TYPESCRIPT: COLLECTED WORKS

https://github.io/przemyslawjanpietrzak

Part I
Migration

Demo (migracja)

package.json

```
"type-check": "tsc bundle.js --allowJs --noEmit",
"build": "npm run lint && npm test && npm run type-check && npm run bundle"
```

Migrate

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

Part 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,

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

// GOOD
const fn1 = (arg: any) => arg;

// GOOD
const fn2 = (arg: number) => arg;

// ALSO GOOD
[1,2,3].map(item => item + 1);
```

"strictNullChecks": true,

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

(document.querySelector('#id') as HTMLElement).getAttribute('class')

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

PART III
Tricks

Auto types

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

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

Property Accessing

```typescript
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;
```

readonly & abstract

```
abstract class AbstractService {
  public method() {}
}

class Service extends AbstractService {
  public readonly field = [42];
}

const service = new Service();
service.field.push(42); // OK
service.field = [43]; // ERROR

const abstractService = new AbstractService(); // ERROR
```

Tuple and 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
};

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

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(''); // Property 'split' does not exist on type 'string | number'.
    arg / 2; // The left-hand side of an arithmetic operation must be of type 'any', 'number', 'bigint' or an enum type.
    arg + 1; // OK
    if (typeof arg === "string") {
        return arg.split('');
    }
    if (typeof arg === "number") {
        return arg / 2;
    }
}
```

Union types

```typescript
interface Dog {
    kind: "dog"
    dogProp: any;
}
interface Cat {
    kind: "cat"
    catProp: any;
}

const catOrDogArray: Dog[] | Cat[] = [];
catOrDogArray.forEach((animal: Dog | Cat) => {
    if (animal.kind === "dog") {
        animal.dogProp;
    }
    else if (animal.kind === "cat") {
        animal.catProp;
    }
});
```

Unknown types

```
function f20(x: unknown) {
  if (typeof x === "string" || typeof x === "number") {
      x;  // string | number
  }
  if (x instanceof Error) {
      x;  // Error
  }
  if (isFunction(x)) {
      x;  // Function
  }
}
```

Never types

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

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

PART V
Values as types

Based on argument

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

let data: Data;
const a = data.fn(42); // null
const b = data.fn("str"); // Array<string>
```

Based on key

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

PART VI
Weird parts

Optional extends

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

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

Infer

```
type ReturnType<T> = T extends (...args: any[]) => infer R ? R : any;
```

Maped types

```
export type DeepReadonlyObject<A> = { readonly [K in keyof A]: DeepReadonly<A[K]> };
type DeepReadonlyObject<A> = { readonly [K in keyof A]: DeepReadonly<A[K]> }

type X = DeepReadonlyObject<{ key: string, key1: number }>; // { readonly key: any; readonly key1: any; }
```

Optional mapped types

```
export type Omit<A extends object, K extends string | number | symbol> = Pick<A, Exclude<keyof A, K>>

type X = Omit<{ key: string, key1: string }, "key"> //  { key1: string; }

type Diff<A extends object, K extends keyof A> = Omit<A, K> & Partial<Pick<A, K>>
```

Grande finale

```
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"];

type Increment<N extends number> = TupleLength<PrependTuple<1, NumberToTuple<N>>>;

type T = Increment<42>
```

Thank you :*