12장 어노테이션과 리플렉션

어노테이션

- `@`와 어노테이션 이름을 선언 앞에 넣어 어노테이션을 적용해 표시를 남긴다

예시) @Deprecated 어노테이션

- 최대 3가지 파라미터를 받는다
  1. 사용 중단 예고의 이유를 설명하는 `message`
  2. 옛 버전을 대신할 수 있는 패턴을 제시하는 `replaceWith`
  3. 점진적인 사용 중단을 지원하는 `level`

```
@Deprecated("Use remove(index) instead.", ReplaceWith("removeAt(index)")
fun remonve(index: Int) { TODO() }
```

- 이렇게 해두면 누군가 이 함수를 쓰려고 할 때 경고 메세지를 표시해줄 뿐 아니라 자동으로 코드가 고쳐지도록 퀵 픽스도 제시해준다

어노테이션의 인자

- **클래스를 어노테이션 인자로 지정**
  - `@어노테이션명(클래스명::class)` 형태
- **다른 어노테이션을 인자로 지정**
  - 인자로 어노테이션을 넣을 때에는 @를 붙여주지 않는다
  - 위의 `@Deprecated`의 `replaceWith` 파라미터도 어노테이션 인자 타입
- 배열을 인자로 지정
  - `@RequestMapping(path = ["/foo", "/bar"])` 처럼 각괄호로 쌓인 배열을 인자로 지정
  - `arrayOf` 함수로 배열 지정하는 것도 가능

어노테이션의 인자는 컴파일 시점에 알 수 있어야 한다

- 따라서 프로퍼티를 인자로 사용하려면 `const` 변경자를 붙여 컴파일 타임 상수로 만들어 컴파일 시점에 알 수 있도록 해야한다
  - `const val`은 파일의 최상위나 `object` 또는 `companion object` 안에서만 선언할 수 있다
  - 또한 `const val`은 `primitive` 타입 또는 `String`만 가능

> `const val`은 `primitive` 타입 또는 `String`만 가능한 이유?
>
> JVM 클래스 파일 내부에는 상수들을 저장하는 테이블이 있는데 여기에 문자열, 숫자, 클래스 이름, 메서드 이름 등의 정보가 들어가야 한다.
> 이 상수 풀에 저장되어야 컴파일러가 100% 확신할 수 있다
> 객체나 리스트는 값이 아닌 상태와 동작을 가진 존재이므로 메모리에 동적으로 생성되고 참조도 가능하기에 컴파일러는 이걸 하나의 고정된 값으로 확신할 수 없다

어노테이션 타깃

- 어노테이션이 참조할 수 있도록 어떤 요소에 어노테이션을 붙일지 정확하게 표시할 필요가 있다

사용 지점 타깃

- `@` 기호와 어노테이션 이름 사이에 사용 지점 타깃을 붙여 요소를 타겟팅한다

```
class CertificateManager {
    @get:JvmName("obtainCertificate") // @JvmName 어노테이션을 프로퍼티 게터
    @set:JvmName("putCertificate")    // @JvmName 어노테이션을 프로퍼티 세터
    var certificate: String = "-----BEGIN PRIVATE KEY-----"
}
```

명시적으로 프로퍼티 게터, 세터에 `@JvmName`으로 지정했으니 자바에서 이 코드를 사용할 때 지정한 이름으로 사용할 수 있다

```
class Foo {
    public static void main(String[] args) {
        var certManager = new CertificateManager();
        var cert = certManager.obtainCertificate();
        certManager.putCertificate("-----BEGIN CERTIFICATE-----");
    }
}
```

사용 지점 타깃 지원 목록

사용 지점(Target)	설명
`property`	프로퍼티 전체에 적용
`field`	backing field에 적용 (직접 메모리에 저장되는 변수)
`get`	프로퍼티 게터에 적용
`set`	프로퍼티 세터에 적용
`receiver`	확장 함수나 프로퍼티의 수신 객체(receiver)에 적용
`param`	생성자 파라미터
`setparam`	setter의 매개변수에 적용
`delegate`	위임(delegate) 인스턴스에 적용
`file`	파일 안에 선언된 최상위 함수와 프로퍼티를 담아두는 클래스
`constructor`	생성자 선언에 적용
`type`	타입 사용 위치 (예: 제네릭, 캐스팅 등)

자바의 어노테이션과 차이점

구분	Java	Kotlin
기본 적용 위치	프로퍼티에 어노테이션을 붙이면 → 필드에 자동 적용	어노테이션을 직접 `property`, `field`, `get`, `set` 등에 선택적으로 지정 가능
사용 대상 제한	클래스, 함수, 타입 등 선언부에 한정	클래스, 함수, 타입 이외의 임의의 식 허용

자바 API를 어노테이션으로 제어하기

- 자바에 노출될 API를 어노테이션으로 제어할 수 있다

어노테이션	설명
`@JvmName`	클래스, 함수, 파일 등의 이름을 Java에서 다르게 보이도록 설정
`@JvmStatic`	객체나 `COMPANION` 객체의 메서드를 정적(`static`) 메서드처럼 노출
`@JvmOverloads`	디폴트 파라미터가 있는 함수에 대해 Java에서 호출할 수 있도록 오버로딩된 여러 버전을 생성
`@JvmField`	`val` / `var` 프로퍼티를 Java에서 직접 공개된(`public`) 필드처럼 노출
`@JvmRecord`	`data class` 에 사용하면 자바 레코드 클래스로 선언 가능

어노테이션으로 JSON 직렬화 제어

직렬화?

- 객체를 저장 장치에 저장하거나 네트워크를 통해 전송하기 위해 텍스트나 이진 형식으로 변환하는 것

역직렬화?

- 텍스트나 이진 형식으로 저장된 데이터에서 원래의 객체를 생성하는 것

jkid 라이브러리 분석

- jkid는 코틀린의 `data class` 를 위한 간단한 JSON 직렬화/역직렬화 라이브러리이다
  - 젯브레인의 `kotlinx.serialization` 처럼 다양한 기능을 제공하거나 유연하지는 않지만 학습하기 좋다
- 객체를 JSON으로 직렬화할 때 jkid는 기본적으로 모든 프로퍼티를 직렬화하며 프로퍼티 이름을 키로 사용한다
  - 이 때 자바의 Jackson이나 Gson처럼 어노테이션을 활용해 직렬화 방식을 제어할 수 있다

@JsonExclude와 @JsonName

- 직렬화 대상에서 제외하고 싶을 땐 `@JsonExclude` 를 프로퍼티에 달아준다
- 프로퍼티를 표현하는 키 이름을 `@JsonName` 으로 지정한다

```
data class Person(
    @JsonName("alias") val firstName: String,
    @JsonExclude val age: Int? = null
)

fun main() {
    val person = Person("John", 30)
    println(serialize(person))
}

// 직렬화 결과
{"alias": "John"}
```

- 이 때 직렬화 대상에서 제외할 age 프로퍼티에는 반드시 기본값을 지정해야 한다
    - 역직렬화 시 실제 Person 인스턴스를 만들 때 키가 없어도 디폴트 값으로 생성자 채울 수 있도록 하기 위함

어노테이션 선언

파라미터가 없는 어노테이션 선언

```
@Target(AnnotationTarget.PROPERTY)
annotation class JsonExclude
```

- 메타데이터의 구조만 정의하기 때문에 내부테 아무 코드도 들어있을 수 없다
- 따라서 컴파일러는 어노테이션 클래스에서 본문 정의를 막는다

파라미터가 있는 어노테이션 선언

```
@Target(AnnotationTarget.PROPERTY)
annotation class JsonName(val name: String)
```

- 어노테이션 클래스의 주 생성자에 파라미터를 선언해야 한다
- 반드시 주 생성자의 모든 파라미터가 `val`이어야 한다

왜 주 생성자의 모든 파라미터가 `val` 이어야 하는가?

실제로 위의 JsonName을 컴파일해보면 인터페이스로 컴파일된다

```
@TargetallowedTargets = {AnnotationTarget.PROPERTY})
public @interface JsonName {
  String name();
}
```

- `val`로 선언한 `name` 프로퍼티는 읽기 전용으로 컴파일 타임 상수라서 어노테이션 속성을 지킨다
- `var`의 경우 getter, setter가 생기는데 어노테이션 속성은 읽기 전용이어야 해서 런타임에 변경될 수 없으므로 JVM 어노테이션 스펙에 맞지 않는다

자바 어노테이션 선언과 차이점

- 자바에는 value라는 특별한 메서드가 있고 코틀린은 name이라는 프로퍼티가 존재한다

항목	Java	Kotlin
`value`만 있을 때	`@MyAnno("abc")` 가능 `@MyAnno(value = "abc")` 가능	`@MyAnno("abc")` 가능 `@MyAnno(value = "abc")` 가능
여러 속성이 있을 때	`value`는 생략 가능 (나머지는 이름 명시 필수) `@MyAnno("abc", count = 3)`	모든 인자에 이름 지정 가능 (일반적인 함수 호출처럼) `@MyAnno(name = "abc", count = 3)`

메타 어노테이션

- 자바와 마찬가지로 메타 어노테이션으로 컴파일러가 어노테이션을 처리하는 방법을 제어할 수 있다

예시) 의존관계 주입 라이브러리가 메타 어노테이션을 사용해 타입이 동일한 여러 주입 가능한 객체를 식별한다

```
@Component
@Primary
class MySqlUserRepository : UserRepository

@Component
@Qualifier("memoryRepo")
class InMemoryUserRepository : UserRepository
```

- 스프링이 런타임에 리플렉션으로 어떤 Bean을 주입할지 판단

대표적인 메타 어노테이션

메타 어노테이션	설명
`@Target`	어노테이션을 어디에 사용할 수 있는지 정의 (`class` , `property` , `function` 등)
`@Retention`	어노테이션을 언제까지 유지할지 결정 (`SOURCE` , `BINARY` , `RUNTIME`)
`@Repeatable`	같은 어노테이션을 여러 번 사용할 수 있게 허용

@Target 메타 어노테이션

- 어노테이션을 적용할 수 있는 요소의 유형을 지정
- 지정하지 않으면 모든 선언에 적용할 수 있는 어노테이션이 된다
- jkid의 경우 프로퍼티 어노테이션만을 사용하므로 @Target을 반드시 지정한다
- 코틀린의 경우 @Target의 인자로 `AnnotationTarget` 를 사용한다

상수	설명
`CLASS`	클래스 또는 객체 선언
`ANNOTATION_CLASS`	어노테이션 클래스 자체 (직접 메타어노테이션을 만들고 싶을 경우)
`TYPE_PARAMETER`	타입 파라미터 (`T`)
`TYPE`	타입 사용 위치 (제네릭, 캐스팅 등)
`FUNCTION`	함수 선언
`PROPERTY`	프로퍼티 선언 (`val` , `var`)
`FIELD`	자바 필드 (backing field)
`LOCAL_VARIABLE`	지역 변수

VALUE_PARAMETER	함수나 생성자 매개변수
CONSTRUCTOR	생성자 선언
PROPERTY_GETTER	프로퍼티의 getter
PROPERTY_SETTER	프로퍼티의 setter
EXPRESSION	식 (표현식)
FILE	파일 전체
TYPEALIAS	타입 별칭

@Target을 `AnnotationTarget.PROPERTY`로 지정한 어노테이션을 자바에서 사용할 수 없다

- 사용하고 싶다면 두번째 @Target으로 `AnnotationTarget.FIELD` 지정해야 한다
- `@Target(AnnotationTarget.PROPERTY, AnnotationTarget.FIELD)`

@Retention 메타 어노테이션

- 정의중인 어노테이션 클래스를 어느 시점까지 유지할지 결정한다

값	설명
SOURCE	컴파일 타임까지만 존재하고 `.class` 파일엔 기록되지 않음
BINARY	`.class` 파일에는 포함되지만 런타임에 리플렉션으로 읽을 수 없음
RUNTIME	런타임에도 유지 → 리플렉션 API로 접근 가능

- 디폴트로 자바는 `SOURCE`, 코틀린은 `RUNTIME`이 지정되어 있다

어노테이션 인자로 클래스 사용

클래스의 타입 정보를 참조할 수 있는 기능이 필요할 때 어노테이션 인자로 클래스를 사용할 수 있다

- ex) 어노테이션만 보고 어떤 타입을 동적으로 처리할 때
  - `@EventListener(classes = [OrderCreatedEvent::class])`
- ex) 직렬화/역직렬화 시 타입 힌트 줄 때
  - `@JsonDeserialize(using = MyDeserializer::class)`

- ex) AOP 사용 시 포인트컷이나 인터셉터 적용될 클래스 지정할 때
  - `@Aspect(targets = [UserService::class])`
- ex) 어떤 예외를 처리할 핸들러인지 명시할 때
  - `@ExceptionHandler(value = [IOException::class])`

예시) jkid의 `@DeserializeInterface`

```
@Target(AnnotationTarget.PROPERTY)
annotation class DeserializeInterface(val targetClass: KClass<out Any>)
```

- 코틀린 클래스에 대한 참조를 저장하는 `KClass<out Any>` 타입을 인자로 받고 있다
  - `<out Any>`로 모든 타입의 하위 타입을 받을 수 있도록 제네릭 상한 지정(공변성)
  - 공변성 지정 안 하면 `@DeserializeInterface(Any::class)`만 가능했을 것

사용 예시)

```
interface Company {
    val name: String
}

data class CompanyImpl(override val name: String) : Company

data class Person(
    val name: String,

    @DeserializeInterface(CompanyImpl::class) // 실제 생성될 객체 지정
    val company: Company, // 인터페이스 타입인 Company로 지정
)
```

- JSON 역직렬화 시에 company는 인터페이스 타입이지만 실제로 생성되어야 하는 객체 정보를 @DeserializeInterface가 제공
- 이 때 : `:class` 키워드로 클래스 리터럴 전달
  - 클래스 리터럴은 클래스 자체를 나타내는 값처럼 표현한 것으로 이 클래스에 대한 정보를 참조하겠다는 의미

- 즉, : `:class`의 결과는 `KClass<T>` 타입이며 어노테이션에 넘기면 컴파일 시 자동으로 JVM의 `Class<T>`로 변환된다
- 위 예제에서는 `CompanyImpl::class`가 `KClass<CompanyImpl>`로 넘겨지고 JVM에선 `CompanyImple.class`로 처리

어노테이션 인자로 제네릭 클래스 사용

- jkid는 기본 타입이 아닌 프로퍼티를(`data class`, `List`, ..) 내포된 객체로 직렬화한다

특정 타입의 직렬화 방식을 커스텀하고 싶을 땐?

- 직렬화 로직을 커스터마이징할 수 있는 `@CustomSerializer` 어노테이션 사용
- `@CustomSerializer`는 인자로 `ValueSerializer` 인터페이스를 구현한 클래스를 전달받아 해당 타입의 직렬화 방식을 교체

`ValueSerializer<T>`란?

```
interface ValueSerializer<T> {
    fun toJsonValue(value: T): Any?
    fun fromJsonValue(jsonValue: Any?): T
}
```

- 코틀린 객체 ↔ JSON 변환 메서드 제공

`@CustomSerializer`

```
@Target(AnnotationTarget.PROPERTY)
annotation class CustomSerializer(val serializerClass: KClass<out ValueSeriali
```

- 인자로 클래스 리터럴(`::class`)를 제공받아서 KClass 타입으로 전달받눈대 반드시 `ValueSerializer`를 구현한 클래스여야 한다
- 하지만 `ValueSerializer`는 제네릭 인터페이스라서 항상 구체적인 타입 인자를 필요로 한다
  - 어노테이션 인자에서는 이 타입 인자가 구체적으로 뭔지 전혀 알 수 없기 때문에 스타 프로젝션을 사용한다

리플렉션

일반적인 코드는 컴파일 시점에 컴파일러가 코드의 이름을 기준으로 그 선언을 정적으로 찾아 빌드 시점에 존재 여부를 확인해준다.

하지만 타입과 관계없이 객체를 다뤄야 하거나 그 이름을 오직 실행 시점에만 알 수 있는 경우가 있다

- ex) 직렬화 라이브러리는 어떤 객체는 JSON으로 변환할 수 있어야 하기 때문에 컴파일 시점에 특정 타입이나 이름을 미리 알 수 없다
- ex) DI 시 프레임워크는 런타임에 주입할 의존성 타입을 동적으로 결정해야 해서 컴파일 시점엔 주입 대상 클래스를 알 수가 없다

이럴 때 리플렉션을 사용하면 실행 시점에 객체의 타입 정보를 조회해 문자열로 된 이름만으로 해당 프로퍼티나 메서드에 동적으로 접근 할 수 있다

코틀린 리플렉션 API

`org.jetbrains.kotlin:kotlin-reflect` 라이브러리 제공

타입	설명
`KClass<T>`	코틀린 클래스에 대한 메타데이터
`KCallable<R>`	함수와 프로퍼티의 공통 상위 타입
`KFunction<R>`	함수 참조 표현 파라미터, 반환 타입 등 메타정보 제공 + 호출 가능
`KProperty<R>`	프로퍼티 참조의 기본 타입(읽기 전용)
`KProperty0<R>`	인스턴스 없이 접근 가능한 프로퍼티 (ex. object, 최상위 val 등)
`KProperty1<T, R>`	인스턴스를 요구하는 프로퍼티 (일반 클래스의 멤버 프로퍼티)

`KType`	타입 정보 표현 제네릭 타입 인자나 null 가능성 등 포함
`KTypeParameter`	제네릭 타입의 타입 파라미터 표현 (ex. `<T>` 자체에 대한 정보)
`KVisibility`	가시성 정보

KClass<T: Any>

- 코틀린 클래스의 메타데이터를 표현하는 타입
- 클래스 안의 모든 선언을 열거하고 각 선언에 접근하거나 상위 클래스 조회 가능
- `::class` 연산자를 통해 실행 시점에 KClass의 인스턴스를 얻을 수 있다

예시) `KClass`의 `memberProperties` 확장함수

```
class Person(val name: String, val age: Int)

fun main() {
    val person = Person("Alice", 30)
    val kClass = person::class  // KClass<out Person> 인스턴스 반환
    println("Class name: ${kClass.simpleName}")     // KClass 이름 출력
    kClass.memberProperties         // KClass의 멤버 프로퍼티를 가져옴
        .forEach { println(it.name) }   // 각 프로퍼티 이름을 출력

    /*
        [Output]
        Class name: Person
        name
        age
     */

}
```

- 런타임에 `::class`로 `KClass<out Person>` 획득

`memberProperties`의 내부 구현

```
@SinceKotlin("1.1")
val <T : Any> KClass<T>.memberProperties: Collection<KProperty1<T, *>>
    get() = (this as KClassImpl<T>).data()
        .allNonStaticMembers.filter {
            it.isNotExtension && it is KProperty1<*, *>
        } as Collection<KProperty1<T, *>>
```

- 먼저 수신 객체 `KClass<T>`는 인터페이스이기에 내부 기능에 접근하기 위해 구체 타입인 `KClassImpl<T>`로 다운캐스팅
- `allNonStaticMembers`로 static이 아닌 모든 멤버를 얻고 그 중 확장 프로퍼티가 아니면서 인스턴스 기반 프로퍼티(`KProperty1`)인 것만 필터링
    - 여기서 `KProperty1`은 수신 객체가 반드시 존재해야 하는 일반적인 프로퍼티를 뜻함

`KClass`의 클래스명을 뜻하는 `simpleName`과 패키지포함 클래스명을 뜻하는 `qualifiedName` 프로퍼티가 `String?` 타입인 이유

- object를 사용해 익명 객체를 만들 경우 KClass의 인스턴스이지만 익명 클래스이기 때문에 이름을 뜻하는 두 프로퍼티가 null일 수 있다

KCallable<out R>

- `KCallable<R>`는 코틀린의 호출 가능한 선언을 표현하는 공통 상위 타입이다
- 즉, 함수, 프로퍼티, 게터/세터 등을 하나의 타입으로 다룰 수 있는 타입
    - `KClass`의 모든 멤버를 표현하는 `members` 프로퍼티의 타입이 `Collection<KCallable<*>>`인 이유!
- 리플렉션으로 함수 또는 프로퍼티의 게터를 대신 실행해줄 수 있다
    - 인자를 순서대로 넘기면 실제로 그 함수나 프로퍼티를 호출한 것처럼 동작한다

KCallable의 call

```
interface KCallable<out R> {
    private fun call(vararg args: Any?): R
```

```
}

fun add(x: Int, y: Int): Int = x + y

fun main() {
    val addFunc = ::add  // KFunction2<Int, Int, Int> 획득
    val result = addFunc.call(10, 20)  // 30
}
```

- 리플렉션으로 함수 또는 프로퍼티의 게터를 직접 호출할 수 있게 해주는 메서드
- 함수 인자를 vararg 리스트로 전달해서 결과 R을 받는다
- `vararg` 형태의 `Any?`로 받기 때문에 타입 안정성이 지켜지지 않아 `KCallable.parameters`에 정의된 순서와 타입을 지키지 않으면 `IllegalArgumentException` 런타임 에러가 발생한다

`KFunctionN`과 `invoke`

`KFunctionN`은 `FunctionN`을 상속받기에 `call()`말고 `invoke()`를 호출해 동일한 동작을 할 수 있다

```
// 예시) KFunction2<Int, Int, Int>
val addFunc: (Int, Int) -> Int = ::add
val resultB = addFunc.invoke(10, 20)       // invode
```

- 참고로 애초에 `invoke`라는 함수 자체가 컴파일러가 생성하기에 런타임 시점의 리플렉션에서는 사용 불가능하다

call vs invoke

위치	`KCallable` (리플렉션용)	`KFunctionN` (함수 타입)
인자 전달 방식	vararg `Any?`	고정된 개수 + 정적 타입
타입 안전성	X (런타임 검사)	O (컴파일 타임 검사)
유연성	파라미터 수 몰라도 사용 가능해서 유연	함수 시그니처를 정확히 알아야 함

[결론] 함수의 시그니처를 정확히 안다면 `invoke()`가 낫고, 리플렉션처럼 동적 호출이 필요할 때에는 `call()`이 낫다

KProperty도 사실 KCallbale의 구현체라서 call을 사용할 수 있다!

- call로 해당 프로퍼티의 게터를 호출한다

```
interface KProperty<out V> : KCallable<V>

fun main() {
    val kProperty = ::counter   // KMutableProperty0<Int> 타입의 counter 참조
    kProperty.setter.call(21)   // 리플렉션으로 setter를 호출하면서 call
    println(kProperty.get())    // get 호출해 프로퍼티 값 획득
}
```

하지만 KProperty는 프로퍼티 값을 얻는 더 좋은 get 메서드를 제공한다

```
class Person(val name: String, val age: Int)

fun main() {
    val person = Person("Alice", 30)
    val memberProperty = Person::age
    println(memberProperty.get(person))  // Outputs: 30
}
```

- `Person::age`는 `KProperty1<Person, Int>` 타입의 리플렉션 객체다
  - 즉, `Person` 타입의 인스턴스를 넘기면 그 인스턴스의 `age` 값을 알려주는 참조이다
- 이 `KProperty1` 객체를 변수에 저장한 뒤 `get(person)`을 호출하면 해당 프로퍼티의 `getter`가 호출되면서 리플렉션을 통해 `person.age` 값을 동적으로 얻을 수 있다.
  - `get()`은 컴파일 타임에 타입 체크를 하기에 안전하게 사용할 수 있다
  - 여기서 변수로 저장되는 `KProperty1<Person, Int>`는 첫번째 타입 파라미터는 수신 객체를 두번째 파라미터는 프로퍼티 타입을 표현하기 때문에 `get()` 사용 시 컴파일 타입 체크가 되어 안전한 것이다
- 즉, 리플렉션 기반 `KProperty`의 `get()`로 안전하게 값을 조회하면, 내부적으로는 해당 프로퍼티의 `call`로 `getter`를 직접 실행해 값을 반환하는 것이다
  - `call`이 `getter`를 리플렉션으로 동적 호출하는 것

- 사용자는 이런 내부 동작 상관없이 안전하게 `get()` 사용

구분	`get()`	`call()`
정의 위치	`KProperty0`, `KProperty1` 등 구체 타입에 정의	`KCallable`(부모 인터페이스)에서 상속됨
호출 의미	프로퍼티 값을 읽음	내부적으로 getter를 리플렉션으로 호출
타입 안전성	컴파일 타임 타입 체크	런타임에 오류 발생 가능

JKID 라이브러리 분석

직렬화 과정

1. **`serialize()`로 직렬화 시 결과 JSON을 `StringBuilder`로 구축을 시작한다**

```
fun serialize(obj: Any): String = buildString { serializeObject(obj) }

private fun StringBuilder.serializeObject(obj: Any) {
    append( /* JSON 형식으로 obj 직렬화 */ )
}
```

- 이 때 수신 객체를 따로 지정하지 않고 `append`를 간결하게 사용하기 위해 확장함수를 구현해서 사용한다
- 원래라면 `StringBuilder`도 같이 파라미터로 넘겨줘야 했을 것

2. **`serializeObject()`로 객체 직렬화 시작**

```
private fun StringBuilder.serializeObject(obj: Any) {
    obj.javaClass.kotlin.memberProperties
            .filter { it.findAnnotation<JsonExclude>() == null }
            .joinToStringBuilder(this, prefix = "{", postfix = "}") {
                serializeProperty(it, obj)
```

```
        }
}
```

- 리플렉션으로 직렬화 대상의 멤버 프로퍼티 획득
- `@JsonExclude`가 붙어있지 않은 프로퍼티만 필터링

```
inline fun <reified T> KAnnotatedElement.findAnnotation(): T?
        = annotations.filterIsInstance<T>().firstOrNull()
```

  - 리플렉션 대상(`KAnnotatedElement`)에 붙은 어노테이션 중 T에 해당하는 어노테이션을 찾는 함수
  - 런타임에 객체의 실제 타입을 검사해서 일치하는 것만 걸러내는 `filterIsInstance<T>()`를 사용하기 위해 `inline` + `reified` 조합 사용한 모습..!
- `joinToStringBuilder()`로 각 프로퍼티들을 직렬화한 뒤 "`{ ... }`"로 감싸기

3. `serializeProperty()`로 각 프로퍼티 직렬화 시작

```
private fun StringBuilder.serializeProperty(
        prop: KProperty1<Any, *>, obj: Any
) {
    val jsonNameAnn = prop.findAnnotation<JsonName>()
    val propName = jsonNameAnn?.name ?: prop.name
    serializeString(propName)
    append(": ")

    val value = prop.get(obj)
    val jsonValue = prop.getSerializer()?.toJsonValue(value) ?: value
    serializePropertyValue(jsonValue)
}
```

- `@JsonName` 여부에 따라 프로퍼티 키 결정
- 키를 `serializeString()`로 이스케이프 처리하면서 JSON 문자열로 안전하게 변환
- 해당 프로퍼티의 값을 `get()`으로 안전하게 추출한 뒤 `getSerializer()`로 커스텀 직렬화기 적용

```
fun KProperty<*>.getSerializer(): ValueSerializer<Any?>? {
    val customSerializerAnn = findAnnotation<CustomSerializer>() ?: return
    val serializerClass = customSerializerAnn.serializerClass

    val valueSerializer = serializerClass.objectInstance
            ?: serializerClass.createInstance()
    @Suppress("UNCHECKED_CAST")
    return valueSerializer as ValueSerializer<Any?>
}
```

- `@CustomSerializer`가 붙어있으면 지정된 `serializerClass`를 인스턴스화한 뒤 `ValueSerializer`로 캐스팅해서 반환
- `object`(싱글톤 객체)인 경우 이미 존재하는 인스턴스 재사용
- `KClass`(일반 클래스)인 경우 매번 새로운 인스턴스 생성해서 사용

4. **`serializePropertyValue()`로 실제 값 직렬화**

```
private fun StringBuilder.serializePropertyValue(value: Any?) {
    when (value) {
        null -> append("null")
        is String -> serializeString(value)
        is Number, is Boolean -> append(value.toString())
        is List<*> -> serializeList(value)
        else -> serializeObject(value)
    }
}
```

- JSON 규칙에 맞게 값 직렬화 (객체의 경우 `serializeObject()` 재귀 호출)

역직렬화 과정

1. **`deserialize()`로 역직렬화 시작**

```
inline fun <reified T: Any> deserialize(json: String): T
```

- 역직렬화를 하려면 실행 시점에 타입 파라미터에 접근해야 한다
- 따라서 `inline` + `reified`로 타입 정보를 실행 시점까지 유지해야 한다

2. **Lexer로 JSON 토큰 리스트화**

- Lexer는 Parser가 문자열을 읽기 전 JSON을 미리 토큰 리스트로 분석한다
  - 문자 토큰 (`{`, `}`, `[`, `]`, `,`, `:`)
  - 값 토큰 (`String`, `Bool`, `Long`, `Double`)

3. **Parser로 문법 분석**

- Lexer로 쪼갠 토큰을 바탕으로 JSON 구조를 분석한 뒤 키-값 쌍과 배열로 변환하는 과정
- 현재 역직렬화하는 중인 객체나 배열을 추적해 새로운 프로퍼티를 발경할 때마다 그 프로퍼티에 해당하는 `JSonObject`의 함수를 호출한다

```
interface JsonObject {
    fun setSimpleProperty(propertyName: String, value: Any?)
    fun createObject(propertyName: String): JsonObject
    fun createArray(propertyName: String): JsonObject
}
```

  - `setSimpleProperty()`: 단일 값(String, 숫자, Bool, null 등)을 키에 매핑
    - ex) `“key”: value`
  - `createObject()`: 중첩된 `{ ... }` 구조가 시작될 때 호출 → 해당 키에 맞는 `JsonObject` 생성
    - ex) `"key": { ... }`
  - `createArray()`: 배열 `[ ... ]` 구조가 시작될 때 호출
    - ex) `“key”: [ ... ]`
- Parser 작업이 끝나면 `JsonObject` 트리 구조가 완성될 것
  - 예시)

```
JsonObject(
  mapOf(
    "name" to JsonString("Sam"),
    "age" to JsonNumber(30),
    "languages" to JsonArray(listOf(
      JsonString("Java"),
      JsonString("Kotlin")
    )),
    "active" to JsonBoolean(true)
  )
)
```

4. **파싱한 결과로 객체 생성 (역직렬화 컴포넌트)**

실제 역직렬화 과정은 `deserialize()`에서 이뤄진다

```
fun <T: Any> deserialize(json: Reader, targetClass: KClass<T>): T {
    val seed = ObjectSeed(targetClass, ClassInfoCache())
    Parser(json, seed).parse()
    return seed.spawn()
}
```

1. 먼저 클래스의 프로퍼티 객체들을 저장해둘 중간 컨테이너인 ObjectSeed를 생성한다
    - Parser가 JSON을 분석해서 `JsonObject`의 함수들을 호출하면 그 구조에 따라 JSON 데이터를 담아둘 중간 컨테이너가 필요하다
        - JKID 객체를 생성한 다음에 프로퍼티를 설정하는 것을 지원하지 않기 때문에 수집해두고 나중에 생성자 호출로 객체를 생성해야 한다
    - JKID는 `Seed` 인터페이스와 그 구현체들로 이 중간 컨테이너를 구성한다

```
interface Seed: JsonObject {
    val classInfoCache: ClassInfoCache

    fun spawn(): Any?

    fun createCompositeProperty(propertyName: String, isList: Boolean): Js
```

```
    override fun createObject(propertyName: String) = createCompositePro

    override fun createArray(propertyName: String) = createCompositeProp
}
```

- JSON 파싱 도중 데이터를 임시 저장할 중간 컨테이너 역할
  - JSON 구조에 따라 3가지 구현체 중 하나가 쓰인다
    - `ObjectSeed`: 일반 객체 (예: `Person`)
    - `ObjectListSeed`: 객체 리스트 (예: `List<Person>`)
    - `ValueListSeed`: 기본형 리스트 (예: `List<Int>`)
- 모든 정보를 수집한 뒤에 `spawn()` 메서드로 실제 코틀린 객체로 변환한다
- 이 때 `ClassInfoCache`를 사용해서 클래스의 프로퍼티 정보를 `ClassInfo`에 캐싱해 성능을 향상시킨다

2. 실제로 Parser로 파싱한다
   - 여기서 Parser는 내부에서 Lexer를 통해 JSON 문자열을 토큰 단위로 읽고 구조에 따라 `seed`에 값을 전달한다
   - `ObjectSeed` 예시

```
class ObjectSeed<out T: Any>(
        targetClass: KClass<T>,
        override val classInfoCache: ClassInfoCache
) : Seed {

        // 대상 클래스에 대한 메타데이터를 포함한 ClassInfo 저장
    private val classInfo: ClassInfo<T> = classInfoCache[targetClass]

        // 단순 값 저장
    private val valueArguments = mutableMapOf<KParameter, Any?>()

    // 중첩 객체나 배열을 위한 하위 seed 저장
    private val seedArguments = mutableMapOf<KParameter, Seed>()

        // 최종적으로 생성자에 넘길 인자 맵 (단순 값과 spawn() 호출로 얻은
```

```
    private val arguments: Map<KParameter, Any?>
        get() = valueArguments + seedArguments.mapValues { it.value.sp

        // JSON 키-값 쌍 중 단순 값(String, Int 등)을 만났을 때 호출
    override fun setSimpleProperty(propertyName: String, value: Any?)
            // 1. JSON 키에 대응되는 생성자 파라미터(KParameter)를 classIn
        val param = classInfo.getConstructorParameter(propertyName)

        // 2. 커스텀 역직렬화기나 타입 변환 적용해 값을 변환한 뒤 맵에 저장
        valueArguments[param] = classInfo.deserializeConstructorArgum
    }

        // JSON 키-값 쌍 중 중첩된 객체나 배열일 경우 호출
    override fun createCompositeProperty(propertyName: String, isList:
            // 1. JSON 키에 대응되는 생성자 파라미터(KParameter)를 classInfo
        val param = classInfo.getConstructorParameter(propertyName)

        // 2. 역직렬화기 선택
        val deserializeAs = classInfo.getDeserializeClass(propertyName)

        // 3. Seed 구현체 생성(ObjectSeed, ObjectListSeed, ValueListSeed)
        val seed = createSeedForType(
            deserializeAs ?: param.type.javaType, isList)
        return seed.apply { seedArguments[param] = this }
    }

        // 파싱이 다 끝나면 호출해 실제 코틀린 객체 생성
    override fun spawn(): T = classInfo.createInstance(arguments)
}
```

3. Parser가 모든 JSON 구조를 Seed에 넘긴 후 spawn()을 호출해 진짜 코틀린 객체를 생성한다

```
override fun spawn(): T = classInfo.createInstance(arguments)

fun createInstance(arguments: Map<KParameter, Any?>): T {
    ensureAllParametersPresent(arguments)
```

```
        return constructor.callBy(arguments)
    }
```

- 이 때 내부적으로 코틀린 리플렉션 API `callBy()`를 사용해 매핑된 파라미터로 생성자를 호출한다

코틀린 리플렉션 API callBy()?

```
interface KCallable<out R> : KAnnotatedElement {
    public fun call(vararg args: Any?): R
    public fun callBy(args: Map<KParameter, Any?>): R
}
```

- `KCallable`의 `call`은 디폴트 파라미터 값을 지원하지 않는다
  - 역직렬화 시 `call`을 사용하면 생성할 객체에 디폴트 생성자 파라미터가 있떠라도 다시 꼭 지정해줘야 할 것이다
- `callBy`은 디폴트 파라미터를 지원하기에 역직렬화 시 전달받은 `arguments`에 파라미터가 없으면 기본값으로 정의된 값을 사용한다

`callBy()`로 전달받은 `args` 맵의 각 타입이 생성자 파라미터 타입과 맞지 않으면 `IllegalArgumentException`가 발생한다

- 즉, `{"age": "30"}` 으로 전달받았을 때 생성자는 `age`가 `Int` 타입이고 JSON은 문자열 "30"이라서 타입이 안 맞는다
- 따라서 타입을 검사해서 미리 바꿔줘야 하는데 이 때 필요한 게 `ValueSerializer`이다
- `serializerForType`을 통해 타입에 맞는 적절한 `ValueSerializer`를 찾는다

```
fun serializerForType(type: Type): ValueSerializer<out Any?>? =
        when (type) {
            Byte::class.java, Byte::class.javaObjectType → ByteSerializer
            Short::class.java, Short::class.javaObjectType → ShortSerializer
            Int::class.java, Int::class.javaObjectType → IntSerializer
            Long::class.java, Long::class.javaObjectType → LongSerializer
            Float::class.java, Float::class.javaObjectType → FloatSerializer
            Double::class.java, Double::class.javaObjectType → DoubleSerializ
            Boolean::class.java, Boolean::class.javaObjectType → BooleanSeria
```

```
            String::class.java → StringSerializer
            else → null
        }
```

- 즉, 위의 경우 생성자 파라미터가 `Int` 니까 `IntSerializer`를 사용해서 `Int` 타입으로 변환한다

```
object ByteSerializer : ValueSerializer<Byte> {
    override fun fromJsonValue(jsonValue: Any?) = jsonValue.expectNu
    override fun toJsonValue(value: Byte) = value
}

private fun Any?.expectNumber(): Number {
    if (this !is Number) throw JKidException("Expected number, was: $t
    return this
}
```

`ClassInfoCache`를 사용하는 이유

- 직렬화와 역직렬화에 사용되는 어노테이션들은 파라미터가 아닌 프로퍼티에 적용된다.
- 하지만 역직렬화 시엔 프로퍼티가 아닌 생성자 파라미터를 다뤄야 하는데 그럼 매번 그 파라미터에 해당하는 프로퍼티를 찾아야 해서 코드가 느려질 수 있다
- 따라서 대상 클래스별로 한 번만 검색하고 캐싱해두는 것이다

`ClassInfo` 분석

1. 코틀린 객체 생성 시 주 생성자로 직렬화/역직렬화하기 때문에 반드시 존재해야 하므로 체크

```
class ClassInfo<T : Any>(cls: KClass<T>) {
    private val constructor = cls.primaryConstructor
            ?: throw JKidException("Class ${cls.qualifiedName} doesn't have a pri
```

2. 매핑용 캐시 맵들

```
private val jsonNameToParamMap = hashMapOf<String, KParameter>()
private val paramToSerializerMap = hashMapOf<KParameter, ValueSerialize
private val jsonNameToDeserializeClassMap = hashMapOf<String, Class<o
```

역할	설명
`jsonNameToParamMap`	JSON 키 → 생성자 파라미터로 매핑
`paramToSerializerMap`	생성자 파라미터 → ValueSerializer 인스턴스
`jsonNameToDeserializeClassMap`	JSON 키 → 구현 클래스 (인터페이스 역직렬화 지원용)

3. 대상 클래스 생성자 파라미터 호출해 캐시 초기화

```
    init {
    constructor.parameters.forEach { cacheDataForParameter(cls, it) }
}

private fun cacheDataForParameter(cls: KClass<*>, param: KParameter) {
        // 일치하는 파라미터 찾기
    val paramName = param.name
            ?: throw JKidException("Class $className has constructor paramet
    val property = cls.declaredMemberProperties.find { it.name == paramNa

    // @JsonName 붙어 있으면 그 이름으로 JSON 매핑
    val name = property.findAnnotation<JsonName>()?.name ?: paramName
    jsonNameToParamMap[name] = param

            // @DeserializeInterface 처리(역직렬화 시 구체 클래스 지정)
    val deserializeClass = property.findAnnotation<DeserializeInterface>()?.t
    jsonNameToDeserializeClassMap[name] = deserializeClass

            // ValueSerializer 지정
    val valueSerializer = property.getSerializer()  // 커스텀 시리얼라이즈 있으면 
            ?: serializerForType(param.type.javaType)
            ?: return
    paramToSerializerMap[param] = valueSerializer
}
```

4. JSON 키로부터 생성자 파라미터/타입을 얻기 위한 getter

```
    fun getConstructorParameter(propertyName: String): KParameter = jsonN
        ?: throw JKidException("Constructor parameter $propertyName is not

    fun getDeserializeClass(propertyName: String) = jsonNameToDeserializeCla
```

5. 역직렬화 시 값 변환하는 메서드

- `ValueSerializer`가 있으면 타입 체크 및 변환
- 없으면 `validateArgumentType()`으로 단순 검사

```
fun deserializeConstructorArgument(param: KParameter, value: Any?): Any?
        // ValueSerializer로 타입 체크 및 변환
    val serializer = paramToSerializerMap[param]
    if (serializer != null) return serializer.fromJsonValue(value)

            // 단순 타입 검사
    validateArgumentType(param, value)
    return value
}

private fun validateArgumentType(param: KParameter, value: Any?) {
        // nullable 체크
    if (value == null && !param.type.isMarkedNullable) {
        throw JKidException("Received null value for non-null parameter ${par
    }
    // 타입 정확히 일치하는지 검사
    if (value != null && value.javaClass != param.type.javaType) {
        throw JKidException("Type mismatch for parameter ${param.name}: "
                "expected ${param.type.javaType}, found ${value.javaClass}")
    }
}
```

7. callBy를 통해 진짜 코틀린 객체로 생성해주는 메서드

```
fun createInstance(arguments: Map<KParameter, Any?>): T {
    ensureAllParametersPresent(arguments)

    // callBy이기 때무넹 디폴트 파라미터를 지원
    return constructor.callBy(arguments)
}

    // 필수 파라미터가 빠졌는지 검사
private fun ensureAllParametersPresent(arguments: Map<KParameter, Any?
    for (param in constructor.parameters) {
            // 기본값도 없고 null 허용도 안 되는 파라미터인데 없으면 예외 터뜨림
        if (arguments[param] == null && !param.isOptional && !param.type.isMa
            throw JKidException("Missing value for parameter ${param.name}")
        }
    }
}
```