Chapter 9: Decorators - Siêu Lập Trình (Meta-programming)

Mô tả tổng quát

Chapter này giới thiệu về **Decorators**, một tính năng mạnh mẽ trong TypeScript, cho phép thực hiện **siêu lập trình (meta-programming)**. Decorators giúp bạn thêm chú thích (annotations) hoặc sửa đổi hành vi của class, method, property, accessor, hoặc parameter tại **thời điểm thiết kế (design time)** mà không cần chỉnh sửa trực tiếp mã nguồn. Đây là một công cụ hữu ích trong việc tự động hóa các tác vụ như ghi log, xác thực dữ liệu, quản lý quyền truy cập, hoặc tiêm phụ thuộc (dependency injection).

Lưu ý quan trọng:

- Từ TypeScript 5.0 trở lên (tính đến 2025), Decorators đã được chuẩn hóa theo đề xuất ECMAScript Stage 3 và không còn là tính năng thử nghiệm. Tuy nhiên, để sử dụng Decorators, bạn vẫn cần cấu hình `tsconfig.json` đúng cách.
- Decorators thường được sử dụng trong các framework như Angular, NestJS, hoặc các thư viện yêu cầu metadata (ví dụ: TypeORM).

Tiêu đề Chapter

Decorators: Mở Rộng Chức Năng với Meta-programming

Tóm tắt lý thuyết chính

1. Khái niệm Decorators và mục đích sử dụng

Decorators là các hàm đặc biệt được gắn vào class, method, property, accessor, hoặc parameter bằng cú pháp `@expression`. Hàm `expression` này sẽ được gọi tại runtime, nhận các thông tin liên quan đến đối tượng được decorate, từ đó cho phép bạn sửa đổi hoặc mở rộng hành vi của nó.

Mục đích sử dụng Decorators

- **Logging và instrumentation**: Ghi log khi một phương thức được gọi hoặc đo thời gian thực thi.
- **Access control và authorization**: Kiểm tra quyền truy cập trước khi thực thi một phương thức.
- **Data validation**: Tự động kiểm tra tính hợp lệ của thuộc tính hoặc tham số.
- **Dependency injection**: Quản lý và tiêm các phụ thuộc vào class.
- **Behavior modification**: Thay đổi hoặc mở rộng hành vi của class hoặc các thành viên của nó.

Ví dụ thực tế

- Trong Angular, Decorators như `@Component` được dùng để định nghĩa metadata cho các thành phần giao diện.
- Trong NestJS, `@Get` hoặc `@Post` được dùng để định nghĩa các endpoint API.

2. Cấu hình Decorators trong `tsconfig.json`

Để sử dụng Decorators, bạn cần bật tùy chọn sau trong `tsconfig.json`:

```
{
  "compilerOptions": {
    "target": "ES2022", // Hoặc cao hơn để hỗ trợ Decorators chuẩn ECMAScrip
    "experimentalDecorators": true, // Bật Decorators (cần thiết cho cú pháp
    "emitDecoratorMetadata": true // Cần cho các thư viện sử dụng metadata (
  }
}
```

Giải thích:

- `experimentalDecorators`: Bật hỗ trợ Decorators. Mặc dù Decorators đã chuẩn hóa, tùy chọn này vẫn cần thiết cho các thư viện sử dụng cú pháp Decorators cũ.
- `emitDecoratorMetadata`: Kích hoạt metadata reflection, thường dùng với thư viện `reflect-metadata` để lưu trữ và truy xuất thông tin về các thành viên được decorate.

3. Các loại Decorators

TypeScript hỗ trợ 5 loại Decorators, mỗi loại áp dụng cho một phần khác nhau của code:

3.1. Class Decorators

- **Áp dụng**: Cho toàn bộ class (constructor).
- **Mục đích**: Quan sát, sửa đổi, hoặc thay thế định nghĩa class.
- **Tham số**: Nhận một tham số là constructor của class.
- **Trả về**: Nếu trả về một giá trị, giá trị đó sẽ thay thế class gốc (phải là một constructor).

Ví dụ:

```
function sealed(constructor: Function) {
  Object.seal(constructor);
  Object.seal(constructor.prototype);
  console.log(`Class ${constructor.name} đã được sealed.`);
}

@sealed
class BugReport {
  type = "report";
  title: string;

  constructor(t: string) {
    this.title = t;
  }
}

const bug = new BugReport("Crash on load");
// BugReport.prototype.newMethod = () => {}; // Lỗi vì prototype đã bị seal
```

Giải thích:

- `Object.seal` ngăn việc thêm hoặc xóa thuộc tính trên constructor hoặc prototype, đảm bảo class không bị sửa đổi ngoài ý muốn.
- Class Decorator thường được dùng để thêm metadata, logging, hoặc khóa cấu trúc class.

3.2. Method Decorators

- **Áp dụng**: Cho một phương thức của class.
- **Mục đích**: Quan sát, sửa đổi, hoặc thay thế định nghĩa phương thức.
- **Tham số**:
  - `target` : Constructor (cho static method) hoặc prototype (cho instance method).
  - `propertyKey` : Tên phương thức (string hoặc symbol).
  - `descriptor` : Property Descriptor của phương thức (`TypedPropertyDescriptor<T>`).
- **Trả về**: Nếu trả về, giá trị sẽ thay thế Property Descriptor của phương thức.

Ví dụ:

```
function enumerable(value: boolean) {
  return function (target: any, propertyKey: string, descriptor: PropertyDes
    descriptor.enumerable = value;
    console.log(`Phương thức ${propertyKey} được đặt enumerable = ${value}`)
  };
}

class Person {
  name: string;

  constructor(name: string) {
    this.name = name;
  }

  @enumerable(false)
  greet() {
    return `Hello, ${this.name}`;
  }
}

const p = new Person("Test");
for (const key in p) {
  console.log(key); // Không in ra "greet" vì enumerable = false
}
```

Giải thích:

- `descriptor.enumerable` kiểm soát việc phương thức có xuất hiện trong vòng lặp `for...in` hay không.
- Method Decorators thường được dùng để logging, kiểm tra quyền, hoặc thay đổi hành vi phương thức.

3.3. Accessor Decorators

- **Áp dụng**: Cho getter hoặc setter của một thuộc tính.
- **Mục đích**: Sửa đổi Property Descriptor của accessor.
- **Tham số**: Tương tự Method Decorator (`target` , `propertyKey` , `descriptor`).
- **Lưu ý**: Không thể áp dụng cho cả getter và setter cùng lúc; chỉ áp dụng cho cái khai báo đầu tiên.

Ví dụ:

```
function configurable(value: boolean) {
  return function (target: any, propertyKey: string, descriptor: PropertyDes
    descriptor.configurable = value;
  };
}

class Point {
  private _x: number = 0;

  @configurable(false)
  get x() {
    return this._x;
  }
}
```

Giải thích:

- `descriptor.configurable` kiểm soát việc thuộc tính có thể bị xóa hoặc cấu hình lại không.

- Accessor Decorators hữu ích khi bạn muốn kiểm soát cách truy cập hoặc sửa đổi thuộc tính.

3.4. Property Decorators

- **Áp dụng**: Cho một thuộc tính của class.
- **Mục đích**: Ghi metadata hoặc thực hiện các tác vụ liên quan đến thuộc tính.
- **Tham số**:
  - `target` : Constructor (cho static property) hoặc prototype (cho instance property).
  - `propertyKey` : Tên thuộc tính (string hoặc symbol).
- **Hạn chế**: Không thể trực tiếp sửa đổi giá trị thuộc tính qua Property Descriptor. Cần dùng getter/setter hoặc metadata.

Ví dụ đơn giản:

```
function LogProperty(target: any, propertyKey: string) {
  console.log(`Property ${propertyKey} được khai báo trên ${target.construct
}

class User {
  @LogProperty
  username: string;

  constructor(name: string) {
    this.username = name;
  }
}

new User("testuser"); // In ra: Property username được khai báo trên User
```

Giải thích:

- Property Decorators thường được dùng với `reflect-metadata` để lưu trữ thông tin về thuộc tính, ví dụ: để xác thực hoặc định dạng giá trị.

3.5. Parameter Decorators

- **Áp dụng**: Cho tham số của constructor hoặc phương thức.
- **Mục đích**: Ghi metadata hoặc thực hiện validation.
- **Tham số**:
  - `target`: Constructor (cho static method/constructor) hoặc prototype (cho instance method).
  - `propertyKey`: Tên phương thức (hoặc `undefined` nếu là constructor).
  - `parameterIndex`: Chỉ số của tham số trong danh sách tham số.
- **Thường dùng**: Kết hợp với `reflect-metadata` để lưu trữ thông tin tham số.

Ví dụ:

```
import "reflect-metadata";

const requiredMetadataKey = Symbol("required");

function Required(target: Object, propertyKey: string | symbol, parameterInd
  let existingRequiredParameters: number[] = Reflect.getOwnMetadata(required
  existingRequiredParameters.push(parameterIndex);
  Reflect.defineMetadata(requiredMetadataKey, existingRequiredParameters, ta
}

function Validate(target: any, propertyName: string, descriptor: TypedProper
  const originalMethod = descriptor.value!;
  descriptor.value = function (...args: any[]) {
    const requiredParameters: number[] = Reflect.getOwnMetadata(requiredMeta
    if (requiredParameters) {
      for (let parameterIndex of requiredParameters) {
        if (parameterIndex >= args.length || args[parameterIndex] === undefi
          throw new Error(`Tham số bắt buộc tại vị trí ${parameterIndex} bị
        }
      }
    }
    return originalMethod.apply(this, args);
  };
}
```

```
class BugReportService {
  @Validate
  updateReport(id: number, @Required title: string, @Required description: s
    console.log(`Cập nhật báo cáo ${id}: ${title} - ${description}`);
  }
}

const service = new BugReportService();
service.updateReport(1, "Crash", "App crashes on startup"); // OK
try {
  service.updateReport(2, "UI Bug", undefined as any); // Lỗi
} catch (e: any) {
  console.error(e.message);
}
```

Giải thích:

- Parameter Decorators lưu chỉ số của các tham số bắt buộc vào metadata.
- Method Decorator `@Validate` kiểm tra xem các tham số bắt buộc có giá trị hợp lệ không.

4. Thứ tự thực thi Decorators

- **Đánh giá (evaluation)**: Các hàm factory của Decorators được gọi theo thứ tự từ trên xuống dưới (top-down) và từ trái sang phải.
- **Thực thi (execution)**: Các hàm Decorators được áp dụng từ dưới lên trên (bottom-up) cho cùng một khai báo, và từ trái sang phải cho các tham số.

Ví dụ:

```
function First(): ClassDecorator {
  console.log("First(): factory được gọi");
  return function (constructor: Function) {
    console.log("First(): decorator được thực thi");
  };
}

function Second(): ClassDecorator {
```

```
  console.log("Second(): factory được gọi");
  return function (constructor: Function) {
    console.log("Second(): decorator được thực thi");
  };
}

@First()
@Second()
class ExampleClass {}
```

Kết quả:

```
First(): factory được gọi
Second(): factory được gọi
Second(): decorator được thực thi
First(): decorator được thực thi
```

Giải thích:

- Factory được gọi trước để tạo ra hàm Decorator.
- Khi áp dụng, Decorator gần class hơn (`@Second`) được thực thi trước.

5. Decorator Factories

Decorator Factory là một hàm trả về một Decorator. Nó cho phép bạn truyền tham số để tùy chỉnh hành vi của Decorator.

Ví dụ:

```
function Logger(logString: string) {
  console.log(`Logger Factory: ${logString}`);
  return function (constructor: Function) {
    console.log(logString);
    console.log(constructor);
  };
```

```
}

@Logger("LOGGING - USER")
class UserClass {
  name = "Max";
  constructor() {
    console.log("Tạo đối tượng user...");
  }
}
```

Kết quả:

```
Logger Factory: LOGGING - USER
LOGGING - USER
class UserClass { ... }
Tạo đối tượng user...
```

Giải thích:

- Factory (`Logger`) nhận tham số `logString` và trả về một hàm Decorator.
- Điều này giúp Decorator trở nên linh hoạt hơn, có thể tái sử dụng với các cấu hình khác nhau.

Code ví dụ tổng hợp

Dưới đây là một ví dụ tổng hợp, kết hợp nhiều loại Decorators để minh họa cách chúng hoạt động cùng nhau:

```
// Class Decorator: Thêm timestamp cho instance
function Timestamped<T extends { new (...args: any[]): {} }>(originalConstru
  return class extends originalConstructor {
    timestamp = new Date();
    constructor(...args: any[]) {
      super(...args);
```

```
      console.log(`[${originalConstructor.name}] Tạo instance tại: ${this.ti
    }
  };
}

// Method Decorator: Ghi log lời gọi phương thức
function LogMethodCall(target: any, propertyKey: string, descriptor: Propert
  const originalMethod = descriptor.value;
  descriptor.value = function (...args: any[]) {
    console.log(`[Method Log] Gọi ${propertyKey} với tham số: ${JSON.stringi
    const result = originalMethod.apply(this, args);
    console.log(`[Method Log] ${propertyKey} trả về: ${JSON.stringify(result
    return result;
  };
  return descriptor;
}

// Property Decorator: Giới hạn độ dài tối thiểu
function MinLength(length: number) {
  return function (target: any, propertyKey: string) {
    let value: string = target[propertyKey];

    const getter = function () {
      return value;
    };
    const setter = function (newValue: string) {
      if (newValue.length < length) {
        console.warn(`[Validation] ${propertyKey} phải dài ít nhất ${length}
      } else {
        value = newValue;
      }
    };

    Object.defineProperty(target, propertyKey, {
      get: getter,
      set: setter,
      enumerable: true,
      configurable: true,
    });
  };
}

// Parameter Decorator: Ghi log thông tin tham số
```

```
function LogParameter(target: any, propertyKey: string | symbol, parameterIn
  console.log(`[Parameter Log] Tham số ${parameterIndex} của ${String(proper
}

@Timestamped
class TaskService {
  @MinLength(3)
  defaultTaskName: string = "Untitled";

  constructor(private tasks: string[] = []) {}

  @LogMethodCall
  addTask(@LogParameter taskName: string): string[] {
    if (taskName.length === 0) {
      console.error("Tên task không được để trống.");
      return this.tasks;
    }
    this.tasks.push(taskName);
    console.log(`Task "${taskName}" được thêm.`);
    return this.tasks;
  }

  @LogMethodCall
  getTasks(): string[] {
    return [...this.tasks];
  }
}

console.log("--- Tạo instance TaskService ---");
const taskService = new TaskService(["Initial Task"]);

console.log("\n--- Thử gán tên task mặc định ---");
taskService.defaultTaskName = "My"; // Cảnh báo vì quá ngắn
console.log("Tên task mặc định sau khi gán ngắn:", taskService.defaultTaskNa
taskService.defaultTaskName = "My Awesome Task";
console.log("Tên task mặc định sau khi gán hợp lệ:", taskService.defaultTask

console.log("\n--- Thêm task ---");
taskService.addTask("Learn Decorators");
taskService.addTask(""); // Lỗi vì task rỗng

console.log("\n--- Lấy danh sách task ---");
```

```
const currentTasks = taskService.getTasks();
console.log("Danh sách task hiện tại:", currentTasks);
```

Giải thích:

- `@Timestamped` : Thêm thuộc tính `timestamp` và in thời gian tạo instance.
- `@MinLength` : Kiểm tra độ dài tối thiểu cho `defaultTaskName` .
- `@LogMethodCall` : Ghi log các lời gọi phương thức và kết quả trả về.
- `@LogParameter` : Ghi log thông tin về tham số của phương thức.

Danh sách bài tập

1. Trắc nghiệm: Decorator nào được áp dụng cho một phương thức của class?

Tiêu đề: Nhận biết loại Decorator.

Mô tả: Chọn loại decorator phù hợp với mô tả.

Câu hỏi: Loại decorator nào sau đây được áp dụng trực tiếp lên một phương thức của class trong TypeScript?

- A. Class Decorator
- B. Method Decorator
- C. Property Decorator
- D. Parameter Decorator

Đáp án: B

Giải thích: Method Decorator được áp dụng trực tiếp lên một phương thức, nhận `target` , `propertyKey` , và `descriptor` để sửa đổi hành vi của phương thức.

2. Code: Viết Class Decorator `@Version`

Tiêu đề: Thực hành Class Decorator.

Mô tả: Viết một Class Decorator factory `@Version(version: string)` để thêm thuộc tính `static version` vào class.

Giải pháp mẫu:

```
function Version(versionString: string): ClassDecorator {
  console.log(`Version factory gọi với: ${versionString}`);
  return function <TFunction extends Function>(targetConstructor: TFunction)
    console.log(`Version decorator áp dụng cho: ${targetConstructor.name}`);
    Object.defineProperty(targetConstructor, "version", {
      value: versionString,
      writable: false,
      enumerable: true,
      configurable: false,
    });
  };
}

@Version("1.0.2")
class MyAppComponent {
  name: string;
  constructor(name: string) {
    this.name = name;
  }
}

type ConstructorWithVersion = { new (...args: any[]): any; version: string }
console.log("App Version:", (MyAppComponent as ConstructorWithVersion).versi
```

Giải thích:

- Decorator thêm thuộc tính tĩnh `version` vào class.
- Sử dụng `Object.defineProperty` để đảm bảo thuộc tính không bị ghi đè.

3. Code: Viết Method Decorator `@LogExecutionTime`

Tiêu đề: Thực hành Method Decorator.

Mô tả: Viết một Method Decorator `@LogExecutionTime` để đo thời gian thực thi của phương thức.

Giải pháp mẫu:

```
function LogExecutionTime(target: any, propertyKey: string, descriptor: Prop
  const originalMethod = descriptor.value;
  descriptor.value = async function (...args: any[]) {
    const start = performance.now();
    let result: any;
    if (originalMethod.constructor.name === "AsyncFunction") {
      result = await originalMethod.apply(this, args);
    } else {
      result = originalMethod.apply(this, args);
    }
    const end = performance.now();
    console.log(`[Execution Time] ${propertyKey}: ${(end - start).toFixed(2)
    return result;
  };
  return descriptor;
}

class DataService {
  @LogExecutionTime
  fetchDataSync(id: number): string {
    let result = "";
    for (let i = 0; i < 1000000 * id; i++) {
      result += String.fromCharCode((i % 26) + 65);
    }
    return `Data for ID ${id} (length: ${result.length})`;
  }

  @LogExecutionTime
  async fetchDataAsync(id: number): Promise<string> {
    return new Promise((resolve) => {
      setTimeout(() => {
```

```
        resolve(`Async data for ID ${id}`);
      }, 50 * id);
    });
  }
}

const service = new DataService();
console.log("--- Sync Fetch ---");
service.fetchDataSync(1);
service.fetchDataSync(2);

async function testAsync() {
  console.log("\n--- Async Fetch ---");
  await service.fetchDataAsync(1);
  await service.fetchDataAsync(3);
}
testAsync();
```

Giải thích:

- `@LogExecutionTime` đo thời gian thực thi bằng `performance.now()`.
- Hỗ trợ cả phương thức đồng bộ và bất đồng bộ.

4. Code: Viết Property Decorator `@DefaultValue`

Tiêu đề: Thực hành Property Decorator (Thử thách).

Mô tả: Viết một Property Decorator `@DefaultValue(value: any)` để gán giá trị mặc định nếu thuộc tính là `undefined`.

Giải pháp mẫu:

```
function DefaultValue(defaultValue: any) {
  return function (target: any, propertyKey: string) {
    let _value: any = target[propertyKey];
    if (_value === undefined) {
      _value = defaultValue;
```

```
    }

    const getter = function () {
      return _value;
    };
    const setter = function (newValue: any) {
      _value = newValue;
    };

    if (delete target[propertyKey]) {
      Object.defineProperty(target, propertyKey, {
        get: getter,
        set: setter,
        enumerable: true,
        configurable: true,
      });
    }
  };
}

class AppConfig {
  @DefaultValue("light")
  theme: string;

  @DefaultValue(100)
  maxUsers: number = 50;

  @DefaultValue(true)
  notificationsEnabled?: boolean;

  constructor() {
    console.log("AppConfig constructor called.");
    console.log(`Initial theme: ${this.theme}`);
    console.log(`Initial maxUsers: ${this.maxUsers}`);
    console.log(`Initial notificationsEnabled: ${this.notificationsEnabled}`
  }
}

let config = new AppConfig();
console.log("--- Config Instance ---");
console.log("Theme:", config.theme); // "light"
console.log("Max Users:", config.maxUsers); // 50
console.log("Notifications:", config.notificationsEnabled); // true
```

```
config.theme = "dark";
console.log("Theme after change:", config.theme); // "dark"
```

Giải thích:

- `@DefaultValue` sử dụng `Object.defineProperty` để thay thế thuộc tính bằng getter/setter.
- Nếu thuộc tính ban đầu là `undefined`, gán giá trị mặc định.

5. Code: Viết Parameter Decorator `@Required`

Tiêu đề: Thực hành Parameter Decorator (Thử thách).

Mô tả: Viết `@Required` và `@ValidateParams` để kiểm tra các tham số bắt buộc.

Giải pháp mẫu:

```
import "reflect-metadata";

const requiredMetadataKey = Symbol("requiredParams");

function Required(target: Object, propertyKey: string | symbol, parameterInd
  let existingRequiredParameters: number[] = Reflect.getOwnMetadata(required
  existingRequiredParameters.push(parameterIndex);
  Reflect.defineMetadata(requiredMetadataKey, existingRequiredParameters, ta
  console.log(`[Required] Tham số ${parameterIndex} của ${String(propertyKey
}

function ValidateParams(target: any, propertyName: string, descriptor: Typed
  const originalMethod = descriptor.value!;
  descriptor.value = function (...args: any[]) {
    console.log(`[ValidateParams] Kiểm tra tham số cho ${propertyName}...`);
    const requiredParameters: number[] = Reflect.getOwnMetadata(requiredMeta
    if (requiredParameters) {
      for (let parameterIndex of requiredParameters) {
        if (parameterIndex >= args.length || args[parameterIndex] === undefi
```

```
          throw new Error(`Tham số bắt buộc tại vị trí ${parameterIndex} bị
        }
      }
    }
    console.log(`[ValidateParams] Tất cả tham số bắt buộc hợp lệ.`);
    return originalMethod.apply(this, args);
  };
}

class OrderService {
  @ValidateParams
  createOrder(@Required customerId: string, @Required items: string[], notes
    console.log(`Tạo đơn hàng cho khách ${customerId} với sản phẩm: ${items.
    if (notes) {
      console.log(`Ghi chú: ${notes}`);
    }
    return { orderId: Math.floor(Math.random() * 1000), status: "created" };
  }
}

const orderService = new OrderService();

try {
  console.log("\n--- Đơn hàng hợp lệ ---");
  orderService.createOrder("cust123", ["itemA", "itemB"], "Giao nhanh");
} catch (e: any) {
  console.error("Lỗi:", e.message);
}

try {
  console.log("\n--- Đơn hàng không hợp lệ (thiếu items) ---");
  orderService.createOrder("cust456", undefined as any);
} catch (e: any) {
  console.error("Lỗi:", e.message);
}
```

Giải thích:

- `@Required` lưu chỉ số tham số bắt buộc vào metadata.
- `@ValidateParams` kiểm tra các tham số bắt buộc trước khi gọi phương thức.

Mẹo và lưu ý khi sử dụng Decorators

- **Hiệu suất**: Decorators có thể làm tăng thời gian xử lý tại runtime, đặc biệt khi sử dụng nhiều hoặc trong các vòng lặp phức tạp.
- **Tính tương thích**: Đảm bảo môi trường runtime của bạn hỗ trợ Decorators (Node.js từ 16 trở lên, hoặc trình duyệt hiện đại).
- **Thư viện hỗ trợ**: Nếu dùng `reflect-metadata`, hãy cài đặt và import nó trước khi sử dụng Decorators.
- **Debugging**: Sử dụng console.log trong Decorators để theo dõi thứ tự thực thi và kiểm tra logic.

Ứng dụng thực tế

- **Trong React**: Decorators có thể được dùng để tự động gắn các hook hoặc state vào component (dù không phổ biến do cú pháp class-based ít được dùng).
- **Trong NestJS**: Decorators như `@Get`, `@Post`, hoặc `@Inject` được dùng để định nghĩa API hoặc quản lý dependency.
- **Trong TypeORM**: Decorators như `@Entity` hoặc `@Column` dùng để ánh xạ database.

Kết luận

Decorators là một công cụ mạnh mẽ để mở rộng chức năng code mà không cần thay đổi trực tiếp mã nguồn. Bằng cách hiểu rõ cách chúng hoạt động và áp dụng đúng cách, bạn có thể viết code linh hoạt, dễ bảo trì và tái sử dụng. Hãy thực hành các bài tập trên để nắm vững hơn!