Typescript

Typescriptとは、以前Javascriptで静的型付け、動的型付けなどのお話をしました。
ここで「変数を宣言するときに型を決める必要がない」ということには気軽にプログラミングをできるメリットがもちろんありますが、**思ってもみない値を変数に入れてしまいバグが起きる可能性がある**というデメリットもあります。

多くのプログラミング言語で静的型付けを採用しているメリットも考えて、**Javascriptでも静的型付けをできるようにしよう！**
という思想で生まれたのがこの**Typescript**です。

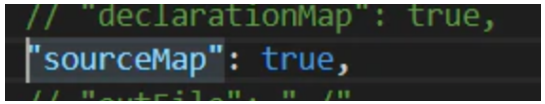
typescriptを使うためにインストールしましょう。
npm install -g typescript

tsc -v //バージョン確認しましょう

Visual Studio Codeで色々と設定していきます。
まずwindowsであればCドライブ直下にtypescript/samplesフォルダを作成。
macであればUser/ユーザー名/document直下にsamplesフォルダを作成してみてくだい。
作成したら、ターミナル、もしくはコマンドプロンプトなどで作成したディレクトリに移動しましょう。

移動したら
npm init -y　//nodeプロジェクトの初期化
package.jsonが作成されます。
npm tsc - - init // typescriptプロジェクトの初期化
tsconfig.jsonが作成されます。

tsconfig.jsonの設定を少し変えます



色々なツールからデバッグができるようにします。

使いやすいように、VSCodeから作成したディレクトリを開いておきましょう。

『ビルド用の設定』
開いたら、メニューのTerminal → Configure Default Build Task…
を選択し「tsc:build - tsconfig.json 」と検索して選択してください。

そして、動作確認の為のindex.tsというファイルを作成し。
console.log(‘hoge!’);と記載しておきましょう。

『デバッグ用の設定』
メニューのRun->Add Configuration…
Node.jsを検索して選択してください。
ここでlaunch.jsonが生成されます

これで準備完了です！。

動作させるには、メニューの
Terminal → Run Build Task　(ビルド)
Run →Start Debugging　（デバッグ）
をします。

デバッグは出てきましたが、ビルドとはなんぞや？と。
typescriptはこのプログラム自体が動作しているわけではなく、
typescriptがjavascriptに変換されて利用されています。
その為、先程のような細かい設定が必要だったわけです。

今回はざっくりとしたTypescriptの使い方を見ていきます。

型付け

Typescriptの最大の特徴である型付けを学んでいきます、
名前のそのまま「型」を決めることができます。
型を決めることで、誤ったデータが入ることを未然に防ぐことが可能です、ですがtypescriptも万能ではありません。
Typescript =変換=> Javascript　になり実行されるため、結局はJavascriptになってしまいます。
その為、Javascriptで型を無視して変更するようなコードを記述してしまうと全く意味がなくなりますので注意が必要です。

型付けの方法

　変数、関数の引数、戻り値に対して**：型『コロン＋型』**の形式で明示的に型を指定することができます。(| 区切りで複数可能)

```
let id: string = "hoge1";
let age: number = 2;

const twice = (count: number): string => {
    return `2倍にすると ${count * 2} です。`
}

let myCount: number
myCount = 100

console.log(twice(myCount))
```

変数と関数の引数は : 型　をつけるだけですが、
戻り値は引数に続いて　:型　で指定しています。
関数宣言であれば
`function 関数名（引数名:引数の型）：関数の戻り値の型 {}`
このような形になりますね。
型ではなく値での指定も可能です　**let food: “ラーメン”**

any型

any型を使うとJavascriptと同じように、なんでも入れることができるようになります。
なんでも入れられますが型のチェックが効かなくなります。

```
var n:number = 12345;
n = '12345';//コンパイルエラー

var hoge:any = 12345;
hoge = '12345';//any型の場合コンパイルエラーにならない

//nullも、undefinedもコンパイルエラーにならない
hoge = undefined;
hoge = null;

console.log(twice(myCount))
```

型推論

　型推論とはプログラムの文脈から、型を明示的に宣言しなくても型のチェックを行ってくれます。
例えば、変数宣言時に数値が代入された場合、その変数には数値が入ると型を設定してくれたり、関数の引数の型を明示的に定義している場合に、引数に受けわたす変数にも型を設定してくれます

```
var str = '';
str = 123;//初期化時に文字列を入れている為、変数の型定義無しでもコンパイルエラーとなる。
function hogef(n:number):void{
    alert(n *10);
}

var num = 10;
hogef(num);

var str = 'あいうえお';//引数の型定義をしている為、変数の型定義無しでもコンパイルエラーとなる。
hogef(str);
```

unknown型

unknown型はanyと同様、方が不明な時に活用できます。

```
const any1: any = null;
const any2: any = undefined;
const any3: any = true;
const any4: any = 0.8;
const any5: any = 'Comment allez-vous';
const any6: any = {
  x: 0,
  y: 1,
  name: 'origin'
};

const unknown1: unknown = null;
const unknown2: unknown = undefined;
const unknown3: unknown = true;
const unknown4: unknown = 0.8;
const unknown5: unknown = 'Comment allez-vous';
const unknown6: unknown = {
  x: 0,
  y: 1,
  name: 'origin'
};
```

anyとunknownはどの値も入れられるという点では似ています。
ですが、anyは型チェックというものを放棄した型です。
anyはプログラムを実行したタイミング（システムを利用しているタイミング）でエラーが出ますが、unknownはコンパイル時にエラーがでます。
型の利用が保証されていなければコンパイル時にプログラムのエラーを知ることができます。

anyとunknownの違いはまだあり。
any型のオブジェクトは、プロパティ、メソッドを利用することが可能です。
ですがunknown型のオブジェクトは、プロパティ、メソッドを利用することができません。

nuknown型の使い方。
1.スコープの中で最初からあるわけではないデータ
2.API をinterface（後で出てきます）で定義していく時に後回しにしたいプロパティの型
3.関数のオーバーロード定義時ただ引数の数を合わせたいだけの時の型
4.型の種類が多い（そして数が不明かつ API ドキュメントの無いカオスプロジェクト←そもそもこれが問題）
このような場合に、一旦unknownで置いておいて、使う時になってから実際の型に置き換えたり、キャストしたりするといいかなと思います。
（基本はunknownではなく型を固定してあげる、というのが前提です。）

オブジェクトに型を指定したい場合

```
//オブジェクトを適宜するとき
let obj: {
    name: string,
    age:number
} = {
    name: 'suzuki',
    age: 29
};
//引数にオブジェクトを設定する場合
function hoge(obj: { key1: string; key2: number; }): void {

}
hoge({ key1: "ABC", key2: 123 });

//プロパティの省略について。
//下記のようにプロパティの定義がないとコンパイルエラーになります
//？を用いることで、型定義済みにのキーを省略してもエラーが発生しなくなります
var obj: {key: string} = {};//エラーになる
var obj2: {key?: string} = {};//省略してもエラーにならない
```

オブジェクトにはこの後に出てくる、interfaceやtypeというものと利用されます。

JavascriptのSymbolとBigint(Javascriptの型)

Symbol型

　Symbol型はその値のみが一意になるように設計されているプリミティブ型です

```
const sym1: symbol = Symbol('Dove is a symbol of peace');
const sym2: symbol = Symbol('Dove is a symbol of peace');
console.log(sym1 === sym1);
// -> true
console.log(sym1 === sym2);
// -> false
```

でもSymbolってどこで使うんだ？と思いますよね。
一意になっていたら比較も自分自身以外falseになる、その使い道ってどうなん？

できた経緯に関して
https://qiita.com/naruto/items/312adeb6145eb6221be7
こちらのQiitaの記事が参考になります。

個人的な見解としては
「プロパティ名の重複を避ける」ことと、**「外部からアクセスできないプロパティを作成する」**が多いのではないかと思います。

```
const name = Symbol();
const log = Symbol();

const obj = {
    name : "太郎",   // nameという名前のプロパティ
    [name] : "花子",  // シンボル値nameのプロパティ

    log : function(){ console.log( this.name ) },
    [log] : function(){ console.log( this[name] ) }
}

console.log( obj.name ); // 結果: "太郎"
console.log( obj[name] );  // 結果: "花子"
```

symbolをプロパティに利用していると、外部から直接プロパティ名を指定して利用することができなくなります。
（プログラムの概念でいうとカプセル化に近いものになります、データをある特定の単位で隠蔽することを言います。）
symbolが定義しているところでなければ、プロパティ名を利用することができないため、言い換えれば宣言している所でしか利用できないということです。

Bigint
BigInt は10進数の整数リテラルの末尾に n をつけて 10n とするか、 BigInt() 関数を呼び出すことで作成することができます。

```
//  末尾にnをつける場合  と  BigInt()関数を呼び出す場合
const bg1: bigint = 100n;
const bg2: bigint = BigInt(100);
//number型と一緒に演算をすることはできません。どちらかに型を合わせる必要があります。
2n + 3;
// Operator '+' cannot be applied to types '2n' and '3'.

//number型が小数部を持っていない限り、より表現幅の広いbigint型に合わせる
2n + BigInt(3);
```

nを末尾につける方法は、es2020以降のバージョンのJavascriptで利用可能になっています。

余談：特定の値を設定するのとconstって何が違うのか？

```
let impossible: false = false;

impossible = false;
impossible = true; //error
```

impossibleには値falseしか入らないように指定していますね。
（値を指定し特定の値しか入らないようにする型をリテラル型と言います）

constでいいんじゃないのか？（値が 1 つしか設定されていない場合）
と思いますが、先にお話ししたTypescriptは実行される時にJavascriptとして実行されますので、定数として利用することはできません。

```
let impossible: false = false;
// ...
// Type assertion
impossible = true as false;
```

asはTypescriptの型のアサーションです、型を変換しています。
後に出てきます。

細かい話ですが
`number`型の`NaN`, `Infinity`, `-Infinity`はリテラル型として使うことができません。

型のアサーション

```
let val1: any = "abcde";
let len1: number = (val1 as string).length;

let num: number = 777;
let numString: string = num as string;
```

型のアサーションはanyからStringのように、規則の緩い型から、厳しい型に変換することは可能ですが、逆はできません。

が、number -> any -> stringということは可能です。

そして、厳密には型を変換してはいません。

型のアサーションが必要になったタイミングでどのようなことができるのか調べてみましょう。

interface　と　type

簡単に言うと、オブジェクトの型に名前をつけることができます。

```
interface Member {
  nickName: string;
  isHuman: boolean;
  level: number;
}

let apple: Member = {
  nickName: "りんご",
  isHuman: true,
  level: 0
};
```

例文を見てみましょう
interfaceを使用するには　**interface 型名 {key:型}**宣言することができます。
作成したinterfaceを利用する方法は他の型付きと同じで、
変数：作成した型名で利用することができます。

typeも見てみましょう（正式には型エイリアス type alias）

```
type Member = {
  nickName: string;
  isHuman: boolean;
  level: number;
};

let apple: Member = {
  nickName: "りんご",
  isHuman: true,
  level: 0
};
```

ぱっと見、少し書き方が変わっただけで同じように見えます。
次は２つの違いについてみていきます。

厳密に見ると

interfaceは型の宣言ですので、型に名前をつけます。

typeはどちらかというと無名で作られた型に参照のため別名をを与えるということをやってます。

前のページのinterfaceとtypeを見比べると セミコロンがついているか、いないのかの違いがあります。（；が不要、必要はのJavascirptの永遠の戦いは今は置いておきます

ブロックはセミコロン不要、代入式はセミコロン必要とい考え方でみています。）

```
function kansu () { ... }
const kansu = function () { ... };
```

関数の関数宣言と関数式の関係性に似ています。

関数を宣言して名前をつけ作成をするのと、変数に作成した関数の処理を代入している関係性です。

定義できる型の種類

typeであればオブジェクト以外の型も参照できます。

```
type Color = "白" | "黒" | "赤" | "緑";

let color: Color = "白";
color = "青"; //Type '"青"' is not assignable to type 'Color'.
```

拡張

interfaceは拡張が可能です。

```
interface User {
  name: string;
}

interface User {
  level: number;
}

const user: User = {
  name: "apple",
  level: 0
};

const user2: User = {
  name: "banana"
}; //Property 'level' is missing in type '{ name: string; }'
but required in type 'User'
```

再度宣言をしているように見えますが、実際にはプロパティを追加しています。
その為、user2を作成する時にはプロパティが足らずエラーになっています。
typeではこのようなことはできません。
（厳密にはできるんですけどね。）

公式ドキュメントにはinterfaceの利用を勧めています。

「Because an interface more closely maps **how JavaScript objects work by being open to extension**, we recommend using an interface over a type alias when possible.」

配列の要素チェック

```
let array1: number[]
let array2: string[]
let array3: any[]
let array4: Array<number>
let array5: Array<string>
let array6: Array<any>
```

3,4,5の<>は型引数と言うものです。
<>はダイヤモンド演算子、Arrayは型でArray型=配列のことです

Tuple

配列の各要素ごとに型指定できます。
以下ケースでエラーとなります。

1.要素ごとに指定した型と異なる型である
2.要素数が違う

```
let tup: [number, string, boolean]
tup = [1,'2',true]
```

Enum

数値集合を名前をつけて管理しやすくなります。

```
enum payStatus {cashOn = 1,creditCard = 2}
```

void

voidは関数の戻り値で利用できます。
戻り値がない場合に使用するのがvoidです。

```
const echo = (name: string): void => {
    console.log(`Hello, ${name}`)
}
```

never

処理が終了しない、決して何も返さない場合 neverを使用します

```
let error = (message: string): never => {
    throw new Error(message)
}
```

このサンプルプログラムは
throw文を使用して、ユーザーが定義したエラーを発生させています。
throw だけでもいいのですが、errorオブジェクトを作成するためにnew Error(エラーメッセージ)を宣言しています。
この関数は、エラーを発生させるだけの関数なので、戻り値はもちろんありません、そして処理が終了することもありません
（エラーが発生して処理が止まるため、実行して終了にはなりません）

throwは今回初めて出てきましたが、プログラムは途中でエラーが出ることはよくあります、例えば外部APIへの通信で、ユーザーのネット環境が悪く接続に失敗してしまうこともあります。
その際に、失敗（エラー）になっても、3回はリトライをする
設定を入れる必要があります、その時はerrorを捕まえることができる
try catch文　を利用していきます。　（下記はJSサンプルです）

```
function getRectArea(width, height) {
  if (isNaN(width) || isNaN(height)) {
    throw 'Parameter is not a number!';
  }
}
try {
  getRectArea(3, 'A');
} catch (e) {
  console.error(e);
  // expected output: "Parameter is not a number!"
}
```

null の場合（そのまま）

```
let hensu: string | null
hensu = null
```

undefined

一般的にはあってはならないことですが、具体的に使うとしたらこう言う感じになります。

```
const fn = (x?: string): string => {
  if (x === undefined) {
    return ''
  }
  return x.toUpperCase()
}

fn('aaa')
fn()
```

複数型の指定

interection(T1 & T2)

```
interface InterfaceA {
  aaa: string;
  bbb: string;
}
interface InterfaceB {
  ccc: string;
}
type Xxx = InterfaceA & InterfaceB
const fnXxx = (): Xxx => {
  return {
    aaa: 'aaa',
    bbb: 'bbb',
    ccc: 'ccc',
  }
}
```

両方の型の要素を兼ね備えている必要があります。

union (T1 | T2)

```
interface InterfaceA {
  hasContent: true;
  name: string;
  body: string;
}

interface InterfaceB {
  hasContent: false;
}

const xxx = (yyy: InterfaceA | InterfaceB) => {
  if (yyy.hasContent) {
    // yyy.hasContent === trueなので、yyyはInterfaceAの型に絞り込まれました。
    const zzz = `${yyy.name}_${yyy.body}`;
  }
};
```

最初に少し出てきました、いずれかの型を満たす場合に利用します。
サンプルコードは、特定条件のときに必須のプロパティを用意したい時の例です。

Classのアクセス修飾子

　これまで使用してきたクラスにはアクセス権というものはありませんでした、しかしどこからでもアクセスできてしまうのはプログラムの保守性に問題が出てきます、その為Typescriptでも他の言語と同じようにアクセス修飾子が使えるようになりました。

簡単にまとめると

アクセス修飾子なし：publicと同じ
public： どこからでもアクセス可能
protected：　自クラス、サブクラスからアクセス可能
private：　自クラスのみアクセス可能

```
class Animal {
  public name: string;  // プロパティにpublicアクセス修飾子
  public constructor(theName: string) { this.name = theName; }
// コンストラクターにpublicアクセス修飾子
  public move(distanceInMeters: number) { // メソッドにpublicアク
セス修飾子
    console.log(`${this.name} moved ${distanceInMeters}m.`); //
publicアクセス修飾子である`this.name`を使用することが可能
  }
}

const gorilla = new Animal('ゴリラ');
gorilla.move(10);
// -> 'ゴリラ moved 10m.'
gorilla.name = 'ゴリラゴリラ';
gorilla.move(20);
// -> 'ゴリラゴリラ moved 20m.'
```

protected の例

```
class Animal {
  public name: string;
  public constructor(theName: string) { this.name = theName; }
  protected move(distanceInMeters: number) { // `public`から
`protected`に変更
    console.log(`${this.name} moved ${distanceInMeters}m.`);
  }
}

const gorilla = new Animal('ゴリラ');
gorilla.move(10); // error TS2339: Property 'move' does not
exist on type 'Animal'
```

今回はmoveがprotectedになっています、自クラス、サブクラスからアクセスは可能です。
publicと同じように使用していますが、アクセス権がなくエラーになります。

```
class Animal {
  public name: string;
  public constructor(theName: string) { this.name = theName; }
  protected move(distanceInMeters: number) { // `public`から
`protected`に変更
    console.log(`${this.name} moved ${distanceInMeters}m.`);
  }
}
class Gorilla extends Animal {
  move(distanceInMeters: number) {
    super.move(distanceInMeters * 10);
  }
}
const gorilla = new Gorilla('早いゴリラ');
gorilla.move(10);
```

このように、サブクラスからスーパークラスのmoveにアクセスをしている、サブクラスを呼び出すようにして利用することが可能です。

private の例

```
class Animal {
  public name: string;
  public constructor(theName: string) { this.name = theName; }
  private move(distanceInMeters: number) {
    console.log(`${this.name} moved ${distanceInMeters}m.`);
  }
}

class Gorilla extends Animal {
  move(distanceInMeters: number) {
    super.move(distanceInMeters * 10); // Property 'move' is
private and only accessible within class 'Animal'.
  }
}
```

privateは自クラス内からのアクセスしかできないのでエラーになります。
他クラスからアクセスさせたくない場合などに有効です。

コンストラクタのアクセス修飾子

　クラスのアクセス修飾子とほとんど一緒です

簡単にまとめると

アクセス修飾子なし：constructorメソッド内のみアクセス可能

public：　自身のクラス内、継承クラス、インスタンス化されたクラスのどれからでもアクセス可能

protected：　自身のクラス、継承クラスからアクセス可能

private：　自身のクラスのみアクセス可能

```
class ConstructorInAccessModifier {
  constructor(
    arg0: number,
    public arg1: number,
    protected arg2: number,
    private arg3: number
  ) {
    console.log({ arg0, arg1, arg2, arg3 });
  }
}

class ConstructorOutAccessModifier {
  public arg1: number;
  protected arg2: number;
  private arg3: number;
  constructor(
    arg0: number,
    arg1: number,
    arg2: number,
    arg3: number
  ) {
    this.arg1 = arg1;
    this.arg2 = arg2;
    this.arg3 = arg3;
    console.log({ arg0, arg1, arg2, arg3 });
  }
}
```

```
const InAccess = new ConstructorInAccessModifier(1, 2, 3, 4);
InAccess.arg0; // エラー プロパティ 'arg0' は型
'ConstructorInAccessModifier' に存在しません。ts(2339)
InAccess.arg1;
InAccess.arg2; // エラー プロパティ 'arg2' は型
'ConstructorInAccessModifier' に存在しません。ts(2339)
InAccess.arg3; // エラー プロパティ 'arg3' は型
'ConstructorInAccessModifier' に存在しません。ts(2339)

const outAccess = new ConstructorOutAccessModifier(1, 2, 3, 4);
outAccess.arg0; // エラー プロパティ 'arg0' は型
'ConstructorOutAccessModifier' に存在しません。ts(2339)
outAccess.arg1;
outAccess.arg2; // エラー プロパティ 'arg2' は型
'ConstructorOutAccessModifier' に存在しません。ts(2339)
outAccess.arg3; // エラー プロパティ 'arg3' は型
'ConstructorOutAccessModifier' に存在しません。ts(2339)
```

結果を見るとわかると思いますが、コンストラクタにアクセス修飾子を記載しているかどうかの違いで動きは変わりません。

Readonly修飾子

名前の通りですが読み込み専用にすることができます、
ですがreadonly修飾子を使用した場合は変数宣言時もしくはコンストラクタ内で初期化する必要があります（最初のタイミングしか値を変更できない為）

Getter Setter

プロパティへのインターセプター（参照、代入、監視の意味）
でGetter/Setterがあります。
メソッドと違い getter/setterは（）不要です。

```
class Human {
  private _name: string;
  // Getter宣言
  get name(): string {
    return this._name;
  }

  // Setter宣言
  set name(name: string) {
    this._name = name;
  }
}

const human = new Human();
// Setterを利用
human.name = `田中太郎`;

// Getterを利用
console.log(human.name);  // 田中太郎
```

Getter/SetterはJavascript以外では一般的に使われている概念になります、さまざまな人がいろいろな書き方でプロパティにアクセスするより、Getter Setterを利用して行うことでアクセスを統一することができプログラムの可読性が上がります。

Getter の書き方

```
get 名前(): 型 {
  必要ならば処理();
  return 戻り値;
}
```

Setter の書き方

```
set 名前(変数 : 型) {
  必要ならば処理();
  保存処理();
}
```