phase 4：TypeScript

いなにわうどん* （@kyoto_inaniwa）

IPC / jsys 合同 Web 研修 2024
2024 年 6 月 15 日（土）

本フェーズでは，**TypeScript** による静的型付け言語を用いたプログラミングを学びます．以前のフェーズと比較して難易度が上がり[1]，初学者の方々は一度に理解することが難しいかもしれませんが，ぜひみなさんで仲良く相談しながら，協力して取り組んでみてください．なお，本資料においては，JavaScript は既習であることを前提として話を進めるため，不明な点があれば，適宜 1 日目のフェーズ 3 の資料や外部の Web サイト，書籍等を参照してください．

1　TypeScript

TypeScript（略称，TS）は、Microsoft により 2012 年に発表された，JavaScript のスーパーセットとなるプログラミング言語です．TypeScript では，JavaScript に対して静的型付けの概念や新たな構文を導入することにより，堅牢性や利便性を保証しています．TypeScript は JavaScript にトランスパイル（変換）されて利用されます．したがって我々は TypeScript を用いて，JavaScript と同様に，ブラウザにて作動するアプリケーションを構築したり，Node.js[2] や Deno[3] 等の実行環境にて作動するサーバを開発したりすることができます．

2　TypeScript を動かしてみる

細かい説明は後に回して，ひとまず使ってみましょう．TypeScript は Node.js 上で作動する `tsc` と呼ばれるコマンドを用いてトランスパイルされます．ここで，トランスパイルとはあるプログラミング言語を別の言語へと変換することを指します．

*`https://github.com/inaniwaudon`
[1]筑波大学においては 2 年次以降で扱う内容に相当すると思われます
[2]`https://nodejs.org/`
[3]`https://deno.com/`

2.1 環境構築

Node.js をインストールしていない方は，まずそのインストールから始めてください．Node.js のインストールにあたっては，後から複数の Node.js のバージョンを共存させることを想定して，バージョン管理ツールを導入することを強く推奨します．nvm[4]（Windows 環境の場合は nvm-windows[5]），fnm[6] に代表されるバージョン管理ツールは，主要な OS（macOS, Windows, Linux）に対応しているようです．本資料では Node.js およびバージョン管理ツールのインストールの詳細には触れないため，インストールは各自の責任にて行ってください．

Node.js のインストールが完了したら，`typescript` および `tsc` パッケージをグローバルにインストールします．

```
$ npm install -g typescript tsc
```

2.2 実行してみる

適当な場所に `test.ts` を作成し，図 1 のコードを記述します．

図 1: test.ts

```
const sample: string = "Hello world!";
console.log(sample);
```

これを，以下のコマンドを用いて実行します．

```
$ tsc test.ts
$ node test.js
Hello world!
```

これが基本的な TypeScript の実行の流れです．よく挙動を観察すると，`tsc` コマンドの実行後に，`test.js` が作成されています．`test.ts`, `test.js` を比較すると，`test.js` では謎の注釈である `:string` が消失しています．これは，TypeScript の独自構文です．以上の挙動から，トランスパイルの段階で TypeScipt の独自構文は削除され，純粋な JavaScript のコードへと変換されていることが読み取れます．

[4] `https://github.com/nvm-sh/nvm`
[5] `https://github.com/coreybutler/nvm-windows`
[6] `https://github.com/Schniz/fnm`

3 TypeScipt のうれしさ

TypeScript を利用するには, 独自の構文を追加した上でトランスパイルという工程を挟む必要があるため, 一見すると冗長な手続きのようにも思えます．しかしながら, TypeScript を用いると，堅牢で保守がしやすいコードを記述することができます.

3.1 静的型付け

その理由の一つに，**静的型付け**と呼ばれる仕組みが挙げられます.

プログラムが扱うデータには**型**があります．型はデータがどのように解釈されるかを規定するものです．例えば JavaScript において，`123` は数値型として，`"abc"` は文字列型として扱われます.

TypeScript の基となった言語である JavaScript は動的型付け言語です．動的型付け言語は，プログラムの実行時に型検査を行うため，型エラー（文字列型を数値型として扱ってしまった等）を実行時まで検出することができません．また，JavaScript の型システムは適当であるため，暗黙的な型変換によって型が “よしなに”変換され，思わぬバグを生むケースも存在します.

これに対し，TypeScript は静的型付け言語に位置づけられ，プログラムの前処理であるトランスパイルの段階にて型検査を行います．ゆえに，記述したプログラムに不備があるかを，実行時ではなく，設計時に検査することが可能となります[7]．また，TypeScript では明示的に型宣言を行いながら，変数や引数を定義するため，設計者の意図せぬうちに型が変化する現象を抑えることに繋がります.

―と文章で説明したところで，初学者には解りにくい概念だと思いますので，実際のコードを 図 2 に示します．図 2 では，宣言時は文字列型だった変数 `aaa` に，次の行では数値が代入されています．最後の行では，`aaa` が文字列であると想定して `replace` 関数を呼び出しています.

図 2: TypeScript の例

```
let aaa = "42";
aaa = 42;
aaa.replace("4", "");
```

これを通常の Node.js で実行すると，ランタイムエラー（実行時エラー）となります．`aaa` は最終的に数値型になっているので，当然のことです.

[7]もちろん，トランスパイル時に検出できないエラーも多数存在します.

```
$ node test.js
aaa.replace("4", "");
    ^
TypeError: aaa.replace is not a function
```

一方，TypeScipt では，TypeScript が持つ型情報によって，このエラーをトランスパイル時に検出できます．こうして，実行時ではなく，トランスパイル時にエラーを未然に防ぐことができました．

```
$ tsc test.ts
test.ts:2:1 - error TS2322: Type 'number' is not assignable to type 'string'.
2 aaa = 42;
  ~~~
Found 1 error.
```

静的／動的型付けには様々な論争がありますが，Web 開発において，TypeScript は概ねデファクトスタンダードとなっています．

4 TypeScript 爆速入門

本章では，TypeScipt 特有の構文と，フェーズ 3 ではカバーできなかった JavaScript の構文の一部を紹介します．

4.1 型

4.1.1 型注釈

TypeScript では，変数および引数の宣言時に，**型注釈**を付けて，その変数および引数が取りうる型を制限することが求められます．型注釈は ： 型名 で付与します．また，TypeScript には**型推論**の機能があるため，初期値等からその変数の型が決定される場合，型注釈は不要です．型注釈および型推論の例を，図 3 に示します．

図 3: 型注釈および型推論の例

```
let a: number = 42;       // 自明であるため型推論される
const b: number = "42"; // コンパイルエラー
a = "foo";                // コンパイルエラー

// 引数 a, b の型は明らかでないため, 型注釈が必須
const sum = (a: number, b: number) => a + b;
```

4.1.2 型の種類

TypeScript が扱う代表的な型の種類を，図 4 に示します．示したもののうち，string, number, bool, null, undefined はプリミティブ型[8]と呼ばれます．

図 4: 代表的な型の種類

```
const a: string            = "foo";           // 文字列型
const b: number            = 42;              // 数値型
const c: bool              = true;            // 論理型(ブーリアン型)
const d: null              = null;            // null
const e: undefined         = undefined;       // undefined
const f: 42                = 42;              // リテラル型
const g: string[]          = ["foo", "bar"]; // 文字列型の配列
const h: [string, number] = ["foo", 42];    // 文字列型と数値型のタプル
```

注意点

- null と undefined は区別されます．この使い分けは難しいのですが，null は自然発生せず，明示的に値が存在しないことを表す型であると憶えておくと良さそうです．
- リテラル型は，特定の値のみを取る型です．`let f: 42 = 43` という指定はできません（コンパイルエラーになります）．リテラル型として，true，false，数値，文字列が指定できます．
- タプルは配列と異なり，要素の長さ，および要素毎の型が指定されます．

4.1.3 型エイリアス

型エイリアスと呼ばれる仕組みを用いて，型に名前を付け，再利用することもできます．型名は多くの場合，アッパーキャメルケース（パスカルケース）にて記述します．プリミ

[8]示したもののほかに symbol, bigint が存在します．

図 6: オブジェクトおよびインタフェースの例

```
type Student = {
  studentNo: number;
  name: [string, string];
  affiliation: string;
};
// 以下でも同じ意味になる
interface IStudent {
  studentNo: number;
  name: [string, string];
  affiliation: string;
}

const student: Student = {
  studentNo: 123,
  name: ["情報", "太郎"],
  affiliation: "情報科学類",
};
const student: IStudent = {
  studentNo: 456,
  name: ["情報", "花子"],
  affiliation: "情報メディア創成学類",
};
```

ティブ型単体に型エイリアスを付けることは稀ですが，ユニオン型（後述）等と併用される機会は多いです．型エイリアスの例を，図 5 に示します．

図 5: 型エイリアスの例

```
type ValueType: number | string;
let a: ValueType = 42;
a = "foo";
```

4.1.4 オブジェクト，インタフェース

TypeScript では，構造化されたオブジェクトを記述することができます．また，類似の機能は `interface` キーワードを用いても実現できます．この例を 図 6 に示します．

4.1.5　ユニオン型

ユニオン型は，「2 つ以上の型のうち，いずれかに当てはまる型」を指します．|（バーティカルバー）を用いて，複数の型を繋げて書きます．ユニオン型の用例を 図 7 に示します．

図 7: ユニオン型の例

```
// string または null が入る
const learnedLanguage: string | null;

// 年月日を表す数値のタプル, または文字列が入る配列
type CustomDate = ([number, number, number] | string);
const dates: CustomDate[] = [[2024, 3, 21], "2024/3/21"];

// エラーコードが 3 つしか存在しない世界線
type statusCode = 200 | 404 | 500;
```

また，**タグ付きユニオン**と呼ばれる機能を用いて，型の絞り込みを行うことができます．図 8 では，酒 Liquor とエナジードリンク EnergyDrink を表す型を定義し，これらのユニオン型として Drink を定義します．健康的であるかを判断する isHealthy では，Drink 型の引数を受け取ります．引数の type が "liquor" であれば，酒であると判断しアルコール含有量を，そうでなければエナジードリンクと判断し，カフェイン含有量の検査を行います．

図 8: タグ付きユニオンの例

```
type Liquor = {
  type: "liquor";
  name: string;
  alcohol: number;
}
type EnergyDrink = {
  type: "energydrink";
  name: string;
  caffeine: number;
}
type Drink = Liquor | EnergyDrink;

const isHealthy = (drink: Drink) =>
  drink.type === "liquor" ? drink.alcohol < 20 : drink.caffeine < 40;
```

4.2 交差型

交差型を用いると，複数の型を結合することができます．図 9 に例を示します．Staff は Person かつ Employee であるため，2 つの型を結合した交差型として表現しています．

図 9: タグ付きユニオンの例

```
type Person = {
  name: string;
  age: number;
};
type Employee = {
  employeeId: number;
  department: string;
};
type Staff = Person & Employee;
```

4.3 ジェネリクス

ジェネリクスは，クラス，関数およびインタフェースを抽象化するための概念です．ジェネリクスを用いることで，異なる型を持つデータに対して，同一の処理を実行することなどが可能となります．説明が難しいので，具体的な例を 図 10 に示します．

この例にて，extract 関数は型引数 T を取ります．そして引数には，T[] 型の array，および T 型の target を取ります．処理の内容としては，array から target に一致した要素をすべて抽出しています．注目すべきは extract 関数の呼び出し部分です．1 回目の呼び出しでは，数値型の配列およびターゲットに対して関数を呼び出し，2 回目の呼び出しでは，文字列型の配列およびターゲット対して関数を呼び出しています．このように，引数をジェネリクスによって抽象化することにより，number[] 用，string[] 用と分けることなく，処理を統一することができます．

図 10: ジェネリクスの例

```
const extract = <T>(array: T[], target: T) => {
  const result: T[] = [];
  for (const item of array) {
    if (item === target) {
      result.push(item);
    }
  }
  return result;
};

extract([1, 2, 2, 3], 2);
// [2, 2]
extract(["あ", "か", "さ", "た", "な"], "な");
// ["な"]
```

4.4 Promise

`setTimeout`（n ミリ秒後に処理を実行する），`fetch`（HTTP リクエストを送信する）等の関数は非同期で実行されるため，開発者の意図しない順序で呼び出されることがあります．これを解決する仕組みが，Promise です．Promise を利用するには複数の記述方法が存在しますが，今日では `async`，`await` キーワードが主に利用されています．

Promise を使用しない例と，使用した例をそれぞれ 図 11，図 12 に示します．Promise を使用しない場合，fetch 関数の戻り値をコンソールに表示すると，Promise が pending（待機中）である旨が返されます．一方，request の戻り値に `await` を付与して Promise を使用した場合は，正しくレスポンスが取得できています．`await` を付与する際，その文は `async` を付与した関数内で実行される必要があります．トップレベルに記述をする場合は，即時関数に `async` を付与することで解決します．

図 11: Promise を使用しない例

```
const request = fetch("https://example.com");
console.log(request);
// Promise { <pending> }
```

図 12: Promise を使用した例

```
(async () => {
  const request = await fetch("https://example.com");
  console.log(await request.text());
  // <!doctype html>
  // <html>
  // <head>
  // ...
})();
```

5 むすびに

TypeScript の学習に推奨する Web サイトを掲示します．本資料は不十分な点も多いので，必要に応じて以下の Web サイトを参照してください．

- サバイバル TypeScript．`https://typescriptbook.jp/`.
- TypeScript Deep Dive 日本語版．`https://typescript-jp.gitbook.io/deep-dive`.
- TypeScript Documentation．`https://www.typescriptlang.org/docs/`

6 演習問題

最後に演習問題を用意しました．やや発展的な内容ですが，ぜひチャレンジしてみてください．

Q1. 筑波大学の構成員

筑波大学の構成員を，TypeScript の型として表現してください．具体的には 図 13 に書きかけのソースコードを示すので，コード中の … で示された部分を埋めてください．タグ付きユニオンを使用できる場合は使用してください．また，リテラルを用いて取りうる型を制限できる場合は，できるだけ制限してください．

- 筑波大学の構成員（Member）は人（Person）であり，かつ，学生（Student）または教員（Teacher）である．
- 学生は学籍番号（studentNo）と所属（affiliation）を持つ．学籍番号は文字列にて表す．所属は，学類生（undergraduate）の場合は学類（School）が，大学院生の場合は研究群（DegreeProgram）が該当する．

- 教員は職階（position）と所属（affiliation）を持つ．職階は教授，准教授，講師，助教の 4 つである．所属は，系（Institute）が該当する．
- すべての人は名前（name），生年月日（birth），居住地（residence）を持つ．名前は姓，名をタプルにて表す．生年月日は年，月，日をタプルにて表す．居住地は文字列にて表す．

図 13: 書きかけのソースコード

```
type School = "情報科学類" | "情報メディア創成学類" | "知識情報・図書館学類";
type DegreeProgram = "システム情報工学研究群" | "人間総合科学研究群";
type Institute = "システム情報系" | "図書館情報メディア系";

type Person = {
  ...
};
type Student = {
  type: "student";
  ...
};
type Teacher = {
  type: "teacher";
  ...
};

type Member = ...;
```

Q2. アプリケーション開発

1 日目の「フェーズ 3：JavaScript」の「9.4.4 演習」に示した，Twitter もどきのアプリケーションを TypeScript を用いて開発してください．React 等のフレームワークは，現段階ではまだ使用しないでください．開発に主に必要な観点を以下に示します．

- ツイートのデータ構造はどう持てば適切か？
- DOM 操作は TypeScript ではどのように行うべきか？
- `tsconfig.json` ファイルはどのように書けば良いか？