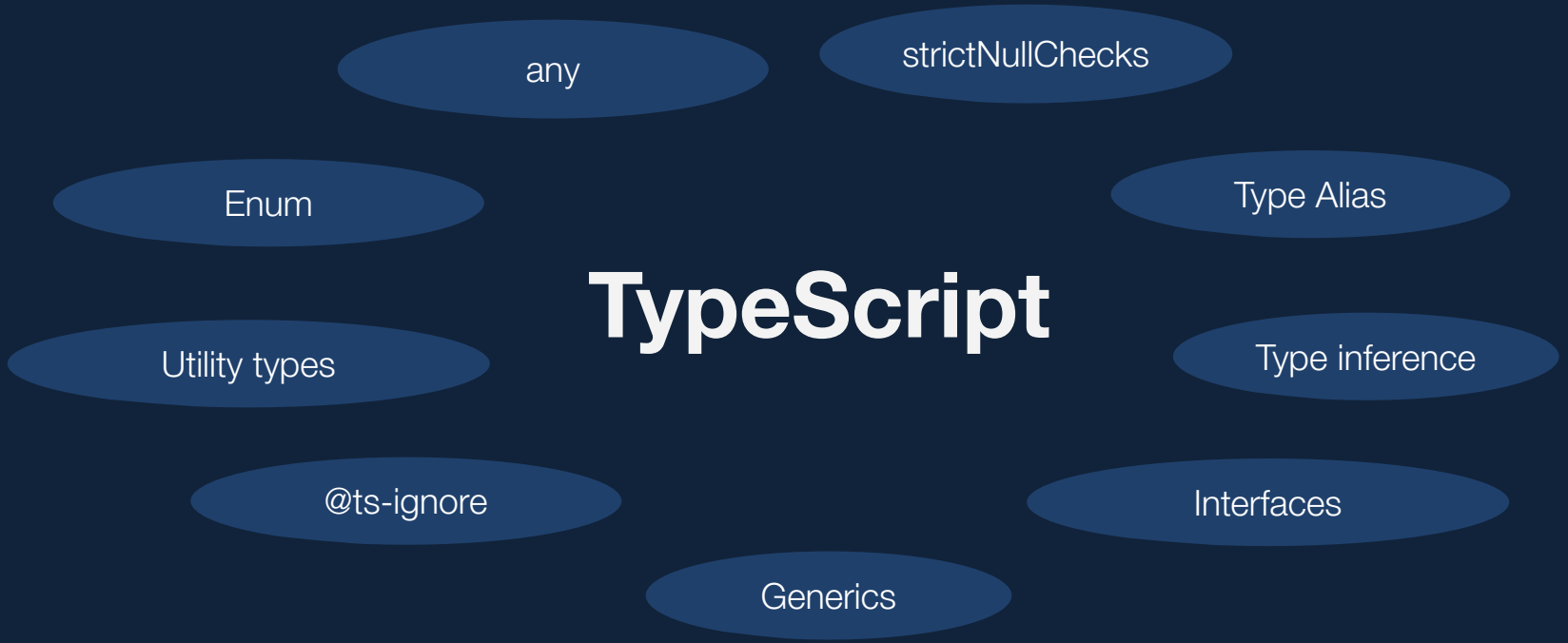
Workshop

TypeScript Introduction

Hint for trainers

- Report each change or addition to the **trainers'** Discord-Channel.
- Tell which Slide is affected, why the change is important and what benefit your change provides.
- Use the code-highlighting-app if you work with code-snippets.
- Use the following slide if you want to repeat certain topics of the workshop.

Task: Test your knowledge



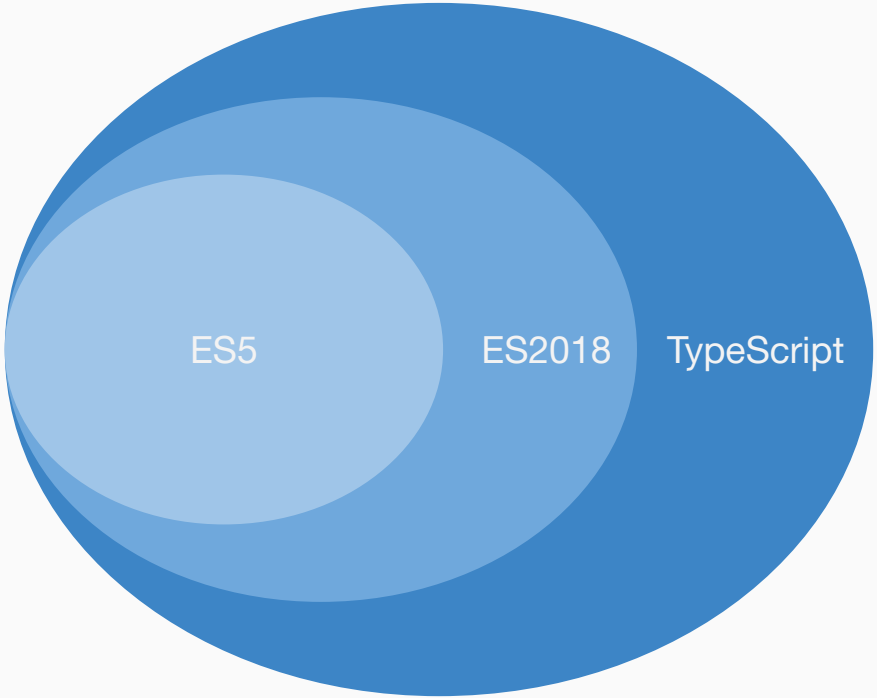
TypeScript

Optional static type-checking

TypeScript is a **typed superset** of JavaScript that **compiles to plain JavaScript**.

TypeScript is a superset

- Superset of EcmaScript
- Compiles to clean code
- Optional Types



Why TypeScript

Why **TypeScript**

- Types make code more secure
- Understand how to use a function by just looking at its declaration
- Types serve as documentation (e.g. what comes in and what goes out of functions)
- Statement completion and code refactoring
- Familiarity with other typed programming languages
- Be safe against JS type coercions
- Avoid simple unit tests `(expect(service.get).toBeDefined)`

The result: better maintenance for long-living projects

Types

Types in TypeScript - Variables

Types exist for primitive types.

```
const isDone:    boolean = true;

const size:      number = 42;

const firstName: string = 'Lena';
```

Types in TypeScript - Variables

Types exist for reference types.

```
const attendees: string[] = ['Elias', 'Anna'];

const attendees: Array<string> = ['Elias', 'Anna'];

const attendees: ReadonlyArray<string> = ['Elias', 'Anna'];
```

Types - any

any takes any type. You are in JavaScript-Land.

```
let question: any = 'Can be a string';

question = 6 * 7;
question = false;
```

Types - Unknown

unknown takes any type - but you cannot access any property unless you check for its existence

```
let vAny: any = 10;           // We can assign anything to any
let vUnknown: unknown =  10; // We can assign anything to unknown just
like any


let s1: string = vAny;      // value of type any is assignable to anything
let s2: string = vUnknown; // Invalid: value of type unknown (vUnknown)
can't be assigned to any other type (without an explicit assertion)

vAny.method();      // ok anything goes with any
vUnknown.method(); // not ok, we don't know anything about this variable
```

The type **unknown** is more defensive than **any**.

Type inference

Type inference

You don’t have to set the type of a variable if the compiler can infer it.

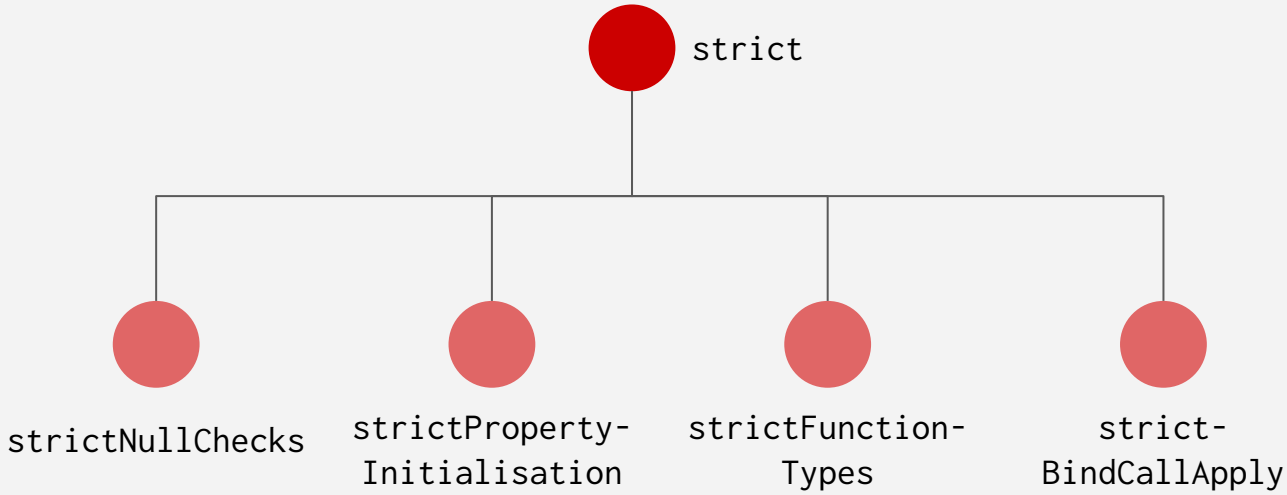
```
const firstName  = 'Max';                    // string
const age        = 30;                       // number
const isEmployed = true;                     // boolean
const friends    = ['Stefan', 'Frederike'];  // string[]
const dayOfBirth = Date.parse('...');        // Date
```

null and undefined

By default each data type in TypeScript does not accept `null` and `undefined`.

Compiler Flag: strict

Strict is a shorthand activating multiple rules



Compiler Flag: `strict`


```
// tsconfig.json
{
  "strict": "true"
  // ...
}
```

Compiler Flag: strict

In strict null checking mode, the null and undefined values are not in the domain of every type.

```
let firstName  : string  | null      = null;
let age        : number  | undefined = undefined;
let isEmployed : boolean | undefined;


// TypeScript Errors
let firstName  : string = null;
let age        : number = undefined;
let isEmployed : boolean; // undefined
```

Functions

Functions - Types

Add types to function parameters and return values.

```
function sayHi(firstName: string): void {
  console.log(firstName);

  return undefined
}
```

Functions - Optional parameters

Parameters can be optional. Use a question mark.

```
function buildName(firstName: string, lastName?: string) {
  if (lastName) {
    return firstName + ' ' + lastName;
  } else {
    return firstName;
  }
}
```

Functions - Default parameters

Function arguments can have defaults for arguments.

```
// type Inference: lastName is a string
function buildName(firstName: string, lastName: string = 'Bond') {
  return firstName + ' ' + lastName;
}
```

Object property checks:

Interfaces & Type Aliases

There are **two** syntactical flavors to type objects and values:

Interface, Type-Alias

Interfaces and Type Aliases

Give the structure of an object a name

Interface

```
interface Book {
    isbn: string;
    title: string;
}
```

Type Alias

```
type Book = {
    isbn: string;
    title: string;
}
```

```
let book: Book;

book = {
  isbn: '978-1593272821',
  title: 'Eloquent JavaScript'
};
```

```
const book: Book = {
  isbn: '978-1593272821',
  title: 'Eloquent JavaScript'
};
```

Optional properties

<code>

Properties can be optional.

Interface

```
interface Book {
    isbn: string;
    title: string;
    pages?: number;
}
```

Type Alias

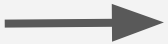
```
type Book = {
    isbn: string;
    title: string;
    pages?: number;
}
```

```
const book: Book = {
  isbn: 'Goethe, Johann Wolfgang: Faust. Der Tragödie Erster Teil',
  title: '978-3-15-000001-4',
}
```

Optional properties

Optional property is **not** equivalent to property which may be undefined.

```ts
interface Book {
    isbn: string;
    title: string;
    pages?: number;
}
```



```ts
interface Book {
  isbn: string;
  title: string;
  pages: number | undefined | null;
}
```

```ts
const book: Book = {
  isbn: 'Goethe, Johann Wolfgang: Faust. Der Tragödie Erster Teil',
  title: '978-3-15-000001-4',
}
// Property 'pages' is missing in type '{ isbn: string; title: string; }' but required
in type 'Book'.
```

Nesting

<code>

Interfaces and Type Aliases can be nested

Interface

```
interface Profile {
  id: number;
  gender: string;
  name: string;
  pictureUrl?: string;
  address: {
    street: string,
    zipCode: string,
    city: string,
  }
}
```

Type Alias

```
type Profile = {
  id: number;
  gender: string;
  name: string;
  pictureUrl?: string;
  address: {
    street: string,
    zipCode: string,
    city: string,
  }
}
```

Extends & Intersection Type

<code>

The properties of Book are merged into Magazine.

Interface

```
interface Book {
    isbn: string;
    title: string;
    pages?: number;
}


interface Magazine extends Book {
  coverUrl: string;
}
```

Type Alias

```
type Book = {
    isbn: string;
    title: string;
    pages?: number;
}


type Magazine = Book & {
  coverUrl: string;
};
```

Union Type

<code>

Expect a property to be either one or the other

Interface

```
interface Style {
  position?: string;
  padding?: string | number;
  margin?: string | number;
  // other style properties
}
```

Type Alias

```
type Style = {
  position?: string;
  padding?: string | number;
  margin?: string | number;
  // other style properties
}
```

Alternative 1

```
const style: Style = {
  padding: '15px'
}
```

Alternative 2

```
const style: Style = {
  padding: 15
}
```

Union Type

Common Pattern: Discriminating Unions

```ts
interface Book {
  title: string;
  isbn: string;
}

interface Magazine {
  title: string;
  issn: string;
}

type ReadingMaterial = Book | Magazine;

const readingMaterial: ReadingMaterial[] = [
    {
        title: 'Vogue',
        issn: '0042-8000',
    },
    {
        title: 'Robinson Crusoe',
        isbn: '978-3401002569'
    }
]
```

Union Types

The properties of Book are merged into Magazine.

```
interface Book {
  isbn: string;
  title: string;
}
```

```
interface Magazine {
  title: string;
  issn: string;
}
```

```
type ReadingMaterial = Book | Magazine;

const getBook = (title: string): ReadingMaterial | undefined =>
  readingMaterial.find(rM => rM.title === title)
```

Interfaces

Give an interface a name and use it as a type for variables.

```
interface Book {
  isbn: string;
  title: string;
}

const book: Book;

book = {
  isbn: '978-1593272821',
  title: 'Eloquent JavaScript'
};
```

Type Aliases

Declare type alias by giving it a name & define its shape

```
type Book = {
  isbn: string;
  title: string;
}

const book: Book;

book = {
  isbn: '978-1593272821',
  title: 'Eloquent JavaScript'
};
```

When to use Interfaces

Interfaces

An Interface can be implemented by a class.

```
class Biography implements Book {
  isbn: string;
  title: string;
}
```

Interface Merging

When re-declaring interfaces its properties merge

```
interface Book {
    isbn: string;
    title: string;
}

// other code...

interface Book {
   pages?: number;
}
```

```
interface Book {
    isbn: string;
    title: string;
    pages?: number;
}
```

Interfaces - Class types

<code>

Forgetting to implement flipPage throws a compile error.

```
interface CanFlip {
  flipPage: Function;
}

class BookListComponent implements CanFlip {

}
```

```
Class 'BookListComponent' incorrectly implements interface 'CanFlip'.
 Property 'flipPage' is missing in type 'BookListComponent' but
required in type 'CanFlip'.
```

When to use Type Aliases

Type Alias for unions of type literals

<code>

You can allow certain valid values for a type

```
// Example 1
type HttpStatusCodes = 200 | 201 | 400;

// Example 2
type TextAlignOptions = 'left' | 'right' | 'center' | 'auto' | 'justify';
```

Type alias for primitive types

Use type aliases to provide more context (as a form of documentation)

```
type ContractId = string;
type CustomerId = string;

export interface Contract {
   id: ContractId;
   customerId: CustomerId;
}
```

Interfaces vs. Type Aliases

Interface vs. Type Alias

TypeScript Docs

“Type aliases and interfaces are **very similar**, and in many cases you can **choose between them freely**. Almost all features of an interface are available in type, the key distinction is that a **type cannot be re-opened** to add new properties vs an **interface** which is **always extendable**..”

https://www.typescriptlang.org/docs/handbook/2/everyday-types.html#differences-between-type-aliases-and-interfaces

Interface vs. Type Alias

Opinionated guide

“Interfaces are generally **preferred over** type literals because interfaces can be *implemented, extended* and *merged*.”

https://github.com/bradzacher/eslint-plugin-typescript/blob/master/docs/rules/prefer-interface.md

Function Types

Function Types as Type Alias

Use type aliases to provide more context (as a form of documentation)

```
type OnSuccess = (result: string) => void;

function doSomething(onSuccess: OnSuccess) {
    // Some async code
}

const onSuccess: OnSuccess = (result: string) => console.log(result);

doSomething(onSuccess);
```

Function Types as Interface

Use type aliases to provide more context (as a form of documentation)

```
interface OnSuccess {
    (result: string): void;
}
function doSomething(onSuccess: OnSuccess) {
    // Some async code
}
const onSuccess: OnSuccess = (result: string) => console.log(result);
doSomething(onSuccess);
```

TypeScript Generics

TypeScript Generics

The Array-Type is implemented as a generic.

```
const books: Array<number> = []

const list: ReadonlyArray<string> = ["foo", "bar"];


const books: Array<{title: string, isbn: string}> = []

const books: Array<Book> = []
```

TypeScript Generics

Generic object

```
type ResponseT<T> = {
  id: number;
  data: T[];
  createdAt: number;
  modifiedAt: number;
};

const response: ResponseT<Book> = {
  id: 1,
  data: [{ isbn: 'abc', title: 'Faust' }],
  createdAt: 12345678,
  modifiedAt: 12345678,
}
```

Generic Functions

Input argument is of same type as return value

```
const logIdentity = <T>(arg: T): T => {
  console.log(arg);
  return arg;
};
```

Generic Functions

Reverse a list: what gets passed into the function should be of the same type of what gets returned

```
const reverse = <T>(items: T[]): T[] => {
  let result = [];
  for (let i = items.length - 1; i >= 0; i--) {
    result.push(items[i]);
  }
  return result;
}
```

Generic Functions

Reverse a list: what gets passed into the function should be of the same type of what gets returned

```
const reversed = reverse([3, 2, 1]) // [1, 2, 3]
// Safety!
reversed[0] = '1'; // Error!
```

```
const reverse = <T>(items: T[]): T[] => {
  let result = [];
  for (let i = items.length - 1; i >= 0; i--) {
    result.push(items[i]);
  }
  return result;
}
```

Generic Functions

Placement of the generic identifier in fat-arrow functions

```
function reverse<T>(items: T[]): T[] { /* ... */ }

// vs

const reverse = <T>(items: T[]): T[] => { /* ... */ }
```

Task

Update the TypeScript definitions

Advanced Types

Enums

Enum

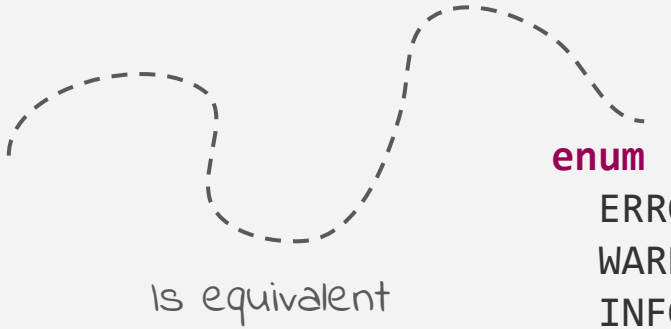
Using enums can make it easier to document intent, or create a set of distinct cases.

```
enum Progress {
  min = 0,
  max = 100,
}
```

Enum

Using enums can make it easier to document intent, or create a set of distinct cases.

```
enum LogLevel {
  ERROR,
  WARN,
  INFO,
  DEBUG
}
```




```
enum LogLevel {
  ERROR = 0,
  WARN = 1,
  INFO = 2,
  DEBUG = 3,
}
```

Enum

Named enum

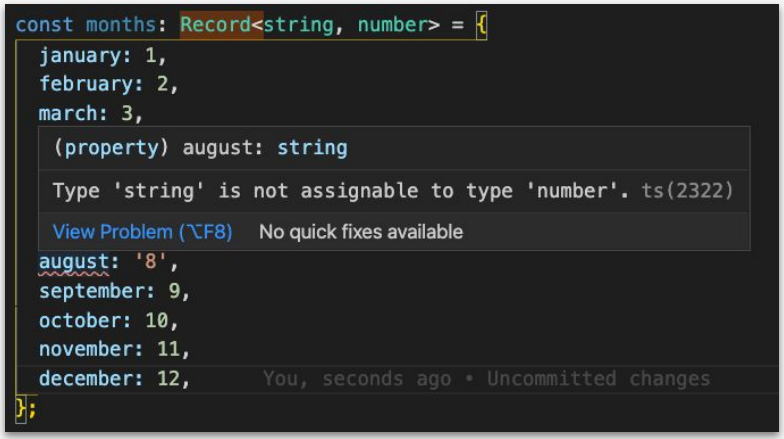
```
enum Gender {
  Female = 'FEMALE',
  Male = 'MALE',
  Other = 'OTHER',
}
```

Records

Record

Define type of all key and value pairs of object

```ts
const months: Record<string, number> = {
  january: 1,
  february: 2,
  march: 3,
  april: 4,
  mai: 5,
  june: 6,
  july: 7,
  august: 8,
  september: 9,
  october: 10,
  november: 11,
  december: 12,
};
```



Record

Define type of all key and value pairs of object

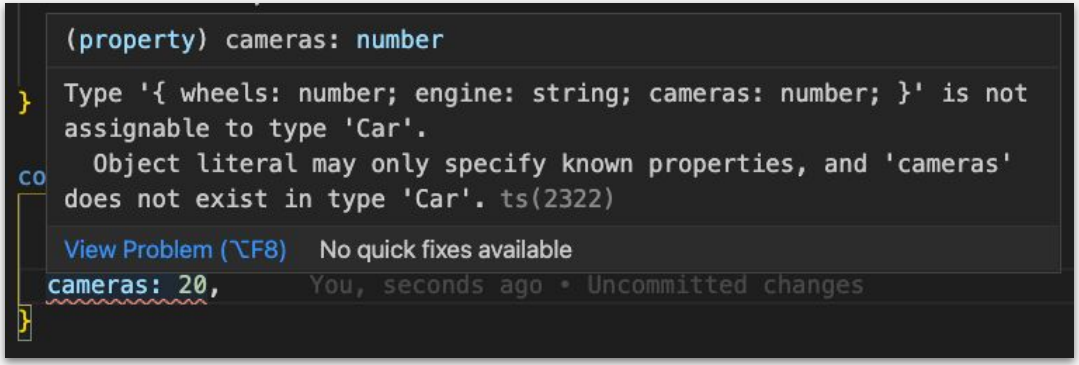
```ts
const arrayToObject = (values: number[]): Record<string, number> => {
  // ...
};

const getNumberFormatSettings = (): Record<string, string> => ({
  decimalSeparator: '.',
  groupingSeparator: ',',
});
```

Excess Property Checks

Excess Property Checks

Not defined properties cause a warning when directly giving a type to an object literal.

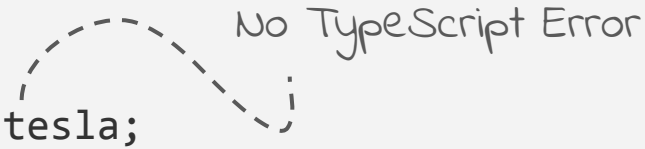
```ts
interface Car {
  wheels: number;
  engine: string;
}

const tesla: Car = {
  wheels: 4,
  engine: 'electric',
  cameras: 20, // Error
}
```



Excess Property Checks

Not defined properties cause an error when directly giving a type to an object literal.

```
interface Car {
  wheels: number;
  engine: string;
}

const tesla = {
  wheels: 4,
  engine: 'electric',
  cameras: 20,
}

const myTesla: Car = tesla;
```



Indexable Types

Object with certain properties but **arbitrary additional properties**.

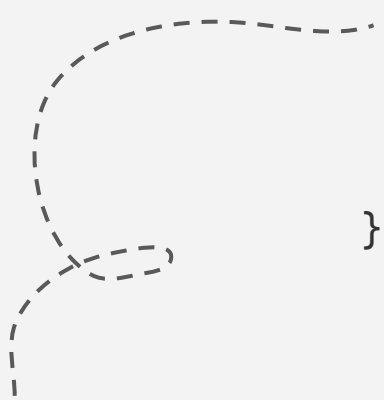
```ts
interface Car {
  wheels: number;
  engine: string;
  [key: string]: unknown;
}

const myTesla: Car = {
  wheels: 4,
  engine: 'electric',
  cameras: 20,
}
```

Utility types

Partial

Make all properties optional

```ts
type FormValues = {
  firstName: string;
  surname: string;
  dateOfBirth: Date;
  gender: Gender;
};
```



```ts
type FormValuesPartial = {
  firstName?: string;
  surname?: string;
  dateOfBirth?: Date;
  gender?: Gender;
};
```

```ts
const initialValues: Partial<FormValues> = { ...storedUserData };
```

`Omit` and `Pick`

`omit` certain properties of an object or `pick` only certain properties

```ts
interface Book {
  id: number;
  isbn: string;
  title: string;
  pages?: number;
}

type BookId = Pick<Book, 'id'>
type BookId = { id: Book['id'] };
type BookId = Omit<Book, 'isbn' | 'title' | 'pages'>;
```

```ts
const Id = {
  id: 12,
}
```

Three solutions which all lead to the same type.

`Omit` and `Pick`

`omit` certain properties of an object or `pick` only certain properties

```ts
interface Book {
  id: number;
  isbn: string;
  title: string;
  pages?: number;
}

type BookDetails = Pick<Book, 'title' | 'pages'>;
type BookDetails = Omit<Book, 'id' | 'isbn'>;
type BookDetails = {
  title: Book['title'];
  pages?: Book['pages'];
}
```

```ts
const details: BookDetails = {
  title: 'Hamlet',
  pages: 67,
}
```

Three solutions which all lead to the same type.

Combining Utility types

Combine utility types with other types to create more powerful types.

```
interface Book {
  id: string,
  title: string,
  isbn: string,
}

// Example 1
type OptionalBookPropsWithoutId = Omit<Partial<Book>, ‘id’>;

// Example 2
type BookWithNumberId = Omit<Book, 'id'> & { id: number };
```

Utility types

Use utility types to remain **rename refactoring safe**

```
interface Titles {
  bookTitles: Array<Pick<Book, 'title'>>
  videoTitles: Array<Pick<Video, 'title'>>
}
```

Compose TypeScript’s utility types

Partial<T> makes all properties of a type optional. What if we want to make only a few properties optional?

We can create our own utility type **Optional<>**.

How to read a complex type definition

This is how you can use it to make a property optional.

```
interface Customer {
    id: string;
    firstName: string;
    lastName: string
}

const optionalCustomer: Optional<Customer, 'firstName' | 'lastName'> = {
    id: "12";
}
```

How to read a complex type definition

Solution: This is how **Optional** can look like.


```
export type Optional<T, K extends keyof T> =  Pick<Partial<T>, K> & Omit<T, K>;
```

How to read a complex type definition


```
export type Optional<T, K extends keyof T> =  Pick<Partial<T>, K> & Omit<T, K>;
```

We create a new type and sharing it with other modules.

How to read a complex type definition


```
export type Optional<T, K extends keyof T> =  Pick<Partial<T>, K> & Omit<T, K>;
```

We name the type “Optional”.

How to read a complex type definition


```
export type Optional<T, K extends keyof T> =  Pick<Partial<T>, K> & Omit<T, K>;
```

“Optional” has two parameters:

1. The Object that should be made partially optional.
2. The properties of the object that should be optional.
   K has to be a property name of the Object.

How to read a complex type definition

<code>

```
export type Optional<T, K extends keyof T> =  Pick<Partial<T>, K> & Omit<T, K>;
```

Make the Object completely optional.

How to read a complex type definition


```
export type Optional<T, K extends keyof T> =  Pick<Partial<T>, K> & Omit<T, K>;
```

Take only the optional properties being specified in **K**.

```
interface AandB { a: string, b: string };

Pick<Partial<AandB>, “b”> // Result: { b?: string }
```

How to read a complex type definition


```
export type Optional<T, K extends keyof T> =  Pick<Partial<T>, K> & Omit<T, K>;
```

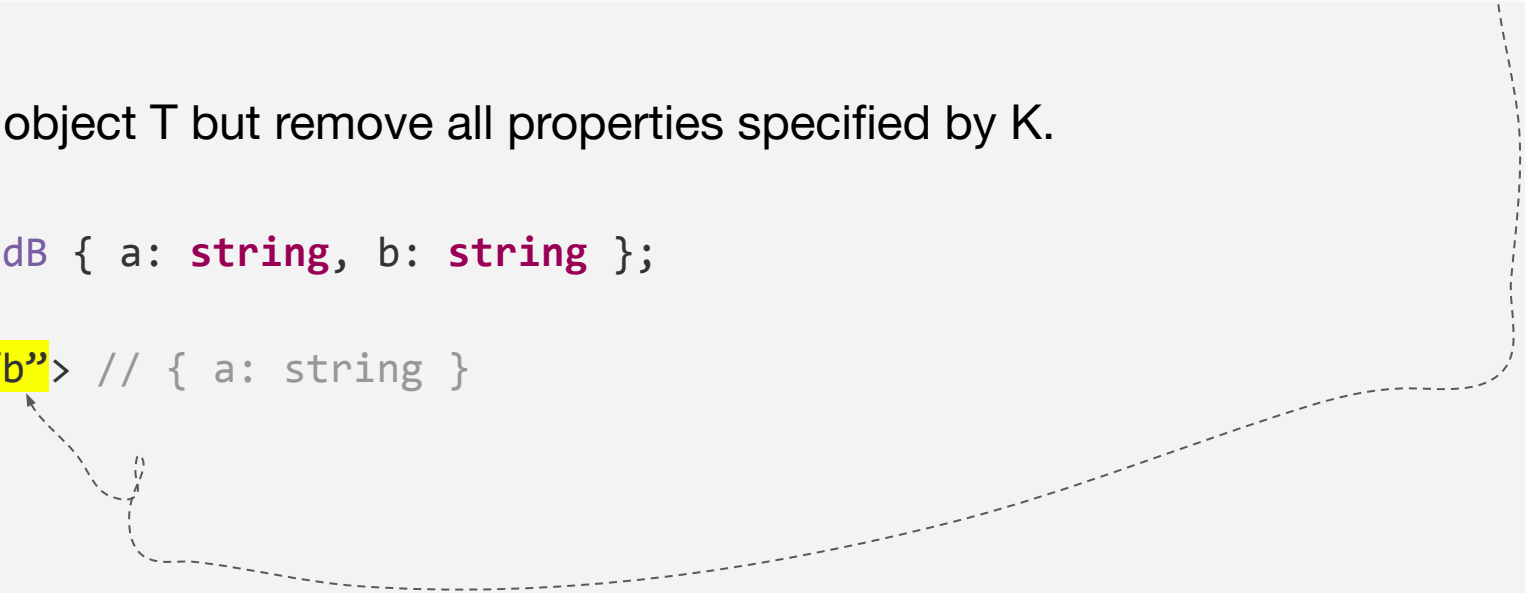
Combine the Result from Pick<...> with the Result form Omit<...>.

How to read a complex type definition


```
export type Optional<T, K extends keyof T> =  Pick<Partial<T>, K> & Omit<T, K>;
```

Use the given object T but remove all properties specified by K.

```
interface AandB { a: string, b: string };

Omit<AandB, “b”> // { a: string }
```



How to read a complex type definition


```
export type Optional<T, K extends keyof T> =  Pick<Partial<T>, K> & Omit<T, K>;
```

Put both results together:

```
// { b?: string }                    // { a: string }
```

How to read a complex type definition


```
export type Optional<T, K extends keyof T> =  Pick<Partial<T>, K> & Omit<T, K>;
```

The specified Key has been made optional.

```
{
  a: string,
  b?: string
}
```

Type Guards

Type Guards | built-in

<code>

With the JavaScript **typeof** operator we can check for a type

```
interface MayBeExecutable {
  work?: () => void; // possibly undefined
}

function executer(unit: MayBeExecutable) {
  if (typeof unit.work === 'function') {
    unit.work();
  }
}
```

Type Guards | built-in

Check for class instance with **instanceof**

```
const a = new Greeting();  // has method hi
const b = new Farewell();  // has method bye

function do(interaction: Greeting | Farewell) {
  if (interaction instanceof Greeting) {
     interaction.hi(); // type Greeting is inferred
  } else {
    interaction.bye(); // type Farewell is inferred,
                       // since no other type is possible
 }
}
```

Type Guards | Custom

Help the TypeScript compiler to understand your types better.

```ts
// Example 1
function isNumber(value: string | number): value is number {
  return !isNaN(value);
}

// Example 2
function isGoldCustomer(customer: NormalCustomer | GoldCustomer)
  : customer is GoldCustomer
{
  return customer.type === 'Gold';
}
```

Type Assertion (Type Cast)

Type Assertions (Type Casts)

Override the compiler's type checker. It comes in two syntax forms:

```
// Casting object literal to Person type: `as` syntax
const person = {
    name: 'Victoria',
    age: 28,
} as Person

// Casting object literal to Person: `ang-bracket` syntax
const person = <Person>{
    name: 'Franz',
    age: 34,
};
```

Type Assertions (Type Casts)

Dangerous use of type assertion: Can you be certain that persons array is not empty?

```ts
type Person = { name: string, age?: number };
// const persons = [ ... array of persons retrieved from API ];

// TS Error:   Type 'undefined' is not assignable to type 'Person'.
const getPerson = (name: string): Person =>
  persons.find(person => person.name === name);
```

```ts
// Solution 1: Union with undefined
const getPerson = (name: string): Person | undefined =>
  persons.find(person => person.name === name);

// Solution 2: Type assertion
const getPerson = (name: string): Person =>
  persons.find(person => person.name === name) as Person;
```

Type Assertions (Type Casts)

Valid use of type assertion: Help the compiler to understand what it could not do itself, but which you as the developer know for sure.

```
const deserialize = <T>(data: string): T => JSON.parse(data) as T;

const victoria = deserialize<Person>(
  '{"name":"Victoria", "age":28}'
);
```

keyof Operator

keyof Operator

<code>

Takes an object type and produces a literal union of its keys

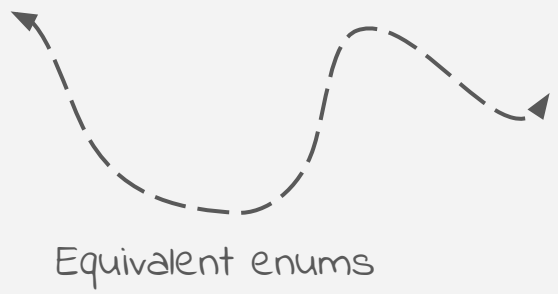
```ts
type Point = { x: number; y: number };
type P = keyof Point;
// Same as: type P = 'x' | 'y';

const x: P = 'x';
const y: P = 'y';

const z: P = 'z'; // Error: Type '"z"' is not assignable to type 'keyof Point'.
```

keyof with Enums

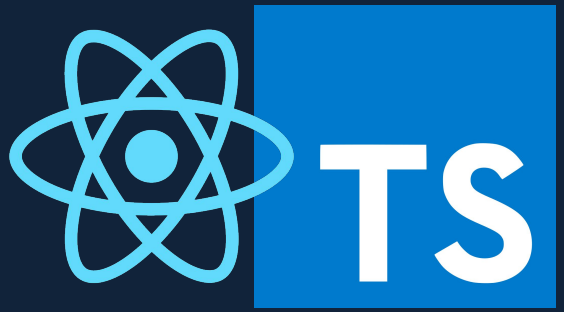
Create type out of enum

```ts
type LogLevelStrings = keyof typeof LogLevel;
// This is equivalent to:
// type LogLevelStrings = 'ERROR' | 'WARN' | 'INFO' | 'DEBUG';
```

```ts
enum LogLevel {
  ERROR,
  WARN,
  INFO,
  DEBUG
}
```




```ts
enum LogLevel {
  ERROR = 0,
  WARN = 1,
  INFO = 2,
  DEBUG = 3,
}
```



TypeScript & React

Generic functions and TSX

Generic Functions

Recap: Reverse list example

```
const reverse = <T>(items: T[]): T[] => {
  let result = [];
  for (let i = items.length - 1; i >= 0; i--) {
    result.push(items[i]);
  }
  return result;
}
```

Generic Functions & React

In `.tsx` files you have to differentiate between JSX syntax and TypeScript generics.

```
const reverse = <T,>(items: T[]): T[] => {
  let result = [];
  for (let i = items.length - 1; i >= 0; i--) {
    result.push(items[i]);
  }
  return result;
}
```

Generic Functions & React

In `.tsx` files you have to differentiate between JSX syntax and TypeScript generics - ***Alternative***.

```ts
const reverse = <T extends any>(items: T[]): T[] => {
  let result = [];
  for (let i = items.length - 1; i >= 0; i--) {
    result.push(items[i]);
  }
  return result;
}
```

Utility Types of the React Library

FunctionComponent

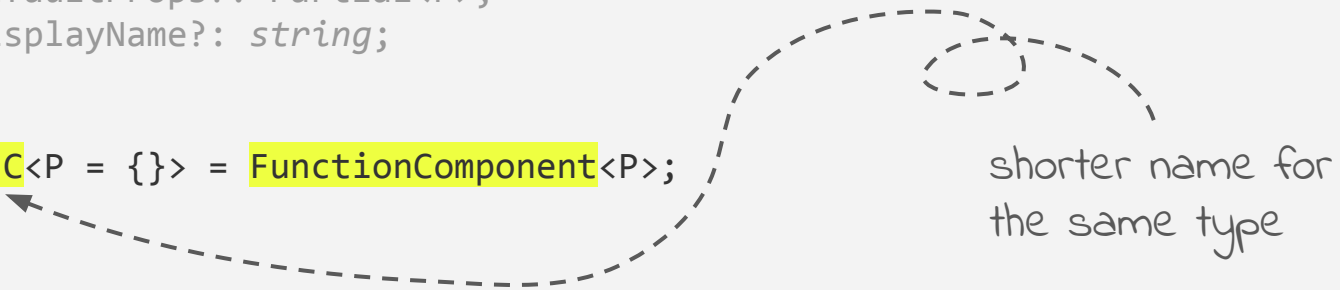
Utility Type which adds *children* prop to provided type of props

```
type ButtonPropsT = {
  variant?: string;
  onPress: (event: Event) => void;
};

const Button: FunctionComponent<ButtonPropsT> = ({
  children,
  variant = 'solid',
  onPress,
}) => (
  <Wrapper variant={variant} onPress={onPress}>
    <Title>{children}</Title>
  </Wrapper>
);
```

FunctionComponent

Interface **FunctionComponent** in *node_modules/@types/react/index.d.ts*

```ts
interface FunctionComponent<P = {}> {
    (props: PropsWithChildren<P>, context?: any): ReactElement<any, any> | null;
    propTypes?: WeakValidationMap<P>;
    contextTypes?: ValidationMap<any>;
    defaultProps?: Partial<P>;
    displayName?: string;
}

type FC<P = {}> = FunctionComponent<P>;
```



Type `useState` hook

Initial value does not give enough information for a strong type

```
const [items , setItems] = useState<string[]>([]);
```

Tips

Hint

Press **F8** in **VSCode** to jump from type error to type error to fix them.

Hint

Don’t try to type everything perfectly right from the start.

It’s ok to have some curly underlines

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

We teach.

workshops.de