TypeScript Class Notes

AltSchool Africa

Are you ready to learn TypeScript? Press `space` on your keyboard →

What is TypeScript?

TypeScript is a strongly typed programming language that builds on JavaScript, giving you better tooling at any scale.

Read more about TypeScript?

What is TypeScript?

TypeScript is a strongly typed programming language that builds on JavaScript, giving you better tooling at any scale.

JavaScript and More

TypeScript adds additional syntax to JavaScript to support a tighter integration with your editor. Catch errors early in your editor. A Result You Can Trust

Read more about TypeScript?

What is TypeScript?

TypeScript is a strongly typed programming language that builds on JavaScript, giving you better tooling at any scale.

JavaScript and More

TypeScript adds additional syntax to JavaScript to support a tighter integration with your editor. Catch errors early in your editor. A Result You Can Trust

TypeScript code converts to JavaScript, which runs anywhere JavaScript runs: In a browser, on Node.js or Deno and in your apps. Safety at Scale

Read more about TypeScript?

What is TypeScript?

TypeScript is a strongly typed programming language that builds on JavaScript, giving you better tooling at any scale.

JavaScript and More

TypeScript adds additional syntax to JavaScript to support a tighter integration with your editor. Catch errors early in your editor. A Result You Can Trust

TypeScript code converts to JavaScript, which runs anywhere JavaScript runs: In a browser, on Node.js or Deno and in your apps. Safety at Scale

TypeScript understands JavaScript and uses type inference to give you great tooling without additional code

Read more about TypeScript?

Table of Content

What are the things we will be covering?

Table of Content

What are the things we will be covering?

The Basics

- Static type-checking
- Non-exception Failures
- Types for Tooling
- tsc, the TypeScript compiler
- Emitting with Errors
- Explicit Types
- Erased Types
- Downleveling
- Strictness
- noImplicitAny
- strictNullChecks

TypeScript Compiler `tsc`

- The TypeScript compiler is a tool that takes TypeScript code and turns it into JavaScript code.
- The TypeScript compiler can be installed as a Node.js package.
- The TypeScript compiler can be run from the command line.
- The TypeScript compiler can be configured using a configuration file.
- The TypeScript compiler can be used to compile multiple files.
- The TypeScript compiler can be used to compile a project.

```
npm install -g typescript

tsc hello.ts

tsc --noEmitOnError hello.ts

tsc --init
```

ni

tsconfig.json

tsconfig.json

- The tsconfig.json file is a configuration file that tells the TypeScript compiler how to compile your TypeScript code.

tsconfig.json

- The tsconfig.json file is a configuration file that tells the TypeScript compiler how to compile your TypeScript code.
- Strictness: You can use the strict flag to enable all strict type-checking options or in the config file. You can opt out of strictness by setting strict to false or `noImplicitAny` to false and `strictNullChecks` to false.

tsconfig.json

- The tsconfig.json file is a configuration file that tells the TypeScript compiler how to compile your TypeScript code.
- Strictness: You can use the strict flag to enable all strict type-checking options or in the config file. You can opt out of strictness by setting strict to false or `noImplicitAny` to false and `strictNullChecks` to false.
- Downleveling: You can use the target flag to specify the version of JavaScript that the TypeScript compiler should output. The default is ES3.

tsconfig.json

- The tsconfig.json file is a configuration file that tells the TypeScript compiler how to compile your TypeScript code.
- Strictness: You can use the strict flag to enable all strict type-checking options or in the config file. You can opt out of strictness by setting strict to false or `noImplicitAny` to false and `strictNullChecks` to false.
- Downleveling: You can use the target flag to specify the version of JavaScript that the TypeScript compiler should output. The default is ES3.
- Emitting with Errors: You can use the noEmitOnError flag to prevent the TypeScript compiler from emitting JavaScript code if there are any errors.

tsconfig.json

- The tsconfig.json file is a configuration file that tells the TypeScript compiler how to compile your TypeScript code.
- Strictness: You can use the strict flag to enable all strict type-checking options or in the config file. You can opt out of strictness by setting strict to false or `noImplicitAny` to false and `strictNullChecks` to false.
- Downleveling: You can use the target flag to specify the version of JavaScript that the TypeScript compiler should output. The default is ES3.
- Emitting with Errors: You can use the noEmitOnError flag to prevent the TypeScript compiler from emitting JavaScript code if there are any errors.
- Explicit Types: You can use the noImplicitAny flag to prevent TypeScript from inferring the any type.

tsconfig.json

- Erased Types: You can use the noUnusedLocals and noUnusedParameters flags to prevent TypeScript from emitting JavaScript code if there are any unused variables or parameters.

```
{
  "compilerOptions": {
    "strict": true,
    "noImplicitAny": true,
    "strictNullChecks": true,
    "target": "ES5",
    "noEmitOnError": true
  }
}
```

Everyday Types

The primitives, any, Type annotations on variables, Functions, Object types, Union Types, Type Aliases, Interfaces, Type Assertions, Literal Types, null and undefined, Enums

```
let person: string | number = "OjoT99";

if (typeof person === "string") {
  person.split("T");
} else {
  // only number
  // person.toFixed(2);
}

let age: number = 99;

let isAltSchoolStudent = false;
let nothing = null;
let something = undefined;
```

Everyday Types

The primitives, any, Type annotations on variables, Functions, Object types, Union Types, Type Aliases, Interfaces, Type Assertions, Literal Types, null and undefined, Enums

```
let arrayOfScores = [99, 45, 56, 67, 99];
let arrayOfScores2: number[] = [99, 45, 56, 67, 99];

let arrayOfNames: string[] = ["bisi", "sola", "augustina", "typescritina"];

let arrayOfTruths = [true, false];

let names: Array<string> = ["dancing", "eating", "sleeping"];
// <> -> generics  Array<number> Array<boolean> Array<null>

let arrayInsideArrays = [["a"], ["b"]];

let newArr = [undefined];
```

@Oluwasetemi

Everyday Types

The primitives, any, Type annotations on variables, Functions, Object types, Union Types, Type Aliases, Interfaces, Type Assertions, Literal Types, null and undefined, Enums

```
let obj: { name: string; age: number; job?: string } = {
  name: "ade",
  age: 99,
};

function greet(msg: string): string {
  return msg + "Hi :dance:";
}

if (typeof obj.job === "string") {
  // typeguard
  greet(obj.job);
} else {
  // strictly undefined
  obj.job;
}
```

Everyday Types

The primitives, any, Type annotations on variables, Functions, Object types, Union Types, Type Aliases, Interfaces, Type Assertions, Literal Types, null and undefined, Enums

```
let profile: Record<string, number> = {
  age: 99,
  height: 6,
  weight: 100,
};

let objFlex: Record<string | symbol, string | boolean | number> = {};

objFlex.name = "lagbaja";
objFlex.animal = "cat";
objFlex[Symbol("id")] = true;
// any or never
//
let objFlexNumber: Record<string, number> = {
  age: 99,
};
```

Everyday Types

The primitives, any, Type annotations on variables, Functions, Object types, Union Types, Type Aliases, Interfaces, Type Assertions, Literal Types, null and undefined, Enums

```
// mixing types
const specialArr: Array<number | string | [] | {}> = [
  "name",
  99,
  {},
  [],
  "ginia",
  100,
];

let result: number[] = person.split("T");

result;

console.log(result);
console.log("Hello", "AltSchool");
```

Everyday Types

The primitives, any, Type annotations on variables, Functions, Object types, Union Types, Type Aliases, Interfaces, Type Assertions, Literal Types, null and undefined, Enums

```
let user: "student" | "admin";

user = "temi";

user = "admin";
```

Everyday Types

The primitives, any, Type annotations on variables, Functions, Object types, Union Types, Type Aliases, Interfaces, Type Assertions, Literal Types, null and undefined, Enums

```
function add(): number {
  console.log("hello");
  return 99;
}

// typing arguments
function add2(a: number, b: number): number {
  return a + b;
}

add2(99, 78);
```

Everyday Types

The primitives, any, Type annotations on variables, Functions, Object types, Union Types, Type Aliases, Interfaces, Type Assertions, Literal Types, null and undefined, Enums

```
// function overloading

function add3(a: number, b: number): number;
function add3(a: string, b: string): string;
function add3(a: any, b: any): any {
  return a + b;
}

add3("na", "me");
add3(99, 78);
let name2: any = "wale";
let age2: any = 99;
add3(name2, age2);
```

Everyday Types

The primitives, any, Type annotations on variables, Functions, Object types, Union Types, Type Aliases, Interfaces, Type Assertions, Literal Types, null and undefined, Enums

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

let person2: Person = {
  name: "ade",
  age: 99,
};
```

Everyday Types

The primitives, any, Type annotations on variables, Functions, Object types, Union Types, Type Aliases, Interfaces, Type Assertions, Literal Types, null and undefined, Enums

```
// interface
interface Person2 {
  name: string;
  age: number;
}

function greet2(person: Person2): string {
  return `Hello ${person.name}`;
}

greet2({ name: "ade", age: 99 });
```

Everyday Types

The primitives, any, Type annotations on variables, Functions, Object types, Union Types, Type Aliases, Interfaces, Type Assertions, Literal Types, null and undefined, Enums

```
// type assertion
let res = JSON.parse('{"name": "ade"}') as { name: string };
```

Everyday Types

The primitives, any, Type annotations on variables, Functions, Object types, Union Types, Type Aliases, Interfaces, Type Assertions, Literal Types, null and undefined, Enums

```
// 'satifies', 'as const', '!'
```

Everyday Types

The primitives, any, Type annotations on variables, Functions, Object types, Union Types, Type Aliases, Interfaces, Type Assertions, Literal Types, null and undefined, Enums

```
const addTwoNumbers = (a: number, b: number): number => {
  return a + b;
};

interface Params {
  a: number;
  b: number;
}

const addTwoNumberObject = (params: { a: number; b: number }): number => {
  return params.a + params.b;
};
```

Everyday Types

The primitives, any, Type annotations on variables, Functions, Object types, Union Types, Type Aliases, Interfaces, Type Assertions, Literal Types, null and undefined, Enums

```
// extending the inteface(obj only!)
interface ThreeParams extends Params {
  c: number;
}
// conditional type
type NewParams = ThreeParams extends Params ? string : number;

const addThreeNumberObject = (params: ThreeParams): number => {
  return params.a + params.b + params.c;
};

addThreeNumberObject({ a: 99, b: 78, c: 100 });
```

Everyday Types

The primitives, any, Type annotations on variables, Functions, Object types, Union Types, Type Aliases, Interfaces, Type Assertions, Literal Types, null and undefined, Enums

```
// make b optional
const addTwoNumberObject2 = (params: { a: number; b?: number }): number => {
  if (params.b) {
    return params.a + params.b;
  }
  return params.a;
};

console.log(addTwoNumberObject2({ a: 99 }));
```

Everyday Types

The primitives, any, Type annotations on variables, Functions, Object types, Union Types, Type Aliases, Interfaces, Type Assertions, Literal Types, null and undefined, Enums

```
const addTwoNumberObject3 = (params: { a?: number; b?: number }): number => {
  if (params.a) {
    return params.a;
  }

  if (params.b) {
    return params.b;
  }

  return 5;
};

addTwoNumberObject3({});
```

Everyday Types

The primitives, any, Type annotations on variables, Functions, Object types, Union Types, Type Aliases, Interfaces, Type Assertions, Literal Types, null and undefined, Enums

```
const addTwoNumber3 = (a: number = 2, b: number = 5) => {
  return a + b;
};

addTwoNumber3();

type Admin = {
  name: boolean;
};
```

Everyday Types

```
function getPersonName(admin: Admin) {
  return admin.name;
}
getPersonName({ name: false });

type AdminModified = {
  name: string;
  role: "client" | "admin" | "superadmin";
};

function getPersonString(admin: AdminModified) {
  return `${admin.name} is a ${admin.role}`;
}

getPersonString({ name: "ken", role: "superadmin" });
```

Everyday Types

```
function getPersonName(admin: Admin) {
  return admin.name;
}
getPersonName({ name: false });

type AdminModified = {
  name: string;
  role: "client" | "admin" | "superadmin";
};

function getPersonString(admin: AdminModified) {
  return `${admin.name} is a ${admin.role}`;
}

getPersonString({ name: "ken", role: "superadmin" });
```

Everyday Types

```
function getPersonName(admin: Admin) {
  return admin.name;
}
getPersonName({ name: false });

type AdminModified = {
  name: string;
  role: "client" | "admin" | "superadmin";
};

function getPersonString(admin: AdminModified) {
  return `${admin.name} is a ${admin.role}`;
}

getPersonString({ name: "ken", role: "superadmin" });
```

Everyday Types

```
function getPersonName(admin: Admin) {
  return admin.name;
}
getPersonName({ name: false });

type AdminModified = {
  name: string;
  role: "client" | "admin" | "superadmin";
};

function getPersonString(admin: AdminModified) {
  return `${admin.name} is a ${admin.role}`;
}

getPersonString({ name: "ken", role: "superadmin" });
```

Everyday Types

```
function getPersonName(admin: Admin) {
  return admin.name;
}
getPersonName({ name: false });

type AdminModified = {
  name: string;
  role: "client" | "admin" | "superadmin";
};

function getPersonString(admin: AdminModified) {
  return `${admin.name} is a ${admin.role}`;
}

getPersonString({ name: "ken", role: "superadmin" });
```

Everyday Types

```
function getPersonName(admin: Admin) {
  return admin.name;
}
getPersonName({ name: false });

type AdminModified = {
  name: string;
  role: "client" | "admin" | "superadmin";
};

function getPersonString(admin: AdminModified) {
  return `${admin.name} is a ${admin.role}`;
}

getPersonString({ name: "ken", role: "superadmin" });
```

Everyday Types

```
function getPersonName(admin: Admin) {
  return admin.name;
}
getPersonName({ name: false });

type AdminModified = {
  name: string;
  role: "client" | "admin" | "superadmin";
};

function getPersonString(admin: AdminModified) {
  return `${admin.name} is a ${admin.role}`;
}

getPersonString({ name: "ken", role: "superadmin" });
```

Everyday Types

```
type Post = {
  title: string;
  author: string;
  id: number;
  body: string;
};

type AdminWithPosts = {
  posts: Array<Post>;
  name: string;
  role: "client" | "admin" | "superadmin";
};

function getPersonPost(person: AdminWithPosts): Array<Post> {
  return person.posts;
}
```

Everyday Types

```
let res = getPersonPost({
  posts: [
    {
      title: "hello",
      author: "ken",
      id: 99,
      body: "hello blogpost content ",
    },
  ],
  name: "ken",
  role: "admin",
});
console.log(res);
```

Everyday Types

```
type NewPost = keyof (typeof res)[0]; // "title" | "author" | "id" | "body"
let newPostKey: NewPost = "author";
console.log(newPostKey);
```

Everyday Types

```
type GitHubUser = {
  login: string;
  id: number;
  node_id: string;
  avatar_url: string;
  gravatar_id: string;
  url: string;
  html_url: string;
  followers_url: string;
  following_url: string;
  gists_url: string;
  starred_url: string;
  subscriptions_url: string;
  organizations_url: string;
  repos_url: string;
  events_url: string;
  received_events_url: string;
  type: string;
  site_admin: boolean;
  name: string;
};
```

Everyday Types

```
type NewGitHub = Pick<GitHubUser, "login" | "id" | "node_id">;

let newGitHub: NewGitHub = {
  login: "ade",
  id: 99,
  node_id: "node_id",
};

type newGitHubModified = Omit<NewGitHub, "node_id">;

let newGitHubModified: newGitHubModified = {
  login: "ade",
  id: 99,
};
```

Everyday Types

```
async function fetchGitHubUser(username: string) {
  return fetch(`https://api.github.com/users/${username}`).then((res) =>
    res.json(),
  );
}


// (async () => {
//   let githubUser = await fetchGitHubUser("Oluwasetemi");
//   console.log(githubUser.avatar_url);
// })();
```

@Oluwasetemi

Everyday Types

```
const listOfStudent = new Set<string>();
listOfStudent.add("ade");
listOfStudent.add("ade");

listOfStudent.has("ade");

console.log(listOfStudent);

let mapOfStudentToScores = new Map<string, number>();
mapOfStudentToScores.set("ade", 99);
console.log(mapOfStudentToScores);
mapOfStudentToScores;
```

Everyday Types

```
// tuples
let tuple: [string, number] = ["ade", 99];

let color: [number, number, number, number?];

color = [255, 0, 0, 0.1];
// rgba

let colorString = `rgb(${color.join(", ")})`;
```

Everyday Types

```ts
// unions |
let str: number | string;

// at the level of types and interface
let advancePostU: Post | { tags: string[] } = {
  title: "hello",
  id: 1,
  author: "Authur Ts",
  body: "hello body",
  tags: ['hello', 'world']
};

// intersection &
type Tags = { tags: string[] };
let advancePost: Post & Tags = {
  title: "hello",
  id: 1,
  author: "Authur Ts",
  body: "hello body",
  tags: ["hello", "world"],
};
```

Everyday Types

```
let NewStringIndex: { [index: number]: string };

NewStringIndex = ["1", "2", "3", "4", "5"];

// NewStringIndex = {
//   name: "ade",
//   age: "99",
// };

NewStringIndex[0] = "hello";

NewStringIndex["job"] = "developer";
```

Everyday Types

```
// readonly
let arrOfCommenter: readonly string[] = ["ade", "bisi", "sola"];

arrOfCommenter.push("aderemi");

let arrOfCommenter2: ReadonlyArray<string> = ["ade", "bisi", "sola"];
arrOfCommenter2.push("aderemi");
```

Everyday Types

```
function longest<Type extends { length: number }>(a: Type, b: Type) {
  if (a.length >= b.length) {
    return a;
  } else {
    return b;
  }
}

let res34 = longest({ length: 4 }, { length: 6 });

function merge<T, U>(firstObject: T, secondObject: U): T & U {
  return {
    ...firstObject,
    ...secondObject,
  };
}

let res35 = merge({ name: "ade" }, { age: 99 });
let res37 = merge({ school: "AltSchool" }, { job: "cleaner" });
```

Everyday Types

```
// enums - user (ADMIN, CLIENT, SUPERADMIN)
enum Role {
  ADMIN,
  CLIENT,
  SUPERADMIN,
}

type User = {
  id: string;
  // enum
  role: Role;
  // union types
  // role: "CLIENT" | "ADMIN" | "SUPERADMIN";
  name: string;
  address: string;
};
```

Everyday Types

```
function checkUserRole(user: User): string {
  const { role } = user;
  if (role === Role.ADMIN) {
    return "admin";
  } else if (role === Role.CLIENT) {
    return "client";
  }
  return "superadmin";
}

let userAltSchool: User = {
  id: "001",
  role: Role.ADMIN,
  name: "ade ojo",
  address: "lagos",
};

let resultAltSchool = checkUserRole(userAltSchool);
console.log(resultAltSchool);
```

Everyday Types

```
// Type manipulation - keyof, typeof, in, infer, extends, in, as, is, &

type U = keyof {x: string, y: number} // 'x' | 'y'
type KeyOfUserType = keyof User;
type Arrayish = { [n: number]: string }; // string[]
type keyOfArray = keyof Arrayish;
let sampleArray: { [n: number]: string } = ["ade", "bisi", "sola"];

let keyOfUser: KeyOfUserType = "name";
```

Everyday Types

```ts
// typeof
let myName = "ade";
type Name = typeof myName;

type Predicate = (x: unknown) => boolean;
type K = ReturnType<Predicate>;

type CheckUserRole = ReturnType<typeof checkUserRole>;

function f() {
  return { x: 10, y: 3 };
}
// infer
type P = ReturnType<typeof f>;
```

Everyday Types

```ts
// indexed access types
type Person3 = { name: string; age: number; address: string };
type Age = Person3["address" | "age"];

// Conditional Types
// SomeType extends OtherType ? TrueType : FalseType
type Exclude<T, U> = T extends U ? never : T;
// type T = Exclude<"a" | "b" | "c", "a" | "c">; // "b"
```

Everyday Types

```ts
// mapped types
type Person4 = {
  [key: string]: string;
};

// Template Literal Types
type World = "world";
type Greeting = `hello ${World}`;
```

```ts
let person: string | number = "helloTtypescript";

let result: number[] = person.split("T");
```

```
Type 'string[]' is not assignable to type 'number[]'.
  Type 'string' is not assignable to type 'number'.
```

```ts
//  //^?

console.log(result);
console.log("Hello", "AltSchool");
```

functions

```
function greeter(fn: (a: string) => void) {
  fn("Hello, World");
}

function printToConsole(s: string) {
  console.log(s);
}

greeter(printToConsole);
```

functions

```
type GreetFunction = (a: string) => void;

function greeter(fn: GreetFunction) {
  fn("Hello, World");
}

function printToConsole(s: string) {
  console.log(s);
}

greeter(printToConsole);
```

functions

```
type DescribableFunction = {
  description: string;
  (someArg: number): boolean;
};

function doSomething(fn: DescribableFunction) {
  console.log(fn.description + " returned " + fn(6));
};

function myFunc(someArg: number) {
  return someArg > 3;
}
myFunc.description = "default description";

doSomething(myFunc);
```

functions

```
// call signatures and constructors
type DescribableFunction = {
  description: string;
  (someArg: number): boolean;
};

type SomeConstructor = {
  new (s: string): SomeObject;
};
function fn(ctor: SomeConstructor) {
  return new ctor("hello");
}
```

```ts
type DescribableFunction = {
  description: string;
  (someArg: number): boolean;
};
function doSomething(fn: DescribableFunction) {
  console.log(fn.description + " returned " + fn(6));
}

function myFunc(someArg: number) {
  return someArg > 3;
}
myFunc.description = "default description";

doSomething(myFunc);
```

```
default description returned true
```

Peek into Generics

```ts
// Inside ./snippets/external.ts
export function emptyArray<T>(length: number) {
  return Array.from<T>({ length })
}
```

Put `emptyArray` function to work

Peek into Generics

```ts
// Inside ./snippets/external.ts
export function emptyArray<T>(length: number) {
  return Array.from<T>({ length })
}
```

Put `emptyArray` function to work

```ts
import { emptyArray } from './external'

console.log(emptyArray<number>(10).reduce(fib => [...fib, fib.at(-1)! + fib.at(-2)!], [1, 1]))
```

```
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144]
```

Peek into Generics

```
// Inside ./snippets/external.ts
export function emptyArray<T>(length: number) {
  return Array.from<T>({ length })
}
```

Put `emptyArray` function to work

```
import { emptyArray } from './external'

console.log(emptyArray<number>(10).reduce(fib => [...fib, fib.at(-1)! + fib.at(-2)!], [1, 1]))
```

```
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144]
```

```
function firstElement<Type>(arr: Type[]): Type | undefined {
  return arr[0];
}
// Note that we didn't have to specify Type in this sample.
// The type was inferred - chosen automatically - by TypeScript.
let s1 = firstElement([1, 2, 4, 5])
let s2 = firstElement(['hello', 'dance'])
```

ni

Solve this using TS Generics

```
function getRandomNumberElement(items: number[]): number {
  let randomIndex = Math.floor(Math.random() * items.length);
  return items[randomIndex];
}

let randyValue = getRandomNumberElement(['ayo', 'ade', 'ojo', 'jerry'])

console.log(randyValue)
```

```
ade
```

@Oluwasetemi

```
function map<Input, Output>(arr: Input[], func: (arg: Input) => Output): Output[] {
  return arr.map(func);
}

// Parameter 'n' is of type 'string'
// 'parsed' is of type 'number[]'
const parsed = map(["1", "2", "3"], (n) => parseInt(n));
```

```ts
function map<Input, Output>(arr: Input[], func: (arg: Input) => Output): Output[] {
  return arr.map(func);
}

// Parameter 'n' is of type 'string'
// 'parsed' is of type 'number[]'
const parsed = map(["1", "2", "3"], (n) => parseInt(n));
```

```ts
function longest<Type extends { length: number }>(a: Type, b: Type) {
  if (a.length >= b.length) {
    return a;
  } else {
    return b;
  }
}

// longerArray is of type 'number[]'
const longerArray = longest([1, 2], [1, 2, 3]);
// longerString is of type 'alice' | 'bob'
const longerString = longest("alice", "bob");
// Error! Numbers don't have a 'length' property
const notOK = longest(10, 100);
```

```
Argument of type 'number' is not assignable to parameter of type '{ length: number; }'.
```

```ts
function minimumLength<Type extends { length: number }>(
  obj: Type,
  minimum: number
): Type {
  if (obj.length >= minimum) {
    return obj;
  } else {
    return { length: minimum };
  }
}

// 'arr' gets value { length: 6 }
const arr = minimumLength([1, 2, 3], 6);
// and crashes here because arrays have
// a 'slice' method, but not the returned object!
console.log(arr.slice(0));
```

```
TypeError: arr.slice is not a function
```

```ts
function combine<Type>(arr1: Type[], arr2: Type[]): Type[] {
  return arr1.concat(arr2);
}

// const arr = combine([1, 2, 3], ["hello"]);
const arr = combine<string | number>([1, 2, 3], ["hello"]);

console.log(arr)
```

```
[1, 2, 3, "hello"]
```

```ts
function merge<T, U>(firstObject: T, secondObject: U): T & U {
  return {
    ...firstObject,
    ...secondObject,
  };
}

type Result<T extends Function> = T extends (...args: never[]) => infer R
  ? R
  : never;

let res35 = merge({ name: "ade" }, { age: 99 });
console.log(res35)
let res37 = merge({ school: "AltSchool" }, { job: "cleaner" });
console.log(res37)
```

```
{
  "name": "ade",
  "age": 99
}
{
  "school": "AltSchool",
  "job": "cleaner"
}
```

@Oluwasetemi

```ts
type FuncWithOneObjectArgument<P extends { [x: string]: any }, R> = (
  props: P
) => R;

type DestructuredArgsOfFunction<
  F extends FuncWithOneObjectArgument<any, any>
> = F extends FuncWithOneObjectArgument<infer P, any> ? P : never;

const myFunction = (props: { x: number; y: number }): string => {
  return "OK";
};

const props: DestructuredArgsOfFunction<typeof myFunction> = {
  x: 1,
  y: 2
};
```

Click the play button to run the code

@Oluwasetemi

Guideline to writing good generics

Guideline to writing good generics

- Push Type Parameters Down

Guideline to writing good generics

- Push Type Parameters Down
- Use Fewer Type Parameters

```
function filter1<Type>(arr: Type[], func: (arg: Type) => boolean): Type[] {
  return arr.filter(func);
}

function filter2<Type, Func extends (arg: Type) => boolean>(
  arr: Type[],
  func: Func
): Type[] {
  return arr.filter(func);
}

const val = filter1([1, 2, 3, 4], n => n % 2 === 0)
const val2 = filter2([1, 2, 3, 4], n => n % 2 === 0)
```

Guideline to writing good generics

- Push Type Parameters Down
- Use Fewer Type Parameters

```
function filter1<Type>(arr: Type[], func: (arg: Type) => boolean): Type[] {
  return arr.filter(func);
}

function filter2<Type, Func extends (arg: Type) => boolean>(
  arr: Type[],
  func: Func
): Type[] {
  return arr.filter(func);
}

const val = filter1([1, 2, 3, 4], n => n % 2 === 0)
const val2 = filter2([1, 2, 3, 4], n => n % 2 === 0)
```

- Type Parameters(Or any annotation used) Should Appear Twice

function overloading

```ts
function add3(a: number, b: number): number;
function add3(a: string, b: string): string;
function add3(a: any, b: any): any {
  return a + b;
}

add3("na", "me");
add3(99, 78);
let name2: any = "wale";
let age2: any = 99;
add3(name2, age2);
```

Enums

```
enum Role {
  ADMIN,
  CLIENT,
  SUPERADMIN,
}

type User = {
  id: string;
  // enum
  role: Role;
  // union types
  // role: "CLIENT" | "ADMIN" | "SUPERADMIN";
  name: string;
  address: string;
};
```

```ts
enum Role { ADMIN, CLIENT, SUPERADMIN, };
type User = { id: string; role: Role; name: string; address: string; };
// union types // role: "CLIENT" | "ADMIN" | "SUPERADMIN";

function checkUserRole(user: User): string {
  const { role } = user;
  if (role === Role.ADMIN) {
    return "admin";
  } else if (role === Role.CLIENT) {
    return "client";
  }
  // Role.SUPERADMIN;
  return "superadmin";
}

let userAltSchool: User = {
  id: "001",
  role: Role.ADMIN,
  name: "ade ojo",
  address: "lagos",
};

let resultAltSchool = checkUserRole(userAltSchool);
console.log(resultAltSchool);
```

```
admin
```

ni

Type Manipulation

Type Manipulation

- keyof

Type Manipulation

- keyof
- typeof

Type Manipulation

- keyof
- typeof
- indexed access types

Type Manipulation

- keyof
- typeof
- indexed access types
- conditional types

Type Manipulation

- keyof
- typeof
- indexed access types
- conditional types
- mapped types

Type Manipulation

- keyof
- typeof
- indexed access types
- conditional types
- mapped types
- template-literal-types