Amrita Vishwa Vidyapeetham

TIFAC-CORE in Cyber Security

20CYS312 - Principles of Programming Languages
Assignment-01: Exploring Programming Paradigms

«Anu Priya»

21st January, 2024

Paradigm 1: <Reactive>

Discuss the principles and concepts of Paradigm 1.

The Reactive paradigm offers a way to write software that's resilient, responsive, and scalable, particularly when dealing with asynchronous data flows and real-time events. It takes functional programming a little bit further, by adding the concept of data flows (see the next section) and propagation of data changes.

0.1 Key features:

- Data Streams: At the core of Reactive programming lies the idea of data streams. These are continuous flows of information, like user actions, sensor readings, or stock prices. Unlike traditional data structures, streams are infinite and processed incrementally, element by element.

  Again, what do I mean by "stream of data?"

  I mean a sequence of events, where an event could be user input (like a tap on a button), a response from an API request (like a Facebook feed), data contained in a collection, or even a single variable. In reactive programming, there's often a component that acts as the source, emitting a sequence of items (or a stream of data), and some other components that observe this flow of items and react to each emitted item (they "react" to item emission).

- Declarative Approach: Instead of explicitly instructing the computer on how to handle each step, Reactive programs tell it what needs to be done with the data and its transformations. This declarative style makes code more concise and easier to reason about.

- Event-Driven Processing: Events trigger reactions. Reactive systems continuously listen for events (data changes, user interactions, etc.) and react accordingly. This promotes a responsive and flexible architecture.

- Non-Blocking Operations: Reactive programs avoid blocking calls that stall the thread while waiting for data. Instead, they rely on asynchronous operations and callbacks to ensure smooth execution and responsiveness.

- Backpressure: Imagine pouring water into a cup too fast. Backpressure ensures streams don't overwhelm downstream consumers by regulating the flow of data based on their processing capacity.

- Resiliency: Reactive systems are designed to recover from failures gracefully. Error handling is built-in, and data can be automatically retried or rerouted, preventing cascading failures.

- Compositionality: Small, independent operators are combined to build complex data pipelines. This modularity makes code reusable, maintainable, and easier to test.

0.2 The Observer pattern:

The Observer pattern is a design pattern in which there are two kinds of objects: observers and subjects. An observer is an object that observes the changes of one or more subjects; a subject is an object that keeps a list of its observers and automatically notifies them when it changes its state. This pattern is the core of reactive programming. It fits perfectly the concept of reactive programming by providing the structures to implement the produce/react mechanism.

0.2.1 Observable and Operators:

Observable: (A data source that emits notifications as data changes.) An Observable is an object that emits a sequence (or stream) of events. It represents a push-based collection, which is a collection in which events are pushed when they are created. An observable emits a sequence that can be empty, finite, or infinite. When the sequence is finite, a complete event is emitted after the end of the sequence. At any time during the emission (but not after the end of it) an error event can be emitted, stopping the emission and canceling the emission of the complete event. When the sequence is empty, only the complete event is emitted, without emitting any item. With an infinite sequence, the complete event is never emitted.

An observable begins emitting a sequence of items when the Observer subscribes to it: they are called cold observables. Cold observables always wait to have at least one observer subscribed to start emitting items. On the other hand, an Observable that begins emitting items before being connected to an observer is called a hot observable. With hot observables, an observer can subscribe and start receiving items at any time during the emission. With hot observables, the observer may receive the complete sequence of items starting from the beginning or not. There's another kind of observable called a connectable observable. This kind of observable begins emitting items when its "connect" method is called, whether or not any observers have subscribed to it.

Operator: (A function that transforms or manipulates data streams.)
Reactive Extensions (Rx): A popular library for implementing Reactive programming patterns.

0.3 What can be achieved through Reactive Programming:

Improved responsiveness: Real-time data handling and event-driven processing enable fast and fluid user experiences.

Easier scalability: Stream-based architecture makes it easier to handle large data volumes and distributed systems.

Increased resilience: Failure-handling mechanisms and asynchronous operations improve system robustness. Simplified code: Declarative approach and modularity lead to cleaner and more maintainable code.

0.4 Where can Reactive Programming be used?

Microservices architecture - Asynchronous communication between loosely coupled services benefits from message streams.

User interfaces - Responsive UI frameworks often leverage Reactive principles for efficient data flows. The Reactive paradigm offers a powerful tool for building modern, resilient, and scalable applications.

Multithreading and Concurrency - Reactive programming can simplify handling concurrency and parallelism. By using reactive constructs, developers can express asynchronous and concurrent operations more concisely and manage the complexity of multithreaded code.

0.4.1 Real-time applications and analyzing where:

1. Financial Trading Systems: Process massive amounts of market data and order updates in real-time. React to price fluctuations and execute trades swiftly. Implement risk management strategies with adaptable data flows.
2. IoT and Sensor Networks: Manage sensor data streams from diverse devices and environments. Analyze sensor readings for predictive maintenance or anomaly detection. Orchestrate responses to real-world events based on sensor data.
3. Online Collaboration Tools: Enable seamless real-time collaboration on documents, whiteboards, or design tools. Synchronize changes across multiple users and devices with minimal latency. Provide instant feedback and notifications for efficient collaboration.
4. Media Streaming Platforms: Deliver adaptive video and audio streams based on network conditions and user preferences. Handle varying bandwidth and device capabilities for smooth playback. Enable live streaming with low latency and high availability.
5. Multiplayer Online Games: Coordinate game state and player interactions across distributed servers. Respond to player actions and events in real-time for immersive gameplay. Handle large-scale player populations and complex game mechanics efficiently.
6. Transportation Systems: Manage real-time traffic data, route optimization, and vehicle tracking. Adapt to changing traffic conditions for efficient route guidance. Provide real-time updates to passengers for improved travel experiences.
7. Healthcare Monitoring Systems: Continuously track patient vitals and alert medical professionals to potential issues. Analyze medical data streams for early detection of risks or emergencies. Facilitate remote patient monitoring and telemedicine services.
8. Fraud Detection Systems: Analyze financial transactions in real-time to identify suspicious patterns. React quickly to potential fraud attempts and prevent financial losses. Adapt to evolving fraud techniques with flexible data processing.
9. Social Media Platforms: Deliver real-time updates, notifications, and content feeds. Handle massive user interactions and data flows smoothly. Personalize content and recommendations based on user behavior and interests.
10. Supply Chain Management: Track inventory levels, orders, and shipments in real-time. Optimize logistics and supply chains for efficiency and cost-effectiveness. Respond to demand fluctuations and disruptions with adaptive strategies.

0.4.2 Analysing why is it used for the above applications:

- Network Requests: Fetching data from APIs, handling responses and errors efficiently.
- Database Operations: Asynchronous data retrieval and updates, reactive queries.
- Event-Driven Systems: Handling user inputs, sensor events, or system notifications.
- Parallel Processing: Optimizing multi-core CPUs for parallel tasks and computations.

- Backpressure Control: Handling data streams with varying rates and preventing bottlenecks.
- Error Handling: Graceful recovery from errors and retries without interrupting the application flow.
- Testing: Easy unit testing of asynchronous code with RxJava's test utilities.

0.5 How can Reactive Programming be implemented?

There are various ways to implement Reactive Programming, and different programming languages and libraries may provide specific tools for this paradigm. By keeping all the above key features, observables, operators concepts, Schedulers, and declarative approach in mind, can be thus implemented.

0.5.1 Through Frameworks and Libraries:

Libraries like RxJava, RxJS, and Reactor provide implementations of reactive programming. They offer constructs like Observables, which emit streams of events, and Observers, which react to these events. Operators allow developers to transform, filter, and combine these streams.

Language for Paradigm 1: <RxJava>

Discuss the characteristics and features of the language associated with Paradigm 1.

0.6 Key Characteristics:

- Asynchronous Data Handling: RxJava excels at managing asynchronous operations and events through Observables, ensuring a responsive and non-blocking experience.
- Declarative Programming Style: You express data transformations and operations concisely using a declarative approach, making code more readable and maintainable.
- Backpressure Support: RxJava handles potential data overflow by allowing consumers to regulate the flow of data from producers, preventing bottlenecks and resource exhaustion.
- Operator-Based Composition: You build complex data flows by combining smaller, reusable operators, promoting modularity and testability.
- Rich Operator Toolkit: RxJava offers a vast library of operators (map, filter, reduce, combine, etc.) for transforming, filtering, combining, and otherwise manipulating data streams.
- Thread Management: It provides fine-grained control over thread execution, allowing you to optimize performance and resource utilization in different scenarios.
- Error Handling Mechanisms: RxJava includes robust error handling strategies, such as retrying failed operations, propagating errors gracefully, and implementing fallback mechanisms.

0.7 Key Features:

- Observable: The core data type representing a stream of events or values, enabling asynchronous emission and consumption.
- Observer: A subscriber that receives notifications from an Observable, reacting to data changes or events.
- Subscription: The link between an Observable and Observer, managing the data flow and allowing for cancellation.
- Schedulers: Control the execution of Observables on specific threads or thread pools, optimizing performance and concurrency.
- Hot vs. Cold Observables: Distinction between Observables that emit data even without subscribers (hot) and those that emit only when subscribed to (cold).
- Subjects: Special types of Observables that can both emit and receive values, serving as intermediaries for communication between components.
- Operator Fusion: Optimization technique where multiple operators are combined into a single execution step for efficiency.

0.7.1 Creating Observables:

The easiest way to create an observable is to use the factory methods that are implemented in the RxJava library. You've already seen how to create an Observable using the Observable.from() method, so let's take a look at the other available methods.

- Observable.just(): Observable.just() creates an Observable that emits the object or the objects that are passed in as parameters: Observable.just("an item")
- Observable.range(): Observable.range(a, n) creates an Observable that emits a range of n consecutive integers starting from a. Observable.just(1, 2, 3, 4, 5) and Observable.range(1, 5) will emit the same sequence.
- Observable.interval(): The previous methods created observables that emit items in sequence, one after another, with no delay between items
- Observable.timer(): Observable.timer(long, TimeUnit) creates an Observable that emits just one item after a given delay. It can be useful when combined with other observables to introduce a delay before the beginning of another observable's sequence, as you will see later.
- Observable.create(): Observable.create() is the method that lets you create an Observable from scratch

0.7.2 Composing and Transforming Observables:

Observables are especially good at being composed and transformed. With the usage of some operators defined in the library, you can compose and transform sequences of data in a easy way that requires little coding, so it's less prone to error.

- Map Operator:

  The map operator is fundamental in reactive programming, allowing the transformation of every item in an emitted sequence using a specified function. It creates a new observable that emits the transformed values whenever the original observable emits values.

- Marble Diagrams:

  Marble diagrams are graphical representations of how operators like map, flatMap, concatMap, zip, and others work over time. They illustrate the transformation or combination of data streams using synchronized timelines.

  Observable Creation and Subscription:

  RxJava provides various methods for creating observables, such as Observable.just() and Observable.from().

  Observers subscribe to observables to receive notifications on emitted values, errors, and completion.

- FlatMap Operator:

  The flatMap operator performs both mapping and flattening actions, allowing the conversion of emitted items into observables and then merging them into a single observable. It can result in interleaved observables, and if order matters, concatMap can be used to ensure sequential concatenation.

- ConcatMap Operator:

  The concatMap operator concatenates observables, ensuring that they are not interleaved but concatenated in order. It waits for each observable to complete before subscribing to the next one.

- Zip Operator:

  The zip operator combines multiple observables by applying a specified function to corresponding items. It emits the result of the function in strict sequence, stopping when the shortest sequence completes.

- Concat Operator:

  The concat operator concatenates emissions from multiple observables, preserving the order of each observable. It waits for each observable to complete before moving on to the next one.

- Filter Operator:

  The filter operator allows only specific items from the source sequence to be emitted, based on a specified condition.

- Distinct Operator:

  The distinct operator filters out duplicate items from the source sequence, emitting only the first occurrence of each item.

- First, Last, and Take Operators:

  The first operator emits only the first item of a sequence, optionally filtered by a condition. The last operator emits the last item of a sequence, optionally filtered by a condition. The take operator allows emitting only the first n items from the source sequence.

- StartWith Operator:

  The startWith operator adds a specified item to the beginning of the source sequence.

- Scan Operator:

  The scan operator applies a function to each pair of sequentially emitted items, creating a new observable with the accumulated results.

Operators for Transforming Observables:

- buffer : Periodically gathers items from an Observable into bundles and emits these bundles rather than emitting the items one at a time.

- groupBy : Divides an Observable into a set of observables that each emit a different group of items from the original Observable, organized by key.

- window : Periodically subdivides items from an Observable into Observable windows and emits these windows rather than emitting the items one at a time.

  Operators for Filtering Observables:

- debounce : Only emits an item from an Observable if a particular timespan has passed without it emitting another item.

0.7.3 OBSERVABLES AND OBSERVERS

observers: (A listener that subscribes to an Observable and receives notifications.) Observers subscribe to an Observable and receive notifications when the data in the stream changes. This pattern provides a clean separation between the producer of data (Observable) and the consumers (Observers).

- elementAt: Emits only item n emitted by an Observable.
- ignoreElements: Does not emit any items from an Observable but mirrors its termination notification.
- sample: Emits the most recent item emitted by an Observable within periodic time intervals.
- skip : Suppresses the first n items emitted by an Observable.
- skipLast : Suppresses the last n items emitted by an Observable.
- takeLast : Emits only the last n items emitted by an Observable. Operators: Operators are a key

  feature of reactive programming and RxJava. They allow developers to perform various operations on observables, such as mapping, filtering, combining, or even dealing with errors. RxJava provides a rich set of operators that enable developers to express complex asynchronous operations in a concise and readable manner.

  Operators for Combining Observables:

- And/Then/When : Combines sets of items emitted by two or more observables by means of pattern and plan intermediaries; these operators are not part of the RxJava library but can be found in the RxJavaJoins library. (https://github.com/ReactiveX/ RxJavaJoins)
- combineLatest : When an item is emitted by either of two Observables, it combines the latest item emitted by each Observable via a specified function and emits items based on the results of this function.
- join : Combines items emitted by two Observables whenever an item from one Observable is emitted during a time window defined according to an item emitted by the other Observable.
- merge : Combines multiple Observables into one by merging their emissions.
- switchOnNext : Converts an Observable that emits Observables into a single Observable that emits the items emitted by the mostrecently-emitted of those Observables.

  The working of observables, including error handling, thread management with schedulers, dealing with backpressure, and transformers:

- Error Handling: An observable's emission can end with either a completed event or an error event, but not both. Errors are exceptions thrown during the emission of the sequence, either by the source observable or any operator applied to it.
- The onError() method is called when an error occurs, and it stops the sequence. Exception handling can be done in the onError() method of the subscriber. Operators like onErrorResumeNext(), onErrorReturn(), and onExceptionResumeNext() provide ways to handle errors.

- Retry Mechanism: RxJava provides two operators for implementing a retry mechanism: retry() and retryWhen().
- retry() resubscribes to the source observable without notifying the error event.
- retryWhen() passes the error event to another observable, allowing for more sophisticated retry logic.
- Schedulers subscribeOn() and observeOn() operators control the threads on which observables operate and observers are notified. Common schedulers include Schedulers.immediate(), Schedulers.computation(), Schedulers.io(), Schedulers.newThread(), Schedulers.trampoline(), and Schedulers.from(Executor). Schedulers are useful for handling multithreading, especially in scenarios like network requests and UI updates.

  Advanced Use of Schedulers Scheduler.Worker allows for more fine-grained control over threading operations. Actions can be scheduled to run immediately, after a delay, or periodically using Worker.schedule() and Worker.schedulePeriodically().
- Backpressure Backpressure occurs when the emission rate of items exceeds the consumption rate. Throttling operators like sample, throttleFirst, and debounce control the emission rate. Buffering operators like buffer and window collect items into buffers or windows. Inside the subscriber, the request(n) method can be used to control the number of items emitted by the observable.
- Transformers
- Observable.Transformer allows for the creation of custom operators that can be applied to observables. Transformers are useful for encapsulating and reusing chains of operators.

0.8 Hello world example

```
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 1: hello world example

0.9 Basic Implementation example:

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

Figure 2: example

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

Figure 3: example

```
Result 3
```

Figure 4: example

```
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 5: example

Result for the fig 5 example:
Adding 2 ::4

Adding 3 ::5

Adding 4 ::6

Adding 5 ::7

Subtracting 1 :: 0

0.10 Why to use RxJava for Reactive Programming:

- Improved responsiveness and scalability in handling asynchronous data flows and real-time events.
- Simplified code through declarative style and operator-based composition.
- Enhanced error handling and resilience for robust applications.
- Better testability due to modularity and clear separation of concerns.
- Integration with various Java libraries and frameworks for seamless development.

  Thus, RxJava has become a popular choice for implementing Reactive programming in Java-based applications, offering a powerful and expressive toolkit for building asynchronous, resilient, and scalable systems.

0.11 Case Studies:

- Netflix API: Handles massive streaming data flows with RxJava for efficient and scalable video delivery.
- LinkedIn: Employs RxJava for real-time data updates, search functionality, and asynchronous communication patterns.
- Twitter: Uses RxJava for reactive streams in its backend services, handling high volumes of tweets and user interactions.
- Square: Adopts RxJava for its payment processing systems, ensuring responsiveness and resilience under load.

Paradigm 2: <Prototype-Based>

Discuss the principles and concepts of Paradigm 2.

The Prototype-Based programming paradigm is a style of object-oriented programming (OOP) where objects are not instantiated from classes but rather cloned or copied from existing objects, known as prototypes. In this paradigm, the focus is on the use and manipulation of objects directly, without the need for class definitions.

- Objects as Prototypes: Rather than using classes as blueprints, objects themselves serve as prototypes for creating new objects. Inheritance is achieved through object cloning and extension.
- Runtime Flexibility: The lack of a rigid class hierarchy provides a high degree of flexibility during runtime. Objects can be modified, extended, and composed on the fly without the need for predefined class definitions.

  No Explicit Classes: Prototype-based languages don't have the concept of classes in the traditional sense. Objects are created directly, and inheritance is dynamically established at runtime.
- Prototype Chain: Each object has a link to a prototype object, forming a chain of inheritance. When a property or method is not found on the object itself, the search continues up the prototype chain until a match is found or the end of the chain is reached.

  Object Cloning: Instead of using classes to create new objects, Prototype-Based programming relies on the ability to clone existing objects. The clone operation creates a new object with the same structure and behavior as the prototype
- Dynamic Property and Method Addition: You can add or modify properties and methods to objects at runtime, even after they've been created. This flexibility offers dynamic behavior and late binding.
- Inheritance Through Prototypal Delegation: New objects inherit properties and methods from their prototypes through delegation. When a method is called on an object, it's actually executed on its prototype object, unless overridden in the object itself.

  Differential Inheritance: Prototypes can be used to implement a form of differential inheritance. Objects can inherit some properties from a prototype while providing their own unique properties. This allows for a more granular and selective form of inheritance

Prototype is also important for accessing properties and methods of objects. The prototype attribute (or prototype object) of any object is the "parent" object where the inherited properties were originally defined.This is loosely analogous to the way you might inherit your surname from your father—he is your "prototype parent." If we wanted to find out where your surname came from, we would first check to see if you created it yourself; if not, the search will move to your prototype parent to see if you inherited it from him. If it was not created by him, the search continues to his father (your father's prototype parent). Similarly, if you want to access a property of an object, the search for the property begins directly on the object.

0.12 Principles

:

- Objects as Prototypes: Rather than using classes as blueprints, objects themselves serve as prototypes for creating new objects. Inheritance is achieved through object cloning and extension.
- No Explicit Classes: Prototype-based languages don't have the concept of classes in the traditional sense. Objects are created directly, and inheritance is dynamically established at runtime.

- Prototype Chain: Each object has a link to a prototype object, forming a chain of inheritance. When a property or method is not found on the object itself, the search continues up the prototype chain until a match is found or the end of the chain is reached.
- Dynamic Property and Method Addition: You can add or modify properties and methods to objects at runtime, even after they've been created. This flexibility offers dynamic behavior and late binding.
- Inheritance Through Prototypal Delegation: New objects inherit properties and methods from their prototypes through delegation. When a method is called on an object, it's actually executed on its prototype object, unless overridden in the object itself.

0.12.1 Real world applications and analyze why/how:

Web Development:

- User Interface Frameworks: React, Vue, and Angular extensively use prototypes for component-based architecture, enabling efficient UI development and dynamic updates. Components inherit properties and methods from prototypes, allowing for code reusability, modularity, and flexible component composition.
- jQuery Library: Prototypes power jQuery's plugin system. Plugins extend jQuery's functionality by adding new methods to its prototype, enabling developers to create custom behaviors and interactions.

Game Development:

- Game Object Creation: Prototypes are often used to create game objects with shared properties and behaviors, such as characters, enemies, items, or obstacles. New objects are spawned based on existing prototypes, conserving memory and code duplication.
- Behavior Customization: Prototypes support dynamic behavior modification at runtime, allowing for adaptive game mechanics and player-driven changes.

  For example, enemies might inherit properties from a "base enemy" prototype but have unique abilities or attack patterns.

Server-Side Development:

- Node.js Modules: Node.js's module system utilizes a prototype-based approach for sharing and extending functionality across modules. Modules can inherit properties and methods from other modules, promoting code reusability and modularity.
- Express.js Framework: Express.js's middleware system employs prototypes to create a chain of request handlers, each potentially modifying the request or response. This flexible approach allows for custom request handling logic and dynamic routing.

Other Domains:

- Graphical User Interfaces (GUIs): Prototypes are often used to create GUI widgets and layouts, enabling dynamic UI construction and customization.
- Artificial Intelligence (AI): Prototypes can represent knowledge structures and concepts in AI systems, allowing for adaptive learning and flexible reasoning.
- Natural Language Processing (NLP): Prototypes can model language patterns and semantic relationships, facilitating tasks like text classification, sentiment analysis, and machine translation.

```
var arr = [1,2,3,4];arr.reverse(); // returns [4,3,2,1]
```

Figure 6: This is available default.

```
var arr1 = [1,2,3,4];var arr2 = Array(1,2,3,4);
```

Figure 7: We have created two arrays in two different ways: arr1 with array literals and arr2 with Array constructor function. Both are equivalent to each other with some differences that don't matter

Language for Paradigm 2: <JavaScript>

Discuss the characteristics and features of the language associated with Paradigm 2.

JavaScript is a prototype-based language, therefore understanding the prototype object is one of the most important concepts that JavaScript practitioners need to know. JavaScript provides a powerful environment for Prototype-Based programming with its support for prototypal inheritance, dynamic typing, object cloning, and flexible object creation. These features make JavaScript well-suited for creating dynamic, adaptable, and expressive applications, especially in the context of web development.

In JavaScript, inheritance works a bit differently compared to C++ or Java. JavaScript inheritance is more widely known as "prototypical inheritance".

Things become more difficult to understand when you also encounter 'class' in JavaScript. The new class syntax looks similar to C++ or Java, but in reality, it works differently.

0.12.2 Why prototype?

If we have ever worked with JavaScript arrays or objects or strings, we would have noticed that there are a couple of methods that are available by default. example:

Now coming to the constructor function Array — it is a predefined constructor function in JavaScript. If you open Chrome Developer tools and go to the console and type console.log(Array.prototype) and hit enter you will see something like below: Let's create our own simple constructor function:

var foo = function(name) this.myName = name; this.tellMyName = function() console.log(this.myName);

var fooObj1 = new foo('James'); fooObj1.tellMyName(); This will print James var fooObj2 = new foo('Mike'); fooObj2.tellMyName(); This will print Mike

The method tellMyName is the same for each and every instance of foo. Each time we create an instance of foo the method tellMyName ends up taking space in the system's memory. If tellMyName is the same for all the instances it's better to keep it in a single place and make all our instances refer from that place.

so what's the right way?

var foo = function(name) this.myName = name; foo.prototype.tellMyName = function() console.log(this.myName); var fooObj1 = new foo('James');fooObj1.tellMyName(); This will print James var fooObj2 = new foo('Mike'); fooObj2.tellMyName(); This will print Mike

Thus, We understand the necessity of the prototype.

There are two interrelated concepts with prototype in JavaScript:

First, every JavaScript function has a prototype property (this property is empty by default), and you attach properties and methods on this prototype property when you want to implement inheritance. This prototype property is not enumerable; that is, it isn't accessible in a for/in loop. But Firefox and most versions of Safari and Chrome have a –proto– "pseudo" property (an alternative syntax) that allows you to access an object's prototype property. You will likely never use this –proto– pseudo property, but you should know that it exists and it is simply a way to access an object's prototype property in some browsers.

Figure 8: Here we will see all the methods that we were wondering about. So now we get from where those functions are coming

Figure 9: property of the instances we only have myname. tellMyName is defined under –proto–.Most importantly note that comparing tellMyName of both the instances evaluates to true. Function comparison in JavaScript evaluates true only if their references are the same. This proves that tellMyName is not consuming extra memory for multiple instances.

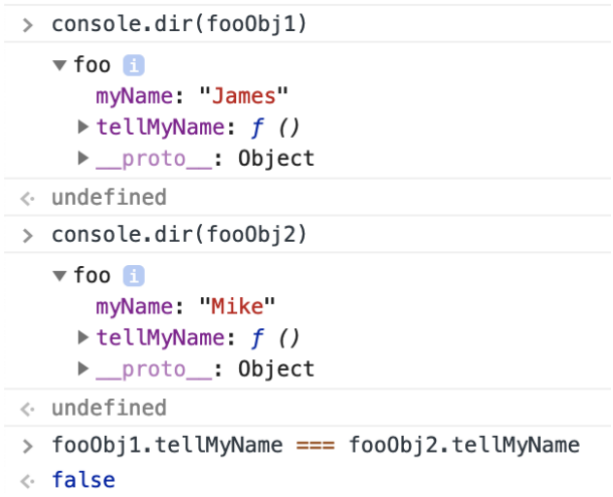

Figure 10: tellMyName is defined as a property of the instances. It's no longer under that –proto–.Also, note that this time comparing the functions evaluates to false. This is because they are at two different memory locations and their references are different.

```
function PrintStuff (myDocuments) {
this.documents = myDocuments;
}

// We add the print () method to PrintStuff prototype property so tha
t other instances (objects) can inherit it:
PrintStuff.prototype.print = function () {
console.log(this.documents);
}

// Create a new object with the PrintStuff () constructor, thus allow
ing this new object to inherit PrintStuff's properties and methods.
var newObj = new PrintStuff ("I am a new Object and I can print.");

// newObj inherited all the properties and methods, including the pri
nt method, from the PrintStuff function. Now newObj can call print di
rectly, even though we never created a print () method on it.
newObj.print (); //I am a new Object and I can print.
```

Figure 11: The output of newObj.print(); will be the string "I am a new Object and I can print." logged to the console.

The prototype property is used primarily for inheritance; you add methods and properties on a function's prototype property to make those methods and properties available to instances of that function.

The second concept with prototype in JavaScript is the prototype attribute. Think of the prototype attribute as a characteristic of the object; this characteristic tells us the object's "parent". In simple terms: An object's prototype attribute points to the object's "parent"—the object it inherited its properties from. The prototype attribute is normally referred to as the prototype object, and it is set automatically when you create a new object.To expound on this: Every object inherits properties from some other object, and it is this other object that is the object's prototype attribute or "parent." (You can think of the prototype attribute as the lineage or the parent). In the example code above, newObj's prototype is PrintStuff.prototype. Note: All objects have attributes just like object properties have attributes. And the object attributes are prototype, class, and extensible attributes. It is this prototype attribute that we are discussing in this second example.

Also, note that the –proto– "pseudo" property contains an object's prototype object (the parent object it inherited its methods and properties from). Constructor: Before we continue, let's briefly examine the constructor. A constructor is a function used for initializing new objects, and you use the new keyword to

```
function Account () {
}
// This is the use of the Account constructor to create the user
Account object.
var userAccount = new Account ();
```

Moreover, all objects that inherit from another object also inherit a *constructor* property. And this constructor property is simply a property (like any variable) that holds or points to the constructor of the object.

```
//The constructor in this example is Object ()
var myObj = new Object ();
// And if you later want to find the myObj constructor:
console.log(myObj.constructor); // Object()

// Another example: Account () is the constructor
var userAccount = new Account ();
// Find the userAccount object's constructor
console.log(userAccount.constructor); // Account()
```

Figure 12: ..

```
// The userAccount object inherits from Object and as such its p
rototype attribute is Object.prototype.
var userAccount = new Object ();

// This demonstrates the use of an object literal to create the
userAccount object; the userAccount object inherits from Object;
therefore, its prototype attribute is Object.prototype just as t
he userAccount object does above.
var userAccount = {name: “Mike”}
```

Figure 13: ..

call the constructor. For example:

Prototype Attribute of Objects Created with new Object () or Object Literal:

All objects created with object literals and with the Object constructor inherits from Object.prototype. Therefore, Object.prototype is the prototype attribute (or the prototype object) of all objects created with new Object () or with . Object.prototype itself does not inherit any methods or properties from any other object.

Prototype Attribute of Objects Created With a Constructor Function Objects created with the new keyword and any constructor other than the Object () constructor, get their prototype from the constructor function.

For Example:

So, there are two general ways an object's prototype attribute is set when an object is created:

If an object is created with an object literal (var newObj =), it inherits properties from Object.prototype and we say its prototype object (or prototype attribute) is Object.prototype. If an object is created from a

```
function Account () {

}
var userAccount = new Account () // userAccount initialized with
the Account () constructor and as such its prototype attribute
(or prototype object) is Account.prototype.
```

Figure 14: Similarly, any array such as var myArray = new Array (), gets its prototype from Array.prototype and it inherits Array.prototype's properties.

constructor function such as new Object (), new Fruit () or new Array () or new Anything (), it inherits from that constructor (Object (), Fruit (), Array (), or Anything ()). For example, with a function such as Fruit (), each time we create a new instance of Fruit (var aFruit = new Fruit ()), the new instance's prototype is assigned the prototype from the Fruit constructor, which is Fruit.prototype.Any object that was created with new Array () will have Array.prototype as its prototype. An object created with new Fruit () will have Fruit.prototype as its prototype. And any object created with the Object constructor (Obj (), such as var anObj = new Object()) inherits from Object.prototype. It is important to know that in ECMAScript 5, you can create objects with an Object.create() method that allows you to set the new object's prototype object. We will cover ECMAScript 5 in a later post.

0.12.3 Why is Prototype Important and When is it Used?

These are two important ways the prototype is used in JavaScript, as we noted above:

Prototype Property: Prototype-based Inheritance Prototype is important in JavaScript because JavaScript does not have classical inheritance based on Classes (as most object oriented languages do), and therefore all inheritance in JavaScript is made possible through the prototype property. JavaScript has a prototype-based inheritance mechanism.Inheritance is a programming paradigm where objects (or Classes in some languages) can inherit properties and methods from other objects (or Classes). In JavaScript, you implement inheritance with the prototype property. For example, you can create a Fruit function (an object, since all functions in JavaScript are objects) and add properties and methods on the Fruit prototype property, and all instances of the Fruit function will inherit all the Fruit's properties and methods.

Demonstration of Inheritance in JavaScript:

function Plant () this.country = "Mexico"; this.isOrganic = true;

// Add the showNameAndColor method to the Plant prototype property Plant.prototype.showNameAndColor = function () console.log("I am a " + this.name + " and my color is " + this.color);

// Add the amIOrganic method to the Plant prototype property Plant.prototype.amIOrganic = function () if (this.isOrganic) console.log("I am organic, Baby!");

function Fruit (fruitName, fruitColor) this.name = fruitName; this.color = fruitColor;

// Set the Fruit's prototype to Plant's constructor, thus inheriting all of Plant.prototype methods and properties. Fruit.prototype = new Plant ();

// Creates a new object, aBanana, with the Fruit constructor var aBanana = new Fruit ("Banana", "Yellow");

// Here, aBanana uses the name property from the aBanana object prototype, which is Fruit.prototype: console.log(aBanana.name); // Banana

// Uses the showNameAndColor method from the Fruit object prototype, which is Plant.prototype. The aBanana object inherits all the properties and methods from both the Plant and Fruit functions. console.log(aBanana.showNameAndColor()); // I am a Banana and my color is yellow.

Note that the showNameAndColor method was inherited by the aBanana object even though it was defined all the way up the prototype chain on the Plant.prototype object.

Indeed, any object that uses the Fruit () constructor will inherit all the Fruit.prototype properties and methods and all the properties and methods from the Fruit's prototype, which is Plant.prototype. This is the principal manner in which inheritance is implemented in JavaScript and the integral role the prototype chain has in the process.

Prototype Attribute: Accessing Properties on Objects

Prototype is also important for accessing properties and methods of objects. The prototype attribute (or prototype object) of any object is the "parent" object where the inherited properties were originally defined.This is loosely analogous to the way you might inherit your surname from your father—he is your "prototype parent." If we wanted to find out where your surname came from, we would first check to see if you created it yourself; if not, the search will move to your prototype parent to see if you inherited it from him. If it was not created by him, the search continues to his father (your father's prototype parent).

Similarly, if you want to access a property of an object, the search for the property begins directly on the object. If the JS runtime can't find the property there, it then looks for the property on the object's prototype—the object it inherited its properties from.

If the property is not found on the object's prototype, the search for the property then moves to prototype of the object's prototype (the father of the object's father—the grandfather). And this continues until there is no more prototype (no more great-grand father; no more lineage to follow). This in essence is the prototype chain: the chain from an object's prototype to its prototype's prototype and onwards. And JavaScript uses this prototype chain to look for properties and methods of an object. If the property does not exist on any of the object's prototype in its prototype chain, then the property does not exist and undefined is returned.

This prototype chain mechanism is essentially the same concept we have discussed above with the prototype-based inheritance, except we are now focusing specifically on how JavaScript accesses object properties and methods via the prototype object.

This example demonstrates the prototype chain of an object's prototype object:

var myFriends = name: "Pete";

// To find the name property below, the search will begin directly on the myFriends object and will immediately find the name property because we defined the property name on the myFriend object. This could be thought of as a prototype chain with one link. console.log(myFriends.name);

// In this example, the search for the toString () method will also begin on the myFriends' object, but because we never created a toString method on the myFriends object, the compiler will then search for it on the myFriends prototype (the object which it inherited its properties from).

// And since all objects created with the object literal inherits from Object.prototype, the toString method will be found on Object.prototype—see important note below for all properties inherited from Object.prototype.

myFriends.toString ();

Object.prototype Properties Inherited by all Objects All objects in JavaScript inherit properties and methods from Object.prototype. These inherited properties and methods are constructor, hasOwnProperty (), isPrototypeOf (), propertyIsEnumerable (), toLocaleString (), toString (), and valueOf (). ECMAScript 5 also adds 4 accessor methods to Object.prototype.

Here is another example of the prototype chain:

function People () this.superstar = "Michael Jackson"; // Define "athlete" property on the People prototype so that "athlete" is accessible by all objects that use the People () constructor.

People.prototype.athlete = "Tiger Woods";

var famousPerson = new People (); famousPerson.superstar = "Steve Jobs";

// The search for superstar will first look for the superstar property on the famousPerson object, and since we defined it there, that is the property that will be used. Because we have overwritten the famousPerson's superstar property with one directly on the famousPerson object, the search will NOT proceed up the prototype chain.

console.log (famousPerson.superstar);

// Steve Jobs

// Note that in ECMAScript 5 you can set a property to read only, and in that case you cannot overwrite it as we just did.

// This will show the property from the famousPerson prototype (People.prototype), since the athlete property was not defined on the famousPerson object itself. console.log (famousPerson.athlete); // Tiger Woods

// In this example, the search proceeds up the prototype chain and find the toString method on Object.prototype, from which the Fruit object inherited—all objects ultimately inherits from Object.prototype as we have noted before.

console.log (famousPerson.toString()); // [object Object] All built-in constructors (Array (), Number (), String (), etc.) were created from the Object constructor, and as such their prototype is Object.prototype.

0.13 Case Studies:

- React: Builds reusable UI components using prototypes, enabling efficient and dynamic UI updates.
- Vue.js: Adopts a similar component-based architecture with prototypes for modular and flexible UI development.
- Angular: Utilizes prototypes for dependency injection and component inheritance, promoting code organization and reusability.
- Node.js: Employs prototypes for its module system and middleware architecture, enabling code sharing and flexible request handling.
- jQuery: Leverages prototypes for its plugin system, allowing developers to extend its functionality with custom behaviors.

Analysis

Provide an analysis of the strengths, weaknesses, and notable features of both paradigms and their associated languages.

- Abstraction Level:

  Reactive Paradigm: Operates at a higher level of abstraction with a focus on data streams and declarative composition.

  Prototype-Based Paradigm: Offers a lower level of abstraction, emphasizing object cloning, composition, and dynamic object manipulation.

- Application Domains:

  Reactive Paradigm: Well-suited for applications requiring real-time updates, event handling, and asynchronous data flows (e.g., web applications with dynamic UIs).

  Prototype-Based Paradigm: Suitable for projects where flexibility, dynamic object creation, and easy code reuse are important (e.g., building UI components in front-end frameworks).

- Learning Curve:

  Reactive Paradigm: May have a steeper learning curve due to the introduction of reactive streams, operators, and backpressure handling.

  Prototype-Based Paradigm: Generally has a more straightforward learning curve, especially for developers familiar with object-oriented concepts.

- Community and Ecosystem:

  Reactive Paradigm: Has a strong community, especially in the context of frameworks like Reactor and RxJava, with a mature ecosystem of libraries and tools.

  Prototype-Based Paradigm: JavaScript has one of the largest and most active communities, and it benefits from a vast ecosystem of libraries and frameworks, including Angular and React.

- Flexibility and Adaptability:

  Reactive Paradigm: Provides flexibility in handling asynchronous operations and reacting to changes in data streams.

  Prototype-Based Paradigm: Offers flexibility through dynamic object creation, composition, and modification, allowing for adaptable code structures.

0.14 Strengths:

0.14.1 Reactive-RxJava

- Excels in handling asynchronous operations and real-time data.
- Builds scalable, resilient systems that handle concurrency well.
- Simplifies complex event-driven architectures.
- Well-suited for:

  User interfaces with real-time updates

  Data processing pipelines

  Distributed systems

  Server-side applications handling high-concurrency.

0.14.2 Prototype-based JavaScript:

- Offers flexibility in object creation and modification.
- Enables runtime adaptability and customization.
- Promotes code reuse and efficient memory usage.
- Well-suited for:

  Dynamic web UI frameworks

  Game development with adaptive behaviors

  Data modeling with evolving structures

  Rapid prototyping and experimentation

0.15 Weaknesses:

0.15.1 Reactive-RxJava

- Learning Curve:

  The asynchronous and declarative nature of reactive programming can be unfamiliar for developers used to traditional imperative programming styles.

  Concepts like Observables, Operators, and Backpressure require a new way of thinking about data flow and event handling.

  Mastering the RxJava library, with its vast number of operators, can be a significant learning investment.

- Potential Overhead:

  Reactive libraries like RxJava introduce additional runtime overhead compared to traditional approaches.

  Creating and managing Observables, especially for tasks that don't inherently benefit from streams, can add to memory consumption and execution time.

  Developers need to be mindful of resource utilization and choose appropriate operators to avoid performance bottlenecks.

- Debugging Complexity:

  Tracking down issues in asynchronous and event-driven code can be more challenging than debugging traditional synchronous code.

  Traditional debuggers might not provide intuitive insights into the flow of data through streams and operators.

  Understanding the chain of events leading to an error requires careful analysis of emitted values, operators applied, and timing considerations.

- Strategies to Address Weaknesses:

  Gradual adoption: Start with simple reactive patterns and gradually build complexity as confidence grows.

  Leverage learning resources: Utilize tutorials, documentation, and community support to understand key concepts and best practices.

  Focus on performance: Choose appropriate operators and optimize code to minimize resource overhead.

  Utilize debugging tools: Employ specialized reactive debugging tools and techniques to trace data flow and identify issues.

  Start small and test thoroughly: Implement reactive components in isolated sections of the codebase and test them thoroughly before widespread adoption.

0.15.2 Prototype-based JavaScript:

- Prototypal Complexity:

  Understanding the Prototype Chain: Grasping how objects inherit properties and methods through the prototype chain can be initially challenging, especially for developers accustomed to class-based inheritance.

  Managing Dependencies: Ensuring proper object initialization and avoiding circular references in large codebases with complex prototype relationships requires careful attention to dependency management.

  Debugging Challenges: Tracing errors or unexpected behavior can be more intricate when inheritance and property lookups involve multiple prototypes.

- Class Syntax Confusion:

  Syntactic Sugar vs. Underlying Mechanism: The class keyword in JavaScript might create a misleading impression of class-based inheritance. However, it's merely syntactic sugar that still relies on prototypal inheritance under the hood.

  Misconceptions: Developers unfamiliar with prototypal inheritance might make incorrect assumptions about object behavior based on class-based expectations, leading to potential errors or misunderstandings.

- Historical Baggage:

  Quirks and Inconsistencies: JavaScript's evolution has resulted in some quirks and inconsistencies that can surprise developers, such as this binding rules, variable scoping, and type coercion.

  Different Mindset: Prototypal inheritance requires a distinct mental model compared to class-based inheritance, potentially demanding a shift in thinking for developers from other language backgrounds.

Strategies to Address These Challenges:

- Solid Understanding of Prototypes: Invest time in learning prototypal inheritance's mechanics, including the prototype chain, object creation patterns, and inheritance patterns.
- Effective Code Organization: Structure code to minimize complexity and make dependencies clear, using design patterns like modules or mixins to organize prototypes effectively.
- Mindful Use of class Syntax: Understand its implications and limitations, avoiding assumptions based on class-based expectations.
- Thorough Testing: Implement comprehensive testing to catch errors early and ensure code behaves as intended.
- Leverage Community Resources: Utilize the vast JavaScript community's knowledge, tutorials, and best practices to guide development.
- Consider Linting and Code Analysis Tools: These tools can help detect potential issues and enforce coding standards, reducing the impact of JavaScript's quirks.

0.16 Key Observations:

Reactive Paradigm: Excels in handling asynchronous data flows, real-time events, and building scalable and responsive systems. Prototype-Based Paradigm: Offers flexibility, dynamic behavior, and code reusability, often used in UI frameworks and modular architectures. Language Choice: The most suitable paradigm depends on the specific project requirements and developer preferences. Hybrid Approaches: Some projects effectively combine both paradigms, leveraging their strengths for different aspects of the application.

0.17 How to choose the Right Paradigm:

- Consider the nature of your application:

  Reactive excels for real-time, data-intensive, and event-driven systems.

  Prototype-based often shines for UI development, dynamic behavior, and flexible code structures.
- Evaluate developer expertise and team preferences.
- Explore potential performance implications and trade-offs.
- Consider the available libraries and frameworks supporting each paradigm in the chosen language.

Comparison

Compare and contrast the two paradigms and languages, highlighting similarities and differences.

As we saw in 0.14 subsection, 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.
Expression in RxJava:
Observable Creation:

In RxJava, observables can be created from various sources, such as collections, arrays, or by chaining operators on existing observables.

Observable<Integer> numbersObservable = Observable.fromArray(1, 2, 3, 4, 5);

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 15: Reactive - RxJava VS Prototype based - JavaScript

Operators:

RxJava provides a rich set of operators for transforming and manipulating data streams.

numbersObservable .map(number -> number * 2) .filter(result -> result > 5) .subscribe(result -> System.out.println(result));

Subscription:

Subscribing to an observable allows observers to receive and react to emitted values.

Disposable disposable = numbersObservable.subscribe(result -> System.out.println(result), error -> System.err.println(error), () -> System.out.println("Completed"));

Prototype-Based Paradigm with JavaScript: Expression in JavaScript:
Object Creation:

Objects in JavaScript can be created using literal notation or by using the Object.create() method.

const person = name: 'John', age: 30, sayHello: function() console.log('Hello, my name is ' + this.name); ;

Object Cloning:

Objects can be cloned using the Object.create() method, creating a new object with the specified prototype.
const employee = Object.create(person); employee.jobTitle = 'Software Developer'; Object Composition:

Objects can be composed by combining properties and methods from multiple prototypes.

const student = school: 'University XYZ', study: function() console.log(this.name + ' is studying.'); ;
const newPerson = Object.assign(, person, student);

Challenges Faced

Discuss any challenges you encountered during the exploration of programming paradigms and how you addressed them.

Understanding abstract concepts of Reactive Programming and the prototype-based was a bit difficult in-between to grasp the actual meaning.

I referred to many related materials that had concepts fragmented into smaller parts, practiced with examples, and related them with their real-world applications for my understanding. Once I saw the code snippets and examples, it gave more clarity on the concepts.

Conclusion

Summarize your findings and conclude the assignment.

In essence, reactive programming and prototype-based programming offer distinct approaches to software development, each excelling in specific domains:

- Reactive programming shines when dealing with the continuous flow of data and events, crafting responsive and adaptable systems that gracefully handle change and concurrency. It's ideal for building applications that thrive on real-time actions and continuous updates.
- Prototype-based programming empowers developers with unparalleled flexibility in object creation and modification, enabling systems to adapt and evolve organically at runtime. It's perfectly suited for scenarios where customization and responsiveness to runtime conditions are paramount.

  The choice between these paradigms hinges on the specific needs of your application:

- Embrace reactive programming for asynchronous data streams, scalable architectures, and declarative elegance.
- Opt for prototype-based programming when dynamic object structures, runtime adaptability, and flexible inheritance are crucial.

  Ultimately, both paradigms offer valuable tools for modern software development, empowering developers to craft well-structured, responsive, and maintainable applications.

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-a2bff2d28
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

For self-learning and basic understanding: chatgpt and YouTube were used https://www.youtube.com/playlist?list=PL5PLFmjtHpl82bxHZHR_7hjAnNSZdYG9l