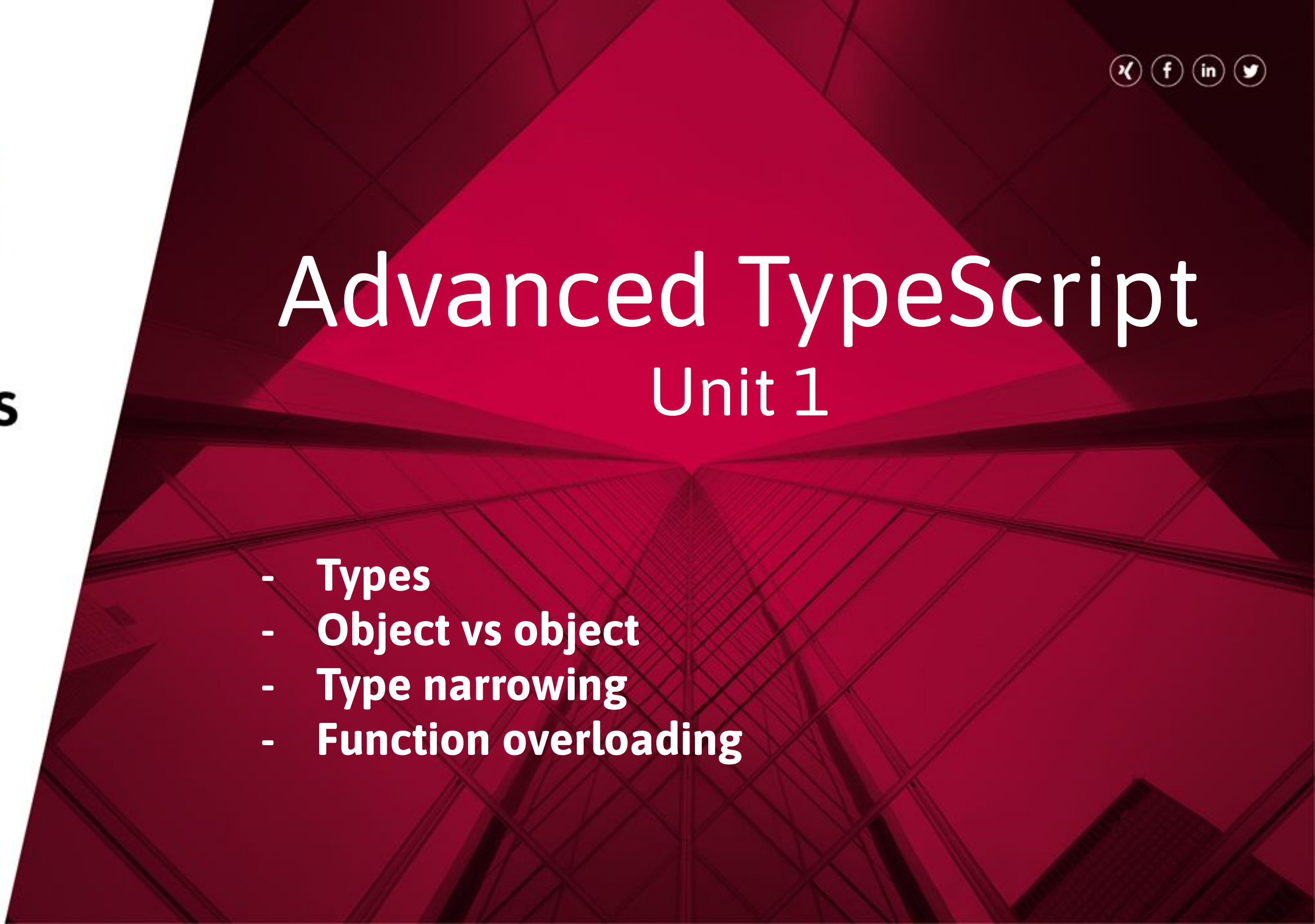
ANGULAR
ARCHITECTS
INSIDE KNOWLEDGE



Advanced TypeScript

Unit 1

- Types
- Object vs object
- Type narrowing
- Function overloading

Data types ECMAScript

- primitives
  - **boolean, number, string**
  - **undefined, null**
  - **symbol, bigint**
- **object**
  - object literal
  - class instances
  - arrays
  - functions

Primitives vs. Classes

- Every primitive type has an equivalent class
- Class is a wrapper with methods and properties
- Classes are rarely used directly
  - Exception: Boolean
- Mutability!
- Auto-Boxing

type & interface

- type and interface almost identical
- type is locked, once defined
    - No redeclaration possible
    - Most used for aliases of types ;)
- Interfaces
    - can be extended
    - object shapes
    - Faster "not significantly"
        - https://github.com/microsoft/TypeScript/wiki/Performance

type & interface

```
interface IBasket {
  milk: number;
  apples: number;
}

type TBasket = {
  milk: number;
  apples: number;
};

const basket1: IBasket = { milk: 0, apples: 1 };
const basket2: TBasket = { milk: 0, apples: 1 };

function shop(basket: { [key: string]: number }) {}

shop(basket1); // fails
shop(basket2);
```

type & interface

```
interface IPerson {
  firstname: string;
  age: number;
}

type TPerson = {
  firstname: string;
  age: number;
};

interface IPerson {
  country: string;
}

// type TPerson = {country: "string"}

const franz: IPerson = { firstname: "Franz", age: 12, country: "AT" };
// franz.lastname = "Müller";
const zoran: TPerson = { firstname: "Zoran", age: 40 };
```

Structural typing

Type compatibility depends on members not the type or inheritance itself

Structural typing

```
type City = {
  name: string;
  country: string;
  isCapital: true;
}

type Town = {
  name: string;
  country: string;
}

declare function add(town: Town): void

const city: City = {name: "Vienna", country: "Austria", isCapital: true};
const town: Town = {name: "Brunn", country: "Austria"};
const london = {name: "London", country: "UK", language: "english"}

add(town);
add(city);
add(london)
```

Structural typing - Careful with object literals

```
type City = {
  name: string;
};
declare function add(city: City): void;

const paris = { name: "Paris", country: "France" };

add(paris);
add({ name: "Paris", country: "France" }); // fails to compile
```

Object types craziness 1/3 🤪

- `Object`
  - superclass of objects
  - e.g.: toString(), hasOwnProperty(), protoype
  - not null or undefined
  - used as an interface or its static methods (freeze, assign, create)
- `object`
  - Non-primitive
  - TypeScript only

Object types craziness 2/3 🤪

- `{[key: string]: unknown}`
  - like `object` but stricter (value is not any)
  - used for index based access
  - key can only be of type `string, number, symbol`
- `Record<string, unknown>`
  - utility type
  - more elegant version of `{[key: string]: unknown}`
  - allows union types for key
- `Map`
  - no restrictions in terms of key type

ANGULAR ARCHITECTS
INSIDE KNOWLEDGE

Object types craziness 3/3 🤪

[Difference between 'object' ,{} and Object in TypeScript - Stack Overflow]()

Type assertion

- Manually defines a type
- "Compile time casting"
- Limited to specialisation or generalisation
- Try to avoid them, except
  - Untyped libraries
  - DOM methods
  - Tricky situations
- Type narrowing should be favoured

Type assertion

```
class Animal {
  eat() {}
}

class Human extends Animal {
  speak() {}
}

class Car {
  drive() {}
}
```

```
function a(animal: Animal) {
  const human = animal as Human;
  // const human: Human = animal;
  human.speak();
}

function b(human: Human) {
  const animal = human as Animal;
  animal.eat();
}

function c(human: Human) {
  // const car = human as ar;
  // const car = human as object as Car;
  // const car = human as number as Car;
  const car = human as unknown as Car;
  car.drive();
}
```

Common TypeScript types

- ECMAScript types
  - **boolean, number, string, undefined, null, symbol, bigint**
- **any**
- **unknown**
- **never**
- **void**
- **enums**
- **union and intersection type**
- **literal type**

any vs. unknown

- both for values, we don't know
- any without type-safety
- unknown with type-safety
- try to avoid any
- type narrowing required for unknown

```
function unsafe(value: any) {
  return value.toUpperCase();
}

function safe(value: unknown) {
  return value.toUpperCase(); // 👎
  if (typeof value === "string") {
    return value.toUpperCase();
  }
}
```

never

- Elimination of properties
- Type-safe switch cases / branches
- Impossible properties on intersecting

```
type DIRECTION = "up" | "down" | "left" | "right" |
"stay";

function move(direction: DIRECTION) {
  switch (direction) {
    case "down":
      return [0, -1];
    case "up":
      return [0, 1];
    case "left":
      return [-1, 0];
    case "right":
      return [1, 0];
    default:
      // fails because we missed "stay"
      const exhaustCheck: never = direction;
  }
}
```

Intersection

- merge types together
- collision behaviour:
  - properties are never
  - functions are overloaded

```
type Person = {
  id: number;
  lendMoney(amount: number): void;
  name: string;
};

type Country = {
  id: bigint;
  lendMoney(amount: bigint): void;
  name: string;
};

type Citizen = Person & Country;

function check(citizen: Citizen) {
  const id = citizen.id; // 👎
  const name = citizen.name; // 👍
  citizen.lendMoney(100); // 👍
  citizen.lendMoney(100_000n); // 👍
}
```

Type narrowing

Type-safe reduction of a multiple types to a single type

ANGULAR
ARCHITECTS
INSIDE KNOWLEDGE

Type narrowing: use cases

- Untyped libraries
- Union types
- Generic functions
- Nullable types
- Class hierarchy

Type narrowing: possibilities

- type guards (typeof)
- `instanceOf`
- discriminated unions
- `in` operator
- type predicates / assertion functions

- truthiness (mainly for undefined | null)
- equalness
- type inference via assignment
- built-in functions like `Array.isArray()`
- control flow analysis

- ~~type assertion~~

Type narrowing with: `typeof` for primitives

- returns type for primitives
  - expect for null it returns "object"
- returns "function" for functions and classes
- returns "object" for everything else
- union type of

```
"string" | "number" | "bigint" | "boolean" |
"symbol" | "undefined" | "object" | "function"
```

Type narrowing: `instanceOf` for class instances

```
class Person {
  constructor(public firstname: string, public lastname: string) {}
}

type Country = {
  name: string;
  code: string;
};

function print(value: unknown) {
  if (value instanceof Date) {
    return value.toISOString();
  }
  if (value instanceof Person) {
    return `${value.firstname} ${value.lastname}`;
  }

  if (value instanceof Country) {}  // 👎
}
```

Type narrowing: discriminator

```
type Age = { birthdate: Date; type: "age" };
type Person = { birthday: Date; type: "person" };


function getDate(value: Person | Age): Date {
  if (value.type === "age") {
    return value.birthdate;
  } else {
    return value.birthday;
  }
}
```

Type narrowing: in operator

```
type Age = { birthdate: Date };
type Person = { birthday: Date };

function getDateViaIn(value: Person | Age): Date {
  if ("birthdate" in value) {
    return value.birthdate;
  } else {
    return value.birthday;
  }
}
```

Type narrowing: type predicates

```
type AgeJson = {birthday: string;};
type Age = {birthday: Date;};

function isAgeJson(value: Age | AgeJson): value is AgeJson {
  return value.birthday instanceof Date;
}

function getDate(value: AgeJson | Age): Date {
  if (isAgeJson(value)) {
    return parse(value.birthday, "dd.MM.yyyy", new Date());
  } else {
    return value.birthday;
  }
}
```

Type narrowing: assertion function

```
function assertTruthy(
  value: string | Date | null | undefined
): asserts value is string | Date {
  if (!value) {
    throw new Error("null or undefined not allowed");
  }
}
```

Function Overloading

Different types/amount of arguments,

different return types

under the same function name

Function Overloading

- Multiple overloading signatures
- One implementation
  - Must fulfill all overloading signatures
  - Extensive use of union types
  - Cannot be called directly from the outside
- Order is important, implementation right after the overloads
- Careful if union types should be allowed
- Advanced use cases with conditional types

Function Overloading

```
function calcAge(birthday: string): number;
function calcAge(birthday: string, reference: string): number;
function calcAge(birthday: Date): number;
function calcAge(birthday: Date, reference: Date): number;

function calcAge(birthday: Date | string, reference?: Date | string): number {
  if (birthday instanceof Date) {
    if (typeof reference === "string") {
      throw "failure";
    }
    let now = reference ?? new Date();
    return now.getTime() - birthday.getTime();
  } else {
    if (reference instanceof Date) {
      throw "failure";
    }
    let now = reference ? new Date(reference) : new Date();
    return now.getTime() - new Date(birthday).getTime();
  }
```