# WEB 3, Session 1

Course

TypeScript

# Ole Ildsgaard Hougaard

- oih@via.dk
- Computer Scientist (PhD)
  - Programming languages
  - Type systems
- Industry experience
  - Uni-C
    - Educational Software, Games
    - Senior Developer
  - Acure / IBM
    - Healthcare solutions
    - Developer, architect, Scrum master
  - Google
    - V8 JavaScript Engine
- VIA since 2009

# Course

- JavaScript/TypeScript on both the client and server side
  - In fact, almost exclusively TypeScript
- Format: Lecture + work on course assignment
  - Few, if any other exercises.
  - I'll try to keep the lecture part to around 2 lessons, but no guarantees
- 6 course assignments
  - You'll implement an UNO game app in 2 different ways - each divided into 3 assignments
- Exam
  - Oral, 20 minutes
  - You'll draw 2 assignments and discuss both them and the surrounding theory

# Topics

- Object-oriented TypeScript programming
- Object-oriented GUI programming and State Management
  - Vue.js + Pinia
- Client/Server programming
  - If time permits: Server-side rendering
- Functional programming with Lodash
- Functional state management with Redux
- Server-side rendering with Next.js
- Reactive programming with RxJS

# How I code JavaScript/TypeScript

- No semicolons 😛
- Arrow functions
- Spread and destructuring
- map + filter + reduce
- interface + object literal - avoiding `this`
- Higher-order functions
  - Provider
  - Function

# Arrow Functions

```
// Standard, I use this for larger functions:
function myFunction(x: number, y: number): number {
  return x + x * y
}

// Arrow function, I use this for smaller functions
const myFunction = (x: number, y: number) => x + x * y
```

# Spread

```
const numbers = [1, 2, 3]
const moreNumbers = [...numbers, 4] // [1, 2, 3, 4]

const point2D = {x: 75, y: 120}
const point3D = {...point2D, z: 80}
// {x: 75, y: 120, z: 80}
```

# Destructuring arrays

```
const numbers = [1, 2, 3]
const [a, b, c] = numbers // a === 1, b === 2, c === 3

const [x, y] = numbers // x === 1, y === 2

const [i, j, k, l] = numbers // l === undefined

const [first, ...rest] = numbers
// first === 1, rest is [2, 3]
```

# Destructuring objects

```
const point3D = {x: 75, y: 120, z: 80}

const {x, y} = point3D // x === 75, y === 120
const {z, ...point2D} = point3D
// z === 80, point2D is {x: 75, y: 120}
```

---

```
const x = 75, y = 120
const point2D = {x, y}
```

# Map + filter + reduce

```
const salesRecord = [
  {type: 'Dog food', amount: 2, price: 19.99},
  {type: 'Cat food', amount: 2, price: 29.99},
  {type: 'Dog food', amount: 1, price: 19.99},
  {type: 'Fish food', amount: 2, price: 9.99},
]

const dogFoodRevenue = salesRecord
  .filter(({type}) => type === 'Dog food')
  .map(({amount, price}) => amount * price)
  .reduce((sum, total) => sum + total, 0)
```

# Interface + object literal

```
interface Counter {
  next(): number
}

function counter(start: number = 0, step: number = 1): Counter {
  let counter = start
  const next = () => counter++
  return { next }
}

const cnt = counter(1)
```

# Provider

```
type Counter = () => number

function counter(start: number = 0, step: number = 1): Counter {
  let counter = start
  return () => counter++
}

const cnt = counter(1)
```

# Function

```
function map<T, U>(ts: T[], f: (t: T) => U) {
  const result: U[] = []
  for(let t of ts) {
    result.push(f(t))
  }
}

let ns = [1, 2, 3]
let doubles = map(ns, x => 2 * x)
console.log(doubles) // [2, 4, 6]
```

# TypeScript

```
const e = 8
const s = '7'

console.log(e * s)
```

# any

```
const e: any = 8
const s: any = '7'

console.log(e * s)


// Note: function parameters are type any unless
// otherwise specified
```

# Object type vs interface
## - What's the difference?

```
interface PointInterface {
  x: number
  y: number
  z?: number
}

type PointType = {
  x: number
  y: number
  z?: number
}
```

# Why is this legal?

```
type Point = {x: number, y: number}

const distanceFromOrigin = (p: Point) => Math.sqrt(p.x * p.x + p.y * p.y)

const p = {
  x: 20,
  y: 35,
  unit: 'px'
}

const dist = distanceFromOrigin(p)
```

# Ordering

- You know $2 \leq 3$
- You also know "goodbye" ≤ "hello"
- These are **total** orders
- How about $\{2, 3\} \subseteq \{1, 2, 3, 4\}$?
  - Note: $\{1, 2\} \not\subseteq \{2, 3\}$ and $\{2, 3\} \not\subseteq \{1, 2\}$
  - $\subseteq$ is a **partial** order
- Another partial order: | (divides)

# Subtype ordering

- The more general type is considered larger.
- { x: number, y: number } <: { x: number }
- { x: number, z: string } <: { x: number }
- { x: number, z: string } and { x: number, y: number } are unrelated

# Type lattice

# Subtype properties

| | | |
|---|---|---|
| $3 < 4$ | min(3, 4) | max(3, 4) |
| $\{2, 3\} \subseteq \{1, 2, 3, 4\}$ | $\{1, 2\} \cap \{2, 3\}$ | $\{1, 2\} \cup \{2, 3\}$ |
| $2 \mid 4$ | gcd(4, 14) | lcm(4, 14) |
| { x: number, y: number }<br><:<br>{ x: number } | { x: number }<br>&<br>{ z: string } | { x: number }<br>\|<br>{ z: string } |

# Intersection types

```
type Point = {x: number, y: number}

type Measurable = {unit: 'px' | 'pt' | 'cm' | 'in'}

const p: Point = {x: 100, y: 200}

const m: Measurable = {unit: 'px'}

const screenPoint: Point & Measurable = {...p, ...m}
```

# Narrowing

```
function double(n: number | undefined) {
  if (n === undefined) {
    return undefined
  }
  return n * 2
}
```

# Narrowing problem

```
type LoadingState = { percentComplete: number }
type FailedState = { statusCode : number }
type OkState = { payload: number[] }

type State = LoadingState | FailedState | OkState

function reportState(state: State) {
  if (state.percentComplete !== undefined) {
    console.log(`Loading ${state.percentComplete}% done`)
  } // And so on
}
```

# Discriminated Unions

```
type LoadingState = { status: 'loading', percentComplete: number }
type FailedState = { status: 'failed', statusCode : number }
type OkState = { status: 'ok', payload: number[] }

type State = LoadingState | FailedState | OkState

function reportState(state: State) {
  if (state.status === 'loading') {
    console.log(`Loading ${state.percentComplete}% done`)
  } // And so on
}
```

# Subsets of discriminated unions

```
type LoadingState = { status: 'loading', percentComplete: number }
type FailedState = { status: 'failed', statusCode : number }
type OkState = { status: 'ok', payload: number[] }

type State = LoadingState | FailedState | OkState

type FinishedState = State & { status: 'failed' | 'ok' }
```

# Working out the type

# Extracting types from object types

```
type LoadingState = { status: 'loading', percentComplete: number }
type FailedState = { status: 'failed', statusCode : number }
type OkState = { status: 'ok', payload: number[] }

type State = LoadingState | FailedState | OkState

type FinishedState = State & {
    status: FailedState['status'] | OkState['status']
}
```

# What is the status type?

```
type LoadingState = { status: 'loading', percentComplete: number }
type FailedState = { status: 'failed', statusCode : number }
type OkState = { status: 'ok', payload: number[] }

type State = LoadingState | FailedState | OkState

type Status = State['status']
```

# Course Assignment