- **Functional Programming : Composing software**
  - FP is a programming paradigm where applications are composed using
    - pure funcitons
    - avoiding shared mutable state
    - avoiding side-effects
    - Is usually more declarative than imperative : we express what to do rather how to do. So tends to be more concise, more predictable, easier to compose and more amenable to be parallelised.
  - Contrast this with :
    - Object Oriented programming : data and behaviours are coloacted
    - procedural programming :an imperative style grouping algorithms into procedures which tend to rely on shared mutable state
  - Some ideas used in FP
    - Pure functions
      - Given the same inputs, always returns the same output
      - Has no side-effects
    - Function Composition
      - Process of combining two or more functions in order to produce a new function or perform some computation
      - Eg Partial application, Currying, map/reduce/filter pipelines
    - Shared State
      - any variable, object, or memory space that exists in a shared scope, or as the property of an object being passed between scopes
      - A shared scope can include global scope or closure scopes.
      - Functional programming avoids shared state
      - Relies on immutable data structures and pure calculations to derive new data from existing data.
        - race condition
        - order of operations matter
    - Immutability
      - trie data structure: Copy on write
        - Tries use structural sharing to share reference memory locations for all the parts of the object which are unchanged after a “mutation”, which

uses less memory, and enables significant performance improvements for some kinds of operations.

    - Immutable.js and Mori

- Side Effects

    - Any application state change that is observable outside the called function other than its return value

        - Modifying any external variable or object property (e.g., a global variable, or a variable in the parent function scope chain)

        - Logging to the console

        - Writing to the screen

        - Writing to a file

        - Writing to the network

        - Triggering any external process

        - Calling any other functions with side-effects

    - Side effects are mostly avoided in functional programming, only controlled side effect allowed, manage state and component rendering in separate, loosely coupled modules.

- Reusability Through Higher Order Functions

    - A higher order function is any function which takes a function as an argument, returns a function, or both

    - Higher order functions are often used to:

        - Abstract or isolate actions, effects, or async flow control using callback functions, promises, monads, etc.

        - Create utilities which can act on a wide variety of data types

        - Partially apply a function to its arguments or create a curried function for the purpose of reuse or function composition.

        - Take a list of functions and return some composition of those input functions

    - JavaScript has first class functions, which allows us to treat functions as data — assign them to variables, pass them to other functions, return them from functions, etc.

- Containers, Functors, Lists, and Streams

    - A functor data structure is a data structure that can be mapped over, that is, provides a map interface. So, Array is a functor

    - So, functor can contain objects of any datatype and all we need is an appropriate mapper function to transform functor

    - Array is just a list of things. A list expressed over time is a stream — so you

can apply the same kinds of utilities to process streams of incoming events

- Declarative vs Imperative
    - Imperative: the flow control: How to do things.
    - Declarative programs abstract the flow control process (the how gets abstracted away), and instead spend lines of code describing the data flow: What to do.
- Expressions over statements