20CYS312 - Principles of Programming Languages
Exploring Programming Paradigms

Assignment-01

Presented by Deepthi J
CB.EN.U4CYS21014
TIFAC-CORE in Cyber Security
Amrita Vishwa Vidyapeetham, Coimbatore Campus

Feb 2024

AMRITA
VISHWA VIDYAPEETHAM

Outline

Introduction to Functional Paradigm

1. Functional programming (FP) is a programming paradigm where composing functions becomes the main driving force behind the development. It is a declarative type of programming style that focuses on "what to solve" rather than "how to solve". FP is an approach to software development that uses pure functions to create maintainable software.
2. Functional programming languages focus on declarations and expressions rather than the execution of statements. Functions are also treated like first-class citizens—meaning they can pass as arguments, return from other functions, and attach to names.
3. FP also uses immutable data and avoids concepts like shared states. This is in contrast to object-oriented programming (OOP), which uses mutable data and shared states.

Importance of Functional Paradigm

1. Functional programming is notable for its ability to efficiently parallelize pure functions. Code for data analysis workflows and tasks is easier to analyze, test, and maintain using the functional programming paradigm.
2. Due to its pure nature, FP is ideally suited for analyzing extensive data sets and machine learning. Pure functions will always generate the same results, with no outside values to influence the final results i.e., with no side effects.
3. Algorithms created using FP can also quickly identify and correct errors.

The 7 Core Functional Programming Concepts

- **Pure functions:** Pure functions form the foundation of functional programming and have two major properties:
  - They produce the same output if the given input is the same
  - They have no side effects i.e. they do not modify any arguments or local/global variables or input/output streams.

  The pure function's only result is the value it returns. They are deterministic. Because pure functions have no side effects or hidden I/O, programs built using functional paradigms are easy to debug. Moreover, pure functions make writing concurrent applications easier.
  When the code is written using the functional programming style, a capable compiler can:
  - Memorize the results
  - Parallelize the instructions
  - Wait for evaluating results

The 7 Core Functional Programming Concepts Contd..

example of the pure function:

```
sum(x, y)              // sum function taking x and y as arguments
return x + y     // sum returning sum of x and y without changing them
```

- **Recursion**: In the functional programming paradigm, there are no for and while loops. Instead, these languages rely on recursion for iteration. Recursion is implemented using recursive functions, which call themselves repeatedly until the base case is reached.
- **Immutability**: In functional programming, we can't modify a variable after being created. We create a variable and assign a value, we can run the program with ease fully knowing that the value of the variables will remain constant and can never change.

The 7 Core Functional Programming Concepts Contd..

- **First-class functions and High order functions**: First-class functions in functional programming are treated as data type variables and can be used like any other variables. These first-class variables can be passed to functions as parameters, or stored in data structures.
  A function that accepts other functions as parameters or returns functions as outputs is called a high-order function. This process applies a function to its parameters at each iteration while returning a new function that accepts the next parameter.
- **Lazy evaluation**: Lazy evaluation is an evaluation strategy that holds the evaluation of an expression until its value is needed.

Immutable Data Structures: Clojure encourages the use of immutable data structures. Once created, data structures cannot be modified, and any operation that seems to modify them returns a new structure.

```
;; Creating an immutable vector
(def my-vector [1 2 3])

;; Adding an element creates a new vector
(def new-vector (conj my-vector 4))

;; my-vector is still unchanged
my-vector ; Returns [1 2 3]
```

Clojure Contd..

First-Class Functions: Functions are first-class citizens in Clojure, meaning they can be assigned to variables, passed as arguments, and returned as values.

```
;; Defining a function
(defn square [x] (* x x))

;; Assigning a function to a variable
(def my-func square)

;; Using the function as an argument
(def result (my-func 5)) ; Returns 25
```

Clojure Contd..

Higher-Order Functions: Clojure supports higher-order functions, allowing functions to take other functions as arguments or return functions as results.

```clojure
;; Higher-order function
(defn apply-twice [fn x] (fn (fn x)))

;; Using the higher-order function
(def square-twice (apply-twice square))
(square-twice 3) ; Returns 81
```

Clojure Contd..

Lazy Sequences: Clojure supports lazy sequences, allowing efficient handling of large data sets by computing values only as needed.

```
;; Lazy sequence example
(defn infinite-sequence [] (iterate inc 0))

;; Using the lazy sequence
(take 5 (infinite-sequence)) ; Returns (0 1 2 3 4)
```

Event-Driven Paradigm

Event-driven programming is a programming paradigm where the flow of a program is determined by events that occur during its execution. Instead of following a linear sequence of operations, an event-driven program waits for specific events to occur and then triggers corresponding event handlers or callbacks to respond to those events.

In event-driven programming, events can be various types of signals, actions, or occurrences, such as user interactions (e.g., button clicks, mouse movements, key presses), system events (e.g., timers, file input/output, network communication), or custom events generated within the program.

Key concepts of event-driven programming

- **Event**: An event is a signal or notification that something has happened. Events can be triggered by user actions, system actions, or other parts of the program.
- **Event Handler (or Callback)**: An event handler is a piece of code that is executed in response to a specific event. When an event occurs, the associated event handler is invoked to handle that event.
- **Event Loop**: The event loop is a core component of event-driven programming. It continuously checks for new events in the program's event queue and processes them in the order they occur. When an event is detected, the corresponding event handler is called.
- **Asynchrony**: Event-driven programming often involves asynchronous operations. Instead of blocking the program's execution while waiting for an event to occur, it can continue processing other events or tasks until the event is ready to be handled.

Event-Driven Contd..

Event-driven programming is commonly used in graphical user interfaces (GUIs), web development (e.g., handling HTTP requests), networking (e.g., handling incoming data from sockets), and many other real-time or event-based applications.

The events are dealt with by a central event-handler (usually called a dispatcher or scheduler) that runs continuously in the background and waits for an event to occur. When an event does occur, the scheduler must determine the type of event and call the appropriate event-handler to deal with it. The information passed to the event handler by the scheduler will vary, but will include sufficient information to allow the event-handler to take any action necessary.

Event-Driven Contd..

The diagram below illustrates the relationship between events, the scheduler, and the application's event-handlers.

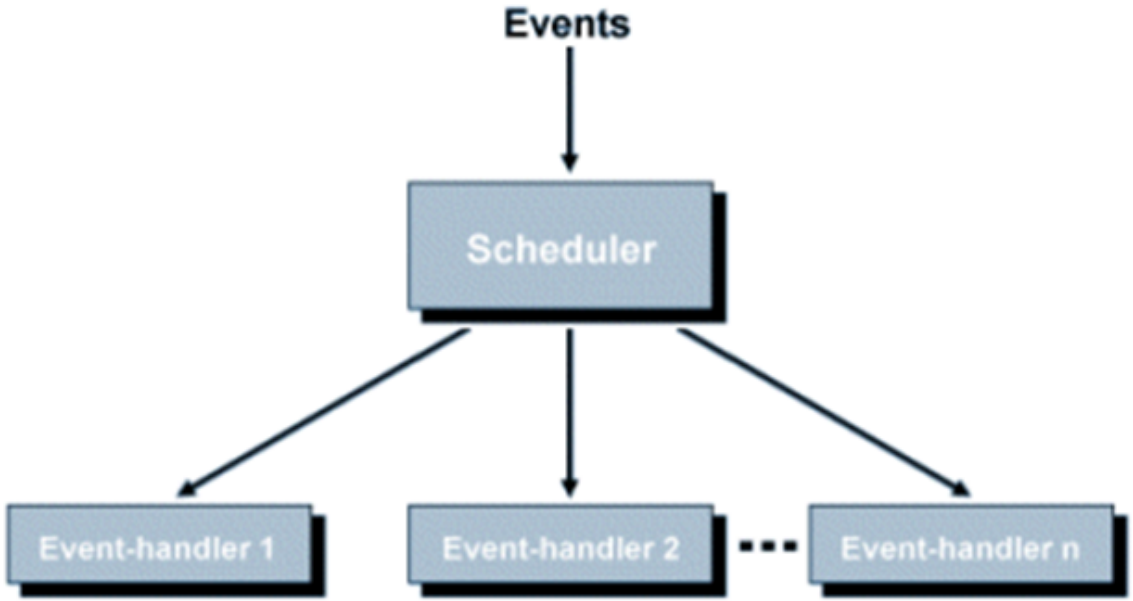

Figure: A simple event-driven programming paradigm

Similarities - Event Driven and Functional Paradigm

1. **Modularity**: Both paradigms promote modularity by encouraging the organization of code into smaller, reusable components. In event-driven programming, modules often correspond to event handlers, while in functional programming, modularity is achieved through the composition of pure functions.
2. **Composability**: Composability is a shared characteristic. In functional programming, functions can be composed to create more complex operations. Similarly, in event-driven programming, event handlers can be composed to achieve desired behavior in response to multiple events.
3. **Asynchronous Programming**: Both paradigms can involve asynchronous programming. In event-driven programming, asynchronous handling of events is common. In functional programming, asynchronous operations can be managed using concepts like Promises or Monads.
4. **State Management**: Both paradigms address state management. While they approach it differently, they both recognize the importance of handling and managing the state of a system. Event-driven programming often involves state changes in response to events, and functional programming promotes immutability to manage state changes.

Differences - Event Driven and Functional Paradigm

1. **State Mutation**: State mutation is handled differently. In event-driven programming, state changes often occur in response to events, and mutable state is common. Functional programming, on the other hand, promotes immutability, minimizing state changes and emphasizing the creation of new states.
2. **Concurrency and Parallelism**: The paradigms differ in their approach to concurrency and parallelism. Functional programming, with its emphasis on immutability and lack of side effects, is well-suited for concurrent and parallel execution. Event-driven programming, while inherently asynchronous, might require additional mechanisms for managing concurrency.
3. **Side Effects**: Side effects are managed differently. Functional programming aims to minimize or eliminate side effects, ensuring that functions have no observable impact beyond producing a result. In event-driven programming, side effects are often inherent, especially when handling events that may lead to changes in the system state.
4. **Data Flow**: The data flow is organized differently. In functional programming, data typically flows through a series of pure functions, each transforming the data. In event-driven programming, the flow is driven by the occurrence of events, and d is often associated with those events.

References

1. https://www.techtarget.com/searchitoperations/definition/declarative-programming
2. https://hackr.io/blog/functional-programming
3. https://www.turing.com/kb/introduction-to-functional-programming
4. https://www.infoworld.com/article/3613715/what-is-functional-programming-a-practical-guide.html
5. https://www.codingdojo.com/blog/what-is-functional-programming
6. https://www.studysmarter.co.uk/explanations/computer-science/computer-programming/event-driven-programming/