Chapter 7: Generics - Tổng Quát Hóa Code

Mô tả tổng quát

Chapter này giới thiệu về **Generics**, một tính năng mạnh mẽ của TypeScript, giúp viết code linh hoạt, tái sử dụng và vẫn đảm bảo an toàn kiểu (type safety). Generics cho phép bạn định nghĩa các thành phần như hàm, interface, hoặc class có thể làm việc với nhiều kiểu dữ liệu khác nhau mà không cần sử dụng kiểu `any` thiếu an toàn. Chúng ta sẽ tìm hiểu cách tạo generic functions, generic interfaces, generic classes, áp dụng ràng buộc (constraints) và sử dụng một số generic utility types cơ bản.

Tiêu đề Chapter học: Generics: Viết Code Linh Hoạt và Tái Sử Dụng.

Tóm tắt lý thuyết chính

1. Vấn đề với kiểu `any` và tại sao cần Generics

Khi viết hàm hoặc thành phần cần xử lý nhiều kiểu dữ liệu, một cách đơn giản là sử dụng kiểu `any`. Tuy nhiên, `any` làm mất đi lợi ích của TypeScript là kiểm tra kiểu tĩnh (static type checking), dẫn đến lỗi runtime khó phát hiện.

Ví dụ minh họa:

```
function identityAny(arg: any): any {
    return arg;
}

let outputAny = identityAny("myString"); // outputAny là any
console.log(outputAny.length); // Không báo lỗi khi biên dịch, nhưng nếu arg
```

```
let outputNumAny = identityAny(100);
console.log(outputNumAny.toFixed()); // Không báo lỗi khi biên dịch, nhưng n
```

Giải thích chi tiết:

- Kiểu `any` cho phép bạn truyền bất kỳ giá trị nào, nhưng TypeScript không thể kiểm tra tính hợp lệ của các thuộc tính hoặc phương thức (như `length` hoặc `toFixed`).
- Điều này dẫn đến việc code có thể chạy đúng trong một số trường hợp, nhưng dễ gây lỗi khi dữ liệu không như kỳ vọng.
- **Generics** khắc phục vấn đề này bằng cách sử dụng **biến kiểu (type variable)**, cho phép định nghĩa kiểu linh hoạt nhưng vẫn đảm bảo an toàn.

Lợi ích của Generics:

- **An toàn kiểu**: TypeScript kiểm tra kiểu tại thời điểm biên dịch.
- **Tái sử dụng**: Một hàm/class/interface có thể làm việc với nhiều kiểu dữ liệu.
- **Dễ bảo trì**: Code rõ ràng hơn, dễ đọc và dễ mở rộng.

---

2. Generic Functions (Hàm Generic)

Hàm generic sử dụng biến kiểu (type variable), thường được khai báo trong dấu `<T>` ngay sau tên hàm. Biến kiểu `T` đại diện cho một kiểu dữ liệu bất kỳ, nhưng vẫn đảm bảo tính nhất quán trong tham số và giá trị trả về.

Cú pháp cơ bản:

```
function identity<T>(arg: T): T {
    return arg;
}
```

Cách sử dụng:

1. **Tường minh chỉ định kiểu:**

```
let outputString = identity<string>("myString"); // T là string
let outputNumber = identity<number>(100);        // T là number

console.log(outputString.toUpperCase()); // "MYSTRING"
console.log(outputNumber.toFixed(2));    // "100.00"
```

2. **TypeScript tự suy luận kiểu (type inference):**

```
let inferredString = identity("anotherString"); // T tự động là string
let inferredNumber = identity(200);             // T tự động là number
```

Giải thích chi tiết:

- Biến kiểu `T` được thay thế bằng kiểu cụ thể khi hàm được gọi.
- Type inference giúp giảm việc phải chỉ định kiểu tường minh, làm code ngắn gọn hơn.
- TypeScript đảm bảo rằng kiểu của tham số đầu vào và giá trị trả về luôn khớp nhau.

Ví dụ thực tiễn:

Hàm lấy phần tử đầu tiên của mảng:

```
function getFirstElement<T>(arr: T[]): T | undefined {
    return arr.length > 0 ? arr[0] : undefined;
}

let numbers = [1, 2, 3];
let firstNum = getFirstElement(numbers); // firstNum: number | undefined
console.log(firstNum); // 1

let strings = ["a", "b", "c"];
```

```
let firstStr = getFirstElement(strings); // firstStr: string | undefined
console.log(firstStr); // "a"
```

Lưu ý:

- Hàm generic không giới hạn số lượng biến kiểu. Ví dụ: `<T, U>` cho hai kiểu khác nhau.
- Type inference hoạt động tốt khi TypeScript có đủ thông tin từ tham số.

---

3. Generic Interfaces (Interface Generic)

Interface generic cho phép định nghĩa cấu trúc dữ liệu linh hoạt, với các kiểu có thể thay đổi.

Ví dụ cơ bản:

```
interface KeyValuePair<K, V> {
    key: K;
    value: V;
}

let entry1: KeyValuePair<string, number> = { key: "age", value: 30 };
let entry2: KeyValuePair<number, boolean> = { key: 101, value: true };
console.log(entry1.value.toFixed(0)); // "30"
// let entry3: KeyValuePair<string, number> = { key: "name", value: "Alice"
```

Giải thích chi tiết:

- `K` và `V` là các biến kiểu, có thể được thay thế bằng bất kỳ kiểu nào khi sử dụng interface.
- Interface generic giúp định nghĩa cấu trúc dữ liệu chung, ví dụ như cặp key-value trong các API hoặc cấu trúc dữ liệu phức tạp.

Ứng dụng thực tiễn: Mô tả kiểu hàm generic

```typescript
interface GenericIdentityFn<T> {
    (arg: T): T;
}

function identityFn<T>(arg: T): T {
    return arg;
}

let myIdentity: GenericIdentityFn<number> = identityFn;
console.log(myIdentity(123)); // 123
// console.log(myIdentity("hello")); // Lỗi: Argument phải là number
```

Giải thích:

- Interface `GenericIdentityFn` định nghĩa một hàm nhận và trả về cùng một kiểu `T`.
- Khi gán `identityFn` cho `myIdentity`, TypeScript khóa kiểu `T` thành `number`, đảm bảo tính an toàn.

---

4. Generic Classes (Lớp Generic)

Class generic cho phép định nghĩa các lớp làm việc với nhiều kiểu dữ liệu khác nhau.

Ví dụ:

```typescript
class DataStorage<T> {
    private data: T[] = [];

    addItem(item: T): void {
        this.data.push(item);
    }

    getItem(index: number): T | undefined {
        return index ⩾ 0 && index < this.data.length ? this.data[index] : u
    }
```

```
    getAllItems(): T[] {
        return [...this.data];
    }
}

let stringStorage = new DataStorage<string>();
stringStorage.addItem("Hello");
stringStorage.addItem("TypeScript");
// stringStorage.addItem(123); // Lỗi: Argument phải là string
console.log(stringStorage.getItem(0)?.toUpperCase()); // "HELLO"

let numberStorage = new DataStorage<number>();
numberStorage.addItem(10);
numberStorage.addItem(20);
console.log(numberStorage.getAllItems().reduce((sum, val) => sum + val, 0));
```

Giải thích chi tiết:

- Biến kiểu `T` được khai báo sau tên class, áp dụng cho tất cả các thuộc tính và phương thức.
- Class generic đảm bảo rằng tất cả phần tử trong `data` đều có cùng kiểu `T`.
- Các phương thức như `addItem` và `getItem` sử dụng `T` để duy trì tính nhất quán.

Ứng dụng thực tiễn:

- Generic classes thường được sử dụng trong các cấu trúc dữ liệu như danh sách, ngăn xếp, hàng đợi, hoặc để quản lý dữ liệu từ API.

---

5. Generic Constraints (Ràng buộc Generic)

Ràng buộc generic giúp giới hạn các kiểu mà biến kiểu có thể nhận, sử dụng từ khóa `extends`.

Ví dụ:

```typescript
interface Lengthwise {
    length: number;
}

function logLength<T extends Lengthwise>(arg: T): void {
    console.log(`Length: ${arg.length}`);
}

logLength("Hello world"); // Length: 11
logLength([1, 2, 3, 4]); // Length: 4
logLength({ length: 10, value: 3 }); // Length: 10
// logLength(10); // Lỗi: number không có thuộc tính length
```

Giải thích chi tiết:

- `T extends Lengthwise` yêu cầu `T` phải có ít nhất thuộc tính `length` kiểu `number`.
- Ràng buộc giúp TypeScript đảm bảo rằng các thuộc tính/phương thức được truy cập là hợp lệ.
- Ràng buộc có thể áp dụng cho interface, class hoặc kiểu cụ thể (như `string`).

Ví dụ nâng cao: Ràng buộc với `keyof`

```typescript
function getProperty<T, K extends keyof T>(obj: T, key: K): T[K] {
    return obj[key];
}

let obj = { a: 1, b: "hello", c: true };
let valA = getProperty(obj, "a"); // valA: number
let valB = getProperty(obj, "b"); // valB: string
// let valD = getProperty(obj, "d"); // Lỗi: "d" không tồn tại trong obj
```

Giải thích:

- `keyof T` tạo ra một union type chứa các key của `T` (ví dụ: `"a" | "b" | "c"`).
- `K extends keyof T` đảm bảo `key` là một key hợp lệ của `obj`.

- `T[K]` trả về kiểu của thuộc tính tương ứng.

---

6. Generic Utility Types cơ bản

TypeScript cung cấp các **utility types** generic để thao tác với kiểu dữ liệu. Một số ví dụ phổ biến:

- `Partial<T>` : Biến tất cả thuộc tính của `T` thành tùy chọn.
- `Readonly<T>` : Biến tất cả thuộc tính của `T` thành chỉ đọc.
- `Pick<T, K>` : Chọn một tập hợp thuộc tính `K` từ `T` .
- `Omit<T, K>` : Loại bỏ một tập hợp thuộc tính `K` từ `T` .

Ví dụ:

```
interface Todo {
    title: string;
    description: string;
    completed: boolean;
}

type PartialTodo = Partial<Todo>;
let todoToUpdate: PartialTodo = { description: "New description" }; // OK

type ReadonlyTodo = Readonly<Todo>;
let completedTodo: ReadonlyTodo = { title: "Finish report", description: "..
// completedTodo.title = "New title"; // Lỗi: Thuộc tính chỉ đọc

type PickedTodo = Pick<Todo, "title" | "completed">;
let smallTodo: PickedTodo = { title: "Task 1", completed: false }; // OK

type NoDescriptionTodo = Omit<Todo, "description">;
let noDescTodo: NoDescriptionTodo = { title: "Task 2", completed: true }; //
```

Giải thích chi tiết:

- Utility types giúp giảm việc phải viết lại các kiểu phức tạp.
- Chúng thường được sử dụng trong các ứng dụng thực tế, như xử lý dữ liệu từ API hoặc định nghĩa state trong React.

Code ví dụ chính (Tổng hợp và mở rộng)

1. Generic Function với nhiều biến kiểu và ràng buộc

```
interface Nameable { name: string; }
interface Ageable { age: number; }

function combineObjects<T extends Nameable, U extends Ageable>(obj1: T, obj2
    return { ...obj1, ...obj2 };
}

const personWithName = { name: "Alice" };
const personWithAge = { age: 30, city: "Wonderland" };
const combinedPerson = combineObjects(personWithName, personWithAge);

console.log(combinedPerson.name); // "Alice"
console.log(combinedPerson.age);  // 30
// console.log(combinedPerson.city); // Lỗi: city không được đảm bảo trong T
```

Giải thích:

- Hàm `combineObjects` kết hợp hai đối tượng, với `T` phải có `name` và `U` phải có `age`.
- Kiểu trả về `T & U` là giao của hai kiểu, chỉ chứa các thuộc tính được đảm bảo.

2. Generic Class với phương thức generic

```typescript
class Collection<T> {
    private items: T[] = [];

    add(item: T): void {
        this.items.push(item);
    }

    findFirst<U extends T>(predicate: (item: T) => item is U): U | undefined
    findFirst(predicate: (item: T) => boolean): T | undefined;
    findFirst(predicate: (item: T) => boolean): T | undefined {
        return this.items.find(item => predicate(item));
    }

    getAll(): T[] {
        return this.items;
    }
}

interface Product { id: number; name: string; price: number; }
interface Book extends Product { author: string; }

let productCollection = new Collection<Product>();
productCollection.add({ id: 1, name: "Laptop", price: 1200 });
productCollection.add({ id: 2, name: "TypeScript Programming", price: 45, au
productCollection.add({ id: 3, name: "Mouse", price: 25 });

let expensiveProduct = productCollection.findFirst(p => p.price > 100);
console.log("Expensive product:", expensiveProduct); // { id: 1, name: "Lapt

function isBook(product: Product): product is Book {
    return (product as Book).author !== undefined;
}
let firstBook = productCollection.findFirst(isBook);
console.log(`First book: ${firstBook?.name} by ${firstBook?.author}`); // "T
```

Giải thích:

- Phương thức `findFirst` hỗ trợ type guard để thu hẹp kiểu trả về (ví dụ: từ `Product` thành `Book`).

- Overloading cho phép định nghĩa hai kiểu trả về khác nhau tùy thuộc vào predicate.

3. Generic Interface cho Factory Function

```
interface Factory<T> {
    create(...args: any[]): T;
}

class User {
    constructor(public id: number, public username: string) {}
}

class Item {
    constructor(public sku: string, public name: string) {}
}

const userFactory: Factory<User> = {
    create: (id: number, username: string) => new User(id, username)
};

const itemFactory: Factory<Item> = {
    create: (sku: string, name: string) => new Item(sku, name)
};

let user = userFactory.create(1, "admin");
let item = itemFactory.create("TS-001", "TypeScript Handbook");

console.log(user); // User { id: 1, username: "admin" }
console.log(item); // Item { sku: "TS-001", name: "TypeScript Handbook" }
```

Giải thích:

- Interface `Factory` định nghĩa một hàm tạo chung cho bất kỳ lớp nào.
- `...args: any[]` cho phép truyền nhiều tham số linh hoạt, nhưng cần cẩn thận để đảm bảo đúng kiểu trong triển khai.

Danh sách bài tập

1. Trắc nghiệm: Mục đích chính của Generics

Tiêu đề: Hiểu về mục đích của Generics.

Câu hỏi: Mục đích chính của Generics trong TypeScript là gì?

- A. Tăng tốc độ biên dịch của code.
- B. Tạo các thành phần tái sử dụng với nhiều kiểu dữ liệu, đảm bảo an toàn kiểu.
- C. Giảm kích thước file JavaScript.
- D. Cho phép sử dụng kiểu `any` an toàn hơn.

Đáp án: B

Giải thích: Generics giúp viết code linh hoạt, tái sử dụng và duy trì an toàn kiểu, khắc phục hạn chế của `any`.

---

2. Code: Viết generic function `reverseArray`

Tiêu đề: Thực hành Generic Function.

Mô tả: Viết hàm `reverseArray<T>(arr: T[]): T[]` trả về mảng đảo ngược mà không thay đổi mảng gốc.

Giải pháp mẫu:

```
function reverseArray<T>(arr: T[]): T[] {
    return [...arr].reverse();
}

let numbers = [1, 2, 3, 4, 5];
let reversedNumbers = reverseArray(numbers);
```

```
console.log("Original:", numbers); // [1, 2, 3, 4, 5]
console.log("Reversed:", reversedNumbers); // [5, 4, 3, 2, 1]

let strings = ["a", "b", "c"];
let reversedStrings = reverseArray(strings);
console.log("Original:", strings); // ["a", "b", "c"]
console.log("Reversed:", reversedStrings); // ["c", "b", "a"]
```

Giải thích:

- `[...arr]` tạo bản sao mảng để tránh thay đổi mảng gốc.
- `reverse()` đảo ngược thứ tự phần tử trong bản sao.

---

3. Code: Tạo generic interface `ResponseData`

Tiêu đề: Thực hành Generic Interface.

Mô tả: Tạo interface `ResponseData<T>` với các thuộc tính `success`, `data` (kiểu `T`), và `timestamp`.

Giải pháp mẫu:

```
interface ResponseData<T> {
    success: boolean;
    data: T;
    timestamp: Date;
}

type UserProfile = { id: string; name: string; email: string };

let userResponse: ResponseData<UserProfile> = {
    success: true,
    data: { id: "user001", name: "Alice", email: "alice@example.com" },
    timestamp: new Date()
};
```

```
let productListResponse: ResponseData<string[]> = {
    success: true,
    data: ["Laptop", "Mouse", "Keyboard"],
    timestamp: new Date()
};

let errorResponse: ResponseData<null> = {
    success: false,
    data: null,
    timestamp: new Date()
};

console.log("User:", userResponse.data.name); // "Alice"
console.log("Products:", productListResponse.data.join(", ")); // "Laptop, M
console.log("Error:", errorResponse.success); // false
```

Giải thích:

- `ResponseData<T>` mô tả cấu trúc phản hồi API chung.
- `T` có thể là bất kỳ kiểu nào, từ object phức tạp đến `null`.

---

4. Code: Viết generic class `Stack<T>`

Tiêu đề: Thực hành Generic Class.

Mô tả: Viết class `Stack<T>` với các phương thức `push`, `pop`, `peek`, `isEmpty`, và `size`.

Giải pháp mẫu:

```
class Stack<T> {
    private items: T[] = [];

    push(item: T): void {
```

```
        this.items.push(item);
    }

    pop(): T | undefined {
        return this.items.pop();
    }

    peek(): T | undefined {
        return this.items.length > 0 ? this.items[this.items.length - 1] : u
    }

    isEmpty(): boolean {
        return this.items.length === 0;
    }

    size(): number {
        return this.items.length;
    }
}

let numberStack = new Stack<number>();
numberStack.push(10);
numberStack.push(20);
numberStack.push(30);
console.log("Top:", numberStack.peek()); // 30
console.log("Size:", numberStack.size()); // 3
console.log("Popped:", numberStack.pop()); // 30
console.log("Empty?", numberStack.isEmpty()); // false

let stringStack = new Stack<string>();
stringStack.push("Hello");
stringStack.push("TypeScript");
console.log("Popped:", stringStack.pop()); // "TypeScript"
console.log("Top:", stringStack.peek()); // "Hello"
```

Giải thích:

- Class `Stack` sử dụng mảng để lưu trữ phần tử.
- Các phương thức đảm bảo tính an toàn kiểu nhờ `T`.

5. Code: Viết generic function `getProperty` với ràng buộc

Tiêu đề: Thực hành Generic Constraints.

Mô tả: Viết hàm `getProperty<T, K extends keyof T>(obj: T, key: K): T[K]` lấy giá trị thuộc tính từ đối tượng.

Giải pháp mẫu:

```
function getProperty<T, K extends keyof T>(obj: T, key: K): T[K] {
    return obj[key];
}

let sampleObject = {
    id: 1,
    name: "Sample Item",
    price: 99.99,
    isActive: true
};

let itemName = getProperty(sampleObject, "name"); // string
console.log("Name:", itemName); // "Sample Item"

let itemPrice = getProperty(sampleObject, "price"); // number
console.log("Price:", itemPrice); // 99.99

let itemIsActive = getProperty(sampleObject, "isActive"); // boolean
console.log("Active:", itemIsActive); // true
// getProperty(sampleObject, "description"); // Lỗi: "description" không tồn
```

Giải thích:

- `K extends keyof T` giới hạn `key` trong các thuộc tính của `obj`.
- `T[K]` trả về kiểu chính xác của thuộc tính.

Kết luận

Generics là công cụ mạnh mẽ trong TypeScript, giúp viết code linh hoạt, tái sử dụng và an toàn kiểu. Bằng cách sử dụng generic functions, interfaces, classes, ràng buộc, và utility types, bạn có thể xây dựng các thành phần phù hợp với nhiều tình huống thực tiễn, từ quản lý dữ liệu API đến xây dựng cấu trúc dữ liệu phức tạp.

Khuyến nghị học viên:

- Thực hành các bài tập để làm quen với cách khai báo và sử dụng generics.
- Áp dụng generics trong dự án thực tế, ví dụ: xử lý dữ liệu API hoặc quản lý state trong React.
- Tìm hiểu thêm về các utility types nâng cao trong TypeScript để mở rộng khả năng.