# Table of Contents


Introduction 1.1

Chapter 2. Fast Track to OOP - Classes and Interfaces 1.2

- Introduction.md 1.2.1
- Implementing object-oriented design using classes 1.2.2
- Using inner classes 1.2.3

Chapter 3. Modular Programming 1.3

- Introduction 1.3.1

Chapter 4. Going Functional 1.4

Chapter 5. Stream Operations and Pipelines 1.5

Chapter 6. Database Programming 1.6

Chapter 7. Concurrent and Multithreaded Programming 1.7

Chapter 8. Better Management of the OS Process 1.8

Chapter 9. GUI Programming Using JavaFX 1.9

Chapter 10. RESTful Web Services Using Spring Boot 1.10

Chapter 11. Networking 1.11

Chapter 12. Memory Management and Debugging 1.12

Chapter 13. The Read Evaluate Print Loop (REPL) Using JShell 1.13

Chapter 14. Scripting Using Oracle Nashorn 1.14

Chapter 15. Testing 1.15

# Java 9 Cookbook

# Chapter 2 Fast Track to OOP - Classes and Interfaces

이 장에서는 다음과 같은 내용을 다룹니다.

- 클래스를 사용하여 객체 지향 디자인 구현
- 내부 클래스 사용
- 상속 및 컴포지션을 사용하여 디자인을 확장 가능하게 만들기
- 인터페이스에 코딩
- 기본 및 정적 메서드를 사용하여 인터페이스 만들기
- 개인 메소드로 인터페이스 만들기
- enum을 사용하여 상수 엔티티 표현
- @Deprecated 주석을 사용하여 API를 비추천
- Javadocs에서 HTML5 사용하기

Implementing object-oriented design using classes Using inner classes Using inheritance and composition to make the design extensible Coding to an interface Creating interfaces with default and static methods Creating interfaces with private methods Using enums to represent constant entities Using the @Deprecated annotation to deprecate APIs Using HTML5 in Javadocs

[[2_1 Introduction]] [[2_2 Implementing object-oriented design using classes]] [[2_3 Using inner classes|내부 클래스]] [[2_4 Using inheritance and composition to make the design extensible|상속 및 컴포지션을 사용하여 디자인을 확장 가능하게 만들기]] [[2_5 Coding to an interface|인터페이스에 코딩]] [[2_6 Creating interfaces with default and static methods|기본 및 정적 메서드를 사용하여 인터페이스 만들기]] [[2_7 Creating interfaces with private methods|private 메소드로 인터페이스 만들기]] [[2_8 Using enums to represent constant entities|enum을 사용하여 상수 엔티티 표현]] [[2_9 Using the @Deprecated annotation to deprecate APIs|@Deprecated 주석을 사용하여 API를 비추천]] [[2_10 Using HTML5 in Javadocs|Javadocs에서 HTML5 사용하기]]

# Introduction

이 장에서는 OOP의 구성 요소에 대해 간략하게 소개하고 Java 8 및 Java 9의 이러한 구성 요소의 새로운 개선 사항에 대해 다룹니다. 또한 적절한 OOD (Object-Oriented Design) 관행에 대해서도 다룰 것입니다.

레시피 전반에 걸쳐 새로운 (Java 8 및 Java 9에서 소개 된) 향상된 기능을 사용하고 특정 코드 예제를 사용하여 OOD의 개념을 정의 및 보여 주며 더 나은 코드 문서화를 위한 새로운 기능을 제공합니다.

책이나 인터넷에서 OOD에 대한 기사와 실용적인 조언을 읽는 데 많은 시간을 할애 할 수 있습니다. 이 활동 중 일부는 어떤 사람들에게는 유익 할 수 있습니다. 그러나 우리의 경험에서 OOD를 잡는 가장 빠른 방법은 자신의 코드에서 초기 원칙을 시험해 보는 것입니다. 이것이 바로 이 장의 목표입니다. OOD 원칙을 보고 사용할 수 있는 기회를 제공하여 공식적인 정의가 즉시 의미를 갖도록 합니다.

잘 쓰여진 코드의 주된 기준 중 하나는 의도를 표현하는 명확성입니다. 잘 동기 부여되고 명확한 설계는 이를 달성하는 데 도움이됩니다. 이 코드는 컴퓨터에 의해 실행되지만 인간에 의해 유지되고 확장됩니다. 이것을 염두에 두면 당신이 작성한 코드의 장수와 심지어 감사와 감사의 표현까지 가능할 것입니다.

이 장에서는 5가지 기본 OOD 개념을 사용하는 방법을 학습합니다.

- 객체 / 클래스 - 데이터와 프로 시저를 함께 유지
- 캡슐화 - 데이터 및 / 또는 절차 숨기기
- 상속 - 다른 클래스 데이터 및 / 또는 프로 시저 확장
- 인터페이스 - 유형 구현 및 코딩 지연
- Polymorphism - 부모 클래스 참조가 자식 클래스 객체를 참조하는 데 사용될 때 모든 확장에 기본 클래스 유형 사용

이 개념은이 장에서 소개하는 코드 스니 j에서 정의되고 시연됩니다. 인터넷을 검색하면 방금 논의한 5 가지 사항에서 다른 많은 개념과 추가 사항을 도출 할 수 있습니다.

다음 텍스트는 OOD에 대한 사전 지식이 필요하지 않지만 Java로 코드를 작성한 경험이 도움이 될 것입니다. 코드 샘플은 Java 9와 완벽하게 호환됩니다. 더 나은 이해를 위해 제시된 예제를 실행 해보십시오.

또한 이 장의 팁과 권장 사항을 팀 경험의 맥락에서 필요에 맞게 적용하는 것이 좋습니다. 동료들과 새로 발견 한 지식을 공유하고 현 프로젝트에 대해 설명 된 원칙을 귀하의 도메인에 어떻게 적용 할 수 있는지 논의하십시오.

This chapter gives you a quick introduction to the components of OOP and covers the new enhancements in these components in Java 8 and Java 9. We will also try to cover a few good object-oriented design (OOD) practices wherever applicable.

Throughout the recipes, we will use the new (introduced in Java 8 and Java 9) enhancements, define and demonstrate the concepts of OOD using specific code examples, and present new capabilities for better code documentation.

One can spend many hours reading articles and practical advice on OOD in books and on the Internet. Some of this activity can be beneficial for some people. But, in our experience, the fastest way to get hold of OOD is to try its principles early in your own code. This is exactly the goal of this chapter: to give you a chance to see and use the OOD principles so that the formal definition makes sense immediately.

One of the main criteria of well-written code is its clarity of expressing its intent. A well-motivated and clear design helps achieve this. The code is run by a computer, but it is maintained and extended by humans. Keeping this in mind will assure longevity of the code written by you and perhaps even a few thanks and mentions with appreciation.

In this chapter, you will learn how to use the five basic OOD concepts:

Object/Class - Keeping data and procedures together Encapsulation - Hiding data and/or procedures Inheritance - Extending another class data and/or procedures Interface - Deferring the implementation and coding for a type Polymorphism - Using the base class type for all its extensions when a parent class reference is used to refer to a child class object

These concepts will be defined and demonstrated in the code snippets presented in this chapter. If you search the Internet, you may notice that many other concepts and additions to them can be derived from the five points we just discussed.

Although the following text does not require prior knowledge of OOD, some experience of writing code in Java would be beneficial. The code samples are fully functional and compatible with Java 9. For better understanding, we recommend that you try to run the presented examples.

We also encourage you to adapt the tips and recommendations in this chapter to your needs in the context of your team experience. Consider sharing your new-found knowledge with your colleagues and discuss how the described principles can be applied to your domain, for your current project.

# Chapter 3 Modular Programming


- [[3_1 Introduction|소개]]
- [[3_2 Using jdeps to find dependencies in a Java application|jdeps를 사용하여 Java 응용 프로그램에서 종속성 찾기]]
- [[3_3 Creating a simple modular application|간단한 모듈 응용 프로그램 만들기]]
- [[3_4 Creating a modular JAR|모듈 형 JAR 생성하기]]
- [[3_5 Using a module JAR with pre-JDK 9 applications]]
- [[3_6 Bottom-up migration]]
- [[3_7 Top-down migration]]
- [[3_8 Using services to create loose coupling between consumer and provider modules]]
- [[3_9 Creating a custom modular runtime image using jlink]]
- [[3_10 Compiling for older platform versions]]
- [[3_11 Creating multirelease JARs]]
- [[3_12 Using Maven to develop a modular application]]

# Introduction

모듈식 프로그래밍을 사용하면 코드를 독립적인 응집력있는 모듈로 구성 할 수 있습니다.이 모듈을 결합하면 원하는 기능을 얻을 수 있습니다. 이렇게 하면 다음과 같은 코드를 작성할 수 있습니다.

- 모듈이 특정 목적으로 구축되기 때문에 응집력이 높아지므로 거기에 있는 코드는 특정 목적을 충족시키는 경향이 있습니다.
- 모듈은 다른 모듈에서 사용할 수있는 API와 만 상호 작용할 수 있기 때문에 캡슐화됩니다.
- 발견 가능성은 개별 유형이 아닌 모듈을 기반으로하므로 신뢰할 수 있습니다. 즉, 모듈이 없으면 종속 모듈이 발견 할 때까지 종속 모듈을 실행할 수 없습니다. 이렇게하면 런타임 오류를 방지하는 데 도움이됩니다.
- 느슨한 결합. 서비스 인터페이스를 사용하면 모듈 인터페이스와 서비스 인터페이스 구현을 느슨하게 결합 할 수 있습니다.

따라서 코드를 설계하고 구성 할 때 고려해야 할 단계는 이제 모듈과 코드 및 모듈 내에서 코드가 구성되는 패키지로 들어가는 구성 파일을 식별하는 것입니다. 그런 다음 모듈의 공개 API를 결정해야 하며 이를 통해 종속 모듈에서 사용할 수 있습니다.

Java Platform Module System의 개발로, **Java Specification Request**(JSR) 376 (https://www.jcp.org/en/jsr/detail?id=376) 에 의해 관리되고 있습니다. JSR은 다음과 같은 근본적인 문제를 해결하는 것이 모듈 시스템의 필요성을 언급합니다.

- **신뢰할 수있는 구성** : 개발자는 프로그램 구성 요소를 구성하기 위해 오래 걸리고 오류가 발생하기 쉬운 클래스 경로 메커니즘으로 어려움을 겪어 왔습니다. 클래스 경로는 구성 요소 간의 관계를 표현할 수 없으므로 필요한 구성 요소가 누락 된 경우 클래스를 사용하려고 시도 할 때까지 발견되지 않습니다. 클래스 패스를 사용하면 동일한 패키지의 클래스를 다른 구성 요소에서 로드 할 수 있으므로 예측할 수 없는 동작과 진단하기 어려운 오류가 발생할 수 있습니다. 제안된 명세는 구성 요소가 다른 구성 요소에 의존한다는 것을 선언하도록 허용 할 것이다.
- **강력한 캡슐화** : Java 프로그래밍 언어 및 JVM의 액세스 제어 메커니즘은 구성 요소가 다른 구성 요소가 내부 패키지에 액세스하지 못하게합니다. 제안된 명세는 컴포넌트가 다른 컴포넌트와 접근 할 수 없는 패키지를 선언 할 수 있도록 한다.

JSR은 다음과 같이 앞의 문제를 해결함으로써 얻은 이점을 나열합니다.

- **확장성있는 플랫폼** : Java SE 플랫폼의 크기가 날로 커짐에 따라 이러한 장치가 SE 클래스 JVM을 실행할 수 있음에도 불구하고 소형 장치에서 사용하기가 점점 더 어려워졌습니다. Java SE 8 (JSR 337)에 도입 된 소형 프로파일은 이러한 점에서 도움이 되지만, 거의 유연하지는 않습니다. 제안 된 사양을 통해 Java SE 플랫폼과 구현을 개발자가 응용 프로그램에서 실제로 요구되는 기능만 포함하는 사용자 정의 구성으로 어셈블 할 수있는 구성 요소 집합으로 분해 할 수 있습니다.
- **플랫폼 무결성 향상** : Java SE 플랫폼 구현 내부의 API를 우연히 사용하는 것은 보안 위험 및 유지 관리 부담입니다. 제안 된 사양에 의해 제공되는 강력한 캡슐화는 Java SE 플랫폼을 구현하는 구성 요소가 내부 API에 대한 액세스를 방지 할 수있게합니다.
- **향상된 성능** : 런타임에 로드되는 클래스가 아닌 다른 특정 구성 요소의 클래스만 클래스가 참조 할 수 있다고 알려진 경우 많은 사전 컴파일 된 전체 프로그램 최적화 기술이 더 효과적 일 수 있습니다. Java SE 플랫폼을 구현하는 구성 요소와 함께 응용 프로그램의 구성 요소를 최적화 할 수있는 경우 성능이 특히 향상됩니다.