udemy TypeScript はむさん

セクション 1

4 typescript とは(2020 年 5 月 17 日)

- TypeScript PlayGround MS が出しとるやつ遊び場

7

```
 git checkout -b create-package-json
```

チェックアウトとリポジトリ作成を両方やる

8 TypeScript をインストールする

```
npm i typescript@3.7.5 --save-dev
```

Powershell でファイルを新規作成

```
  New-Item src/install-typescript.ts
```

- ts をコンパイル tsc はグローバルにインストールされていないので`node_modules`から指定する

```
./node_modules/.bin/tsc src/install-typescript.ts
```

- gitbash でコードを見る

```
cat install-typescript.ts


```

- 生成された js ファイルを削除するまでの流れ 全部加える

```
 git add .
```

status 見る

```
git status
```

git の追跡から外したくない場合は`git reset`する

```
makito.mori@PC790 MINGW64 ~/Desktop/self_study/udemy/udemy-typescript (install-ts)
$ git reset src/install-typescript.ts

makito.mori@PC790 MINGW64 ~/Desktop/self_study/udemy/udemy-typescript (install-ts)
$ git status
On branch install-ts
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        modified:   memo.md
        modified:   package.json
        new file:   src/install-typescript.js

Untracked files:
  (use "git add <file>..." to include in what will be committed)
        src/install-typescript.ts
```

ファイル削除(参考：https://qiita.com/k0uh0t/items/ae885bf2d5e05614b80f)

```
$ git clean -f
Removing src/install-typescript.ts
```

- master にマージする

```
git checkout -
```

ハイフンは前回のブランチ

```
git merge -
```

9 ts-node の導入

コンパイルと実行を同時に行う。

- インストール

```
npm i ts-node
```

- 実行

```
npx ts-node src/install-typescript.ts
```

10 ts-node-dev の導入

ファイルが修正されるたびにコンパイルと実行を行う。

- インストール

```
npm i ts-node-dev --save-dev
```

- 実行(github:https://github.com/whitecolor/ts-node-dev)

```
npx ts-node-dev --respawn src/install-typescript.ts
```

11 vs-code インストール

settings.json を変更する

```
  "prettier.semi": true,
  "prettier.singleQuote": false,
```

- ts-config.json を作成する

```
 npx tsc --init
```

セクション 2 基本的な型

12 boolean

src 下にファイルを作成する これで作ると変な文字が入ってコンパイルされない。

```
echo "export {};" > src/boolean.ts
```

代わりにこっちを使う

```
New-Item .\src\test.ts -Value "export{}"
```

- boolean.ts

```
export {};
let name = "TypeScript";
```

export がないと name は警告が出る。変数 name は ts 側ですでに宣言されているので使えない。 export してモジュール化してあげることで警告を回避できる。 これからの講義で export{}をしばしば書く

15 Array 型

書き方 ただし 2 つ目は非推奨

```
let numbers: number[] = [1, 2, 3];
let numbers2: Array<number> = [1, 2, 3];
```

- array.ts
  - 1 次元 配列

```
let strings: string[] = ["TS", "JS", "CS"];
```

  - 2 次元配列

```
let nijigenHairetsu: number[][] = [
[50, 100],
[150, 300],
];
```

- 型が混合した配列(union type 共用型) `|`は「または」の意味なので`1`,`false`,`Japan`の順番は任意

```
let hairetsu: (string | number | boolean)[] = [1, false, "Japan"];
```

16 tuple 型

共用型と違って順番も制限を入れる

```
let profile: [string, number] = ["Ham", 43];
profile = [43, "ham"]; //エラーが出る
```

17 any 型

axios の github:https://github.com/axios/axios

自分の好きな型を作れる。（研修で interface やったけど忘れた）

```
interface Article {
    id: number;
    title: string;
    description: string;
  }
let data: Article[]; //Article型を定義する
```

18 void 型

関数の戻り値に対するアノテーション 関数の横に:~~でアノテーションできる

- void.ts

```
function returnNothing(): void {
  console.log("I don't return anythings");
}
```

19 null undefined 型

```
let absence: null = null;
// absence = "123"; //エラー出る
let data: undefined = undefined;
// data = 123;//エラー出る
```

20 never 型

never:return されない void :空が return される

- never.ts

```
function error(message: string): never {
  throw new Error(message);
}
```

```
try {
  let result = error("test");
  console.log({ result });//errorという関数を実行した時点でエラーが投げられるのでconsole.logの
処理は実行されない
} catch (error) {
  console.log({ error });
}
```

void は`undefined`と`null`は入れられるけど never 型は無理。

```
let foo: void = undefined; //エラーでない
let foo2: void = null; //エラーでない
let bar: never = undefined; //エラー出る
let bar2: never = error("only me"); //never型のみ代入できる。使い道はない。
```

21 object 型

アノテーションで`:object`と書ける。ただし`:object`は幅が広くてキーが上書き出来て、バリューも違う型のもので上書きできる。

```
let profile: object = { name: "Ham" };
profile = { birthYear: 1976 }; //上書きできる
```

もっとアノテーションを限定的にする

```
let profile2: {
  name: string;
} = { name: "makito" };
// profile2 = { birthYear: 1976 }; //上書きできない
profile2 = { name: "mori" };//上書きできる
```

22 型エイリアス

エイリアス：別の名前を付ける

- type-aliases.ts 基本形 必ず最初は大文字にする string 型を別の呼び方にする

```
type Mojiretsu = string; //先頭は大文字
const fooString: string = "hello";
const fooMojiretsu: Mojiretsu = "hello";
```

`:Mojiretsu`が使えるようになる

- 応用的な使い方その 1 Profile という型を作る

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

const example2: Profile = {
  name: "Makito",
  age: 26,
};
```

- 応用的な使い方その 2 型を指定していない変数の型を取る type ＜任意の型の名前（string とか number 以外）＞ typeof ＜オブジェクト＞

```
//型を指定していないオブジェクト
const example1 = {
  name: "Makito",
  age: 26,
};

//Profile2という型を宣言
type Profile2 = typeof example1;

//作った型を利用
const example3:Profile2 ={
  name:"Mori",
  age:26
}
```

23 interface

オブジェクトに対してアノテーションする別の方法

- interfaces.ts エイリアスと違って=はいらない

```
//エイリアスを使ったやり方
type ObjectType = {
  name: string;
  age: number;
};

//インターフェースを使ったやり方(イコールがいらない)
interface ObjectInterface {
  name: string;
  age: number;
}
```

```
//エイリアスと同様に`:~~~`とする。
let object: ObjectInterface = {
  name: "Makito",
  age: 26,
};
```

24 型安全とは

いろいろあった。

25 unknown 型

端的には型安全な any 型。any 型を使うくらいなら unknown 型の方がエラーが出るからこっちを使うべき（？）

タイプガード、型ガード

```
//any型とunknown型に値を代入
const kansu = (): number => 43;
let numberAny: any = kansu();
let numberUnknown: unknown = kansu();

let sumAny = numberAny + 10;//エラーでない
let sumUnknown = numberUnknown + 10; //コンパイルエラーが出る

//typeof で型を確認するとエラーが出ない
if (typeof numberUnknown === "number") {
  let sumUnknown = numberUnknown + 10;
  console.log(123);
}
```

26 交差型(intersection 型)

複数の型を合成する。 ピッチャーの特長を持つ型:Pitcher1 バッターの特長を持つ型:Batter1

これらを合成した:TwoWayPlayer を作る

- intersection.ts

```
//ピッチャー
type Pitcher1 = {
  throwingSpeed: number;
};

//バッター
type Batter1 = {
  battingAverage: number;
```

```
};

//ピッチャー生成
const DaimajinSasaki: Pitcher1 = {
  throwingSpeed: 154,
};

//バッター生成
const OchiaiHiromitsu: Batter1 = {
  battingAverage: 0.367,
};
```

- これらの型を合成する &で型を結ぶ

```
type TwoWayPlayer = Pitcher1 & Batter1;

const OtaniShouhei: TwoWayPlayer = {
  throwingSpeed: 165,
  battingAverage: 0.286,
};
```

27 union 型(共用体型)

複数の型（string と number）を使いたい時に利用する型

パイプを使って複数の型を書く めっちゃ使うらしい

```
let value: number | string = 1;
value = "foo";
value = 1;
```

28 Literal 型

primitive 型より細かく設定できる

- String 型の Literal
    - 「日、月、火、水、木、金、土」以外の値は入れたくないときなどに使う

```
let dayOfTheWeek: "日" | "月" | "火" | "水" | "木" | "金" | "土" = "日";
dayOfTheWeek = "月";
// dayOfTheWeek = "31"; // エラーになる
```

- number 型の Literal

```
let month: 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 = 1;
month = 12;
// month = 13; // エラーになる
```

- boolean 型の Literal
    - あんまり使いどころはない。

```
let TRUE: true = true;
// TRUE = false; // エラーになる
```

29 enum 型（列挙型）

```
enum Months {
  January = 1,
  February,
  March,
  April,
  May,
  Jun,
  July,
  August,
  September,
  October,
  November,
  December,
}

console.log(Months.January); // 0
console.log(Months.February); // 1
console.log(Months.December); // 11
```

このままだと 0 月とかになるから enum を初期化する

```
enum Months {
  January = 1,
  February,
  ...
  December
}
console.log(Months.January); // 1
console.log(Months.February); // 2
console.log(Months.December); // 12
```

- String 型の enum

```
enum COLORS {
  RED = "#ff0000",
  WHITE = "#ffffff",
  GREEN = "#008000",
  BLUE = "#0000FF",
  BLACK = "#000000",
}

//アクセス方法
let green = COLORS.GREEN; // 存在しない要素を指定してもjsだとエラーが出ない
console.log({ green });

//追加方法
enum COLORS {
  YELLOW = "#FFFF00",
  GRAY = "#808080",
}
```

セクション 3 関数で型を使う

30 function により関数定義（5 月 18 日）

返り値に型を付ける `function ＜関数名＞: ＜numberとかstring＞` 具体的には

- function.ts

```
function bmi(height: number, weight: number): number {
  return weight / height ** 2;
}
console.log(bmi(1.8, 65));
```

31 無名関数

- 無名関数のアノテーション =の前後で分けて考える
  - =の前： （引数のアノテーション）=>(返り値のアノテーション)
  - =の後：いつも通り

- anonymous-function.ts

```
let bmi: (height: number, weight: number) => number = function (
  height: number,
  weight: number
): number {
  return weight / height ** 2;
};
```

32 アロー関数のアノテーション

アロー関数は`return`を省略できる

```
let bmi: (weight: number, height: number) => number = (
  weight: number,
  height: number
): number => weight / height ** 2;

console.log(bmi(65, 1.8));
```

33 オプショナルなパラメータ（引数）を定義する

書いても書かなくてもよい引数の書き方。 `bmi(1.8,67,true)`でも`bmi(1.8,67)`でも実行できるようにする。つまり第 3 引数を入れるか入れないかは任意とすることが目的 結論は`＜変数名＞?:＜型＞`のように?を付けるだけで OK 講義では関数のアノテーションにも`?`を付けているがコードを動かすだけなら引数だけに`?`を付けるだけでよい。

- optional-arguments.ts 1 行目の`?`は無くても動く

```
let bmi: (height: number, weight: number, printable?: boolean) => number = (
  height: number,
  weight: number,
  printable?: boolean
): number => {
  const bmi: number = weight / height ** 2;
  if (printable) {
    console.log({ bmi });
  }
  return bmi;
};

bmi(1.9, 65, true);//動く
bmi(1.9,65)//動く
bmi(1.9)//エラー
```

34 デフォルトパラメータの設定

関数を実行する時に引数を指定しなかった場合に関数側で引数を（デフォルトで）指定するもの。TypeScript 特有の機能ではなく JavaScript にもともとある機能

関数の引数に`rate: number = 1.1`のようにしておくと引数で値が指定されなかったら値が rate の値が 1.1 になる

- default-parameters.ts

```
const nextYearSalary = (currentSalary: number, rate: number = 1.1) => {
  return currentSalary * rate;
};
```

```
console.log(nextYearSalary(1000, 1.05)); //1050
console.log(nextYearSalary(1000)); //1100
```

35 Rest パラメータの設定

JS にもともとあったパラメータの一つ

引数が何個あるか不明な時に使う。 `...value`スプレッド演算子を使っている。

- rest-parameters.ts

```
const reducer = (accumulator: number, currentValue: number) => {
  return accumulator + currentValue;
};

const sum: (...values: number[]) => number = (...values: number[]): number => {
  return values.reduce(reducer);
};

console.log(sum(1, 2, 3, 4, 5));
```

- 関数 reduce の使い方
  - コード
    1. reduce の引数に関数を取る
    2. return で返した値が accumulator に入る
    3. 繰り返し

`reduce(＜関数＞,＜accumulatorの初期値＞)`具体的には`reduce(sumCalc,10)`

```
const numbers=[1,2,4,8,16]
const sumCalc = (accumulator, currentValue, currentIndex, array)=> {
  console.log({accumulator, currentValue, currentIndex, array})
  return accumulator + currentValue
}

console.log(`[${numbers}]の合計値は${numbers.reduce(sumCalc,10)}`)
```

- 出力

```
{
  accumulator: 1,
  currentValue: 2,
  currentIndex: 1,
  array: [ 1, 2, 4, 8, 16 ]
}
{
```

```
  accumulator: 3,
  currentValue: 4,
  currentIndex: 2,
  array: [ 1, 2, 4, 8, 16 ]
}
{
  accumulator: 7,
  currentValue: 8,
  currentIndex: 3,
  array: [ 1, 2, 4, 8, 16 ]
}
{
  accumulator: 15,
  currentValue: 16,
  currentIndex: 4,
  array: [ 1, 2, 4, 8, 16 ]
}
'[1,2,4,8,16]の合計値は31'
```

36 オーバーロード

同じ名前の関数で引数、返り値の型が違う関数を作りたい。

1. シグネチャーを宣言する
2. 引数も返り値も any 型の関数を作る（any にしてもよいのはシグネチャー側で型を制限しているため）

- overloads.ts

```
//関数の宣言（シグネチャー）
function double(value: number): number;
function double(value: string): string;

//any型で関数を作る
function double(value: any): any {
  // console.log(typeof value);
  if (typeof value === "number") {
    return value * 2;
  } else {
    return value + value;
  }
}

console.log(double(100));//シグネチャーの中に含まれているので実行できる
console.log(double("Go "));//シグネチャーの中に含まれているので実行できる
// console.log(double(true)); //シグネチャーの中にbooleanがないとエラーになる
```

セクション 4 クラスで型を使う

37 クラスを作ってみる(2020 年 5 月 19 日)

- class を作るときは最初の文字は大文字にする(person ではなく Person)
- constructor は必ず呼び出される
- my-first-class.ts

```
class Person {
  name: string;
  age: number;

  //コンストラクターの戻り値は書かない（returnしないから）
  //constructor():voidみたいなのはいらない
  constructor(name: string, age: number) {
    this.name = name;
    this.age = age;
  }

  //関数
  profile(): string {
    return `name:${this.name}, age:${this.age}`;
  }
}
```

38 アクセス修飾子

- アクセス修飾子は TypeScript 独自の仕様である。

コピーコマンド

```
cp src/my-first-class.ts src/access-modifiers.ts
```

- メンバ変数にアクセス修飾子を付ける
    - public : どこからでもアクセスできる

```
class Person {
//メンバ変数
public name: string;
protected age: number;
protected nationality: string;
...
}
let taro = new Person("makito",26,"Janan")
console.log(taro.name);//エラーでない
console.log(taro.age);//エラーでる
console.log(taro.nationality);//エラーでる
```

    - protected : 継承した子クラスでもアクセスできる
    - private : そのクラス内からでしかアクセスできない
        - method を作ってそこからならアクセスできる

- access-modifiers.ts

```
export {};

class Person {
  //メンバ変数
  public name: string;
  // private age: number;
  protected age: number;
  protected nationality: string;

  constructor(name: string, age: number, nationality: string) {
    this.name = name;
    this.age = age;
    this.nationality = nationality;
  }

  profile(): string {
    return `name:${this.name}, age:${this.age}`;
  }
}

class Android extends Person {
  constructor(name: string, age: number, nationality: string) {
    super(name, age, nationality);
  }
  protected(): string {
    return `name:${this.name}, age:${this.age}, nationality:${this.nationality}`;
  }
}

let taro = new Person("Makito", 26, "Japan");

console.log(taro.profile());
console.log(taro.name);
```

39 コンストラクター—をもっと使う

- コンストラクター—の中にアクセス修飾子を入れると自動で初期化ができる。
- メンバや`this.name=name`みたいな物も書く必要がなくなる
- more-constructor.ts

```
class Person {
  //アクセス修飾子を付けることで初期化までできる。
  constructor(public name: string, protected age: number) {}
}

const me = new Person("Makito", 26);
console.log(me);
```

40 getter と setter

研修で Java やっとってよかった。

- メンバ変数は_で始める
  - private _name:string
- getter,setter では_なしで書く
  - `get name(){}`, `set name(name:string){}`(詳しくは下で)

```
private _owner: string;
get owner() {
  return this._owner;
}
```

- getter-and-setter.ts getter と setter の書き方

```
get owner() {
  return this._owner;
}

set secretNumber(secretNumber: number) {
  this._secretNumber = secretNumber;
}
```

使い方 setter は secretNumber()みたいにはならない。 変数_secretNumber に関しては getter を作っていないのでアクセスできない

```
card.secretNumber = 1111111111; //セッター経由でアクセスできる
// card._secretNumber = 1111111111; //privateなのでアクセスできない
console.log(card.secretNumber); //undefined
console.log(card.owner); //表示できる
```

セクション 5