# 20CYS312 - Principles of Programming Languages
# Exploring Programming Paradigms

**Assignment-01**

**Presented by «Rakshan K»**
**«CB.EN.U4CYS21059»**
**TIFAC-CORE in Cyber Security**
**Amrita Vishwa Vidyapeetham, Coimbatore Campus**

Feb 2024


AMRITA
VISHWA VIDYAPEETHAM

# Outline



# Introduction to Reactive Programming

- Reactive programming is a paradigm that revolutionizes the handling of asynchronous data streams and events in software development.
- Unlike traditional imperative programming, reactive programming focuses on reacting to changes as they happen, emphasizing responsiveness and real-time updates.
- Widely employed in scenarios demanding dynamic data handling, such as user interfaces, event-driven systems, and applications requiring continuous data stream management.
- The essence of reactive programming lies in creating systems that seamlessly respond to evolving conditions, offering enhanced interactivity and a more engaging user experience.

# Basic Principles of Reactive Programming

- **Observables:** Represent asynchronous data streams or sequences of events. They can emit values over time, and other components can subscribe to react to these emissions.
- **Observers (Subscribers):** Components that subscribe to observables to receive and react to emitted values. They define the actions to be taken when data is emitted, completed, or an error occurs.
- **Operators:** Transform, filter, or combine observables to create new streams. Operators provide powerful ways to manipulate and compose asynchronous data streams.
- **Subscription:** Establishes a connection between an observable and an observer. It defines when the observer starts receiving values and how long it will continue to listen.
- **Schedulers:** Control the execution context of observables and their observers. Schedulers help manage concurrency, allowing developers to control when and where computations take place.
- **Backpressure:** Mechanism to handle situations where an observable produces values at a rate that the observer cannot process. It helps prevent overwhelming the sy with data.

# Real-World Examples of Reactive Programming

1. **Financial Trading Systems:**
   - Reactive programming is employed in financial trading systems to handle real-time market data streams and execute trades with low latency. It ensures timely responses to market changes and efficient processing of financial events.
2. **Social Media Feeds:**
   - Platforms like Twitter and Facebook use reactive programming to update social media feeds in real-time. As new posts, comments, or notifications occur, the system reacts dynamically to provide users with instant updates.
3. **Online Gaming:**
   - In online gaming, reactive programming is employed to manage multiplayer interactions, synchronize game state changes, and deliver real-time updates to players. It enhances the gaming experience by ensuring responsive and interactive gameplay.
4. **Internet of Things (IoT) Systems:**
   - IoT applications use reactive programming to handle streams of sensor data from various devices. It enables efficient processing, analysis, and reaction to real-time events, contributing to the responsiveness and reliability of IoT systems.
5. **E-commerce Platforms:**
   - Reactive programming is applied in e-commerce platforms to manage inventory changes, process customer orders, and update product availability in real-time. It ensures that users receive accurate and timely information during their shopping experience.

# Implementing Reactive Programming in Dart

- **Use RxDart Library:** Dart developers often leverage the RxDart library, a reactive programming library for Dart and Flutter, to implement reactive patterns.
- **Working with Streams and Observables:** RxDart provides the ‘Stream‘ and ‘Observable‘ classes, which are essential for representing and working with asynchronous data streams in Dart.
- **Creating Observables:** Developers can create observables using various methods, such as ‘Observable.create‘ or using factory methods like ‘Observable.fromIterable‘. This allows defining how data will be emitted over time.
- **Applying Operators:** RxDart includes a rich set of operators to transform, filter, and combine observables. Operators like ‘map‘, ‘filter‘, ‘debounce‘, and ‘merge‘ facilitate powerful data stream manipulation.
- **Subscribing to Observables:** Components subscribe to observables using the ‘listen‘ method, specifying how to react to emitted values, errors, and completion signals.
- **Managing Subscriptions:** Dart developers manage subscriptions using the ‘StreamSubscription‘ class, allowing them to control when to start listening and how long the subscription should last.
- **Integrating with Flutter:** In Flutter applications, reactive programming is commonly used to handle UI updates, user interactions, and asynchronous tasks, enhancing the overall responsiveness of the app.

# Example Code: Reactive Programming in Dart with RxDart

**Example Code 1 :** Below is a simple example demonstrating the use of RxDart in Dart for reactive programming.

```
import 'package:rxdart/rxdart.dart';

void main() {
  final subject = BehaviorSubject<int>();

  subject.stream
      .where((value) => value % 2 == 0)
      .map((value) => 'Even: $value')
      .listen((data) => print(data));

  subject.add(1);
  subject.add(2);
  subject.add(3);
  subject.add(4);

  subject.close();
}
```

# Code Snippet: Reactive Programming in Dart

**Example Code 2 :** This Dart code snippet uses the StreamController to create a simple stream, adds values to it, and listens for those values.

```
import 'dart:async';

void main() {
  final streamController = StreamController<String>();

  final streamSubscription = streamController.stream.listen(
    (value) => print(value),
    onDone: () => print('Stream completed'),
  );

  streamController.sink.add('Hello');
  streamController.sink.add('World');
  streamController.close(); // Completes the stream
}
```

# Introduction to Aspect-Oriented Programming (AOP)

- **Definition:** AOP is a programming paradigm for modularizing cross-cutting concerns, making it easier to manage functionalities that affect multiple parts of a program.
- **Cross-Cutting Concerns:** AOP addresses challenges related to cross-cutting concerns such as logging, security, and error handling, which span multiple modules or layers in a program.
- **Key Concepts:**
  - **Advice:** Code for a specific cross-cutting concern.
  - **Pointcut:** Specifies where advice should be applied in the program's execution.
  - **Aspect:** Combination of advice and pointcut, encapsulating a cross-cutting concern.
- **Benefits of AOP:** AOP improves separation of concerns, enhances code maintainability, encourages code reusability, and results in cleaner, more readable code.

# Basic Principles of Aspect-Oriented Programming (AOP)

1. **Separation of Concerns:**
   - AOP promotes the separation of cross-cutting concerns from the main business logic, leading to cleaner and more modular code.
   - Concerns such as logging, security, and error handling are isolated into aspects, enhancing maintainability.
2. **Aspect:**
   - An aspect is a modular unit that encapsulates a cross-cutting concern, combining advice (code) and a pointcut (where the advice should be applied).
   - Aspects promote code reusability and can be applied to multiple modules or layers in an application.
3. **Advice:**
   - Advice contains the code that needs to be executed for a specific cross-cutting concern.
   - Types of advice include "before" (executed before a join point), "after" (executed after a join point), and "around" (wraps a join point).
4. **Pointcut:**
   - A pointcut specifies a set of join points in the program's execution where advice should be applied.
   - Pointcuts define the locations or conditions for weaving aspects into the code.

# Real-World Examples of Aspect-Oriented Programming (AOP)

1. **Logging:** AOP simplifies consistent logging by weaving logging aspects into methods, ensuring centralized and uniform log implementation.
2. **Security:** AOP aids in implementing security measures, such as authentication and authorization, by applying security aspects to relevant methods or classes.
3. **Transaction Management:** AOP centralizes transaction-related code, simplifying maintenance and reducing redundancy in scenarios requiring consistent transaction behavior.
4. **Error Handling:** AOP provides a centralized approach to handle exceptions uniformly across an application, improving code reliability and maintainability.
5. **Performance Monitoring:** AOP facilitates systematic performance monitoring by applying aspects to collect metrics and profiling data without modifying core application logic.

# Implementing AOP in Django

1. **Install ‘django-aspect‘:**
   - Use ‘pip install django-aspect‘ to install the ‘django-aspect‘ package.
   - Add ‘’django-aspect’‘ to ‘INSTALLED-APPS‘ in Django settings.
2. **Create an Aspect:**
   - Define an aspect class inheriting from ‘django-aspect.Aspect‘.
   - Implement advice methods within the aspect for specific cross-cutting concerns.
3. **Apply Aspects with Decorators:**
   - Use ‘@aspect.before‘ or ‘@aspect.after‘ decorators to apply aspects to functions or methods in Django views, models, or other components.
4. **Example Code:**
   - See a basic example below applying logging aspects to Django views for performance monitoring.

# Example Code: Applying Logging Aspect in Django

```
from django_aspect import aspect

class PerformanceAspect(aspect.Aspect):
    @aspect.before("your_app.views.*")
    def log_request_start(self, sender, **kwargs):
        print("Request started")

    @aspect.after("your_app.views.*")
    def log_request_end(self, sender, **kwargs):
        print("Request ended")
```

# AOP vs. Reactive Programming

## Aspect-Oriented Programming (AOP)

- **Focus:** Separation of concerns by modularizing cross-cutting aspects.
- **Concerns:** Addresses issues like logging, security, and error handling.
- **Modularity:** Improves code maintainability and reusability.
- **Key Concepts:** Aspects, Advice, Pointcut.
- **Implementation:** Implemented using decorators, annotations, or dedicated AOP frameworks.
- **Example Use Case:** Logging, Security, Transaction Management.

## Reactive Programming

- **Focus:** Asynchronous data streams and event-driven programming.
- **Data Streams:** Emphasizes the flow of data and events over time.
- **Key Concepts:** Observables, Observers, Operators, Schedulers.
- **Concurrency:** Manages asynchronous computations and events.
- **Implementation:** Utilizes reactive libraries such as RxJava, RxJS, or RxDart.
- **Example Use Case:** User Interfaces, Real-time Applications, IoT Systems.

# References


1. https://medium.com/@alvaro.armijoss/reactive-programming-in-flutter-c577d8e056d2
2. https://dev.to/bplab98/a-taste-of-reactive-programming-in-flutter-with-rxdart-and-flutterbloc-3p1
3. https://www.linkedin.com/pulse/reactive-programming-principles-explained-luis-soares-m-sc-/
4. https://www.javatpoint.com/dart-features
5. https://dev.to/ayoubzulfiqar/reactive-programming-with-rxdart-in-flutter-with-example-3m8e