20CYS312 - Principles of Programming Languages
Exploring Programming Paradigms

Assignment-01

Presented by «Anu Priya»
«CB.EN.U4CYS21007»
TIFAC-CORE in Cyber Security
Amrita Vishwa Vidyapeetham, Coimbatore Campus

Feb 2024

AMRITA
VISHWA VIDYAPEETHAM

Outline

Principles and Concepts:

- Data Streams: Continuous flows of information (e.g., user actions, sensor readings) processed incrementally.
- Declarative Approach: Code specifies what needs to be done with data transformations rather than explicit instructions.
- Event-Driven Processing: Systems listen for events and react to changes, promoting responsiveness.
- Non-Blocking Operations: Avoids thread-blocking calls through asynchronous operations.
- Backpressure: Regulates data flow to prevent overwhelming downstream consumers. Resiliency: Designed to recover from failures gracefully, with built-in error handling.
- Compositionality: Small, independent operators combine for complex data pipelines, enhancing code modularity.

Observer Pattern and Observables:

- Observer Pattern: Design pattern with observers and subjects. Observers observe changes in one or more subjects. Subjects keep a list of observers and notify them of state changes. Observable and Operators:
- Observable: Emits a sequence of events; can be cold, hot, or connectable.
- Operators: Functions that transform or manipulate data streams.
- Reactive Extensions (Rx): Library for implementing reactive programming patterns.

Applications of Reactive Programming

Where to use?

- Microservices Architecture
- User Interfaces
- Multithreading and Concurrency
- Real-time Applications (e.g., Financial Trading, IoT, Collaboration Tools, Media Streaming)
- Network Requests, Database Operations, Event-Driven Systems, Parallel Processing, Backpressure Control, Error Handling, Testing.

Benefits:

- Improved responsiveness, scalability, and resilience.
- Simplified, modular, and testable code.

Introduction:

- RxJava is a Java VM implementation of Reactive Extensions: a library for composing asynchronous and event-based programs by using observable sequences.
- It extends the observer pattern to support sequences of data/events and adds operators that allow you to compose sequences together declaratively while abstracting away concerns about things like low-level threading, synchronization, thread-safety and concurrent data structures.

Key Characteristics of RxJava:

- Asynchronous Data Handling
- Declarative Programming Style
- Backpressure Support
- Operator-Based Composition
- Rich Operator Toolkit
- Thread Management
- Error Handling Mechanisms

Key concepts:

- Observable, Observer, Subscription, Schedulers
- Creating Observables (e.g: Observable.just(), Observable.range())
- Composing and Transforming Observables (e.g: map, flatMap, concatMap, zip)

Why RxJava for Reactive Programming?

- Improved responsiveness and scalability.
- Simplified code through a declarative style.
- Enhanced error handling and resilience.
- Better testability due to modularity.
- Integration with Java libraries for seamless development

Hello world

```java
package rxjava.examples;

import io.reactivex.rxjava3.core.*;

public class HelloWorld {
    public static void main(String[] args) {
        Flowable.just("Hello world").subscribe(System.out::println);
    }
}
```

Figure: hello world example

Example 1

```
Observable<Integer> observable = Observable.fromCallable(() -> {
return 1;
});
```

Figure: example1

```
observable.subscribe((x) -> {
 System.out.println("Result " + (x + 2));
});
```

Figure: example1

```
Result 3
```

Figure: result

Example 2

```
Observable<Integer> observable1 = Observable.just(1, 2, 3, 4, 5);
                Consumer<Integer> addValues = (x) -> System.out.println("Adding " + x + " ::" + (x + 2));
                Consumer<Integer> subValues = (x) -> System.out.println("Subtracting " + x + " :: " + (x - 1));
                observable1.filter(x -> x > 1).subscribe(x -> addValues.accept(x));
                observable1.filter(x -> x < 2).subscribe(x -> subValues.accept(x));
```

Figure: example2

```
Adding 2 ::4
Adding 3 ::5
Adding 4 ::6
Adding 5 ::7
Subtracting 1 :: 0
```

Figure: result

Principles and concepts

- Objects as Prototypes: Objects serve as blueprints for creating new objects.
- Runtime Flexibility: Dynamic modification and extension of objects during runtime.
- No Explicit Classes: Direct object creation without traditional class definitions.
- Prototype Chain: Objects linked in a chain for inheritance.
- Dynamic Property Addition: Properties and methods can be added or modified at runtime.
- Inheritance Through Prototypal Delegation: Objects inherit from prototypes through delegation.

Applications of Prototype-Based Programming

Real-World Applications of Prototype-Based Programming:

- Web Development: React, Vue, Angular, and jQuery use prototypes for UI development.
- Game Development: Efficient object creation and behavior customization.
- Server-Side Development: Node.js modules and Express.js framework utilize prototypes.
- Other Domains: GUIs, AI, NLP for dynamic UI construction, adaptive learning, and language modeling.

The prototype-based programming paradigm is used in JavaScript for its simplicity, flexibility, and alignment with the language's object-oriented nature.

It offers dynamic and runtime extensibility, efficient memory usage, and easy code reusability through prototypal inheritance. This approach eliminates the need for strict class definitions, allowing for quick object creation and manipulation. While JavaScript also supports class syntax, the prototype-based paradigm remains fundamental to the language's design.

Introduction

Prototypes are the mechanism by which JavaScript objects inherit features from one another. In this article, we explain what a prototype is, how prototype chains work, and how a prototype for an object can be set.
Prerequisites: Understanding JavaScript functions, familiarity with JavaScript basics (see First steps and Building blocks), and OOJS basics (see Introduction to objects).
Objective: To understand JavaScript object prototypes, how prototype chains work, and how to set the prototype of an object.

Key Concepts

- Prototype Chain: Every object in JavaScript has a built-in property called its prototype. The prototype is itself an object, forming a chain of prototypes known as the prototype chain. The chain ends when a prototype with null as its own prototype is reached. The property that points to an object's prototype is often named –proto–. Object.getPrototypeOf() is the standard way to access an object's prototype.
- Accessing Prototype Properties: Objects in JavaScript have additional properties beyond those explicitly defined, like –defineGetter–, –proto–, and others. These properties are part of the prototype chain and can be accessed using the object's name followed by a period in the console.
- Object's Prototype: Every object in JavaScript inherits from a prototype, typically Object.prototype. Object.getPrototypeOf() can be used to determine an object's prototype. The prototype of Object.prototype is null, indicating the end of the prototype chain.
- Shadowing Properties: If a property is defined in an object, and the same-named property is in the object's prototype, the object's property shadows the prototype's property. This behavior is known as "shadowing."

Key Concepts

- Setting a Prototype: The Object.create() method allows the creation of a new object with a specified prototype. Constructors in JavaScript automatically set their prototype property as the prototype of newly created objects when called with new. The prototype can be set explicitly using methods like Object.assign().
- Own Properties: Properties defined directly in an object are called own properties. Data properties are often defined in the constructor, while methods are commonly defined in the prototype. Object.hasOwn() can be used to check whether a property is an own property of an object.
- Constructor Prototype: The prototype property of a constructor function is automatically set as the prototype of objects created using that constructor.

```
// Creating an object prototype
const animal = {
    makeSound: function() {
        console.log("Generic animal sound");  }
};
// Creating a new object using the prototype
const cat = Object.create(animal);
cat.makeSound(); // Outputs: "Generic animal sound"
// Modifying the object at runtime
cat.purr = function() {
    console.log("Purrr...");
};
cat.purr(); // Outputs: "Purrr..."
```

Figure: example

Comparison

The choice between the two paradigms depends on the specific requirements of the application. Reactive programming is suitable for systems requiring responsiveness, scalability, and complex event handling, making it well-suited for modern applications with real-time features. Prototype-based programming is beneficial when flexibility, dynamic object manipulation, and differential inheritance are essential, making it a good fit for certain aspects of web development and game development.

Paradigm	Key Concepts	Expression in RxJava (Reactive)	Expression in JavaScript (Prototype-Based)
Reactive	Data Streams, Observables, Operators, Reactivity, Non-Blocking, Backpressure	Observables, Subscribers, Operators, Schedulers, Reactive Extensions (Rx) library	Event emitters, Asynchronous functions, Promises, Async/await, Custom reactive libraries
Prototype-Based	Objects as Prototypes, Prototypal Inheritance, Dynamic Property Addition, No Explicit Classes	Not directly supported, but can be simulated using design patterns or external libraries	Prototype chain, Prototype property (`__proto__`), Object creation patterns (`Object.create()`)

Figure: Reactive - RxJava VS Prototype based - JavaScript

Conclusion

Both reactive programming and prototype-based programming are powerful paradigms with distinct characteristics and applications. Reactive programming, exemplified by RxJava, excels in handling asynchronous data flows, real-time events, and building scalable systems. It emphasizes modularity, compositionality, and a declarative programming style. On the other hand, prototype-based programming provides flexibility in object creation, dynamic modification, and differential inheritance. It is well-suited for scenarios where runtime adaptability and object manipulation are critical.

References

Books:
Reactive Programming with RxJava
Reactive Java Programming
this and Object Prototypes

URLs:
https://gist.github.com/staltz/868e7e9bc2a7b8c1f754
https://github.com/ReactiveX/RxJava
https://www.youtube.com/playlist?list=PL5PLFmjtHpl82bxHZHR_7hjAnNSZdYG9l
https://www.youtube.com/watch?v=583MGxjypgU
https://alistapart.com/article/
prototypal-object-oriented-programming-using-javascript/
https://www.freecodecamp.org/news/
all-you-need-to-know-to-understand-javascripts-prototype-a2bff2d28f03/
https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Objects/Object_
prototypes
https://www.geeksforgeeks.org/prototype-in-javascript/
https://alistapart.com/article/
prototypal-object-oriented-programming-using-javascript/ http:
//javascriptissexy.com/javascript-prototype-in-plain-detailed-language
https://www.turing.com/kb/prototype-vs-class-in-js