関数

目次

0.関数に関する用語の整理

関数の種類

TypeScript の関数の書き方は 3 つある

- function 命令
- 関数リテラル
- アロー関数

関数のパラメーター

種類	概要
仮パラメーター	関数宣言時に渡される値(パラメーター)
実パラメーター	関数を呼び出すときに渡す値(実パラメーター)
戻り値	関数が返す値

1.関数：関数の基本構文(function 命令)

function 命令を使った関数定義

```
function name(param:ptyoe,...):rtypr{...statements...}
//name:関数名
//param:仮引数
//ptype:仮引数のデータ型
//rtype:戻り値のデータ
//statements:関数の本体
```

```
function circle(radius: number): number {
  const PI = 3.14;
  return radius * radius * PI;
}

console.log(circle(4));
```

サンプルコード：2_pm_samplecode_1.ts

2.関数：関数型(関数リテラル)

関数を式として表し、変数に代入することができる

```
(param:ptype,...) => rtype
//param:仮引数
//ptype:仮引数のデータ型
//rtype:戻り値のデータ型
```

型推論

型推論によって、1 個の`number`型引数を受け取り、`number`型の値を返す`function`型と認識される

```
let circle = function (radius: number): number {
  const PI = 3.14;
  return radius * radius * PI;
};
console.log(circle(3));
```

型を明示的に記載

```
let circle2: (radius: number) => number = function (radius: number): number {
  const PI = 3.14;
  return radius * radius * PI;
};
console.log(circle2(3));
```

サンプルコード：2_pm_samplecode_2.ts

3.関数：アロー関数(ラムダ式)

ECMAScript2015 で採用された関数記法=>(アロー)を利用することで、よりシンプルに関数を表現

```
(param:ptype,...):rtype=>{...statements...}
//param:仮引数
//ptype:仮引数のデータ型
//rtype:戻り値のデータ型
//statements:関数の本体
```

```
let circle = (radius: number): number => {
  const PI = 3.14;
  return radius * radius * PI;
};
console.log(circle(3));
```

サンプルコード：2_pm_samplecode_3.ts

関数の本体が 1 文である場合には、{}とreturnは省略可能。式の値がそのまま戻り値となる。

```
let circle = (radius: number): number => radius * radius * 3.14;
```

4.関数：アロー関数(ラムダ式)は this を固定

アロー関数と関数リテラルとでは this の扱いが異なる

関数リテラルの場合

- `Timer`コンストラクタの直下では`this`はインスタンス自身
- setInterval メソッドの配下では`this`は変化してしまい、インスタンスを参照しない。=`_this`に退避させる必要がある

用語	概要
コンストラクタ	実体を作成する関数のこと
インスタンス	実体

コンストラクタやインスタンスは class 構文とセットで使用されるイメージだが、JavaScript(TypeScript)では関数オブジェクトとコンストラクタを使うことで、疑似的なクラスを再現できる仕組みになっている。

```
type TimerFunc = {
  timer: number;
  setInterval(): void;
};
let Timer = function (this: TimerFunc) {
  let _this = this; //現在のthisを退避
  _this.timer = 0;

  //1000ミリ秒間隔でtimerプロパティをインクリメント
  setInterval(function () {
    _this.timer++;
  }, 1000);
};
```

アロー関数の場合

- `this`はアロー関数自身が宣言された場所によって決まる
- `_this`への退避が不要

```
let TimerArrow = function (this: TimerFunc) {
  this.timer = 0;
  setInterval(() => {
    this.timer++;
  }, 1000);
};
type TimerFunc = {
  timer: number;
  setInterval(): void;
};
```

this の値について

- 関数リテラルの場合、実行時に決まる(途中で変わる)
- アロー関数の場合、定義時に決まる(途中で変わらない)

サンプルコード：2_pm_samplecode_4.ts

5.関数でよく利用するデータ型(void,never)

`void`は関数が戻り値を返さないことを表す

```
function sayHello(name: string): void {
  console.log(`こんにちは${name}さん`);
}
sayHello("山田");
```

サンプルコード：2_pm_samplecode_5-1.ts

`never`は決してありえないことを意味する。常に例外を発生する、無限ループで終点に到達できないなどを示す

```
//例外スローで、終端には到達しない

function error(): never {
  throw new Error("Error");
}

//無限ループなので、終端には到達しない
function eternal(): never {
  while (true) {
    //処理
  }
}
```

サンプルコード：2_pm_samplecode_5-2.ts

6.関数の高度な表現(1)：オプションパラメーター

省略可能な引数(オプションパラメーター)

引数`userName` を省略したい場合、仮引数の名前の前に`?`を付与。 オプションパラメーターとも呼ばれる。使っても使わなくてもよいパラメーター。

オプションパラメーターを関数内で使用する時は *undefined* でないかどうかのチェックをいれること

```
function userInfo(userId: string, userName?: string): string {
  if (userName == undefined) {
    return `あなたのIDは${userId}です`;
  } else {
    return `${userName}さんのIDは${userId}です`;
  }
}
console.log(userInfo("st12345"));
```

サンプルコード：2_pm_samplecode_6.ts

7.関数の高度な表現(2)：デフォルトパラメーター

デフォルトパラメーター

- 仮引数の後ろに=既定値で表す

```
function userInfo(userId: string, userName = "NON NAME"): string {
  if (userName == "NON NAME") {
    return `あなたのIDは${userId}です`;
  } else {
    return `${userName}さんのIDは${userId}です`;
  }
}

console.log(userInfo("dummy", "NON NAME"));
```

サンプルコード：2_pm_samplecode_7.ts

- 既定値には式も指定できる
- 参照できるのは自分よりも前に定義されたものだけ

```
function plus(x: number = 1, y: number = x) {} //OK
function plus(x: number = y, y: number = 1) {} //NG
```

- 任意引数の後ろに必須引数は配置できない

```
function plus(x: number = 1, y: number) {}
plus(1); //エラー：引数yが省略されたとみなされる
```

- undefined(未定義値)を明示的に指定した場合にも、引数は省略されたとみなされる
- null を指定した場合には、既定値は適用されない。null は undefined と異なり、「値がない」という明示的な状態を表しているため

```
console.log(userInfo("dummy", undefined)); //OK省略とみなされる
console.log(userInfo("dummy", null)); //エラー nullを割り当てることはできない
```

8.関数の高度な表現(3):可変長引数

可変長引数(レストパラメーター)

- 仮引数の前に...(ピリオドを 3 個)を付与することで、可変長引数となる。渡された任意個数の引数を配列としてまとめて受け取る

```
function sumPrice(...price: number[]): number {
  return price.reduce((pre: number, next: number) => {
    return pre + next;
  }, 0);
}

console.log(sumPrice(100, 300, 500));
```

サンプルコード：2_pm_samplecode_8.ts

- 複数の値を受け取るので可変長引数の型も配列となる
- 複数引数を持つ場合は、最後に 1 つだけ指定できる

余談だが、例えば JavaScript では配列を分割代入する際に使用する

```
const arr = [1, 2, 3, 4, 5];
//すべて変数を書き出すことなくcに書き出すことができる
const [a, b, ...c] = arr;
console.log(a, b, c); // 1 2 [3,4,5]
```

9.関数のオーバーロード

- オーバーロードは同じ名前で、引数リスト、戻り値の型が異なる関数を定義すること
- 多重定義ともよばれる
- ひとつの関数に異なる関数シグネチャを複数持つ
- 関数シグネチャとは、どのような引数をとるか、どのような戻り値を返すかという関数の型のこと

```
//関数シグネチャ
function print(value: number): void; //シグネチャ1
function print(value: boolean): void; //シグネチャ2

//関数の実装部分(全てのシグネチャを網羅する)
function print(value: any): void {
  if (typeof value == "number") {
    console.log(value.toFixed(0));
  } else {
    console.log(value ? "真" : "偽");
  }
}

print(3.1415); //3
print(false); //偽
print("hogehoge"); //エラー
```

サンプルコード：2_pm_samplecode_9.ts

- 定義したシグネチャにないものは型違反でエラーとなる
- Java/C#など一般的なプログラミング言語ではこのように書かないので注意が必要

10.高度な型：共用型

共用型

- 共用型(Union Type)は複数の型の中のどれかを表す。型をパイプ|で区切ることで表現。

```
let val: string | number;
val = "hogehoge"; //ok
val = 10; //ok
val = false; //エラー

//配列の場合
let val: (string | number)[] = ["hogehoge", 10, "hoge"];
```

引数/戻り値型としての共用型

- 関数が複数の型を受け取る、あるいは、複数の型を返す可能性がある場合に、シンプルに表現できる。

```
function circle(radius: number): number | boolean {
  if (radius < 0) {
    return false;
  } else {
    return radius * radius * 3.14;
  }
}
console.log(circle(10)); //314
console.log(circle(-10)); //false
```

サンプルコード：2_pm_samplecode_10.ts

複数の型を許容できるのであれば、any型を利用しても良いのではと考えるかもしれないが、あらかじめ格納される値が想定できるのであれば、安易なanyは避けるべき。

11.高度な型：型ガード

型ガード(Type Guards)は変数の型を判定することで、対象となった変数の型を特定する。

```
function change(value: string | number) {
  //引数valueを大文字に
  return value.toUpperCase(); //エラー
}
```

typeof演算子を利用する

```
function change2(value: string | number) {
  if (typeof value === "string") {
    //ここでは引数valueはstring
    return value.toUpperCase();
  } else {
    //ここではnumber型
    return value.toFixed(1);
  }
}
```

サンプルコード：2_pm_samplecode_11.ts

クラス型の判定ではinstanceofまたはin演算子を利用

12.高度な型：ユーザー定義の型ガード関数

- 型判定のために関数を型ガードとして利用する
- 型ガード関数は戻り値が`引数 is 型名`

```
type Comic = {
  isbn: string;
  title: string;
};

type Magazine = {
  mcode: string;
  tite: string;
};

function getInfo(): Comic | Magazine {
  return {
    isbn: "",
    title: "",
  };
}
//型ガード関数の定義
function isComic(inf: Comic | Magazine): inf is Comic {
  return (inf as Comic).isbn != undefined;
}
//型に応じて操作を分岐
let info1 = getInfo();
if (isComic(info1)) {
  //ISBNコードを取得(Comic型の操作)
  console.log(info1.isbn);
} else {
  //雑誌コードを取得(Magazine型の操作)
  console.log(info1.mcode);
}
```

サンプルコード：2_pm_samplecode_12.ts

`if`ブロックは上記の型が`Comic`に特定され、そのメインバーにアクセスできるのみならず、`else`ブロックでは(`Comic`型である可能性が除外された結果)`Magazine`型のメンバーにアクセスできる