# Typescript



## Typescriptとは

- 2012年にMicrosoftが開発したオープンソースのプログラミング言語でJavaScriptの上位互換（JavaScriptが持つ機能を維持しつつ、追加の機能をそなえる）
- JavaScriptと同じように記述ができる +α 型の指定ができる
- JavaScriptと同じ構文を用いているので、JavaScriptの経験があれば比較的習得は容易
- JavaScriptでできることはTypeScriptでもできるので、webサイトやwebアプリケーション開発に適した言語だが、特に大規模なWebアプリケーションを開発する際に威力を発揮する
- 型定義をすることでプログラムを動かさずとも、バグが無いか未然に検知できるようになり大規模な開発における生産性を向上させる
- TypeScriptは「AltJS」のひとつで次世代のJavaScriptの最有力候補と言われる

## Typescriptの特徴

- 型推論付きの静的型付け言語
  - 型推論：言語自体が変数の型を予測して補完してくれる機能。高速に処理できるメリットがある。
  - 静的型付け言語：変数やメソッドの戻り値にあらかじめ型を指定する
  - 動的型付け言語：実行時にデータ型を決める
- インターフェースとクラスの定義が可能（→大規模なシステムをクラスを使って細分化し、チームで分担しながら開発を進めるのに適している）JavaScriptで使えないクラス生成ができるので、どうしてもJavaScriptだと長くなるコーディングが、TypeScriptのクラスベースの開発によって簡略化することが可能
- AltJsの一つでJavascriptと相互に互換性を持っている
  - AltJsとはJavaScriptより優れた機能を持ち、コンパイル（トランスパイル）後はJavaScriptのコードが生成される言語
  - TypeScriptで書いたコードをJavaScriptに変換できるので、JavaScriptで書かれたコードを、処理に応じてTypeScriptで書き換えられる。またTypeScriptから直接JavaScriptを呼び出したり、反対にJavaScriptからTypeScriptを呼び出すこともできる。→ プログラムのメンテナンス性↑

## 学習内容まとめ

### 基本の型推論とアノテーション

- typescriptは勝手に型を推論してくれるけど、明示的に型をつけたい時は型アノテーションを使う。
- 絶対に知っておくべき基本の型
    - プリミティブ型
        - string：すべての文字列を扱う型
        - number：整数、浮動小数点数、整数、負数、Infinity（無限大）、NaN（非数）など全ての数値を扱う型
        - boolean：trueとfalseの2つの値を扱う型
    - 存在しないことを表現する型
        - null：値が欠如していることを表す
        - undefined：初期化されておらず値が割り当てられていないことを表す
    - 型を許容する
        - any：どんな型でも許容する = 全く安全でないのでなるべく回避する
        - unknown：どんな型になるのか不明、代入した値によって型が変化する

## 関数

- 関数で使われる特別な型
    - void：戻り値を持たない関数の戻り値
    - never：決して戻ることのない関数の戻り値
- 関数の型定義
    - パラメーター（関数宣言時に渡される値）
        - オプションパラメーター（省略可能なパラメーター）：オプショナルを表す `?` をつける

```
const sample1 = (a: string, b?:string): boolean => {}
```

        - デフォルトパラメーター（初期値をもつパラメーター）：`=` で指定する、順序は関係なく記述できる

```
const sample2 = (a: string, b = 'sample'): boolean => {}
```

        - レストパラメーター（不特定多数の引数を配列として受け取るパラメーター）：`...` を用いる、パラメーターの最後に1つだけ指定できる

```
const sample3 = (...numbers: number[]): number => {}
```

    - 戻り値（関数が返す値）
- 呼び出しシグネチャ
    - どのような関数なのかを表現する型定義
    - 記法は2種類
        - 省略記法：アロー関数と似た形

```

```

```
type Func1 = (param: string) => void
const func1: Func1 = () => {}
```

- 完全な記法：オブジェクトと似た形

```
type Func2 = {
 (param: string): void
}
const func2: Func2 = () => {}
```

## オブジェクト

- TypeScriptのobject型はobjectであることを伝えるだけで構造は定義されないので定義しないといけない
- オブジェクトの型定義
  - オブジェクトリテラル記法
    - 基本の形

```
let obj: {
  name: string,
  age: number,
} = {
  name: 'Tom',
  age: 20,
}
```

    - 特別なプロパティ
      - オプショナル（`?`）のついたプロパティ = あってもなくてもOK
      - `readonly` のついたプロパティ = 上書きできない

```
let obj: {
  readonly name: string,
  age?: number,
} = {
  name: 'Tom',
}
obj.name = 'John' // Error!
```

    - インデックスシグネチャ（ = オブジェクトの柔軟な型定義）
      - オブジェクトが複数のプロパティを持つ可能性を示す
      - `[key: T]: U` のように定義する
      - keyはstringかnumberのみ

```

```

```
const capitals: {
 [countryName: string]: string,
} = {
 Japan: 'Tokyo',
 Korea: 'Seoul',
}
```

- 型エイリアス
  - typeを使って型に名前をつけて宣言できる
  - 再利用できるので同じ型を何度も定義する必要がない + コードの見通しも良くなる
  - 型に名前をつけることで変数の役割を明確化

```
type Country: {
  language: string,
  name: string,
}
const japan: Country = {
  language: 'Japanese',
  name: 'Japan',
}
```

- 複雑な型の定義
  - 合併型（Union Types）：型 A, B の和集合（ = どちらかの型をもつ）を表す型
  - 交差型（Intersection Types）：型 A, B の積集合（ = 両方の型をもつ）を表す型

```
type Kick = {
  effect: boolean;
  hit: boolean;
};

type Punch = {
  hit: boolean;
  damage: number;
};

type KickOrPunch = Kick | Punch;
type KickAndPunch = Kick & Punch;
```

## 配列/タプル

- 配列の型定義
  - T[]
  - Array
  - 合併型も使える（けどあまり使わない方がいい）
- 配列の型推論

```
const generateArray = () => {
  const _array = [] // any[]と推論される
  _array.push(123) // number[]と推論される
  _array.push('ABC') // (string | number)[]と推論される
}
const array = generateArray()
array.push(true) // Error!
```

- タプル（各要素の数と型を定義した配列）の型定義
  - 基本の記法

    ```
    let response: [number, string] = [200, 'OK]
    ```

  - 可変長（レストパラメーター）も使える

    ```
    const friends: [string, ...string[]] = ['Tom', 'John', 'Mary']
    ```

- イミュータブル（= 書き換え不可）な配列/タプル
  - JavaScriptの配列はconstで宣言してもミュータブル（書き換え可能）
  - `readonly`でイミュータブルな配列/タプルを作れる

    ```
    const commands: readonly string[] = ['git add', 'git commit',
    'git push']
    const commands: ReadonlyArray<string> = ['git add', 'git commit',
    'git push']
    const commands: Readonly<string[]> = ['git add', 'git commit',
    'git push']
    ```

## ジェネリック型（総称型）

- 型の種類は異なるが同じデータの構造を持っているものを共通化するときに使う（= 型のポリモーフィズム）
- ポリモーフィズム（多態性）：オブジェクト指向の考え方の一つで、ある1つの関数（メソッド）の呼び出しに対し、オブジェクト毎に異なる動作をすることを言う。
- 型をパラメーター化する（後から実パラメータを渡す）
- `T`, `U`, `V`, `W` などがよく使われる

```
const stringReduce = (array: string[], initialValue: string): string
=> {}
const numberReduce = (array: number[], initialValue: number): number
=> {}
↓
type Reduce<T> = {
  (array: T[], initialValue: T): T
```

```
}
const reduce: Reduce<string> = (array, initialValue) => {} // 呼び出す時に具体的な型をバインド
```

- 呼び出しシグネチャ
  - 完全な記法
    - シグネチャ全体にジェネリック型を割り当てる

```
type genericReduce1<T> = {
  (array: T[], initialValue: T): T
}
```

    - 個々のシグネチャ全体にジェネリック型を割り当てる

```
type genericReduce2 = {
  <T>(array: T[], initialValue: T): T
  <U>(array: U[], iniUialValue: U): U
}
```

  - 省略記法（こっちの方が一般的）

```
type genericReduce1<T> = (array: T[], initialValue: T) => T
type genericReduce2 = <T>(array: T[], initialValue: T) => T
```

## クラス

- 役割
  1. まとめる：ある機能についてのデータと振る舞いをまとめる
  2. 隠す：外部から参照・改変できないようにする
  3. たくさん作る：同じ機能を持つクローンを量産できる
- 用語
  - プロパティ：クラスが持つデータ、フィールド、メンバ変数とも呼ばれる。
  - メソッド：クラスで宣言する関数のこと。
  - コンストラクタ：クラスからインスタンスを作る時に行う初期化。
  - インスタンス：クラスから作られたオブジェクト。クラスの機能を持つクローンみたいなもの。
- アクセス修飾子
  - private：そのクラスでのみアクセス可能
  - protected：そのクラスとサブクラスでのみアクセス可能
  - public：どこからでもアクセス可能（デフォルト）
- 抽象クラス
  - abstract修飾子がついたクラスで継承でサブクラスをつくるためのクラス
  - 抽象クラスはインスタンス化できない
- オブジェクト思考

    - 再利用のための仕組み・手段
    - JavaScriptはオブジェクト思考開発ができなくはないが型をつけれないのでやりにくい/オブジェクト思考の恩恵を受けにくい

## Interfaceと型エイリアスの違い

- Interface
    - interface宣言子で定義
    - 型エイリアスと違って `=` は不要
    - 同名のinterfaceを宣言すると型が自動的に結合（マージ）される

```
interface Bread {
  calories: number
}
interface Bread {
  type: string
}
const fracePan: Bread = {
  calories: 350,
  type: 'hard',
}
```

    - `extends` を使うことで継承したサブインターフェースを作れる

```
interface Book {
  page: number
  title: string
}
interface Magazine extens Book {
  cycle: 'daily' | 'weekly' | 'monthly' | 'yearly'
}
const jump: Magazine = {
  cycle: 'weekly',
  page: 300,
  title: '週刊少年ジャンプ',
}
```

    - `implements` を使うことでclassに型を定義できる

```
interface Book {
  page: number
  title: string
}
class Comic implements Book {
  page: number
  title: string
```

```
  constructor(page: number, title: string) {
   this.page = page
   this.title = title
  }
}
const popularComic = new Comic(200, '鬼滅の刃')
```

- 比較

| | 型エイリアス | Interface |
|---|---|---|
| **用途** | 複数の場所で再利用する<br>型に名前をつけるため | オブジェクト・クラス・関数の<br>構造を定義するため |
| **拡張性** | 同名のtypeを宣言するとエラー | 同名のinterfaceを宣言するとマージされる<br>（宣言のマージ） |
| **継承** | 継承はできない<br>交差型で新しい型エイリアスを作る | extendsによる継承ができる |
| **使用できる型** | オブジェクトや関数以外の<br>プリミティブ、配列、タプルも宣言可能 | オブジ ェクトと関数の型のみ宣言できる |
| **考慮事項** | **拡張しにくい不便さがある** | **拡張できることによりバグを生む可能性** |
| **いつ使う** | アプリ開発では型エイリアス | 不特定多数の人が使う<br>ライブラリ開発では Interface |

非同期処理

- 非同期処理とは
  - 通信が発生する処理で起きる
    - Web APIを叩く
    - データベースへクエリを投げる
  - 実行完了を待たずに次の処理に進む
  - Javascriptはシングルスレットの言語だけど効率よく処理を行うための機能
  - 長所は複数の処理を並行して効率よく実行できること
  - 短所は制御が難しい
    → Promiseやasync/awaitで非同期処理を同期的に制御する
    → 型をつけることでよりわかりやすく
  - 型の書き方
    - Promise

```
type Fetch = () => Promise<Item | null>
const fetch: Fetch = () => {
  // 非同期処理を行い、最終的にItemかnullを返す
}
```

    - async/await

```
async find(id: string): Promise<User> {
  const { name, age } = await findById(id)
  return {
    id,
    name,
    age,
  }
}
```