その他の型定義

目次

9.文字列リテラル

文字列リテラル(特定の文字列)をそのまま型として利用できる

```
type CardType = "heart" | "diamond" | "spade" | "clover";

function getCard(c: CardType) {
  //引数cに応じた処理を実装
}
getCard("heart");

getCard("star"); //エラー
```

サンプルコード：1_am_samplecode_9.ts

10.その他のリテラル型

もともとは文字列での利用に限られたリテラル型だったが、TypeScript2 以降では`number,boolean,enum`などの型も対象

```
//falsyな値の集合
type FalsyType = 0 | false | null | undefined;

//番号
type Num = 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9;

//列挙型の部分型

enum Lang {
  ENGLISH,
  SPANISH,
  KOREAN,
  VIETNAMESE,
  JAPANESE,
}
type AsianLang = Lang.KOREAN | Lang.VIETNAMESE | Lang.JAPANESE;
```

サンプルコード：1_am_samplecode_10.ts

11.リテラル型と型推論

const で宣言された定数 a は数値リテラル型 10

```
const a = 10;
```

数値リテラル型 50 とみなされるのは不便(50 以外の値を代入できなくなる)。`let`宣言された変数`b`を一般的な`number`型とみなす。これを型の widening という。

```
let b = 50;
```

サンプルコード：1_am_samplecode_11.ts

12.任意の方を受け入れるunknown型

なんでもありの型

unknown型の変数valが数値でも文字列でも boolean 型の配列でも受け入れる

```
let val: unknown = 10;
val = "Target";
val = [true, false, false];
console.log(val.trim()); //エラー

//型ガードを入れる
if (typeof val === "string") {
  console.log(val.trim());
}

let val2: any = 10;
val2 = "Target";
val2 = [true, false, false];
console.log(val2.trim()); //コンパイルは通るが実行時エラー
```

サンプルコード：1_am_samplecode_12.ts

unknown型のなんでもありは「なにか判らない=なんでもありうる」。よって、以降のメソッド/演算子の呼び出しを許さない

unknown と any の違い

種類	概要
any	どんな型でも許容する
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)

13.null 非許容型

`tsconfig.json`で`strict`、または`strictNullChecks`オプションに`true`を設定

```
let val1: string = undefined; //エラー
let val2: string = null; //エラー

let val3: string | undefined = undefined;
let val4: string | null = null;
```

サンプルコード：1_am_samplecode_13.ts

`string|undefined`で`string`型と`undefined`型だけを認める型となる。 `strictNullChecks`オプションを有効にした場合も`undefined/null` を代入可能(null 許容型)

値がないの種類	概要
undefined	初期化されておらず、値が割り当てられていない(何も存在しない)
null	(本当は値が存在しているはずだけれど)値が欠如している

> 変数があるにもかかわらず値が欠如している(null)は設計上おかしい場合が多いため極力使用しない。

14.型エイリアス

- 型エイリアス(Type Alias)は特定の型に対して別名(エイリアス)を設定するためのしくみ
  - 同じ型を何度も定義する必要がないので再利用性が高い
  - 型に名前を付けることで変数の役割を明確化できる
  - 複雑な型をより読みやすく短く表現することができる
- 主にタプル型、共用型、交差型などに対して短い名前を付ける用途で利用

共用型・タプル型

- 共用型とは、複数の型のうちいずれか 1 つを持つ型
- タプル型は異なる型の複数の要素を持つ配列を表現する型
- 通常の配列は同じ型の要素の集合を表現する
- タプル型は異なる型の要素を持つことができる

```
//タプル型にAnyTypeという名前を付与
type AnyType = [string, number, boolean];
//AnyType型の変数valを定義
let val: AnyType = ["abc", 12, false];
let val2: AnyType = ["abc", "12", false]; //エラー

//共用型の場合
type OrType = number | string;
function add(a: OrType, b: OrType): string {
  if (typeof a === "number" && typeof b === "number") {
    return String(a + b);
  }
  if (typeof a === "string" && typeof b === "string") {
    return a.concat(b);
  }
  return String(a) + String(b);
}
```

> 多用すると型があいまいになり、コードの読みやすさや保守性が低下する可能性がある

サンプルコード：1_am_samplecode_14-1.ts

共用体型・交差型

- 共用体型とは 2 つ以上の型のプロパティをすべて持つ型

```
//オブジェクトの場合
//共用体型
type Admin = {
  name: string;
  privileges: string[];
};
type Employee = {
  name: string;
  contractDate: Date;
};
//AdminとEmployeeを結合して新しくElevetedEmployeeを作成
type ElevetedEmployee = Admin & Employee;
const e1: ElevetedEmployee = {
  name: "rick",
  privileges: ["create-server"],
  contractDate: new Date(),
};

//リテラル型の場合
type Comb = string | number;
type Numeric = number | boolean;

type Mix = Comb & Numeric;
```

サンプルコード：1_am_samplecode_14-2.ts

- 交差型は、異なるプロパティを結合することはできる。
- `Mix`は、`Comb`と`Numeric`の両方の特性を持つ交差型
- `Comb`と`Numeric`は両方とも`number`型を含んでいるため`Mix`型は`number`型になる