Advanced TypeScript

What are we going to cover?

Advanced Types

Decorators

Async-Await

Using noImplicitAny, noImplicitThis and strictNullChecks

Advanced Types

The **unknown** and **never** types

Type Aliases

- The **type** keyword creates an alias for a type
- Very similar to an interface but more flexible in some cases

Intersection Types

- The result is a type that combines all listed types

Union Types

- The result it a type that is either one of the listed types

The unknown type

The **unknown** type is the type-safe counterpart of **any**

- Anything can be assigned to unknown
- But it be must be cast before it can be used

Better than any because it is more restrictive

- An unknown type must be cast before it can be used

New in TypeScript 3

The never type

Never represents a type that can never occur

- No type, other than never, can be assigned to a variable of type never

Very useful for **exhaustive checks**

- Make sure you handle each possible case the compiler knows

Also useful for functions that **never return**

- Because it always throws an error

Exhaustive check

```
type Animal = "dog" | "cat" | "fish";

function printAnimal(animal: Animal) {
  switch (animal) {
    case "dog":
      console.log("It's a dog");
      break;
    case "cat":
      console.log("It's a cat");
      break;
    default:
      // Error: Type '"fish"' is not assignable to type 'never'.
      const shouldNotHappen: never = animal;
      throw new Error(`Unknown animal: ${animal}`);
  }
}
```

Type alias

Type aliases create a new name for a type

Can also be used with **generics**

Type alias

```
type Cat = {
  sleep();
  miauw();
};

type Dog = {
  sleep();
  bark();
};

type CatOrDog = Cat | Dog;
let pet: CatOrDog;
pet.sleep();
```

Intersection & Union Types

Create new types based on other existing types

- An intersection type combines multiple types into one
- A union type describes a value that can be one of several types

Intersection & Union Types

```
type CatAndDog = Cat & Dog;

let animal1: CatAndDog;
animal1.name;
animal1.bark();
animal1.sleep();

type CatOrDog = Cat | Dog;

let animal2: CatOrDog;
animal2.name;
// Property 'bark' does not exist on type 'CatOrDog'
// animal2.bark();
```

Tagged unions

Allows for a number of valid types to be combined.

Each type requires a kind property to differentiate between them.

- Has to be a string literal

Inside a **switch** statement on the differentiator the compiler knows the correct type!

Tagged unions example

```
class Car { model: 'Car’ = 'Car'; drive(){} }

class Plane { model: 'Plane’ = 'Plane'; fly(){} }

type Vehicle = Car | Plane;

function move(vehicle: Vehicle) {
  switch (vehicle.model) {
    case 'Plane':
      // vehicle.drive();
      // error TS2339: Property 'drive' does not exist on type 'Plane'.
      vehicle.fly();
      break;
    case 'Car':
      vehicle.drive();
  }
}
```

Type Guards

A type guard is a function that returns a **type predicate**

- The result is typed as: **parameter is Type**

Type guards are used **both at run- and compile time**

- The compiler knows that a type assertion will always be true at runtime

Use property **checks**, **typeof** and **instanceof** as needed

Very useful with union types

Type Guards

```
type Cat = {
  name: string;
  sleep(): void;
};

function isCat(pet: any): pet is Cat {
  return typeof pet.sleep === "function";
}

function doPetStuff(pet: any) {
  if (isCat(pet)) {
    pet.sleep();
  }
}
```

Mapped types

Mapped types are types that are based of other types

Create **new types** that are similar to the original

- Make all properties optional or read-only

Create a string literal type with all **key names** of a type or object

- Using the typeof and keyof

Mapped types

```
function printText(options: Partial<typeof defaults>) {
  const defaults = {
    text: 'Some message',
    count: 1
  };

  const actual = { ...defaults, ...options };
  for (let index = 0; index < actual.count; index++) {
    console.log(actual.text);
  }
}

printText({ text: 'Hello there' });
printText({ count: 5 });
// Error: Object literal may only specify known properties
// printText({ now: true });
```

Mapped types

Make all properties in T read-only
- Readonly<T>

Make all properties in T optional or required
- Partial<T>
- Required<T>

From T pick a set of properties K
- Pick<T, K>:

Construct a type with a set of properties K of type T
- Record<K, T>

Exclude from T those types that are assignable to U
- Exclude<T, U>

Extract from T those types that are assignable to U
- Extract<T, U>

Using a read-only type

```
interface Person {
  firstName: string;
  lastName: string;
}

type ReadOnlyPerson = Readonly<Person>;

function printPerson(person: ReadOnlyPerson) {
  console.log(`${person.firstName} ${person.lastName}`);
}

const person: Person = {
  firstName: "Maurice", lastName: "de Beijer"
};
printPerson(person);
```

More mapped types

Obtain the parameters of a function type in a tuple

- Parameters<T>

Obtain the return type of a function type

- ReturnType<T>

Obtain the parameters of a constructor function type in a tuple

- ConstructorParameters<T>

Extracting types from a function

```
function getFullName(person:
  { firstName: string; lastName: string }) {
  const name = `${person.firstName} ${person.lastName}`;
  return name;
}

type Person = Parameters<typeof getFullName>[0];
type FullName = ReturnType<typeof getFullName>;

const person: Person = {
  firstName: 'Maurice',
  lastName: 'de Beijer'
};

const fullName: FullName = getFullName(person);
```

Decorators

With **decorators** you can annotate TypeScript classes and their members.

- They are used a lot with Angular development
- It's just a function which is passed the **class**, **property** and a **descriptor** as parameters

Decorator **factories** can be used when a decorator needs to be parameterized.

- Just a function that returns the actual decorator function

Note: Decorators are not yet standardized in ECMAScript and may change.

- They require the **--experimentalDecorators** command line option

Creating a decorator

```
function log(target: any,
  key: string,
  descriptor: PropertyDescriptor) {

  const original = target[key];

  target[key] = function (...args) {
    console.log(`=> ${key}(${args}).`);
    original.call(this, ...args);
  }
  return target;
}
```

Using the decorator

```
class Cat {
  constructor(private name: string) {}

  @log
  eat(food) {
    console.log(
      `${this.name} is eating ${food}.`);
  }
}

const zorro = new Cat('Zorro');
zorro.eat('meat');
```

Async-Await

Using **async** and **await** makes writing asynchronous code much easier.

- Instead of using callbacks and nested functions code can be written like synchronous code

Every function that returns a **promise** can be awaited.

- This must be done in a function marked as async

The feature is based on the C# async/await feature.

Async-Await example

```
async function getMovies() {
  var rsp = await fetch('./movies.json')
  var movies = await rsp.json();
  console.table(movies);
}

getMovies();
```

noImplicitAny

Ensures all variables are declared or resolved to a known type.

- Resolving to **any** is a frequent cause of TypeScript not catching errors

Prevents accidental usage of the **any** type.

- The **any** type can still be used explicitly where needed

noImplicitAny example

```
function add(x: number, y: number) {
  return x + y;
}

// error TS7006:
// Parameter 'y' implicitly has an 'any' type.
function subtract(x: number, y) {
  return x - y;
}
```

noImplicitThis

The type of the **this** variable is not always known and can be inferred as **any**.

This flag causes compiler errors when this is the case.

- Explicitly declare **this** in the function parameters

noImplicitThis example

```
var zorro = {
  name:'Zorro',
  eat(this: {name: string}, food: string) {
    console.log(
      this.name, 'is eating', food);
  }
};

zorro.eat('meat');
```

strictNullChecks

With **strictNullChecks** enabled the compiler checks and complains about potential **null** or **undefined** references.

- Dereferencing null or undefined is one of the most frequent runtime errors

Declaring a type as **any** still allows for **null** and **undefined**.

strictNullChecks

```
class Cat {
  constructor(public name: string) {}
}

function getCat(name) {
  if (name) return new Cat(name);
  return null;
}
function printCat(cat: Cat | null) {
  // At runtime: Uncaught TypeError:
  // Cannot read property 'name' of undefined
  console.log(cat.name);
}
// error TS2322:
// Type 'Cat | null' is not assignable to type 'Cat'.
var zorro: Cat = getCat('');
printCat(zorro);
```

Conclusion

Use the advanced types

- Makes the type system much more powerful

Use Async-Await where appropriate

- It makes writing and reading asynchronous code much easier

Use checks like noImplicitAny, noImplicitThis and strictNullChecks

- They help catch a lot of possible logic errors and missing type declarations