Web3, Session 2

TypeScript (II)

Ole I Hougaard, olehougaard@gmail.com

Questions about the assignment?

Beyond simple TypeScript

- We want *precise* types:
  - Guarantee against (certain types of) errors
  - Allows everything you reasonably want to do
- That makes type systems complex
  - TypeScript is one of the more complex
  - They still need ‘any’ as an escape clause
- On top of that, we want to follow the DRY principle
  - Only write the same piece of logic once
  - Only one place to change things

Keeping it DRY

Discriminator

- From last time:

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

Discriminated Union

- Bad:

```
type Status = 'loading'| 'failed' | 'ok'
```

What if we add Connecting?

- Good:

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

Covered in this session

- Type predicates
- Immutability
- Utility Types
- Type manipulations
- (time permitting) Type helpers

Why Type Predicates?

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

type State = LoadingState | FailedState | OkState


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

Type predicates

```
function isLoading(state: State): state is LoadingState {
  return (state as LoadingState).percentComplete !== undefined
}

function isFailed(state: State): state is FailedState {
  return (state as FailedState).statusCode !== undefined
}

function isOk(state: State): state is OkState {
  return (state as OkState).payload !== undefined
}
```

Using type predicates

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

Immutability

- It's important to manage state change
  - Better correctness
  - Easier asynchronous programming
- Immutability means: You *cannot* change this
  - Easier to reason about the code
- Example: Playing cards

```
type PlayingCard = {
  readonly suit: Suit,
  readonly rank: Rank
}
```

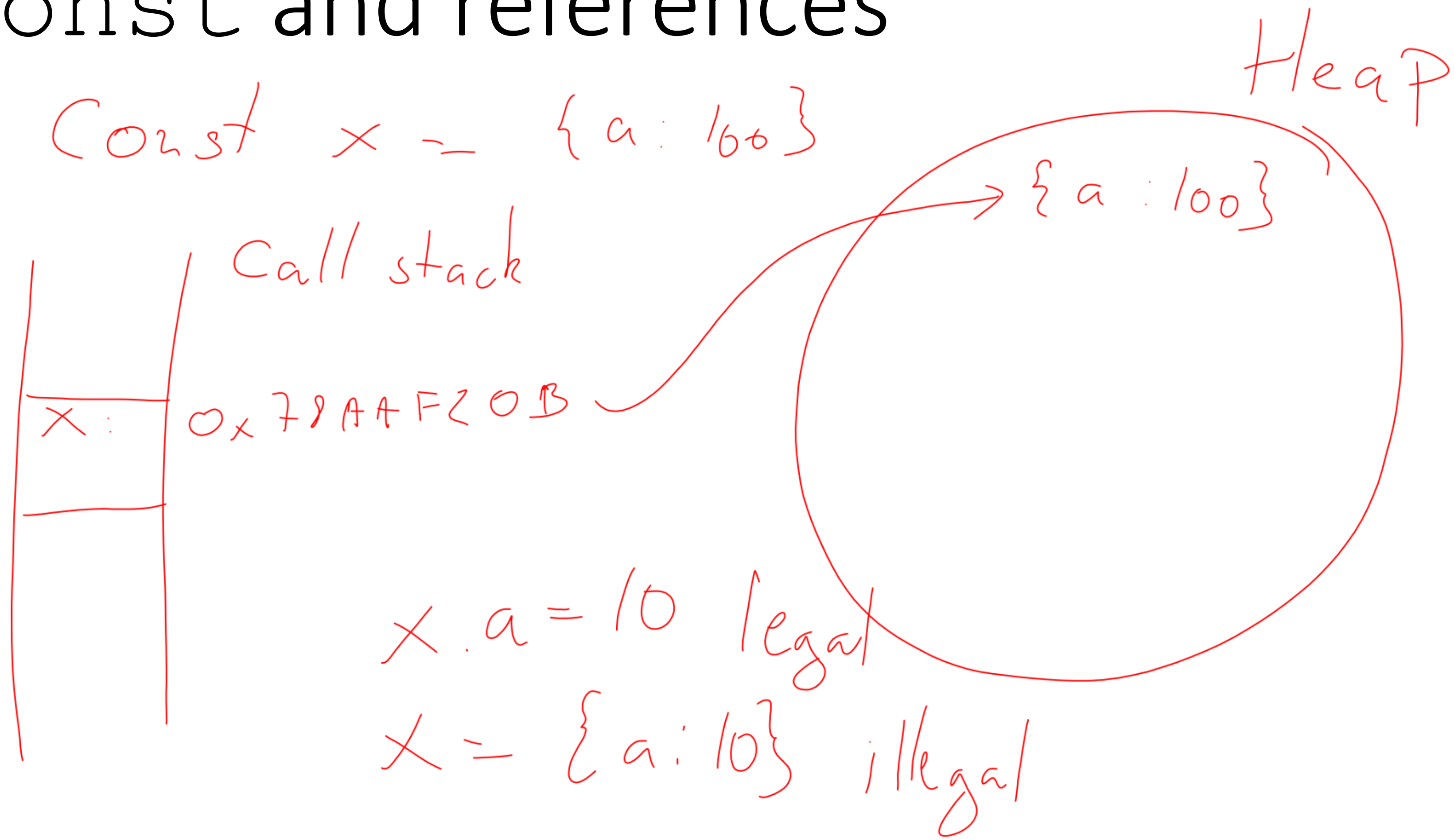
What's going on here?

```
const statusCodes = {
  ok: 200,
  "Not found": 404,
  "Internal Server Error": 500
}

statusCodes.ok = 201 // Not an error
```

2

`const` and references



as const

```
const constStatusCodes = {
    ok: 200,
    "Not found": 404,
    "Internal Server Error": 500
} as const
```

constStatusCodes.ok = 201

illegal

```
type ImmutableStatusCodes = typeof constStatusCodes
```

const array

```
const Suits = ['Clubs', 'Diamonds', 'Hearts', 'Spades'] as const

type SuitsType = typeof Suits

type ClubType = SuitsType[0]

type Suit = SuitsType[number]
```

['Clubs', 'Diamonds', ...]

Readonly

```
type StatusCodeType = {
    ok: number,
    "Not found": number,
    "Internal Server Error": number
}

type ImmutableStatusCodeType2 = Readonly<StatusCodeType>
```

Utility types

- Utility types are standard type manipulations implemented by TypeScript
- Readonly<{ a: number }> == { readonly a: number }
- They *look* just like generics, but they aren't
- They are a form of type helpers
- Think of them as functions that takes types and return new types

Partial

```
type Employee = {
    name: string,
    age: number,
    salary: number
}

const e1: Employee = {
    name: 'Donald Duck',
    age: 33
}
```

```
const e2: Partial<Employee> = {
    name: 'Donald Duck',
    age: 33
}
```

Pick and Omit ~ For object types work on properties

```
const e3: Pick<Employee, "name" | "age"> = {
    name: 'Donald Duck',
    age: 33
}

const e4: Omit<Employee, "salary"> = {
    name: 'Donald Duck',
    age: 33
}
```

Extract and Exclude ~ For union 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 = Extract<State, {status: 'failed' | 'ok'}>

type FinishedState2 = Exclude<State, {status: 'loading'}>
```

Combining utility types

```
const statusCodes: Readonly<Record<string, number>> = {
    "ok": 200,
    "Not found": 404,
    "Internal Server Error": 500
}
```

Type manipulations

- Creating types from other types
- keyof creates a union of the property keys of an object type
  - keyof {n:number, s: string} === 'n' | 's'
- Index signatures creates an object type from other types
  - Exactly like a Record
- String type manipulations for unions of string types
  - Template literal types

keyof

```
const constStatusCodes = {
    ok: 200,
    "Not found": 404,
    "Internal Server Error": 500
} as const

type StatusCodes = typeof constStatusCodes
type StatusCodeKeys = keyof StatusCodes
```

Using keyof with generics

```
function objectKeys<T extends {}>
  (obj: T): Array<keyof T> {
    return Object.keys(obj) as Array<keyof T>
}
```

```
function getter<T extends {}, K extends keyof T>
  (obj: T, k: K): () => T[K] {
    return () => obj[k]
}
```

Index signatures

```
type Keys = 'ok' | 'Not Found' | 'Internal Server Error'

type StatusCodes = {
    readonly [key in Keys]: number
}

type StatusCodesHandler = {
    readonly [key in keyof StatusCodes]: (code: StatusCodes[key]) => void
}
```

Type Helpers

- Type helpers look like generic types, but they are not
- They are a kind of type function: They take types and returns new types
- Like generic types you can put type constraints ("extends")
- This is how the utility types are made

Type Helper Example

```
type Species = 'Dog' | 'Cat'

type Dog = 'Boxer' | 'Husky' | 'German Shepard'
type Cat = 'Siamese' | 'Persian' | 'Manx'


type Annotated<S extends Species, R extends string> = `${S}: ${R}`


type Animal = Annotated<'Dog', Dog> | Annotated<'Cat', Cat>
```

Conditionals in type helpers

- Since type helpers are functions, we might need conditionals
- Conditionals have the form

```
SomeType extends OtherType? TrueType : FalseType
```

- The types in the expression can be any type expression
- Example:

```
type PrimitiveArray<T> =
  T extends number | string | boolean ? T[] : never
```

- If the type is a primitive make an array, otherwise don't bother

Type helper with conditionals

```
type PrimitiveArray<T> =
  T extends number | string | boolean ? T[] : never

type A = PrimitiveArray<string | number | Object>

type B = PrimitiveArray<boolean>
```

Distributive conditionals

PrimitiveArray<string | number | Object>

= Primative<string> | PA<number> | PA<Object>

= string[] | number[] | never

= string[] | number[] ≠ (string | number)[]

Type inference in conditionals

```
type FieldType<T, K extends string | symbol | number> =
  T extends { [key in K]: infer U }? U : never

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

type State = LoadingState | FailedState | OkState

type X = FieldType<State, 'status'>   ~ = State['status']
type Y = FieldType<State, 'statusCode'>
```