TypeScript and React

What? Why? How?

Mateusz Wachowicz


Typescript 101

„**TypeScript** is an **open-source** programming language developed and maintained by Microsoft. It is a **strict syntactical superset** of JavaScript and adds **optional static typing** to the language. TypeScript is designed for development of large applications and **transcompiles to JavaScript.**[4] ”

~ **Wikipedia**

161%

Change in programming language use, 2018-2019

Source: octoverse.github.com

Why?

- Compile time safety
- Easier to understand
- Easier to refactor (especially in large codebases)
- Types! (duuh)
- Better autocompletion
- More and more job offers include TS knowledge
- More features

TypeScript

Types by Inference

```
let myString = "TypeScript is great!"
```

```
let myString: string
```

TypeScript
let vs const

```
const constString = "My String”
```

```
const constString: "My String"
```

```
let letString = "MyString"
```

```
let letString: string
```

TypeScript

Basic types

```
let myNum: number = 1
let myBool: boolean = true
let myString: string = "XD"
let myStringArray: string[] = ["Array", "of", "strings"]
```

The type names `String`, `Number`, and `Boolean` (starting with capital letters) are legal, but refer to some special built-in types that shouldn't appear in your code. *Always* use `string`, `number`, or `boolean`.

TypeScript

any, unknown, never

```
const myUnknown: unknown = "unknown is weird" // oh! it’s a string?
const myInvalidString: string = myUnknown


const myAny: any = "whatever" // I don't want typechecking thx
const myAnyButElsewhere: string = myAny // Zzz...


const throwException = (message: string): never => {
    throw new Error(message);
  }; // value is never returned, bc of thrown Exception

const pissOfGoogleChrome = (): never => {
  while (true) {
    console.log(„LÖÖÖÖÖÖÖÖÖÖÖÖÖÖP");
  }
}; // value is never returned, bc of loop
```

TypeScript
Functions

```
const myFunction = (arg1: string, arg2: number): void => {
    console.log(arg1, arg2)
}
```

TypeScript
Object Types

```
let myArrayGeneric: Array<number> = [1, 2, 3]

let myTuple: [number, string, boolean] = [1, "XD", true]

const fetchCat = (): Promise<Response> => fetch("lookatmycat.com/getcat")
```

TypeScript
Type Aliases

```
type AllowedNumbers = 1 | 2 | 3 | 4 | 5;

type StringOrNumberArray = string[] | number[];

type Person = {
  age: number;
  name: string;
};
```

TypeScript allows to create reusable aliases for every valid type.

TypeScript
Interfaces

```ts
interface Dog {
  age: number;
  name: string;
}

let myDog: Dog = {
  age: 3,
  name: "Brian",
};

const getMyDog = (): Dog => myDog;
```

```ts
interface MyFunc {
    (arg: string): string
}

const functionTypedWithInterface: MyFunc =
(arg) =>
    arg + "something"
```

TypeScript

Interface vs Types

Unique for **Interfaces**:

- Used with Objects, Classes and Functions
- Can be extended
- Declaration merging

Unique for **Type Aliases**:

- Just aliases for other types
- Can be combined with unions and intersections
- Primitive types

```ts
type Cat = {
  fluffy: true;
  age: number;
};
```

VS

```ts
interface Cat {
  fluffy: true;
  age: number;
}
```

TypeScript
as keyword

```ts
const mockDog = {
  age: 3,
};

const getDogAge = (dog: Dog) =>
  dog.age;

getDogAge(mockDog)
```

```
const mockDog: {
    age: number;
}
Argument of type '{ age: number; }' is not
assignable to parameter of type 'Dog'.
  Property 'name' is missing in type '{ age:
number; }' but required in type 'Dog'.ts(2345)
Interfaces.ts(3, 3): 'name' is declared here.
```

```ts
const mockDog = {
  age: 3,
} as Dog;

const getDogAge = (dog: Dog) =>
  dog.age;

getDogAge(mockDog) // 👌👌👌
```

TypeScript
Enums

```
enum Colors {
  RED,
  GREEN,
  BLUE,
}

console.log(Colors.RED);
// 0
```

```
enum Colors {
  RED = 1,
  GREEN,
  BLUE,
}

console.log(Colors.BLUE);
// Auto increment to 3
```

```
enum Colors {
  RED,
  GREEN,
  BLUE = "BLUE",
}

console.log(Colors.BLUE);
// BLUE
```

Enums are predefined sets of named constants and are compiled to JS at compile time.

TypeScript

Utility Types

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

Examples:

- Partial<T> : Constructs type with all props of type T set to optional
- Record<Keys, T> : Constructs a type with a set of properties `Keys` of type `Type`.
- Pick<T, Keys>: Constructs a type by picking the set of properties `Keys` from `Type`.
- Omit<T, Keys>: Constructs a type by picking all properties from `Type` and then removing `Keys`.

TypeScript

Type guards

Type guard is some expression that performs a runtime check that guarantees the type in some scope.

```
interface Fish {
  lives: boolean;
  swims: boolean;
}

interface Mouse {
  lives: boolean;
  swims: boolean;
  eatsCheese: boolean;
}

const getAnimal = (): Fish | Mouse => ({
  lives: true,
  swims: true,
  eatsCheese: true,
});

getAnimal().eatsCheese
```

```
Property 'eatsCheese' does not exist on type
'Fish | Mouse'.
  Property ,eatsCheese' does not exist on type
'Fish'.ts(2339)
```

TypeScript

Type guards: example solution - custom type guard

```ts
const isMouse = (animal: Fish | Mouse): animal is Mouse =>
  (animal as Mouse).eatsCheese !== undefined;

let myAnimal = getAnimal();
isMouse(myAnimal) && myAnimal.eatsCheese; // 👌👌👌
```

*Demo other solutions

TypeScript
Nullish Coalescing

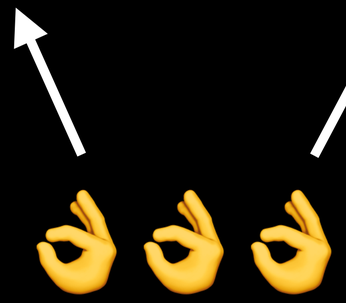
```
undefined || console.log("Undefined says: Nice 👌");
null || console.log("Null says: Nice 👌");
0 || console.log("0 says: Nice 👌");
"" || console.log("Empty string says: Nice 👌");
false || console.log("False says: Nice 👌");

// CONSOLE OUTPUT:
// Undefined says: Nice 👌
// Null says: Nice 👌
// 0 says: Nice 👌
// Empty string says: Nice 👌
// False says: Nice 👌
```

```
undefined ?? console.log("Undefined says: Nice 👌");
null ?? console.log("Null says: Nice 👌");
0 ?? console.log("0 says: Nice 👌");
"" ?? console.log("Empty string says: Nice 👌");
false ?? console.log("False says: Nice 👌");

// CONSOLE OUTPUT:
// Undefined says: Nice 👌
// Null says: Nice 👌
```

TypeScript
Generics

```
const identity = (arg: any):any => arg
```

```
const identity = <TypeVariable>(arg: TypeVariable):TypeVariable => arg
```



TypeScript

Generics: extending Type variables

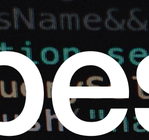
```
interface Coffee {
    milk: boolean;
    sugar: boolean
}

const hasMilk = <T>(myCoffee: T) => myCoffee.milk

const hasMilk = <T extends Coffee>(myCoffee: T) => myCoffee.milk    👌👌👌

hasMilk({
    milk: true,
    sugar: true,
    chocoChips: "yes, plz"
})
```

```
Property 'milk' does not exist on type
'T'.ts(2339)
```

TypeScript

tsconfig.json

```
{
  "compilerOptions": {
    "module": "system",
    "noImplicitAny": true,
    "removeComments": true,
    "preserveConstEnums": true,
    "outFile": "../../built/local/tsc.js",
    "sourceMap": true
  },
  "include": ["src/**/*"],
  "exclude": ["node_modules", "**/*.spec.ts"]
}
```

React + Typescript = 

React

Components

```tsx
interface MyButtonProps {
  text: string;
}

export const MyButton: React.FC<MyButtonProps> = ({ text }) => {
  return <button>{text}</button>;
};
```

It is advised to use function components with hooks instead of class components*

React

Hooks

Rules of Hooks 👮🏽

🚫 Calling hooks inside js functions, loops, conditions

✅ Calling hooks on top level inside React components

✅ Calling hooks inside custom hooks

React

Hooks: useState()

```
const [state, setState] = useState()
```

```
const state: undefined
```

```
const [state, setState] = useState(1)
```

```
const state: number
```

```
const [state, setState] = useState<number>()
```

```
const state: number
```

```
setState(newNumber)
```

Setting new state

```
setState(prevState => prevState + 21)
```

Setting new state using a callback

React

Hooks: useEffect()

```
useEffect(() => {
    console.log('Render!')
})


useEffect(() => {
    console.log('Render!')
},[])


const myString = "I like pizza 🍕";

useEffect(() => {
  console.log(myString);
}, [myString]);
```

React

Hooks: useEffect() cleanup

```
useEffect(() => {
    window.addEventListener("resize", handleResize);
    return () => {
      window.removeEventListener("resize", handleResize);
    };
  }, [handleResize]);
```

Function passed to useEffect may return a clean-up function.

React

Hooks: useContext()

```
const ContextObj = React.createContext({
    info: "very important info”,
  });

const { info } = useContext(ContextObj);
```

NOTE: We’re passing whole context object inside the useContext hook!

React

Hooks: useReducer()

Setup:

- Reducer function
- Initial state

```
enum CounterAction {
  INCREMENT,
  DECREMENT,
}

const initialState = { count: 0 };

const reducer = (
  state: typeof initialState,
  action: { type: CounterAction,
            payload?: string }
) => {
  switch (action.type) {
    case CounterAction.INCREMENT:
      return { count: state.count + 1 };
    case CounterAction.DECREMENT:
      return { count: state.count - 1 };
  }
};
```

React

Hooks: useReducer()

```
const Counter = () => {
  const [state, dispatch] = useReducer(reducer, initialState);
  return (
    <>
      Count: {state.count}
      <button onClick={() => dispatch({ type: CounterAction.DECREMENT })}>
        -
      </button>
      <button onClick={() => dispatch({ type: CounterAction.INCREMENT })}>
        +
      </button>
    </>
  );
};
```

React

Hooks: useCallback() & useMemo()

```
const memoizedWithCallback = useCallback(() => {
  doSomething(a, b);
}, [a, b]);
```

```
const memoizedWithMemo = useMemo(
  () => () => {
    doSomething(a, b);
  },
  [a, b]
);
```

React

Hooks: useRef()

```
const myRef = useRef(1)
console.log(myRef) // {current: 1}
```

This variable will persist between renders

„useRef is like a “box” that can hold a mutable value in its .current property.”

~React Docs

React

Hooks: custom

You can write your own custom hooks

Demo time

React
Context

Provided value can be anything, even a function, return value from useState or useReducer!

```
const MyContext = React.createContext(null)

const ParentComponent = () => {
    return (
        <MyContext.Provider value={'Cats are cute'}>
            <ChildComponent />
        </MyContext.Provider>
    )
}

const ChildComponent = () => {
    const importantMessage = useContext(MyContext)

    return(
        <h1>
            {importantMessage}
        </h1>
    )
}
```

React
Redux

```
const store = createStore(counterReducer)
```

```
ReactDOM.render(
  <Provider store={store}>
    <App />
  </Provider>,
  rootElement
)
```

React

Redux: Actions

```
enum ActionTypes {
  INCREMENT,
  DECREMENT,
}

interface Action {
  type: ActionTypes;
  payload?: string;
}
```

```
const decrementCounter = (payload?: string): Action =>
({
  type: ActionTypes.DECREMENT,
  payload,
});


const incrementCounter = (payload?: string): Action =>
({
  type: ActionTypes.INCREMENT,
  payload,
});
```

React

Redux: Reducer

```
interface CounterState {
  count: number;
  note: string;
}
```

```
const initialState: CounterState = {
  count: 0,
  note: "",
};
```

```
const counterReducer = (state: CounterState = initialState, action: Action) => {
  switch (action.type) {
    case ActionTypes.INCREMENT:
      return {
        ...state,
        count: state.count += 1,
        note: action.payload ?? state.note,
      };
    case ActionTypes.DECREMENT:
      return {
        ...state,
        count: state.count -= 1,
        note: action.payload ?? state.note,
      };
  }
};
```

React

Redux: Consume state

Get state using useSelector

```
const count =
     useSelector<CounterState, CounterState["count"]>(state => state.count)
```

Change state using useDispatch

```
const dispatch = useDispatch()

dispatch(incrementCounter("User #324 entered the room"))
```

CSS-in-JS

Emotion 

„Emotion is a library designed for writing css styles with JavaScript.”

CSS-in-JS

Emotion 🦹‍♀️ : css function

```
const color = 'white'

<div
  className={css`
    padding: 32px;
    background-color: blue;
    font-size: 24px;
    border-radius: 4px;
    &:hover {
      color: ${color};
    }
  `}
>
  Hover to change color.
</div>
```

```
const color = 'white'

<div
  className={css({
    padding: 32,
    backgroundColor: "blue",
    fontSize: 24,
    borderRadius: 4,
    '&:hover': {
      color: color,
    }
  })}
>
  Hover to change color.
</div>
```

CSS-in-JS

Emotion  : cx function

```
const flex = css({
    display: 'flex'
})

const column = css({
    flexDirection: 'column',
    justifyContent: 'center',
    alignItems: 'center'
})

<div className={cx(flex, column)}>
    [...]
</div>
```

CSS-in-JS

Emotion 💅 : keyframes function

```
const bounce = keyframes`
  from, 20%, 53%, 80%, to {
    transform: translate3d(0,0,0);
  }

  40%, 43% {
    transform: translate3d(0, -30px, 0);
  }

  70% {
    transform: translate3d(0, -15px, 0);
  }

  90% {
    transform: translate3d(0,-4px,0);
  }
`
```

```
<div
  className={css({
    animation: `${bounce} 1s ease infinite`
  })}
>
  some bouncing text!
</div>
```

Q&A

Questions?

Thank you!