複合型

他のプリミティブ型、もしくは複合型自身を内部に含み、大きなデータを定義できるデータ型を「複合型」と呼びます。 配列、オブジェクトなどがこれにあたります。クラスを定義して作るインスタンスも複合型ですが、リテラルで定義できる配列、およびオブジェクトをここでは取り上げます。

配列

配列はTypeScriptの中でかなり多用されるリテラルですが、スプレッド構文、分割代入などが加わり、また、数々のメソッドを駆使することで、関数型言語のような書き方もできます。 配列は、次に紹介するオブジェクトと同様、リテラルで定義できる複合型の1つです。

```
// 変数に代入。型名を付けるときは配列に入れる要素の型名の後ろに[]を付与する
// 後ろの型が明確であれば型名は省略可能
const years: number[] = [2019, 2020, 2021];
const divs = ['tig', 'sig', 'saig', 'scig'];

// 配列に要素を追加。複数個も追加可能
years.push(2022);
years.push(2023, 2024);

// 要素から取り出し
const first = years[0];
```

タプル

Javaなどの配列は要素のすべての型は同じです。TypeScriptでは、配列の要素ごとに型が違う「タプル」というデータ型も定義できます。 裏のデータ型は配列ですが、コンパイラが特殊なモードの配列として扱います。 この場合違う型を入れようとするとエラーになります。 後述の `readonly` を使うことで読み込み専用のタプルを作ることもできますが、デフォルトは変更可能です。

配列のインデックスごとに何を入れるか、名前をつけることはできないため、積極的に使うことはないでしょう。

```
const movie: [string, number] = ['Gozilla', 1954];
movie[0] = 2019;
// error TS2322: Type 'number' is not assignable to type 'string'.
```

固定長の配列を表現する手段としても利用できます。

```
const r = 10;
const t = Math.PI * 0.5;
const pos: [number, number] = [r * Math.cos(r), r * Math.sin(r)];
// Tuple type '[number, number]' of length '2' has no element at index '2'.
```

`[string, ...string[]]`と書けば、1つは必ず要素があり、2つ以上の要素が格納できるタプル、というのも表現できますが、「これよりも少ない」は表現できません。

注釈

Pythonにもタプルはあります。これは要素の変更が不可能で、辞書のキーに使えたりするということで、配列とはかなり性質が異なっていて、言語にとっては重要な要素となっています。

一方、通常のデータ型として使うにはやはりインデックスアクセスしかできないため、可読性が劣り、インデックスとデータの種類の対応付けを人間が覚えるのはストレスがあるため、かなり初期から`namedtuple`（名前付きタプル）と呼ばれるクラスが提供されています。これは名前で要素アクセスができる、タプルと可換なデータ構造です。

TypeScriptの場合はリテラルで簡単にオブジェクトが作れますし、多くのタプルはオブジェクトで代替可能でしょう。

配列からのデータの取り出し

以前のJavaScriptは、配列やオブジェクトの中身を変数に取り出すには一つずつ取り出すしかありませんでした。 現在のJavaScriptとTypeScriptは、分割代入（`=`の左に配列を書く記法）を使って複数の要素をまとめて取り出すことができます。 `slice()`を使わずに、新しい残余（Rest）構文（`...`）を使って、複数の要素をまとめて取り出すことができます。

残余構文は省略記号のようにピリオドを3つ書く構文で、あたかも複数の要素がそこにあるかのように振る舞います。 残余構文は取り出し以外にも、配列やオブジェクトの加工、関数呼び出しの引数リストに対しても使える強力な構文です。 ここでは、2つめ以降のすべての要素を`other`に格納しています。

配列の要素の取り出し

```
const smalls = [
  "小動物",
  "小型車",
  "小論文"
];
// 旧: 一個ずつ取り出す
var smallCar = smalls[1];
var smallAnimal = smalls[0];
// 旧: 2番目以降の要素の取り出し
var other = smalls.slice(1);

// 新: まとめて取り出し
const [smallAnimal, smallCar, essay] = smalls;
// 新: 2番目以降の要素の取り出し
const [, ...other] = smalls;
```

配列の要素の存在チェック

以前は、要素のインデックス値を見て判断していましたが、配列に要素が入っているかどうかをbooleanで返す`includes()`メソッドが入ったので、積極的にこれを使っていきましょう。

要素の存在チェック

```
const places = ["小岩駅", "小浜市", "小倉駅"];

// 旧: indexOfを利用
if (places.indexOf("小淵沢") !== -1) {
  // 見つかった！
}

// 新: includesを利用
if (places.includes("小淵沢")) {
  // 見つかった！
}
```

配列の加工

配列の加工は、他言語の習熟者がJavaScriptを学ぶときにつまづくポイントでした。`splice()`という要素の削除と追加を一度に行う謎のメソッドを使ってパズルのように配列を加工していました。 配列のメソッドによっては、配列そのものを変更したり、新しい配列を返したりが統一されていないのも難解さを増やしているポイントです。 スプレッド構文を使うと標準文法の範囲内でこのような加工ができます。 さきほどのスプレッド構文は左辺用でしたが、こちらは右辺で配列の中身を展開します。

近年のJavaScriptでは関数型言語のテクニックを借りてきてバグの少ないコードにしよう、という動きがあります。 その1つが、配列やオブジェクトを加工していくのではなく、値が変更されたコピーを別に作って、最後にリプレースするという方法です。`splice()`は対象の配列を変更してし

まいますが、スプレッド構文を使うと、この方針に沿ったコーディングがしやすくなります。 配列のコピーも簡単にできます。

配列の加工

```
const smalls = [
  "小動物",
  "小型車",
  "小論文"
];
const others = [
  "小市民",
  "小田急"
];


// 旧: 3番目の要素を削除して、1つの要素を追加しつつ、他の配列と結合
smalls.splice(2, 1, "小心者");
// [ '小動物', '小型車', '小心者' ]
var newSmalls = smalls.concat(others);
// [ '小動物', '小型車', '小心者', '小市民', '小田急' ]

// 新: スプレッド構文で同じ操作をする
//     先頭要素の削除の場合、分割代入を使えばslice()も消せます
const newSmalls = [...smalls.slice(0, 2), "小心者", ...others]
// [ '小動物', '小型車', '小心者', '小市民', '小田急' ]

// 旧: 配列のコピー
var copy = Array.from(smalls);

// 新: スプレッド構文で配列のコピー
const copy = [...smalls];
```

配列のソート

配列は`sort()`メソッドを使います。これはインプレースで、その配列を変更します。ソートをそのまま実行すると、中の要素をすべて文字列化した上で、辞書順でソートします。

デフォルトでは文字列としてソートする

```
const numbers = [30, 1, 200];

numbers.sort();
// 1, 200, 30
```

数値が入っている場合に、期待と異なる動作をします。比較関数を引数に渡し、0より小さい数値（左辺を左側に）、0（等価）、0より大きい数値（左辺を右側に移動）を返すことで要素の並び替えのルールを設定できます。オブジェクトの場合はどのキーを使うかなども比較関数で吸収します。左辺が小さい時に負の数を返せば昇順に、逆を返せば降順になります。

ソート関数を渡す

```
const numbers = [30, 1, 200];
numbers.sort((a, b) => a - b);
// 1, 30, 200

const stations = [
   {name: "池袋", users: 558623},
   {name: "新宿", users: 775386},
   {name: "渋谷", users: 366128},
   {name: "東京", users: 462589}
];
// 駅の利用者数でソート
stations.sort((a, b) => a.users - b.users);
```

複数の条件でソートしたい場合は、if文を重ねて書いていってもいいのですが、同値条件が抜けたりしがちなので、いったん全て-1, 0, 1にしておいて、足し合わせて総合スコアを計算する方がミスが減りますし、条件の入れ替えはしやすいでしょう。

複合条件でソート

```
const stations = [
   {name: "大手町", lines: 5, yomi: "おおてまち"},
   {name: "飯田橋", lines: 7, yomi: "いいだばし"},
   {name: "永田町", lines: 5, yomi: "ながたちょう"},
];

function cmpNum(a: number, b: number) {
  return (a < b) ? -1 : (a === b) ? 0 : 1;
}
function cmpStr(a: string, b: string) {
  return (a < b) ? -1 : (a === b) ? 0 : 1;
}
// 乗り入れ本数→読みでソート
stations.sort((a, b) => {
  const lineScore = cmpNum(a.lines, b.lines);
  const yomiScore = cmpStr(a.yomi, b.yomi);
  // わかりやすく10倍しているが、2倍でもOK
  return lineScore * 10 + yomiScore;
});
```

非破壊のソートはないので、元の配列を変更せずにソートした結果だけを得たい場合は、前節のスプレッド構文を組み合わせて行います。

非破壊ソート

```
// 駅の利用者数でソート
const sorted = [...stations].sort((a, b) => a.users - b.users);
```

ループは`for ... of`を使う

ループの書き方は大きくわけて3通りあります。 C言語由来のループは昔からあるものですがループ変数が必要です。 `forEach()`はその後ES5で追加されましたが、その後は言語仕様のアップデートとともに`for ... of`構文が追加されました。 この構文は`Array`, `Set`, `Map`、 `String`などの繰り返し可能（iterable）オブジェクトに対してループします。 配列の場合で、インデックス値が欲しい場合は、 `entries()`メソッドを使います。 関数型主義的なスタイルで統一するために、 `for ... of`を禁止して`forEach()`のみを使うというコーディング標準を規定している会社もあります（Airbnb）。

```
var iterable = ["小金井", "小淵沢", "小矢部"];

// 旧: C言語由来のループ
for (var i = 0; i < iterable.length; i++) {
    var value = iterable[i];
    console.log(value);
}

// 中: forEach()ループ
iterable.forEach(value => {
  console.log(value);
});

// 新: for ofループで配列のインデックスが欲しい
for (const [i, value] of iterable.entries()) {
  console.log(i, value);
}
// 要素のみ欲しいときは for (const value of iterable)
```

注釈

この`entries()`メソッドは、出力ターゲットをES2015以上にしないと動作しません。 次のようなエラーがでます。

```
// error TS2339: Property 'entries' does not exist on type 'string[]'.
```

Polyfillを使うことで対処もできますが、Polyfillを使わない対処方法としては、 `forEach()`を使う（2つめの引数がインデックス）、旧来のループを使うしかありません。

速度の面で言えば、旧来の`for`ループが最速です。 `for ... of`や`forEach()`は、ループ1周ごとに関数呼び出しが挟まるため、実行コストが多少上乗せされます。 といっても、ゲームの座標計算で1フレームごとに数万要素のループを回さなければならない、といったケース以外ではほぼ気にする必要はないでしょう。

`forEach()`、 `map()`などのメソッドは関数型プログラミングの章でも紹介します。

iterableとイテレータ

前節の最後に `entries()` メソッドが出てきました。これは、一度のループごとに、インデックスと値のタプルを返すイテレータを返します。 配列のループのときに、インデックスと値を一緒に返すときにこのイテレータが登場しています。

```
const a = ["a", "b", "c"];
const b = [[0, "a"], [1, "b"], [2, "c"]];

// この2つの結果は同じ
for (const [i, v] of a.entries()) { console.log(i, v); }
for (const [i, v] of b) { console.log(i, v); }
```

この `entries()` は何者なんでしょうか？正解は、 `next()` というメソッドを持つイテレータと呼ばれるオブジェクトを返すメソッドです。 この `next()` は、配列の要素と、終了したかどうかのboolean値を返します。 イテレータ（厳密には外部イテレータと呼ばれる）はJavaやPython、C++ではおなじみのものです。

上記の `b` のように全部の要素を持つ二重配列を作ってしまえばこのようなイテレータというものは必要ありませんが、その場合、要素数が多くなればなるほど、コピーに時間がかかってループが回る前の準備が遅くなる、という欠点を抱えることになります。そのため、このイテレータという要素を返すオブジェクトを使い、全コピーを防いでいます。

オブジェクトにループの要素を取り出すメソッド（ `@@iterator` ）があるオブジェクトはiterableなオブジェクトです。 繰り返し処理に対する約束事なので「iterableプロトコル」と呼ばれます。 このメソッドはイテレータを返します。 配列は、 `@@iterator` 以外にも、 `keys()` 、 `values()` 、 `entries()` と、イテレータを返すメソッドが合計4つあります。

`for...of` ループなどは、このプロトコルにしたがってループを行います。 これ以外にも、分割代入や、スプレッド構文など、本特集で紹介した機能がこのiterableプロトコルを土台に提供されています。

`Array` , `Set` , `Map` 、 `String` などのオブジェクトがこのプロトコルを提供していますが、将来的に出てくるデータ構造もこのプロトコルをサポートするでしょう。 また、自作することもできます。

イテレータはループするときには問題ありませんが、任意の位置の要素へのアクセスなどは不便です。 イテレータから配列に変換したい場合は `Array.from()` メソッドか、スプレッド構文が使えます。

```
// こうする
const names = Array.from(iterable);

// これもできる
const names = [...iterable];
```

注釈

イテレータはES2015以降にしか存在しないため、スプレッド構文を使ってイテレータを配列に変換するのは、出力ターゲットがES2015以上でなければなりません。

```
const names = [...iterable];
```

読み込み専用の配列

TypeScriptの「`const`」は変数の再代入をさせない、というガードにはなりますが、C++のように、「変更不可」にはできません。TypeScriptにはこれには別のキーワード、`readonly`が提供されています。型の定義の前に`readonly`を付与するか、リテラルの後ろに`as const`をつけると読み込み専用になります。

```
// 型につける場合はreadonly
const a: readonly number[] = [1, 2, 3];
// 値やリテラルに付ける場合はas const
const b = [1, 2, 3] as const;
a[0] = 1;
// Index signature in type 'readonly number[]' only permits reading.
```

読み込み専用の配列は普通の変更可能な配列よりは厳しい制約となります。変更可能な配列は、readonlyな配列の変数や引数には渡すことができます。逆に読み込み専用の配列を変更可能な配列の変数に格納したり関数の引数に渡したりしようとするとエラーになります。

```
const readonlyArray: readonly number[] = [1, 2, 3];
const mutableArray: number[] = [1, 2, 3];

function acceptReadonlyArray(a: readonly number[]) {
}

function acceptMutableArray(a: number[]) {
}

// OK
const readonlyVar: readonly number[] = mutableArray;

// NG
const mutableVar: number[] = readonlyArray;
// The type 'readonly number[]' is 'readonly' and cannot be assigned to the mutable type 'number[]'.

// OK
acceptReadonlyArray(mutableArray);

// NG
acceptMutableArray(readonlyArray);
// Argument of type 'readonly number[]' is not assignable to parameter of type 'number[]'.
// The type 'readonly number[]' is 'readonly' and cannot be assigned to the mutable type 'number[]'.
```

内部的には同じ配列ではありますので、型アサーションで`readonly`なしのものにキャストすれば格納したり呼び出し時に渡したりは可能です。しかし、C/C++ではいわゆる「const外し」はプログラムの安全性を脅かす邪悪な行為として忌み嫌われます。C/C++の場合は組み込み機器で、読み込みしかできないメモリ領域にデータがおかれることもあり、動作が未定義で不正な挙動がおきうる、という意味ではTypeScriptよりもはるかに危険な行為ではありますが、「不変だと思っていた」変数がいつの間にかに書き換わっていたりして、開発者を混乱させる点では同じです。

この`readonly`を無理やり外したりせずに自然と使うためには、上から下までコード全体で`readonly`を使うように徹底するか、あるいは、まったく使わないかの二者択一になります。利用しているライブラリが`readonly`を使っているかというと、使っていないことが多いので、外部ライブラリとの接点では必ず`readonly`外しが必要になるかもしれません。ここはプロジェクト全体での意思統一が必要になる場面となります。

```
const mutableVar: number[] = readonlyArray as number[];
acceptMutableArray(readonlyArray as number[]);
```

TypeScriptと配列

`for ... of`には速度のペナルティがあるということを紹介しました。しかし、TypeScriptを使っている場合には少し恩恵があります。

TypeScriptを使っていると、ES5への出力の場合型情報を見て、`Array`型の`for ... of`ループの場合、旧来の最速の`for`ループのJavaScriptコードが生成されますので、速度上のペナルティがまったくない状態で、最新の構文が使えるメリットがあります。また、ChromeなどのJavaScriptエンジンの場合は、同一の型の要素だけを含む配列の場合、特別な最適化を行います。

TypeScriptを使うと、型情報がついて実装が簡単になるだけではなく、速度のメリットもあります。

配列のようで配列でない、ちょっと配列なオブジェクト

TypeScriptがメインターゲットとしてるブラウザ環境では、配列に似たオブジェクトがあります。HTMLのDOMを操作したときに得られる、`HTMLCollection`と、`NodeList`です。前者は`document.forms`などでフォームを取得してきたときにも得られます。どちらも`.length`で長さが取得でき、インデックスアクセスができるため、一見配列のようですが、配列よりもメソッドがかなり少なくなっています。`NodeList`は`forEach()`はありますが、`HTMLCollection`にはありません。`map()`や`some()`はどちらにもありません。

どちらもイテレータは利用できますので、次のようなコードは利用できます。

- `for .. of`ループ
- スプレッド構文
- `Array.from()`で配列に変換してから各種配列のメソッドを利用

オブジェクト

オブジェクトは、JavaScriptのコアとなるデータですが、クラスなどを定義しないで、気軽にまとまったデータを扱うときに使います。 配列は要素へのアクセス方法がインデックス（数値）でしたが、オブジェクトの場合は文字列です。 キー名が変数などで使える文字だけで構成されている場合は、名前をそのまま記述できますが、空白文字やマイナスなどを含む場合にはダブルクオートやシングルクオートでくくります。 また、キー名に変数を書く場合は`[ ]`でくくります。

オブジェクト

```
// 定義はキー、コロン(:)、値を書く。要素間は改行
const key = 'favorite drink';

const smallAnimal = {
  name: "小動物",
  favorite: "小籠包",
  'home town': "神奈川県警のいるところ",
  [key]: "ストロングゼロ"
};

// 参照は `.`+名前、もしくは [名前]
console.log(smallAnimal.name); // 小動物
console.log(smallAnimal[key]); // ストロングゼロ
```

おおきなプログラムをきちんと書く場合には、次の章で紹介するクラスを使うべきですが、次のようなクラスを定義するまでもない場面で出てきます。

- Webサービスのリクエストやレスポンス
- 関数のオプショナルな引数
- 複数の情報を返す関数
- 複数の情報を返す非同期処理

JSON（JavaScript Object Notation）

オブジェクトがよく出てくる文脈は「JSON」です。JSONというのはデータ交換用フォーマットで、つまりは文字列です。プレーンテキストであり、書きやすく読みやすい（XMLやSOAPと比べて）こともありますし、JavaScriptでネイティブで扱えるため、API通信で使われるデータフォーマットとしてはトップシェアを誇ります。

JSONをパースすると、オブジェクトと配列で階層構造になったデータができあがります。 通信用のライブラリでは、パース済みの状態でレスポンスが帰ってきたりするため、正確ではないですが、このオブジェクト/配列も便宜上、JSONと呼ぶこともあります。

JSONとオブジェクト

```
// 最初の引数にオブジェクトや配列、文字列などを入れる
// 2つめの引数はデータ変換をしたいときの変換関数（ログ出力からパスワードをマスクしたいなど）
//   省略可能。通常はnull
// 3つめは配列やオブジェクトでインデントするときのインデント幅
//   省略可能。省略すると改行なしの1行で出力される
const json = JSON.stringfy(smallAnimal, null, 2);

// これは複製されて出てくるので、元のsmallAnimalとは別物
const smallAnimal2 = JSON.parse(json);
```

JSONはJavaScript/TypeScriptのオブジェクト定義よりもルールが厳密です。 たとえば、キーは必ずダブルクオートでくくらなければなりませんし、配列やオブジェクトの末尾に不要なカンマがあるとエラーになります。 その場合はJSON.parse()の中で `SyntaxError` 例外が発生します。 特に、JSONを便利だからとマスターデータとして使っていて、非プログラマーの人に、編集してもらったりしたときによく発生します。 あとは、JSONレスポンスを期待しているウェブサービスの時に、サーバー側でエラーが発生して、 `Forbidden` という文字列が帰ってきた場合（403エラー時のボディ）にも発生します。

JSONパースのエラー

```
SyntaxError: Unexpected token n in JSON at position 1
```

オブジェクトからのデータの取り出し

オブジェクトの場合も配列同様、分割代入でまとめて取り出せます。 また、要素がなかったときにデフォルト値を設定したり、指定された要素以外のオブジェクトを抜き出すことが可能です。 注意点としては、まとめて取り出す場合の変数名は、必ずオブジェクトのキー名になります。 関数の返値や、後述の `Promise` では、この記法のおかげで気軽に複数の情報をまとめて返せます。

オブジェクトの要素の取り出し

```
const smallAnimal = {
  name: "小動物",
  favorite: "小籠包"
};

// 旧: 一個ずつ取り出す
var name = smallAnimal.name;
var favorite = smallAnimal.favorite;
// 旧: 存在しない場合はデフォルト値を設定
var age = smallAnimal.age ? smallAnimal.age : 3;

// 新: まとめて取り出し。デフォルト値も設定可能
const {name, favorite, age=3} = smallAnimal;
// 新: name以外の要素の取り出し
const {name, ...other} = smallAnimal;
```

ES2020で追加された機能として、オプショナルチェイニングがあります。TypeScriptでも3.7から導入されました。TypeScriptでは、変数の型として、文字列だけでなく、場合によっては無効な値として `null` や `undefined` が入る可能性がある、といったバリエーションを持たせることができます。型定義の話は基本的な型付けで触れるので、先行した説明になりますが、例えば次の定義は `smallAnimal` 自身がオブジェクト、もしくは `null` となりえますし、 `favorite` というメンバーも `undefined` になりえるという意味になります。

この場合、深い階層にアクセスする場合は、一つずつ、`null`や`undefined`になりえるところでチェックを行っていました。`&&`演算子が、一つでも途中にfalseyな値があると評価を止める、そうでなければ最後の値を返すという挙動を持っているため、それを活用したコーディングが行われていました。

オプショナルチェイニングは同じことを実現する演算子として`?.`が導入されました。途中でnullish（`null`か`undefined`）な値があると、式全体の評価結果が`undefined`になります。

```
const smallAnimal: {name: string, favorite?: string} | null = {
  name: "小動物",
  favorite: "小籠包"
};

// 旧: 一個ずつ確認してアクセスし、大文字の好物を取得
var favorite = smallAnimal && smallAnimal.favorite && smallAnimal.favorite.toUpperCase()

// 新: 一個ずつ確認してアクセスし、大文字の好物を取得
const favorite = smallAnimal?.favorite?.toUpperCase()
```

オブジェクトの要素の加工

JavaScriptではオブジェクトがリテラルで作成できるデータ構造として気軽に利用されます。 オブジェクトの加工（コピーや結合）も配列同様にスプレッド構文で簡単にできます。

```
const smallAnimal = {
  name: "小動物"
};

const attributes = {
  job: "小説家",
  nearStation: "小岩駅"
}

// 最古: オブジェクトをコピー
var copy = {};
for (var key1 in smallAnimal) {
   if (smallAnimal.hasOwnProperty(key1)) {
      copy[key1] = smallAnimal[key1];
   }
}

// 旧: Object.assign()を使ってコピー
const copy = Object.assign({}, smallAnimal);

// 新: スプレッド構文でコピー
const copy = {...smallAnimal};

// 最古: オブジェクトをマージ
var merged = {};
for (var key1 in smallAnimal) {
   if (smallAnimal.hasOwnProperty(key1)) {
      merged[key1] = smallAnimal[key1];
   }
}
for (var key2 in attributes) {
   if (attributes.hasOwnProperty(key2)) {
      merged[key2] = attributes[key2];
   }
}

// 旧: Object.assign()を使ってオブジェクトをマージ
const merged = Object.assign({}, smallAnimal, attributes);

// 新: スプレッド構文でマージ
const merged = {...smallAnimal, ...attributes};
```

辞書用途はオブジェクトではなくて`Map`を使う

ES2015では、単なる配列以外にも、`Map`/`Set`などが増えました。これらは子供のデータをフラットにたくさん入れられるデータ構造です。 これも配列と同じiterableですので、同じ流儀でループできます。 古のコードはオブジェクトを、他言語の辞書やハッシュのようにつかっていましたが、今時は`Map`を使います。 他の言語のようにリテラルで簡単に初期化できないのは欠点ですが、キーと値を簡単に取り出してループできるほか、キーだけでループ（`for (const key of map.keys())`）, 値だけでループ（`for (const value of map.values())`）も使えます。

辞書用途で見た場合の利点は、オブジェクトはキーの型に文字列しか入れることができませんが、`Map`や`Set`では`number`なども扱えます。

オブジェクトは、データベースでいうところのレコード（1つのオブジェクトはいつも固定の名前がある）として使い、`Map`はキーが可変の連想配列で、値の型が常に一定というケースで使うと良いでしょう。

`WeakMap`や`WeakSet`という弱参照のキャッシュに使えるコレクションもありますし、ブラウザで使えるウェブアクセスの`Fetch`APIの`Headers`クラスも似たAPIを提供しています。 これらのクラスに慣れておくと、コレクションを扱うコードが自在に扱えるようになるでしょう。

```
// 旧: オブジェクトを辞書代わりに
var map = {
  "五反田": "約束の地",
  "戸越銀座": "TGSGNZ"
};

for (var key in map) {
    if (map.hasOwnProperty(key)) {
        console.log(key + " : " + map[key]);
    }
}

// 新: Mapを利用
// ``<キーの型、 値の型>`` で明示的に型を指定すると
// ``set()`` 時に型違いのデータを入れようとするとチェックできるし、
// ループなどで値を取り出しても型情報が維持されます
const map = new Map<string, string>([
  ["五反田", "約束の地"],
  ["戸越銀座", "TGSGNZ"]
]);

for (const [key, value] of map) {
    console.log(`${key} : ${value}`);
}
```

注釈

`Map`、`Set`はES2015以降に導入されたクラスであるため、出力ターゲットをこれよりも新しくするか、ライブラリに登録した上でPolyfillを使うしかありません。

TypeScriptとオブジェクト

オブジェクトは、プロトタイプ指向というJavaScriptの柔軟性をささえる重要な部品です。 一方、TypeScriptはなるべく静的に型をつけて行く事で、コンパイル時にさまざまなチェックが行えるようになり不具合を見つけることができます。オブジェクトの型の定義については基本的な型付けの章で紹介します。

型定義をすると、プロパティの名前のスペルミスであったり、違う型を入れてしまうことが減ります。エラーチェックのコードを実装する手間も減るでしょう。

読み込み専用のオブジェクト

配列は`readonly`や`as const`をつけて読み込み専用にできましたが、オブジェクトも同様のことができます。ただし、`readonly`キーワードではできず、型ユーティリティの`Readonly<>`を使います。これには、型を定義しておく必要があります。これ以外にも、フィールドごとに`readonly`を付与することもで可能です。あるいは、型ではなく、値の最後に`as const`を付与します。前節でも触れましたが、これも詳しくは基本的な型付けの章で紹介します。

```
type User = {
    name: string;
    age:  number;
};

const u: Readonly<User> = {name: "shibukawa", age: 39};

// こちらでも良い
const u = {name: "shibukawa", age: 39} as const;

// NG
u.age = 17;
// Cannot assign to 'age' because it is a read-only property.
```

まとめ

JavaScriptの2大複合型の配列とオブジェクトを紹介しました。また、オブジェクトの関連のデータ構造として`Map`や`Set`も紹介しました。

Javaと比べると、TypeScriptで実装する場合、同じようなものを実装する場合にもクラス定義の数は減るでしょう。ちょっとしたデータを格納するデータ構造などは、これらの型を使って定義なしで使うことが多いからです。Javaからやってくると、これらの型を乱用しているように見えて不安になるかもしれません。しかし、TypeScriptを使えば、型推論やインラインでの明示的な型定義によって、これらの型でもきちんとしたチェックが行われるようになります。不安はあるかもしれませんが、安全にコーディングができます。