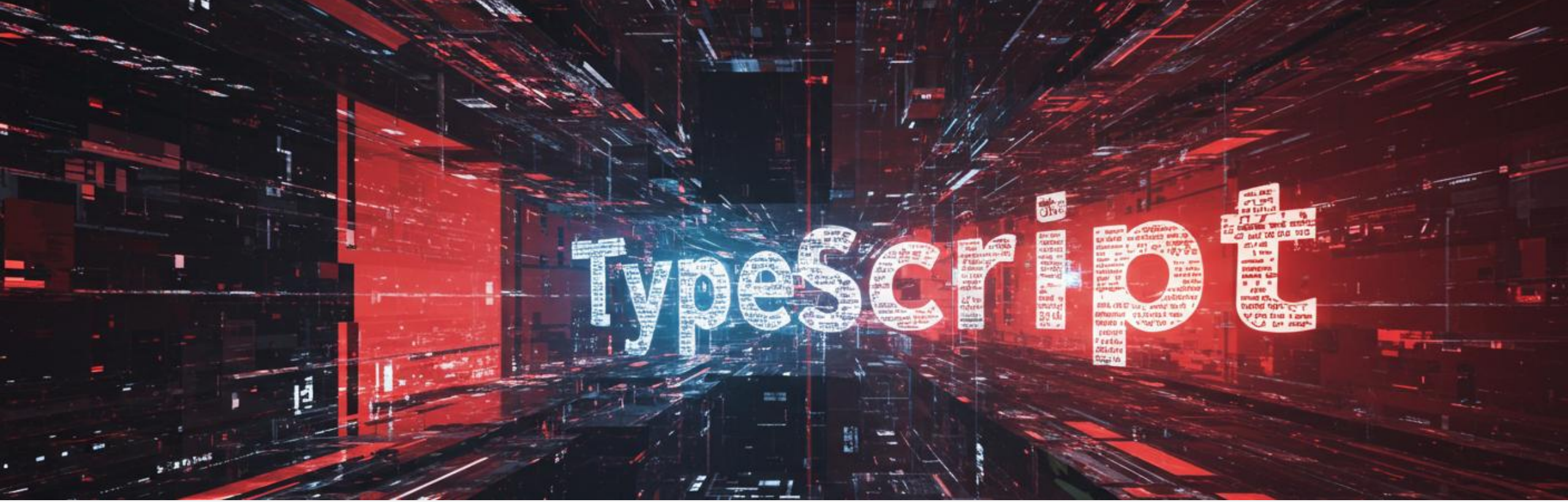


# 全2回で学ぶ TypeScript入門講座 2回目

2025/7/15

Kazuma SEKIGUCHI

# 前回のアジェンダ

- TypeScriptとは？
- 型システムの理解
- 基本的な型
- 関数の型

# 今回のアジェンダ

- ジェネリクスとは
- 条件型と型ガードとは
- 型定義ファイルとの付き合い方

# ジェネリクス

- 型を動的に受け取り、その型に基づいた処理を行う機能
  - ある関数が引数の型を特定せずにその型に沿って処理をするとき、などに利用する

```
function identity<T>(arg: T): T {
    return arg;
}
const result = identity<string>("hello"); // result は string 型
const num = identity<number>(42);//numはnumber型
```

  - Tは型パラメータで任意の型を受け取る
  - 関数呼び出し時に<string>や<number>で型を指定できる
  - 受け取った型に基づいて戻り値も同じ型になる

# ジェネリクスを使う理由

- 型再利用の向上
  - 型に依存せずに同じロジックを使い回せる
  - Anyでも型に依存せずに利用できるが、型安全を維持できる
- 配列操作関数で、配列内の要素型を保持しつつ安全に操作する

```
function getFirstElement<T>(arr: T[]): T {
    return arr[0];
}

const num = getFirstElement([1, 2, 3]);          // num: number
const str = getFirstElement(["a", "b", "c"]);    // str: string
```

# ジェネリクス

```
async function fetchData<T>(url: string): Promise<T> {
    const response = await fetch(url);
    return response.json();
}
type User = { id: number; name: string };
async function loadUser() {
    const user = await fetchData<User>("/api/user");
    console.log(user.name); // userはUser型、安全に補完可能
}
```

- APIレスポンスの型をジェネリクスで受け取り、取得時に型安全に取り扱うことが可能
- APIごとに異なるレスポンス型をジェネリクスで渡すだけで安全に扱える

# 複数の型のパラメータ

```
function merge<T,U>(obj1:T,obj2:U):T & U{
  return {...obj1,...obj2};
}
const merged = merge({name:"Taro"},{age:24});
console.log(merged.name);//Taro
```

{...obj1, ...obj2} はスプレッド構文で、obj1 と obj2 のプロパティを結合した新しいオブジェクトを返す

{ name: "Taro" } の型は { name: string }
{ age: 24 } の型は { age: number }
返り値の型は { name: string } & { age: number } となり、{ name: string, age: number } と同じ型で扱える。

- 複数の型パラメータも持つことが可能
- obj1: T, obj2: U とすることで、どの型のオブジェクトでも受け取れる
- 戻り値の型 T&Uは交差型（Intersection Type）と呼ばれ、TとU の両方のプロパティを持つ型となる
  - 与えるプロパティ名が同じで違う型だとエラーになるので注意

# 条件型

- TypeScriptで開発する目的は型安全性を保ちながら開発効率・保守性を高めること
- しかし、実際の開発では
  - 外部入力（APIレスポンス、ユーザー入力、設定ファイル）
  - ユニオン型（複数の可能性のある型）

  を扱うため、型が一意に決まらない場面が必ず発生する

- 条件型 → 型を変換・除去・抽出するために必要
- 型ガード → 実行時の値の型を判別し、型推論を絞り込むために必要

# 条件型

- 条件型はある型が条件を満たすかどうかによって返す型を切り替える仕組み

```
type IsString<T> = T extends string ? "Yes":"No";
type A = IsString<string>;//"Yes"
type B = IsString<number>;//"No"
```

- 条件型は型に応じて異なる型を返す
  - 型の変換に利用可能

# 条件型

T が null または undefined なら never、
そうでなければ T を返す型

```
type MyNonNullableResponse<T> = T extends null | undefined ? never : T;
type APIResponse = string | null;
type Processed = MyNonNullableResponse<APIResponse>; // string
```

- APIのレスポンスを処理するときなど
  - nullが返って来る可能性があるが、呼び出し後はnullを除外した型で扱いたい場合に利用する
- 条件型がユニオン型に適用されると、ユニオン型の各要素に対して個別に適用され、その結果のユニオンが取られるというルール（distributive conditional types：分配型条件）で動作する
  - neverはユニオン型からは自動的に取り除かれる
- ユニオン型から特定の型を抜くことが可能となる

# 条件型の例

```
type ApiResponse = User | null;
function handle(res: NonNullable<ApiResponse>) {
  // res は User 型として扱える
}
const data: ApiResponse = await fetchUser();
if (data !== null) {
  handle(data);
}
```

この段階ではUser型かnullが返って来るか分からない

- NonNullable<T>はこの関数にnullを与えてはいけないという型の契約
  - 実行時の防御にはならないから呼び出し時にnullチェックが必要

# 型ガード

- TypeScriptにおいて実行時に値の型を判別し、その結果を基にコンパイラが型を絞り込める機能
- typeofを使うことで、型ガードが可能になる

```
function example(x: number | string) {
    if (typeof x === "string") {
        x.toUpperCase(); // x は string 型と推論
    } else {
        x.toFixed();     // x は number 型と推論
    }
}
```

# in型ガード

- オブジェクトが特定のプロパティを持っているかで型を絞り込むことが可能

```
type User = { name: string };
type Admin = { name: string; role: string };

function process(person: User | Admin) {
    if ("role" in person) {
        console.log(person.role); // personはAdmin型に絞り込まれる
    } else {
        console.log(person.name); // personはUser型に絞り込まれる
    }
}
```

roleというプロパティを持っているならpersonはAdmin型になる

# instanceof 型ガード

- classに使用し、インスタンスを判定する

```
class Dog {
    bark() { console.log("ワン"); }
}
class Cat {
    meow() { console.log("ニャー"); }
}
function process(pet: Dog | Cat) {
    if (pet instanceof Dog) {
        pet.bark(); // petはDog型に絞り込まれる
    } else {
        pet.meow(); // petはCat型に絞り込まれる
    }
}
```

petがDogクラスのインスタンスかどうかを判定する

# ユーザー定義型ガード

```
function isString(x: unknown): x is string {
    return typeof x === "string";
}

function process(x: unknown) {
    if (isString(x)) {
        x.toUpperCase(); // x は string 型と推論
    }
}
```

これがユーザー定義型ガード

関数が true を返す場合、引数 x は string 型であると TypeScript に伝える

- unkown型はTypeScriptの型で、どんな値でも代入できるが、使用する際には型チェックが必要な型
- unknown は 使用時に型チェック（型ガードやキャスト）をしないと操作できないため、型安全性を担保しながら柔軟性を確保できる。

# 型ガードの例

```
function isNumber(value: unknown): value is number {
  return typeof value === "number";
}
const input: unknown = form.get("age");
if (isNumber(input)) {
  input.toFixed()//整数値に変換
}
```

- ユーザー定義型ガードのisNumber()を使っているので、もしstring型が来てもinput.toFixed()は実行されない

# 条件型と型ガード

| | 条件型 | 型ガード |
|---|---|---|
| **目的** | 型変換・フィルタ | 実行時型判別・絞込 |
| **例** | T extends U ? X : Y | typeof, instanceof, ユーザー定義 |
| **使う場面** | APIレスポンス加工 | ユニオン型・unknown処理 |

# ユーティリティ型

- TypeScriptに存在する既存の型を変形再利用するための型操作ツール
  - 型の柔軟性や可読性を高めるために利用する
- Partial<T>
  - 全てのプロパティを省略可にする
- Readonly<T>
  - すべてのプロパティを読み取り専用にする
- Record<K,T>
  - キーと型を指定してオブジェクト型を生成
- Pick<T,K>
  - 指定したプロパティだけを取り出す
- Omit<T,K>
  - 指定したプロパティを除外する

# Partial<T>

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

type UserUpdate = Partial<User>; // 全プロパティが省略可能になる
const update: UserUpdate = {
  name: "Taro" // age や id がなくてもOK
};
```

- 更新フォームやAPIの送信データなど、部分的な値のみ送るときに使う

# Readonly<T>

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

const user: Readonly<User> = {
  id: 1,
  name: "Taro"
};
user.name = "Jiro"; //読み取り専用にしているのでエラーになる
```

- 設定値や定数など、変更されるべきでないオブジェクトに使う

# Record<K, T>

```
type Status = "draft" | "published" | "archived";
const postLabels: Record<Status, string> = {
  draft: "下書き",
  published: "公開",
  archived: "アーカイブ済"
};
function getStatusLabel(status: Status): string {
  return postLabels[status];
}
const s:string = getStatusLabel("draft"); // "下書き"
```

- リテラルユニオン型は実行時に存在しない型のみの定義
  - Recordと組み合わせることでenumのように利用可能

# Pick<T,K>

```
type User = {
  id: number;
  username: string;
  email: string;
};
type PublicUser = Pick<User, "id" | "username">;
const u: PublicUser = {
  id: 1,
  username: "Taro"
};
```

- idとnameだけの型を定義する
- 個人情報などを除いて、外部公開用の型を作成したいときなどに使う

# Omit<T,K>

```
type User = {
  id: number;
  username: string;
  email: string;
};
type UserInput = Omit<User, "id">;
const newUser: UserInput = {
  username: "Hanako",
  email: "hanako@example.com"
};
```

- idを除いた型を定義
- 自動生成されるidなどを除外して、登録用フォームの型を作るときに使う

# TとかUとかKとか

- 型パラメータ名は決められていない
  - 慣例としてその大文字が使われているだけ
  - できるだけ慣例に従った方が良い

| 名前 | 意味・使い方の目安 | よく出る場面 |
| --- | --- | --- |
| T | Type（汎用の型） | 基本のジェネリクス |
| U | Typeの2番目（Tとの組み合わせ） | 複数の型を受け取る関数 |
| K | Key（オブジェクトのキー） | keyof との組み合わせ |
| V | Value（値の型） | Record<K, V> など |
| E | Element（要素） | 配列やイベントなど |
| R | Result（結果） | 関数の戻り値の型など |

# 型定義ファイル（@types）との付き合い方

- 外部ライブラリの型定義ファイルの扱い方を知っておく必要がある
  - TypeScriptでJavaScriptライブラリーを安全に使うためにはライブラリーの型情報が必要
  - この役割を担うのが@typesという型ファイル群
  - 型情報だけを別ファイルで提供し、TypeScriptの型チェックを安全に利用できるようにするものが型定義ファイル（.d.tsファイル）

# @types

- npmでパッケージを導入するときに@typesを付けることで型定義を取得することが可能
  - express→@types/express
  - jquery→@types/jquery

  のようにすることで取得可能
- 型定義ファイルはdevDependencies（開発時だけ使う）に入れるのが普通
  - --save-devを付けてインストールを行う

```
npm install --save-dev @types/jquery
```

# @typesが提供されていない場合

- 最近ではほとんどのライブラリーで提供されてはいるが、提供されていない場合は、自作することが可能
  - 自作は比較的大変なので別ライブラリーにするか、危険を承知でanyとして扱う

src/types/mylib.d.tsに以下を記述
declare module "mylib";

この場合読み込みたいが型定義が無いライブラリー名がmylib

- JSライブラリーを読み込むときはモジュール形式で読み込む
  - 型定義ファイルも同時に読み込まれる

```
import * as jquery from "jquery";
```

# scriptタグで読み込む場合

- @typesパッケージが存在する場合は、npmで型定義だけを読み込んでおけばOK
  - jqueryなどは型定義がnpmに存在する
  - jquery本体はscriptタグで読み込めばOK
- 型定義が無ければ.d.tsファイルを自作する必要がある
  - 自分で作成した場合なども.d.tsファイルを作成する必要あり

# fetchなどで外部との通信を行う場合

- fetchなどで取得したレスポンスには型が与えられないので、明示的に記述しておく必要がある

```
type User = {
id:number;
email:string;
isAdmin:boolean
}
```

```
async function
fetchUser(id:number):Promise<User>{
  const response = await fetch(`/api/getuser/${id}`);
if(response.ok){
  const data:User = await response.json();
  return data;
}
else{
  throw new Error('ユーザーデータの取得に失敗');
}
```

# 前回不参加の方は以下の手順を

- フォルダーを作成
- ターミナルなどを起動
- TypeScriptのインストール
- tsconfig.jsonを上書きする

# TypeScriptの作成

- NPMでTypeScriptを導入する
  - グローバルにインストールする方法と、ローカルにインストールする方法が存在
  - 今回はグローバルにインストールする

```
npm install -g typescript
tsc -v
```

TypeScriptをグローバルにインストールする

TypeScriptのバージョンを確認する

```
npx tsc --init
```

作業フォルダー上で実行することで、tsconfig.jsonファイルが生成される

# TypeScriptでのプロジェクト作成

- フォルダーを作成
- ターミナル等でフォルダーに移動し、

```
npm init -y
```

でpackage.jsonを作成

- TypeScriptの設定ファイルを作成する

```
npx tsc --init
```

作業フォルダー上で実行することで、
tsconfig.jsonファイルが生成される

- tsconfig.jsonというTypeScriptの動作条件を設定するためのファイルを再生する

# TypeScriptからJSへの変換

- TypeScriptは拡張子を.tsとして保存する
  - トランスパイルすると、.jsファイルとして生成される

```
npx tsc index.ts
```

index.tsをindex.jsにトランスパイルする

  - index.jsが生成されるので、必要に応じてhtmlに読み込ませたりして実行する

# tsconfig.jsonの設定

```
{
  "compilerOptions": {
    "target": "ES2022",
    "module": "ES2022",
    "moduleResolution": "Node",
    "strict": true,
    "esModuleInterop": true,
    "skipLibCheck": true,
    "forceConsistentCasingInFileNames": true,
    "noUncheckedIndexedAccess": true,
    "resolveJsonModule": true,
    "isolatedModules": true,
    "verbatimModuleSyntax": true,
    "incremental": true,
    "outDir": "dist",
    "rootDir": "src",
    "sourceMap": true
  },
  "include": ["src/**/*.ts"],
  "exclude": ["node_modules", "dist"]
}
```

- 現状ブラウザーで実行するJSを生成するなら左側の設定で良いはず
- targetで出力するJSのバージョンを指定することが可能
  - ターゲットとするブラウザーに合わせて変更する
  - es2016,es2020などが指定可能
- moduleも同様
  - ES2022,esnextなどを指定
- Nodeでも実行するなら、moduleをNodeNextにする

# ありがとうございました。

TypeScript自体は型をどう扱うか、という点に尽きます。

型を上手く扱うことで、効率的でバグの少ない
開発が可能になります。
できるところから導入していくことができるので、
少しずつでも導入していくと良いでしょう