Training TypeScript

Module: Mapped Types & Utility Types

Peter Kassenaar

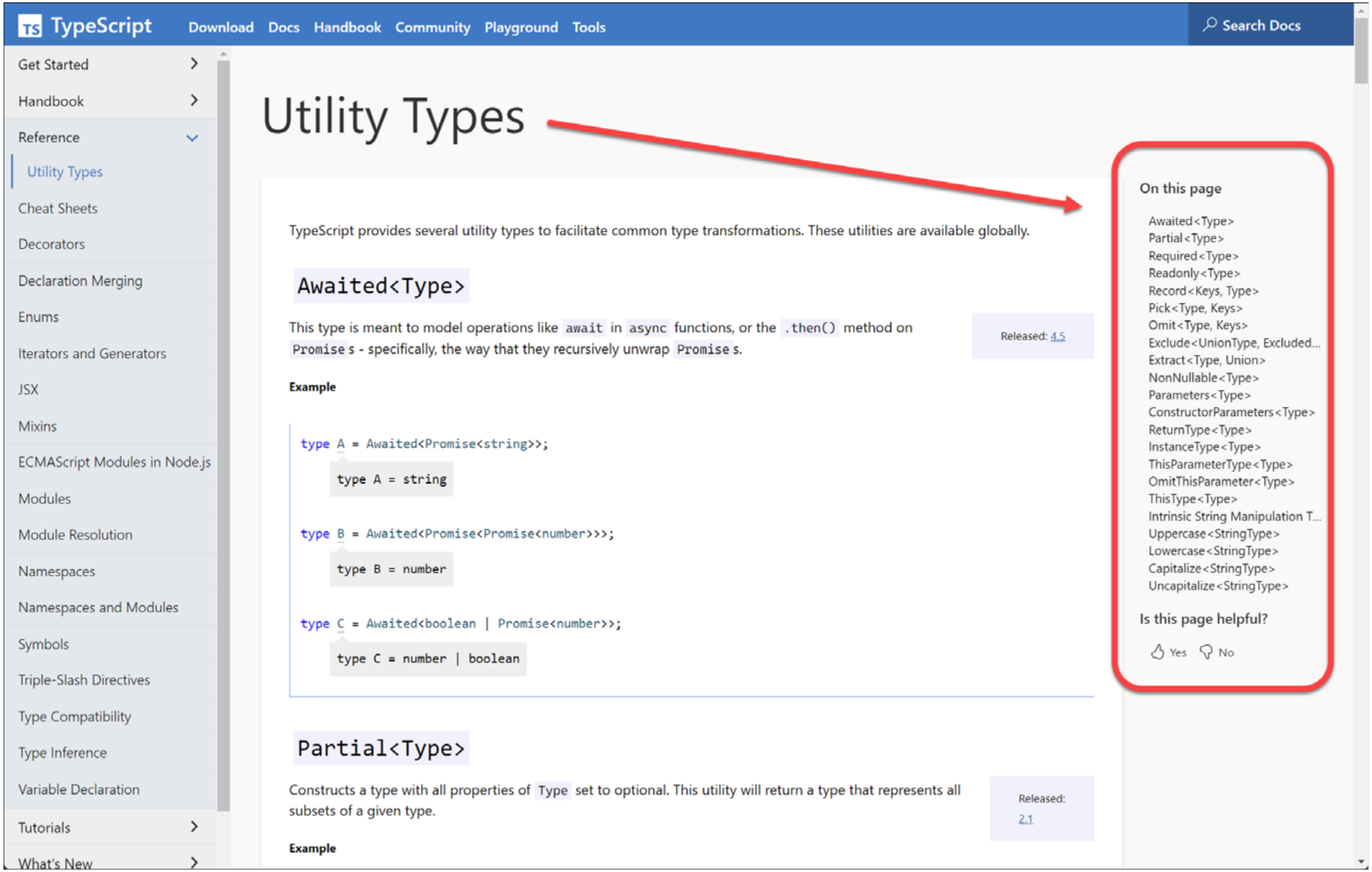
info@kassenaar.com

What are utility types?

“TypeScript provides several utility types to facilitate common type transformations. These utilities are available globally.”

Generally we use a utility type if we don’t want to repeat ourself (DRY programming) and we want a type, based on another type that is already there.

There are A LOT of utility types



Utility Types

TypeScript provides several utility types to facilitate common type transformations. These utilities are available globally.

`Awaited<Type>`

Released: 4.5

This type is meant to model operations like `await` in `async` functions, or the `.then()` method on `Promise` s - specifically, the way that they recursively unwrap `Promise` s.

Example

```
type A = Awaited<Promise<string>>;

    type A = string

type B = Awaited<Promise<Promise<number>>>;

    type B = number

type C = Awaited<boolean | Promise<number>>;

    type C = number | boolean
```

`Partial<Type>`

Released: 2.1

Constructs a type with all properties of `Type` set to optional. This utility will return a type that represents all subsets of a given type.

Example

On this page

- Awaited<Type>
- Partial<Type>
- Required<Type>
- Readonly<Type>
- Record<Keys, Type>
- Pick<Type, Keys>
- Omit<Type, Keys>
- Exclude<UnionType, Excluded...
- Extract<Type, Union>
- NonNullable<Type>
- Parameters<Type>
- ConstructorParameters<Type>
- ReturnType<Type>
- InstanceType<Type>
- ThisParameterType<Type>
- OmitThisParameter<Type>
- ThisType<Type>
- Intrinsic String Manipulation T...
- Uppercase<StringType>
- Lowercase<StringType>
- Capitalize<StringType>
- Uncapitalize<StringType>

Is this page helpful?

https://www.typescriptlang.org/docs/handbook/utility-types.html

Utility types

Mapped Types

Transforming from one type to another

What is a TypeScript Mapped Type?

- Compare it to the `.map()` operator on arrays for example:
  - transform a value into another value;
- In TypeScript: Compare types and possibly change one type to another type (i.e. *map* it).
- And there are several ways to do that:
  - `Readonly` Mapped Type
  - `Partial` Mapped Type
  - `Required` Mapped Type
  - `Pick` Mapped Type
  - `Record` Mapped Type
  - …

Utility types HEAVILY rely on Generics

- Generally: You pass in a generic type and TypeScript calculates the resulting type
- It also works (under the hood) a lot with the in, keyof and extends operator
    - Make sure to understand this!

```
// Example from TypeScript source code
type Readonly<T> = {
    readonly [P in keyof T]: T[P];
};
```

This basically says "Take the generic type T, read all the properties P in the keys of T, make the value of property P in T readonly and return it"

Utility types

Readonly<T>

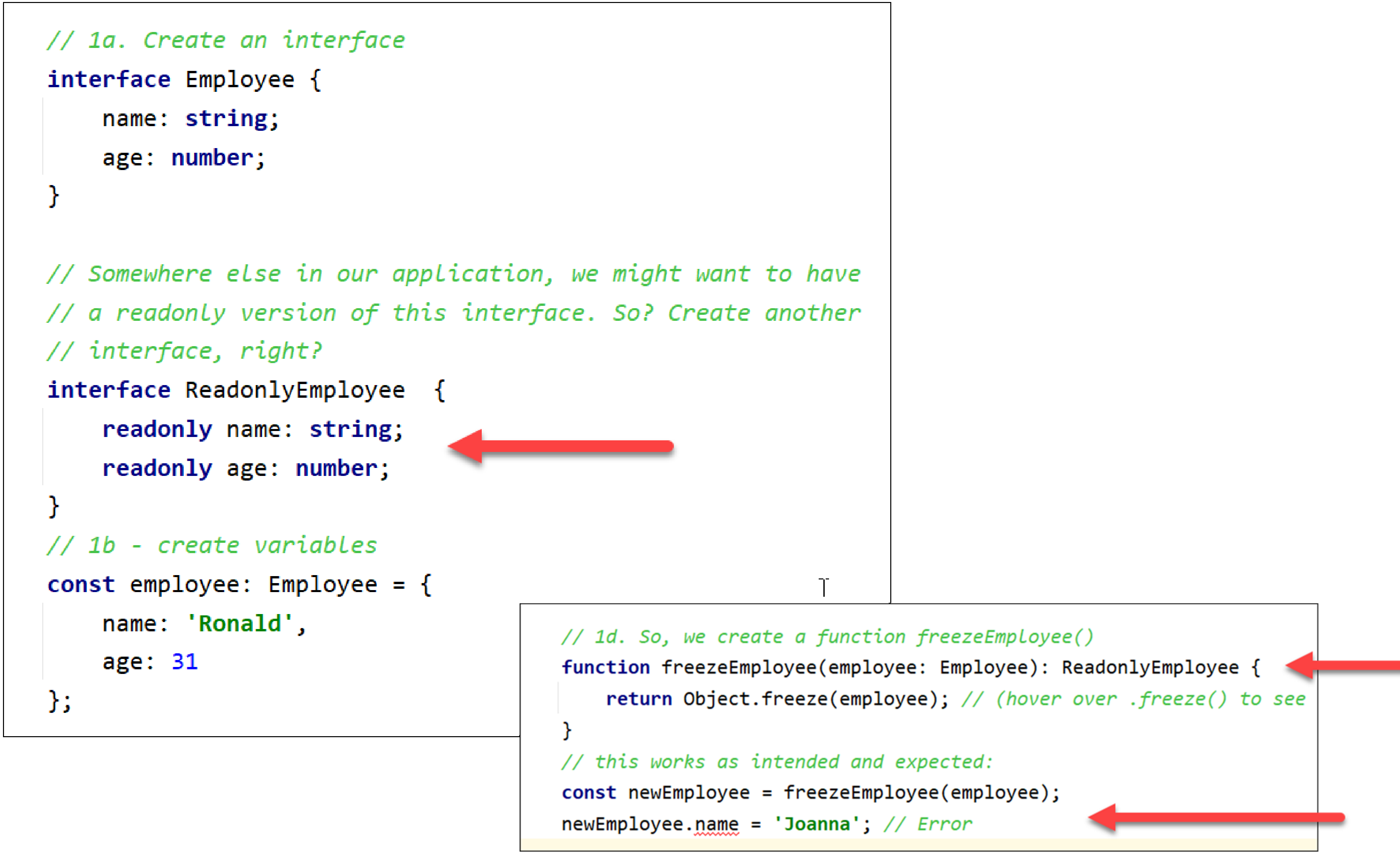
Making properties readonly (or: ‘freeze’ properties).

Mapped Type `Readonly<T>`

“Constructs a `type` (or `interface`) with all properties of `Type` set to `readonly`, meaning the properties of the constructed type cannot be reassigned.”

If you would want to do this by hand, you would create an `Object.freeze()` method, but TypeScript can do this for you by using the `Readonly<T>` generic utility type

Let’s say we were to do this by hand…

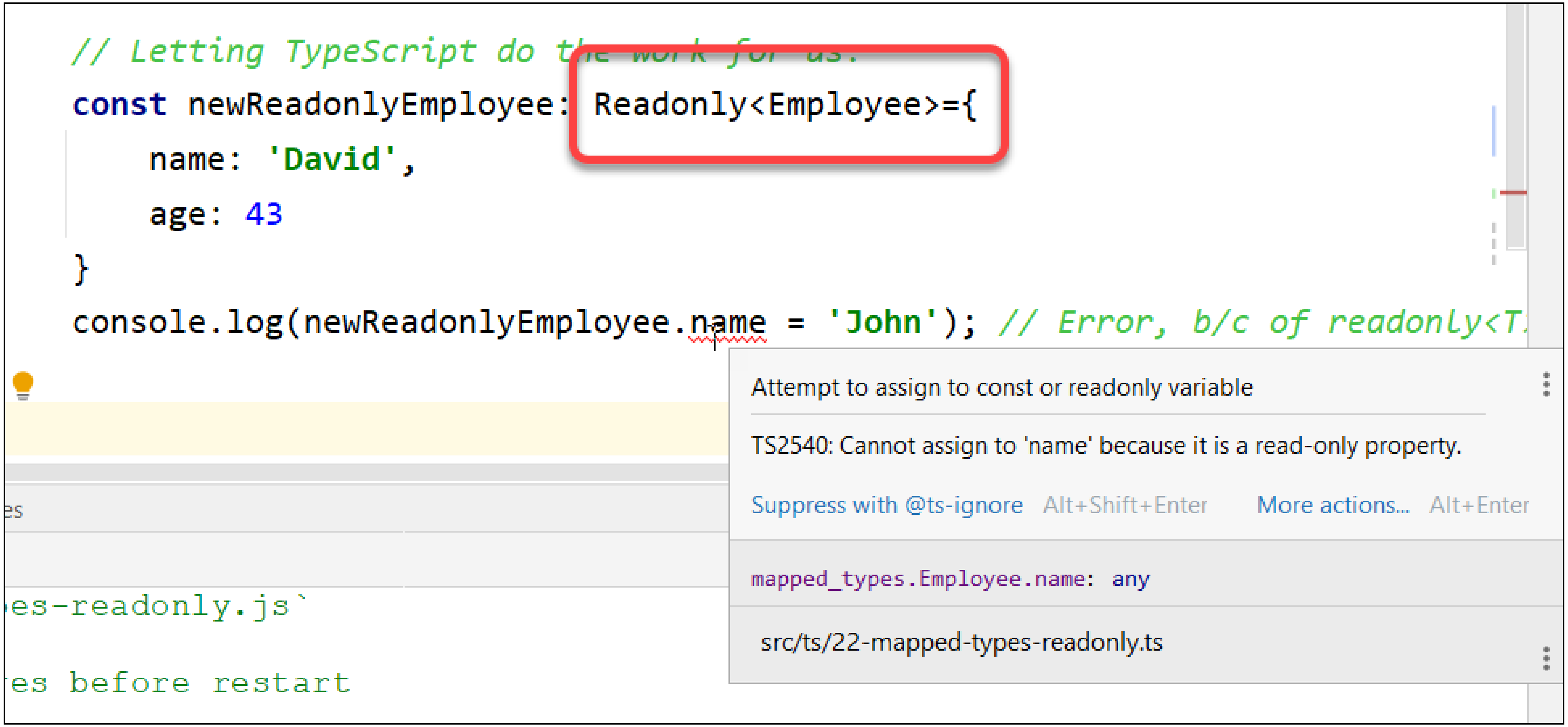


```typescript
// 1a. Create an interface
interface Employee {
    name: string;
    age: number;
}

// Somewhere else in our application, we might want to have
// a readonly version of this interface. So? Create another
// interface, right?
interface ReadonlyEmployee  {
    readonly name: string;
    readonly age: number;
}
// 1b - create variables
const employee: Employee = {
    name: 'Ronald',
    age: 31
};
```

```typescript
// 1d. So, we create a function freezeEmployee()
function freezeEmployee(employee: Employee): ReadonlyEmployee {
    return Object.freeze(employee); // (hover over .freeze() to see
}
// this works as intended and expected:
const newEmployee = freezeEmployee(employee);
newEmployee.name = 'Joanna'; // Error
```

22-mapped-types-readonly.ts

But, let TypeScript do the work for us



```
// Letting TypeScript do the work for us.
const newReadonlyEmployee: Readonly<Employee>={
    name: 'David',
    age: 43
}
console.log(newReadonlyEmployee.name = 'John'); // Error, b/c of readonly<T.
```

Attempt to assign to const or readonly variable

TS2540: Cannot assign to 'name' because it is a read-only property.

Suppress with @ts-ignore Alt+Shift+Enter
More actions... Alt+Enter

mapped_types.Employee.name: any

src/ts/22-mapped-types-readonly.ts

```
es-readonly.js`
es before restart
```

Utility types

Partial<T>

Making properties optional

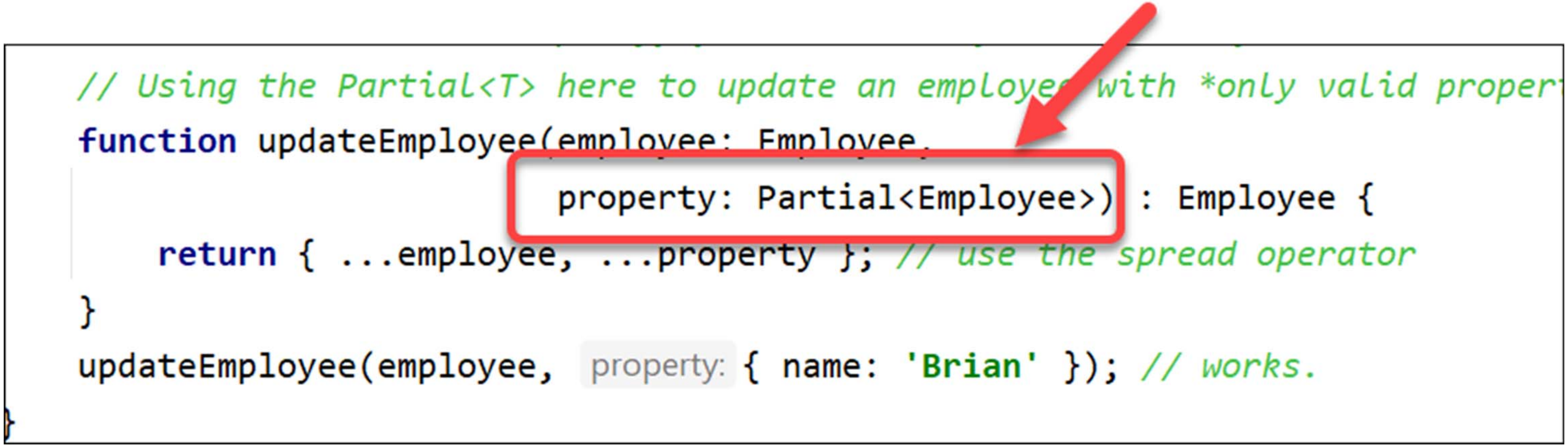
Mapped type `Partial<T>`

> *“Constructs a type with all properties of Type set to optional. This utility will return a type that represents all subsets of a given type.”*

If you would want to do this by hand, you would create a new interface with all props set to `optional?`, but TypeScript can do this for you by using the `Partial<T>` generic utility type

Using Partial <T>

```
// Create interface and constant as in the previous file
interface Employee {
    name: string;
    age: number;
}
const employee: Employee = {
    name: 'Ronald',
    age: 31
};
```



```
    // Using the Partial<T> here to update an employee with *only valid proper
    function updateEmployee(employee: Employee,
                            property: Partial<Employee>) : Employee {
        return { ...employee, ...property }; // use the spread operator
    }
    updateEmployee(employee, property: { name: 'Brian' }); // works.
}
```

Verdict

The utility type `Partial<T>` is VERY HANDY in preventing updating non-existent properties accidentally

Utility types

Required<T>

Making properties required – the opposite of Partial<T>

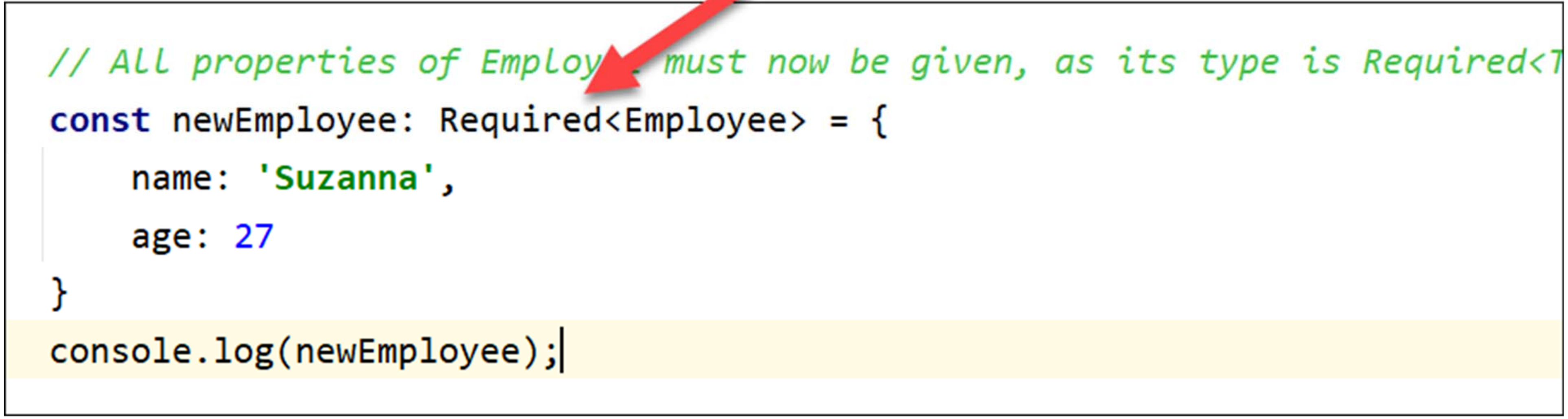
Mapped type `Required<T>`

“Constructs a type consisting of all properties of Type<T> set to required. It is the opposite of Partial.”

If your interface or type has `optional?` parameters by itself, but you want a derived type to have ALL the properties, use the built-in TypeScript `Required<T>` generic utility type.

Using Required<T>

```
interface Employee {
    name?: string;
    age?: number;
}
const employee: Employee = {
    name: 'Ronald'// this is now OK, as the props of Employee are optional
};
```



```
// All properties of Employee must now be given, as its type is Required<T
const newEmployee: Required<Employee> = {
    name: 'Suzanna',
    age: 27
}
console.log(newEmployee);
```

../24-mapped-types-required.ts

Utility types

Extract<T>

Extracting types of a union Type Partial<T>

Mapped type `Extract<Type, Union>`

> *“Constructs a type by extracting from `Type` all union members that are assignable to `Union`.”*

If your interface or type has `optional?` parameters by itself, but you want a derived type to have ALL the properties, use the built-in TypeScript `Required<T>` generic utility type.

Using `Extract<Type, Union>`

- With `Extract<T, U>` you pass in first a `type` you want to extract from
- The second parameter is the one (literal, `interface` or `type`) you want to match on.
- Really useful when using code generation, as you typically get a lot of union types returned. You can now `Extract` the correct (sub)type

Extract example

```
interface Employee {
    name: string;
    age: number;
}

interface Department {
    depName: string;
}

interface Company {
    city: string;
}

// We NEED a type here, as an interface can't be extracted.
type Recipient = Extract<Employee | Department | Company, Employee>

// we now KNOW Recipient is of type Employee, b/c of the extraction above.
const recipient: Recipient = {
    name: 'Peter',  // therefore these props are required
    age: 30
}
```

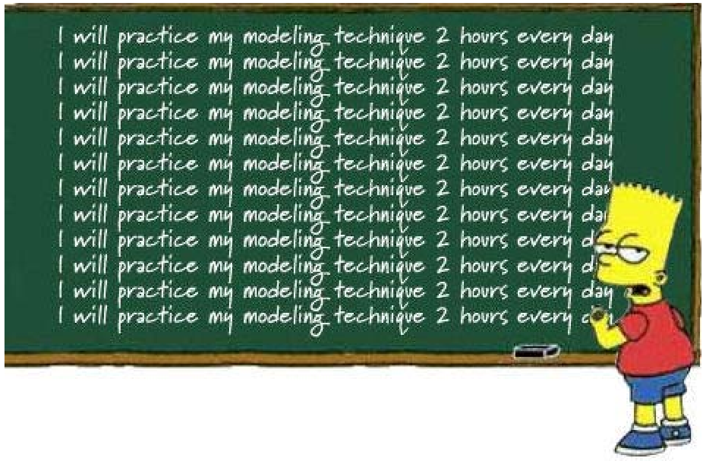
Another example Extract<>

```
type Trip = {
    origin: {
        uuid: string;
        city: string;
        state: string;
    };
} | {
    originUuid: string;
};

type TripWithOriginRef = Extract<Trip, { originUuid: string }>;

type TripWithOriginWhole = Extract<Trip, { origin: { uuid: string } }>;
```

Workshop

- Create a type or an interface and create some variables, based on that type
- Create a new variable, but make it `Readonly<T>`.
  - Try changing some properties and see what happens
  - When would you use the `Readonly<T>` utility type?
- Create a new variable, using the `Partial<T>` function
- Create a new type/interface with optional properties
  - Create a new variable using `Required<T>` for the props
- Create a complex type and `Extract<>` another type from it.
- Docs:

  https://www.typescriptlang.org/docs/handbook/utility-types.html


I will practice my modeling technique 2 hours every day
I will practice my modeling technique 2 hours every day
I will practice my modeling technique 2 hours every day
I will practice my modeling technique 2 hours every day
I will practice my modeling technique 2 hours every day
I will practice my modeling technique 2 hours every day
I will practice my modeling technique 2 hours every day
I will practice my modeling technique 2 hours every day
I will practice my modeling technique 2 hours every d
I will practice my modeling technique 2 hours every d
I will practice my modeling technique 2 hours every day
I will practice my modeling technique 2 hours every d

Utility types

Pick<Type, Keys>

Creating a type with a subset with the given keys

Mapped type `Pick<Type, Keys>`

> *“Constructs a type by picking the set of properties Keys (string literal or union of string literals) from Type”*

If your interface or type has a bunch of parameters, but you want a derived type that only has a selection of these properties, use the built-in TypeScript `Pick<T, Keys>` generic utility type.

Using `Pick<Type, Keys>`

- With `Pick<Type, Keys>` you pass in first a `type` you want to extract from
- The second parameter is the one (literal union, `interface` or `type`) you want the selection to be.
- Really useful when you have a gigantic base `type` that you want to create subtypes from. You can now `Pick` the correct (sub)type.

Pick<Type, Keys> example

```
// Our example, some interfaces (or types, whatever you want):
interface Employee {
    name: string;
    age: number;
    address: {} // <== new property
}

// the variable pickedEmployee has a selection of keys from
// the full (or: original) object. Of course you can turn
// the properties 'name' | 'age' into their own Union Type.
const pickedEmployee: Pick<Employee, 'name' | 'age'> = {
    name: 'Sandra',
    age: 20

}
```

Pick<T, K> behind the scenes

- You can create your own type that does the same.
  - The first parameter T is the Type
  - The second parameter is all the keys that must exist in Type, they are assigned to K
  - The function body looks for the properties in K and if they exist (using the in operator), returns an object that has the shape as defined in <K extends keyof T>

```
// Background: the Pick<Type, Keys> actually looks like this:
// Understand the notation K extends keyof T!
type myPick<T, K extends keyof T> = {
    [P in K]: T[K]
}
```

Utility types

Record<Keys, Type>

Use properties as keys for another type

Mapped type `Record<Keys, Type>`

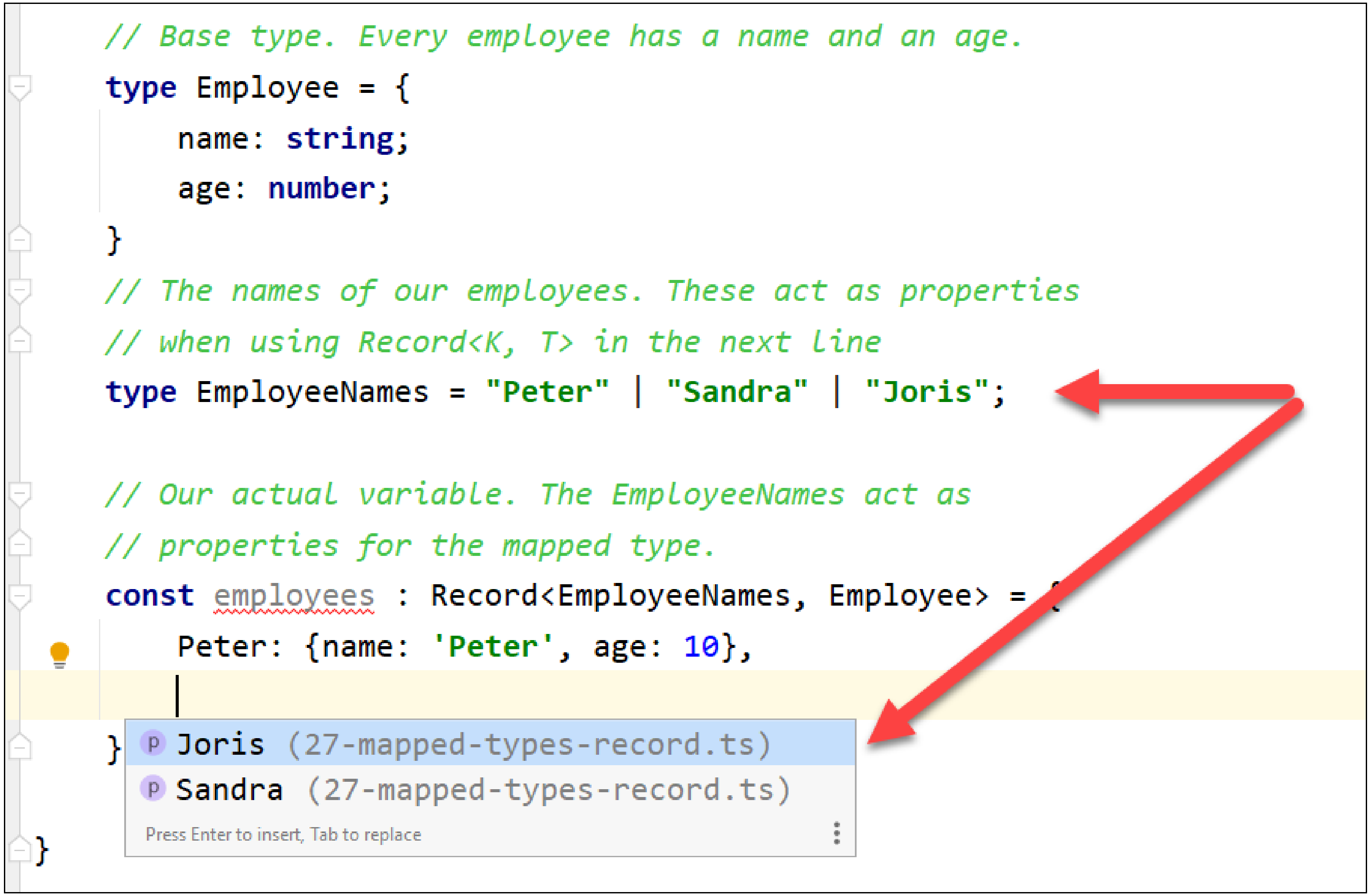
“Constructs an object type whose property keys are Keys and whose property values are Type. This can be used to map the properties of a type to another type.”

If you have two types, one containing the keys and another one containing the values, you can use Record<K, T> to create a new type with mapped properties

Using `Record<Keys, Type>`

- With `Record<Keys, Type>` you pass in first a series of Keys that you want to be the properties for the resulting variable.
- The second parameter is the type (`interface` or `type`) that the properties must have.
- This is useful when following the dictionary pattern and you want to be type safe on the keys and on the values. You can combine two types into one new type

Record<Keys, Type> example



```
// Base type. Every employee has a name and an age.
type Employee = {
    name: string;
    age: number;
}
// The names of our employees. These act as properties
// when using Record<K, T> in the next line
type EmployeeNames = "Peter" | "Sandra" | "Joris";

// Our actual variable. The EmployeeNames act as
// properties for the mapped type.
const employees : Record<EmployeeNames, Employee> = {
    Peter: {name: 'Peter', age: 10},
    
}
```

P Joris (27-mapped-types-record.ts)
P Sandra (27-mapped-types-record.ts)
Press Enter to insert, Tab to replace

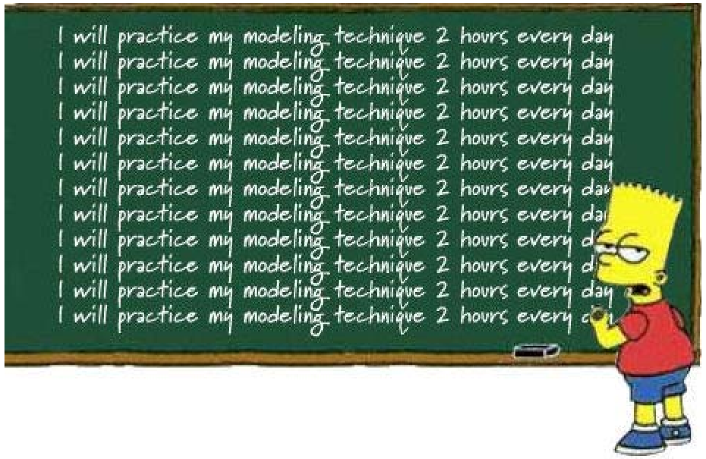
Record<Keys, Type> behind the scenes

- This is the definition of Record<K, T>
  - We can use any type as property, as it extends from any.
  - Earlier versions of TypeScript had only strings (!), but now it’s broader.

```
// from typescript source code
type Record<K extends keyof any, T> = {
    [P in K]: T;
};
```

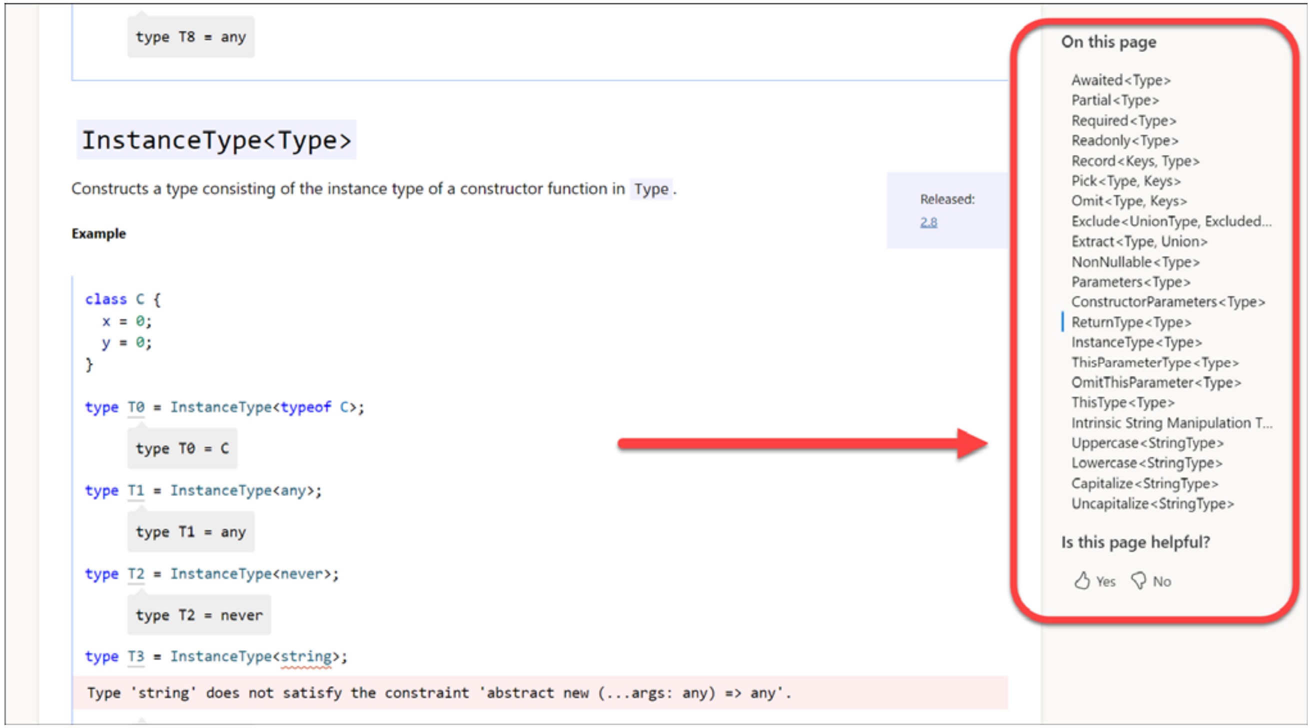
Workshop

- Create a type or an interface and create some variables, based on that type (we used `Employee` throughout)
- Create a new variable, but use `Pick<Type, Keys>`.
  - Try changing some properties and see what happens
  - When would you use the `Pick<T, K>` utility type?
- Create a base type and a `Key` type. Use `Record<Type, Keys>` to create a derived type that has the `Keys` type as keys for the values of that type
  - See `27-mapped-type-record.ts` as example
- Docs: https://www.typescriptlang.org/docs/handbook/utility-types.html


I will practice my modeling technique 2 hours every day
I will practice my modeling technique 2 hours every day
I will practice my modeling technique 2 hours every day
I will practice my modeling technique 2 hours every day
I will practice my modeling technique 2 hours every day
I will practice my modeling technique 2 hours every day
I will practice my modeling technique 2 hours every da
I will practice my modeling technique 2 hours every da
I will practice my modeling technique 2 hours every d
I will practice my modeling technique 2 hours every d
I will practice my modeling technique 2 hours every day
I will practice my modeling technique 2 hours every

More utility Types:

- `Omit<Type, Keys>`
- `Exclude<UnionType, ExcludedMembers>`
- `Parameters<Type>`
- `ReturnType<Type>`
- `InstanceType<Type>`
- And many, many more….



```
type T8 = any
```

`InstanceType<Type>`

Constructs a type consisting of the instance type of a constructor function in `Type`.

Released:
2.8

Example

```
class C {
  x = 0;
  y = 0;
}

type T0 = InstanceType<typeof C>;
```

```
type T0 = C
```

```
type T1 = InstanceType<any>;
```

```
type T1 = any
```

```
type T2 = InstanceType<never>;
```

```
type T2 = never
```

```
type T3 = InstanceType<string>;
```

```
Type 'string' does not satisfy the constraint 'abstract new (...args: any) => any'.
```

On this page

- Awaited<Type>
- Partial<Type>
- Required<Type>
- Readonly<Type>
- Record<Keys, Type>
- Pick<Type, Keys>
- Omit<Type, Keys>
- Exclude<UnionType, Excluded...
- Extract<Type, Union>
- NonNullable<Type>
- Parameters<Type>
- ConstructorParameters<Type>
- ReturnType<Type>
- InstanceType<Type>
- ThisParameterType<Type>
- OmitThisParameter<Type>
- ThisType<Type>
- Intrinsic String Manipulation T...
- Uppercase<StringType>
- Lowercase<StringType>
- Capitalize<StringType>
- Uncapitalize<StringType>

Is this page helpful?

https://www.typescriptlang.org/docs/handbook/utility-types.html