R2S
Resource Software Solution

OOP in Dart (cont)

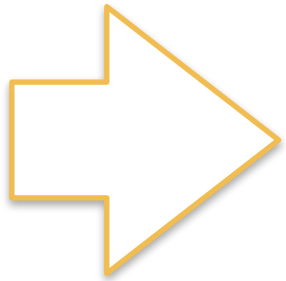
Contents

Constructor in Dart (1)

- For example, the following code creates a **Person** class object and sets the values for the **name** and **age** properties.

```
Class Person {
  String? name;
  int? age;
}
```

```
Person person = Person();
person.name = "John";
person.age = 30;
```



```
Person person = Person("John", 30);
```

Constructor

Constructor in Dart (2)

- **A constructor** is a special **method** used to **initialize** an object (**initial values** for the **object's properties).** It is **called automatically** when an **object is created.**
- The **constructor's name** should be the **same** as the **class name.**
- Constructor **doesn't** have any **return type.**
- Syntax

```
class ClassName {
  // Constructor declaration
 ClassName() {
    // body of the constructor
  }
}
```

```
Class Person {
 String? name;
 int? age;
 Person(String name, int age) {
    this.name = name;
    this.age = age;
  }
}
```

```
var p = Person('John', 18);
```

Constructor in Dart (3)

- Example with single line

```dart
// Constructor in short form
Person(this.name, this.age, this.subject, this.salary);
```

```dart
class Teacher {
  String? name;
  int? age;
  String? subject;
  double? salary;

  // Constructor
  Teacher(String name, int age, String subject, double salary) {
    this.name = name;
    this.age = age;
    this.subject = subject;
    this.salary = salary;
  }
  // Method
  void display() {
    print("Name: ${this.name}");
    print("Age: ${this.age}");
    print("Subject: ${this.subject}");
    print("Salary: ${this.salary}\n"); // \n is used for new line
  }
}

void main() {
  // Creating teacher1 object of class Teacher
  Teacher teacher1 = Teacher("John", 30, "Maths", 50000.0);
  teacher1.display();

  // Creating teacher2 object of class Teacher
  Teacher teacher2 = Teacher("Smith", 35, "Science", 60000.0);
  teacher2.display();
}
```

Types of Constructors (1)

- **Default** Constructors: If you don't define any constructors, Dart provides a default one with **no arguments**. It initializes all instance variables to their default values (null for objects, 0 for numbers, etc.)
- Example

```
class User {
  String id;
  String name;
}

void main() {
  User user = User();        // Uses the default constructor
  print(user.name);
}
```

Types of Constructors (2)

- **Parameterized** Constructor: A parameterized constructor allows you to pass **arguments to initialize** an object
- Example

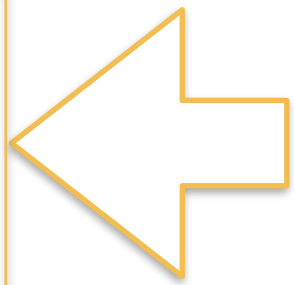
```
class User {
  String id;
  String name;

  // Parameterized constructor
  User(this.id, this.name);
}

void main() {
  User user = User('u123', 'John');
  print('Id: ${user.id}, Name: ${user.name}');
}
```

Types of Constructors (3)

- **Named** Constructor: Named constructors are useful when you need **multiple constructors** for a class.
- Example

```
class User {
  late String id;
  late String name;

  // Parameterized constructor
  User(this.id, this.name);

  // Named constructor
  User.fromId(this.id) {
      this.name = 'No name';
  }
}
```

```
void main() {
  User user = User.fromId('U0001');
  print(user.id);
  print(user.name);
}
```



Second constructor

Types of Constructors (4)

- **Constant** Constructor: It is used to create compile-time **constant objects**. They ensure the object's state is known at compile time and cannot be changed after creation.
- Example

```
class Point {
  final int x;
  final int y;

  const Point(this.x, this.y);
}

void main() {
  const point = Point(10, 20);
  // point.x = 30; // This would cause an error (constant cannot be changed)
}
```

Types of Constructors (5)

- Example 1: **Factory Constructor with Conditional Logic (handling complex instantiation logic)**

```
class Shape {
  // Factory constructor
  factory Shape(String type) {
    if (type == 'circle') {
      return Circle();
    } else if (type == 'square') {
      return Square();
    } else {
      throw ArgumentError('Invalid shape type');
    }
  }
}
```

Types of Constructors (6)

- Example 2: **Factory Constructor for Singleton (reusing existing instances)**

```
class APIService {
  // Private constructor
  APIService._getInstance();

  // The single instance of the class
  static final APIService _instance = APIService._getInstance();

  // Factory constructor
  factory APIService() {
    return _instance; // Always return the same instance
  }
}
```

```
void main() {
  // Both variables will refer to the same instance
  var api1 = APIService();
  var api2 = APIService();
  print(identical(api1, api2)); // true
}
```

Some common scenarios

1. **Database Connection**: Singleton to manage a single database connection instance
2. **API Service**: Singleton to manage network requests with a single instance

Example: Singleton for API Service

Imagine you have an app that communicates with a remote server using HTTP requests. You want to ensure that you use a single instance of the http.Client (or another HTTP package like Dio) for all API calls, as creating multiple instances could cause resource overhead.

Types of Constructors (7)

- **Factory** Constructor: Factory constructors are useful when you need **control over object creation**, such as **reusing existing instances** or **handling complex instantiation logic.**
- Syntax

```
class ClassName {
  // Factory constructor
  factory ClassName() {
    // Custom instantiation logic here
  }
}
```

```
class Shape {
  // Factory constructor
  factory Shape(String type) {
    if (type == 'circle') {
      return Circle();
    } else if (type == 'square') {
      return Square();
    } else {
      throw ArgumentError('Invalid shape type');
    }
  }
}
```

```
class APIService {
  // Private constructor
  APIService._getInstance();

  // The single instance of the class
  static final APIService _instance = APIService._getInstance();

  // Factory constructor
  factory APIService() {
    return _instance; // Always return the same instance
  }
}
```

Exercise

- Challenge:
  - Create a class **Patient** with three properties **name**, **age**, and **disease**. The class has **one constructor**. The constructor is used to **initialize the values of the three properties**.
  - Create an object of the class **Patient** called **patient**.
  - **Print the values** of the three properties using the object.

Encapsulation in Dart (1)

- Example

employee.dart

```
class Employee {
  // Private properties
  int? _id;
  String? _name;

// Setter method to update private property _id
  void setId(int id) {
    this._id = id;
  }

  // Getter method to access private property _id
  int getId() {
    return _id ?? 0;
  }

  // Setter method to update private property _name
  // Getter method to access private property _name
}
```

employee_managment.dart

```
void main() {
  // Create an object of Employee class
  Employee employee = Employee();

  // setting values to the object using setter
  //employee._id = 1;
  employee.setId(1);
  employee.setName("John");

  // Retrieve the values of the object using getter
  print("Id: ${employee.getId()}");
  print("Name: ${employee.getName()}”);
}
```

Encapsulation in Dart (2)

- **Encapsulation** means **hiding data** within a library, preventing it from outside factors.
- Encapsulation can be achieved by:
  - Declaring the class properties as **private** by using **underscore(_)**.
  - Providing public **getter** and **setter** methods to **access** and **update the value** of private property.
- Why encapsulation is important?
  - **Security**: It allows you to restrict access to the class members.
  - **Flexibility**: It allows you to change the implementation of the class without affecting the code outside the class.

Encapsulation in Dart (3)

- Example **employee.dart**

```
class Employee {
  // Private property
  int? _id;
  // Setter method to update private property _id
  void setId(int? id) {
    this._id = id;
  }

 // Getter method to access private property _id
  int getId() {
    return _id ?? 0;
  }
}

void main() {
  var employee = Employee();
  employee._id = 1001; // It is working, but why?
  print(employee.getName());
}
```

Why aren't private properties private?

- Summary
  - Using underscore (_) before a variable or method name **makes it library (library is one file) private not class private.**
  - If you write the **main method in a separate file**, this will **not work.**
- A more concise way to define

```
class Employee {
  int? _id;

  // Getter
  int get id => _id ?? 0;

  // Setter
  set id(int? id) {
    _id = id;
  }
}
```

Inheritance in Dart (1)

- Example

person.dart

```
class Person {
  String name;
  int age;

  Person(this.name, this.age);

  void displayInfo() {
    print("Name: $name, Age: $age");
  }
}
```

employee.dart

```
class Employee extends Person {
  int employeeId;
  String department;

  // Constructor
  Employee(String name, int age, this.employeeId, this.department)
      : super(name, age);

  @override
  void displayInfo() {
    super.displayInfo(); // Call base class method
    print("Employee ID: $employeeId, Department: $department");
  }
}
```

Inheritance in Dart (2)

- One class (called a "subclass" or "child class") **inherits properties and methods** from another class (called a "superclass" or "parent class")
- This helps reusable codebase by allowing shared code to reside in a common base class, while specific behaviors can be defined in subclasses.
- Key Concepts of Inheritance Dart
  1. **Super Class**: The class whose properties and methods are inherited.
  2. **Sub Class**: The class that inherits the properties and methods.
  3. **extends Keyword**: Used to specify inheritance in Dart.
  4. **Method Overriding**: Allows a subclass to provide a specific implementation of a method already defined in its superclass.

Inheritance in Dart (3)

- Using the Classes

hr_managment.dart

```
void main() {
  Person person = Person("Alice", 30);
  person.displayInfo();
  // Output:
  // Name: Alice, Age: 30

  Employee employee = Employee("Bob", 25, 101, "Engineering");
  employee.displayInfo();
  // Output:
  // Name: Bob, Age: 25
  // Employee ID: 101, Department: Engineering
}
```

Inheritance in Dart (4)

- Example

person.dart

```
abstract class Person {
  String name;
  int age;

  Person(this.name, this.age);

  void displayInfo() {
    print("Name: $name, Age: $age");
  }
}
```

employee.dart

```
class Employee extends Person {
  int employeeId;
  String department;

  // Constructor
  Employee(String name, int age, this.employeeId, this.department)
      : super(name, age);

  @override
  void displayInfo() {
    super.displayInfo(); // Call base class method
    print("Employee ID: $employeeId, Department: $department");
  }
}
```

```
void main() {
  Person person = Person("Alice", 30);
  person.displayInfo();
}
```

Abstraction in Dart (1)

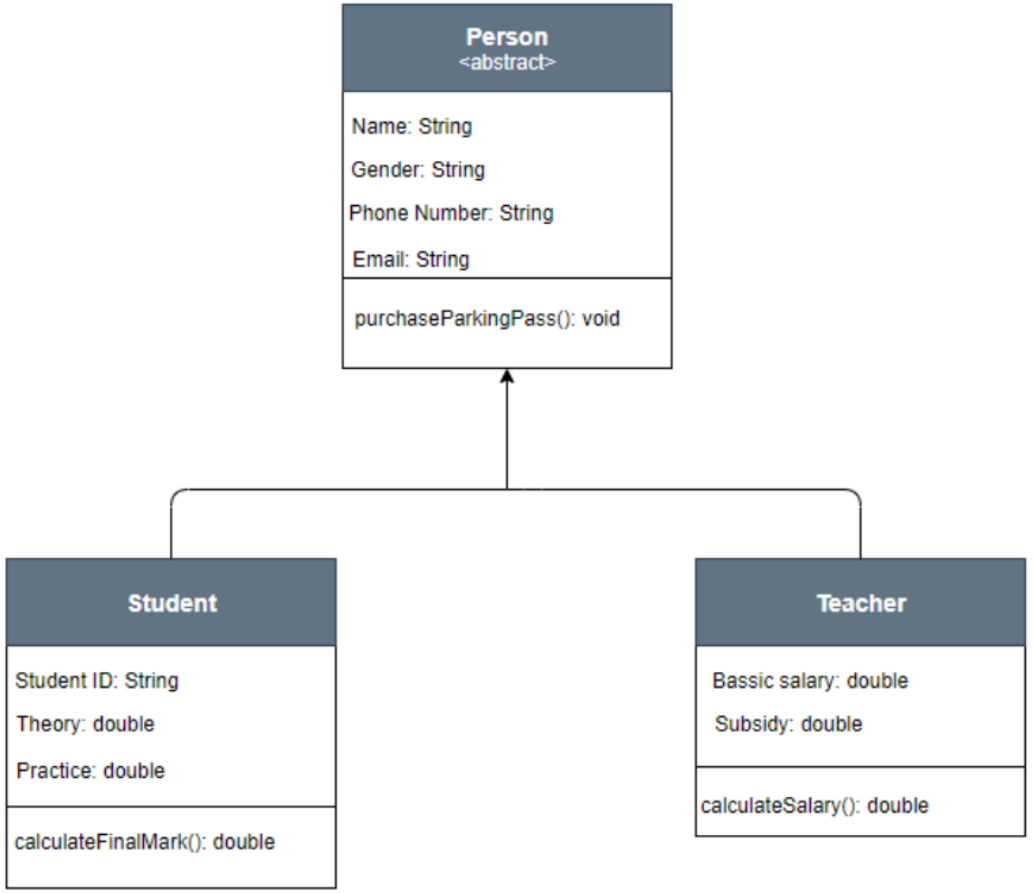
- It is declared with **abstract** keyword. Abstraction is the **class** that contains **one** or **more** or may **not** contain **abstract methods** (methods without implementation).
- Abstract class cannot be **initialized**. ~~**Person p = Person;**~~
- It can be inherited using the **extend** keyword then the **abstract methods** need to be **implemented**.

```
abstract class Person {
  String name;
  int age;

  Person(this.name, this.age);

  void displayInfo() {
    print("Name: $name, Age: $age");
  }
}
```

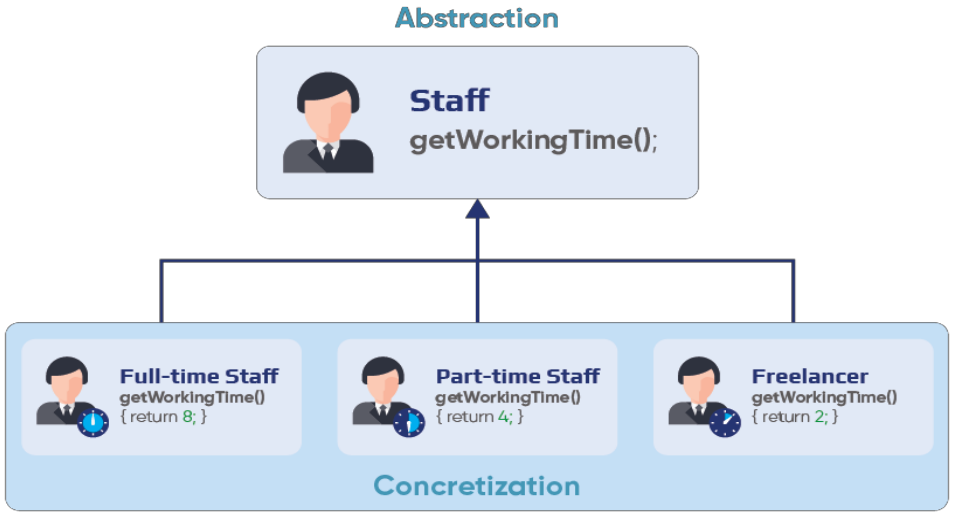
Abstraction in Dart (2)

Resource Software Solution

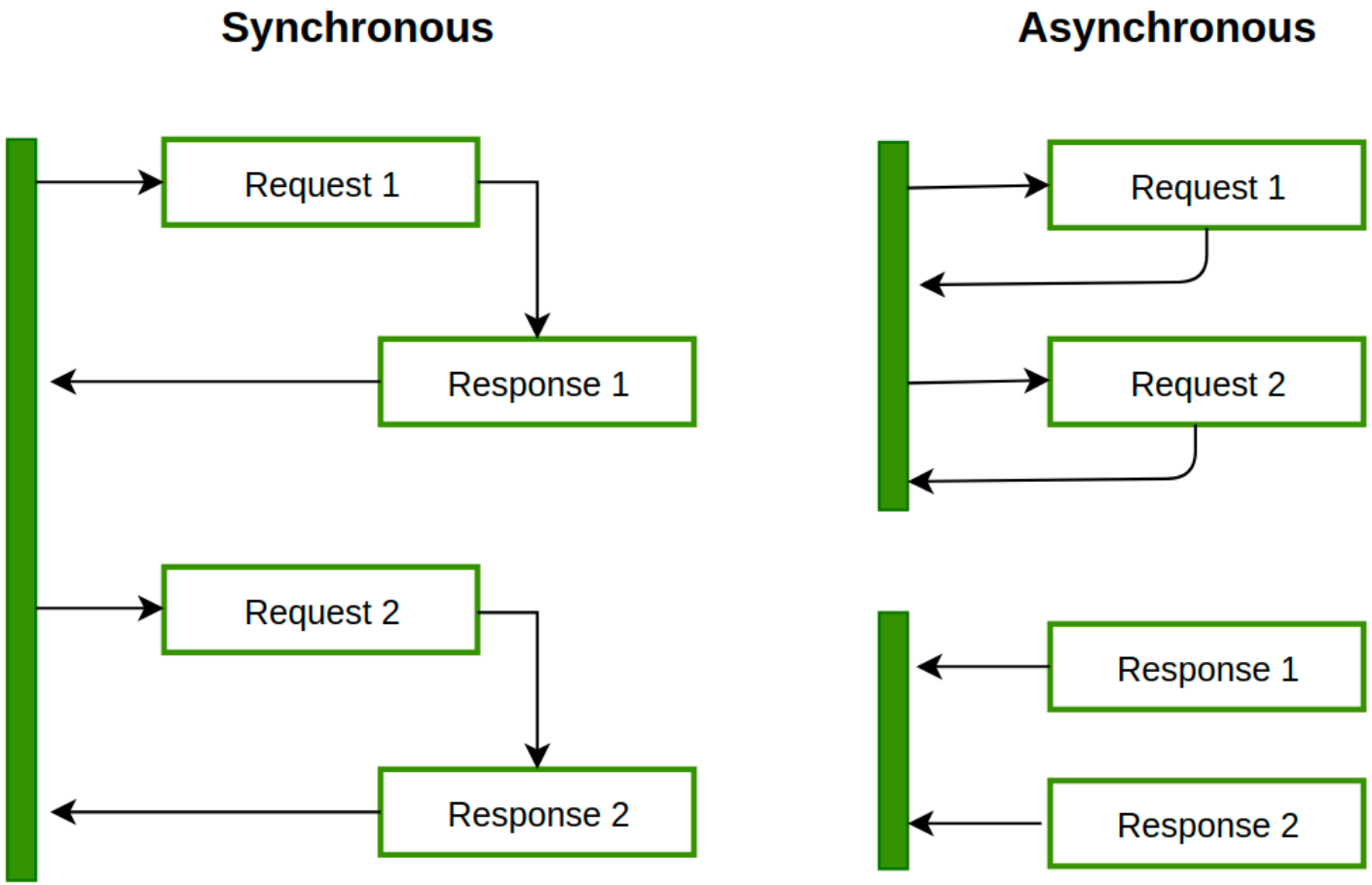
It serve as blueprints for subclasses



Contract



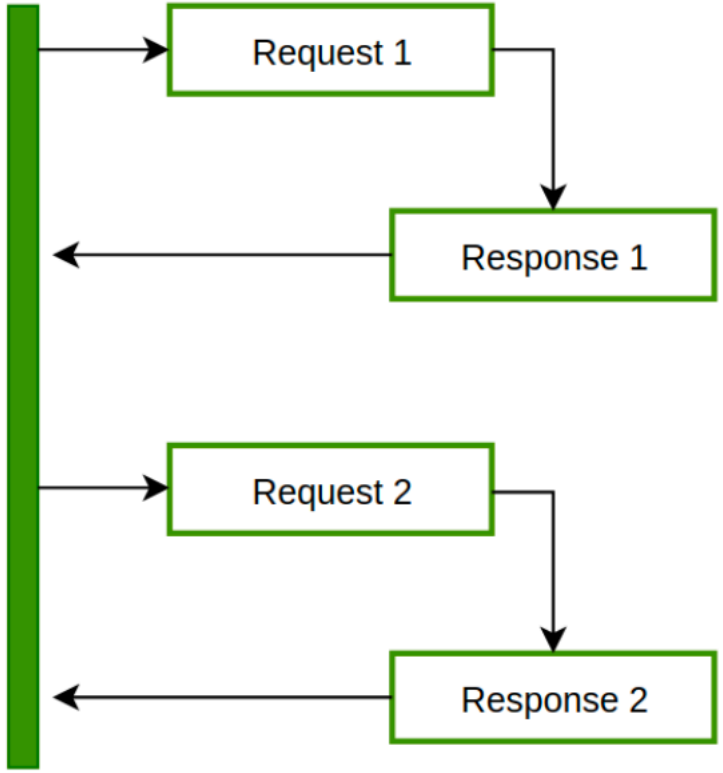
Synchronous vs Asynchronous



Synchronous Programming

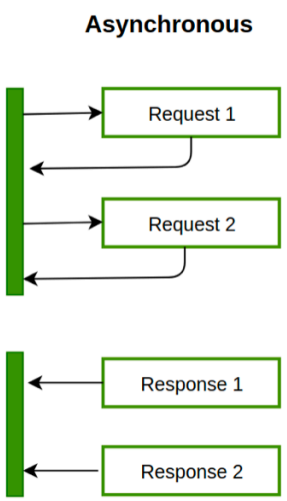
- The program is **executed line by line**, one at a time. Synchronous operation means a **task that needs to be solved before proceeding to the next one**.
- Example

```
main() {
    print("First Operation");
    print("Second Big Operation");
    print("Third Operation");
    print("Last Operation");
}
```



Asynchronous Programming (1)

- Program **execution continues to the next line without waiting to complete other work.**



async, await and **Future**

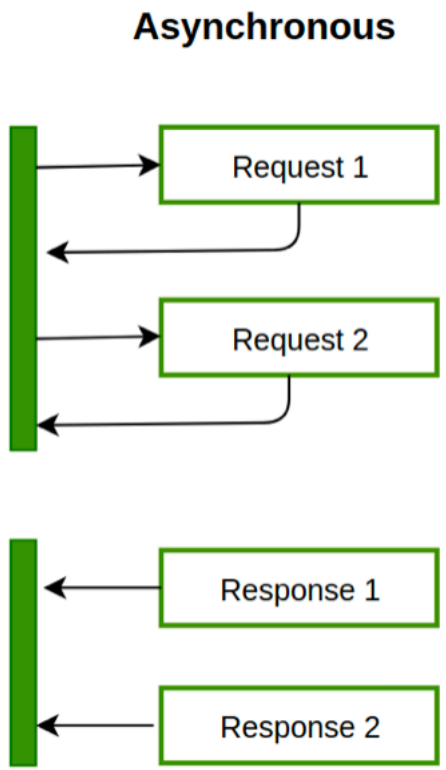
1. Use the **async** keyword before a **function** body to make it **asynchronous**.
2. Use the **await** keyword to **get the completed result** of an asynchronous expression.
3. **Future** represents an **asynchronous** operation that may complete with a value

```
main() {
      print("Start");
      printData();
      print("End");
}

void printData() async {
  String data = await getData();
  print(data);
}

Future<String> getData(){
  return Future.delayed(
                  Duration(seconds:3), () => "Hello");
}
```

Asynchronous

Asynchronous Programming (2)

- Why we need asynchronous
  - To fetch data from Internet,
  - To write something to Database,
  - To read data from File, and etc



Dart Coding Convention

- A surprisingly important part of good code is good style.
- Consistent naming, ordering, and formatting helps code that is the same look the same.
- If we use a consistent style across the entire Dart ecosystem, it makes it easier for all of us to learn from and contribute to each others' code.
- Link: https://dart.dev/effective-dart/style

*Keeping up those **inspiration** and the **enthusiasm** in the **learning path**. Let confidence to bring it into **your career path** for getting gain the **success** as your expectation.*

Thank you

Contact

- Name: R2S Academy
- Email: daotao@r2s.edu.vn
- Phone/Zalo: 0919 365 363
- FB: https://www.facebook.com/r2s.tuyendung
- Website: www.r2s.edu.vn

Questions and Answers