TypeScript

コードレシピ集

鹿野壮 著

技術評論社

サンプルファイルは、以下の環境で動作確認をしています。

・TypeScript 6
・TypeScript 7 beta
・Node.js 24
・2026年4月現在の最新版のブラウザ
（Apple Safari、Google Chrome、Mozilla Firefox）

注 意

ご購入・ご利用の前に必ずお読みください

本書に記載された内容は、情報の提供のみを目的としています。したがって、本書を用いた運用は、必ずお客様自身の責任と判断によっておこなってください。これらの情報の運用の結果について、技術評論社および著者はいかなる責任も負いません。

本書記載の情報は、2026年4月現在のものを掲載していますので、ご利用時には、変更されている場合もあります。また、ソフトウェアに関する記述は、特に断わりのないかぎり、2026年4月現在での最新バージョンをもとにしています。ソフトウェアはバージョンアップされる場合があり、本書での説明とは機能内容や画面図などが異なってしまうこともありえます。

以上の注意事項をご承諾いただいた上で、本書をご利用願います。これらの注意事項をお読みいただかずに、お問い合わせいただいても、技術評論社および著者は対処しかねます。あらかじめ、ご承知おきください。

はじめに

TypeScriptはいまや多くの開発現場で標準的に使われるプログラミング言語です。ReactやNode.jsをはじめ、多くのフレームワークやライブラリがTypeScript前提で開発されています。

TypeScriptは2012年にMicrosoftから公開されました。JavaScriptに静的型付けを加えることで、大規模な開発でもミスを未然に防げるよう設計された言語です。2026年に登場したTypeScript 7では、長年TypeScriptで実装されてきたコンパイラーがGo言語へ移植され、コンパイル速度は従来のおよそ10倍となりました。大規模なプロジェクトでも待ち時間を気にせず型チェックができます。

本書では新世代のTypeScriptに合わせて、satisfies演算子や型述語の推論といった新しいAPIを取り上げました。ジェネリクス、Mapped Types、型ガードなど、長く使える基礎も丁寧に解説しています。JavaScriptの新しい構文や実行環境の進化もあわせて紹介し、明日から現場で使える知識を身につけられる構成です。

各レシピは冒頭の「利用シーン」と「Syntax」で要点をつかめるようにし、続く解説とコード例で使い方を確認できる構成です。独立したTips形式なので、どこから読み始めても理解できます。現在直面している課題に近いレシピから逆引き辞典のように読み進めるのもおすすめです。ブラウザやNode.jsでそのまま動くデモも数多く収録しました。実際に手元で動かし、値を書き換えながら読み進めると理解が深まるでしょう。

生成AIが日常的にコードを書く時代になりましたが、だからこそ言語そのものの基礎を理解しておく意味は大きくなっています。AIが出力したコードを読み解き、誤りに気づき、適切に修正する力は、自分で書いて動かしながら学んだ経験のうえに積み上がるものです。本書を通じて新しいTypeScriptに触れ、手を動かす楽しさを味わってもらえたら何よりです。

最後に、執筆を支えてくれた妻と子、そして愛猫に心から感謝します。

2026年4月　鹿野壮

本書の読み方

❶ 項目名

TypeScriptを使って実現したいテクニックを示しています。

❷ 利用シーン

実現したいテクニックがどのようなシーンで利用できるのかを示しています。

❸ Syntax

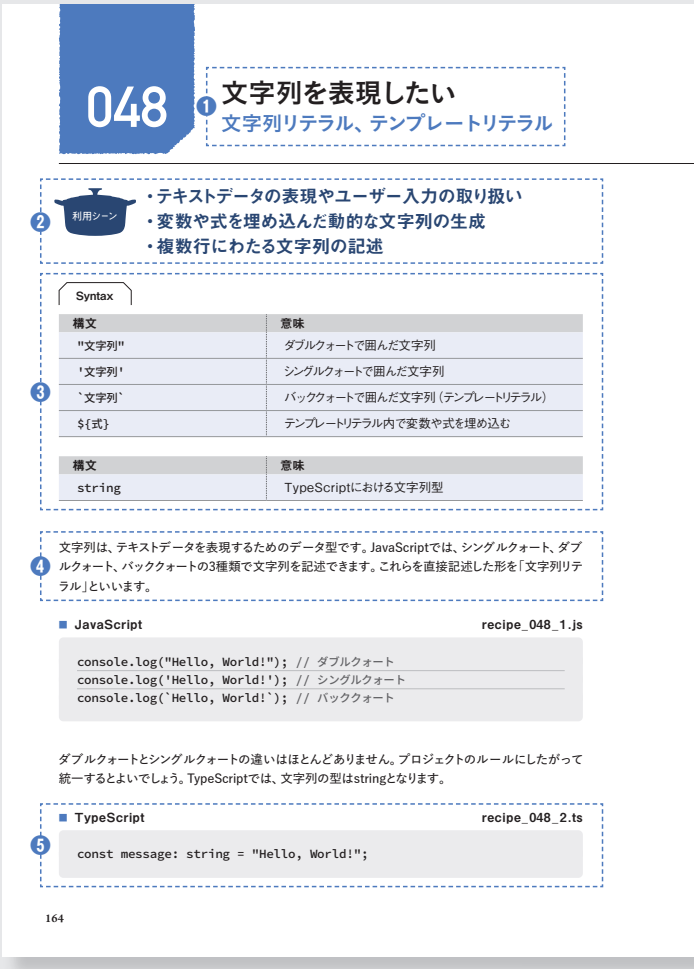
目的のテクニックを実現するために必要なTypeScriptの機能や構文です。

❹ 本文

目的のテクニックを実現するために、どの機能をどのような考えで使用するかなど、方針や具体的な手順を解説しています。

❺ TypeScriptコード

目的のテクニックを構成するTypeScriptコードを示しています。なお、TypeScriptはJavaScriptの拡張プログラミング言語であるため、JavaScriptコードも多数掲載しています。



048 ❶ 文字列を表現したい
文字列リテラル、テンプレートリテラル

❷ 利用シーン
・テキストデータの表現やユーザー入力の取り扱い
・変数や式を埋め込んだ動的な文字列の生成
・複数行にわたる文字列の記述

❸ Syntax

構文	意味
"文字列"	ダブルクォートで囲んだ文字列
'文字列'	シングルクォートで囲んだ文字列
`文字列`	バッククォートで囲んだ文字列（テンプレートリテラル）
${式}	テンプレートリテラル内で変数や式を埋め込む

構文	意味
string	TypeScriptにおける文字列型

❹ 文字列は、テキストデータを表現するためのデータ型です。JavaScriptでは、シングルクォート、ダブルクォート、バッククォートの3種類で文字列を記述できます。これらを直接記述した形を「文字列リテラル」といいます。

JavaScript recipe_048_1.js

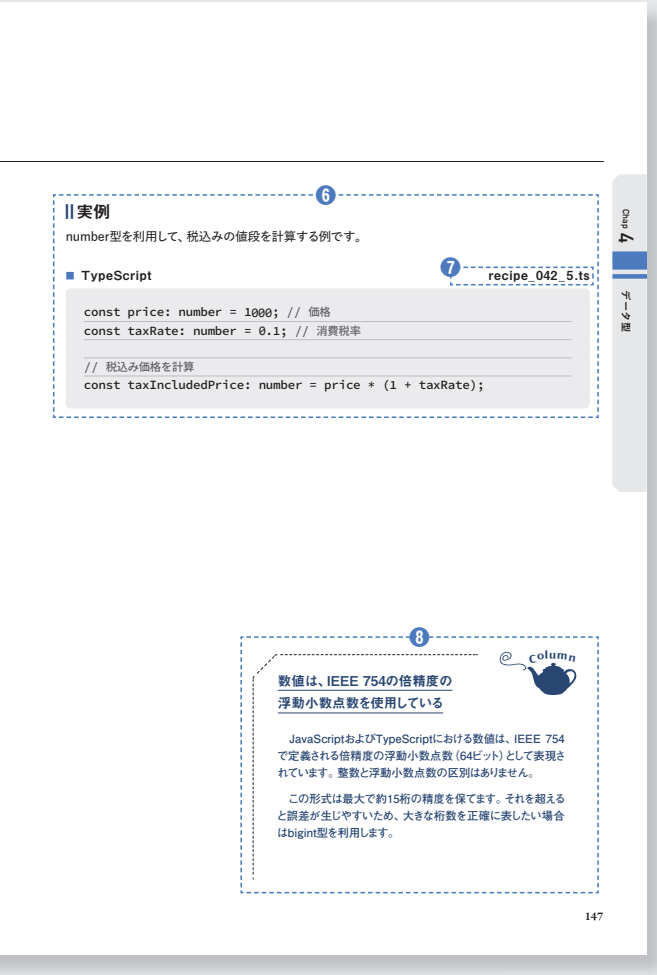
```
console.log("Hello, World!"); // ダブルクォート
console.log('Hello, World!'); // シングルクォート
console.log(`Hello, World!`); // バッククォート
```

ダブルクォートとシングルクォートの違いはほとんどありません。プロジェクトのルールにしたがって統一するとよいでしょう。TypeScriptでは、文字列の型はstringとなります。

❺ TypeScript recipe_048_2.ts

```
const message: string = "Hello, World!";
```

164



❻ 実例

目的のテクニックの実例として、実際の開発現場で参考になるコード例を紹介しています。

❼ ファイル名

サンプルファイルとして提供しているコードのファイル名を示しています。

❽ コラム

テクニックに関連する補足情報です。

サンプルファイルについて

本書掲載の多くのテクニックは、サンプルファイルを用意しています。
以下の技術評論社 Web サイトからダウンロード方法を確認してください。

URL https://gihyo.jp/book/2026/978-4-297-15628-2/support

CONTENTS

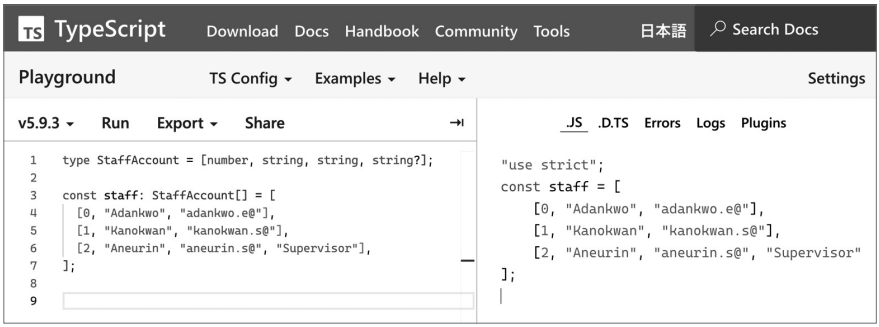
JavaScriptとTypeScriptの基礎

Chapter 1

004 TypeScriptをオンラインで手軽に試したい

TypeScriptを使う際、通常であればコマンドラインを使って環境構築が必要です。しかし、面倒な準備をせずとも手軽にオンラインでTypeScriptを試せる「TS Playground」があります。
TS Playgroundは、Microsoft公式のオンラインツールです。ブラウザだけでTypeScriptのコードを書いて試せるので、TypeScriptの学習やコードの実験に向いています。次のURLからアクセスできます。

https://www.typescriptlang.org/play/

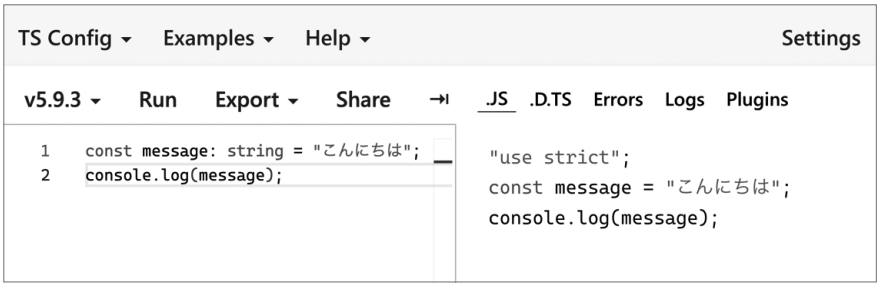


TS Playgroundの主要な機能

TS Playgroundには多くの機能がありますが、その中で主要な機能をいくつか紹介します。

コンパイル結果の確認

左側のエディターにTypeScriptのコードを入力すると、右側のJSタブにコンパイルされた結果がリアルタイムで表示されます。変換の違いを確認しながら学習できます。



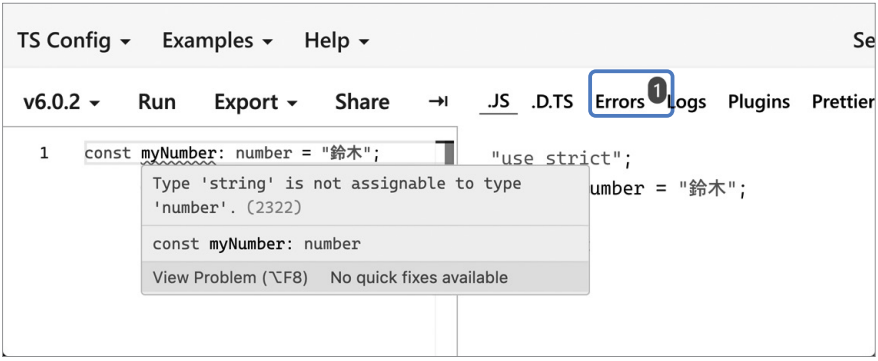
エラーの確認

TypeScriptのコードにエラーがある場合、エディターがその位置をハイライトし解決策のヒントを示します。入力中に気づけるので、実行前に修正できます。
次のコードを記述してみましょう。number型の定数なのに文字列を代入しているので、エラーが発生します。

TypeScript recipe_004_1.ts

```
const myNumber: number = "鈴木";
```

エディターがmyNumber定数のところを赤く波線でハイライトします。そこにマウスを乗せると、詳細なエラーメッセージを表示します。

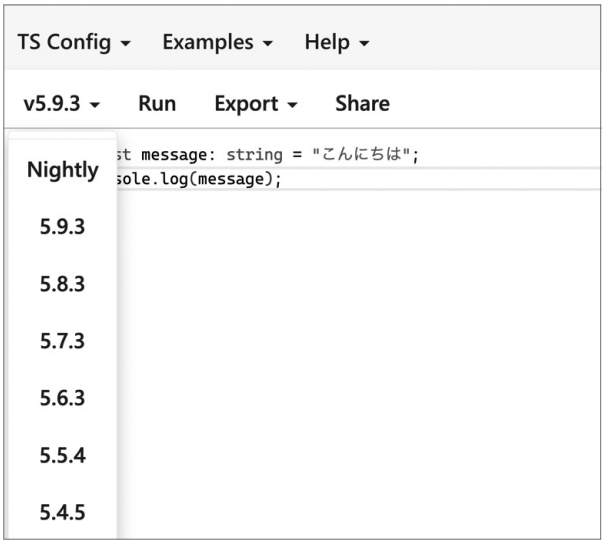


コードの実行

画面上部の「Run」ボタンをクリックすると、TypeScriptがJavaScriptへコンパイルされたうえで実行され、コンソール出力が右側に表示されます。
次の例では、console.logを使ってメッセージの出力をしています。

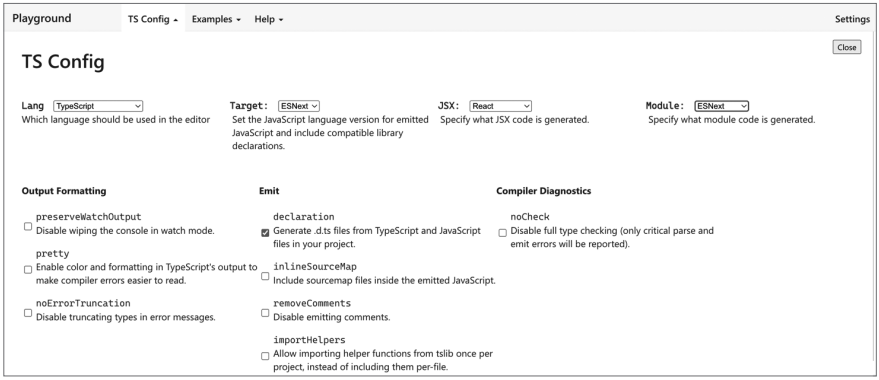
TypeScriptのバージョンの変更

画面上部のツールバーにあるバージョン番号のプルダウンより、TypeScriptのバージョンを変更できます。現行の最新バージョンはもちろん、過去のバージョンから開発中の最新バージョンまで試せます。プロジェクトで使っているTypeScriptの挙動を確かめたり、新しく追加された機能を試す際に便利です。



TS Configの設定変更

TypeScriptの設定は、プロジェクトに配置したtsconfig.jsonファイルで制御されます。TS Playgroundでは、画面上部にある「TS Config」ボタンからTarget（コンパイル対象のJavaScriptバージョン）やstrictモードを切り替えられます。設定は共有URLにも含まれるため、同じ条件でサンプルを再現してもらうときにも便利です。



型情報をその場で確認できる

エディター内で変数名や関数名にマウスカーソルを合わせる（またはキャレットを置く）と、推論された型情報がツールチップに表示されます。補完候補とあわせて型を確認しながら編集できるため、型定義を行き来せずに学習を進められます。
次の例では、名前と得点を持つシンプルなオブジェクトを定義し、プロパティを別の定数に取り出しています。profileやuserScoreにマウスをホバーさせると、それぞれ推論された型がツールチップで確認できます。

TypeScript recipe_004_2.ts

```ts
const profile = {
  name: "鈴木",
  score: 92,
};

const userScore = profile.score;
```

次の例では、userScoreにマウスをホバーしています。「const userScore: number」と表示され、userScoreがnumber（数値型）であることがわかります。

型推論の基本的な考え方はレシピ027「型が何かを推論させたい」で解説しています。

「^?」記法で型情報を表示する（Twoslash Queries）

Playgroundでは、「^?」を含むコメントを置くと、その位置にある識別子の型がエディター上にリアルタイムで表示されます。コードの型情報を表示できるので、型の確認時や学習時に便利です。筆者のお気に入りの機能の1つです。
次の例では、userScoreの下の位置に書いた^?が、推論された型（この例ではnumber）をインラインで表示します。

■ TypeScript　recipe_004_3.ts

```ts
const profile = {
  name: "鈴木",
  score: 92,
};

const userScore = profile.score;
//    ^?
```

なお、正しく「userScore」の文字列の下に^?を置かないと、型情報が表示されません。

基本構文

Chapter 2

023 null/undefinedになり得る値に安全にアクセスしたい

?.

- 深いネストのプロパティに安全にアクセスする
- nullやundefinedかもしれない配列や関数を安全に操作する
- 予想外の形式のデータでもエラーなく処理を続ける

Syntax

構文	意味
値?.プロパティ	プロパティに安全にアクセスする
配列?.[インデックス]	配列の要素に安全にアクセスする
関数?.(引数)	関数を安全に呼び出す

※ いずれも左辺がnullまたはundefinedの場合はundefinedを返す。

?.（オプショナルチェーン）を使うと、左辺がnullやundefinedでもエラーにならず、undefinedが返されます。

■ TypeScript　recipe_023_1.ts

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

const user1: User = { name: "鈴木", details: { age: 25 } };
console.log(user1.details?.age); // 25

const user2: User = { name: "田中" };
console.log(user2.details?.age); // undefined
```

?.を使わない場合との比較

?.を使わない場合、&&演算子で各階層をチェックする必要があり冗長です。

TypeScript recipe_023_2.ts

```
// ?.を使わない場合
const age = user && user.details && user.details.age;
```

TypeScript recipe_023_3.ts

```
// ?.を使う場合
const age = user?.details?.age;
```

?.は?.で1セット

?.は?.で1セットです。?と.の間にスペースや改行を入れると構文エラーになります。

TypeScript recipe_023_4.ts

```
// 構文エラー: ?と.の間にスペースや改行は入れられない
user.details? .age;
```

?.の連続使用

?.は連続して使用できます。深くネストしたオブジェクト構造でも安全にアクセスできます。

TypeScript recipe_023_6.ts

```
type User = {
  name: string;
  profile?: { settings?: { theme: string } };
};

const user1: User = {
  name: "鈴木",
  profile: { settings: { theme: "ダーク" } },
};
console.log(user1.profile?.settings?.theme); // "ダーク"

const user2: User = { name: "田中" };
console.log(user2.profile?.settings?.theme); // undefined
```

配列のオプショナルチェーン

配列?.[インデックス]と書くと、配列が存在する場合だけ指定したインデックスの要素にアクセスします。

■ TypeScript　recipe_023_7.ts

```
function getFirstFruit(fruits: string[] | undefined) {
  return fruits?.[0];
}

console.log(getFirstFruit(["りんご", "バナナ"])); // "りんご"
console.log(getFirstFruit(undefined)); // undefined
```

配列の要素がオブジェクトの場合、要素?.でプロパティにアクセスできます。配列自体がundefinedの可能性もあるなら、配列?.[インデックス]?.プロパティと組み合わせます。

■ TypeScript　recipe_023_8.ts

```
type User = { name?: string };

function getUserName(users: User[] | undefined) {
  return users?.[0]?.name;
}

console.log(getUserName([{ name: "鈴木" }, { name: "田中" }]));
// "鈴木"
console.log(getUserName(undefined)); // undefined
```

関数のオプショナルチェーン

関数?.(引数)と書くと、関数が定義されている場合のみ実行し、未定義であればundefinedを返します。

■ TypeScript　recipe_023_9.ts

```
function callGreet(greet: (() => string) | undefined) {
  return greet?.();
}

console.log(callGreet(() => "こんにちは")); // "こんにちは"
console.log(callGreet(undefined)); // undefined
```

024 trueやfalseとみなされる値と、&&、||、??演算子の違いについて知りたい

- truthy/falsyの判定ルールを理解する
- &&、||、??の違いを把握して使い分ける
- 値に応じたデフォルト値の設定や条件処理を行う

Syntax

構文	意味
`式1 && 式2`	式1がtruthyなら式2を返す※1
`式1 || 式2`	式1がfalsyなら式2を返す※2
`式1 ?? 式2`	式1がnullやundefinedなら式2を返す※3

※1 式1がtruthyであれば式2の値を返し、そうでなければ式1の値を返す。
※2 式1がfalsyであれば式2の値を返し、そうでなければ式1の値を返す。
※3 式1がnullまたはundefinedであれば式2の値を返し、そうでなければ式1の値を返す。

JavaScriptでは、真偽値（true・false）ではないものの、trueやfalseと「みなされる値」があります。それぞれtruthy（真値）またはfalsy（偽値）と呼びます。この挙動を理解しておかないと、nullやundefined・空文字（""）・0・falseなどの値を||演算子や&&演算子で使用したときに、予期せぬ挙動が生まれます。truthy・falsyの値の例を示します。

● truthyな値（trueとみなされる値）

値	説明
`true`	真偽値のtrue
`1`	ゼロ以外の数値
`"hello"`	空でない文字列
`"0"`	文字列として表現されたゼロ
`[]`	空の配列
`{}`	空のオブジェクト
`function() {}`	関数

● falsyな値（falseとみなされる値）

値	説明
`false`	真偽値のfalse
`0`	数値のゼロ
`-0`	負のゼロ
`0n`	BigIntのゼロ
`""`	空文字列
`null`	null値
`undefined`	undefined値
`NaN`	Not a Number

&&演算子

&&演算子は、左側がtruthyと評価される場合に、右側の値を返します。次に示すのはJavaScriptのtruthy/falsyの挙動を確認するための例です。TypeScriptでは一部のリテラル式に対してコンパイルエラーが発生します。

■ JavaScript　recipe_024_1.js

```
console.log(true && "値あり"); // '値あり'
console.log(1 && "値あり"); // '値あり'
console.log("hello" && "値あり"); // '値あり'
console.log([] && "値あり"); // '値あり'
```

左側がfalsyの場合は左側の値をそのまま返します。

■ JavaScript　recipe_024_2.js

```
console.log(false && "値あり"); // false
console.log(0 && "値あり"); // 0
console.log("" && "値あり"); // ''
console.log(null && "値あり"); // null
console.log(undefined && "値あり"); // undefined
```

||演算子

||演算子は、左側がfalsyと評価される場合に、右側の値を返します。左側がtruthyと評価される値であれば、そのまま左側の値が返されます。||演算子は、左側がfalsyの場合にのみ右側の値を使用するため、デフォルト値を提供したい場合によく使われます。

TypeScript recipe_024_3.ts

```
console.log(null || "デフォルト値"); // 'デフォルト値'
console.log("実際の値" || "デフォルト値"); // '実際の値'
```

左側がfalsyとして評価される値（0、""、null、undefinedなど）であれば、右側の値がそのまま返されます。

TypeScript recipe_024_4.ts

```
console.log(false || "デフォルト値"); // 'デフォルト値'
console.log(0 || "デフォルト値"); // 'デフォルト値'
console.log("" || "デフォルト値"); // 'デフォルト値'
console.log(null || "デフォルト値"); // 'デフォルト値'
console.log(undefined || "デフォルト値"); // 'デフォルト値'
```

??演算子

??演算子は、左側がnullまたはundefinedの場合のみ、右側の値を返します。他のfalsy値（0、空文字列、falseなど）は有効な値として扱われ、そのまま返されます。

TypeScript recipe_024_5.ts

```
console.log("鈴木" ?? "デフォルト値"); // 鈴木
console.log(null ?? "デフォルト値"); // 'デフォルト値'
console.log(undefined ?? "デフォルト値"); // 'デフォルト値'
```

型システム

Chapter 3

027 型が何かを推論させたい

- 型を明示せずリテラル型へ推論させる
- オブジェクトや配列の型を自動で推論させる
- データ構造の変更に型を自動で追従させる

Syntax

構文	意味
`let myName = '鈴木'`	myNameはstring型に推論される
`const myName = '鈴木'`	myNameは"鈴木"リテラル型に推論される

TypeScriptでは、const myName: string = '鈴木';のように、型名を明示することで、変数の型を指定できる「型注釈」という機能があります。
一方で、TypeScriptでは、型を明示的に指定せずとも、その変数がどの型を持つかをコンパイラーが自動的に判定してくれる機能があります。「型を推論する」（type inference）といいます。
次の例では、myNameに"鈴木"という文字列が代入されています。この場合、型を明示しなくてもmyNameがstring型（文字列型）に「推論」されます。

■ TypeScript recipe_027_1.ts

```
let myName = "鈴木"; // myNameはstring型に推論される
```

数値・真偽値・配列・関数といった、他のデータ型も、同様に推論されます。次のように宣言時の値に応じて型が決まります。

■ TypeScript recipe_027_2.ts

```
let myAge = 24; // myAgeはnumber型に推論される
let isGood = false; // isGoodはboolean型に推論される
let myArray = [1, 2, 3]; // myArrayはnumber型の配列に推論される
```

型が推論された変数に対して、別の値を入れようとすると、コンパイルエラーが発生します。次の例では、myNameはstring型に推論されていますが、後から数値を代入しようとしているので、コンパイルエラーとなります。

■ TypeScript recipe_027_3.ts

```
let myName = "鈴木"; // myNameはstring型に推論される
myName = 24; // コンパイルエラー
```

━ constで宣言した場合のリテラル型推論

constでプリミティブ型の値を変数に代入した場合、より具体的なリテラル型に推論されます。リテラル型とは、「string」型ではなく「"鈴木"」型のように特定の値そのものを型として扱う型です。

■ TypeScript recipe_027_4.ts

```
const myName = "鈴木"; // "鈴木"リテラル型に推論される
const myAge = 24; // 24リテラル型に推論される
const isGood = false; // falseリテラル型に推論される
```

こういった特性上、プリミティブ型をconstで宣言する場合は、型を明示せずリテラル型へ推論させるほうが、型をより狭く表現できるので型安全です。
次のようにstring型などを明示してしまうと、せっかくのリテラル型が失われてしまいます。

■ TypeScript recipe_027_5.ts

```
const myName: string = "鈴木"; // "鈴木"リテラル型ではなくstring型になる
```

リテラル型を明示できますが、二重に"鈴木"などを書くことになり、煩雑です。

■ TypeScript recipe_027_6.ts

```
const myName: "鈴木" = "鈴木"; // "鈴木"リテラル型
```

constで変数を定義する場合は、型を明示せず、リテラル型へ推論させる記法が簡潔かつ型安全といえます。型を書かなくても自動的にリテラル型へ推論されます。

■ TypeScript recipe_027_7.ts

```
const myName = "鈴木"; // "鈴木"リテラル型になる
```

オブジェクトの場合の型推論

オブジェクトの場合は各プロパティがリテラル型に推論されるようなことはありません。
次の例では、userオブジェクトは{ name: string; age: number }型に推論されます。{ name: "鈴木"; age: 30 }型とはなりません。したがって、user.name = "田中";のように、nameプロパティの書き換えができます。

■ TypeScript recipe_027_8.ts

```
// オブジェクトの場合
const user = { name: "鈴木", age: 30 };

user.name = "田中"; // OK：プロパティは変更可能
user.age = 25; // OK：プロパティは変更可能
```

配列の場合も、各要素は同様にプリミティブ型に推論されます。次の例では、myArrayはnumber[]型に推論されます。

■ TypeScript recipe_027_9.ts

```
const myArray = [1, 2, 3];
```

もしオブジェクトのプロパティや配列の要素もリテラル型として推論させたい場合は、as constを使用します。as constを付けるとプロパティがreadonlyかつリテラル型になり、値の更新を防げます。

■ TypeScript recipe_027_10.ts

```
const user = { name: "鈴木", age: 30 } as const;
// { readonly name: "鈴木"; readonly age: 30 } 型に推論される
// user.name = "田中"; // エラー：readonlyなプロパティは書き換えができない
```

関数の戻り値の型推論

関数についても、TypeScriptは自動的に型を推論します。
次の例では、add関数の戻り値はnumber型に推論されます。戻り値の型注釈を省略しても、TypeScriptが計算結果をnumber型だと判断してくれます。

■ TypeScript recipe_027_11.ts

```
function add(a: number, b: number) {
  return a + b;
}
// add関数の戻り値はnumber型に推論される
const result = add(1, 2);
```

オブジェクトを返す関数の場合も、戻り値の型が推論されます。次の例では、user関数の戻り値は{ name: string; age: number }型に推論されます。

■ TypeScript recipe_027_12.ts

```
function user() {
  return { name: "鈴木", age: 30 };
}
// { name: string; age: number } 型に推論される
```

ただし、関数の場合は中身を見ないと戻り値の型がわからないため、戻り値の型注釈は指定することをオススメします。

関数の引数の型は推論されない

関数の引数については、開発者しか情報を知りえないので型推論されません。引数の型を省略した場合はany型になります。tsconfig.jsonのstrictモードが有効な場合、引数の型省略はコンパイルエラーになります。
次の例では、badFunction関数の引数はany型に推論されます。

■ TypeScript recipe_027_13.ts

```
function badFunction(x, y) {
  return x * y;
}
```

030 値そのものしか代入できない「リテラル型」を作りたい

利用シーン

- 特定の文字列のみを許容する型を作る
- 設定値として決まった値だけを受け入れる
- 特定パターンの文字列や数値のみを許容する

Syntax

構文	意味
`type Suzuki = "鈴木"`	文字列リテラル型 …… "鈴木"しか代入できない型を作る
`const name: "鈴木"`	文字列リテラル型 …… "鈴木"しか代入できない定数name
`const price: 1200`	数値リテラル型 …… 1200のみ代入可能
`type 型名 = "red" | "blue"`	"red"または"blue"のどちらかを許容する型を作る

string型の変数を定義する際、「let myName: string」のように宣言すると、myNameには任意の文字列を代入できます。

■ TypeScript recipe_030_1.ts

```
let myName: string;
myName = "鈴木"; // 有効
myName = "マリオ"; // 有効
```

任意の文字列ではなく、特定の文字列のみを許容したい場合は、「リテラル型」を使います。リテラル型は、TypeScriptで特定の値のみを許容する型を作成するための方法です。
たとえば、次のmyName変数には"鈴木"しか代入できません。「マリオ」を代入しようとするとエラーになります。

■ TypeScript recipe_030_2.ts

```
let myName: "鈴木";
myName = "鈴木"; // 有効
myName = "マリオ"; // NG
```

typeキーワードを使って、リテラル型を作ることもできます。

■ TypeScript recipe_030_3.ts

```ts
type UserName = "鈴木";

const userA: UserName = "鈴木"; // 有効
const userB: UserName = "佐藤"; // エラー
```


さまざまなリテラル型

リテラル型は数値や真偽値にも使用できます。次のように特定の値だけを許可するリテラル型を定義できます。

■ TypeScript recipe_030_4.ts

```ts
const myAge: 20 = 20; // 有効
const myFlag: true = true; // 有効
```

ユニオン型とリテラル型を組み合わせる

リテラル型は、ユニオン型（|）と組み合わせて使うケースも多いです。ユニオン型と組み合わせることで、許容する値を限定し、予期しない値の代入や使用を防げます。たとえば次の例では、"鈴木"または"田中"のどちらかを許容する型を作り、その型で変数を宣言しています。ユニオン型にすることで複数の候補からの選択肢を明示できます。

■ TypeScript recipe_030_5.ts

```ts
type UserName = "鈴木" | "田中";

const userA: UserName = "鈴木"; // 有効
const userB: UserName = "田中"; // 有効
const userC: UserName = "佐藤"; // エラー
```

実例

ライトテーマとダークテーマのどちらかを許容する型を作る例です。ライトを表す"light"とダークを表す"dark"のどちらかを許容する型を定義し、その型で変数を宣言しています。Theme型により想定外のテーマ名が渡らなくなり、分岐も明確になります。

■ TypeScript recipe_030_6.ts

```ts
type Theme = "light" | "dark";

function applyTheme(theme: Theme) {
  if (theme === "light") {
    // 明るいテーマを適用
  } else {
    // 暗いテーマを適用
  }
}
```

データ型

Chapter 4

058 絵文字を崩さずに文字数を数えたい

- 複合絵文字を1文字として正しく数える
- 見た目どおりの文字数でカーソル移動を制御する
- UTF-16コード単位と表示文字数のズレを防ぐ

Syntax

構文	意味
`new Intl.Segmenter(locale?, { granularity })`	セグメンターを生成する
`segmenter.segment(text)`	指定テキストを区切り情報付きイテレーターに変換する

文字数カウントの問題点

テキスト入力で絵文字を扱うと、見た目は1文字でもtext.lengthが2以上になり、カーソル移動や削除がずれます。lengthはUTF-16のコード単位を返すため、絵文字の多くが複数単位で構成されているからです。サロゲートペアは1文字を2つの16ビット値で表す仕組みです。コードポイントはUnicodeが文字に付ける番号で、組み合わせることで家族絵文字などの複合記号が作られます。
次の例では、見た目が2文字の絵文字列でもlengthは4になっています。

TypeScript　recipe_058_1.ts

```
// 😀（U+1F600）👍（U+1F44D）
const text = "😀👍";
console.log(text.length); // 4（UTF-16コード単位の個数）
console.log([...text].length); // 2（見た目どおりの文字数）
```

家族の絵文字にはゼロ幅接合子という結合用の制御文字が含まれ、lengthと見た目の差がさらに広がります。

■ TypeScript　recipe_058_2.ts

```
// 👨‍👩‍👧 (U+1F468 U+200D U+1F469 U+200D U+1F467 U+200D U+1F466)
const family = "👨‍👩‍👧";

console.log(family.length); //  11（UTF-16コード単位の個数）
console.log([...family].length); // 7（見た目は1文字だが、コードポイントは7個）
```

「見た目どおりの1文字」を扱いたい場合につかうのが、Intl.Segmenterです。「見た目どおりの1文字」のことを「書記素」あるいは「grapheme」とよびます。

Intl.Segmenter

Intl.Segmenterは、国際化API（Intl）に含まれる文字列分割専用のクラスです。ロケール（言語設定）とgranularity（グラニュラリティ、区切りの粒度）を渡してセグメンター（セグメント情報を提供するオブジェクト）を作り、segmentメソッドで文字列を書記素や単語に切り分けます。主な使い方は次のとおりです。

- **new Intl.Segmenter(locale, { granularity })** …… ロケール（例："ja", "en"）とgranularity（区切り単位）を渡してセグメンターを生成する
- **segmenter.segment(text)** …… 文字列をセグメント情報付きのイテレーターに変換する
- **各セグメントのsegmentに部分文字列、indexに元の位置が格納される。granularityが"word"の場合はisWordLikeに単語かどうかも含まれる**

granularityには次の選択肢があります。

- **"grapheme"** …… 見た目の1文字ごとに分割。絵文字や結合文字も1単位として扱いたいときに使う
- **"word"** …… 単語ごとに分割。ダブルクリックで単語選択する機能や、単語数カウントに使える
- **"sentence"** …… 文ごとに分割。文章編集や校正ツールで利用できる

Intl.Segmenterではgranularityオプションで区切り単位を指定します。"grapheme"は見た目の1文字、"word"は単語、"sentence"は文単位です。grapheme（グラフェム）は人が1文字として認識する最小単位、セグメンテーションは文字列を意味のあるかたまりへ切り分ける処理のことです。

Intl.Segmenterで文字数を数える

Intl.Segmenterを使うと、絵文字を含めた文字数を正しくカウントできます。関数化して使いまわしておくと便利です。
Intl.Segmenterが使えるかを先に確認し、書記素（見た目の1文字）を配列化してから件数を数える関数を用意します。granularityに"grapheme"を指定すると、複数のコードポイントで構成された絵文字も1件として扱えます。segmenter.segment(text)が返すイテレーターをスプレッド構文で展開すると各書記素の情報が得られ、mapでsegment.segment（実際の文字列部分）だけを抜き出すことで、見た目どおりの文字が並んだ配列を作れます。

■ TypeScript　recipe_058_3.ts

```
function countGraphemes(text: string): number {
  if (
    !("Intl" in globalThis)
    || typeof Intl.Segmenter !== "function"
  ) {
    throw new Error("Intl.Segmenterに未対応の環境です");
  }

  const segmenter = new Intl.Segmenter("ja",
    { granularity: "grapheme" });
  const graphemes = [...segmenter.segment(text)].map(
    (segment) => segment.segment,
  );
  return graphemes.length;
}

console.log(countGraphemes("😀👍🏽")); // 2
console.log(countGraphemes("👨‍👩‍👧‍👦")); // 1
console.log(countGraphemes("👨‍👩‍👧‍👦家族")); // 3
```

書記素の配列を得られるため、入力上限の検証や残り文字数の表示にもそのまま流用できます。

バックスペースで絵文字を安全に削除する

テキストエディタで1文字だけ消す処理を実装するとき、従来のtext.slice(0, -1)だと複合絵文字を途中で切ってしまいます。Intl.Segmenterで書記素の配列を作ってから末尾を取り除けば、見た目どおりに1文字だけ削除できます。

■ TypeScript　recipe_058_4.ts

```
function deleteLastGrapheme(text: string): string {
  const segmenter = new Intl.Segmenter("ja",
    { granularity: "grapheme" });
  const graphemes = [...segmenter.segment(text)].map(
    (segment) => segment.segment,
  );
  graphemes.pop();
  return graphemes.join("");
}

console.log(deleteLastGrapheme("😀👍🏽")); // 😀
```

配列・オブジェクト

Chapter 5

069 配列を並び替えたり、逆順にしたい

利用シーン

- 数値やオブジェクトの配列を昇順・降順に並び替える
- 配列の要素を逆順にする
- 元の配列を破壊せずに並び替え・逆順の結果を得る

Syntax

構文	意味	元の配列を破壊するか?
配列.toSorted(比較関数※)	並び替えた新しい配列を返す	しない
配列.sort(比較関数※)	配列を並び替える	する
配列.toReversed()	要素を逆順にして新しい配列を返す	しない
配列.reverse()	要素を逆順にする	する

※ 省略可能

● 比較関数の戻り値と並び順

戻り値	意味
負の数	要素1を要素2より前に配置する
0	順序を変更しない
正の数	要素1を要素2より後ろに配置する

toSortedメソッドによる並び替え

配列の要素を並び替えるにはtoSortedメソッドを使います。引数に渡した比較関数の中で2つの要素を比較し、戻り値に応じて並び順を決めます。
次の例では、数値の配列を昇順に並び替えています。

■ JavaScript recipe_069_1.js

```
const numbers = [10, 2, 30];

const result = numbers.toSorted((a, b) => a - b);
console.log(result); // [2, 10, 30]
```

降順に並べたい場合は、差の計算を逆向きにします。

■ JavaScript recipe_069_2.js

```
const result = numbers.toSorted((a, b) => b - a);
console.log(result); // [30, 10, 2]
```

オブジェクトの配列の並び替え

実際の開発では、APIから受け取ったオブジェクト配列の要素を並び替えるケースが多いです。次の例では、ユーザー情報を年齢で昇順に並び替えています。

■ JavaScript recipe_069_3.js

```
const users = [
  { name: "田中", age: 24 },
  { name: "鈴木", age: 18 },
  { name: "佐藤", age: 36 },
];

const sortedByAge = users.toSorted((a, b) => a.age - b.age);
console.log(sortedByAge);
// [
//   { name: "鈴木", age: 18 },
//   { name: "田中", age: 24 },
//   { name: "佐藤", age: 36 }
// ]
```

比較関数の省略

比較関数が与えられない場合、要素は文字列として扱われ、UTF-16コードポイントの昇順に並び替えられます。数値の配列では意図しない結果になるため、可能な限り比較関数を指定したほうがよいです。

■ JavaScript recipe_069_4.js

```
const numbers = [3, 1, 20];
const sorted = numbers.toSorted();
console.log(sorted); // [1, 20, 3]（辞書順になる）
```

toReversedメソッドによる逆順化

配列の要素を逆順にするにはtoReversedメソッドを使います。元の配列を変更せずに、逆順に並び替えた新しい配列を返します。

■ JavaScript recipe_069_5.js

```
const numbers = [1, 2, 3];
const reversed = numbers.toReversed();
console.log(reversed); // [3, 2, 1]
```

非破壊的メソッドと破壊的メソッドの違い

toSortedメソッドとtoReversedメソッドは非破壊メソッドであり、元の配列を変更せずに新しい配列を返します。一方、sortメソッドとreverseメソッドは破壊的メソッドであり、元の配列を直接変更します。

■ JavaScript recipe_069_6.js

```
// 破壊的メソッド：元の配列が変更される
const nums1 = [10, 2, 30];
nums1.sort((a, b) => a - b);
console.log(nums1); // [2, 10, 30]（元の配列が変更されている）

// 非破壊メソッド：元の配列は変更されない
const nums2 = [10, 2, 30];
const sorted = nums2.toSorted((a, b) => a - b);
console.log(sorted); // [2, 10, 30]
console.log(nums2); // [10, 2, 30]（元の配列は変更されない）
```

069 配列を並び替えたり、逆順にしたい

元の配列を保持したまま結果を得たいときは、非破壊のtoSortedメソッド・toReversedメソッドを選択するとよいでしょう。

実例

チャットメッセージを新しい順に表示する例です。APIから取得したメッセージ配列をtoReversedメソッドで逆順に変換し、新しいメッセージを先頭に配置します。

■ JavaScript　recipe_069_7.js

```
const messages = [
  { id: 1, text: "こんにちは", timestamp: "10:00" },
  { id: 2, text: "お元気ですか？", timestamp: "10:05" },
  { id: 3, text: "はい、元気です", timestamp: "10:10" },
];

const latestMessages = messages.toReversed();
console.log(latestMessages);
// [
//   { id: 3, text: "はい、元気です", timestamp: "10:10" },
//   { id: 2, text: "お元気ですか？", timestamp: "10:05" },
//   { id: 1, text: "こんにちは", timestamp: "10:00" }
// ]
```

型の絞り込みと高度な型操作

Chapter 9

115 型述語を推論させたい

- 型述語を書かずに型ガードを実装する
- filterでnullやundefinedを型安全に除外する
- instanceofやタグ付きユニオンの絞り込みを自動推論させる

Syntax

構文	`function isType(value: unknown)`
意味	変数が特定の型であるかを確認するユーザー定義型ガード関数
構文	`array.filter((value) => value != null)`
意味	nullやundefinedを除外する配列フィルタリング
構文	`array.filter((value) => value instanceof Class)`
意味	特定のクラスのインスタンスのみを抽出

TypeScriptでは「x is S」や「変数 is 型」と書くことで、その値が特定の型かどうかを判定し、コンパイラーに「この値はこの型」と伝える「型述語」を表現できます。型述語は本来戻り値型に明示しますが、関数本体から自動で推論させることもできます。
次のisNumber関数を見てみましょう。引数が数値かどうかを判定する関数です。型述語の記述「: value is number」をしていませんが、正しく型ガードが行われ、valueがnumber型に絞り込まれています。

■ TypeScript　recipe_115_1.ts

```
function isNumber(value: number | string) {
  return typeof value === "number";
}

function main(value: number | string) {
  if (isNumber(value)) {
    // valueはnumber型に絞り込まれる
    value.toFixed(2);
  }
}
```

型述語の推論結果の確認

TS PlaygroundやVS Codeでユーザー定義型ガードの関数を確認すると、戻り値として型述語が推論されていることがわかります。

■ TypeScript　recipe_115_2.ts

```
function isString(value: unknown) {
  return typeof value === "string";
}
// ホバーすると: function isString(value: unknown): value is string

function isNumber(value: number | string) {
  return typeof value === "number";
}
// ホバーすると:
// function isNumber(value: number | string): value is number
```

● 型述語の推論結果の確認

filterメソッドでの応用

型述語の推論は、配列のfilterメソッドで型を絞り込むときに使えます。数値、null、undefinedが混在する配列からnullとundefinedを取り除く場合、次のコードで型安全に実現できます。

■ TypeScript　recipe_115_3.ts

```
const result
  = [12, null, 24, undefined, 48].filter((value) => value != null);
// resultはnumber[]に推論される
```

型述語の推論機能を使わない場合は、明示的な型述語を記述する必要があります。ただし、明示的な型述語は実装と一致しなくてもコンパイルエラーにならないため、ランタイムエラーの原因となる危険性があります。

■ TypeScript recipe_115_4.ts

```
const result = [12, null, 24, undefined, 48].filter(
  (value): value is number => value != null,
);
```

value != nullのようにnullとundefinedだけをふるい落とす比較は推論が正しく効きます。一方でfilter(Boolean)のような「truthy判定」ではTypeScriptが型を十分に絞り込めず、falsyに評価される0や空文字まで除外されかねません。必ず残したい値と落としたい値を明示し、条件式を意図的に書き分けましょう。

instanceofやタグ付きユニオンによる絞り込み

instanceofやタグ付きユニオンによる絞り込みの結果も推論できます。これはif文での型ガードだけでなく、filterメソッドでも同様に機能します。

■ TypeScript recipe_115_5.ts

```
type A = { type: "A"; a: number };
type B = { type: "B"; b: number };

function isA(x: A | B) {
  return x.type === "A";
}

function check(foo: A | B) {
  if (isA(foo)) {
    // OK!
    console.log(foo.a);
  }
}
```

実例

商品検索システムでは、検索結果の一部がnullやundefinedになることがあります。次の例では、無効なデータを除外し、在庫のある商品の平均価格を計算しています。

■ TypeScript　　recipe_115_6.ts

```
type Product = {
  id: number;
  name: string;
  price: number;
  stock: number;
};

// 商品検索結果（一部の商品データが取得できない場合がある）
const searchResults: (Product | null | undefined)[] = [
  { id: 1, name: "ノートPC", price: 80000, stock: 5 },
  null, // データ取得エラー
  { id: 2, name: "マウス", price: 2000, stock: 0 },
  undefined, // 商品が存在しない
  { id: 3, name: "キーボード", price: 5000, stock: 10 },
];

// 有効な商品データのみを抽出
const validProducts = searchResults
  .filter((product) => product != null)
  .filter((product) => product.stock > 0); // さらに在庫ありの商品のみ

// 型安全に商品価格の合計を計算
const totalPrice = validProducts.reduce(
  (sum, product) => sum + product.price,
  0,
);
const averagePrice =
  validProducts.length > 0 ? totalPrice / validProducts.length : 0;

console.log(`在庫商品数: ${validProducts.length}`); // 2
console.log(`平均価格: ${averagePrice.toLocaleString()}円`);
// "平均価格: 42,500円"
```

117 オブジェクトの型推論結果を保持しつつ型チェックをしたい

- オブジェクトのwideningを防ぎつつ型チェックする
- 設定オブジェクトや定数を定義する
- リテラル型を保持しながら型安全性を確保する

Syntax

構文	意味
`オブジェクト as const satisfies 型`	wideningを防ぎつつ型構造をチェック

「as const satisfies」は、as constとsatisfies演算子を組み合わせた機能です。オブジェクトのwidening（型の拡大）を防ぎながら、同時に型チェックも実行できます。

オブジェクトはwideningされる

次のようにオブジェクトを変数定義すると、user変数は{ name: string, age: number }型にwideningします。具体的には、nameプロパティのリテラル型"鈴木"がstring型に、ageプロパティのリテラル型24がnumber型に拡大されます。

■ TypeScript recipe_117_1.ts

```
const user = {
  name: "鈴木",
  age: 24,
};
```

as constを使うとwideningを防げるが、型チェックができない

as const（レシピ116「値の型の拡大（widening）を防ぎたい」参照）を使うと、wideningを防げます。次のようにas constを付与すると、user変数は{ readonly name: "鈴木"; readonly age: 24 }型になります。nameプロパティとageプロパティのリテラル型が保持され、さらにreadonly修飾子もつきます。

■ TypeScript recipe_117_2.ts

```
const user = {
  name: "鈴木",
  age: 24,
} as const;
```

しかし、as constだけでは特定の型構造との適合性チェックができません。次の例では、nameをnumber型にしてもエラーにならず、想定する「nameがstring型でageがnumber型」という構造を崩してしまいます。

■ TypeScript recipe_117_3.ts

```
const user = {
  name: 123,
  age: 24,
} as const;
```

なお、型注釈を付けた場合、as constでリテラル型に絞り込んでいるのが無効化されてしまいます。次の例では、User型で型注釈をしているため、as constの効果が失われ、nameプロパティはstring型、ageプロパティはnumber型として推論されます。

■ TypeScript recipe_117_4.ts

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

const user: User = {
  name: "鈴木",
  age: 24,
} as const; // as constより型注釈が優先される

// user.nameはstring型、user.ageはnumber型
```

satisfiesを使うと型チェックができるが、リテラル型推論が失われる

レシピ124「型推論結果を保持しつつ型チェックをしたい」で説明しているように、satisfies演算子を使うと、型チェックができます。次の例では、オブジェクトが{ name: string, age: number }型かどうかをチェックしています。

■ TypeScript recipe_117_5.ts

```
const user = {
  name: "鈴木",
  age: 24,
} satisfies {
  name: string;
  age: number;
};
```

型の一致しない不正なオブジェクトを定義しようとすると、コンパイルエラーになります。次の例では、ageプロパティがnumber型であることを期待しているのに、string型にしてしまっているので、コンパイルエラーになります。

■ TypeScript recipe_117_6.ts

```
const user = {
  name: "鈴木",
  age: "24", // × Type 'string' is not assignable to type 'number'
} satisfies {
  name: string;
  age: number;
};
```

ただし、satisfies演算子を使うと、リテラル型推論が失われます。次の例では、user変数は{ name: string, age: number }型になります。{ name: "鈴木", age: 24 }のリテラル型が失われています。

■ TypeScript recipe_117_7.ts

```
const user = {
  name: "鈴木",
  age: 24,
} satisfies {
  name: string;
  age: number;
};
```

as const satisfiesを使うと、wideningを防ぎつつ型チェックができる

as constとsatisfies演算子を組み合わせると、wideningを防ぎつつ型チェックができます。次の例では、as const satisfiesにより、オブジェクトの型チェックを行いつつ、型推論結果を保持します。

■ TypeScript recipe_117_8.ts

```
const user = {
  name: "鈴木",
  age: 24,
} as const satisfies {
  name: string;
  age: number;
};
```

この場合、型推論結果が保持されます。

■ TypeScript recipe_117_9.ts

```
user.name;
// ^? "鈴木"
```

satisfiesにあわないオブジェクトのプロパティを定義しようとすると、コンパイルエラーになります。次の例では、nameプロパティがstring型であることを期待しているのに、number型にしてしまっているので、コンパイルエラーになります。

■ TypeScript recipe_117_10.ts

```
const user = {
  name: 123, // × Type 'number' is not assignable to type 'string'
  age: 24,
} as const satisfies {
  name: string;
  age: number;
};
```

実例

API定数での活用例です。
次の例では、API_ENDPOINTSオブジェクトの各エンドポイントが正しい構造（methodとpathプロパティを持つ）かをチェックします。あわせて、API_ENDPOINTS.users.methodは"GET"、API_ENDPOINTS.createUser.methodは"POST"というリテラル型で保持されます。

■ TypeScript recipe_117_11.ts

```
type HttpMethod = "GET" | "POST" | "PUT" | "DELETE";

const API_ENDPOINTS = {
  users: {
    method: "GET",
    path: "/api/users",
  },
  createUser: {
    method: "POST",
    path: "/api/users",
  },
} as const satisfies
  Record<string, { method: HttpMethod; path: string }>;
```

TypeScript設定

Chapter 14

174 存在しない可能性がある配列のインデックスアクセスを型安全にしたい

利用シーン

- 配列やオブジェクトの未定義要素へのアクセスを防ぐ
- インデックスアクセスの戻り値にundefinedを含めて安全にする
- 値の取得時に存在確認を強制する

Syntax

構文	意味
noUncheckedIndexedAccess	インデックスアクセスにundefinedチェックを要求

noUncheckedIndexedAccessを有効にすると、配列やオブジェクトへのインデックスアクセスの戻り値型に自動的にundefinedが付与されます。インデックスで指定された要素が存在しない場合、その値はundefinedとして型が推論されるため、存在チェックの漏れをコンパイル時に検出できます。

noUncheckedIndexedAccessがOFFの場合の挙動

次の例は、食べ物名の配列foodsから要素を取得し、要素の先頭の文字を出力するためのプログラムです。先頭文字の出力のために、文字列用のメソッドatを使用しています。インデックス0と1の要素は存在するため問題なく実行されますが、インデックス2の要素は存在しないため、ランタイムエラーが発生します。

TypeScript recipe_174_1.ts

```
const foods: string[] = ["カレー", "うどん"];
console.log(foods[0].at(0)); // 「カ」が出力される
console.log(foods[1].at(0)); // 「う」が出力される
console.log(foods[2].at(0)); // ランタイムエラーが発生する
```

なぜこういったことが起こるのかというと、foods[0]、foods[1]、そしてfoods[2]も、型推論によりstring型として推論されているためです。foods[2]は存在しないためundefinedが返され、undefinedに対してatメソッドを実行してランタイムエラーになっています。開発者としてはfoods[2]が存在しないことを明らかに知っていますが、TypeScriptコンパイラーがその情報を判別できていないのです。

174 存在しない可能性がある配列のインデックスアクセスを型安全にしたい

noUncheckedIndexedAccessがONの場合の挙動

次のようにnoUncheckedIndexedAccessを有効化します。

■ JSON　recipe_174_2/tsconfig.json

```
{
  "compilerOptions": {
    "noUncheckedIndexedAccess": true
  }
}
```

この場合、foods[0]、foods[1]、そしてfoods[2]は、型推論によりstring | undefined型に推論されます。したがって、各要素に対してat(0)を実行する場合はオプショナルチェーン(?.)を使用してundefinedの場合の処理を記述する必要があります。

■ TypeScript　recipe_174_2/index.ts

```
const foods: string[] = ["カレー", "うどん"];
console.log(foods[0]?.at(0)); // 「カ」が出力される
console.log(foods[1]?.at(0)); // 「う」が出力される
console.log(foods[2]?.at(0)); // undefinedが出力される
```

foods[0]やfoods[1]が明らかに存在している場合、?.を使うのは冗長に見えるかもしれません。本書では、存在しない要素へのアクセスを防ぐ利点を重視し、noUncheckedIndexedAccessを有効にしたうえで?.を使う運用を推奨します。

177 Node.jsでTypeScriptを実行するとき、削除不可能構文をエラーにしたい

- Node.jsのTS実行で非対応構文を事前に検出する
- enumやnamespaceなど削除不可能な構文を避ける
- tsconfigでNode.jsとの互換性を高める

Syntax

オプション	説明
`erasableSyntaxOnly`	削除可能な構文のみを許可し、削除不可能構文をエラーにする

erasableSyntaxOnlyフラグを使うと、enumやnamespace、クラスのパラメータプロパティ、moduleキーワードなど、Node.jsでTypeScriptを実行するときに非互換な構文をエラーとして検出できます。Node.jsとTypeScriptの互換性を高めるのに役立ちます。フラグはtsconfig.jsonのcompilerOptionsに設定します。次のように記述すると、削除不可能構文がコンパイル時にエラーになります。

■ JSON　recipe_177_1/tsconfig.json

```
{
  "compilerOptions": {
    "erasableSyntaxOnly": true
  }
}
```

有効化した状態でenumやnamespace、クラスのパラメータプロパティを含むコードを書くと、コンパイルエラーになります。

■ TypeScript　recipe_177_1/index.ts

```
enum MyEnum {
  A,
  B,
  C,
}

namespace myNameSpace {
  export const foo = 1;
```

```
}

class MyClass {
  constructor(private myField: string) {}
}
```

コンパイルすると、次のようにerasableSyntaxOnlyでは許可されない構文である旨のメッセージが表示されます。

```
This syntax is not allowed when 'erasableSyntaxOnly' is enabled.
```

なぜこのフラグが必要なのか

Node.jsはTypeScriptファイルをそのまま実行できる

Node.jsはTypeScriptファイルから型注釈を取り除き、そのままJavaScriptとして実行できます。
次の例では、TypeScriptファイルindex.tsをNode.jsで実行しています。

■ TypeScript　recipe_177_2/index.ts

```
const myName: string = "とんこつ";
console.log(myName);
```

```
node index.ts
```

削除不可能構文を排除する

enumやnamespace、クラスのパラメータプロパティなどの構文は、単純な型注釈の削除ではなく変換処理が必要になるため、「削除不可能な構文」とみなされます。erasableSyntaxOnlyフラグを有効にすると、こうした構文をコンパイル時にエラーとして検出でき、Node.jsでそのまま実行できるコードに保てます。
erasableSyntaxOnlyがエラーとする主な構文は次のとおりです。

- enum
- namespace
- クラスのパラメータプロパティ
- レガシーなmoduleキーワード
- import = エイリアス

削除不可能構文の代替パターン

削除不可能構文が使われているコードは、代替パターンで書き換えられます。

enumの代わりにオブジェクトリテラルを使う

enumを使った削除不可能コードは、次のようなものです。

Before（erasableSyntaxOnlyでエラーになる例）

TypeScript recipe_177_4.ts

```
enum MyEnum {
  A,
  B,
  C,
}
```

定数オブジェクトとas constの組み合わせで代替できます。

After

TypeScript recipe_177_5.ts

```
const MyEnum = {
  A: 0,
  B: 1,
  C: 2,
} as const;

type MyEnum = (typeof MyEnum)[keyof typeof MyEnum];
```

177 Node.jsでTypeScriptを実行するとき、削除不可能構文をエラーにしたい

namespaceの代わりにモジュールとimport * as

namespaceを使った削除不可能コードは、次のようなものです。

- Before（erasableSyntaxOnlyでエラーになる例）

■ TypeScript recipe_177_6.ts

```
namespace myNameSpace {
  export const myName = "とんこつ";
}
```

これは、モジュールとimport * asを組み合わせます。

- After

■ TypeScript recipe_177_7/foo.ts

```
export const myName = "とんこつ";
```

■ TypeScript recipe_177_7/main.ts

```
import * as myNameSpace from "./foo";
console.log(myNameSpace.myName); // "とんこつ"
```

パラメータプロパティの代わりに明示的なフィールド宣言

クラスのパラメータプロパティを使った削除不可能コードは、次のようなものです。

- Before（erasableSyntaxOnlyでエラーになる例）

■ TypeScript recipe_177_8.ts

```
class MyClass {
  constructor(private myField: string) {}
}
```

フィールド宣言と代入を分けることで、代替可能です。

- **After**

■ **TypeScript** recipe_177_9.ts

```
class MyClass {
  private myField: string;
  constructor(myField: string) {
    this.myField = myField;
  }
}
```