in types (such as `int` and `String`) and user-defined classes.
- **General use**: It can hold any value, but unlike `dynamic`, the type is known, and Dart can still perform type checks.

4. String

- **Immutable sequence of characters**: `String` is used to represent a sequence of characters. Once a `String` is created, it cannot be changed.
- **Interpolation**: Dart allows string interpolation to embed expressions inside string literals.

1 0 1 1 0 1 1 0 1 1 0 1 1 0 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 1 0 1 1 0 1 1 0 1 1 1 1 1 0 1

Variables in dart

5. int

- **Whole numbers**: The `int` type represents whole numbers in Dart. These numbers do not have decimal points.
- **Range**: On 64-bit systems, the range of `int` values is from -2^63 to 2^63-1

6. double

- **Floating-point numbers**: `double` is used to represent numbers with decimal points. It is based on IEEE 754 standard for double-precision floating-point numbers.

7. bool

- **True or false**: The `bool` type is used for Boolean values (i.e., true or false). In Dart, conditions evaluate to `true` or `false`.

1 0 1 1   0 1 1   0 1   1 0 1 1 0 0 1   1 0   1 1 0 1 1   0 1 1   0 1   1 1 0 1 1 0   1 1 0 1 1 1   1 1 0 1

Variables in dart

8. List

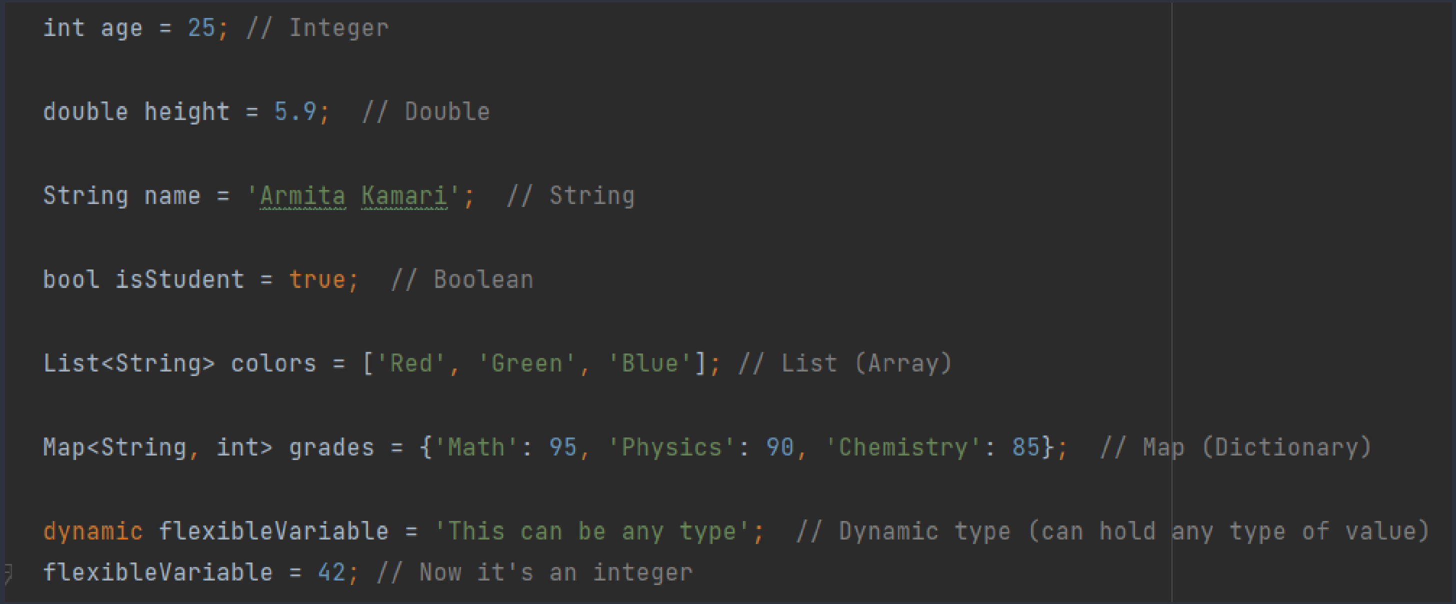
- **Ordered collection of items**: A `List` is an ordered group of objects. In Dart, `List` is a generic type, meaning you can specify the type of elements it contains.

9. Map

- **Key-value pairs**: `Map` represents a collection of key-value pairs. Each key in a map is associated with a value, and both keys and values can be of any type.

1 0 1 1   0 1 1   0 1   1 0 1 1 0 0 1   1 0   1 1 0 1 1   0 1 1   0 1   1 1 0 1 1 0   1 1 0 1 1 1   1 1 0 1

</Code/>



```dart
int age = 25; // Integer

double height = 5.9;  // Double

String name = 'Armita Kamari';  // String

bool isStudent = true;  // Boolean

List<String> colors = ['Red', 'Green', 'Blue']; // List (Array)

Map<String, int> grades = {'Math': 95, 'Physics': 90, 'Chemistry': 85};  // Map (Dictionary)

dynamic flexibleVariable = 'This can be any type';  // Dynamic type (can hold any type of value)
flexibleVariable = 42; // Now it's an integer
```

1 0 1 1 0 1 1 0 1 1 0 1 1 0 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 1 0 1 1 0 1 1 0 1 1 1 1 1 0 1

</>

Null Safety in Dart

?

nullabale

? , ! , ?? , late

1 0 1 1   0 1 1   0 1   1 0 1 1 0 0 1   1 0   1 1 0 1 1   0 1 1   0 1   1 1 0 1 1 0   1 1 0 1 1 1   1 1 0 1

</ Null Safety in Dart [late ,? , ?? , !]

Non-Nullable Types

Definition: A non-nullable type is one that cannot hold a null value. Every non-nullable variable must be initialized before it's accessed.

Nullable Types

Definition: A nullable type can hold either a value or `null`. Nullable types are defined by adding a question mark ? to the type.

1 0 1 1 0 1 1 0 1 1 0 1 1 0 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 1 0 1 1 0 1 1 0 1 1 1 1 1 0 1

</ late , ! , .?

Late keyword

Definition: `late` is used to declare a non-nullable variable that will be initialized later, but before it's used.
Use Case: This is useful when the variable's value is not known at compile-time but is guaranteed to be initialized before use.

Null Assertion Operator !

Definition: The `!` operator forces Dart to treat a nullable variable as non-nullable. This is risky because if the variable is actually `null`, it throws an error.

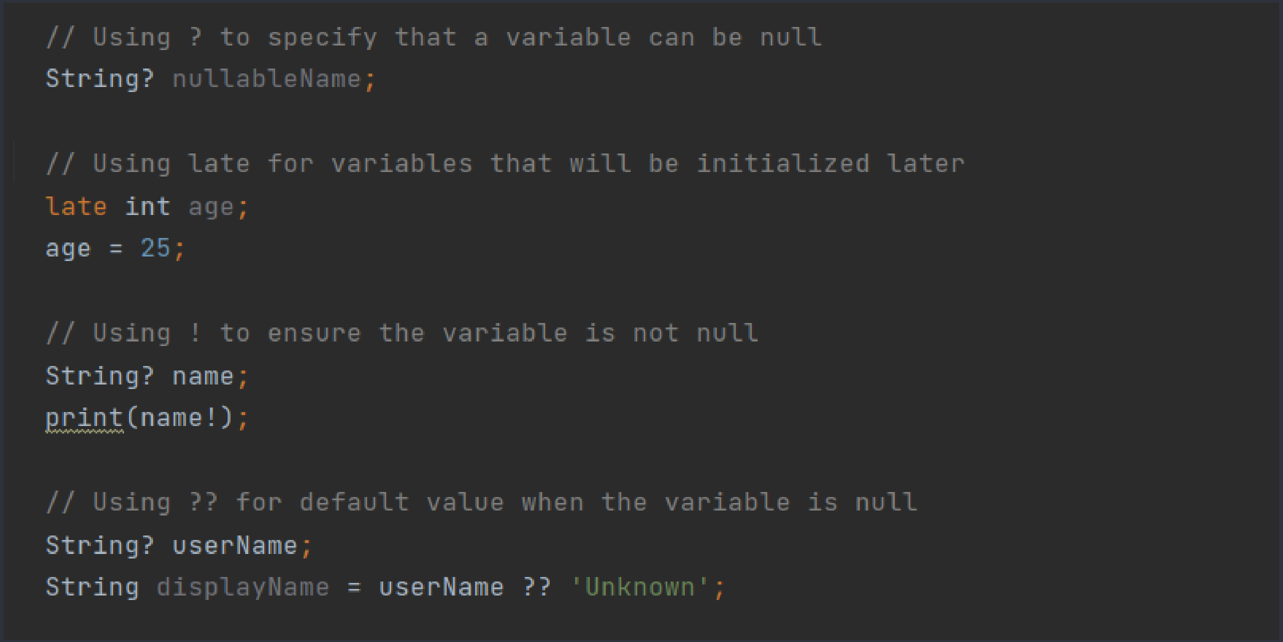
Safe Navigation Operator ?.

Definition: The `?.` operator is used to safely access properties or methods of an object that might be `null`. If the object is `null`, the expression evaluates to `null` instead of throwing an error.

.?/ late / !>

1011 011 01 1011001 10 11011 011 01 110110 110111 1101

</Code/>



```dart
// Using ? to specify that a variable can be null
String? nullableName;

// Using late for variables that will be initialized later
late int age;
age = 25;

// Using ! to ensure the variable is not null
String? name;
print(name!);

// Using ?? for default value when the variable is null
String? userName;
String displayName = userName ?? 'Unknown';
```

1 0 1 1   0 1 1   0 1   1 0 1 1 0 0 1   1 0   1 1 0 1 1   0 1 1   0 1   1 1 0 1 1 0   1 1 0 1 1 1   1 1 0 1

</ Operators in Dart

Arithmetic operators:

+, -, *, /

Comparison operators:

==, !=, >, <

Logical operators:

&&, ||

1011 011 01 1011001 10 11011 011 01 110110 110111 1101

Final and const

Final:

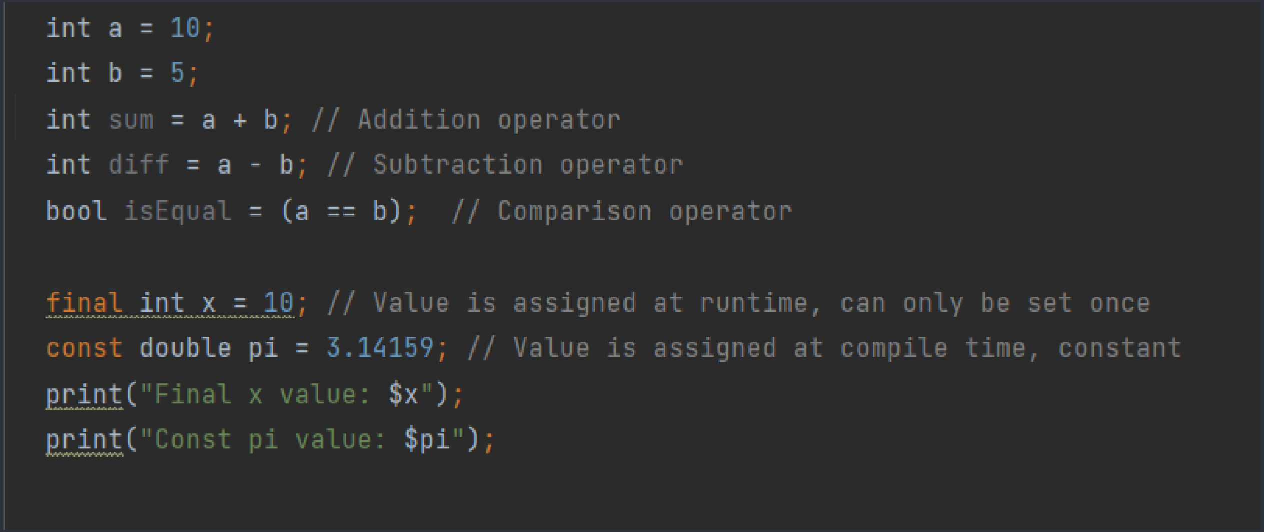
A variable that cannot be reassigned after initialization

Const:

A compile-time constant that cannot change once set

1 0 1 1   0 1 1   0 1   1 0 1 1 0 0 1   1 0   1 1 0 1 1   0 1 1   0 1   1 1 0 1 1 0   1 1 0 1 1 1   1 1 0 1

</Code/>



```
int a = 10;
int b = 5;
int sum = a + b; // Addition operator
int diff = a - b; // Subtraction operator
bool isEqual = (a == b);  // Comparison operator

final int x = 10; // Value is assigned at runtime, can only be set once
const double pi = 3.14159; // Value is assigned at compile time, constant
print("Final x value: $x");
print("Const pi value: $pi");
```

1 0 1 1   0 1 1   0 1   1 0 1 1 0 0 1   1 0   1 1 0 1 1   0 1 1   0 1   1 1 0 1 1 0   1 1 0 1 1 1   1 1 0 1

Access Modifiers in Dart

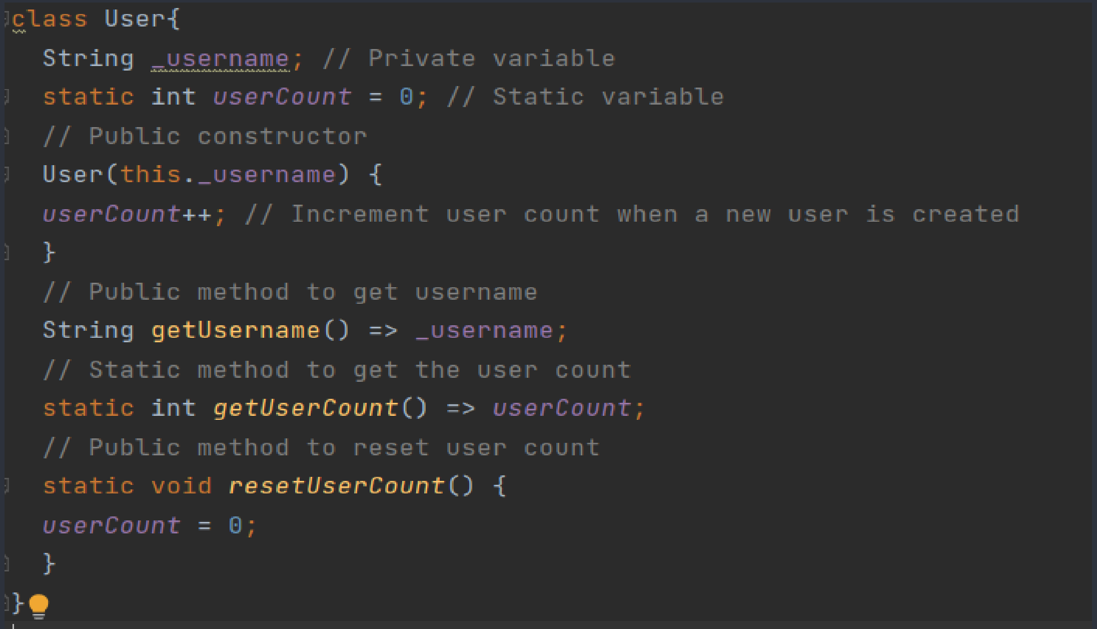
private: Indicated by a variable or method name starting with an underscore (_), accessible only within the same library.

public: By default, all variables and methods are public and accessible from any part of the code.

static: A member of a class that belongs to the class itself rather than to instances of the class.

1011 011 01 1011001 10 11011 011 01 110110 110111 1101

</Code/>



```dart
class User{
  String _username; // Private variable
  static int userCount = 0; // Static variable
  // Public constructor
  User(this._username) {
  userCount++; // Increment user count when a new user is created
  }
  // Public method to get username
  String getUsername() => _username;
  // Static method to get the user count
  static int getUserCount() => userCount;
  // Public method to reset user count
  static void resetUserCount() {
  userCount = 0;
  }
}
```

1 0 1 1 0 1 1 0 1 1 0 1 1 0 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 1 0 1 1 0 1 1 1 1 0 1 1 1 1 1 0 1

</ Methods in Dart

.Getter and Setter Methods

Dart provides getters and setters to access and update object properties

. Positional Parameters

Definition:
Parameters passed in a specific order without names. Positional parameters can be optional by using square brackets [].

. Regular Methods

Standard methods in Dart that take parameters and return values

. Named Parameters

Dart allows you to define methods with named parameters, which make your code more readable

1 0 1 1 0 1 1 0 1 1 0 1 1 0 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 1 0 1 1 0 1 1 1 1 1 0 1

</ Methods in Dart

. Anonymous Functions

Dart allows the creation of functions without names, also known as **lambdas** or **closures**.

.Static Methods

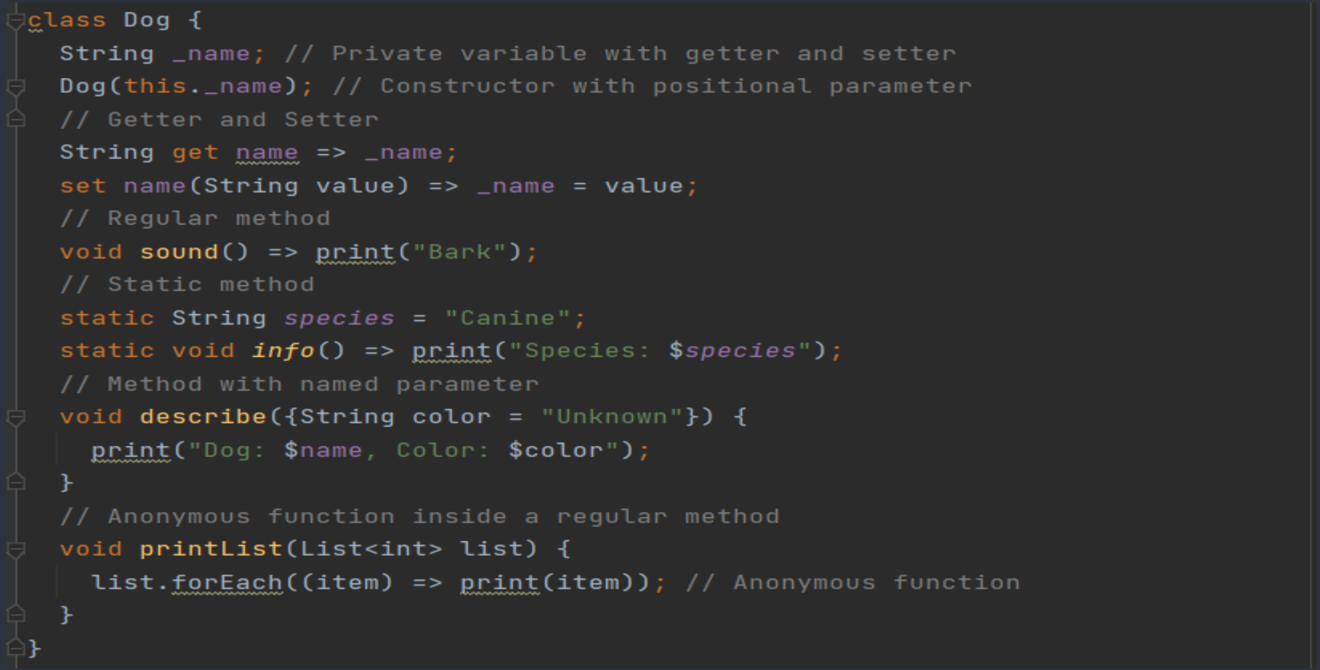
Static methods belong to the class rather than instances of the class. They can be called directly on the class without creating an object.

. Method Overriding

In Dart, methods can be overridden in child classes using the @override annotation

1 0 1 1 0 1 1 0 1 1 0 1 1 0 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 1 0 1 1 0 1 1 1 0 1 1 1 1 1 0 1

</Code/>



```dart
class Dog {
  String _name; // Private variable with getter and setter
  Dog(this._name); // Constructor with positional parameter
  // Getter and Setter
  String get name => _name;
  set name(String value) => _name = value;
  // Regular method
  void sound() => print("Bark");
  // Static method
  static String species = "Canine";
  static void info() => print("Species: $species");
  // Method with named parameter
  void describe({String color = "Unknown"}) {
    print("Dog: $name, Color: $color");
  }
  // Anonymous function inside a regular method
  void printList(List<int> list) {
    list.forEach((item) => print(item)); // Anonymous function
  }
}
```

1 0 1 1   0 1 1   0 1   1 0 1 1 0 0 1   1 0   1 1 0 1 1   0 1 1   0 1   1 1 0 1 1 0   1 1 0 1 1 1   1 1 0 1

Inheritance in Dart

} /> [

Definition: Inheritance is a mechanism where a class (subclass or child) can inherit properties and methods from another class (superclass or parent).
Use Case: It allows code reuse, simplifies the development process, and enables the extension of existing functionality.

dart

1 0 1 1 0 1 1 0 1 1 0 1 1 0 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 1 0 1 1 0 1 1 0 1 1 1 1 1 0 1

Inheritance in Dart

The `extends` Keyword:

In Dart, a subclass is created using the `extends` keyword

Overriding Methods:

A subclass can modify the behavior of methods from its superclass by overriding them using the `@override` annotation.

Calling Superclass Methods:

The `super` keyword is used to call a method from the superclass inside the subclass

Inheritance Hierarchy:

Dart supports **single inheritance**, meaning a class can only inherit from one superclass.

Constructors and Inheritance:

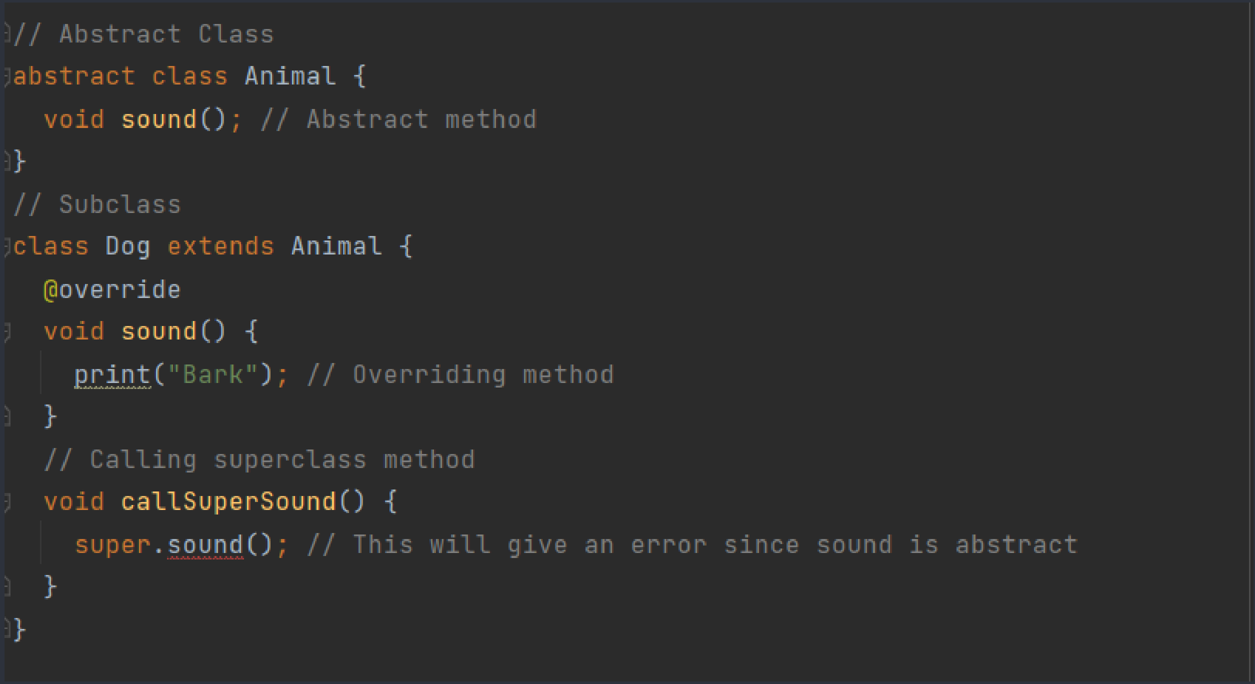
When a subclass is instantiated, the constructor of the superclass is also called. If the superclass constructor has parameters, the subclass must call it explicitly using `super()`.

Abstract Classes:

An abstract class is a class that cannot be instantiated. It serves as a blueprint for other classes.

1 0 1 1 0 1 1 0 1 1 0 1 1 0 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 1 0 1 1 0 1 1 0 1 1 1 1 1 0 1

</Code/>



```
// Abstract Class
abstract class Animal {
  void sound(); // Abstract method
}
// Subclass
class Dog extends Animal {
  @override
  void sound() {
    print("Bark"); // Overriding method
  }
  // Calling superclass method
  void callSuperSound() {
    super.sound(); // This will give an error since sound is abstract
  }
}
```

1 0 1 1   0 1 1   0 1   1 0 1 1 0 0 1   1 0   1 1 0 1 1   0 1 1   0 1   1 1 0 1 1 0   1 1 0 1 1 1   1 1 0 1

Using implements in Dart

} /> [

Definition: In Dart, a class can use the `implements` keyword to enforce the implementation of an interface (a contract of methods and properties that a class must provide).
Difference with `extends`: Unlike inheritance with `extends`, `implements` requires the class to implement **all** the methods and properties of the interface, even if the interface is another class or abstract class.

/>

1 0 1 1 0 1 1 0 1 1 0 1 1 0 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 1 0 1 1 0 1 1 0 1 1 1 1 1 0 1

Using `implements` in Dart

Defining and Implementing an Interface:

In Dart, a subclass is created using the `extends` keyword

Interfaces in Dart:

In Dart, every class can act as an interface. Any class can be used as an interface and then be implemented by another class.

Implementing Multiple Interfaces:

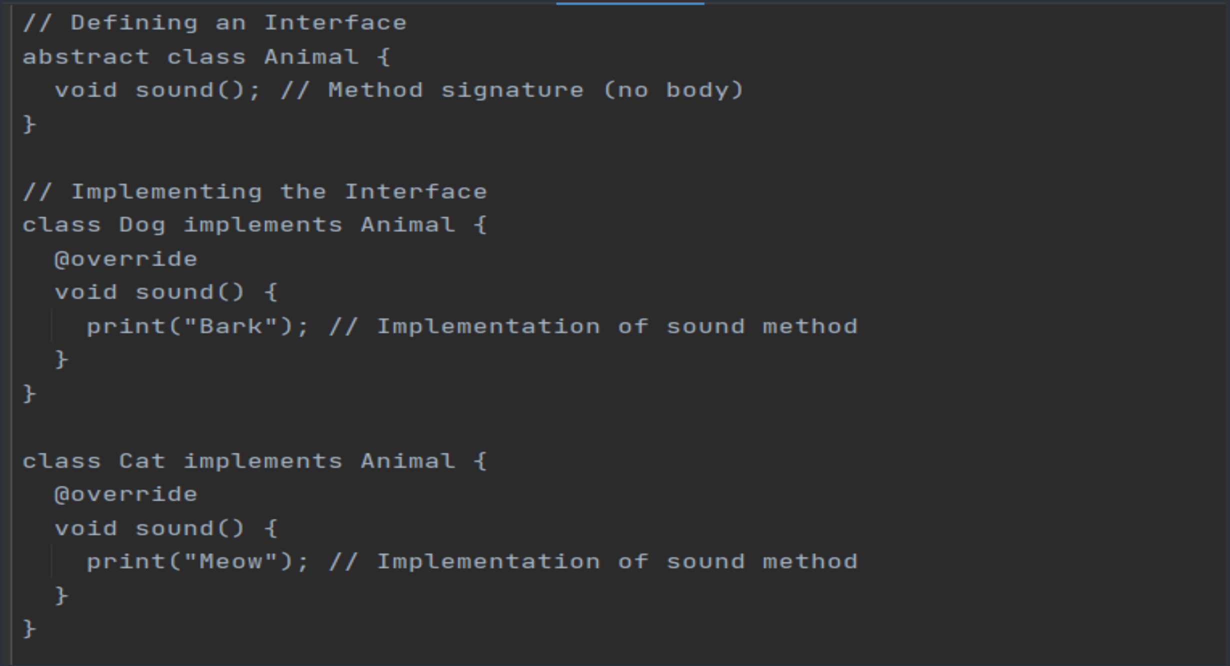
A class in Dart can implement multiple interfaces by separating them with commas

Interfaces and Abstract Classes:

An interface can be derived from an abstract class. The `implements` keyword forces the subclass to override all methods from the abstract class.

1 0 1 1 0 1 1 0 1 1 0 1 1 0 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 1 0 1 1 0 1 1 0 1 1 1 1 1 0 1

</Code/>



```
// Defining an Interface
abstract class Animal {
  void sound(); // Method signature (no body)
}

// Implementing the Interface
class Dog implements Animal {
  @override
  void sound() {
    print("Bark"); // Implementation of sound method
  }
}

class Cat implements Animal {
  @override
  void sound() {
    print("Meow"); // Implementation of sound method
  }
}
```

1 0 1 1   0 1 1   0 1   1 0 1 1 0 0 1   1 0   1 1 0 1 1   0 1 1   0 1   1 1 0 1 1 0   1 1 0 1 1 1   1 1 0 1

Pitch Deck

20xx

</

Constructors in Dart

/>

}/>[

1 0 1 1   0 1 1   0 1   1 0 1 1 0 0 1   1 0   1 1 0 1 1   0 1 1   0 1   1 1 0 1 1 0   1 1 0 1 1 1   1 1 0 1

</Constructors in Dart

Default Constructor

If no constructor is explicitly defined, Dart provides a default constructor

Named Constructors

Dart allows the creation of multiple constructors in a class by using named constructors.

Redirecting Constructors

A constructor in Dart can redirect to another constructor within the same class using the : this() syntax.

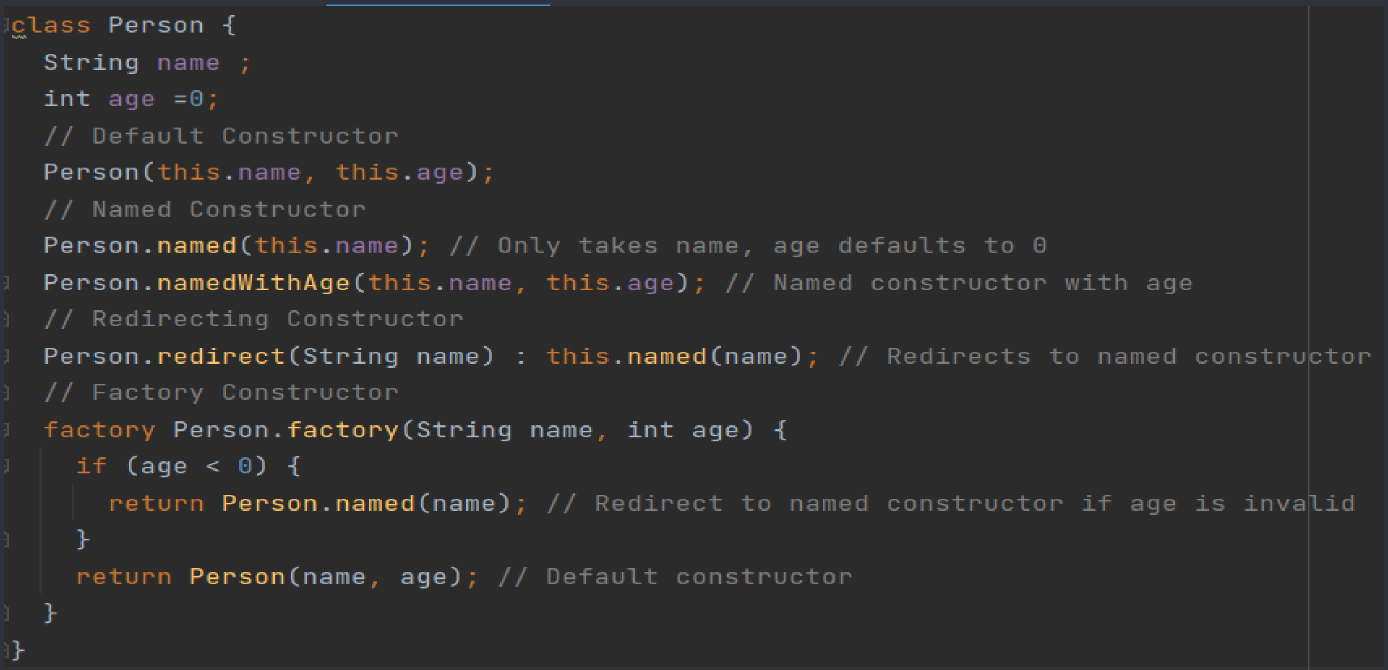
Factory Constructor

A factory constructor is used to return an instance of a class, often used for singleton patterns or when complex initialization is needed.

- **Definition**: A constructor is a special method used to initialize objects of a class.
- **Purpose**: It sets the initial values for object properties.
- **Types of Constructors**: Dart supports several types of constructors such as default, named, and factory constructors.

1 0 1 1 0 1 1 0 1 1 0 1 1 0 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 1 0 1 1 0 1 1 0 1 1 1 1 1 0 1

</Code/>



```
class Person {
  String name ;
  int age =0;
  // Default Constructor
  Person(this.name, this.age);
  // Named Constructor
  Person.named(this.name); // Only takes name, age defaults to 0
  Person.namedWithAge(this.name, this.age); // Named constructor with age
  // Redirecting Constructor
  Person.redirect(String name) : this.named(name); // Redirects to named constructor
  // Factory Constructor
  factory Person.factory(String name, int age) {
    if (age < 0) {
      return Person.named(name); // Redirect to named constructor if age is invalid
    }
    return Person(name, age); // Default constructor
  }
}
```

1 0 1 1 0 1 1 0 1 1 0 1 1 0 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 1 0 1 1 0 1 1 0 1 1 1 1 1 0 1

Summary of Constructors in Dart

- **Default Constructors**: Automatically created if no constructor is defined.
- **Named Constructors**: Allow for multiple ways of constructing an object.
- **Factory Constructors**: Control object creation and return instances based on custom logic.
- **Redirecting Constructors**: Redirect to other constructors in the same class.
- **Named and Optional Parameters**: Provide flexibility and readability.

1011 011 01 1011001 10 11011 011 01 110110 110111 1101

</Thanks!

?, !, .?

armytakmry464@gmail.com

Fall 1403-1404

1 0 1 1 0 1 1 0 1 1 0 1 1 0 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 1 0 1 1 0 1 1 0 1 1 1 1 1 0 1