

# インターフェイス

---

## 目次

1. [インターフェイス宣言](#)
2. [構造的部分型](#)
3. [型としての this](#)
4. [型注釈としてのインターフェイス](#)

## 1. インターフェイス宣言

---

全てのメソッドが抽象メソッドである特別なクラス

```
interface name {  
    definition;  
}  
//name: インターフェイス名  
//definition: インターフェイスの定義
```

- メソッドはすべて抽象メソッド。`abstract`修飾子は不要
- アクセス修飾子も指定できない。`public`修飾子も不要(`public`は明らかなため)
- `static`メンバーも宣言できない

```
//getAreaメソッドを持ったFigureインターフェイスを定義  
interface Figure {  
    getArea(): number;  
}  
  
//Figureインターフェイスを実装したCircleクラス  
class Circle implements Figure {  
    constructor(private radius: number) {}  
    //getAreaメソッドを実装  
    getArea(): number {  
        return this.radius * this.radius * 3.14;  
    }  
}  
  
let c = new Circle(5);  
console.log(c.getArea());
```

### [3\\_am\\_samplecode\\_9.ts](#)

TypeScript の継承は一度に 1 つの継承しかできない(単一継承) ただし、複数のインターフェイスを継承することができる

## 2. 構造的部分型

型の構造にフォーカスして、それが互換性のある型であるかを判定する方式

```
interface Figure {
  getArea(): number;
}

//getAreaメソッドを持つが、Figureインターフェイスを明示的に実装しない
class Circle {
  constructor(private radius: number) {}
  //getAreaメソッドを実装
  getArea(): number {
    return this.radius * this.radius * 3.14;
  }
}

//Figure型の変数にCircleオブジェクトを代入
let c: Figure = new Circle(5);
console.log(c.getArea());
```

### 3\_am\_samplecode\_10.ts

明示的に特定のクラス/インターフェイスを継承/実装していなくても、`Circle`クラスは`Figure`インターフェイスが備えるメソッドを全て備えているので、`TypeScript`は`Circle`と`Figure`とが互換性があるとみなす

公称的部分型：C#/Java のように明示的にクラス/インターフェイスを継承/実装することによってのみ、型の互換性を判定するアプローチ

### 3.型としての this

---

戻り値を`this`型(自分自身)とすることで、メソッドの結果でもって別のメソッドを呼び出すメソッドチェーンのような記述が可能

```
class Calc {  
  constructor(private _count: number) {}  
  get count(): number {  
    // 現在値を取得するgetter  
    return this._count;  
  }  
  plus(count: number): this {  
    // 与えられた値countで加算  
    this._count += count;  
    return this;  
  }  
  minus(count: number): this {  
    // 与えられた値countで減算  
    this._count -= count;  
    return this;  
  }  
}  
let calc = new Calc(20);  
console.log(calc.plus(10).minus(5).count); // 結果: 25
```

#### [3\\_am\\_samplecode\\_11.ts](#)

**Polymorphic this types** 呼び出し元のクラスに応じて型が変化(`calc`から呼び出された`plus`メソッドは`Calc`オブジェクトを返す)

## 4.型注釈としてのインターフェイス

---

オブジェクト/クラス/関数の構造を定義する

```
interface Dog {  
  //Dog型を定義  
  type: string; //プロパティシグニチャ  
  info(): void; //メソッドシグニチャ  
}  
  
//Dog型の変数dを宣言  
let d: Dog = {  
  type: "チワワ",  
  info() {  
    console.log(`${this.type}は小型犬です。`);  
  },  
};  
d.info();
```

[3\\_am\\_samplecode\\_12.ts](#)