第3章

オブジェクトの基本とオブジェクトの型

本章では、TypeScriptにおいて非常に重要な概念であるオブジェクトについて解説します。オブジェクト指向言語として知られるプログラミング言語は多くあり、JavaScript/TypeScriptもそのうちのひとつです。しかし、それでは具体的にオブジェクトとは何か、ということになると言語によってその詳細には差が生じます。ですから、すでにオブジェクトという概念に何らかの形で触れたことがある読者の方も、油断せずにTypeScriptにおけるオブジェクトについて学びましょう。

3.1 オブジェクトとは

それでは、まずTypeScriptの**オブジェクト**とはどういうものかについて解説します。ただし、実はこの章で解説するのはTypeScriptのオブジェクトの機能のうち半分くらいです。もう半分の解説はもう少しあと、クラスの章（第5章）で行います。逆に言えば、この章ではクラスの話は出てこないということです。Javaなどの言語ではクラスとオブジェクトは切っても切り離せない関係にあることを考えると、クラスに触れずにオブジェクトの話をするというのは人によっては不思議に思えるかもしれませんが、TypeScriptではオブジェクトは必ずしもクラスに由来するものではないのです。

この章では、クラスに由来しないオブジェクト、すなわち**ただのオブジェクト**を扱います[注1]。TypeScriptではただのオブジェクトがたいへん幅広く利用されます。TypeScriptで何を作るのかにもよりますが、ただのオブジェクトの出番が圧倒的に多く、クラスはほぼまったく使われないということも珍しくありません。

3.1.1 オブジェクトは“連想配列”である

TypeScriptのオブジェクトは、一言で言えば**連想配列**です。連想配列という概念は聞き慣れない方が多いかもしれません。プログラミング言語によっては、連想配列は「辞書（dictionary）」とか「ハッシュ」といった名前で呼ばれることがあります。

改めてTypeScriptのオブジェクトについて説明すると、オブジェクトは**いくつかの値をまとめたデータ**です。オブジェクト内のそれぞれの値は名前で区別されます。さっそく例を出して説明します。

```
const obj = {
  foo: 123,
  bar: "Hello, world!"
};

console.log(obj.foo); // 123 と表示される
console.log(obj.bar); // "Hello, world!" と表示される
```

コードの冒頭、変数objに{ }で囲まれた何かが代入されています。これは**オブジェクトリテラル**（➡3.1.2）です。オブジェクトリテラルについては次の項で説明しますが、オブジェクトはこのようにオブジェクトリテラルを用いて作成されます。

オブジェクトリテラルの中には、**プロパティ**の定義がコンマ（,）で区切って並べられています。プロパティというのは、オブジェクトの中のひとつひとつの値を指す言葉です。オブジェクトリテラル中のプロパティの定義はプロパティ名: 式という形をとります。コロン（:）の右の式は、オブジェクトが作られた際にそのプロパティに入っている値を指定するものです。この例の場合、このオブジェクトにはfooとbarという2つのプロパティがあり、fooプロパティの値は123という数値で、barプロパティの値は"Hello, world!"という文字列です。この例からわかるように、オブジェクト内のそれぞれのプロパティは型が別々でもかまいません。

オブジェクトの中身（プロパティ）の値を得ることを**プロパティアクセス**と呼びます。プロパティアクセス

注1　プレーンオブジェクト（plain object）と呼ばれることもあります。

のためには式.プロパティ名という構文を用います。この構文は式の一種です。たとえば、変数objに入っているオブジェクトのfooプロパティを得るにはobj.fooという式を用います。今回の例では、objのfooプロパティには123が入っていたため、obj.fooは123となります。

オブジェクトの最も基本的な作り方と使い方は、以上で説明できました。しかし、オブジェクトに関してはまだまだ解説すべきことがあります。この章全体を通してオブジェクトについて学んでいくことになります。

繰り返しになりますが、オブジェクトはTypeScriptプログラミングにおいて非常に重要な概念です。現実のプログラムが扱う対象は複雑で、「文字列」とか「数値」といったプリミティブ1つ程度で表せるようなものはあまりありません。実際には、それらが複数組み合わさってできたデータを扱うことが要求されます。たとえば、SNSのユーザーを思い浮かべてみましょう。ものにもよりますが、1人のユーザーというデータを表現する際には、「名前」や「プロフィール画像」とか「フォロワー数」などといった複数のデータが必要になります。TypeScriptでは、このような場合にオブジェクトを用います。複数のデータをまとめる必要があればそれはオブジェクトの出番です。

3.1.2 オブジェクトリテラル（1）基本的な構文

では、次に**オブジェクトリテラル**[注2]の構文を細かく説明します。まず、オブジェクトリテラルは**式**の一種です。前項では`const obj = { ... }`のようにして変数objにオブジェクトを代入していましたが、変数宣言の構文では=の右に来るのは式でしたね。このことからもオブジェクトリテラルが式であることがわかります。`{ }`という記号を使っていますが、ブロックではないので注意しましょう。ブロックとオブジェクトリテラルの区別は、`{ }`が文の位置に書いてあるか式の位置に書いてあるかによってつけることができます。

`{ }`の中にはプロパティの定義を`,`で区切って書くことができます。プロパティの定義としては、プロパティ名: 式の形の構文を使うことができます。プロパティ名は、変数名と同様に識別子（➡2.2.2）を使用可能です。また、変数名に使用できない予約語（catch, defaultなど）もプロパティ名としては使用可能です。たとえば、**あいう**は識別子として利用可能なので`{ あいう: 0 }`は正しいプロパティ定義ですが、`↑↓↑↓`は識別子として利用不可能なので`{ ↑↓↑↓: "" }`は文法エラーとなります。また、最後のプロパティの後ろにも`,`を書くことが許されています。これにより、次のようにオブジェクトリテラルを書くことができます。fooだけでなくbarの定義の後ろにも`,`が書かれている点に注目してください。コンマはもともと区切り文字でしたが、要素の間だけでなく最後の要素の後ろにも区切り文字を書くことができるという文法が最近のトレンドです[注3]。実際に最後に区切り文字を書くかどうかは人やチームによるようです。とはいえ、本書執筆時点でのトレンドはPrettierに代表されるフォーマッターに自動的にプログラムの細かい体裁を整えてもらうことで、細部のこだわりはあまり持っていない人が増えています。

```
const obj = {
  foo: 555,
  bar: "文字列",
};
```

注2 正式には**オブジェクト初期化子**（object initializer）と言いますが、わかりやすさのため本書では通称としてオブジェクトリテラルと呼んでいます。のちのち出てくる配列リテラルについても同様です。

注3 最後の要素の後ろにも区切り文字を書くことができると、各要素の対称性が向上するという利点があります。すなわち、要素の間に`,`が並んでいるという見方を改めて、「要素`,`」のセットが並んでいるというよりシンプルな見方にすることができるのです。これは、Gitのようなバージョン管理システムで余計な差分を発生させない（新しい要素を追加する際に前の要素の最後に`,`を追加するという差分が発生しない）という実際的なメリットも存在します。

:の右に関しては、上で述べたように式を書くことができます。すなわち、これまでの例のように固定された数値や文字列を書くばかりではなく、変数の値を用いたりプロパティの値を直接計算したりすることができます。次の例は、ユーザーの名前（nameプロパティ）としてユーザーが入力した値（input）を用いるが、入力が空文字列の場合は"**名無し**"にするという例です。条件分岐は条件演算子（➡2.4.7）を用いています。コロン（**:**）が2回出てきて少し紛らわしいですが、よく見ると左はオブジェクトリテラルの一部で右は条件演算子の一部です。

```
const user = {
  name: input ? input : "名無し",
  age: 20,
};
```

オブジェクトリテラルには、よく使用される**省略記法**があります。次の例のように、プロパティに入れたい内容をあらかじめ計算して変数に入れた場合を考えましょう。

```
const name = input ? input : "名無し";
const user = {
  name: name,
  age: 20,
};
```

このように**プロパティ名: 変数名**という形の場合で、しかもこの例のように**プロパティ名**と**変数名**が同じである場合は、次のように省略できます。

```
const name = input ? input : "名無し";
const user = {
  name,
  age: 20,
};
```

すなわち、**: 変数名**の部分を省略して**プロパティ名,**だけになりました。一見何が起こっているのかわかりにくいように思えるかもしれませんが、この省略記法は頻繁に利用されます。ちょうどこの例のように、省略記法を使うためにこれから作るプロパティ名と同名の変数に値を入れておくということも行われがちです[注4]。プロパティが1つだけのオブジェクトを作る場合などは{ name }だけで済む場合もあり、たいへんコンパクトな記述が可能です。

3.1.3　オブジェクトリテラル（2）プロパティ名の種々の指定方法

オブジェクトリテラルにおいてコロンの左のプロパティ名として使えるものは、実は識別子だけではありません。ほかにも使えるものがいくつかありますが、特筆すべきは**文字列リテラル**です。すなわち、プロパティ名として、以下のように""や''で囲んだ名前を使用することができます。

```
const obj = {
  "foo": 123,
```

注4　もちろん、変数をいちいち経由せずに**プロパティ名: 式**の式部分に直接計算式を書くのが最もプログラムが短くなります。しかし、そうするとオブジェクトリテラルが巨大になってしまい読みにくくなるという問題があります。そのため、いったん変数にプロパティの中身を入れておいてからオブジェクトリテラルを用いるという行為自体は高い頻度で行われます。

```
  "foo bar": -500,
  '↑↓↑↓': ""
};

console.log(obj.foo);         // 123 と表示される
console.log(obj["foo bar"]); // -500 と表示される
```

このようにプロパティ名として文字列リテラルを用いることには、1つ大きな利点があります。それは、識別子としては使えないような文字列でもプロパティ名にできるという点です。こちらの方法では任意の文字列がプロパティ名として使用可能です[注5]。

言い方を変えれば、オブジェクトリテラル内で""や''で囲まずに宣言できるプロパティ名は一部のみ（名前が識別子で表現できるもののみ）であり、それ以外の名前は""や''が必須だということです。上の例では、最初の"foo":はfoo:でもかまいませんが、"foo bar":はfoo bar:とすることができません。また、前項で説明したとおり、'↑↓↑↓':も↑↓↑↓:とすると文法エラーとなります。

ただし、識別子ではないプロパティ名に対してはobj.fooのような式でアクセスすることができません。のちほど詳説しますが、[ ]の構文を用いる必要があります。

文字列リテラルのほかにもう1つ、オブジェクトリテラル内でプロパティとして使用できるものがあります。それは**数値リテラル**です。次の例のように:の左に数値リテラルを書くことが可能です。

```
const obj = {
    1: "one",
    2.05: "two point o five",
};
console.log(obj["1"]);    // "one" と表示される
console.log(obj["2.05"]); // "two point o five" と表示される
```

ただし、たとえプロパティ名に数値リテラルを書いたとしても、オブジェクトのプロパティ名が文字列であることは変わりません。あくまでオブジェクトリテラルの書き方として数値リテラルの形で書くことが許容されているだけです[注6]。

最後に、プロパティ名を動的に決めるための構文である**計算されたプロパティ名**（computed property name）を解説します。動的というのは、プログラム中での計算によって初めて決まるという意味です。これまで見てきた構文はすべてプロパティ名がソースコードにベタ書き（静的なプロパティ名）でしたが、ここで紹介する構文を使えば、変数名に入っている文字列をプロパティ名にするというようなことが可能になります。この構文は、プロパティ名のところ（コロンの左）に[式]という構文を書きます。お察しのとおり、式を評価した結果の文字列がプロパティ名として用いられます[注7]。具体例は次のとおりです。

```
const propName = "foo";
const obj = {
  [propName]: 123
};
```

注5 厳密には、文字列に加えてさらに**シンボル**もプロパティ名として使用することができますが、本書ではシンボルについては解説しません。

注6 1e3: "prop"のようにコロンの左に特殊な数値リテラルを書いた場合はとくに注意が必要です。数値リテラルとしては1e3は1000を意味するため、これは"1000": "prop"と同じ意味になります。"1e3": "prop"という意味にはなりません。もっとも、こんな書き方をすることはめったにありませんが。

注7 結果が文字列以外だった場合は文字列に変換されます（シンボルの場合を除く）。

```
console.log(obj.foo); // 123 と表示される
```

この例では変数propNameの値は"foo"ですから、[propName]: 123という構文で作られるプロパティの名前はfooです。よって、obj.fooとすると123が表示されました。

この構文を使う際は、型推論に関して気をつけなければならないことがあります。それについては本章のコラム9で説明します。

3.1.4　プロパティアクセス：値の取得と代入

プロパティアクセスとは、オブジェクトのプロパティの値を得たり、プロパティに代入したりすることです。前項までにも少し出てきたように、プロパティアクセスにはuser.ageという構文を用いることができます。これは式.プロパティ名という形の構文で、user.ageの場合はuserが式（変数名は式であり、その変数の中身を取得するという意味になるのでした）であり、ageがプロパティ名です。ageの部分は識別子（➡2.2.2）を与えます（オブジェクトリテラルの項（➡3.1.2）でも説明したように、予約語も可能です）。

この構文は**プロパティの値を取得する**、または**プロパティに代入する**という2種類の用途で使うことができます。プロパティの値を取得する場合は、単純にuser.ageを式として用います。たとえば、console.log(user.age)とすればuserに入っているオブジェクトのageプロパティの値が表示されます。一方、プロパティに値を代入する場合は、変数と同様に代入演算子（➡2.4.8）を用います。=演算子の左にプロパティアクセス構文を配置することで、そのプロパティに代入することができます。さっそく具体例を示します。この例では、最初25だったuser.ageが代入することで26に変わっています。

```
const user = {
  name: "uhyo",
  age: 25,
};

user.age = 26;
console.log(user.age); // 26 が表示される
```

ここでは=演算子を用いましたが、+=などほかの種類の代入演算子ももちろん使用可能です。

プロパティアクセス構文にはもう1種類あり、そちらの構文ではアクセスするプロパティ名を動的に決めることができます。こちらの構文は式1[式2]という形です。ここで式1はオブジェクトを表す式であり、式2はプロパティ名を表す式です。プロパティ名を動的に決めるために[ ]という記号を使うという点が前項の動的なプロパティ名と共通で、わかりやすくなっています。具体例としては、user.ageはuser["age"]と書きなおすことができます。この構文はプロパティ名を式で決めることができる点が特徴であり、これによりどのプロパティにアクセスするかをプログラムで決めることができます。

プロパティ名は文字列であるため、[ ]の中の式はstring型とするのが原則です。ただ、後述の配列（➡3.5）との兼ね合いから、数値（number型）も可能です[注8]。

やや人工的な例ですが、たとえばこんな使い方が考えられます。これはユーザーに数字を入力してもらってその数値が0以上かどうかによって表示を変える例です。

注8　本書には名前しか出てきませんが、実際にはさらにシンボルもプロパティ名として利用可能です。

```
import { createInterface } from 'readline';

const rl = createInterface({
  input: process.stdin,
  output: process.stdout
});

const messages = {
    good: "0以上の数値が入力されました！",
    bad: "負の数値を入力しないでください！"
}

rl.question('数値を入力してください:', (line) => {
  const num = Number(line);
  console.log(messages[num >= 0 ? "good" : "bad"]);
  rl.close();
});
```

このプログラムに123などの数を入力すると「0以上の数値が入力されました！」と表示される一方、-5などを入力すると「負の数値を入力しないでください！」と表示されます。これらのメッセージはあらかじめmessagesオブジェクトのgoodプロパティとbadプロパティに入っています。ポイントは、どちらのメッセージを表示するか決めるためのnum >= 0 ? "good" : "bad"という条件演算子（➡2.4.7）を用いた式です。これはnumが0以上なら"good"という文字列になり、それ以外は"bad"になります。この式はmessages[式]の式の部分で使われているため、numが0以上ならmessages["good"]が、それ以外ならmessages["bad"]が表示されます。

最後に余談ですが、これまで用いてきたconsole.log(...)のような関数は、よく見るとプロパティアクセスの構文を含んでいるように見えますね。実際、これはconsoleオブジェクトが持つlogプロパティを参照しています。このプロパティは関数が入っているので、関数呼び出しの構文で呼び出すことができます。ただし、プロパティに入っている関数（メソッド）を呼び出す際には注意が必要です。詳しくは第5章で説明します（➡5.4.1）。

コラム 8

オブジェクトのプロパティとconst

この項では、代入演算子を用いてuser.ageを書き換える例を紹介しました。注意深い読者の方は、「変数userはconstで宣言されているのに代入演算子を使えるのか」という疑問を持ったかもしれません。すでに見たとおり、答えはYesです。変数がconstで宣言されている場合は変数に再代入することはできませんが、それはあくまで変数そのものに対する規制です。すなわち、次のようにuser自体に別のオブジェクトを再代入する場合はコンパイルエラーになります。

```
const user = {
  name: "uhyo",
  age: 25,
};
```

```
// エラー: Cannot assign to 'user' because it is a constant.
user = {
  name: "John Smith",
  age: 15,
};
```

その一方で、変数に入っているオブジェクトの中身（プロパティ）を書き換えるのはconstによって制限されません。これは3.1.6の話題とも関連しますが、オブジェクトの中身がいくら書き変わろうとも（たとえ全部のプロパティの値を書き換えてしまったとしても）、変数userに入っているのは“同じ”オブジェクトのままだからです。TypeScriptのオブジェクトというのはこのように書き換えて中身を変えることができる存在であり、const変数は自身に“同じ”オブジェクトが入り続けていれば、その中身が書き換えられても文句を言わないのです。

とはいえ、最近はオブジェクトの書き換えを行わないプログラミングスタイルが流行しています。そのため、ケースバイケースではありますが、既存のオブジェクトの書き換えは忌避される傾向にあります。書き換えできないオブジェクトを宣言したい場合は、少しあとで紹介する読み取り専用プロパティ（➡3.2.7）をオブジェクト型の中で使うとよいでしょう。

3.1.5　オブジェクトリテラル（3）スプレッド構文

オブジェクトリテラル中では**スプレッド構文**（spread syntax）と呼ばれる構文を使用することができます。この構文を用いると、オブジェクトの作成時にプロパティを別のオブジェクトからコピーすることができます。スプレッド構文は**...**式という形の構文で、プロパティ**:** 式の代わりに使用することができます。

```
const obj1 = {
  bar: 456,
  baz: 789
};

const obj2 = {
  foo: 123,
  ...obj1
};

// obj2は { foo: 123, bar: 456, baz: 789 }
console.log(obj2);
```

この例ではobj2のオブジェクトリテラルの中でスプレッド構文が使われています。これにより、このオブジェクトリテラルはfooプロパティに加えてobj1由来のbarとbazプロパティを持っています。このように、スプレッド構文は既存のオブジェクトを拡張した別のオブジェクトを作りたい場合に有用です。この場合、obj2はobj1にさらにfooプロパティを加えたオブジェクトとなっています。

スプレッド構文と通常のプロパティ宣言が同じプロパティを与える場合、あとに書かれているほうが採用されます。次の例では、obj2では**...**obj1よりあとにfoo**:** -9999が書かれているため、obj2のfooプロパティはobj1由来の123ではなく-9999となります。

```
const obj1 = {
  foo: 123,
```

```
  bar: 456,
  baz: 789
};

const obj2 = {
  ...obj1,
  foo: -9999,
};

// obj2は { foo: -9999, bar: 456, baz: 789 }
console.log(obj2);
```

一方、次のように...obj1よりも前にfoo: -9999を置くのはコンパイルエラーとなります。その理由は、...obj1によってfooが上書きされると決まっているのに、それより前にfooを書くのは無意味だからです。

```
const obj1 = {
  foo: 123,
  bar: 456,
  baz: 789
};

// エラー: 'foo' is specified more than once, so this usage will be overwritten.
const obj2 = {
  foo: -9999,
  ...obj1
};
```

スプレッド構文は1つのオブジェクトリテラルの中で複数回使うこともできます。たとえば{ ...obj1, ...obj2 }のようなオブジェクトリテラルが可能です。もしobj1とobj2が同じ名前のプロパティを持つ場合、やはりあとのものが優先されます。この場合はobj2にあるものが優先されることになります。

```
const obj1 = {
  foo: 123,
  bar: 456,
};

const obj2 = {
  bar: -999,
  baz: -9999,
};

const obj3 = {
  ...obj1,
  ...obj2
};

// obj3は { foo: 123, bar: -999, baz: -9999 }
console.log(obj3);
```

上の例では、obj3にはobj1とobj2の両方のプロパティが含まれています。両方に存在するbarについては、あとに書かれたobj2のbarが採用されます。

なお、スプレッド構文によって行われるのはプロパティの**コピー**であるという点に注意してください。コピー元のオブジェクトのプロパティを変更しても、コピー先のオブジェクトには影響しません（例は次項を参照してください）。

3.1.6　オブジェクトはいつ“同じ”なのか

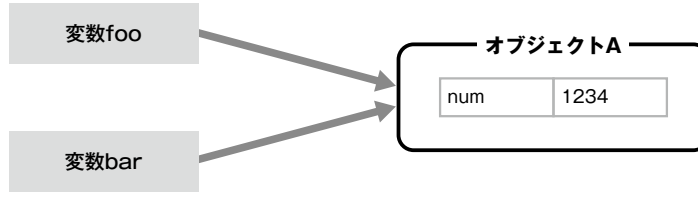
TypeScriptでは、オブジェクトがいつ“同じ”なのかということに注意を払わなければなりません。オブジェクトの等値性に対する理解があいまいだと、同じだと思っていたオブジェクトが同じでなかったり、同じでないと思っていたオブジェクトが同じだったりという事態が発生してプログラムが思わぬ挙動をすることになります。

とくに、TypeScriptではオブジェクトが暗黙にコピーされることはなく、複数の変数（やオブジェクトのプロパティ）に同じオブジェクトが入る場合があります。このような特徴を持つプログラミング言語は珍しくありませんが、それでもなお間違えやすいところです。次の例は、変数fooとbarに入っているオブジェクトが同じである例です。

```
const foo = { num: 1234 };
const bar = foo;
console.log(bar.num); // 1234 と表示される
bar.num = 0;
console.log(foo.num); // 0 と表示される
```

この例では変数fooには`{ num: 1234 }`というオブジェクトが入っています。これをオブジェクトAと呼ぶことにしましょう。すなわち、`{ num: 1234 }`というオブジェクトリテラルでオブジェクトAを作成し、それを変数fooに代入したのです。そして、変数barにfooの中身を代入しました。変数fooの中身とはオブジェクトAですから、変数barにはオブジェクトAが入ります。ここで、変数fooとbarにはどちらもオブジェクトAが入っているので、fooとbarが同じオブジェクトであると言えます。ポイントは、fooとbarという2つの変数があるのに対して、それらに入っているオブジェクトの実体はオブジェクトAという1つしかないということです。**図3-1**のように、変数はオブジェクトそのものというより、別のところにあるオブジェクトの実体を指し示すものであると考えるのがよいでしょう[注9]。

図3-1　変数たちとオブジェクトの関係



上の例を読み進めると、3行目で`bar.num`を取得しています。これはオブジェクトAのnumプロパティを取得するという意味になるので、`bar.num`は1234です。4行目の`bar.num = 0`という代入も、オブジェクトA

注9　このことは、**参照**（reference）という用語を用いて「変数foo・barに入っているのはオブジェクトAへの参照である」と説明されることもあります。「参照渡し」という用語もありますが、これは別の概念なので注意しましょう。

のnumプロパティに0を代入していることになります。最後に5行目でfoo.numを取得したときについても、fooの中身がオブジェクトAであるためオブジェクトAのnumプロパティの値が取得されます。これはたった今0に書き換えられたばかりでした。

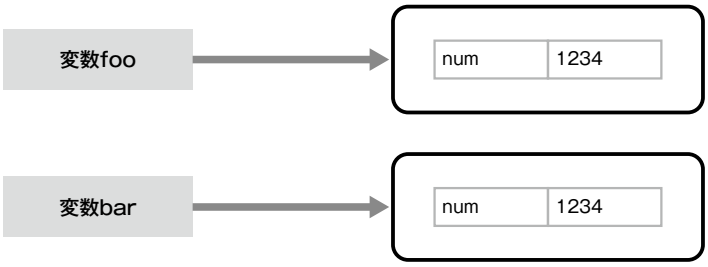
このように、変数にオブジェクトが入っていても、変数がそのオブジェクトを占有しているとは限りません。ほかの場所でも同じオブジェクトを保持しており、そちらからオブジェクトが書き換えられることがあるのです。「オブジェクトはいつ"同じ"なのか」という問いに対する答えは、「**明示的にコピーしなければ同じである**」ということになります。上の例のように「オブジェクトを別の変数に入れる」という操作はオブジェクトをコピーしていないので、同じオブジェクトが複数の変数に入る結果となります。

では、オブジェクトを明示的にコピーするとはどういうことでしょうか。1つの方法は、前項で説明したスプレッド構文を使って次のようにすることです。原則として、別々のオブジェクトを得るにはオブジェクトを別々に作成する必要があります。スプレッド構文はオブジェクトリテラルの中で使える構文であり、オブジェクトリテラルは新しいオブジェクトを作る構文ですから、確かにオブジェクトのコピーになっています。

```
const foo = { num: 1234 };
const bar = { ...foo }; // { num: 1234 } になる
console.log(bar.num); // 1234 と表示される
bar.num = 0;
console.log(foo.num); // 1234 と表示される
```

こうした場合、barはfooのプロパティをコピーして得られた新しいオブジェクトになります。つまり、fooに入るオブジェクトとbarに入るオブジェクトは別々のオブジェクトであるということです（**図3-2**）。これならば、bar.numを書き換えてもfoo.numが影響を受けることはありません。このように、別々のオブジェクトが欲しければ別々にオブジェクトを作成するというのが基本原則です。これを怠ると、別々のオブジェクトを作ることを意図していたのに実は同じオブジェクトが使いまわされていたという事態に陥ります。

図3-2 fooとbarに別々のオブジェクトが入る様子



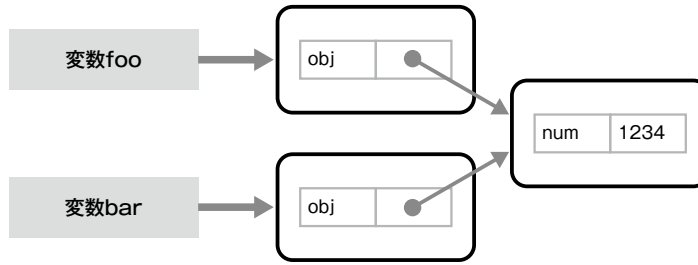
なお、同じ中身のオブジェクトを作るからといって必ずしもスプレッド構文を使う必要はありません。上の例の場合、次のように愚直に書くことも可能です。スプレッド構文を使うと後述の罠にはまりやすいので、このように書くのが望ましいこともあります。

```
const foo = { num: 1234 };
const bar = { num: 1234 };
```

スプレッド構文でオブジェクトをコピーする場合、ネストしたオブジェクトに注意が必要です。スプレッド

記法を使う方法はあくまで「各プロパティが同じ値を持つ新しいオブジェクトを作る」というものであり、プロパティの中にオブジェクトが入っていた場合はそれは相変わらず同じオブジェクトのままです。次の**図3-3**の例ではfooとbarは別々のオブジェクトですが、foo.objに入っているオブジェクトとbar.objに入っているオブジェクトは同じオブジェクトです。

図3-3　ネストしたオブジェクトの模式図




```
const foo = { obj: { num: 1234 } };
const bar = { ...foo };
bar.obj.num = 0;
console.log(foo.obj.num); // 0 と表示される
```

ネストしたオブジェクトも含めて全部コピーしたい（ネストしたオブジェクトも使いまわされないようにしたい）場合の標準的な方法は今のところありません[注10]。スプレッド記法をネストしたオブジェクトに対しても使用する（const bar = { obj: { ...foo.obj } };）のが1つの方法ですが、これは記述が多くなってしまうという欠点があります。これを行ってくれるライブラリはいろいろありますから、それらを用いて行うのが実際のところよくとられる選択肢です。

最後に、オブジェクトに対して一致判定の演算子===（➡2.4.4）を用いた場合の挙動について説明します。オブジェクト同士を===で比較した場合は、両辺が同じオブジェクトである場合にtrueとなります。

```
const foo = { num: 1234 };
const bar = foo;
const baz = { num: 1234 };

console.log(foo === bar); // true と表示される
console.log(foo === baz); // false と表示される
```

この例では、fooとbarは同じオブジェクトなのでfoo === barはtrueとなります。一方、fooとbazは別々に作られたオブジェクトであり同じではないので、foo === bazはfalseとなります。このように、中身がまったく同じだったとしても、別々のオブジェクトは===で比較するとfalseになります。オブジェクト自体が"同じ"かどうかではなく中身が一致しているかどうかで比較したいという場面は多くありますが、そのための標準的な方法は今のところありません。

ちなみに、{ num: 1234 } === 1234のようにオブジェクトとオブジェクト以外を比較した場合は必ずfalseとなります。===ではなく==を使った場合はまた挙動が違いますが、==をあえて使う必要もないので本

注10　Webブラウザ上で動くプログラムの場合、structuredCloneという関数がHTML標準として定義されているためこれが利用可能かもしれません。

書では解説を省略します。

3.2 オブジェクトの型

この節では**オブジェクトの型**について解説します。これまで数値を表すnumber型や文字列を表すstring型などが登場しましたが、ここで登場するのは**オブジェクトを表す型**です。オブジェクト型を用いることで、TypeScriptで扱うオブジェクトの型を宣言・制限できます。オブジェクトはTypeScriptで非常に頻繁に使われるデータですから、それを表すオブジェクト型もまた非常に頻出です。

この節では、さまざまな種類のオブジェクト型を理解し、オブジェクト型を使って型注釈を書く方法を学びます。また、オブジェクト型の扱いを通して自分で型を宣言する方法も学びます。

3.2.1 オブジェクト型の記法

まず最初に、オブジェクト型の記法を学びましょう。VS Code（または自分のお好きなエディタ）で以下のコードを書き、変数objの上にマウスカーソルを乗せてみましょう。すると、変数objの型が表示されます（**図3-4**）。この章ではこれまで変数に型注釈を書いてきませんでしたが、型推論により変数の型は常に存在しています。変数objに入るのはオブジェクトですから、TypeScriptがオブジェクト型を推論してくれたことが期待できます。まずはそれを調べてみましょう。

```
const obj = {
  foo: 123,
  bar: "Hello, world!"
};
```

調べてみると、objの型はこのように表示されます。

```
const obj: {
  foo: number;
  bar: string;
}
```

図3-4　VS Codeで変数objの型が表示されるところ