

20CYS312 - Principles of Programming Languages

Exploring Programming Paradigms

Assignment-01

Presented by Suganth Sarvesh S

CB.EN.U4CYS21076

TIFAC-CORE in Cyber Security

Amrita Vishwa Vidyapeetham, Coimbatore Campus

Feb 2024



AMRITA
VISHWA VIDYAPEETHAM



- 1 Reactive Programing Paradigm
- 2 Reactive Programing in RxJava
- 3 Prototype-Based Programing Paradigm
- 4 Prototype-Based Programing Paradigm - Javascript
- 5 Comparison
- 6 Bibliography



Reactive Programming



RxJava



The Need for Reactive Programming

- **Challenges in Traditional Programming:**

- **Asynchronous Task Management:** Traditional programming struggles with running multiple tasks concurrently, leading to potential performance issues.
- **UI Responsiveness:** Long tasks causes delays and unresponsiveness in the user interface, which will negatively impact the user's overall experience and frustration.
- **Real-Time Event Handling:** Effectively managing and reacting to real-time events poses difficulties in traditional programming models.

- **Reactive Programming as a solution for challenges in Traditional Approach:**

- **Asynchronous Task Management:** Reactive programming addresses this using an event-driven approach and non-blocking execution.
- **UI Responsiveness:** Reactive programming ensures UI responsiveness by executing tasks concurrently without blocking.
- **Real-Time Event Handling:** Reactive programming simplifies this process with its event-driven architecture and streamlined event handling techniques.



Reactive Programming (RP) is a declarative programming based Paradigm that is aimed at creating efficient and responsive applications. Rooted in some of the principles of functional programming and influenced by observer patterns in object-oriented languages. RP uses the core concepts for usage:

- **Event-Driven Paradigm:**

- It adapts dynamically to changing conditions and events.
- It also facilitates real-time interaction and responsiveness.

- **Concurrent and Non-blocking Execution:**

- It supports parallelized execution of tasks.
- It also enhances application flow efficiency during multi-operation handling.

- **Optimized Data Stream Management:**

- It simplifies administration of continuous data streams.
- It also improves the effectiveness in processing and utilizing extensive quantities of streaming data.



Basic Components of Reactive Programming

Reactive Programming utilizes three fundamental components, which form the cornerstone of its architecture. They are:

1 Observables:

- **Definition:** Observables are the source of data or events in Reactive Programming.
- **Functionality:** They provide items over time, providing data streams.
- **Role:** They are the only producers in RP, producing and pushing data streams or events.

2 Observers:

- **Definition:** Observers react to the items produced by Observables.
- **Functionality:** They handle the produced data or events, responding to changes.
- **Role:** Consumers that observe and react to the data produced by Observables.

3 Operators:

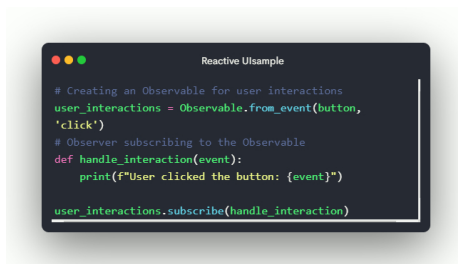
- **Definition:** Operators transform, filter, or combine data streams in Reactive Programming.
- **Functionality:** They manipulate the items created by Observables, resulting in powerful transformations.
- **Role:** They shape and process data streams efficiently by applying transformations.

These components establish the groundwork for Reactive Programming when implemented together, they offer a structured approach to solve tasks.



Example Scenario: Responsive user interface in python

Reactive programming in a sample user interace:



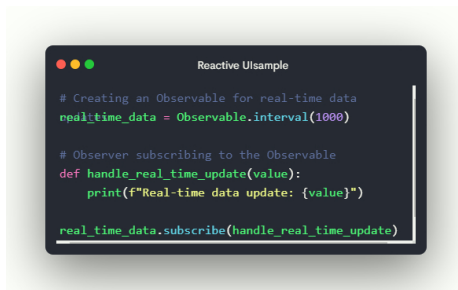
- *user-interactions* is the Observable. It shows the stream of events that occur when the 'click' event happens on the button.
- *handle-interactions* is the observer. It decides the result of the event, here the button is clicked is the event.
- *subscribe* is the Operation. It establishes a connection between the Observable (*user-interactions*) and the Observer (*handle-interaction*).



Reactive Programming in Practice - II

Example Scenario: Handling Data Streams

Reactive programming in a sample data streams:



- *real-time-data* is the Observable. It shows the stream of data updates emitted at regular intervals (given interval is 1000).
- *handle-real-time-updates* is the observer. It decides the action to be taken when a value is updated. (prints the value here).
- *subscribe* is the Operation. It establishes a connection between the Observable and the Observer.



Example Scenario: Managing UI

Reactive programming in a managing a user interface:



- *user-interactions* is the first Observable. It represents a stream of 'click' events on the button.
- *data-stream* is the second Observable. It is got from the first Observable (*user-interactions*) using the map operation. The map operation transforms each 'click' event by applying the process-event function to it.
- *subscribe* is the Operation. It establishes a connection between the Observable and the Observer.



- *handle-processed-data* is the Observer. It decides the action to be taken when a processed event is emitted by the second Observable (*data-stream*). In this case, it prints a message indicating the processed data.
- The operation here is the *subscribe* method. It establishes a connection between the second Observable and the Observer. When a processed event occurs, the Observer's *handle-processed-data* function is executed.



RxJava, short for Reactive Extensions for Java, is part of the larger **ReactiveX** project that originated at Microsoft in 2009. The idea was to provide a unified model for handling *asynchronous* and *event-based programming* across different programming languages. ReactiveX gained popularity quickly, and various implementations, including RxJava for Java, emerged. RxJava was developed by Netflix as an open-source project to help manage the complexity of their video streaming service. It became a powerful tool for dealing with asynchronous and concurrent programming challenges.

RxJava is now an *open-source*, Java-compatible implementation of the Reactive Extensions (ReactiveX) library, specifically targeting the JVM environment and Android platforms. RxJava is essentially a Java library that allows reactive programming in Java, designed to simplify asynchronous and event-driven application development. In general, RxJava revolves around the reactive programming concepts of *observer* and *observables*. *Observables* are general data producers, and observers are consumers of those data produced.



RxJava: Key Aspects and Advantages

Some aspects that highlight the reasons for the implementation of reactive programming with RxJava include:

- ➊ **Effortless Multithreading:** Simplify background task management with RxJava's seamless threading integration, especially important in UI threading.
- ➋ **Easy Integration with Third-party Libraries:** Easily adapt RxJava to existing tools and infrastructure by integrating with third-party libraries like RxAndroid (RxJava implementation in Android Development), Vert.x (Event Driven Application framework), etc.
- ➌ **Error Handling:** Robust error management features, such as retry logic and centralized exception aggregation, alleviate debugging burdens for easier error handling.
- ➍ **Advanced Caching:** Boost performance with customizable caching options based on time, size, count, or custom policies.
- ➎ **Clean Code, Fluent APIs:** Enhance code readability with RxJava's user-friendly APIs for easy function calls.



Key Components of RxJava

Observables :

- RxJava relies on observables to support its reactive programming foundation.
- They act as foundations as they release data streams over time.
- They are ideal for asynchronous and event-driven coding, they also tackle dynamic and live situations deftly.

Observers:


- They have the responsibility for reacting to the items supplied by observables.
- They have the function of receiving data or events originating from observables, by which observers promote independence among components.
- Using observers lets us separate parts of the reactive system, which makes us easy to design and change our approach accordingly.

Operators:

- Operators hold a very significant role in RxJava since they are needed for applying transformations, altering, and looking through data streams.
- By utilizing appropriate operators, developers modify the items given by observables according to specific needs for the problem.
- Operators handle the asynchronous sequences making them less susceptible to blocking and other problems like it.



Observable:



```
// Creating an observable for user interactions with a map operator
Observable<String> userInteractions = Observable.create(emitter -> {
    // Simulating user clicks
    emitter.onNext("Click Event 1");
    emitter.onNext("Click Event 2");
    emitter.onNext("Click Event 3");
    emitter.onComplete();
}).map(event -> event.toUpperCase()); // Using the map operator to transform the
events
```

- *Observable<String>*: Here we define an observable that produces data streams or events of the type strings.
- *Observable.create*: Here we create an observable using the create method.
- *emitter.onNext*: Here we emit the events, if particular trigger is triggered.



RxJava implementation - II Contd.

Observers:

```
Title

// Creating an observer for processing user interactions
Observer<String> userInteractionObserver = new Observer<String>() {
    @Override
    public void onSubscribe(Disposable d) {
        // Subscription logic if needed
    }

    @Override
    public void onNext(String event) {
        System.out.println("User clicked: " + event);
    }

    @Override
    public void onError(Throwable e) {
        // Error handling logic
    }

    @Override
    public void onComplete() {
        System.out.println("User interaction processing
completed.");
    }
};
```



Observers (Contd.):

- *Observer<String>*: Here we define an observer that receives data streams or events of the type strings.
- *new Observer<String>()* ... : Here we create a new instance that implements observer interface.
- *onError(Throwable e)*: Here we implement an error handling method.



Operators:

```

// Creating an observable for user interactions with a map operator
Observable<String> userInteractions = Observable.create(emitter -> {
    // Simulating user clicks
    emitter.onNext("Click Event 1");
    emitter.onNext("Click Event 2");
    emitter.onNext("Click Event 3");
    emitter.onComplete();
}).map(event -> event.toUpperCase()); // Using the map operator to transform the
events
```

- `.map(event -> event.toUpperCase())`: Here we use the map operator to transform the emitted events into uppercase.
- Similarly, we can apply more operators and transform the data to the wishes of the developer.
- Some of the generally used operators in RxJava are Filter Operator, Transforma Operator, Combining Operator and etc



Neflix: Netflix faced challenges in dealing with the complexities of event-driven and asynchronous programming. To address these issues, they adopted RxJava, a library that significantly improved their backend servers and Android clients. This adoption resulted in tangible benefits, such as reduced memory leaks and a decrease in the number of bugs in their software.

SoundCloud: SoundCloud, utilized RxJava to effectively handle various media-related events like buffer pauses, seek controls, and shifts in audio focus. Following the implementation, the development team observed significant enhancements in code organization, readability, and error correction.



Pros and Cons of Reactive Programming - RxJava

Benefits and Pros :

- **Quick Response Time:** Reactive Programming rapidly answers user actions and real-time occasions.
- **Scalability:** Systems utilizing Reactive Programming flexibly manage fluctuating workloads and user activity.
- **Better Resource Use:** Smart allocation improves performance and cuts downtime.

Cons :

- **Appropriateness:** Not all apps gain from Reactive Programming, as extra complexity might complicate possible perks in simple, straight forward apps.
- **Performance Latency:** More elaborate event flows can increase delay and additional computation cost in systems.
- **Memory Issues:** Improper subscription management may trigger memory leaks, raising long-run resource usage.



Prototype-Based Programming



The Need for Prototype-Based Programming

Challenges in Traditional Programming:

- ❶ Linear Approach: Development delays due to the linear nature.
- ❷ Limited Flexibility: New changes may require reworking of existing parts.
- ❸ Late Error Detection: Identification of errors occurs late in development.
- ❹ User Involvement: Users involved in later phases, leading to changing specifications.

Prototype-Based Programming as a Solution:

- ❶ Simultaneous Work: Parallel development speeds up the process.
- ❷ Flexibility in Design: Easily accommodates new changes.
- ❸ Early Error Detection: Identifies errors during development.
- ❹ User Involvement: Actively involves users, accommodating changes efficiently.



Prototype-Based Programming is a powerful style of dynamic object-oriented programming paradigm known for its adaptable approach to object creation and inheritance. At its core, this paradigm revolves around the utilization of prototypical objects, acting as templates to generate new objects and share common attributes, methods, and functionality among related entities.

Two Core Components:

- **Object:** It is an entity containing data and functions, It often represents real-world entities or abstract ideas.
- **Prototype:** A "blueprint" object used as a basis for creating new objects; typically embodying shared properties and behaviors amongst similar entities.



Key principles in Prototype-Based Programming:

- **Prototypes as Foundational Building Blocks:** Prototype-based programming create objects based on existing prototypes. Developers duplicate and modify these prototypes to create new objects, promoting efficient code reuse and flexibility.
- **No hierarchy in Objects Prototypes:** The hierarchical general model is not used in the objects in prototype-based programming interact directly with each other, acquiring attributes and methods from their inherited prototypes.
- **Delegation:** When searching for features like attributes or methods, objects first check themselves. If unsuccessful, they check their prototypes dynamically during program execution, enhancing usability compared to traditional static method of check.



Creation of Objects:

- Objects are dynamically created using their respective prototypes in Prototype-Based programming, providing flexibility for developers to define and modify prototypes independently.
- In JavaScript, constructor functions play a crucial role in creating objects. They act as blueprints, determining the properties and methods inherited by the created objects.

Prototype Inheritance:

- Prototypal inheritance is a key concept in JavaScript for object-oriented programming, allowing objects to inherit properties and methods from their prototypes.
- The prototype chain is a hierarchical structure illustrating inheritance relationships. Objects inherit properties not only from their direct prototype but also from higher-level prototypes in the chain.



Core Concepts of Prototype-Based Programming in Practice:

```
// Code here// Constructor function for creating Person objects
function Person(name, age) {
  this.name = name;
  this.age = age;
}

// Method shared by all Person instances
Person.prototype.sayHello = function() {
  console.log(`Hello, my name is ${this.name} and I am ${this.age} years
  old.`);
}

// Constructor function for creating Student objects, inheriting from Person
function Student(name, age, grade) {
  Person.call(this, name, age);
  this.grade = grade;
}
```



Core Concepts of Prototype-Based Programming in Practice: Contd.

```
// Inheriting from the Person prototype
Student.prototype =
Object.create(Person.prototype);
// Method specific to Student instances
Student.prototype.showGrade = function() {
  console.log(`I am in grade ${this.grade}.`);
};

// Creating instances
const person1 = new Person("John", 30);
const student1 = new Student("Alice", 25, 8);

// Calling methods
person1.sayHello();
student1.sayHello();
student1.showGrade();
```



Explanation of Code:

- The *Person* constructor function initializes an object with *name* and *age* properties. The *sayHello* method is added to the *Person* prototype, allowing all instances to share this behavior.
- The *Student* constructor function is created, inheriting from *Person* using *Person.call(this, name, age)*. The *Student* prototype is set to be an object created from *Person.prototype*, establishing the inheritance.
- The *showGrade* method is added to the *Student* prototype, providing functionality specific to student instances.



Prototype-Based Programming in JavaScript

JavaScript, a highly versatile and extensively utilized programming language, involves prototype-based programming as an important part of it. This programming paradigm facilitates *dynamic object creation* and *prototypal inheritance*, offering a powerful and efficient basis for constructing prototypal systems.

In JavaScript, prototype-based programming centers around prototypes, essentially acting as blueprints for other objects. Every JavaScript object comes with a *prototype*, and during the object creation process, it inherits attributes and functionalities from its prototype. *Constructors*, which are specialized functions for initializing objects, play a important role in implementing this paradigm.



Prototype-Based Programming in JavaScript: Key Aspects and Advantages

Some aspects that highlight the reasons for the implementation of prototype-based programming in JavaScript include:

- Provides a simple and flexible way to create and manipulate objects.
- Enables efficient memory usage through the use of shared prototypes.
- Utilizes a prototype chain for a hierarchical organization of objects, promoting a clear and logical code structure.
- Results in a clean and easy-to-understand code structure, making it simpler for developers to comprehend the relationships between objects and their prototypes.
- Objects can be easily extended and modified during runtime, promoting a more agile and adaptable coding style, making it easier for updating without massive code refactoring.



Key Components of Prototype-Based Programming in JavaScript

- **Objects:** They are essentially containers for storing data attributes and function methods.
- **Prototypes:** They act as templates, serving as blueprints for creating new objects based on the prototype.
- **Constructor Functions:** Used for instantiating objects, serving as blueprints for creating new instances.
- **Prototype Chain:** Creates a hierarchy illustrating how objects acquire properties from their respective prototypes and the higher-level prototypes.
- **Dynamic Object Creation:** Objects are created dynamically based on their respective prototypes, facilitating easy definition and updating.



JavaScript Implementation:

```
Title

// Code here// Constructor function for the base object
function Person(name, age) {
    this.name = name;
    this.age = age;
}

// Adding a method to the prototype of the base object
Person.prototype.sayHello = function () {
    console.log(`Hello, my name is ${this.name} and I am ${this.age} years
    old.`);
};

// Creating an instance of the base object
const person1 = new Person("John", 25);
person1.sayHello();

// Constructor function for an object inheriting from the base object
function Student(name, age, grade) {
    // Call the constructor of the base object
    Person.call(this, name, age);

    // Additional property for the derived object
    this.grade = grade;
}
```



JavaScript Implementation:

```
Title

// Setting up the prototype chain - Student prototype inherits from Person
Student.prototype = Object.create(Person.prototype);

// Adding a method to the prototype of the derived object
Student.prototype.showGrade = function () {
    console.log(`I am a student and my grade is ${this.grade}.`);
};

// Creating an instance of the derived object
const student1 = new Student("Alice", 20, "A");
student1.sayHello(); // Inherited method from Person
student1.showGrade(); // Method specific to Student

// Dynamically adding a method to the base object's prototype
Person.prototype.introduce = function () {
    console.log(`Hi, I'm ${this.name}. Nice to meet you!`);
};

person1.introduce(); // New method added to the prototype
student1.introduce(); // Inherited method from Person's prototype
```



- *Person* is a constructor function for a basic object with properties *name* and *age*.
- *Student* is another constructor function that inherits from *Person* and has an additional property *grade*.
- The *prototype chain* is created using *Object.create* to set *Student.prototype* as an object that inherits from *Person.prototype*.
- *Dynamic object creation* is shown here clearly by adding a new method (*introduce*) to the prototype of the base object (*Person*) after instances have already been created. The new method becomes available to all existing and future instances.



Instagram:

- Overview: Instagram migrated its web application to React, an open-source UI library developed by Facebook, constructed with the Prototype-Based Programming paradigm in JavaScript.
- Results: This move led to a remarkable 50

BMW Group:

- Overview: BMW's "i Window Into the Future" campaign showcases interactive 3D vehicle interior images using Three.js, a JavaScript library that leverages Prototype-Based Programming.
- Impact: The captivating website attracts millions of viewers, utilizing JavaScript's prototype programming for managing scenes, cameras, and geometry. It has received prestigious industry recognition.



Pros and Cons of Prototype-based Programming

Pros:

- **Dynamic Object Creation:** Objects can be created dynamically during runtime based on their prototypes, providing flexibility in defining and updating objects.
- **Efficient Memory Usage:** Shared prototypes allow for efficient memory usage as multiple objects can reference and reuse the same prototype.
- **Clear Code Structure:** The prototype chain creates a hierarchical organization of objects, promoting a clear and logical code structure.

Cons:

- **Limited Support:** Compared to class-based languages, prototype-based programming may have limited tooling support and may not be as widely adopted in certain development ecosystems.
- **Large-Scale System Challenges:** In large-scale applications, managing and maintaining prototypes and their relationships can become challenging, leading to potential complexities.



Comparison between Programming Paradigms

Similarities:

1. **Dynamic Behavior:**

- Both paradigms excel in dynamic behavior.
- RxJava handles dynamic data streams.
- Prototype-based JavaScript dynamically creates and modifies objects.

2. **Object-Centric:**

- RxJava and Prototype-based JavaScript are object-centric.
- RxJava uses Observables and Observers.
- Prototype-based JavaScript manipulates objects through prototypes.

3. **Hierarchical Organization:**

- Both embrace hierarchical organization.
- RxJava uses Observables and Operators for data stream hierarchy.
- Prototype-based JavaScript establishes hierarchy via the prototype chain.

4. **Flexibility and Adaptability:**

- Both offer flexibility.
- RxJava adapts to changing data streams.
- Prototype-based JavaScript allows dynamic changes in object structure through prototypes.

5. **Asynchronous Capabilities:**

- Both support asynchronous programming.
- RxJava explicitly caters to it.



Comparison between Programming Paradigms

Differences:

1. Execution Context:

- RxJava excels in handling dynamic data streams, focusing on events and reactions.
- Prototype-Based JavaScript is more geared towards object creation, manipulation, and inheritance.

2. Primary Focus:

- RxJava primarily revolves around managing asynchronous data streams and transformations.
- Prototype-Based JavaScript focuses on the dynamic creation, modification, and inheritance of objects.

3. Programming Abstraction:

- RxJava introduces the abstraction of Observables, Observers, and Operators for reactive programming.
- Prototype-Based JavaScript employs the abstraction of prototypes and constructor functions for dynamic object creation.

4. Use Cases:

- RxJava is commonly used in scenarios where reactive and event-driven programming is crucial, such as UI interactions or real-time data processing.
- Prototype-Based JavaScript finds applications in object-oriented design, particularly when flexibility and dynamic changes in object structure are essential.



The exploration of Reactive Programming in RxJava and Prototype-based Programming in JavaScript has unveiled distinctive paradigms, each offering unique solutions to specific challenges in software development. Reactive Programming, exemplified by RxJava, proves invaluable in scenarios demanding responsiveness and efficient handling of asynchronous tasks. Its core concepts of Observables and Operators facilitate a streamlined approach to managing dynamic data streams. On the other hand, Prototype-based Programming in JavaScript centers around object-oriented flexibility, emphasizing dynamic object creation and modification through prototypes. The prototype chain provides a hierarchical structure for objects, enhancing adaptability. Both paradigms exhibit strengths and challenges, with RxJava excelling in real-time and concurrent applications, while JavaScript's prototype-based approach shines in object-oriented designs. The choice between these paradigms relies on the nature of the application and its specific requirements.



Bibliography

Links for Articles and Blogs:

- <https://dev.to/efkumah/why-javascript-is-a-prototype-based-oop-4b4g>
- <https://www.koombea.com/blog/what-is-reactive-programming/>: :text=Reactive
- <https://en.wikipedia.org/wiki/ReactiveX>
- <https://rxjs.dev/guide/overview>
- <https://rxjs.dev/guide/observable>
- <https://rxjs.dev/guide/operators>
- <https://rxjs.dev/guide/observer>
- <https://react.dev/learn/writing-markup-with-jsx>
- <https://www.oreilly.com/library/view/building-reactive-systems/9781449361022/>
- <https://medium.com/@jamesmeaney/what-is-reactive-programming-bafebbbbbfc>
- <https://netflixtechblog.com/reactive-programming-in-the-netflix-api-with-rxjava-7811c3a1496a>: :text=Reactive
- <https://javascript.info/prototype-inheritance>
- https://www.w3schools.com/js/js_object_prototypes.asp
https://eloquentjavascript.net/06_object.html
- <https://www.geeksforgeeks.org/types-of-observables-in-rxjava/>
- <https://interviewprep.org/backend-software-engineer-interview-questions/>
- <https://www.lirmm.fr/~dony/postscript/proto-book.pdf>



Thank You!

