TYPESCRIPT: COLLECTED WORKS

https://github.io/przemyslawjanpietrzak

https://twitter.com/przemyslawjanp

Chapter I
Migration

@ts-check

package.json

```
"type-check": "tsc src/main.js --allowJs --out
/dev/null",
```

@ts-ignore

```
// @ts-ignore
[] + {}; // OK
```

Migrate

```
for f in src/**/*.js; do
  git mv "$f" "${f%.js}.ts"
done
```

Chapter II
Compiler options

Dead code elimination

```
"noUnusedParameters": true,
"noUnusedLocals": true,

const fn = (_unusedArg) => 42 // OK
```

strictFunctionTypes: true,

```
const fn = (arg: number, arg1: (string) => string)
=> {}

fn(42, str => str / 2); // ERROR
```

"noImpicitAny": true,

```
// ERROR
const fn = (arg) => arg;


// OK
const fn1 = (arg: any) => arg;
```

"strictNullChecks": true,

```
document.querySelector('#id').getAttribute('class');
 // ERROR;

const element = document.querySelector('#id');
if (element !== null) {
  element.getAttribute('id'); // OK
}

(document.querySelector('#id') as
HTMLElement).getAttribute('id'); // OK
```

"strictNullChecks": true, #beta

```
document.querySelector('#id')?.getAttribute('class');
 // OK;
```

"strictNullChecks": true, #beta

```
const element = document.querySelector('#id');
assert(element !== null)

element.getAttribute('id'); // OK
```

Honorable mentions

```
"noImplicitReturns": true,
"noImplicitThis": true,
"strictBindCallApply": true,
"paths": {
  "@core/*": [
    "app/*"
  ],
}
```

CHAPTER III
Cheap Tricks

Auto types

```
const fn = (): number =>
42;
const variable: string =
'';
public attr: boolean =
true;
[1, 2, 3].map((x: number)
=> x + 1);
const number$ = of<number>
(42);
```

```
const fn = () => 42;
const variable = '';
public attr = true;
[1, 2, 3].map(x => x + 1);
const number$ = of(42);
```

Property Accessing

```
interface Data {
  field: {
    name: string;
  }
}

export const fn = (arg: Data['field']) => {
  return arg.name; // { name: string }
}
export const fn1 = (name: Data['field']['name'])
=> {
  return name; // string
}
```

Big integer

```
const bigNumber = 123_456_789;
```

Abstract class

```
abstract class Page {
  constructor() {
    // ...logic
  }
}
new Page(); // ERROR: Cannot create an instance of
an abstract class.
```

Readonly

```
class Service extends AbstractService {
  public readonly url; = '...';
}

const service = new Service();
service.url = '???'; // ERROR: Cannot assign to
'url' because it is a read-only property.
```

Dict

```
const fn = (arg: { [key: string]: number }) => {
  const val =  arg.key1 + arg.key2 + arg.key3; // number
  const val1 = arg.totallyRandomKey; // number
  const val2 = arg['wpłynąłem na suchego przestwór oceanu']; // number
};

type Dict<T> = { [key: string]: T };
```

Tuple

```
let tuple: [string, number];
tuple = ["hello", 10]; // OK
tuple = [10, "hello"]; // Error
let str = tuple[0]; // string
let num = tuple[1]; // number
let len = tuple.length // 2
```

PART IV
Zbiory

Merged types

```
const fn = (arg: { key: string } & { key1: number
}) => 42;

fn({ key: '42' }); // ERROR
fn({ key1: 42 }); // ERROR
fn({ key: '42', key1: 42 }); // GOOD

type TableRow = Item & { selected?: boolean };
```

Union types

```
const fn = (arg: string | number) => {
    arg.split(''); // ERROR
    arg / 2; // ERROR
    arg + 1; // OK
    if (typeof arg === "string") {
        arg.split('');
    }
    if (typeof arg === "number") {
        arg / 2;
    }
}
```

Unknown types

```
function fn(arg: unknown) {
  if (typeof arg === "string" || typeof arg === "number") {
      arg;  // string | number
  }
  if (x instanceof Error) {
      arg;  // Error
  }
  if (isFunction(x)) {
      arg;  // Function
  }
}
```

Never types

```
function error(x): never {
  throw new Error("Unexpected object: " + x);
}

let variable = error(42); // never
```

Never types

```
function(arg: never) {
   ...
}
```

CHAPTER VI
Values as types

Based on argument

```ts
interface Data {
  fn(arg: -1): never
  fn(arg: 0): []
  fn(arg: number): Array<number>
}

let data: Data;
const a = data.fn(42); // Array<number>
const c = data.fn(0); // [];
const d = data.fn(-1); // never
```

Based on key

```
export interface API {
    "/users": { params: [], response: IUser[]}
    "/user/:id": { params: [number], response:
IUser}
}
```

Constant

```
const ROUTES = {
  LOGIN: '/login',
} // { LOGIN: "/login"; }

const ROUTES = {
  LOGIN: '/login',
} as const // { readonly LOGIN: "/login"; }
```

Constant

```
const someReduxAction = () => ({
  type: ActionTypes.Some,
}); // () => { type: type: ActionTypes }

const otherReduxAction = () => ({
  type: ActionTypes.Other,
} as const); // () => { type: ActionTypes.Other }
```

Resolve Union

```
const some reducer(state, action: SomeAction |
OtherAction) => {
  if (action.type === ActionTypes.Some) {
    action.payload; //SomeAction
  }
}
```

Manual Resolving

```
interface A { key: string; }
interface B { key: string, b: string }

let fn = (arg: A | B) => {
  if (arg.key[0] === 'b') arg.b // OK
}
```

Manual Resolving #2

```
const isB = (arg: A | B): arg is B => arg.key[0]
=== 'b';
```

Manual Resolving #2

```
let fn = (arg: A | B) => {
  if (isB(arg)) {
    arg.b // ERROR: Property 'b' does not exist on
type 'A | B'.
  } else {
    arg // A
  }
  arg // A | B
}
```

typeof

```
export const someAction = (payload: string) => ({
  type: '',
  payload
});

export type SomeActionConstructor = typeof
someAction;
```

CHAPTER VII
Weird parts

Generics extends

```
function loggingIdentity<T>(arg: T): T {
    arg.length;  // Error: T doesn't have .length
    return arg;
}
function loggingIdentity1<T extends Array<any>>
(arg: T): T[number] {
    arg.length; // OK
    return arg[0];
}

loggingIdentity1(42) // ERROR: Argument of type
'42' is not assignable to parameter of type
'any[]'.
loggingIdentity1([]) // OK
loggingIdentity1([42]) // number
```

Keyof

```
export type Omit<T, K extends keyof T> = {
  [P in Exclude<keyof T, K>]: T[P];
};

type R = Omit<{ a: string, b: string }, 'a'>; // { b: string }
type R1 = Omit<{ a: string, b: string }, 'b'>; // { a: string }
type R3 = Omit<{ a: string, b: string }, 'c'>}; // ERROR: Type '"c"' does not satisfy the constraint '"a" | "b"'.
```

Optional extends

```
type If<A extends boolean, T, U> = A extends true
? T : U;

let a: If<true, string, number>; // string
let b: If<false, string, number>; // number
```

Conditional extends

```
const first = (arr) => arr[0]

const first1 =
   <T extends any[]>(arg: T): T extends [] ? never
: T[number] => arr[0];

let never = first1([]); // never
let num = first1([42]); // number
```

Infer

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

type R = ReturnType<() => 42> // 42
```

Mapped types

```
type Optional<A extends object> = { [K in keyof
A]?: A[K] };
type Required<A extends object> = { [K in keyof
A]-?: A[K] };}

type OptionalItem = Optional<{ key: string, key1:
number}>; // { key?: string, key1?: string }

put(data: Data) {}
patch(data: Optional<Data>)
```

Examples # json api

```
interface User {
  id: string;
  name: string;
  age: number;
  courses: Array<Course>
}

interface UserResponse {
  id: string;
  attributes: {
    name: string;
    age: number
  }
  relationships { courses: { data: []}}
}
```

Examples

```
interface Response<T extends { id: string }, R =
void> {
  id: string;
  attributes: Omit<T, 'id'>;
  relationships: R extends void ? void : { [key:
string]: { data: Array<R> } };
}

type UserResponse = Response<User, Course>;
```

Class Type

```
class View {};

interface Model {
    View: { new(...args: any[]): View };
}

class Shape implements Model {
  View = class extends View {
    constructor(){ console.log("view created"); }
  }
}
```

Grande finale

```ts
type EmptyTuple = [];

type TupleLength<T extends Array<any>> =
T["length"];

type PrependTuple<A, T> = T extends Array<any>
  ? (((a: A, ...b: T) => void) extends (...a:
infer I) => void ? I : [])
  : [];

type NumberToTuple<N extends number, L extends
Array<any> = EmptyTuple> = {
  true: L;
  false: NumberToTuple<N, PrependTuple<1, L>>;

}[TupleLength<L> extends N ? "true" : "false"];
```

Thank you :*