プログラミング

TypeScript

・まえがき ← TypeScriptを使うことのメリット(型付けなど)を全体的に説明しているだけ

・1章(p1〜p3) ← 再読の必要無し

・2章(p5〜p16) ← 再読の必要無し

・3章(p17〜p46) ← SymbolとObject型についてはもう一度読む。それ以外は再読する必要無し

・4章 関数(p47〜p85) ← ジェネレーター、イテレーター、ポリモーフィズムは再確認

・5章 クラスとインターフェイス(p87〜p115) ← ミックスインとデコレーターに関して再確認する。それ以外は再読の必要無し。ただし、スライドでクラスとインターフェースの違いなどについて再確認しておくこと

・6章 高度な型(p117〜p163) ← 最初のページ以外は再読の必要無し。スライドで再確認する

・

1章　イントロダクション

javascriptがミスを指摘するのはコードを実行した時であるが、typescriptはテキストエディターの中で入力したタイミングでエラーメッセージを与えてくれるため、テストの手間が省かれる。

イントロダクションは再読しなくてOK。

2章　TypeScript：全体像

型が何であるかを明示的にtypescriptに伝えるには、アノテーション(annotation)を使う。typescriptに型を推論させたければ、型を省略する。

let a: number = 1

let b: string = ‘hello’

let c: boolean[] = [true, false]

tsconfig.jsonはどのファイルをコンパイルすべきか、コンパイルの結果をどのディレクトリーに格納するか、どのバージョンのJavaScriptを出力するか、などのTypeScriptプロジェクトが定義する場所。

tslint.jsonは作成するコードに対して望むスタイルの規約を成文化する。つまり、コーディングスタイルを定義する ← 今後はJavaScriptのリンターであるESLintを、TypeScript用のプラグインとともに使うことが推奨されている。

復習すること

・p6の図解について理解し直す

・p12のtsconfigの概要について理解し直す

<https://qiita.com/ShinKano/items/88b67f49f088896e8de5>

3章　型について

型(type)は『値と、それを使ってできる事柄の集まり』

型アノテーション(type annotation：明示的な型の指定)

リテラル型

『ただ一つの値を表し、それ以外の値は受け入れない型』であり、具体例に関してはp23のコードを参照。

const c = true // true

let f: true = false // エラー

型の初歩

・any ← 「すべて」の値の集まりであり、anyを使えば「何でも」行うことができる。型が何か分からない場合のデフォルトの型。

・unknown ← unknown値が特定の型であることを想定した事柄は行えない。ただし、値同士の比較はできる(unknown型であることを意図的に明示する必要がある)

・boolean ← trueとfalseの二つの値があり、それらを比較したり否定したりできる

・bigint ← 大きな整数を扱うことができる。加減乗除と比較をサポート

・symbol ← よく分からないので復習

・null ← 値が欠如していることを意味する

・undefined ← あるもがまだ定義されていない

・void ← 明示的に何も返さない関数の戻り値の型(例えばconsole.log)。return文を持たない関数

・never ← 決して戻ることのない関数の型(例外をスローする関数や、永久に実行される関数など)

・enum(列挙型) ← ある型について取り得る値を列挙する方法。キーを値にマッピングする

let a: {

b: number

c?: string //stringであるプロパティcを持つ可能性がある

[key: number]: boolean　//aはbooleanである数値プロパティを任意の数だけ持つことができる

}

インデックスシグネチャ [key: T]: U

オブジェクトがより多くのキーを含む可能性があることをTypeScriptに伝える方法で「このオブジェクトは、型Tの全てのキーは、型Uの値を持たなければならない」

let airplaneSeatingAssignments: {

[seatNumber: string]: string

} = {

‘34D’:’Boris Chery’,

‘34E’:’Bill Gates’

}

型エイリアス(type alias:型の別名の意)

ある型を指し示す型エイリアスを宣言する。

また、型エイリアスはブロックスコープである。

繰り返される複雑な型を同じ記述を何回もしないように(DRY: dont repeat yourself)したり、ある変数が何の目的で使われているかを明確にしたりするために役立つ。

type Age = number

type Person = {

name: string

age: Age

}

配列

配列は初期化の時点で型が決まり、関数内で配列を定義していてそれを返した場合、TypeScriptはその配列にそれ以上拡張できない最終的な型を割り当てる。また、空の配列はany型として扱われる。

function buildArray() {

let a = []　// any[]

a.push(1) // number []

a.push(‘x’) // (string | number)[]

return a

}

let myArray = buildArray()

myArray.push(true) // 型 string | number のパラメーターに割り当てることができません

tapuru

タプルは配列のサブタイプ(派生系)で、固定長の配列を型付けするための特別な方法であり、タプルを宣言するときには、明示的に型付けする必要がある。

let moreTrainFares: ([number] | [number, number])[] = {

[3.75],

[8.25, 7.70],

[10.50]

}

可変長

let list: [number, Boolean, …string[]] = [1, false,‘a’,’b’,’c’]

4章　関数

オプションパラメーター(?)は必須パラメーターの後に設定するが、デフォルトパラメーターはその制約を受けない。TypeScriptはデフォルトパラメーターからパラメーターの型を推論できるので、明示的にアノテーションをする必要が無い。

map関数 ← 繰り返しの結果を配列として返す

reduce関数 ← 繰り返しの結果を値として返す

レストパラメーター

任意の数の引数を安全に受け付ける

function sumVariadic(...numbers: number[]): number {

return numbers.reduce((total, n) => total + n, 0)

}

sumVariadic(1, 2, 3) // 6と評価される

thisについて

メソッドが呼び出される時、thisはそのメソッドがプロパティとなっているオブジェクトを指す

APIについて

どこからでもアクセスできる口を開けておき、決まった形式でアクセスを受け付けて、仕様通りの結果を返すインターフェースとなる

ジェネレーター function\* createFibonacciGenerator

利用者が要求した時だけに次の値を計算するので無限のリストを生成するといった他の方法では困難なことができる。javascript関連に色濃い内容なので、違う部分で詳しく調べる

呼び出しシグネチャ

関数の型についてのTypeScriptの構文。型のみの構文で、値はなし

type Log = (message: string, userId?: string) => void

let log: Log = (

message,

userId = 'Not signed in'

) => {

let time = new Date().toISOString()

console.log(time, message, userId)

}

完全な呼び出しシグネチャ

type Log = {

(message: string, userId?: string): void

}

オーバーロードされた関数とは「複数の呼び出しシグネチャを持つ関数」

実装する側では、実際に実装可能な結合された一つの型が存在する必要があるため、オーバーロードする時は実装側で呼び出しシグネチャを手動で結合する必要がある。

type Reserve = {

(from: Date, to: Date, destination: Date): Reservation

(from: Date, destination: string): Reservation

}

let reserve: Reserve = {

from: Date,

toOrDestination: Date | string,

destination?: string

} => {

if (toOrDestination instanceof Date && destination !== undefined) {

// 宿泊旅行を予約する

} else if (typeof toOrDestination === 'string') {

// 日帰り旅行を予約する

}

}

ジェネリック型パラメーター

複数の場所で型レベルの制約を強制するために使われるプレースホルダーの型。多層型パラメーターとも呼ばれる。

type Filter = {

<T>(array: T[], f: (iterm: T) => boolean): T[]

}

配列から要素を取り除く代わりに、マッピング関数を使って、それぞれの要素を変換する

function map<T, U>(array: T[], f: (item: T) => U): U[] {

let result = []

for (let i = 0; i < array.length; i++) {

result[i] = f(array[i]);

}

return result

}

型駆動開発(type-driven development)

まず型シグネチャで概略を記述し、その後で値を埋め込むプログラミングスタイル。関数の型シグネチャを見るだけで、その関数について知る必要のあることのほとんどが分かるようになる。

再読

・thisの型付け(p52〜p56) ← そもそもjavascriptのthisについて理解できていない

・イテレーター (p56〜p57) ← 説明の内容が理解できていない

・文脈的型付け (p61) ← コードの内容が理解できていない

・p66の後半部分

・制限付きポリモーフィズム

5章 クラスとインターフェイス

スライドに全てまとまっているので参照する。

新しいインスタンスが生成されるときには、「コンストラクタ」が呼び出される。コンストラクタがオブジェクトのインスタンスを初期化する。

private(アクセス修飾子)

生成したクラスと同じインスタンス内のコードからは読み書きができるが、他の全てのクラスのインスタンスはアクセスできない。サブクラスも不可。

<https://techtechmedia.com/public-private-protected/>

protected

privateとは違ってサブクラスのインスタンスもアクセス可能。

readonly

最初の割り当て(constructer)の後では読み取りのみが可能。

abstract

抽象クラスと呼ばれ、インスタンス化するとエラーになる。基本的にはサブクラスを設定するために利用し、それらはその抽象メソッドも実装しなければならない。ただし、オーバーライド可能。

super

superを使うことで親クラスのメソッドやコンストラクターを呼び出すことができる

実装に関して

クラスを宣言する時にimplementsキーワードを使うと、そのクラスが特定のインターフェースを満たしていることを表現できる。過不足は許されない。

クラスに関して

・TypeScriptはクラスを、その名前によってではなく、その構造によって比較する

・クラスは値レベルと型レベルで語句を生成するだけでなく、型レベルで二つの語句(インスタンスとコンストラクター)を生成することができる

final ← クラスを拡張不可と指定したり、メソッドをオーバーライド不可と指定したりするために、いくつかの言語で使われるキーワード。TypeScriptで同じ挙動をするためにはコンストラクターをプライベートにする。

再読

・ミックスイン

・デコレータ

6章　高度な型

メモ

・スライドを見る

・再読の必要無し

スーパータイプとスーパークラス、サブタイプとサブタイプを区別すること。

サブクラスはJavaScriptにおける継承されたクラスであり、サブクラスは親のメソッドにプロトタイプチェインでアクセスできる。サブタイプはTypeScriptの型に関係している。

↓エラーになる

function deleteUser(user: {id?: number, name: string}) {

delete user.id

}

type LegacyUser = {

id?: number | string

name: string

}

let legacyUser: LegacyUser = {

id: '703331',

name: 'Xin Yang'

}

deleteUser(legacyUser)

オブジェクトAがオブジェクトBに割り当て可能となるためには、対応するそれぞれのプロパティに対して「Aのプロパティ <: Bのプロパティ(AのプロパティがBのプロパティのサブタイプ)」とならなければならない。

TypeScriptでは、複雑な型(オブジェクト、クラス、配列、関数の戻り値の型)は全て上記のように <:T であるものを要求する共変性であると言う。

割り当て可能性

型Bが要求されているところで別の型Aを使用できるかどうか

理解できなかったところ

・反変性について p122〜p125

7章 エラー処理

・スライドを見る

・Option型の処理