# TypeScript & Nest.js Metadata Programming

MashUp Node팀 15기 세미나

윤준호

# Nest.js의 여러가지 Decorator

이 데코레이터들에는 공통점이 있어요

@Controller

@Inject

@Injectable

@Get

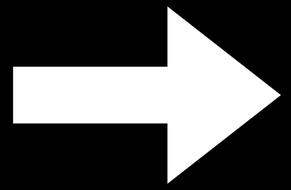
@Module

@Global

@UseGuards

@UseInterceptors

…etc



“MetaData”

# Metadata Programming

**Metadata Programming이란, 코드에 추가적인 정보(메타데이터)를 심은 뒤,
이를 기반으로 코드의 동작을 확장 및 변경하는 기법을 의미해요**

1. 선언적 프로그래밍 지원
2. 관심사 분리
3. 코드 재사용성 및 확장성 증가
4. 런타임에서 동적인 기능 개발 가능
5. 코드의 간결성

# Metadata Programming

**Plain JavaScript에서의 Metadata Programming이에요**
**예시에는 없지만 Proxy로도 Metadata Programming이 가능해요**

```
// Object
const user = {
  name: "Junho",
  age: 27,
  metadata: {
    role: ["user"],
    created: "1999-08-15",
  },
};

// Symbol
const metadataSymbol = Symbol("metadata");

class Person {
  constructor(name, age, role) {
    this.name = name;
    this.age = age;

    this[metadataSymbol] = {
      created: new Date(),
      role,
    };
  }

  get metadatas() {
    return this[metadataSymbol];
  }
}

const junho = new Person("Junho", 27, ["user"]);
console.log(junho.metadatas);
```

Symbol, Object 활용

```
// 모든 함수가 'metadata' 메소드를 가지게됨 (프로토타입 상속)
Function.prototype.metadata = function (key, value) {
  if (!this._metadata) {
    this._metadata = {};
  }
  this._metadata[key] = value;
  return this;
};

function Person(name) {
  this.name = name;
}

// Prototype에 메타데이터 추가
Person.metadata("key1", "value1").metadata("key2",
"value2");

// { key1: 'value1', key2: 'value2' }
console.log(Person._metadata);
```

Prototype 활용

```
function setMetadata(obj, metadataKey, value) {
  if (!Object.hasOwnProperty.call(obj, "metadata")) {
    Object.defineProperty(obj, "metadata", {
      value: {},
      enumerable: false,
      writable: true,
      configurable: true,
    });
  }
  obj.metadata[metadataKey] = value;
}

const person = { name: "Junho", age: 27 };
setMetadata(person, "role", ["user"]);
setMetadata(person, "created", "1999-08-15");

// [ 'name', 'age' ]
// metadata프로퍼티에 대해 Enumerable이 False이기 때문에,
metadata는 열거되지 않음
console.log(Object.keys(person));

// { role: [ 'user' ], created: '1999-08-15' }
console.log(person.metadata);
```

Property Descriptor 활용

# Metadata Programming

## TypeScript 에서의 Metadata Programming이에요

```typescript
/**
 * TypeScript Decorator
 *
 * - Class Decorator
 * - Property Decorator
 * - Method Decorator
 */
function Logger(logMessage: string) {
  return function (constructor: new (...args: any[]) => any) {
    console.log(logMessage);
  };
}

function LogMethod() {
  return function (
    target: any,
    propertyKey: string,
    descriptor: PropertyDescriptor
  ) {
    const original = descriptor.value;

    descriptor.value = function (...args: any[]) {
      console.log(`Calling ${propertyKey} with args:`, args);
      const result = original.apply(this, args);
      console.log(`Result:`, result);
      return result;
    };

    return descriptor;
  };
}

function LogProperty() {
  return function (target: any, propertyKey: string) {
    let value = target[propertyKey];

    const getter = () => {
      console.log(`Getting ${propertyKey}:`, value);
      return value;
    };

    const setter = (newValue: any) => {
      console.log(`Setting ${propertyKey} to:`, newValue);
      value = newValue;
    };

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

```typescript
@Logger("User class initialized")
class Person {
  @LogProperty()
  private name: string = "just plain text";

  @LogProperty()
  private age: number = 0;

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

  @LogMethod()
  introduce() {
    console.log(`I'm ${this.name} and I'm ${this.age} years old`);
  }

  @LogMethod()
  saySomething(message: string): string {
    return `${this.name} says: ${message}`;
  }

  @LogMethod()
  askSomething(message: string): void {
    console.log(`${this.name} asks: ${message}`);
  }
}

const user = new Person("Junho", 27);
user.saySomething("Hello");
user.askSomething("How are you?");
user.introduce();
```

Decorator를 활용

# Metadata Programming

Decorator는 reflect-metadata를 만나면

더욱 강력한 Metadata Programming 기능을 제공해요

# Metadata Programming

## Reflect Metadata

ECMAScript Decorator 제안에 기반한 메타데이터 Reflect API에요.
Object 혹은 Property에 메타데이터를 심고 검색할 수 있는 표준안에 대한것입니다.


TC39는 ECMAScript(JS표준) 사양을 발전시키는 위원회에요.

Stage 0 (Strawman): 초기아이디어
Stage 1 (Proposal): 공식제안
Stage 2 (Draft): 초안 사양
Stage 3 (Candidate): 완성된 사양
Stage 4 (Finished): 다음 ECMAScript 표준에 채택

https://github.com/tc39/proposal-decorators
https://github.com/tc39/proposal-decorator-metadata
https://rbuckton.github.io/reflect-metadata/

# Metadata Programming

**Reflect Metadata는 import를 하면 전역 Reflect 객체를 확장합니다**
**내부적으로는 WeakMap을 활용해 Metadata를 저장해요**

```
// Design-time type annotations
function Type(type) { return Reflect.metadata("design:type", type); }
function ParamTypes(...types) { return Reflect.metadata("design:paramtypes", types); }
function ReturnType(type) { return Reflect.metadata("design:returntype", type); }
```

design:type - Property Type

design:paramtypes - Method, Constructor Type

design:returntype - Method Return Type

이외에는 Custom Key에 해당되요

# Metadata Programming

## Reflect Metadata를 활용한 Class Validator 원리 살펴보기

```ts
import "reflect-metadata";

// Custom Metadata Keys
const METADATA_KEY = {
  validate: "validate",
  required: "required",
};

function Required(target: any, propertyKey: string) {
  Reflect.defineMetadata(METADATA_KEY.required, true, target,
propertyKey);
}

function Validate(validationFn: (value: any) => boolean) {
  return function (target: any, propertyKey: string) {
    Reflect.defineMetadata(
      METADATA_KEY.validate,
      validationFn,
      target,
      propertyKey
    );
  };
}

const validateEmail = (email: string) =>
  /^[^\s@]+@[^\s@]+\.[^\s@]+$/.test(email);

class User {
  @Required
  @Validate(validateEmail)
  email: string;

  name: string;

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

Reflect.defineMetadata

1. Metadata Key
2. Prototype Chain
3. Target (Prototype Chain)
4. Property (메타데이터 적용 프로퍼티)

Decorator에서 target, 즉 Prototype Chain을 받아오는점에 주목하세요

# Metadata Programming

## Reflect Metadata를 활용한 Class Validator 원리 살펴보기

```
function validateObject<T extends Record<string, any>>(obj: T):
boolean {
  const prototype = Object.getPrototypeOf(obj);
  const properties = Object.getOwnPropertyNames(obj);

  for (const property of properties) {
    const isRequired = Reflect.getMetadata(
      METADATA_KEY.required,
      prototype,
      property
    );
    if (isRequired && !obj[property]) {
      console.error(`${property} is required`);
      return false;
    }

    const validateFn = Reflect.getMetadata(
      METADATA_KEY.validate,
      prototype,
      property
    );
    if (validateFn && !validateFn(obj[property])) {
      console.error(`Property ${property}'s value is invalid`);
      return false;
    }
  }

  return true;
}

const validUser = new User("hoplin.dev@gmail.com", "Junho");
console.log(validateObject(validUser)); // true

const invalidUser = new User("Seems to be invalid email",
"Someone");
console.log(validateObject(invalidUser)); // false
```

Reflect.getMetadata

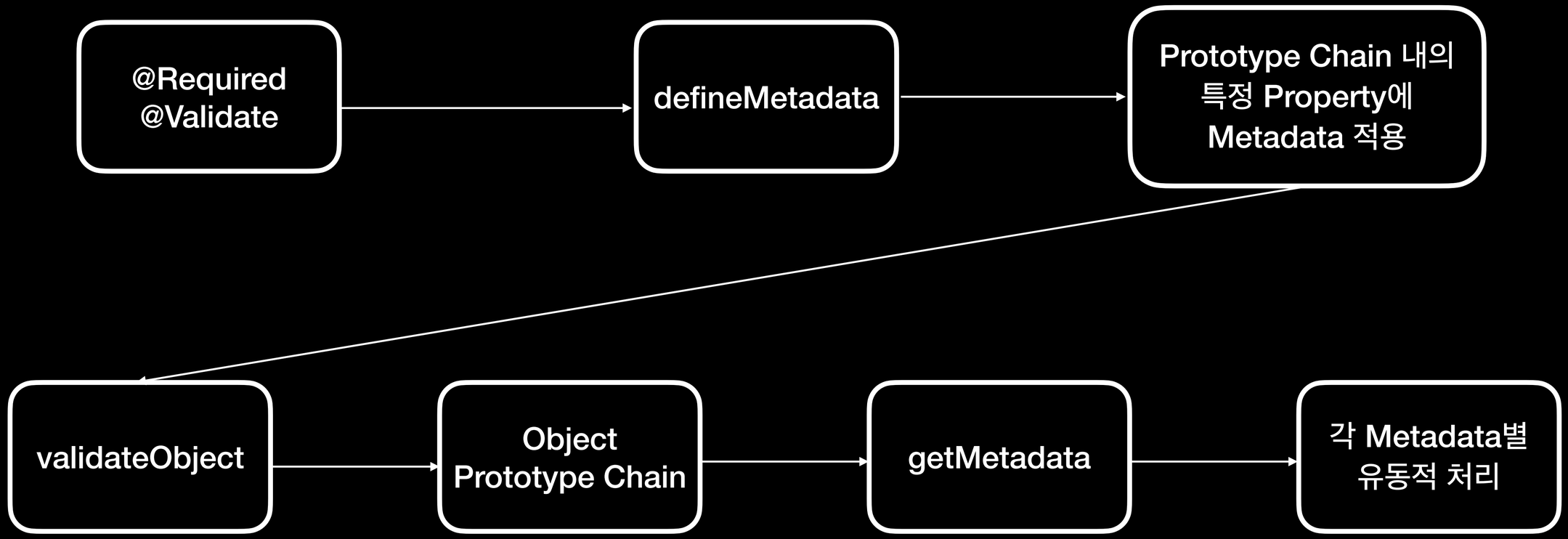
1. Metadata Key
2. Prototype Chain
3. Property Key

메타데이터 키마다 가지고 있는 value가 다를 수 있습니다.

isRequired: Boolean / validateFn: Function

# Metadata Programming

## Reflect Metadata를 활용한 Class Validator 원리 살펴보기



# Metadata Programming

## Nest.js Dependency Injection에 활용되는 @Injectable과, @Inject를 살펴봐요

```ts
export enum Scope {
  DEFAULT,
  TRANSIENT,
  REQUEST,
}

export interface ScopeOptions {
  scope?: Scope;
  durable?: boolean;
}


export function Injectable(options?: InjectableOptions):
ClassDecorator {
  return (target: object) => {
    Reflect.defineMetadata(INJECTABLE_WATERMARK, true, target);
    Reflect.defineMetadata(SCOPE_OPTIONS_METADATA, options, target);
  };
}

export function mixin<T>(mixinClass: Type<T>) {
  Object.defineProperty(mixinClass, 'name', {
    value: uid(21),
  });
  Injectable()(mixinClass);
  return mixinClass;
}
```

```ts
export function Inject(
  token?: InjectionToken | ForwardReference,
): PropertyDecorator & ParameterDecorator {
  const injectCallHasArguments = arguments.length > 0;

  return (target: object, key: string | symbol | undefined, index?: number) => {
    let type = token || Reflect.getMetadata('design:type', target, key!);
    // Try to infer the token in a constructor-based injection
    if (!type && !injectCallHasArguments) {
      type = Reflect.getMetadata(PARAMTYPES_METADATA, target, key!)?.[index!];
    }

    if (!isUndefined(index)) {
      let dependencies =
        Reflect.getMetadata(SELF_DECLARED_DEPS_METADATA, target) || [];

      dependencies = [...dependencies, { index, param: type }];
      Reflect.defineMetadata(SELF_DECLARED_DEPS_METADATA, dependencies, target);
      return;
    }
    let properties =
      Reflect.getMetadata(PROPERTY_DEPS_METADATA, target.constructor) || [];

    properties = [...properties, { key, type }];
    Reflect.defineMetadata(
      PROPERTY_DEPS_METADATA,
      properties,
      target.constructor,
    );
  };
}
```

# Metadata Programming

**JS/TS Metadata Programming이 가지는 한계점**

[ 성능 오버헤드 ]

- 객체에 메타데이터가 많을수록 메모리 사용량 증가
- Proxy객체가 많이 사용되는 경우 프로퍼티 접근, 메소드 호출마다 함수 호출 오버헤드 발생

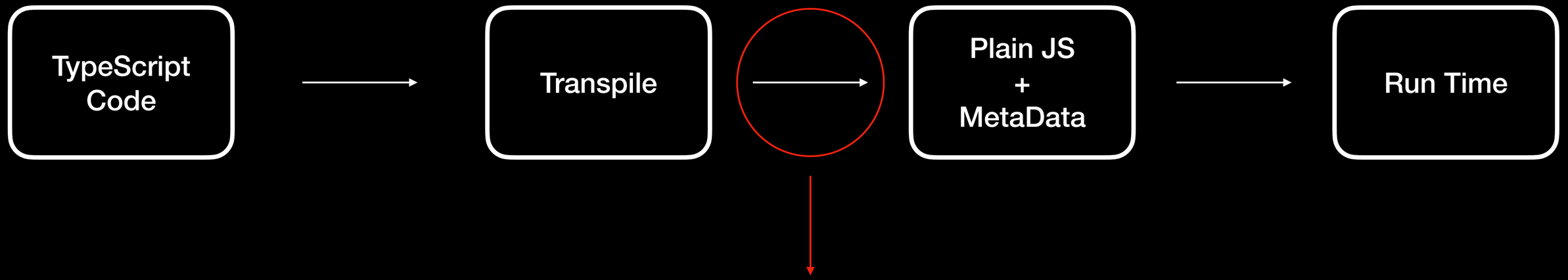
[ 표준화의 부족 ]

- Stage3이지만, 너무 오랜시간동안 고착화됨
- 상용언어는 표준 스펙인데 반해, 아직 표준화가 되지 않았으며, 조금의 변화로 많은 영향을 받을 가능성이 높음
- 이로 인해 상용언어들보다 기능이 제한적인 면이 있음

JavaScript Proxy: https://developer.mozilla.org/ko/docs/Web/JavaScript/Reference/Global_Objects/Proxy

# Nest.js속 Metadata Programming

## 1. TypeScript 코드에서 Runtime까지



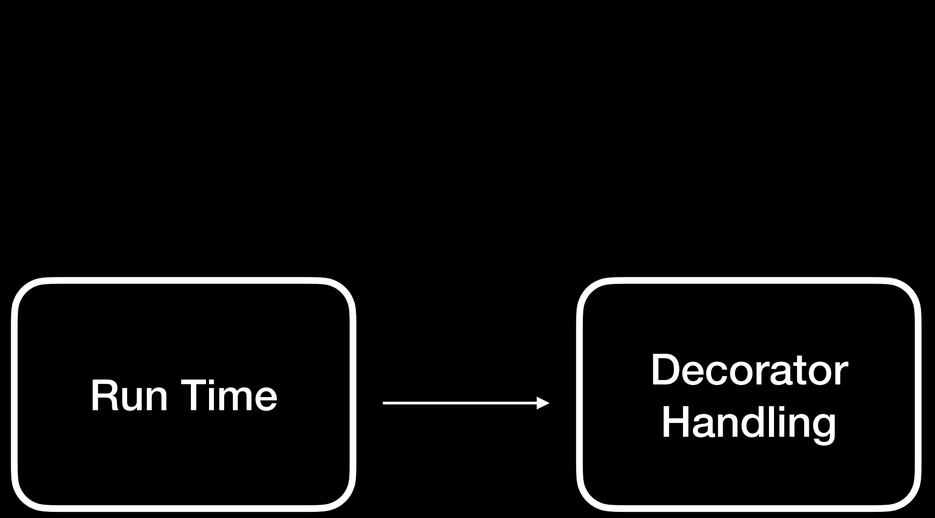

**emitDecoratorMetadata 가**
**활성화 되어있는 경우에만**

# Emit Decorator Metadata - `emitDecoratorMetadata`

Enables experimental support for emitting type metadata for decorators which works with the module `reflect-metadata`.

# Nest.js속 Metadata Programming

## 2. Runtime이후 과정




Reflection API를 통해 메타데이터 조회 및 저장

@Injectable, @Controller 등

Reflection API

MetaData를 정의하고 저장

MetaData 조회

중앙 Metadata 저장소와 상호작용

Nest.js DI Container

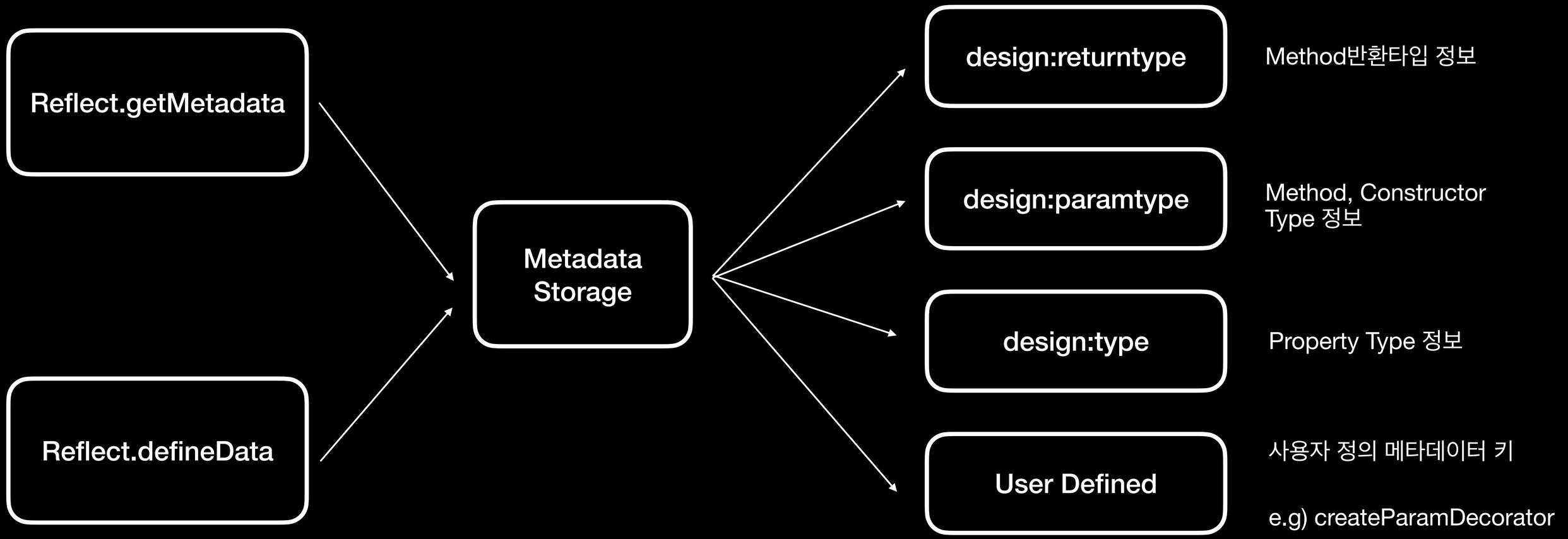
메타데이터를 분석해 의존성 관계 파악

필요한 클래스 인스턴스 생성

분석된 메타데이터 기반 인스턴스 주입

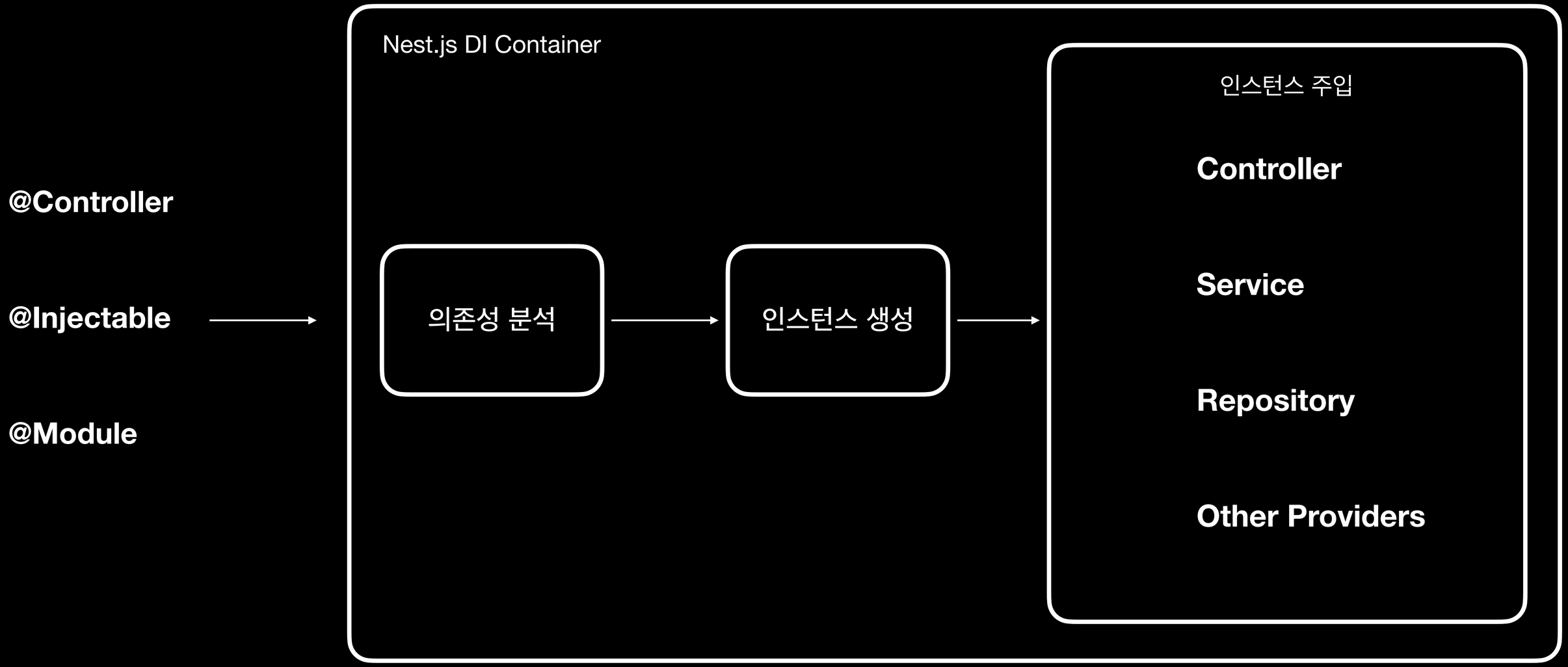
# Nest.js속 Metadata Programming

## 3. Reflection API



# Nest.js속 Metadata Programming

## 4. Nest.js DI Container



# Thank You