# TYPESCRIPT

# About TypeScript

- Developed by Microsoft
  - first public release 2012
- Superset of JavaScript
- Adds static type checking
  - compiles to JavaScript
- is turing complete
  - https://github.com/microsoft/TypeScript/issues/

# WHY TYPESCRIPT

- Prevent runtime errors
- self documenting code
- IDE support
  - code completion
  - automated refactoring
- generate API docs from types
- easier code lookup

# Optional Type System

```
let someValue: number = 10
someValue = '10'
// ^^^^^
// Type '"10"' is not assignable to type 'number'.
```

# Data Types in TS

- any
- primitives
  - number
  - boolean
  - string
  - Array
  - Tuple

# Number

- Same data type as in JavaScript
- floating point numbers
  - supports decimal, hex, octal

```
let decimal: number = 6;
let hex: number = 0xf00d;
let binary: number = 0b1010;
let octal: number = 0o744;
```

# Boolean

- Simple true/false values

```
let isDone: boolean = false;
let trueOnly: true = true;
let falseOnly: false = false;

trueOnly = false
// ^^^^^^^^^^^^^
// Error: Type 'false' is not assignable to type 'true'
```

# STRING

- same as in JS

```
let color: string = "blue";
color = "green";

color = 10;
// ^^^^^^^^
// Error: Type '10' is not assignable to type 'string'
```

# Any

- opt out of type checking
  - might be useful when a type is unknown during development
  - eg. dynamic content from a 3rd party API
- used to gradually adapt TypeScript
  - is sometimes misused
  - might be valuable for generics

# ANY

```
let notSure: any = 4;
notSure = "maybe a string instead";
notSure = false;
```

# Arrays

- two ways to specify arrays
  - short notation string[]
  - as generic Array<string>

```
let colors1: string[] = ['green', 'blue'];
let colors2: Array<string> = ['green', 'blue'];
```

# Tuple

- express array with fixed number of elements
  - eg. vectors

```
let vector: [number, number] = [1, 2];

// Declare a tuple type
let x: [string, number];
// Initialize it
x = ["hello", 10]; // OK
// Initialize it incorrectly
x = [10, "hello"]; // Error
```

# Interfaces

- focus on the shape of types (eg.: duck typing)
  - if it quakes like a duck it is considered a duck

```
type User = {
  name: string,
}

const printUserName = (user: User) => console.log(user.name)
const user = {
  name: 'Sepp',
  age: 55
}
printUser(user)
//       ^^^^^^
// No compile error even if the `user` contains the
// property `age` which is not part of the User interface.
```

# OPTIONAL VALUES 1

- not all properties of interfaces might be required
- TS allows to specify optional properties

```
type User = {
  name: string,
  age?: number
//   ^
// makes the property optional
}

const userWithoutAge: User = {
  name: 'Sepp',
};
```

# Optional Values 2

» Optional values are type checked

```
type User = {
  name: string,
  age?: number
}

const printUserAge = (user: User) => {
  return console.log(user.age.toString())
  //     ^^^^^^^^
  // Error: Object is possibly 'undefined'
}
```

# Unions

- a type in typescript can be of more than one type
  - this is called a union and a pipe | is used
- the | could be seen as an 'or' operator

```
type AnonymousUser = { id: number }
type RegisteredUser = { id: number, email: string, password: string }
type User = AnonymousUser | RegisteredUser
//                        ^^^
// A user is either an AnonymousUser OR a RegisteredUser
```

# Literals

- literal is subset of a more generic type
  - eg: "Hello MMT" is a string, but not every string is "Hello MMT"
  - allows to narrow types
- can be used to express constants
  - makes readability of code more expressive

# Literals

```
type Lecture =
  | 'Fullstack Development'
  | 'Client side engineering'
  | 'Software Quality Assurance'

const assignToLecture = (email: string, lecture: Lecture) => { /* irrelevant */ }

assignToLecture('sepp@fh-salzburg.ac.at', 'Fullstack Development')
assignToLecture('sepp@fh-salzburg.ac.at', 'HCI')
//                                        ^^^^^
// Argument of type '"HCI"' is not assignable to parameter of type 'Lecture'.
```

# Generics

- Task build an identity function
    - identity function returns the same value that it was given as argument

```
const identityInJS = (arg) => arg
```

# Generics

- Possible implementation in TS

```
const stringIdentity = (arg: string) => arg
const numberIdentity = (arg: number) => arg
const booleanIdentity = (arg: boolean) => arg
// ...
```

# Generics

- Possible implementation with any

```
const identity = (arg: any) => arg
const value = identity(1)
value.toUpperCase()
//   ^^^^^^^^^^^^^^
// Will throw `value.toUpperCase is not a function`
```

# GENERICS

- possibility to reuse types with other types
- written in angle brackets <NameOfTypeVariable>
- I like to see them as:
  - type level functions

```
const identity = <T>(arg: T): T => arg
//             1)^^^    2)^  3)^
// 1) Define a type argument T
// 2) type of arg is assigned to the type argument T
// 3) the function returns the type argument T
```

# GENERICS

```
const identity = <T>(arg: T): T => arg

const stringValue1 = identity<string>('Hallo MMT')
//                                   ^^^^^^^^
// explicitly assign string as the type argument to the identity function
// string value1 will be of type string

const stringValue2 = identity('Hallo MMT')
// typescript automatically infers type of the type argument T
// string value2 will be of type string
```

# Generics for Records

- » Generics can be used to compose

```
type ServerResponse<ResponsePayload> = {
//                 ^^^^^^^^^^^^^^^^
// Define a type variable called `ResponsePayload`
  payload: ResponsePayload
// Use the type variable `ResponsePayload`
}

type UserResponse = ServerResponse<{ name: string }>
// resulting type: { payload: {name: string} }
```

# Utility Types

- `Pick<T>`
- `Omit<T>`
- `Partial<T>`
- `Required<T>`
- `Arguments<T>`
- `ReturnType<T>`
- `Readonly<T>`

# PICK<T>

- Creates a subset of an existing record type
- Allows to specify a list of properties to extract
- similar to lodash pick, but on a type level

```
type User = {
  firstName: string,
  lastName: string,
  age: number
}

type UserName = Pick<User, 'firstName' | 'lastName'>
// type will be { firstName: string, lastName: string }
```

# Omit<T>

- Creates a subset of an existing record type
- Allows to specify a list of properties to remove
- similar to lodash omit, but on a type level

```
type User = {
  firstName: string,
  lastName: string,
  age: number
}

type UserName = Omit<User, 'age'>
// type will be { firstName: string, lastName: string }
```

# PARTIAL<T>

- makes all properties of an object optional
- might be used to overwrite default configs etc.

```
const defaultTSConfig = {
  target: 'ES2015',
  declaration: true,
}
type TSConfig = typeof defaultTSConfig
//              ^^^^^^
// automatically infer the type of defaultTSConfig
// TSConfig will be of type { target: string, declaration: boolean }

const start = (options: Partial<TSConfig>) => { /* irrelevant*/ }
//                      ^^^^^^^^^^^^^^^^^
// options will be of type { target?: string, declaration?: boolean }

start({ target: 'es5' })
start({ declarations: false })
```

# REQUIRED<T>

- opposite of partial
- makes all values of a record required

```
type User = Required<{
  name: string,
  age?: number
}>
const user: User = { name: 'Sepp' }
//    ^^^^
// Error: Property 'age' is missing in type
```

# ARGUMENTS<T>/RETURNTYPE<T>

» Argument<T> returns the type of the arguments of a function

» ReturnType<T> returns the return type of a function

```
const add = (a: number, b: number) => a + b
type AddArguments  = Arguments<typeof add> // [number, number]
type AddFirstArgument  = AddArgument[0]
type AddSecondArgument  = AddArgument[1]

type AddReturnType = ReturnType<typeof add> // number
```

# Access a subtype in object

```
type User = {
  firstName: string,
  lastName: string,
  age: number
}

type Age = User['age'] // => type: number
```

# UNWRAP ARRAY

```
type MyArray = number[]

type ArrayItem = MyArray[number] // => type: number
```

# Type Inference

- automatic deduction of a type from an expression

```
let mutableValue = 10 // => type number
const constantValue = 10 // => type 10
```

# TYPE INFERENCE
## GENERICS

» Some generics can be inferred automatically

```
const numberArray = [0,1,2,3] // => Array<number>
const stringArray = ['A','B','C','D'] // => Array<string>
const booleanArray = [true,false] // => Array<boolean>
const mixedArray = [1, 'A', true] // Array<number | string | boolean>
```

# Type Inference

## const Assertions

- JS values are mutable
- JS value can be altered despite being defined as const

```
const numberArray = [0,1,2,3] // => type Array<number>
numberArray[0] = 10;
```

# Type Inference
## const assertions

- const assertion mark a value as immutable
- type is narrowed

```
const numberArray = [0,1,2,3] as const // => type readonly [0, 1, 2, 3]
numberArray[1] // => type 1
```

# Type Inference
## const assertions objects

- const assertion works on objects as well

```
const myObject = { a: 1, b: 'one' } as const
// => type readonly { readonly a: 1; readonly b: "one" }

numberArray[b] // => type 'one'
```

# React and TypeScript

- » React integrates with TypeScript
- » can replace prop types from React

```
type AvatarsProps = {
  images: string[]
}

const Avatars = (props: AvatarsProps) => (
    <div className={css(styles.wrapper)}>
      { props.images.map((imageUrl) => (
        <img key={imageUrl} src={imageUrl} className={css(styles.image)} />
      ))}
    </div>
)

<Avatars images={[1,2,3]} />
//                ^^^^^^
// Error: number is not assignable to string
```

# Exercise create a user type:

- Requirements:
  - a user needs to have a first and last name
  - a user needs to have exactly one contact
  - a contact is either:
    - address (contains street/zip code/country)
    - phone (contains phone)
    - email (contains email)

# Exercise Create a User Type

## Possible Solution

```
type User = {
  firstName: string,
  lastName: string,
  street?: string,
  zipCode?: string,
  country?: string,
  isAddressVerified?: bool,
  email?: string,
  isEmailVerified?: bool,
  phone?: string,
  isPhoneVerified?: bool,
}
```

# Exercise create a User type

## Can you spot issues with this model?

```
type User = {
  firstName: string,
  lastName: string,
  street?: string,
  zipCode?: string,
  country?: string,
  isAddressVerified?: bool,
  email?: string,
  isEmailVerified?: bool,
  phone?: string,
  isPhoneVerified?: bool,
}
```

# EXERCISE CREATE A USER TYPE

## CAN YOU SPOT ISSUES WITH THIS MODEL?

```
const user = {
  firstName: 'Sepp',
  lastName: 'Dupfinger',
  street: 'Hinterholz 8',
};
```

# Exercise Create a User Type

## Issues

- 1 correct state and 8 falsy states

```
type User = {
  // ...
  street?: string,
  zipCode?: string,
  country?: string,
  // ...
}
```

# Exercise Create a User Type

## Requirements

- A user needs to have a first and last name
- A user needs to have exactly one contact
  - a contact is either:
  - address (contains street/zip code/country)
  - phone (contains phone)
  - email (contains email)
  - a contact can be verified

# Exercise Create a User Type

## Classify the type

```
type User = {
  firstName: string,
  lastName: string,

  // via post
  street?: string,
  zipCode?: string,
  country?: string,
  isAddressVerified?: bool,

  // via email
  email?: string,
  isEmailVerified?: bool,

  // via phone
  phone?: string,
  isPhoneVerified?: bool,
}
```

# EXERCISE CREATE A USER TYPE

## EXTRACT SMALLER BITS

```
type PostContact = { street: string, zipCode: string, country: string, isVerified: bool }
type EmailContact = { email: string, isVerified: bool }
type PhoneContact = { phone: string, isVerified: bool }
type Contact = PostContact | EmailContact | PhoneContact

type User = {
  firstName: string,
  lastName: string,
  contact: Contact,
}
```

# EXERCISE CREATE A USER TYPE

## EXTRACT COMMON PROPERTIES

```
type Verifiable<T> = T & { isVerified: boolean }

type PostContact = Verifiable<{ street: string, zipCode: string, country: string }>
type EmailContact = Verifiable<{ email: string }>
type PhoneContact = Verifiable<{ phone: string }>
type Contact = PostContact | EmailContact | PhoneContact

type User = {
  firstName: string,
  lastName: string,
  contact: Contact,
}
```

# Use it

```
const user:User = {
  firstName: 'Sepp',
  lastName: 'Dupfinger',
  contact: { email: 'sepp@hinterholz.at', isVerified: true },
}
```

# “Can you still spot issues with this model?”

```
const user:User = {
  firstName: '',
  lastName: '',
  contact: { email: '', isVerified: true },
}
```

# Type Aliases

```
type Email = string
```

# Verifying Type Aliases

```
type Maybe<T> = T | null
type Email = string

const validateEmail = (maybeEmail: unknown): Maybe<Email> => {
    if (typeof maybeEmail === 'string' && maybeEmail.match(/.@./)) {
        return maybeEmail as Email;
    }
    return null;
}
```

# Feedback

- Questions: tmayrhofer.lba@fh-salzburg.ac.at
- https://s.surveyplanet.com/x1ibwm85