ANGULAR
ARCHITECTS
INSIDE KNOWLEDGE

Advanced TypeScript

Unit 2

- typeof & keyof
- Indexed access types
- Generics
- Variance
- Utility types

typeof

- Generates a new type (not value) out of a variable
- Extension of ECMAScript's typeof for TypeScript's Type
- Useful in mapped types or utility types

```
const word = "hello";
const five = 5;

const franz: Person = { id: 1, firstname: "franz" };

type WordType = typeof word; // string
type FiveWord = typeof five; // number
type Franz = typeof franz; // Person

type A = typeof true; // fails
```

ANGULAR ARCHITECTS
INSIDE KNOWLEDGE

keyof

- Generates a **union type** (string | number) of an object's keys
- Is a type and not a value
- Like typeof used in combination with other type manipulations
  - typesafe access to properties in Generics
  - critical for mapped types

```
type A = Record<string, unknown>;
type B = Record<number, unknown>
type C = {id: number, name: string};

type AProps = keyof A; // string
type BProps = keyof B; // number

type Keys = keyof C; // 'id' | 'name'
const keys = keyof C; // fails to compile
```

Index Access Type

- Derived type or types from other types' properties
- Applicable to everything
- Type-safe way to access a property via its index
- Used in combination with generic functions

ANGULAR
ARCHITECTS
INSIDE KNOWLEDGE

Index access type

```
type Person = {
  id: number;
  firstname: string;
};

const franz: Person = { id: 1, firstname: "Franz" };

type PersonId = Person["id"];
const personId: PersonId = franz["id"];
```

"id" is a type (type id = "id")

number

Index access type

```
type PersonPropertyTypes = Person[keyof Person] // string | number

function readProperty(person: Person, property: keyof Person) {
    return person[property];
}

const id: number = readProperty(franz, 'id'); // does not compile yet (generics needed)
```

Generics

- Makes a single function usable
  - for different and "unknown" types
  - stays fully type-safe
- Generic types (e.g. utility types)
- Generic classes (e.g. Observable, Promise, Map)
- Must-have skill for library developers
- Used in shared parts of application code

ANGULAR
ARCHITECTS
INSIDE KNOWLEDGE

Common variations

- None (value is not returned)
- Simple
- External processing
- Constrained types for direct processing
- Multiple types
- Derived type
- Type which itself is a generic

Generics 1: Type is passed on

```
function log<T>(value: T): T {
  console.log(value);
  return value;
}

people.map(log).reduce((names, curr) => `${names}, ${curr.lastname}`, "");
cities.map(log).reduce((names, curr) => `${names}, ${curr.name}`, "");
```

ANGULAR
ARCHITECTS
INSIDE KNOWLEDGE

Generics 2: "externally processed"

```
function log<T>(value: T, format: (value: T) => string): T {
  console.log(format(value));
  return value;
}

people
  .map((person) => log(person, (person) => person.lastname))
  .reduce((names, curr) => `${names}, ${curr.lastname}`, "");
cities
  .map((city) => log(city, (city) => city.name))
  .reduce((names, curr) => `${names}, ${curr.name}`, "");
```

Generics 3: "internally processed", aka. constrained

```
function log<T extends { isLogged: boolean }>(
  value: T,
  format: (value: T) => string
): T {
  if (!value.isLogged) {
    console.log(format(value));
  } else {
    value.isLogged = true;
  }

  return value;
}

people
  .map((person) => ({ ...person, isLogged: false }))
  .map((person) => log(person, (person) => person.lastname))
  .reduce((names, curr) => `${names}, ${curr.lastname}`, "");
```

Generics 4: multiple types

```
function log<T, U>(value: T, processor: (value: T) => U): U {
  console.log(value);
  return processor(value);
}

const processedFranz = log(franz, (person) => ({
  ...person,
  processedDate: new Date(),
}));
```

Generics 5: derived type

```
function unsafeLog<T extends Record<string, unknown>>(
  value: T,
  property: string
) {
  if (property in value) {
    console.log(value[property]);
    return value[property];
  }

  throw new Error(`${property} does not`);
}

function safeLog<T, P extends keyof T>(value: T, property: P) {
  console.log(value[property]);
  return value[property];
}

const lastname = unsafeLog(franz, "lastName"); // unknown
const id: number = safeLog(franz, "id"); // number
```

ANGULAR
ARCHITECTS
INSIDE KNOWLEDGE

Generic Class

```
class MyMap<Key extends { toString: () => string }, Value> {
  #store: Record<string, Value> = {};

  get(key: Key) {
    return this.#store[key.toString()];
  }

  put(key: Key, value: Value) {
    this.#store[key.toString()] = value;
  }
}

const map = new MyMap<number, string>();
map.put(5, "five");

const numberName: string = map.get(5);
```

Generic Type: basis for unit 3

```
type Person = {
  id: number;
  firstname: string;
};

type Loggable<T> = T & { format: () => string };

type LoggabelPerson = Loggable<Person>;

const franz: LoggabelPerson = {
  id: 1,
  firstname: "franz",
  format() {
    return "I am Franz";
  },
};
```

Variance

- Rules if and when super- or subtypes can be used
- Usually follows common sense
- Formally defined by the position of a type
  - Parameter
  - Return Type
  - Property

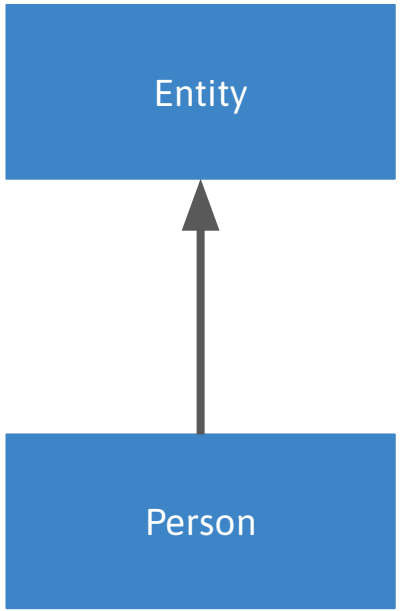
Covariance

```
class Entity {
 id: number = 0;
}
class Person extends Entity {
 name: string = "";
}

interface Factory<T> {
 create: () => T;
}

let personFactory: Factory<Person> = { create: () => new Person() };
let entityFactory: Factory<Entity> = personFactory; // works

entityFactory = { create: () => new Entity() };
personFactory = entityFactory; //fails
```



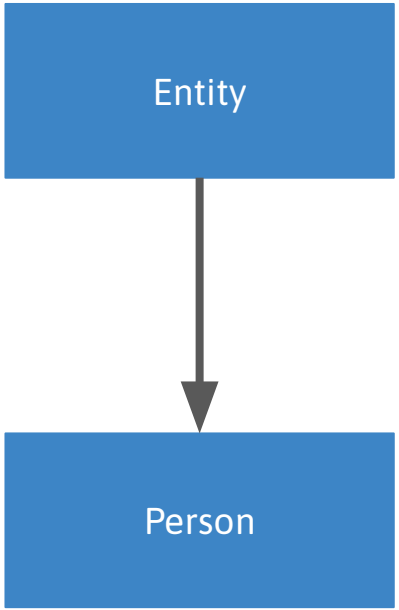
Contravariance

```
class Entity {
 id: number = 0;
}
class Person extends Entity {
 name: string = "";
}

interface Store<T> {
 save: (entity: T) => void;
}

let personStore: Store<Person> = { save: (person: Person) => void true };
let entityStore: Store<Entity> = personStore; // fails

entityStore = { save: (entity: Entity) => void true};
personStore = entityStore; // works
```



ANGULAR ARCHITECTS
INSIDE KNOWLEDGE

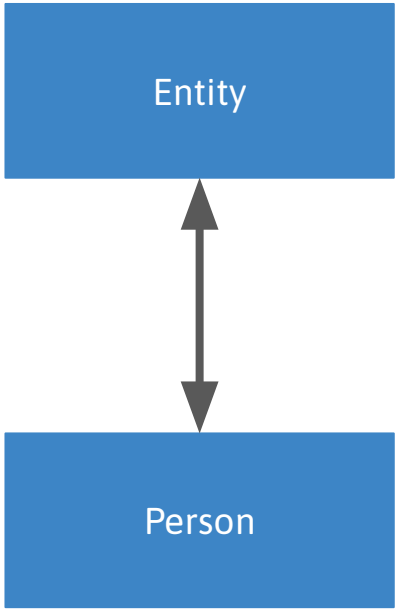
Bivariance

```
class Entity {
 id: number = 0;
}
class Person extends Entity {
 name: string = "";
}

interface Store<T> {
 save(entity: T): void;
}

let personStore: Store<Person> = { save: () => new Person() };
let entityStore: Store<Entity> = personStore; // works

entityStore = { save: () => new Entity() };
personStore = entityStore; // works
```



ANGULAR ARCHITECTS
INSIDE KNOWLEDGE

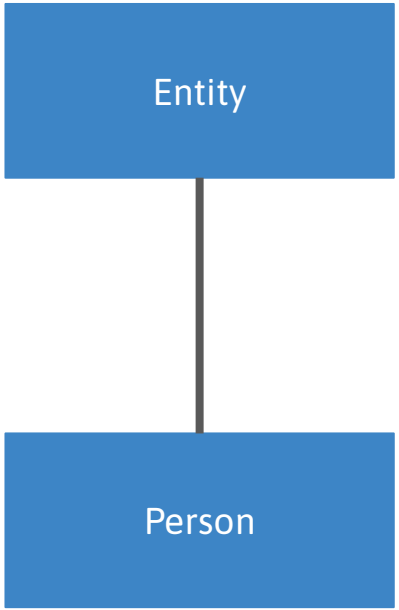
Invariance

```
class Entity { id: number = 0; }
class Person extends Entity { name: string = ""; }

interface Store<T> {
 save: (entity: T) => void;
 create(): T;
}

let personStore: Store<Person> = {
 save: (person: Person) => void true,
 create: () => new Person(),
};
let entityStore: Store<Entity> = personStore; // fails

entityStore = {
 save: (entity: Entity) => void true,
 create: () => new Entity(),
};
personStore = entityStore; // fails
```



Variance (simplified)

- covariant
  - properties
- contravariant
  - Function parameter when function is defined as a property
- bivariant
  - Function parameter when function is defined as a method
- invariant
  - Both covariant and contravariant appear together

Variance (simplified)

- covariant
  - properties
- contravariant
  - Function parameter when function is defined as a property
- bivariant
  - Function parameter when function is defined as a method
- invariant
  - Both covariant and contravariant appear together

Variance (common sense)

Covariant = goes out

Contravariant = goes in

Utility Types

- `NonNullable`
- `Partial`
- `Omit`
- `Readonly`
- `Required`
- `ReturnType`
- `Parameters`

ANGULAR
ARCHITECTS
INSIDE KNOWLEDGE