Amrita Vishwa Vidyapeetham

TIFAC-CORE in Cyber Security

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

Venkata Revan Nagireddy

21st January, 2024

## Paradigm 1: Fuctional

Functional programming is a declarative programming paradigm style where one applies pure functions in sequence to solve complex problems. Functions take an input value and produce an output value without being affected by the program. Functional programming mainly focuses on what to solve and uses expressions instead of statements. Functional programming excels mostly at mathematical functions where the values don't have any correlation and doesn't make use of concepts like shared state and mutable data used in object-oriented programming. Functional programming tries to keep data and behavior separate.

**Core FP Concepts:**

1. **Pure Functions** - only input in only output out
   Purely functional programming is a subset of Functional Programming that treats all functions as deterministic mathematical functions. In deterministic mathematical functions, the development of future states of the system does not allow any randomness.
   Pure functions, meanwhile have function return values that are identical for identical arguments. The function application also has no side effects (i.e., it does not modify state variable values outside its local environment).
   A side effect occurs in a program when you insert external code into your function. This prevents the function from properly performing its task.

2. **First-Class 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.

3. **Recursion**
   Unlike object-oriented programming, functional programming doesn't make use of "while" or "for" loops or "if-else" statements. Functional programs avoid constructions that create different outputs on every execution. Instead, recursive functions call themselves repeatedly until they reach the desired state or solution known as the base case.

4. **Immutability**
   In functional programming, we can't modify a variable after being created. The reason for this is that we would want to maintain the program's state throughout the runtime of the program. It is best practice to program each function to produce the same result irrespective of the program's state. This means that when 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.

5. **High order function**
   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.

**Merits:**

- Easy to debug - It is also simpler and faster to examine code for faults since there are no modifications or additional hidden outputs generated because pure functions yield the same output as the input.
- Easy to read - Functions in functional programming are easy to read and understand. Since functions are treated as values, immutable, and can be passed as parameters, it is easier to understand the codebase and purpose.
- Lazy evaluation - Functional programming adopts the lazy evaluation concept, whereby the computations are only evaluated the moment they are needed. This gives programs the ability to reuse results produced from previous computations.
- Supports parallel programming - Functional programming uses immutable variables, creating parallel programs is easy as they reduce the amount of change within the program. Each function only has to deal with an input value and have the guarantee that the program state will remain constant.
- Efficient - Functional programs don't rely on any external sources or variables to function, they are easily reusable across the program. This makes them more efficient as there isn't extra computation needed to source the programs or run operations on runtime.

**Demerits:**

- Terminology Problems - Because of its mathematical roots, functional programming has a lot of terminologies that may be difficult to explain to a layman. Terms like "pure functions" can easily scare off people looking to learn more about functional programming.
- Recursion - Although recursion is one of the best features in functional programming, it is very expensive to use. Writing recursive functions requires higher memory usage which can be costly.

**Best Practices**

- Function should accept atleast one argument.
- Function should return data, or another function.
- Don't use loops.

**Programming Languages** - Languages that support Functional Programming
The foundation for Functional Programming is Lambda Calculus. It was developed in the 1930's for functional application and recursion.
**LISP** was the first functional programming language designed in 1960.
In the year 2004 Innovation of Funcional language **'Scala'**.
Languages made for Functional Programming:

- Haskell
- Ocaml
- F
- Clojure
- Wolfram Language

Not designed for, but offering strong support for Functional Programming:

- C++
- Perl
- Python

## Language for Paradigm 1: Clojure

### Why Clojure?

Clojure is a programming language that lives up to that standard. Forged of a unique blend of the best features of a number of different programming languages including various Lisp implementations, Ruby, Python, Java, Haskell, and others. Clojure provides a set of capabilities suited to address many of the most frustrating problems programmers struggle with today and those we can see barreling toward us over the horizon.

- Clojure is hosted on the Java Virtual Machine (JVM)
  Clojure code can use any Java library, Clojure libraries can in turn be used from Java, and Clojure applications can be packaged just like any Java application and deployed anywhere other Java applications can be deployed: to web application servers; to desktops with Swing, SWT, or command-line interfaces; and so on. This also means that Clojure's runtime is Java's runtime, one of the most efficient and operationally reliable in the world.

- Clojure is a functional programming language
  Clojure encourages the use of first-class and higher-order functions with values and comes with its own set of efficient immutable data structures. The focus on a strong flavor of functional programming encourages the elimination of common bugs and faults due to the use of unconstrained mutable state and enables Clojure's solutions for concurrency and parallelization.

- Clojure is a dynamic programming language
  Clojure is dynamically and strongly typed yet function calls are compiled down to (fast!) Java method invocations. Clojure is also dynamic in the sense that it deeply supports updating and loading new code at runtime, either locally or remotely. This is particularly useful for enabling interactive development and debugging or even instrumenting and patching remote applications without downtime.

- Clojure offers innovative solutions to the challenges inherent in concurrency and parallelization.

### History of Closure

Clojure is a dialect of Lisp. Lisp is best understood as a family of languages. This language family is ancient, one of the oldest languages still in use today. The fact that it is still in use is a testimony to its expressive power and its ability to be modified to the will of the programmer. Today there are many dialects of Lisp in use, some quite new. These include Common Lisp, Scheme, Racket, Guile, and of course, Clojure.
In 1958, John McCarthy created the Lisp programming language at MIT. It was meant to be a theoretical exercise, but then one of his graduate students converted it to assembly language, and thus the first Lisp interpreter came into being.

In 1962, the first Lisp compiler was written in Lisp. A language is usually considered a "toy" language until it can compile itself.

In spite of this, people continued to work with Lisp, writing production code and creating new dialects. We will ignore them all except for the one that concerns us now. In 2009, Rich Hickey released Clojure publicly. Because it could run on the JVM and have access to all of Java's libraries, it generated a lot of excitement, and is quickly finding its way into industry. You will definitely want a Clojure-aware editor. If you want to get started quickly, the IDE called LightTable is very promising. It's a simple .jar file, and runs almost everywhere.

### Getting Clojure

The very short version is "get Leinengen". Leinengen, or lein as the command is called, is a Clojure build tool. It can run Clojure programs, but it can also manage Clojure packages very easily.

The modular symbol lein repl is the Unix prompt and command to start Clojure, and the user=> symbol is the Clojure prompt. Your own prompts may look different depending on how your environment is setup. The extension for clojure program files is **.clj**

```
% lein repl
nREPL server started on port 43244
REPL-y 0.2.0
Clojure~1.5.1
    Docs: (doc function-name-here)
          (find-doc "part-of-name-here")
  Source: (source function-name-here)

user=> (load-file "foo.clj")
#'user/plus
user=> (plus 10 20)
30
```

Figure 1: Running the program from the interactive environment will look like this using lein

```
PS C:\Users\revan\Downloads> clojure -M .\hello.clj
Hello world
```

Figure 2: Compiling using clojure in windows

## Fundamentals of Clojure

Clojure contains many numeric types. The usual ones are there, such as integers and floats. They look like you would expect. The semicolon is the comment character in Clojure.

### Variable Naming Rules

- Variables are typically named using lowercase letters and hyphens for multi-word identifiers.
- Underscores and hyphens are common in variable names.
- Clojure encourages immutability, so variables are not typically reassigned.
- Clojure is dynamically typed, meaning you don't explicitly declare the type of a variable. Types are inferred at runtime.
- Use namespaces to avoid naming conflicts.

```
(def my-int 42)
(def my-float 3.14)
```

Figure 3: Defining variable name and assigning values

**Function**

In Clojure, every computation begins with a parenthesis. This is the most noticeable feature of the language, and perhaps the most important. In non-lisps, a function call might look like **f(x,y)**. In Clojure, it looks like **(f x y)**.
So, to add two numbers you would write **(+ 10 20)**.
To add 3*3 to 4*4 you would write **(+ (* 3 3) (* 4 4))**.

There are a few strengths to this setup. First, precedence is explicit. You all know the algebraic rules governing times and plus, but there are many other operators. Second, this allows functions to take a variable

number of arguments. For example, you can say (+ 2 4 8) to get 14, and you can say (< 10 x 20) to check if variable x is between 10 and 20.

```
(+ 10 20 30) ;     % 60
(- 50 20 1) ;      % 29
(* 8 6 7 5 3 9) ; % 45360
(mod 10 4) ; =>    % 2
```

Figure 4: example code

**Mathematics**

Clojure uses Java's Math library to handle machine math. There is a separate library if you want to use such functions on extended precision numbers.

```
Math/E ;          => 2.718281828459045
(Math/sqrt 2) ; => 1.4142135623730951
(Math/exp 34) ; => 5.8346174252745 49E14
Math/PI ;         => 3.141592653589793
```

Figure 5: Math Library

## Creating and Using Variables

**Define**

There are two common ways to create variables in Clojure. The first is with def. The syntax is simple: **(def var exp)** defines a variable named var and assigns to it the value given by expression exp.

Note: Variables created by def persist throughout their scope.

```
(def a 10)
(def b (+ 15 5))
(+ a b) ; => 30
```

Figure 6: Defining variables using def

**Let**

The second way to create variables is with the let form. The syntax is **(let [v1 e1 ...] body)**. Unlike def, the**variables created by let are temporary and local**. They exist only in the body part of the let, and then disappear. You can define more than one variable at a time by adding more pairs.

Variables are defined by let one at a time. Subsequent definitions have access to previous definitions. Similarly as let* as used in other Lisp dialects.

## Function Programming

Clojure is a functional programming language. It provides the tools to avoid mutable state, provides functions as first-class objects, and emphasizes recursive iteration instead of side-effect based looping. Clojure is

```
(def a 10)
(def b 20)
(let [a 50] a) ; => 50
 a ; => 10
(let [a 20
      b 40]
(+ a b)) ; => 60
(+ a b)  ; => 30
```

Figure 7: Defining Variable using let

impure, in that it doesn't force your program to be referentially transparent, and doesn't strive for 'provable' programs. The philosophy behind Clojure is that most parts of most programs should be functional, and that programs that are more functional are more robust.

**First-class Functions**

- *fn* creates a function object. It yields a value like any other - you can store it in a var, pass it to functions etc.

```
(def hello (fn [] "Hello world"))
(println (hello)) ;
```

Figure 8: Function object created using fn

- *defn* is a macro that makes defining functions a little simpler. Clojure supports arity overloading in a single function object, self-reference, and variable-arity functions using

```
(defn argcount
  ([] 0)
  ([x] 1)
  ([x y] 2)
  ([x y & more] (+ (argcount x y) (count more))))

(println (argcount))
(println (argcount 1))
(println (argcount 1 2))
(println (argcount 1 2 3 4 5))
```

Figure 9: Defining a function using defn

**Immutable Data Structures**

The easiest way to avoid mutating state is to use immutable data structures. Clojure provides a set of immutable lists, vectors, sets and maps. Since they can't be changed, 'adding' or 'removing' something from


Figure 10: Output for figure 10

an immutable collection means creating a new collection just like the old one but with the needed change. Persistence is a term used to describe the property wherein the old version of the collection is still available after the 'change', and that the collection maintains its performance guarantees for most operations. Specifically, this means that the new version can't be created using a full copy, since that would require linear time. Inevitably, persistent collections are implemented using linked data structures, so that the new versions can share structure with the prior version. Singly-linked lists and trees are the basic functional data structures, to which Clojure adds a hash map, set and vector both based upon array mapped hash tries.

```
(let [my-vector [1 2 3 4]
      my-map {:fred "ethel"}
      my-list (list 4 3 2 1)]
(println (list
  (conj my-vector 5)
  (assoc my-map :ricky "lucy")
  (conj my-list 5)
  my-vector
  my-map
  my-list)))
```

Figure 11: The collections have readable representations and common interfaces

Applications often need to associate attributes and other data about data that is orthogonal to the logical value of the data. Clojure provides direct support for this metadata. Symbols, and all of the collections, support a metadata map. It can be accessed with the meta function. Metadata does not impact equality semantics, nor will metadata be seen in operations on the value of a collection. Metadata can be read, and can be printed.

**Extensible Abstractions**

Clojure uses Java interfaces to define its core data structures. This allows for extensions of Clojure to new concrete implementations of these interfaces, and the library functions will work with these extensions. This is a big improvement vs. hardwiring a language to the concrete implementations of its data types.
A good example of this is the seq interface. By making the core Lisp list construct into an abstraction, a wealth of library functions are extended to any data structure that can provide a sequential interface to its contents. All of the Clojure data structures can provide seqs. Seqs can be used like iterators or generators in other languages, with the significant advantage that seqs are immutable and persistent. Seqs are extremely simple, providing a first function, which return the first item in the sequence, and a rest function which returns the rest of the sequence, which is itself either a seq or nil.
You can define your own lazy seq-producing functions using the lazy-seq macro, which takes a body of expressions that will be called on demand to produce a list of 0 or more items.

**Recursive Looping**

In the absence of mutable local variables, looping and iteration must take a different form than in

```
% cycle produces an 'infinite' seq!

(take 15 (cycle [1 2 3 4]))

% output -> (1 2 3 4 1 2 3 4 1 2 3 4 1 2 3)
```

Figure 12: Clojure library functions produce and consume seqs lazily

```
(defn take [n coll]
  (lazy-seq
    (when (pos? n)
      (when-let [s (seq coll)]
       (cons (first s) (take (dec n) (rest s)))))))
```

Figure 13: Lazy-seq

languages with built-in for or while constructs that are controlled by changing state. In functional languages looping and iteration are replaced/implemented via recursive function calls. Many such languages guarantee that function calls made in tail position do not consume stack space, and thus recursive loops utilize constant space. Since Clojure uses the Java calling conventions, it cannot, and does not, make the same tail call optimization guarantees. Instead, it provides the recur special operator, which does constant-space recursive looping by rebinding and jumping to the nearest enclosing loop or function frame. While not as general as tail-call-optimization, it allows most of the same elegant constructs, and offers the advantage of checking that calls to recur can only happen in a tail position.

```
(defn my-zipmap [keys vals]
  (loop [my-map {}
         my-keys (seq keys)
         my-vals (seq vals)]
    (if (and my-keys my-vals)
      (recur (assoc my-map (first my-keys) (first my-vals))
             (next my-keys)
             (next my-vals))
      my-map)))
(my-zipmap [:a :b :c] [1 2 3])

% output -> {:b 2, :c 3, :a 1}
```

Figure 14: Recursion

```
(import '(javax.swing JFrame JLabel JTextField JButton)
        '(java.awt.event ActionListener)
        '(java.awt GridLayout))
(defn celsius []
  (let [frame (JFrame. "Celsius Converter")
        temp-text (JTextField.)
        celsius-label (JLabel. "Celsius")
        convert-button (JButton. "Convert")
        fahrenheit-label (JLabel. "Fahrenheit")]
    (.addActionListener
     convert-button
     (reify ActionListener
            (actionPerformed
             [_ evt]
             (let [c (Double/parseDouble (.getText temp-text))]
               (.setText fahrenheit-label
                         (str (+ 32 (* 1.8 c)) " Fahrenheit"))))))
    (doto frame
      (.setLayout (GridLayout. 2 2 3 3))
      (.add temp-text)
      (.add celsius-label)
      (.add convert-button)
      (.add fahrenheit-label)
      (.setSize 300 80)
      (.setVisible true))))
(celsius)
```

Figure 15: Swing app


Figure 16: Swing app output

## JVM Hosted

Sharing the JVM type system, GC, threads etc. It compiles all functions to JVM bytecode. Clojure is a great Java library consumer, offering the dot-target-member notation for calls to Java. Class names can be referenced in full, or as non-qualified names after being imported. Clojure supports the dynamic implementation of Java interfaces and classes using reify and proxy:

## Lamda

The keyword fn is called lambda in most other lisps. That in turn is named for the Greek symbol , and represents a function. This notation is from  calculus, developed in the 1930's by Alonzo Church to study the dynamics of functions. The -calculus is a foundation for the functional languages like Clojure, and  has come to be a near-religious symbol for those who program in these languages. Even Clojure uses it in its logo. It was renamed fn because we want to be able to use it a lot, and this is much easier to type.

# Paradigm 2: Event-Driven

## Why Event Driven Programming?

During the early days of computing, a program starts execution and then it continues through its steps until it is completed. If the user of the computer is involved, then the interaction is strictly controlled and limited to data entering into fields.The program may be simple and prints a prompt, wait for the user to enter a name, displays a message with this name in it, and then waits for him to strike a key to exit the program. This style is easy to comprehend and program. The problem arises when the programs need to become more sophisticated and the user must deal with more than just inputting data with the keyboard.
Today's embedded system, Graphical User Interface (GUI) programming model, and many other programming needs a different paradigm, known as event-driven programming.
Event-driven programming is an easy way to enable the programs to respond to many different inputs or events.
Today's programming presents a user interface and waits for an action to be taken by the user. Many different actions may be taken by the user, such as making selections in menu, pushing buttons, updating text fields, clicking icons, and many others. Each action makes an event to be raised. Other events can be raised without the direct action of the user, such as events corresponding to timer ticks of the internal clock, email being received, file-copy operations completing, etc. In programming, a situation where a particular action needs to be executed is often presented, but the method or even the object which is called upon to be executed is not known in advanced. For example, a button might know that it must notify some object when it is pressed but it might not know which object or objects need to be notified.

In addition to the GUI, the computer operating systems are another classic example of event-driven programs on at least two levels. At the lowest level, interrupt handlers behave like direct event handlers for hardware events, with the CPU hardware performing the role of the dispatcher. Operating systems mostly function as dispatchers for software processes, passing data and software interrupts to user processes that in many cases are programmed as event handlers themselves.
A command line interface can be considered as a special case of the event-driven model in which the system, which is inactive, waits for one very complex event – the entry of a command by the user.
Event-driven programs upgrade on sequential programs by acquiring a central event handler and dispatcher that waits for an event (any event) to occur, and then execute that event by calling that event handler.
Separation of the event detection and the event handling is an important technique for maintaining the simplicity and flexibility of the program. Applications of Event-driven programs are not bound by the constraints of procedural programs. Rather than the top-down approach of procedural languages, event-driven programs contain logical sections of code placed within events. There is no predefined order in which events occur, and usually the user has complete control over what code to be executed in an event-driven program by interactively triggering specific events, such as by clicking a button. The code which is contained in the event is called an event procedure

## What is Event Driven Programming

Event-driven programming is a programming paradigm in which the flow of a program is determined by events that occur, rather than by a predetermined sequence of instructions. In event-driven programming, the program's response to an event is triggered by a user action or by an external system event, such as a message or notification.
Event-driven programming is commonly used in graphical user interfaces (GUIs) and interactive applications, where the program needs to respond to user input in real-time. In event-driven programming, the program waits for an event to occur and then responds to it with an appropriate action or function. The program's event loop continuously checks for new events and responds to them as they occur.

## Event Handling

An **event** is a notification that something specific has occurred, such as a mouse click on a graphical button. The **event handler** is a segment of code that is executed in response to an event.

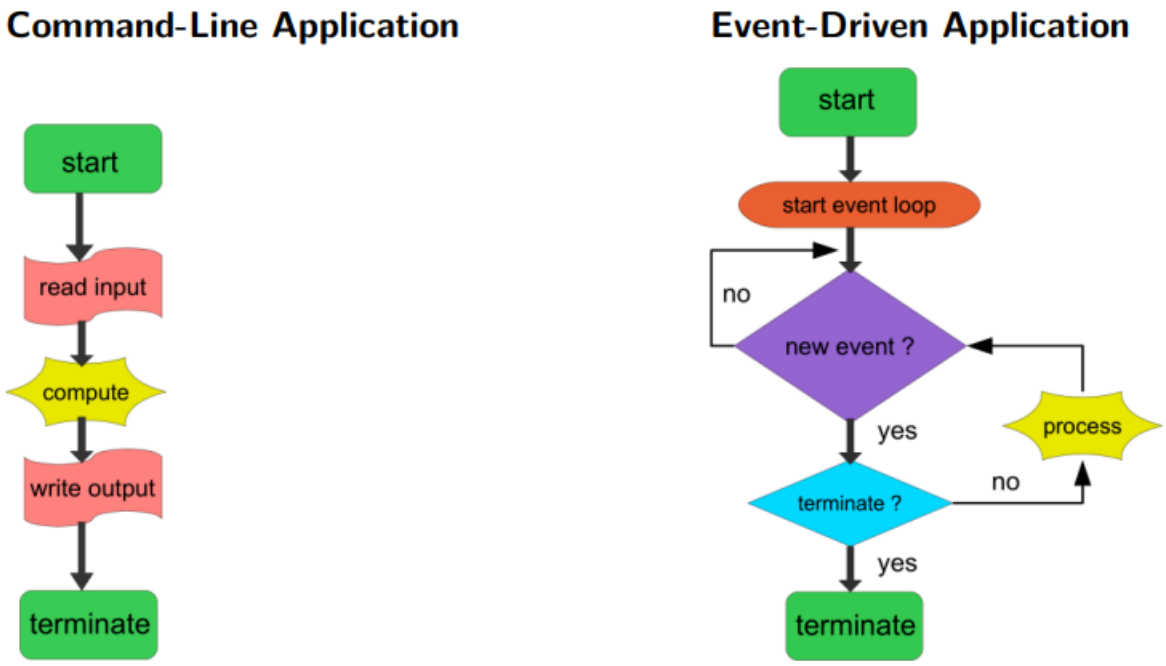


Figure 17: Program Architecture

Event handling is consisting of dealing with a situation whereby something has occurred and the software developer has to be notified of that situation. Sometimes the code written to take care of these situations is known as a callback and sometimes as an event handler. In both cases, the same fundamental principle is applied. That is, the developer has to implement (in C for example) a method that matches some specification that allows it to be called when the "event" occurs. This event can for example be some threshold being reached in some sensor, it could be a message being received from some broadcast mechanism or it could be some user interaction with a GUI. Generally, this event handling method will be invoked when the event occurs and will be passed some data to enable it identify the sender of the event and any event specific data. The action that the application must take is then determined by the event handler gives an illustration of the interactions that take place when handling GUI events in a little more detail. The three main steps with respect to a button are illustrated as follows:

- The user clicks on the button.
- An Event Args object is created by the button. This is an object that contains any additional data that must be made available to the event handler. This step is known as raising an event in C terminology.
- The button then invokes a suitable handler method (on an object somewhere) passing in a reference to itself (as the sender of the event) and the event args.
- The handler method can then apply any appropriate operation that it needs to perform.

**Properties**

- An event-driven program has no perceived stopping point.
- The traditional read-eval-print loop does not explicitly appear.
- An application processes an input and exits.

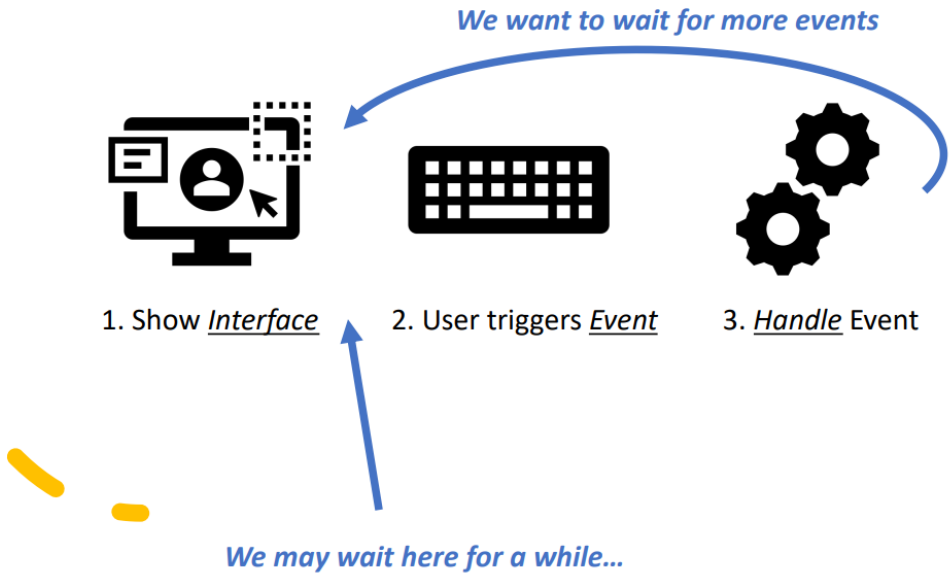


Figure 18: Event Handling

## Event handling Models

**Polling** (old, single-threaded)

- Application has the main loop that checks repeatedly for input or timeouts, then calls the appropriate routine.
- Examples include most (pre OS-X) Macintosh programs, Unix programs that just use basic X window.
- Leads to a more complex application, messy interfaces to the library.
- Largely replaced by the callback mechanism.

**Callbacks** (modern, multi-threads)

- A callback is when a routine in the application is called not from within the application, but from the library (by a widget when it detects a signal).
- The application must first tell the toolkit what events it wants to handle and which routines to call.
- Main loop and decoding of events are done by the library.
- This happens in more modern higher-level Unix GUI libraries like Xt, Qt, Motif or GTK; also the Microsoft Foundation Classes, Cocoa Framework (MacOSX).
- In Java programming callbacks is done using events and event listeners.

## Event Driven Programminig Approaches

- **Callbacks**
  Callbacks are the most basic and widely-used approach to event-driven programming. A callback function is registered to be called when a specific event occurs. When the event occurs, the program invokes the appropriate callback function to handle the event.

- **Observables**
  Observables are a more advanced approach to event-driven programming that allows for more complex data flows. Observables represent data streams that emit events over time. When a new event occurs, the program reacts by propagating changes through the data stream.
- **Promises**
  Promises are a popular approach to handling asynchronous operations in event-driven programming. A promise represents the eventual completion of an asynchronous operation and allows the program to register callbacks to be executed when the operation is complete.
- **Actors** Actors are a more complex approach to event-driven programming that is based on the concept of concurrent, independent entities that communicate with each other through message passing. Each actor is responsible for its own state and processing, and communication occurs through asynchronous message passing.
- **Reactive programming**
  Reactive programming is an approach that focuses on the flow of data and the propagation of change. In reactive programming, the program reacts to events by propagating changes through a network of data streams. Reactive programming uses a functional programming style that emphasizes immutability and declarative programming.

Reactive programming and Callback based programming are most commonly used approaches.

## Merits

- **Flexibility**: Because the flow of the application is monitored by events instead of a sequential program, it is not necessary for the user to conform to the programmer's understanding of how tasks should be performed.
- **Robustness**:Event-driven applications happen to be more robust because of their less sensitivity to the order in which users perform activities. In conventional programming, the programmer has to expect every sequence of activities virtually that the user might execute and define feedbacks to these sequences

## Demerits

The prime disadvantage of event-driven programs is that it is often difficult to find the source of errors when they can occur. This problem unfolds from the object-oriented nature of event-driven applications— since events are associated with a particular object, which the user may have to examine many objects before discovering the inappropriate procedure. This is especially true when events cascade (i.e., an event for one object triggers an event for a different object, and so on)

## History

The evolution of event-driven programming is told from the perspective of a business applications programmer who started programming in the late 1970's, worked mostly on IBM and Microsoft platforms, and most recently began working with Java and Python on Unix platforms. A professor of computer science – or someone who worked on IBM's CICS transaction processing monitor, or on the Mesa programming environment, or on the Andrew windowing system – would undoubtedly tell a different story, or at least tell it differently.

## Applications

- Graphical User Interfaces
- Web Development
- Game Development

- Automated Systems and IoT
- Middleware and Message Brokers
- Multithreading and Concurrency

### Programming languages

Languages Specifically designed for Events:

- **Erlang** Built for distributed, fault-tolerant systems, it uses message passing and events for concurrency.
- **Elixir** A modern language built on top of Erlang, offering a clean syntax for event-driven programming.
- **ReactiveX (Rx)** Not a language itself, but a library/framework available in various languages (like JavaScript, Java, Python, C) for implementing reactive programming, a paradigm closely aligned with event-driven principles.

Languages with Built-in Event Handling:

- **JavaScript** Core to web development, it's event-driven by nature, handling user interactions, network requests, and more.
- **Python** Features comprehensive frameworks like Tkinter, PyQt, and Kivy for building event-driven GUI applications.
- **Java** Offers the AWT and Swing frameworks for GUI development with event listeners and handlers.
- **C** Provides Windows Forms and WPF, mature frameworks for creating event-driven GUIs.
- **Visual Basic** A popular choice for rapid development of event-driven Windows applications.
- **TypeScript**

### Language for Paradigm 2: TypeScript

*TypeScript is a syntactic superset of JavaScript which adds static typing.* This basically means that TypeScript adds syntax on top of JavaScript, allowing developers to add types.
It was designed by Anders Hejlsberg (designer of C) at Microsoft. TypeScript is both a language and a set of tools.

### Why TypeScript?

JavaScript was introduced as a language for the client side. The development of Node.js has marked JavaScript as an emerging server-side technology too. However, as JavaScript code grows, it tends to get messier, making it difficult to maintain and reuse the code. Moreover, its failure to embrace the features of Object Orientation, strong type checking and compile-time error checks prevents JavaScript from succeeding at the enterprise level as a full-fledged server-side technology. **TypeScript was presented to bridge this gap**.
The benefits of TypeScript include:

- **Compilation:** JavaScript is an interpreted language. Hence, it needs to be run to test that it is valid. It means you write all the codes just to find no output, in case there is an error. Hence, you have to spend hours trying to find bugs in the code. The TypeScript transpiler provides the error-checking feature. TypeScript will compile the code and generate compilation errors, if it finds some sort of syntax errors. This helps to highlight errors before the script is run.

- **Strong Static Typing:** JavaScript is not strongly typed. TypeScript comes with an optional static typing and type inference system through the TLS (TypeScript Language Service). The type of a variable, declared with no type, may be inferred by the TLS based on its value.
- TypeScript **supports type definitions** for existing JavaScript libraries. TypeScript Definition file (with .d.ts extension) provides definition for external JavaScript libraries. Hence, TypeScript code can contain these libraries.
- TypeScript **supports Object Oriented Programming** concepts like classes, interfaces, inheritance, etc.

## Introduction

Despite its success, JavaScript remains a poor language for developing and maintaining large applications. TypeScript is an extension of JavaScript intended to address this deficiency. Syntactically, TypeScript is a superset of EcmaScript 5, so every JavaScript program is a TypeScript program. TypeScript enriches JavaScript with a module system, classes, interfaces, and a static type system. As TypeScript aims to provide lightweight assistance to programmers, the module system and the type system are flexible and easy to use. In particular, they support many common JavaScript programming practices. They also enable tooling and IDE experiences previously associated with languages such as C and Java. For instance, the types help catch mistakes statically, and enable other support for program development (for example, suggesting what methods might be called on an object). The support for classes is aligned with proposals currently being standardized for EcmaScript 6.

The TypeScript compiler checks TypeScript programs and emits JavaScript, so the programs can immediately run in a huge range of execution environments. The compiler is used extensively in Microsoft to author significant JavaScript applications. For example, recently3 Microsoft gave details of two substantial TypeScript projects: Monaco, an online code editor, which is around 225kloc, and XBox Music, a music service, which is around 160kloc. Since its announcement in late 2012, the compiler has also been used outside Microsoft, and it is opensource.

The TypeScript type system comprises a number of advanced constructs and concepts. These include structural type equivalence (rather than by-name type equivalence), types for object-based programming (as in object calculi), gradual typing (in the style of Siek and Taha [14]), subtyping of recursive types, and type operators. Collectively, these features should contribute greatly to a harmonious programming experience. One may wonder, still, how they can be made to fit with common JavaScript idioms and codebases. We regard the resolution of this question as one of the main themes in the design of TypeScript.

The TypeScript language is defined in a careful, clear, but informal document [11]. Naturally, this document contains certain ambiguities. For example, the language permits subtyping recursive types; the literature contains several rules for subtyping recursive types, not all sound, and the document does not say exactly which is employed. Therefore, it may be difficult to know exactly what is the type system, and in what ways it is sound or unsound. Nevertheless, the world of unsoundness is not a shapeless, unintelligible mess, and unsound languages are not all equally bad (nor all equally good). In classical logic, any two inconsistent theories are equivalent. In programming, on the other hand, unsoundness can arise from a great variety of sins (and virtues). At a minimum, we may wish to distinguish blunders from thoughtful compromises—many language designers and compiler writers are capable of both.

## Primary Goal

The primary goal of TypeScript is to give a statically typed experience to JavaScript development. A syntactic superset of JavaScript, it adds syntax for declaring and expressing types, for annotating properties, variables, parameters and return values with types, and for asserting the type of an expression. This paper's main aim is to formalize these type-system extensions.

TypeScript also adds a number of new language constructs, such as classes, modules, and lambda expressions. The TypeScript compiler implements these constructs by translation to JavaScript. However, these constructs are essentially back-ports of upcoming JavaScript features and, although they interact meaningfully with the type system, they do not affect its fundamental characteristics.

The intention of TypeScript is not to be a new programming language in its own right, but to enhance and support JavaScript development. Accordingly, a key design goal of the type system is to support current JavaScript styles and idioms, and to be applicable to the vast majority of the many existing—and very popular—JavaScript libraries.
This goal leads to a number of distinctive properties of the type system:

- **Full erasure:** The types of a TypeScript program leave no trace in the JavaScript emitted by the compiler. There are no run-time representations of types, and hence no run-time type checking. Current dynamic techniques for "type checking" in JavaScript programs, such as checking for the presence of certain properties, or the values of certain strings, may not be perfect, but good enough.

- **Structural types:** The TypeScript type system is structural rather than nominal. Whilst structural type systems are common in formal descriptions of object-oriented languages, most industrial mainstream languages, such as Java and C, are nominal. However, structural typing may be the only reasonable fit for JavaScript programming, where objects are often built from scratch (not from classes), and used purely based on their expected shape.

- **Unified object types:** In JavaScript, objects, functions, constructors, and arrays are not separate kinds of values: a given object can simultaneously play several of these roles. Therefore, object types in TypeScript can not only describe members but also contain call, constructor, and indexing signatures, describing the different ways the object can be used. In Featherweight TypeScript, for simplicity, we include only call signatures; constructor and index signatures are broadly similar.

- **Type inference:** TypeScript relies on type inference in order to minimize the number of type annotations that programmers need to provide explicitly. JavaScript is a pretty terse language, and the logic shouldn't be obscured by excessive new syntax. In practice, often only a small number of type annotations need to be given to allow the compiler to infer meaningful type signatures.

- **Gradual typing:** TypeScript is an example of a gradual type system, where parts of a program are statically typed, and others dynamically typed through the use of a distinguished dynamic type, written any. Gradual typing is typically implemented using run-time casts, but that is not practical in TypeScript, because of type erasure. As a result, typing errors not identified statically may remain undetected at run-time.

The significant initial uptake of TypeScript certainly suggests that this is the case of Gradual typing.

In addition to gradual typing, a few other design decisions deliberately lead to type holes and contribute to the unsoundness of the TypeScript type system

- **Downcasting:** The ability to explicitly downcast expressions is common in most typed object-oriented languages. However, in these languages, a downcast is compiled to a dynamic check. In TypeScript, this is not the case, as no trace of the type system is left in the emitted code. So incorrect downcasts are not detected, and may lead to (trapped) run-time errors.

- **Covariance:** TypeScript allows unsafe covariance of property types (despite their mutability) and parameter types (in addition to the contra variance that is the safe choice). Given the ridicule that other languages have endured for this decision, it may seem like an odd choice, but there are significant and sensible JavaScript patterns that just cannot be typed without covariance.

- **Indexing:** A peculiar fact of JavaScript is that member access through dot notation is just syntactic sugar for indexing with the member name as a string. Full TypeScript permits specifying indexing signatures, but (in their absence) allows indexing with any string. If the string is a literal that corresponds to a property known to the type system, then the result will have the type of that member (as usual with the dot notation). On the other hand, if the string is not a literal, or does not correspond to a known member, then the access is still allowed, and typed as any. Again, this aspect of TypeScript corresponds to commmon JavaScript usage, and results in another hole in the type system.

## Compiler

TypeScript is special is that instead of compiling straight to bytecode, TypeScript compiles to... JavaScript code! You then run that JavaScript code like you nor- mally would—in your browser, or with NodeJS. But the purpose of TypeScript is to make code safer!.
How?? **After the TypeScript Compiler generates an AST for your program but before it emits code—it typechecks your code**.

**Typechecker** - A special program that verifies that your code is typesafe.

This typechecking is the magic behind TypeScript. It's how TypeScript makes sure that your program works as you expect, that there aren't obvious mistakes.

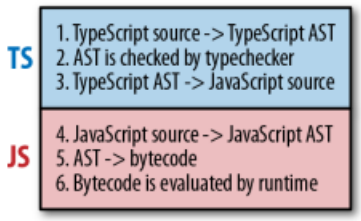


Figure 19: Compiling and Running TypeScript

The file extension for TypeScript is **.ts**

```
console.log("Hello, World!");
```

Figure 20: Sample Code for TypeScript

```
C:\Users\revan\Downloads>tsc hello.ts

C:\Users\revan\Downloads>node hello.ts
Hello, World!
```

Figure 21: Compilation and Running in windows terminal

## The Type System

**Type System - A set of rules that a typechecker uses to assign types to your program.**

In TypeScript you can explicitly annotate your types, or you can let TypeScript infer most of them for you.

To explicitly signal to TypeScript what your types are, use annotations. Annotations take the form value: type and tell the typechecker.

And if you want TypeScript to infer your types for you, just leave them off and let TypeScript get to work:
The types of variable are checked at **Compile time Types:** - A ser of values and the things you can do with them.

```
let a: number = 1                    // a is a number
let b: string = 'hello'              // b is a string
let c: boolean[] = [true, false] // c is an array of booleans
```

Figure 22: Defined variables explicitly

```
let a = 1                // a is a number
let b = 'hello'          // b is a string
let c = [true, false] // c is an array of booleans
```

Figure 23: Infering type of variable by TypeScript

- **any**: is the Godfather of types. It does anything for a price, but you don't want to ask any for a favor unless you're completely out of options. In TypeScript everything needs to have a type at compile time, and any is the default type when you (the pro-grammer) and TypeScript (the typechecker) can't figure out what type something is.It's a last resort type, and you should avoid it when possible.
- **unknown**: If any is the Godfather, then unknown is Keanu Reeves as undercover FBI agent Johnny Utah in Point Break: laid back, fits right in with the bad guys, but deep down has a respect for the law and is on the side of the good guys. For the few cases where you have a value whose type you really don't know ahead of time, don't use any, and instead reach for unknown. Like any, it represents any value, but TypeScript won't let you use an unknown type until you refine it by checking what it is.
- **boolean**
- **number**
- **string**
- **symbol**
- **objects**
- **bigint**: It is a newcomer to JavaScript and TypeScript,it lets you work with large integers without running into rounding errors.

## Event-Driven Architecture in TypeScript with JS

So just a breakdown before we start coding, to implement this we would have a method that performs an action and then declares the occurrence of an event and another file that contains these events. Finally, anywhere we call this method, we would have to initialise our event handler. So in summary we would have 3 files, one that contains our methods, another that contains our events and finally one that calls our method. I would be demonstrating this by creating a simple server using Express that just logs some messages whenever a particular endpoint is accessed. I would be implementing this in typescript and would be using the events package to emit these events.
Create 3 files: services.ts, events.ts, and index.ts. To run our typescript codes I'll be using the ts-node-dev module.
In events.ts file lets create our event.
First, we import the event emitter. Next up we create a class EventHandlers. After that we create an eventEmitter property and assign the EventEmiiter type to it and also make it private to this class, so it can only

```
import {EventEmitter} from 'events'

export class EventHandlers{

    private eventEmitter: EventEmitter;

    constructor(eventEmitter: EventEmitter){
        this.eventEmitter = eventEmitter;
    }

    registerRouteEventHandlers() {
        console.log('Events FiredUp')

        this.eventEmitter.on('routeAccess', (route) => {
            try{
                this.anounceRouteAccess(route);
                this.upgradeSecurity(route)
            }
            catch(error){
                console.log('An error occured while perform Route accessed actions')
            }
        });
    }

    private anounceRouteAccess( route: string){
        console.log(`${route} has been accessed`)
    }

    private upgradeSecurity(route: string){
        console.log(`Locking all other routes and initiating higher security measures on ${route} route`)
    }
}
```

Figure 24: Event Handling Code

be accessed within this class. Next, we initialize this event using a constructor, assigning the eventEmitter property of our class to the eventEmitter supplied. Next up we create our method 'routeEventHandlers' which would listen for all events as regards route access. We could have multiple methods for handling different events.

Inside the 'registerRouteEventHandlers' method we just print a message to show that this event handler is up and running. Then we define our events. We would be going with just one event for now 'routeAcess'. We can take in data that will be passed/sent anytime the occurrence of this event is declared. For this case, we would be supplying the route's name. With this information, we can perform various actions, but for the sake of this article, I would just be printing simple messages. Our event would perform 2 actions, one would be a function that announces that a route has been accessed, and another that announces that we are upgrading security.
For error handling, we would be using the try-catch module so if any occurs we can catch it. These functions would be private to this class alone. We use the 'this' keyword because we are referring to instance properties of the class.
Finally, we would create our functions, which said ab into (from the beginning) are private to this class. They take in the route name and print simple messages. You can always replace these messages with the actions you want to perform.

We import the EventEmitter from the events. We then create our class and then create a eventEmitter property private to this class that we would use to declare the occurrence of our events. We then initialize it using a constructor. Next up, we create a method that takes in the route's name. In this method, We would simply just print a basic message, and then emit(declare/announce) the event 'routeAccess'.

The request and response types, then the eventHandlers we created from our event file, then our service class from the service file. Finally, we import the EventEmitter from the events package.

Next create an instance of our express app. Then instances of the EventEmitter, our EventHandler, and our service. Next, we call the registerRouteEventHandlers method from the instance of the event handler we created. This now starts listening for events. Before I move on, for every instance of EventHandler and MyService we create, we pass in the eventEmitter because of the eventEmitter property they both posses. Next, we create an endpoint that listens for requests. Then inside it call the accessRoute method. Then

```
import {EventEmitter} from 'events'

export class MyService{

    private eventEmitter: EventEmitter;

    constructor(eventEmitter: EventEmitter){
        this.eventEmitter = eventEmitter
    }

    accessRoute (routeName: string ){
            try{
                console.log(`Opening ${routeName} Route`)
                this.eventEmitter.emit('routeAccess', routeName);

            }
            catch(e: any){
                console.log(e)
            }
    }
}
```

Figure 25: Service.ts file code EventEmiter

finally listen for requests. You can start the server using the npm run dev command, when you hit the start route using postman or any other option of your choice you can see the Event Handling in action.
So thats how event is handled.

## Event Handling with TypeScript

**Creating a Simple Project**:
Our objective for the day is to learn how event handlers work. Now, to add an event handler, we have two options at hand:

1. Using addEventListener().
2. Using specific onevent handlers.

If we look up for the function definition of addEventListener, we get
We see that the first parameter is type and the second parameter is listener. This listener in turn is a function itself, that accepts a HTMLElement and HTMLElementEventMap as parameters as shown in figure-29.

So, addEventListener("click", listenerFunction) means that we want to listen to the "click" event (a single mouse click) and then execute the listenerFunction whenever a click is encountered. Pretty simple right?

The listenerFunction is a simple function that takes in two parameters, this and ev, having the types of "HTMLElement" and "Event" respectively. Since we are getting a HTMLElement, we can directly work on it to change any property we want (to change the background color in our case).

We are using "preventDefault()" on the event to prevent the default action take as it normally would be.

```
import express, {Request, Response} from 'express'
import {EventHandlers} from './events'
import {MyService} from './services'
import {EventEmitter} from 'events'

const app = express()

const eventEmitter = new EventEmitter();

const eventHandler = new EventHandlers(eventEmitter)
const myService = new MyService(eventEmitter)

eventHandler.registerRouteEventHandlers()

app.get('/start', (req: Request,res: Response) => {
    try {
        myService.accessRoute('start')
        return res.status(200).json({"message" : "Welcome to EDA"})
    } catch (error) {
        return res.status(500).json({"message" : "An Error Occured on The Server"})
    }

})

const PORT = 8083

app.listen(PORT, ()=>{
    console.log(`Server Running on Port ${PORT}`)
})
```

Figure 26: Index.ts file - Request and Responses

**Executing Code**
**Step-1:**Lets use a very simple HTML file for ececution.
We are having a single div element with height and width as 200px and the background color initially set to blue. The element has an id of "sample", which our script will use to identify the element. We also have our script imported in the head section since it is a very small script. And we are executing the initFunction() to attach the event listener to our div element as shown in figure-30.

**Step-2**: Run the Compiler [**npx tsc script.ts**]
Our file is in TypeScript, but we are improving a JavaScript file.

That ends our play time with Event Handlers using TypeScript. It is super easy to attach event listeners and work with elements in TypeScript, as long you follow the types of the parameters properly. You can add any events besides click. You can put any logic in your handler function and it will work like hot butter through knife. You can follow up the links in the "Further Ahead" section to learn more on various topics.

# Analysis

## Analysis of Paradigams

**Functional Programming:**
**Strengths:**

- **Predictable and testable:** Pure functions and immutable data lead to consistent behavior and simplify testing.
- **Modular and composable:** Functions act as building blocks, enabling easy code reuse and combination.

```
└─$ npm run dev

> eda@1.0.0 dev
> ts-node-dev --respawn --transpile-only index.ts

[INFO] 16:07:27 ts-node-dev ver. 2.0.0 (using ts-node ver. 10.9.1, typescript ver. 5.1.
6)
Events FiredUp
Server Running on Port 8083
Opening start Route
start has been accessed
Locking all other routes and initiating higher security measures on start route
```

Figure 27: Event-Driven Architecture using TypeScript with JS

```typescript
export function initFunction() {
  const element = document.getElementById("sample");
  element?.addEventListener("click", listenerFunction);
}

function listenerFunction(this: HTMLElement, ev: Event) {
  ev.preventDefault();
  this.style.backgroundColor = "red";
}
```

Figure 28: code to change background colour

```
(method) HTMLElement.addEventListener<K extends keyof
HTMLElementEventMap>(type: K, listener: (this: HTMLElement, ev:
HTMLElementEventMap[K]) => any, options?: boolean |
AddEventListenerOptions | undefined): void (+1 overload)
```

Figure 29: definition of addEventListener

- **Concurrency-friendly:** Immutability allows parallel processing without data corruption concerns.
- **Referential transparency:** Makes reasoning about code easier due to deterministic function outputs.

**Weaknesses:**

- **Learning curve:** Can be less intuitive for object-oriented programmers, requiring different problem-solving approaches. **Can be less performant:** Some operations or algorithms might be less efficient compared to imperative approaches.
- **Error handling can be tricky:** Immutability can make handling errors and state changes more complex.

**Notable Features:**

```html
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta http-equiv="X-UA-Compatible" content="IE=edge" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>HTML Event Handling using TypeScript</title>
    <script src="script.js"></script>
  </head>
  <body>
    <div
      id="sample"
      style="height: 200px; width: 200px; background-color: blue"
    ></div>
  </body>
  <script>
    initFunction();
  </script>
</html>
```

Figure 30: HTML Script.ts

- **Pure functions:** Always return the same output for the same input, guaranteeing consistent behavior.
- **Immutable data structures:** Data cannot be directly modified, promoting predictable outcomes.
- **Recursion:** Enables solving problems through repeated function calls, often resulting in elegant solutions.
- **Pattern matching:** Powerful tool for concisely handling different input variations in functions.

**Event-Driven Paradigm:**
**Strengths:**

- **Highly responsive and interactive:** Responds dynamically to user actions and system events, leading to user-friendly experiences.
- **Asynchronous and concurrent-friendly:** Handles multiple events efficiently without blocking the main thread.
- **Fault-tolerant:** Events can be handled even if parts of the system fail, improving robustness.
- **Modular and decentralized:** Logic can be distributed across event handlers, simplifying large systems.

**Weaknesses:**

- **Debugging can be challenging:** Asynchronous flow and scattered logic can make tracing errors difficult.

- **State management crucial:** Keeping track of changing state in response to events requires careful design.
- **Potential for unpredictable behavior:** Sequencing of events can be non-deterministic, leading to unexpected outcomes.
- **Overlapping event handling can be complex:** Coordinating responses to multiple events simultaneously requires careful orchestration.

**Notable Features:**

- **Event loop:** Continuously monitors for events and triggers their corresponding handlers.
- **Event listeners:** Functions attached to elements or systems that react to specific events.
- **Reactive programming:** Continuous data flow and automatic responses to changes in data streams.
- **Callback functions:** Functions passed as arguments to be invoked later at specific points in the program.

## Analysis of Paradigams Languages

**Clojure:**
**Strengths:**

- **Powerful functional paradigm:** Built-in support for immutability, pure functions, and data persistence promotes predictable and testable code.
- **Concise and expressive syntax:** Lisp-like syntax allows for compact and elegant code, often with fewer lines compared to imperative languages.
- **Concurrency and parallelism:** Immutable data simplifies parallel processing without data race conditions.
- **Macro system:** Enables powerful code generation and domain-specific languages, extending the language for custom needs.

**Weaknesses:**

- **Steeper learning curve:** Lisp-like syntax and functional paradigm can be challenging for programmers familiar with object-oriented languages.
- **Smaller ecosystem:** Compared to mainstream languages, Clojure libraries and frameworks may be less vast.
- **Performance concerns:** Immutable data structures and heavy reliance on recursion can sometimes lead to performance overhead.
- **Debug complexity:** Nested function calls and lack of traditional debuggers can make tracing errors more challenging.

**Notable Features:**

- **Persistent data structures:** Data remains immutable, leading to referential transparency and predictable outcomes.
- **Immutable sequences:** Lists, vectors, and maps offer efficient and thread-safe data structures.
- **Recursion as a core concept:** Powerful tool for solving repetitive problems, often leading to concise and elegant solutions.

- **REPL-driven development:** Live coding environment facilitates rapid prototyping and experimentation.

**TypeScript:**
**Strengths:**

- **Gradual adoption of type safety:** Adds static typing to JavaScript's dynamic nature, catching errors early and improving code reliability.
- **Large and familiar ecosystem:** Leverages JavaScript's vast libraries and frameworks, making it easier to find existing solutions.
- **Compiles to JavaScript:** Runs seamlessly in any JavaScript environment, including browsers and Node.js.
- **Modern JavaScript features:** Supports latest ECMAScript versions, enabling features like classes, decorators, and async/await.

**Weaknesses:**

- **Can be verbose:** Static typing and additional syntax can lead to longer and more explicit code compared to dynamic languages.
- **Potential for runtime errors:** TypeScript catches static errors but cannot guarantee complete error-free execution due to JavaScript's dynamic nature.
- **Learning curve for object-oriented developers:** Functional features like immutability and higher-order functions can be unfamiliar for traditional OOP programmers.
- **Complexity with advanced features:** Decorators and other advanced features can introduce complexity if not used judiciously.

**Notable Features:**

- **Static type system:** Enforces data types for variables and functions, preventing runtime errors and improving code clarity.
- **Interfaces and classes:** Enable object-oriented programming patterns like inheritance and polymorphism.
- **Generics:** Allow writing code that works with different types of data without sacrificing type safety.
- **Asynchronous programming support:** Facilitates handling user interactions, network requests, and other asynchronous operations efficiently.

# Comparison

## Comparing and contrast the highlights between paradigms

**Similarities**:

- **Abstraction:** Both paradigms emphasize abstraction to manage complexity, hiding internal details and focusing on high-level concepts.
- **Componentization:** Both encourage dividing the problem into smaller, well-defined units for easier development and maintenance.
- **Concurrency potential:** Both can readily handle concurrent execution when properly structured.

**Differences:**

- **Focus**
  Functional: Emphasizes data transformation with immutable data and pure functions.
  Event-driven: Emphasizes response to events from users, systems, or external sources.
- **Control flow**
  Functional: Explicit control flow dictated by function calls and sequencing.
  Event-driven: Asynchronous control flow driven by event loops and reactions to incoming events.
- **Data management**
  Functional: Immutable data structures, minimizing side effects and promoting referential transparency.
  Event-driven: Can involve mutable data changing in response to events, requiring careful state management.
- **Logic organization**
  Functional: Logic grouped around functions, promoting modularity and composability.
  Event-driven: Logic often spread across event handlers, potentially challenging debugging and maintenance.

**Benefits**:
Functional: Predictable code, easier testing and debugging, strong parallelization potential.
Event-driven: Highly responsive and user-interactive, well-suited for asynchronous and distributed systems.

**Drawbacks**:
Functional: Can be less intuitive, may require different problem-solving approaches, potential performance overhead.
Event-driven: Debugging can be challenging, state management becomes crucial, logic spread can lead to complexity.

## Comparing and contrast the highlights between paradigm Languages

**Similarities**:

- **Strong Typing**: Both languages emphasize type safety, catching errors early and improving code reliability.
- **Rich Ecosystems**: They have vibrant communities, extensive libraries, and frameworks for diverse tasks.
- **JavaScript Interoperability**: They seamlessly interact with JavaScript code, enabling integration with existing codebases and web development.
- **Run on JVM**: Clojure and TypeScript code can be executed on the Java Virtual Machine (JVM), benefiting from its portability and performance optimizations.

**Differences:**

- **Paradigm**
  Clojure: Functional programming language, emphasizing immutability, pure functions, and data transformation.
  TypeScript: Superset of JavaScript, adding static typing to JavaScript's object-oriented and prototype-based nature.
- **Compilation**
  Clojure: Compiles to JVM bytecode, running on various platforms with a JVM.
  TypeScript: Compiles to JavaScript, running in any JavaScript environment (browsers, Node.js).
- **Syntax**
  Clojure: Lisp-like syntax with parentheses, prefix notation, and homoiconicity (code as data).
  TypeScript: Similar to JavaScript, with added type annotations and features from newer ECMAScript versions.

## Challenges Faced

- As TypeScript is not specifically designed for event-driven programming, but it is suited for it I faced challenging for finding resources regarding the topic at last with a reference of a article in medium I created my own objective to showcase the event handling in TypeScript.
- While placing the images in latex it took a lot of time for alligning it properly and some of the images are left unalligned properly.

## Conclusion

In conclusion, the exploration of different programming paradigms in this assignment has provided valuable insights into the diverse approaches to software development. Functional programming, with its emphasis on pure functions and immutable data, offers a powerful way to manage complexity and promote code reliability. The event-driven paradigm, on the other hand, focuses on responsiveness and interactivity, making it well-suited for user-friendly experiences and asynchronous operations. TypeScript, as a superset of JavaScript, addresses the need for static typing and other modern features, bridging the gap between client-side and server-side technologies.

By understanding the strengths and weaknesses of each paradigm, programmers can make informed decisions about which approach best suits their specific requirements. While functional programming promotes predictability and ease of testing, event-driven programming excels in responsiveness and fault tolerance. TypeScript, with its gradual adoption of type safety and seamless integration with existing JavaScript environments, offers a compelling solution for developers seeking a balance between dynamic and static typing. Furthermore, the comparison and contrast of these paradigms have highlighted the importance of abstraction, componentization, and concurrency potential as common threads across different programming approaches. It is evident that each paradigm has its unique set of benefits and challenges, and the choice of paradigm should align with the specific needs of the project and the preferences of the development team.

In the ever-evolving landscape of software development, the knowledge and understanding of different programming paradigms are invaluable assets for programmers and software engineers. By embracing the principles and concepts underlying these paradigms, developers can expand their problem-solving capabilities and adapt to the dynamic demands of the industry, ultimately contributing to the creation of robust, scalable, and maintainable software solutions.

## References


- https://github.com/readme/guides/functional-programming-basics
- https://link.springer.com/chapter/10.1007/978-3-031-34144-1$_1$1$https://rb.gy/8ohnby$
- https://www.codingdojo.com/blog/what-is-functional-programming
- https://doc.lagout.org/programmation/Functional
- https://wr.informatik.uni-hamburg.de/$_m edia/teaching/sommersemester_2020/siw-20-christian_wolff_clojure.pdf$
- http://blackjack-security.net/private$_f tp/Clojure/Clojurehttps://clojure.org/about/functional_programming$
- https://courses.engr.illinois.edu/cs225/fa2017/honors/handouts/introduction-to-clojure.pdf

Event Driven

- https://dif7uuh3zqcps.cloudfront.net/wp-content/uploads/sites/11/2018/07/17035708/2017$_I ssue1_P aper4.pdfhttps://www.cl.cam.ac.uk/ nk480/essence-of-events.pdf$
- https://core.ac.uk/download/pdf/33428548.pdf

TypeScript

- https://users.soe.ucsc.edu/ abadi/Papers/FTS-submitted.pdf
- https://www.tutorialspoint.com/typescript/typescript$_{t}utorial.pdfhttps://typescript-book.com/files/Typescript-Sample-Chapter.pdf$
- https://www.typescriptlang.org/docs/handbook/typescript-from-scratch.html
- https://riptutorial.com/typescript/example/29544/finding-object-in-array