2-3 TypeScriptコードを実行してみる

さあ、最後の仕上げです。前節でコーディングした初めてのTypeScriptコードを実行させましょう。

2-3-1 ターミナルパネルを起動する

1-1-3項で説明したように、TypeScriptコードは、そのままでは実行できません。一度、JavaScriptコードへの変換、つまり、**コンパイル**が必要です。リスト2-1の内容で作成したhelloworld.tsファイルをコンパイルしてみましょう。

tsファイルのコンパイルは、コマンドで行います。そのコマンドは、対象tsファイルが含まれているフォルダー上で実行する必要があります。サイドバーのchap02フォルダーを右クリックし、表示されたメニューから［統合ターミナルで開く］を選択します（図2-17）。

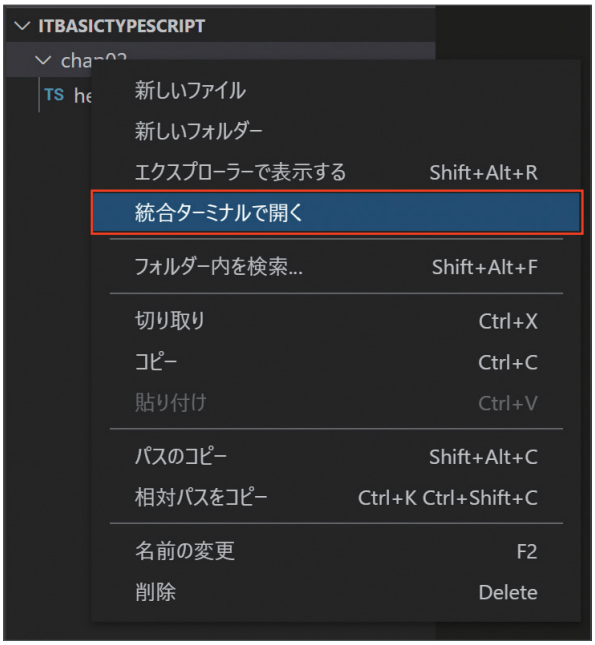


図2-17 chap02フォルダーを右クリック

すると、図2-18のようにターミナルパネルが開きます。この状態で、すでにchap02を作業

3-1 リテラルとデータの種類を理解する

早速、このChapterのテーマである変数と演算子の話に入りたいところですが、その前に理解しておいた方がよいリテラルについて話をしていきましょう。

3-1-1 リテラルとは何かを知ろう

Chapter 2の復習がてら、まず、ひとつコーディングしてみましょう。

Chapterが変わったので、このChapter用のフォルダーを作成しましょう。2-2-3項を参考に、ITBasicTypeScriptフォルダー内にchap03フォルダーを作成してください。さらに、2-2-4項を参考に、その中にshowLiterals.tsファイルを作成しましょう（図3-1）。

図3-1 新しいフォルダーとファイルが作成されたVS Codeの画面

作成したshowLiterals.tsにリスト3-1のコードを記述し、2-3-2項と2-3-3項を参考に、コンパイル、実行しましょう。なお、以降は、特に記述がない限り、コンパイル、実行まで行ってください。

リスト3-1 chap03/showLiterals.ts

```
001  export{} ❶
002
003  console.log("123");
004  console.log(456);
```

4-2 if構文の続きを知る

前節で紹介した条件分岐は不完全です。条件に合致しない場合、何も処理が行えません。そこを次に学びましょう。

4-2-1 条件に合致しない場合の処理を記述してみよう

リスト4-3に、条件に合致しない場合の処理を追加しましょう。図にすると、図4-9のようになります。

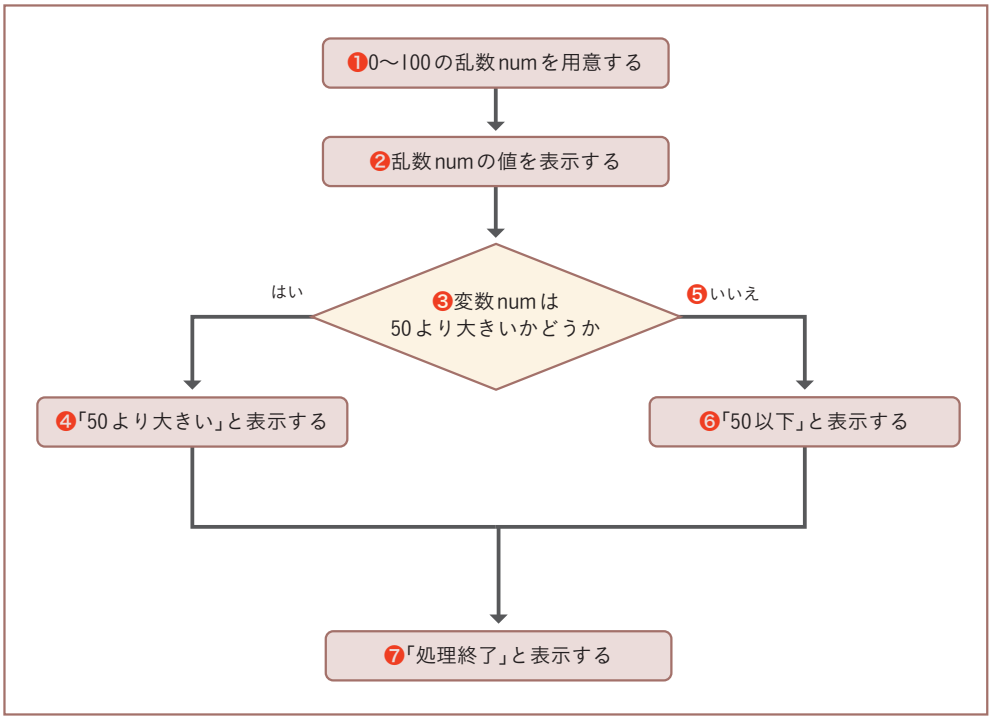


図4-9 条件に合致しない場合が追加された処理の流れ

このような処理を記述したリスト4-4のifElseStatement.tsファイルを作成してください。リスト4-4中の番号は、それぞれ、図4-9中の番号に対応しています。

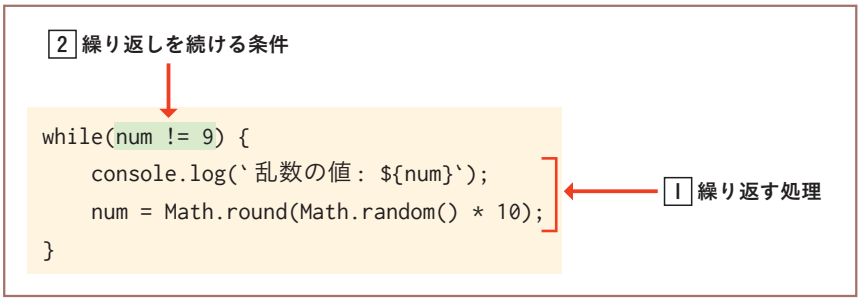


図 5-3 リスト 5-1 中のループ構文

先の条件がtrueである限り、この{ }ブロック内の処理が繰り返し実行されます。ここでも、条件分岐と同様に、{ }ブロックが処理の範囲を表す役割をしています。

5-1-3 無限ループには注意しよう

ここで、リスト 5-1 の❺の役割に注目してみます。これは、図 5-2 からも明らかで、新たな乱数を発生させて、それを num に代入しています。もし、この行を記述し忘れたとしたら、どうなるかを考えてみます。すると、num は❷で発生した乱数の値のまま、変更がありません。

もし❷で乱数として 9 が発生した場合、❸の( )内の条件は false となり、続く{ }ブロック内の処理は全く実行されません。これは、❺の有無にかかわらず、そのようになります（図 5-4）。

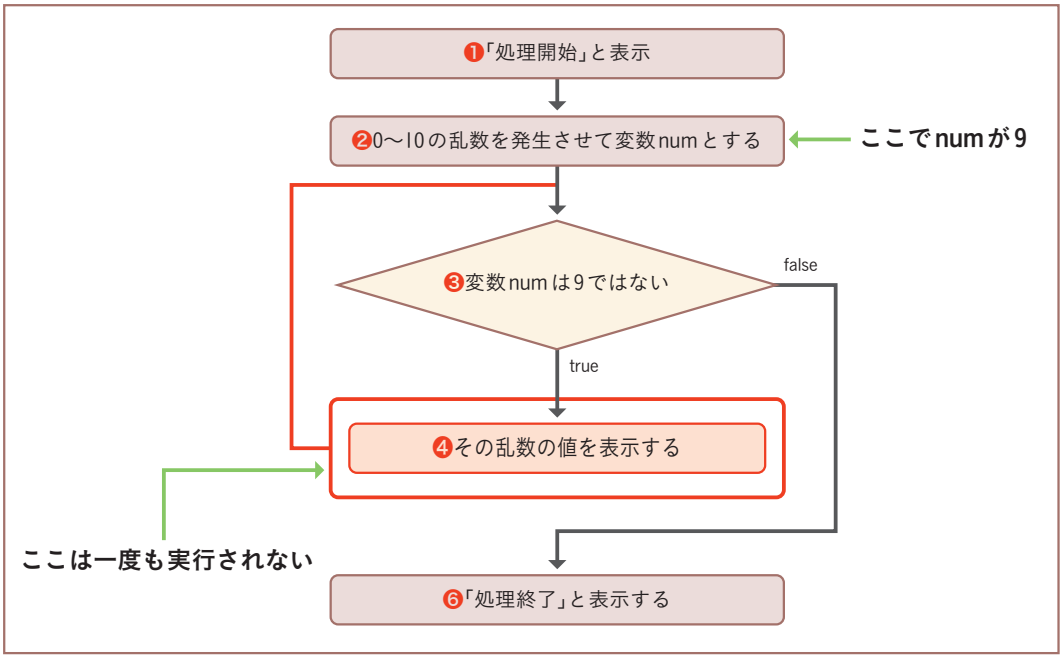


図 5-4 num の初期値が 9 の場合

一方、9 以外の値が発生した場合、❸の( )内の条件は true です。続く{ }ブロック内の処理は実行されますが、もう一度❸に処理が戻ってきたときに、num は依然 9 以外のままです（図 5-5）。

という値があります。このように、既に存在するインデックスに値を代入すると、値が上書きされます。もっとも、この仕組みは通常の変数と同様ですので、特に疑問点はないでしょう。

問題は、その次の❹の処理です。❹では、存在しないインデックス5に値を代入しています。その場合、配列の末尾に、6個目の値を追加していることになります。このように、新たな要素への代入が行われた時点で、自動的に要素が追加される仕組みとなっているのです。結果、❺の表示内容からわかるように、問題なく追加されています（図6-8）。

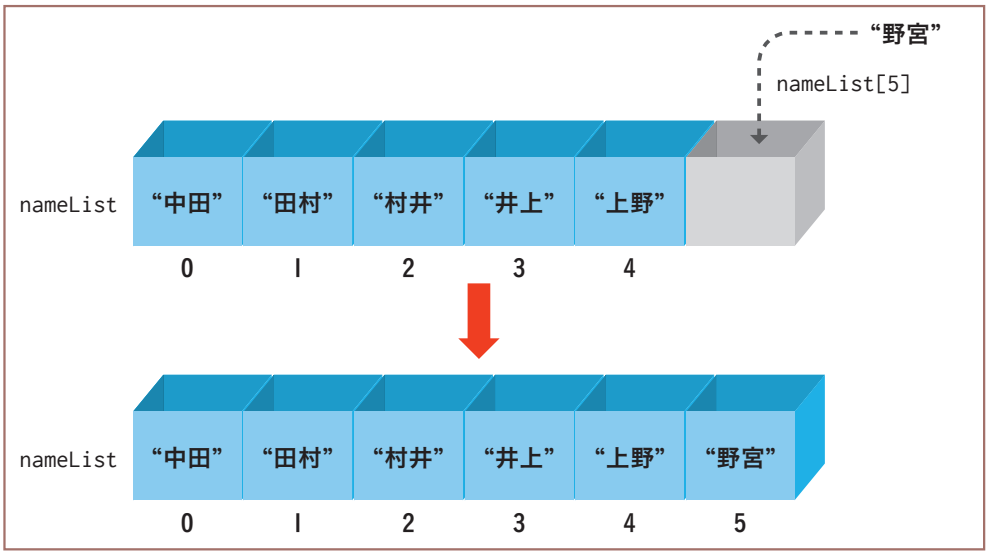


図6-8 存在しないインデックスへの代入は要素の追加

では、存在しない要素に対して、代入される前、つまり、追加されていない状態で値の呼び出しを行うとどうなるかというのが、リスト6-6の❻です（図6-9）。

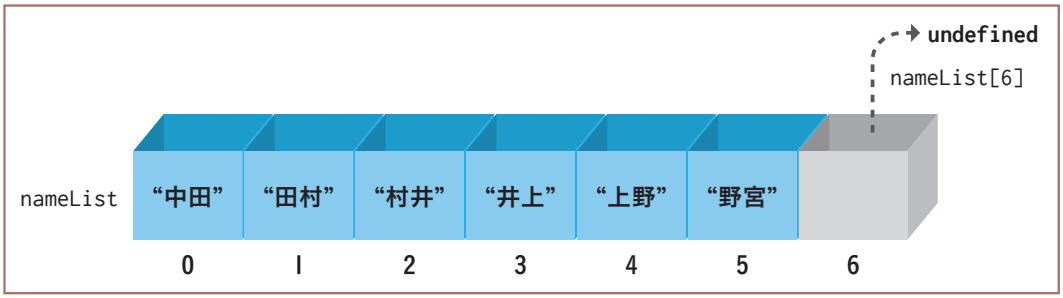


図6-9 値が存在しないインデックスの指定

これは、実行結果からわかるように、**undefined**となります。undefinedというのは、未定義を表す特殊な値です。

最後に、リスト6-6は、❼でインデックス8への値の代入を行っています。一方、❻で確認したように、インデックス、6、7は値が格納されていません。つまり、インデックスが飛んでいるのです。この状態で、配列の内容を表示したのが❽です。実行結果では、「2 empty items」と記述されているように、インデックス6と7がempty、つまり未定義として扱われる一方で、

リスト7-2では、このように定義された関数を❸～❺で利用しています。それぞれ、引数として100、150、200の数値リテラルを渡していますので、それぞれの値までの足し算結果が戻り値として戻ってきます。せっかく関数が戻り値を返してくれるので、関数の呼び出し側ではそれを受け取って利用する必要がありますが、その受け取った値の利用方法は関数の呼び出し元の自由です。リスト7-2では、それぞれ変数sigma100、sigma150、sigma200に格納した上で、さらに、これらの変数を足し合わせた合計値totalを算出し、最終的にこのtotalを表示させています。

note

7-1-2項で紹介した戻り値のない関数の場合、関数名()の次に戻り値の型を記述していません。これをあえて記述する場合、次のように、戻り値なしのデータ型を表すvoidとします。

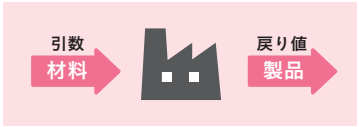
```
function showSigma2N(n: number): void {
    :
}
```


7-1-6 引数と戻り値の組み合わせパターンは4種類ある

ここまで、関数の基本形と引数、戻り値を紹介してきました。その引数は必要な場合と不要な場合、戻り値も必要な場合と不要な場合があります。それらをまとめると、関数の作り方としては、図7-6の4パターンあることになります。

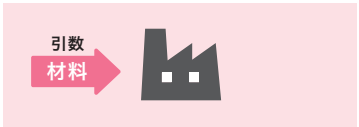
1 引数有・戻り値有

```
function 関数名(引数名: 引数の型, …): 戻り値の型 {
    :
    return 戻り値;
}
```



2 引数有・戻り値無

```
function 関数名(引数名: 引数の型, …) {
    :
}
```



3 引数無・戻り値有

```
function 関数名(): 戻り値の型 {
    :
    return 戻り値;
}
```

4 引数無・戻り値無

```
function 関数名() {
    :
}
```

図7-6 引数と戻り値の組み合わせパターン

8-4-3 関数が引数の関数を作ってみる

ここで、リスト8-8の❷と❸を、さらにひとつの関数として定義し、funcを引数として受け取ることを考えます。この関数名をshowConcatNameとすると、リスト8-9のコードになります。

リスト8-9 chap08/execFuncExpression2.ts

```
001  export{}
002
003  function showConcatName(f) {  ❶
004      const result = f("田中", "太郎");  ❷
005      console.log(result);
006  }
007
008  const func =  function(lastName: string, firstName: string): string {
009      return `${lastName} ${firstName}`;
010  }
011  showConcatName(func);  ❸
```

リスト8-9の❶の関数showConcatName()の引数fがまさにコールバック関数であり、この引数fが関数そのものとなります。リスト8-8の❷と同様に、関数内では❷のように、引数名fに( )を記述してその関数を実行できます。

実行部分では、❸のように、このshowConcatName()関数を呼び出す時に、コールバック関数そのものを格納したfuncを渡しています（図8-6）。

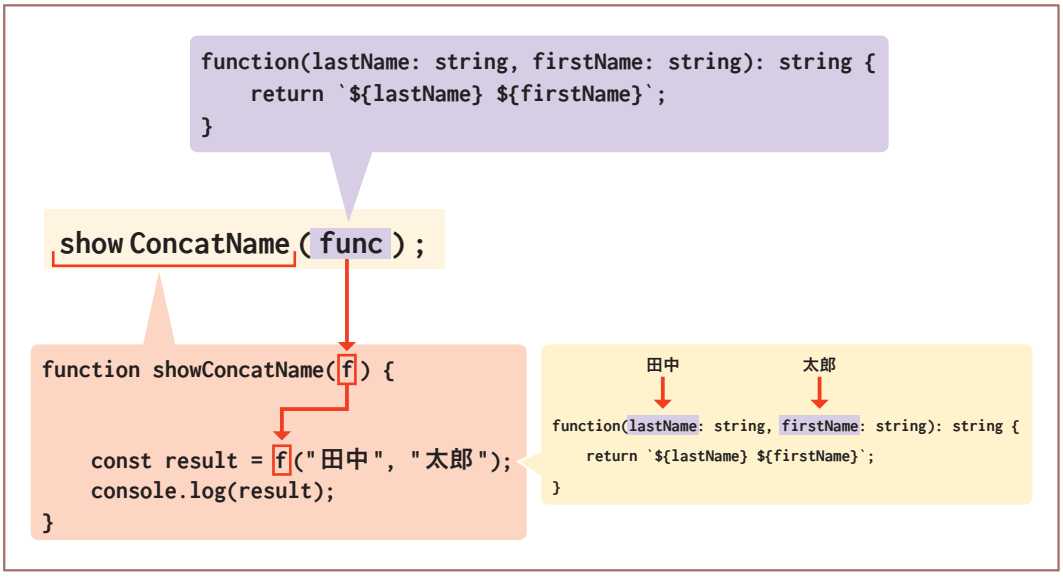


図8-6 リスト8-9のコードの仕組み

❸のコードは、リスト8-5のforEach()に、funcを渡しているのと全く同じ構造です。

```
const jiro =
{
    name: "小林次郎",
    english: 83,
    :
    history: 68
}
```

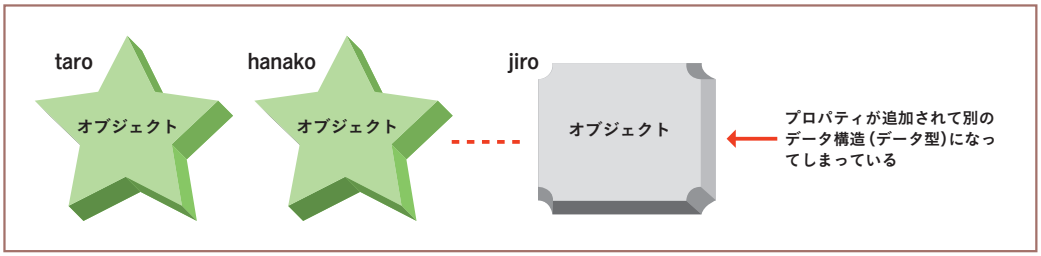


図9-1 オブジェクトリテラルではデータ型を同一にできない

全てのオブジェクトのデータ構造（データ型）を同一にするためには、その雛形が必要になります。これが、**クラス**です（図9-2）。

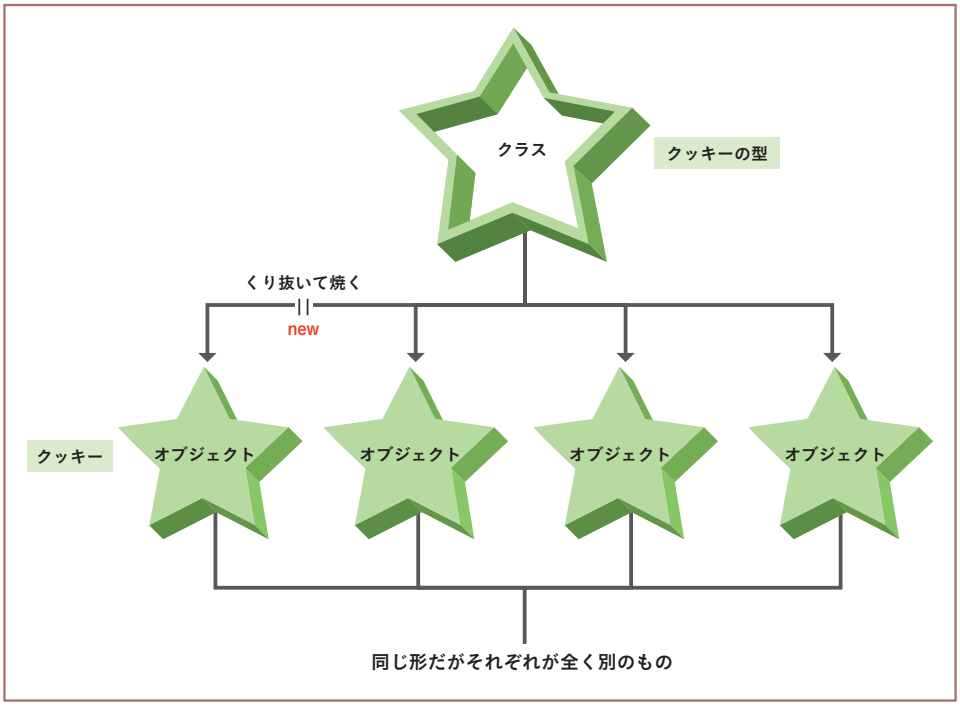


図9-2 クラスはオブジェクトの雛形

イメージとしては、**クラス**はクッキーの型であり、そのクッキーの型によってくり抜かれて焼かれたクッキーが**オブジェクト**です。クッキーは、全て同じ形をしているにもかかわらず、各々が全く別のものとして扱われます。

10-3 タプルと Enum を知る

さて、ここで少し趣旨を変えてみます。クラスやインターフェースを学んできた今だからこそ、学ぶ価値があるデータ型を２個紹介します。それは、タプルと Enum です。

10-3-1 データ型の違う値をまとめられるタプル

これまでも扱ってきたように、例えば、ある人の名前と身長と体重のように、違うデータ型の値をまとめてワンセットとして扱いたい場面はよくあります。そのために、クラスとそのクラスを new して利用できるオブジェクト、あるいは、インターフェースとオブジェクトリテラルのような方法が用意されており、それらを学んできました。

一方、もっと手軽に、使い捨て感覚でそのようなデータ型の違う値をワンセットとして扱いたい場面では、**タプル**が便利です。タプルというのは、データ型と個数があらかじめ決まっている配列のようなものです。実際にコードを見ていきましょう。リスト 10-8 の useTuple.ts を作成してください。

リスト10-8 chap10/useTuple.ts

```
001  export{}
002
003  const personalData: [string, number, number] = ["太郎", 167.5, 65.1];  ❶
004  for(const element of personalData) {
005      console.log(element);                                               ❷
006  }
```

実行結果

```
> tsc useTuple.ts
> node useTuple.js

太郎
167.5
65.1
```

リスト 10-8 の❶がタプルです。ほぼ配列と似た書式ですが、違いは、データ型の記述部分です。次のように３個のデータ型を記述しています。

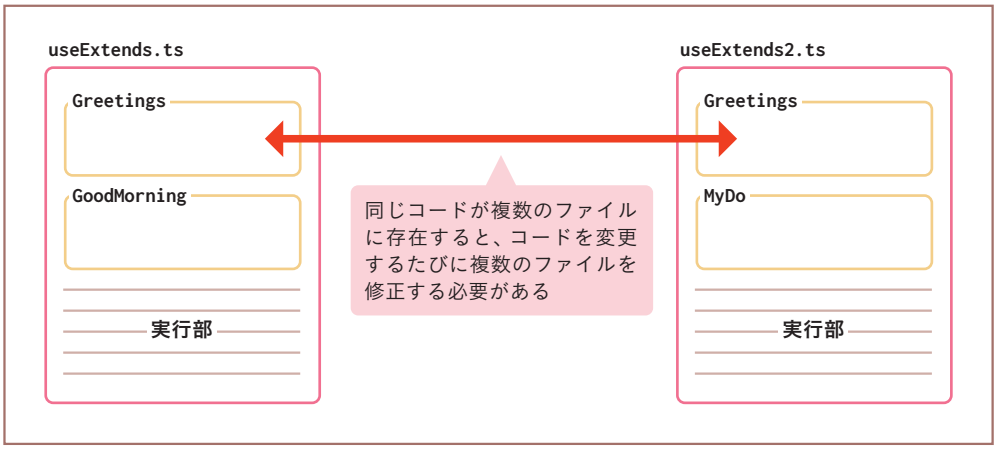


図11-2 同じコードが複数ファイルに存在するのは問題

やはり、ひとつのクラス定義は、ひとつのファイルにのみ記述し、他のクラスや実行部では、そのファイルを読み込んで利用する、つまりは再利用できるようにしたほうが、コーディング効率の面でも、メンテナンスの面でもメリットが大きいです。

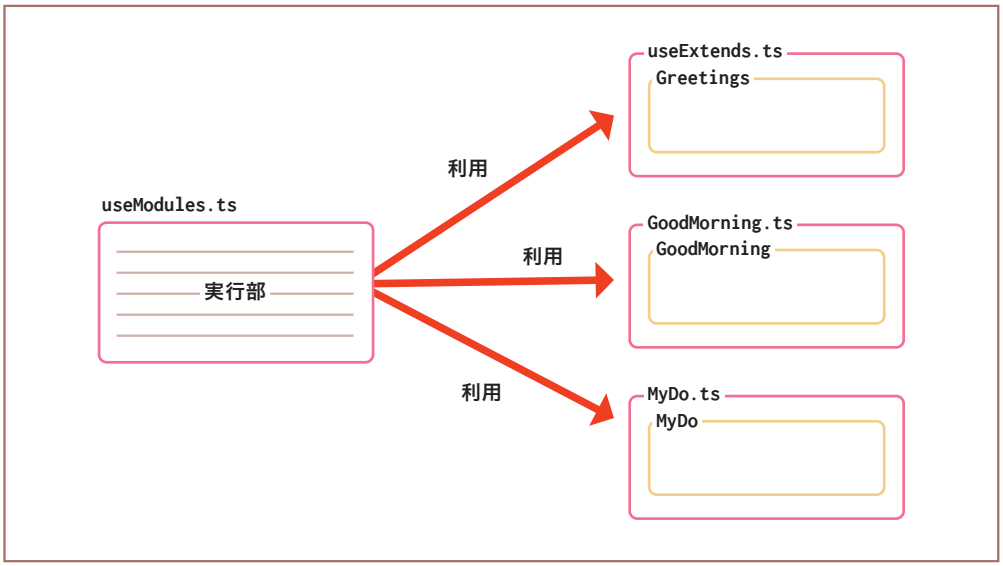


図11-3 ひとつのコードはひとつのファイルにのみ記述するべき

この図11-3のように、別のファイルに記述されたクラスなどを読み込んで使えるようにした仕組みが、**モジュール**です。

12-3 非同期処理とWebアクセスを知る

かなりの部分が完成してきましたが、まだ、main.tsにエラーがあります。一番肝心のweatherinfo-receiver.tsを作成していないからです。最後に、このweatherinfo-receiver.tsを作成し、それに合わせて、main.tsの続きの部分を作成し、完成させることにします。

12-3-1 非同期処理が何かを理解しよう

早速、weatherinfo-receiver.tsのコーディングを行いたいところですが、その前に理解しておかないといけないことがあります。それは、**非同期処理**です。これまでのコーディングでの処理は、全て、**同期処理**です。同期と非同期、どういう違いがあるのでしょうか。図12-10を見てください。

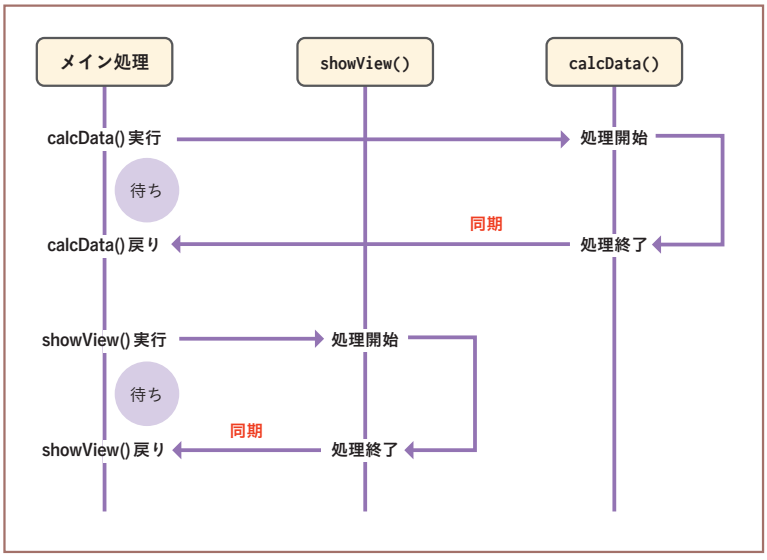


図12-10 同期処理の流れ

例えば、メイン処理があるとします。今回作成しているアプリケーションでは、main.tsが該当します。そこから、例えば、calcData()関数を実行したとします。すると、処理は、この関数に移ります。その間、実は、メイン処理は何も処理を実行せずに、ずっと待ったままとなるのです。calcData()関数の処理が終了し、メイン処理に処理が戻ってきて初めて、続きの処理が行われます。showView()についても同様です。