~~COURSE OUTLINE~~

~~Composable Software Construction~~

~~Practical approaches to writing simple, reusable, maintainable and dependable code in Typescript.~~

This version of the course is no longer available.

It has been updated with new content, and reorganized to address feedback in the initial versions with a new pricing structure that better represents the time spent on behalf of the instructor.

INSTRUCTOR

Lakin Wecker

lakin@structuredabstraction.com

structuredabstraction.com

Introduction

This course is intended for students who have learned how to code, and know how to solve algorithmic challenges, but who are never quite happy with how their code turns out in the end. Sure, their code accomplishes its goals, but it's not as reusable and generic as what they see in other well-regarded projects. Typically, the primary experience they will have is from a course or book they have read on "Software Engineering". The term Software Engineering frequently refers to a list of 4 important, but incomplete, aspects of producing software :

1. Working in groups.
2. Requirements/project management.
3. Design patterns
4. System Architecture

While these are important subjects, it skips over the most fundamental aspect of software development: actually writing the code. In Code Complete, McConnel [1] uses the term Software Construction to label what comes after the design and before the deployment. It is the act of taking the requirements and actually writing the code to satisfy them. This course aims to improve on the students ability to take a well-defined task and produce software that will be praised by their colleagues for its clarity, succinctness, dependability and reusability.

Overview

The course takes 40 hours with 16 classes over an 8-10 week timeline. Each class takes 2.5 hours and is split into three parts:

1. **Theory**
   Introduction to each of the concepts in a mathematical, diagrammatic way using a functional style notation. Motivated by a real-world problem, I will show the students how they can use existing theory to solve these problems. At the end of these sections, a simple in-class quiz will be given to assess learning outcomes.
2. **Typescript**
   I cover typescript features which are directly related to the previously introduced theory in detail. I develop code on-screen during the session, with feedback and interaction from the students. Care will be taken to show how the typescript code is compiled to javascript. A simple online in-class quiz format is used to assess learning outcomes. A laptop or access to a computer for development will be necessary to complete some sections of the quiz.
3. **Industry examples + Group coding session.**
   An example from real projects (typically a project from the company) will be presented. These examples will use imperative coding paradigms that are taught in many courses and are common in industry among new programmers. The examples will not use the concepts taught during the class and students will be challenged to improve the code into something which does use these concepts with guidance from the instructor.

Logistics & Scheduling

This course is provided on a per-client scheduled basis for a minimum of 2 students and maximum of 4 students at a time. Tuition is due in full before the start of the first class. The course can be held over zoom, or in person at a client-provided location within Calgary, AB.

The course includes 16 classes, which are scheduled according to the availability of the students and instructor within a 10 week period.

Cost

$4,000 CAD per student. (40 hours @ $100 CAD per hr per student)

Outcomes:

Students of this course are expected to gain a measurable elevation of their software development abilities. The primary outcome is an increased ability to read, design and write high quality software. The particulars of this are:

- Observable decrease in run time errors with a goal of 0
- Increased ability to construct maintainable software through an improved ability to identify, design and construct code with well-known composable design-patterns
- Increase in ability to write self-documenting code, expressing ideas, constraints and domain-specific concepts in types and letting the compiler enforce them
- Increased ability to approach software modelling based on mathematical concepts and notation.
- Knowledge and implementation of algebraic data types
- Knowledge and implementation of functional programming best practices

These outcomes will be materialized by the students through the course requirements of in class discussion, hands on coding, and industry examples.

Retention and proper implementation of course material will be evaluated on a continuous basis to ensure the acquired skills persist beyond the completion of the course.

Syllabus

Week 1 > Class 1 > Theory

- Functions revisited:
    - Basics:
        - Notation
        - Inputs / Outputs
        - Purity and data immutability
        - Side Effects
        - Signature
        - Examples from Elm
    - Composition
        - What does this mean?
        - Why do we care
        - Where does composition show up?
            - Objects
            - Functions
            - Examples from scala

Week 1 > Class 1 > Typescript

- Functions in typescript:
    - Typical functions
    - Alternate function syntax (function keyword vs =>)
    - IIFE
    - Function literals
    - Typing functions
    - Mutability
    - Immutability
    - Writing a compose operator in typescript
    - Scope
    - React composition model

            - This.props.children
            - React.PropsWithChildren
        - Composition / using `compose` from popular libraries (ex. Redux, ramda, lodash etc.)

Week 1 › Class 2 › Theory

- Expressions vs Statements
- Functions Again:
    - Multiple Arguments
    - Application
    - Partial Application
    - Currying
    - Why do we use this? Why do we care?
    - Example in Elm

Week 1 › Class 2 › Typescript

- Typescript:
    - When do we use these in typescript?
    - Application
    - Partial Application
    - Currying
    - fp-ts

Week 2 › Class 3 › Theory

- Function Customization
    - Identifying the parts of a function
    - Which parts are static?
    - Which parts are dynamic?
    - How can we customize these parts?
    - Scope (again).
    - Closures
    - Higher Order Functions

Week 2 › Class 3 › Typescript

- Functions as variables in typescript
    - How to store them
        - What is their scope?
    - How to call them
    - Are they objects?
    - What attributes do they have?
    - How to define functions which receive functions as parameters
    - Existing examples within builtin javascript types.

Week 2 > Class 4 > Theory

- Types revisited
    - What are they?
        - primitive types
        - builtin types
        - user-defined types
        - Why do we care?
    - Composite types (composition again!):
        - Product types
            - Records.
            - What number of valid states are there?
            - What implications does this have for maintenance?
            - Examples in Elm
        - Sum types
            - Variants
            - Enums
            - Wrapping Data
            - What number of valid states are there?
            - Examples in Elm
            - Matching
                - Sum types
                - Elm
                - What about product types?

Week 2 > Class 4 > Typescript

- Types in typescript:
    - Records
    - Types
    - Interfaces
    - Mappings
    - Variants
    - Matching
    - Making the compiler work for you.

Week 3 > Class 5 > Theory

- Generics
  - Similar to dynamic portions of a function, how do we get dynamic portions of a type?
  - Where is this useful?
  - Constraints on types

Week 3 > Class 5 > Typescript

- Introduction to advanced typescript types
- Utility-types
  - Pick
  - Omit
  - Required
- Type composition
- Typeguards
- discriminated/tagged unions

Week 3 > Class 6 > Theory

- Containers
  - Lists
    - Lists in javascript (different types)
    - Lists in typescript (same type)
    - Lists in typescript with Variants
  - Map
    - Records in javascript (different types)
    - Records in typescript (same type)
    - Records in typescript with Variants
- Functions as transformations between types
  - What about side-effect functions?
- Function lifting basics
  - Taking a function that operates on a given type
  - Applying to a list (.map)
  - Applying it to a Mapping (.map)
  - Making a new function that takes a List or takes a Record

- Covariance / Contravariance

Week 3 > Class 6 > Typescript

- Lifting in typescript.
- Implementing our own .lift functions (List/Array/Map)
- fp-ts versions of .map/.lift

Week 4 › Class 7 › Theory

- Type Invariants & Data Encapsulation
    - Functional data encapsulation
        - closures
        - scope
        - composition
        - immutability
    - Modules as encapsulation
    - Classes
        - Constructors
        - Mutability
        - Immutability
        - How can we model data that changes with immutability?

Week 4 › Class 7 › Typescript

- Revisiting closures in typescript
    - How to use them for encapsulation
- Modules and visibility in typescript
    - How to use them for encapsulation
- Classes
    - How to use them for encapsulation
    - Immutability
    - Immutable classes with changing data.

Week 4 › Class 8 › Theory

- Making your tools work for you.
- How can we do less work, but write better code?
- Continuous integration
- Continuous testing
- Continuous delivery
- Editors
- Refactoring
- Docs on hover
- Git
- Learning from colleagues

Week 4 › Class 8 › Typescript

- Typescript compiler configuration.
- Configuring VS Code.
- Turning off "any".
- Using linters
- Using hinters
- Using CI
- Using CD
- Refactoring tools built into your editor.
- Type-info on hover.
- Using type information to navigate the code base.s

Week 5 > Class 9 > Theory

- Making impossible states a compiler error
  - Let's build a questionnaire
    - State knows which question they're on, and information about their answers.
    - No empty questionnaires, never beyond the limits.
  - Let's make it so that we literally can't represent invalid states

Week 5 > Class 9 > Typescript

- Let's build a complete zippered list in typescript
  - Functions for going backwards and forwards and answering questions.
- zipper list in other libraries (fp-ts?)

Week 5 > Class 10 > Theory

- Introducing functions which may not have a valid return value
- How can we represent them?
- What can we do with them?
- How can we use the compiler to our advantage
- What else can we do with these types?
- What sorts of ways do languages that you use deal with this?
- Is it statically typesafe?

Week 5 > Class 10 > Typescript

- null | undefined in typescript
- Should we use them?
- Can we use the other pattern that we have?
- What might it look like?
- Implementing Optional<T> and then using it in typescript.
- Optional<T> in fp-ts.

Week 6 > Class 11 > Theory

- How can we deal with functions that may have an error condition?
- Exceptions?
- What about languages that don't have exceptions?
- Result<T>
- Either<L, R>
- Other uses of Either

Week 6 > Class 11 > Typescript

- Implementing a Result type in typescript.
- fp-ts Either

Week 6 > Class 12 > Theory

- Making impossible states impossible
- Fetching data from a remote data store
- What are the various valid states?
- How can we represent them?
- RemoteData<T>

Week 6 > Class 12 > Typescript

- How can we implement Remote data in typescript?
- Let's implement a simple one.
- What about match?

Week 7 › Class 13 › Theory

- Advanced functions
- Thinking of them as a pipeline
- Thinking about what a function does
- Not how it does it.
- Advanced operations available
- pipe, fold, match, etc. Other more advanced algorithms.
    - partition, etc.
    -

Week 7 › Class 13 › Typescript

- Implementing these advanced function types in typescript
- Identifying where they are used.
- fp-ts.

Week 7 › Class 14 › Theory

- Refactoring code to be readable.
- What is readable code?
- Why do we care?

Week 7 › Class 14 › Typescript

- More group sessions refactoring code from existing code base

Week 8 > Class 15 > Theory

- Software architecture
- How do we make software that works well with each other?
- How do you design interfaces
- What code goes where?
- Single-responsibility

Week 8 > Class 15 > Typescript

- Implementing all of this in typescript
- Making certain things private
- Testing
- Linting

Week 8 > Class 16 > Theory

- The onion architecture
- Dependency injection
- Domain specific design

Week 8 > Class 16 > Typescript

- Implementing all of this in typescript.

The course includes 8 weeks of material. Depending on the level of the students, and their progress, the following "Bonus" classes may be taught near the end of the course.

Week 9 > Class 17 > Theory

- Understanding Javascript Module Formats
    - Commonjs
    - Require
    - AMD
    - UMD
    - ESM
    - Transpiling
    - What works in a browser/node and what doesn't

Week 9 > Class 17 > Typescript

- Transpiling Javascript with babel
- Transpiling compiled typescript with babel
- package.json/tsconfig.json inclusions/modifications for facilitating different scenarios

Week 10 › Class 18 › Theory

- Domain Driven Design
    - Modeling software and data for a business domain
    - Placing uncertainties at the edge
    - Comparison to OOP, Object type, and FP
    - Why is this important?

Week 10 › Class 18 › Typescript

- Types revisited
    - User defined types
    - Type composition
    - Required vs optional

References

[1] - https://www.oreilly.com/library/view/code-complete-second/0735619670/