Table of Contents

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.

Using inner classes

이 레서피에서는 세 가지 유형의 내부 클래스를 배웁니다.

- **내부 클래스(Inner class)** : 다른 클래스 (내부 클래스) 내부에 정의 된 클래스입니다. 동봉하는 클래스 바깥에서 액세스 할 수 있는지 여부는 public, protected 및 private 키워드에 의해 규제됩니다. 그것은 둘러싸는 클래스의 private 멤버에 액세스 할 수 있어 둘러싸는 클래스는 그 내부 클래스의 private 멤버에 액세스 할 수 있습니다.
- **메소드 로컬 내부 클래스(Method-local inner class)** : 이것은 메소드 내부에서 정의 된 클래스입니다. 범위는 메서드 내로 제한됩니다.
- **익명의 내부 클래스(Anonymous inner class)** : 이것은 객체 인스턴스화 중에 정의 된 익명 클래스입니다.

준비하기

하나의 클래스가 하나의 다른 클래스에 의해 사용될 때, 디자이너는 그러한 클래스를 공개 할 필요가 없다고 결정할 수 있습니다. 예를 들어 Engine 클래스가 Vehicle 클래스에서만 사용된다고 가정 해 봅시다.

그것을하는 방법 ...

- 1. Vehicle 클래스의 내부 클래스로 Engine 클래스를 만듭니다.

```
public class Vehicle {
    private int weightPounds;
    private Engine engine;

    public Vehicle(int weightPounds, int horsePower) {
        this.weightPounds = weightPounds;
        this.engine = new Engine(horsePower);
    }

    public double getSpeedMph(double timeSec){
        return this.engine.getSpeedMph(timeSec);
    }

    private int getWeightPounds(){ return weightPounds; }

    private class Engine {
        private int horsePower;

        private Engine(int horsePower) {
            this.horsePower = horsePower;
        }

        private double getSpeedMph(double timeSec){
            double v = 2.0 * this.horsePower * 746;
            v = v * timeSec * 32.174 / getWeightPounds();
            return Math.round(Math.sqrt(v) * 0.68);
        }
    }
}
```

- 1. Vehicle 클래스의 getSpeedMph() 메서드는 Engine 클래스에 액세스 할 수 있습니다 (Private 클래스로 선언되었지만). Engine 클래스의 private getSpeedMph () 메서드에도 액세스 할 수 있습니다. 내부 클래스는 내부 클래스의 모든 private 요소에도 액세스 할 수 있습니다. 이것이 Engine 클래스의 getSpeedMph()가 Vehicle 클래스의 private

getWeightPounds () 메소드에 접근 할 수있는 이유입니다.

- 1. 내부 클래스 Engine의 사용법을 자세히 살펴보십시오. getSpeedMph() 메서드 만 사용합니다. 디자이너가 앞으로도 그렇게 될 것이라고 생각하면 내부 클래스의 두 번째 유형 인 메서드 로컬 내부 클래스로 만드는 것이 타당합니다:

```
public class Vehicle {
    private int weightPounds;
    private int horsePower;

    public Vehicle(int weightPounds, int horsePower) {
        this.weightPounds = weightPounds;
        this.horsePower = horsePower;
    }

    private int getWeightPounds() { return weightPounds; }

    public double getSpeedMph(double timeSec) {
        class Engine {
            private int horsePower;

            private Engine(int horsePower) {
                this.horsePower = horsePower;
            }

            private double getSpeedMph(double timeSec){
                double v = 2.0 * this.horsePower * 746;
                v = v * timeSec * 32.174 / getWeightPounds();
                return Math.round(Math.sqrt(v) * 0.68);
            }
        }

        Engine engine = new Engine(this.horsePower);

        return engine.getSpeedMph(timeSec);
    }
}
```

인 캡슐 레이션 - 객체의 상태와 동작을 숨김으로써 의도하지 않은 변경이나 무시로 인한 예기치 않은 부작용을 피할 수 있습니다. 그것은 행동을 더 예측 가능하고 이해하기 쉽게 만듭니다. 그렇기 때문에 좋은 디자인은 외부에서 액세스 할 수 있어야 하는 기능만을 제공합니다. 일반적으로 이 클래스는 우선 클래스 생성에 동기를 부여하는 주요 기능입니다.

How it works...

Engine 클래스가 내부 클래스 또는 메소드 로컬 내부 클래스로 구현되는지 여부에 관계없이 테스트 코드는 동일하게 보입니다.

```
public static void main(String arg[]) {
  double timeSec = 10.0;
  int engineHorsePower = 246;
  int vehicleWeightPounds = 4000;

  Vehicle vehicle = new Vehicle(vehicleWeightPounds, engineHorsePower);

  System.out.println("Vehicle speed (" + timeSec + " sec) = " +
                     vehicle.getSpeedMph(timeSec) + " mph");
}
```

이 프로그램을 실행하면 동일한 출력을 얻습니다.

```
Car speed (10.0 sec) = 117.0 mph
```

이제 getSpeedMph() 메소드의 여러 구현을 테스트해야 한다고 가정 해 봅시다.

```
public double getSpeedMph(double timeSec){ return -1.0d; }
```

이 속도 계산 공식이 당신에게 이해가 되지 않는다면, 당신은 맞습니다. 그렇지 않습니다. 결과를 예측 가능하고 이전 구현의 결과와 다르게하기 위해이 작업을 수행했습니다.

이 새로운 구현을 소개하는 방법은 여러 가지가 있습니다. 예를 들어 Engine 클래스의 getSpeedMph () 메소드 구현을 변경할 수 있습니다. 또는 Vehicle 클래스에서 동일한 메소드 구현을 변경할 수 있습니다.

이 방법에서는 익명 내부 클래스라는 세 번째 유형의 내부 클래스를 사용하여이 작업을 수행합니다. 이 방법은 가능한 한 새로운 코드를 작성하지 않거나 기존 코드를 일시적으로 무시하여 새 동작을 신속하게 테스트하려는 경우에 특히 유용합니다. 코드는 다음과 같이 보입니다.

```
public static void main(String... arg) {
  double timeSec = 10.0;
  int engineHorsePower = 246;
  int vehicleWeightPounds = 4000;

  Vehicle vehicle = new Vehicle(vehicleWeightPounds, engineHorsePower) {
    public double getSpeedMph(double timeSec){
      return -1.0d;
    }
  };

  System.out.println("Vehicle speed (" + timeSec + " sec) = " +
                     vehicle.getSpeedMph(timeSec) + " mph");
}
```

이 프로그램을 실행하면 결과는 다음과 같습니다.

```
Vehicle speed (10.0 sec) = -1.0 mph
```

하드 코딩 된 값을 반환하는 getSpeedMph() 메서드 하나만 남겨 두어 Vehicle 클래스 구현을 재정의했습니다. 다른 메소드를 오버라이드하거나 새 메소드를 추가 할 수도 있지만, 데모 목적으로는 간단하게 유지할 것입니다.

정의에 따르면 익명의 내부 클래스는 세미콜론으로 끝나는 명령문의 일부인 표현식이어야합니다. 표현식은 다음으로 구성됩니다.

- new 연산자
- 구현된 인터페이스의 이름 (기본 생성자를 나타내는 괄호 ()가 뒤에옵니다) 또는 확장 클래스의 생성자 (후자는 우리의 경우, 확장 클래스는 Vehicle 임)
- 메소드 선언이있는 클래스 본문 (익명의 내부 클래스의 본문에는 명령문을 사용할 수 없습니다)

모든 내부 클래스와 마찬가지로 익명 내부 클래스는 포함하는 클래스의 모든 멤버에 액세스 할 수 있으며 해당 변수의 값을 캡처 할 수 있습니다. 이를 수행하려면이 변수를 final로 선언해야합니다. 그렇지 않으면 암시적으로 final이됩니다. 이는 값을 변경할 수 없음을 의미합니다 (좋은 IDE는 이러한 값을 변경하려고 하면 이 제한 조건 위반에 대해 경고합니다).

이러한 기능을 사용하여 샘플 코드를 수정하고 메소드 매개 변수로 전달하지 않고 새로 구현 된 getSpeedMph() 메소드에 대한 더 많은 입력 데이터를 제공 할 수 있습니다.

```
public static void main(String... arg) {
  double timeSec = 10.0;
  int engineHorsePower = 246;
  int vehicleWeightPounds = 4000;

  Vehicle vehicle = new Vehicle(vehicleWeightPounds, engineHorsePower) {
    public double getSpeedMph(double timeSec){
      return -1.0d;
    }
```

```
  };

  System.out.println("Vehicle speed (" + timeSec + " sec) = " +
                     vehicle.getSpeedMph(timeSec) + " mph");
}
```

변수 timeSec, engineHorsePower 및 vehicleWeightPounds는 내부 클래스의 getSpeedMph () 메소드로 액세스 할 수 있으며 수정할 수 없습니다. 이 코드를 실행하면 결과는 이전과 같습니다.

```
Car speed (10.0 sec) = 117.0 mph
```

하나의 추상 메소드 (함수 인터페이스라고 함)가 있는 인터페이스의 경우 람다 표현식이라고 하는 특정 유형의 익명의 내부 클래스가 있으므로 더 짧은 표기법을 사용할 수 있지만 인터페이스 구현을 제공합니다. 다음 장에서 함수 인터페이스와 람다 표현에 대해 논의 할 것입니다.

There's more... 더 있다.

내부 클래스는 비정적 중첩 클래스입니다. 자바는 또한 내부 클래스가 비공개 속성과 둘러싸는 클래스의 메서드에 대한 액세스를 필요로 하지 않을 때 사용할 수 있는 정적 중첩 클래스를 만들 수있게합니다. 다음은 예제입니다 (키워드 static이 Engine 클래스에 추가됨).

```
public class Vehicle {
  private Engine engine;

  public Vehicle(int weightPounds, int horsePower) {
    this.engine = new Engine(horsePower, weightPounds);
  }

  public double getSpeedMph(double timeSec){
    return this.engine.getSpeedMph(timeSec);
  }

  private static class Engine {
    private int horsePower;
    private int weightPounds;

    private Engine(int horsePower, int weightPounds) {
      this.horsePower = horsePower;
    }

    private double getSpeedMph(double timeSec){
      double v = 2.0 * this.horsePower * 746;
      v = v * timeSec * 32.174 / this.weightPounds;
      return Math.round(Math.sqrt(v) * 0.68);
    }
  }
}
```

정적 클래스가 비 정적 멤버 (포함하는 클래스 Vehicle의 getWeightPounds() 메서드)에 액세스 할 수 없으므로 구성하는 동안 Engine 클래스에 가중치를 전달해야했습니다 (getWeightPounds() 메서드가 제거되었습니다) 더 이상 필요하지 않았기 때문에).

Chapter 3 Modular Programming

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.

3 3 Creating a simple modular application

이 모듈성이 무엇인지, Java로 모듈형 응용 프로그램을 만드는 방법에 대해 궁금해 해야합니다. 이 레시피에서는 간단한 예제를 통해 Java로 모듈 애플리케이션을 만드는 것에 대한 수수께끼를 풀려고 합니다. 우리의 목표는 모듈식 응용 프로그램을 만드는 방법을 보여주는 것입니다. 따라서 우리는 우리의 목표에 초점을 맞추기 위해 간단한 예를 선택했습니다.

이 예제는 숫자가 소수인지 검사하고, 소수의 합을 계산하고, 수가 짝수인지 확인하고, 짝수와 홀수의 합을 계산하는 간단한 고급 계산기입니다.

Getting ready

우리는 우리의 응용 프로그램을 두 개의 모듈로 나눌 것입니다 :

- 수학적 계산을 수행하기 위한 API가 포함 된 **math.util** 모듈
- 고급 계산기를 시작하는 **calculator** 모듈

How to do it...

1 : **com.packt.math.MathUtil** 클래스에서 **isPrime(Integer number)**로 시작하는 API를 구현해 보겠습니다.

```
public static Boolean isPrime(Integer number) {
    if ( number == 1 ) { return false; }
    return IntStream.range(2,num).noneMatch(i -> num % i == 0 );
}
```

2 : 다음 단계는 **sumOfFirstNPrimes(Integer count)** API를 구현하는 것입니다.

```
public static Integer sumOfFirstNPrimes(Integer count){
    return IntStream.iterate(1,i -> i+1)
                    .filter(j -> isPrime(j))
                    .limit(count).sum();
}
```

3 : 숫자가 짝수인지 확인하는 함수를 작성해 보겠습니다.

```
public static Boolean isEven(Integer number){
    return number % 2 == 0;
}
```

4 : **isEven**의 부정은 숫자가 이상한 지 여부를 알려줍니다. 다음과 같이 처음 N 개의 짝수와 첫 번째 N 개의 홀수의 합을 구하는 함수를 가질 수 있습니다.

```
public static Integer sumOfFirstNEvens(Integer count){
    return IntStream.iterate(1,i -> i+1)
                    .filter(j -> isEven(j))
                    .limit(count).sum();
}

public static Integer sumOfFirstNOdds(Integer count){
    return IntStream.iterate(1,i -> i+1)
                    .filter(j -> !isEven(j))
                    .limit(count).sum();
}
```

앞의 API에서 다음 작업이 반복됨을 알 수 있습니다.

- 1부터 시작하는 숫자의 무한 시퀀스
- 조건에 따라 숫자 필터링하기
- 숫자의 흐름을 주어진 수로 제한
- 이렇게 얻은 숫자의 합 찾기

우리의 관찰을 기반으로, 우리는 앞의 API를 리팩터링하고 다음과 같은 방법으로 이러한 연산을 추출 할 수 있습니다.

```
private static Integer computeFirstNSum(Integer count, IntPredicate filter){
    return IntStream.iterate(1,i -> i+1)
                    .filter(filter)
                    .limit(count).sum();
}
```

여기에서 count는 합계를 찾기 위해 필요한 숫자의 제한이며 filter는 합계를 위한 숫자를 선택하기 위한 조건입니다.

방금 수행 한 리팩터링을 기반으로 API를 다시 작성해 보겠습니다.

```
public static Integer sumOfFirstNPrimes(Integer count){
  return computeFirstNSum(count, (i -> isPrime(i)));
}

public static Integer sumOfFirstNEvens(Integer count){
  return computeFirstNSum(count, (i -> isEven(i)));
}

public static Integer sumOfFirstNOdds(Integer count){
  return computeFirstNSum(count, (i -> !isEven(i)));
}
```

당신은 다음에 대해 궁금해 할 것입니다.

- **IntStream** 클래스 및 메서드의 관련 체인
- 코드베이스에서 **->** 사용
- **IntPredicate** 클래스의 사용

참으로 궁금하신 분은 걱정할 필요가 없습니다. 4 장 Going Functional 및 5 장 스트림 작업 및 파이프 라인에서 이러한 내용을 다룰 것입니다.

지금까지 우리는 수학 계산과 관련하여 몇 가지 API를 보았습니다. 이러한 API는 **com.packt.math.MathUtil** 클래스의 일부입니다. 이 클래스의 전체 코드는이 책에서 다운로드 한 코드베이스의 chp3/2_simple-modular-math-util/math.util/com/packt/math 위치에서 찾을 수 있습니다.

이 작은 유틸리티 클래스를 **math.util**이라는 모듈의 일부로 만들어 보겠습니다. 다음은 모듈을 만드는 데 사용되는 규칙입니다.

1. 모듈과 관련된 모든 코드를 **math.util** 디렉토리에 두고 모듈 루트 디렉토리로 처리하십시오.
2. 루트 폴더에서 파일 이름을 **module-info.java**로 지정하십시오.
3. 그런 다음 패키지와 코드 파일을 루트 디렉토리 아래에 배치합니다.

module-info.java에는 무엇이 들어 있습니까?

- 모듈의 이름
- 패키지가 내보내는 패키지, 즉 다른 모듈에서 사용할 수있는 패키지 - The packages it exports, that is, makes available for other modules to use
- 의존하는 모듈
- 사용하는 서비스
- 구현을 제공하는 서비스

1 장 설치와 자바 9에 입문하기에서 언급했듯이 JDK에는 많은 모듈이 번들로 제공됩니다. 즉, 기존 Java SDK가 모듈화되어 있습니다! 이러한 모듈 중 하나는 **java.base**라는 모듈입니다. 모든 사용자 정의 모듈은 암시적으로 **java.base** 모듈을 의존하거나 필요로 합니다 (암시적으로 **Object** 클래스를 확장하는 모든 클래스를 생각하십시오).

우리의 **math.util** 모듈은 다른 모듈에 의존하지 않습니다 (당연히 **java.base** 모듈 제외). 그러나 다른 모듈에서도 API를 사용할 수 있습니다 (그렇지 않은 경우이 모듈의 존재 여부는 의심스럽습니다). 이 문장을 코드에 삽입 해 보겠습니다.

```
module math.util{
  exports com.packt.math;
}
```

우리는 Java 컴파일러와 런타임에서 **math.util** 모듈이 **com.packt.math** 패키지의 코드를 **math.util**에 종속 된 모듈로 내보내고 있음을 알립니다.

> 이 모듈의 코드는 위치, chp3/2_simple-modular-math-util/math.util에서 찾을 수 있습니다.

이제 **math.util** 모듈을 사용하는 또 다른 모듈 계산기를 작성해 보겠습니다. 이 모듈에는 실행할 수학 연산과 그 다음에 연산을 실행하는 데 필요한 입력에 대한 사용자의 선택을 받아들이는 **Calculator** 클래스가 있습니다. 사용자는 5가지 사용 가능한 수학 연산 중에서 선택할 수 있습니다.

1. 소수 체크
2. 짝수 체크
3. N 소수의 합
4. N의 합
5. N 개의 확률의 합

코드에서 이것을 보자.

```
private static Integer acceptChoice(Scanner reader){
  System.out.println("************Advanced Calculator************");
  System.out.println("1. Prime Number check");
  System.out.println("2. Even Number check");
  System.out.println("3. Sum of N Primes");
  System.out.println("4. Sum of N Evens");
  System.out.println("5. Sum of N Odds");
  System.out.println("6. Exit");
  System.out.println("Enter the number to choose operation");
  return reader.nextInt();
}
```

그런 다음 각 선택 항목에 대해 필요한 입력을 받아들이고 다음과 같이 해당 **MathUtil** API를 호출합니다.

```
switch(choice){
  case 1:
    System.out.println("Enter the number");
    Integer number = reader.nextInt();
    if (MathUtil.isPrime(number)){
      System.out.println("The number " + number +" is prime");
    }else{
      System.out.println("The number " + number +" is not prime");
    }
  break;
  case 2:
    System.out.println("Enter the number");
    Integer number = reader.nextInt();
    if (MathUtil.isEven(number)){
      System.out.println("The number " + number +" is even");
    }
  break;
  case 3:
    System.out.println("How many primes?");
```

```
    Integer count = reader.nextInt();
    System.out.println(String.format("Sum of %d primes is %d",
          count, MathUtil.sumOfFirstNPrimes(count)));
  break;
  case 4:
    System.out.println("How many evens?");
    Integer count = reader.nextInt();
    System.out.println(String.format("Sum of %d evens is %d",
          count, MathUtil.sumOfFirstNEvens(count)));
  break;
  case 5:
    System.out.println("How many odds?");
    Integer count = reader.nextInt();
    System.out.println(String.format("Sum of %d odds is %d",
          count, MathUtil.sumOfFirstNOdds(count)));
  break;
}
```

Calculator 클래스의 전체 코드는 chp3/2_simple-modular-math-util/calculator/com/packt/calculator/Calculator.java에서 찾을 수 있습니다.

math.util 모듈과 같은 방식으로 **calculator** 모듈에 대한 모듈 정의를 만듭니다.

```
module calculator{
  requires math.util;
}
```

앞의 모듈 정의에서 **calculator** 모듈은 **required** 키워드를 사용하여 **math.util** 모듈에 종속된다는 것을 언급합니다.

> 이 모듈의 코드는 chp3 / 2_simple-modular-math-util / calculator에서 찾을 수 있습니다.

이제 코드를 컴파일 해 봅시다.

```
javac -d mods --module-source-path . $(find . -name "*.java")
```

명령 앞의 명령은 chp3/2_simple-modular-math-util에서 실행되어야합니다.

또한 **mods** 디렉토리에 있는 두 모듈, **math.util** 및 **calculator**에서 컴파일 된 코드를 가져야합니다. 아주 간단하지 않았습니까? 하나의 명령과 모듈 간의 종속성을 포함한 모든 것이 컴파일러에 의해 처리됩니다. 모듈 컴파일을 관리하려면 **ant**와 같은 빌드 도구가 필요하지 않았습니다.

--module-source-path 명령은 모듈 소스 코드의 위치를 지정하는 **javac**의 새로운 명령 행 옵션입니다.

이제 앞의 코드를 실행 해 보겠습니다.

```
java --module-path mods -m calculator/com.packt.calculator.Calculator
```

--module-path 명령은 **--classpath**와 유사하게 컴파일 된 모듈의 위치를 지정하는 **java**에 대한 새로운 명령 행 옵션입니다.

위의 명령을 실행하면 계산기가 작동하는 것을 볼 수 있습니다.

```
***********Advanced Calculator***********
1. Prime Number check
2. Even Number check
3. Sum of N Primes
4. Sum of N Evens
5. Sum of N Odds
6. Exit
Enter the number to choose operation
1
Enter the number
11
The number 11 is prime
***********Advanced Calculator***********
1. Prime Number check
2. Even Number check
3. Sum of N Primes
4. Sum of N Evens
5. Sum of N Odds
6. Exit
Enter the number to choose operation
3
How many primes?
5
Sum of 5 primes is 28
***********Advanced Calculator***********
1. Prime Number check
2. Even Number check
3. Sum of N Primes
4. Sum of N Evens
5. Sum of N Odds
6. Exit
Enter the number to choose operation
6
```

축하해! 이를 통해 간단한 모듈 식 응용 프로그램을 만들고 실행할 수 있습니다.

우리는 Windows와 Linux 플랫폼 모두에서 코드를 테스트하는 스크립트를 제공했습니다. Windows의 경우 run.bat를, Linux의 경우 run.sh를 사용하십시오.

How it works...

예제를 통해 이제 모든 모듈에서 동일한 패턴을 적용 할 수 있도록 일반화하는 방법을 살펴 보겠습니다. 우리는 모듈을 생성하기 위한 특별한 규칙을 따랐습니다.

|application_root_directory |--module1_root |----module-info.java |----com |------packt |--------sample |----------MyClass.java |--module2_root |----module-info.java |----com |------packt |--------test |----------MyAnotherClass.java

폴더의 루트에 해당 **module-info.java**가 있는 폴더 내에 모듈 특정 코드를 배치합니다. 이렇게 하면 코드가 잘 정리됩니다.

module-info.java가 포함 할 수있는 것을 살펴 보자. Java 언어 사양 (http://cr.openjdk.java.net/~mr/jigsaw/spec/lang-vm.html)에서 모듈 선언의 형식은 다음과 같습니다.

{Annotation} [open] module ModuleName

구문은 다음과 같습니다.

- **{Annotation}** : **@Annotation(2)** 형식의 주석입니다.
- **open** :이 키워드는 선택 사항입니다. 열린 모듈은 모든 구성 요소를 리플렉션을 통해 런타임에 액세스 가능하게 만듭니다. 그러나 컴파일 타임 및 런타임에서는 명시적으로 내 보낸 구성 요소만 액세스 할 수 있습니다.
- **module** : 모듈 선언에 사용되는 키워드입니다.
- **ModuleName** : 식별자 이름 사이에 허용되는 점 (.)이있는 올바른 Java 식별자인 모듈의 이름입니다. math.util과 유사합니다.
- **{ModuleStatement}** : 모듈 정의 내에서 허용되는 명령문의 집합입니다. 다음으로 이것을 확장합시다.

모듈 선언문은 다음 형식을 취합니다.

```
ModuleStatement:
  requires {RequiresModifier} ModuleName ;
  exports PackageName [to ModuleName {, ModuleName}] ;
  opens PackageName [to ModuleName {, ModuleName}] ;
  uses TypeName ;
  provides TypeName with TypeName {, TypeName} ;
```

모듈문은 여기에서 디코딩됩니다.

- requires : 모듈에 대한 의존성을 선언하는 데 사용됩니다. {RequiresModifier}는 전 이적이거나 정적이거나 둘 다일 수 있습니다. 이항 (Transitive)은 주어진 모듈에 의존하는 모든 모듈이 주어진 모듈에 대해 과도기적으로 필요한 모듈에 절대적으로 의존함을 의미합니다. Static은 모듈 종속성이 컴파일 타임에는 필수이지만 런타임에는 선택 사항임을 의미합니다. 몇 가지 예는 requires math.util, requires transitive math.util 및 requires static math.util입니다.
- exports : 이것은 주어진 패키지를 종속 모듈이 액세스 할 수있게하는 데 사용됩니다. 선택 사항으로 모듈 이름을 지정하여 특정 패키지에 대한 액세스 가능성을 강제 설정할 수 있습니다 (예 : exports com.package.math와 claculator).
- opens : 특정 패키지를 여는 데 사용됩니다. 앞에서 모듈 선언으로 open 키워드를 지정하여 모듈을 열 수 있음을 알았습니다. 그러나 이것은 덜 제한적일 수 있습니다. 따라서 제한적으로 만들기 위해 opens 키워드를 사용하여 런타임에 반사 액세스를 위한 특정 패키지를 열 수 있습니다. com.packt.math가 열립니다.
- uses : java.util.ServiceLoader를 통해 액세스 할 수있는 서비스 인터페이스에 대한 종속성을 선언하는 데 사용됩니다. 서비스 인터페이스는 현재 모듈 또는 현재 모듈이 의존하는 모듈에있을 수 있습니다.
- provides : 이것은 서비스 인터페이스를 선언하고 적어도 하나의 구현을 제공하는데 사용됩니다. 서비스 인터페이스는 현재 모듈이나 다른 종속 모듈에서 선언 할 수 있습니다. 그러나 서비스 구현은 동일한 모듈에 제공되어야합니다. 그렇지 않으면 컴파일 타임 오류가 발생합니다.

우리는 **uses**와 **provides** 절을 레시피, 서비스를 사용하여 소비자 모듈과 공급자 모듈 간의 느슨한 결합을 생성하는 방법에 대해 자세히 살펴볼 것입니다.

모든 모듈의 모듈 소스는 **--module-source-path** 명령행 옵션을 사용하여 한 번에 컴파일 할 수 있습니다. 이렇게 하면 모든 모듈이 컴파일되어 **-d** 옵션이 제공하는 디렉토리 아래의 해당 디렉토리에 배치됩니다. 예를 들어 "javac -d mods --module-source-path. $ (find. -name" *.java ")"*는 현재 디렉토리의 코드를 **mods* 디렉토리로 컴파일합니다.

코드를 실행하는 것도 똑같이 간단합니다. 명령행 옵션 **--module-path**를 사용하여 모든 모듈이 컴파일되는 경로를 지정합니다. 그런 다음 명령줄 옵션 **-m**을 사용하여 정규화 된 주 클래스 이름과 함께 모듈 이름을 언급합니다 (예 : java --module-path mods -m calculator/com.packt.calculator.Calculator).

Chapter 4 Going Functional

4 장. 기능적으로 가기

이 장에서는 함수형 프로그래밍이라는 프로그래밍 패러다임과 Java 9에서의 적용 가능성을 소개합니다. 다음 레시피를 다룹니다.

- 함수형 인터페이스의 이해와 생성
- 람다식의 이해
- 메서드 참조 사용
- 람다 친화적인 API 생성 및 호출
- 프로그램에서 람다식을 활용

- Introduction

- Understanding and creating a functional interface
- Understanding lambda expressions
- Using method references
- Creating and invoking?lambda friendly APIs
- Leveraging lambda expressions in your programs

Introduction

함수형 프로그래밍 - 특정 함수를 객체로 취급하고 이를 매개 변수 또는 메소드의 반환 값으로 전달하는 기능은 많은 프로그래밍 언어에서 제공되는 기능입니다. 오브젝트 상태 및 변경 가능한 데이터의 변경을 방지합니다. 함수의 결과는 호출 된 횟수에 관계없이 입력 데이터에만 의존합니다. 이 스타일을 사용하면 결과를 더 예측 가능하게 만들 수 있으며 이는 함수형 프로그래밍의 가장 매력적인 면입니다.

> Functional programming--the ability to treat a certain piece of functionality as an object and to pass it as a parameter or the return value of a method--is a feature present in many programming languages. It avoids the changing of an object state and mutable data. The result of a function depends only on the input data, no matter how many times it is called. This style makes the outcome more predictable, which is the most attractive aspect of functional programming.

Java에 대한 소개를 통해 클라이언트 코드에서 라이브러리로 병렬 처리의 책임을 이동함으로써 Java 8의 병렬 프로그래밍 기능을 향상시킬 수 있습니다. 이 전에 Java 콜렉션의 요소를 처리하기 위해 클라이언트 코드는 콜렉션에서 반복자를 확보하고 콜렉션의 처리를 구성해야했습니다.

> Its introduction to Java also allows you to improve parallel programming capabilities in Java 8 by shifting the responsibility of parallelism from the client code to the library. Before this, in order to process elements of Java collections, the client code had to acquire an iterator from the collection and organize the processing of the collection.

Java 8에서 함수 (함수 인터페이스 구현)를 매개 변수로 받아들이고 이를 콜렉션의 각 요소에 적용하는 새로운 (default)메소드가 추가되었습니다. 따라서 병렬 처리를 구성하는 것은 라이브러리의 책임입니다. 한 가지 예는 모든 Iterable 인터페이스에서 사용할 수 있는 forEach(Consumer) 메소드입니다. 여기서 Consumer는 함수형 인터페이스입니다. 또 다른 예는 모든 Collection 인터페이스에서 사용할 수 있는 removeIf(Predicate) 메소드입니다. Predicate는 함수형 인터페이스이기도 합니다. 그런 다음 List에 사용할 수있는 sort(Comparator) 및 replaceAll(UnaryOperator) 메서드와 Map에 대한 compute()와 같은 다른 여러 메서드가 있습니다.

> In Java 8, new (default) methods were added that accept a function (implementation of a functional interface) as a parameter and then apply it to each element of the collection. So, it is the library's responsibility to organize parallel processing. One example is the forEach(Consumer) method that is available in every Iterable interface, where Consumer is a functional interface. Another example is the removeIf(Predicate) method that is available for every Collection interface, where Predicate is a functional interface too. Then we have the sort(Comparator) and replaceAll(UnaryOperator) methods that are available for List and several other methods, such as compute() for Map.

43개 함수형 인터페이스가 java.util.function 패키지에 제공됩니다. 각각에는 하나의 추상 메소드 만 포함됩니다. 람다식은 하나의 추상화 방법 제한을 이용하고 이러한 인터페이스의 구현을 크게 단순화합니다.

> Forty-three functional interfaces are provided in the java.util.function package. Each of them contains only one abstract method. Lambda expressions take advantage of the one-abstract-method limitation and significantly simplifies the implementation of such an interface.

함수형 프로그래밍이 없다면 Java에서 매개 변수로 일부 기능을 전달하는 유일한 방법은 인터페이스를 구현하는 클래스를 작성하고 해당 객체를 만든 다음 이를 매개 변수로 전달하는 것입니다. 그러나 익명의 클래스를 사용하는 최소한의 스타일조차도 너무 많은 코드를 작성해야합니다. 함수형 인터페이스와 람다(lambda)표현식을 사용하면 코드를 더 짧고, 명료하게 표현할 수 있습니다.

> Without functional programming, the only way to pass some functionality as a parameter in Java would be through writing a class that implements an interface, creating its object, and then passing it as a parameter. But even the least involved style--using an anonymous class--requires writing too much of code. Using functional interfaces and lambda expressions makes the code shorter, clearer, and more expressive.

이 장에서 새로운 Java기능 (함수형 인터페이스 및 람다식)을 정의하고 설명하고 코드 예제에서의 적용 가능성을 보여줍니다. 이러한 새로운 기능을 Java로 가져오면 언어의 일급시민이 됩니다. 그러나 그들의 힘을 이용하려면 함수형 프로그래밍에 노출되지 않은 사람들을 위해 코드를 사고하고 구성하는 새로운 방법이 필요합니다.

> Throughout the chapter, we will define and explain these new Java features--functional interfaces and lambda expressions--and demonstrate their applicability in code examples. Bringing these new features into Java makes functions first-class citizens of the language. But taking advantage of their power requires, for those not exposed to functional programming yet, a new way of thinking and organizing the code.

이러한 기능을 보여주고 이를 사용하는 최상의 방법을 공유하는 것이 이 장의 목적입니다.

함수형 인터페이스의 이해와 생성

이 레서피에서는 Java 8부터 지원되는 함수형 인터페이스에 대해 배우게 됩니다.

Getting ready

오직 하나의 추상 메소드가 있는 인터페이스를 함수형 인터페이스라고합니다. 런타임 오류를 피하기 위해 @FunctionalInterface 어노테이션이 컴파일러에 의도에 대해 알려주는 Java 8에 도입되었습니다. 이전 장의 데모 코드에서 이미 함수형 인터페이스 예제를 보았습니다.

```
public interface SpeedModel {
  double getSpeedMph(double timeSec,
  int weightPounds, int horsePower);
  enum DrivingCondition {
    ROAD_CONDITION,
    TIRE_CONDITION
  }
  enum RoadCondition {
    //...
  }
  enum TireCondition {
    //...
  }
}
```

enum 유형 또는 구현된 (default 또는 static) 메소드가 존재하더라도 그것이 작동하지 않는 인터페이스가 되지는 않습니다. 추상 (구현되지 않은) 메소드만 계산됩니다. 그래서 이것은 함수형 인터페이스의 한 예입니다.

```
public interface Vehicle {
  void setSpeedModel(SpeedModel speedModel);
  default double getSpeedMph(double timeSec){ return -1; };
  default int getWeightPounds(){ return -1; }
  default int getWeightKg(){
    return convertPoundsToKg(getWeightPounds());
  }
  private int convertPoundsToKg(int pounds){
    return (int) Math.round(0.454 * pounds);
  }
  static int convertKgToPounds(int kilograms){
    return (int) Math.round(2.205 * kilograms);
  }
}
```

이전 장에서 인터페이스에 대해 이미 배웠던 것을 요약하면 getWeightKg()에 의해 호출 될 때 getWeightPounds() 메소드의 구현이 -1을 리턴합니다. 그러나 getWeightPounds() 메서드가 클래스에 구현되어 있지 않은 경우에만 true입니다. 그렇지 않으면 런타임에 클래스 구현이 사용됩니다.

default 인터페이스 및 static 인터페이스 메소드 외에도 함수형 인터페이스에는 java.lang.Object의 모든 추상 메소드가 포함될 수 있습니다. Java에서는 모든 객체가 java.lang.Object 메소드의 기본 구현으로 제공되므로 컴파일러와 Java 런타임은 이러한 추상 메소드를 무시합니다.

예를 들어 이것은 함수형 인터페이스입니다.

```
public interface SpeedModel {
  double getSpeedMph(double timeSec, int weightPounds, int horsePower);
  boolean equals(Object obj);
  String toString();
```

```
}
```

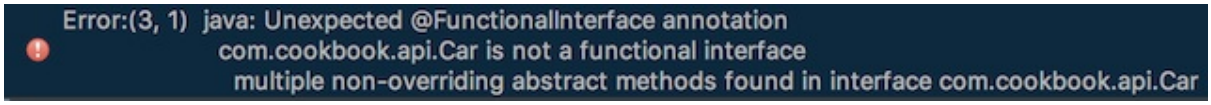
다음은 함수형 인터페이스가 아닙니다.

```
public interface Car extends Vehicle {
  int getPassengersCount();
}
```

Car 인터페이스에는 자체 getPassengersCount() 메소드와 Vehicle 인터페이스에서 상속된 setSpeedModel() 메소드의 두 가지 추상 메소드가 있기 때문입니다. 예를 들어, @FunctionalInterface 주석을 Car 인터페이스에 추가합니다.

```
@FunctionalInterface
public interface Car extends Vehicle {
  int getPassengersCount();
}
```

이렇게하면 컴파일러에서 다음 오류를 생성합니다.



@FunctionalInterface 어노테이션을 사용하면 컴파일 타임에 오류를 catch하는 데 도움이 될뿐만 아니라 설계 의도에 대한 안정적인 통신을 보장합니다. 그것은 당신이나 다른 프로그래머들이이 인터페이스가 하나 이상의 추상적인 방법을 가질 수 없다는 것을 기억하는 데 도움이 됩니다. 이 방법은 코드가 이미 그러한 가정에 의존하고 있는 경우에 특히 중요합니다.

같은 이유로 Java 8의 Runnable 및 Callable 인터페이스 (이전 버전부터 Java에 존재)는 @FunctionalInterface로 어노테이션 처리되어 이 구분을 명시적으로 표시하고 사용자에게 또 다른 추상 메소드를 추가하려고 시도하는 사람들에게 상기시킵니다.

```
@FunctionalInterface
interface Runnable { void run(); }

@FunctionalInterface
interface Callable<V> { V call() throws Exception; }
```

메소드의 매개 변수로 사용할 계획인 고유한 함수형 인터페이스를 작성하기전에 먼저 java.util.function 패키지에 제공된 43 개의 기능 인터페이스 중 하나를 사용하여 이를 피하는 것을 고려하십시오. 대부분은 Function, Consumer, Supplier 및 Predicate의 네 가지 인터페이스로 이루어져 있습니다.

How to do it...

다음은 함수형 인터페이스에 익숙해 지도록 수행 할 수 있는 단계입니다.

1. 함수형 인터페이스 살펴보기 Function :

```
@FunctionalInterface
public interface Function<T,R>
```

그것의 Javadoc은 T 타입의 하나의 인수를 받아들이고 R 타입의 결과를 만들어 낸다. 함수 메소드는 apply (Object)이다. 익명 클래스를 사용하여 이 인터페이스의 구현을 만들 수 있습니다.

```
Function<Integer, Double> ourFunc = new Function<Integer, Double>() {
    public Double apply(Integer i) {
        return i * 10.0;
    }
};
```

유일한 메소드, apply() 는 Integer 타입의 값 (또는 autoboxed 될 원시 타입 int)의 값을 인자로 받고, 10을 곱하고, Double 타입 (primitive double 또는 unboxed 타입)의 값을 반환한다. 다음과 같이 작성할 수 있습니다.

```
System.out.println(ourFunc.apply(1));
```

결과는 다음과 같습니다.

```
10.0
```

다음 레서피에서는 람다식을 소개하고 그 사용법이 구현을 훨씬 더 짧게 만드는 방법을 보여줄 것입니다. 그러나 지금은 익명의 수업을 계속 사용하겠습니다.

1. 함수형 인터페이스 Consumer(이름은 이 인터페이스의 메소드가 값을 받아들이지만 아무 것도 반환하지 않는다는 것을 기억하는 데 도움이됩니다. 단지 소비합니다).

```
public interface Consumer<T>
    형식 T의 단일 입력 인수를 허용하고 결과를 반환하지 않습니다.
    함수 메서드는 accept (Object)입니다.
```

이 인터페이스의 구현은 다음과 같습니다.

```
Consumer<String> ourConsumer = new Consumer<String>() {
    public void accept(String s) {
        System.out.println("The " + s + " is consumed.");
    }
};
```

accepts() 메서드는 String 유형의 매개 변수 값을 받아서 인쇄합니다. 다음과 같이 씁니다.

```
ourConsumer.accept("Hello!");
```

이 결과는 다음과 같습니다.

```
The Hello! is consumed.
```

1. 함수형 인터페이스 Supplier (이 인터페이스의 메소드가 어떤 값도 받아들이지 않지만 뭔가를 반환한다는 것을 기억하는 데 도움이되는 이름 - 공급품 만 보임)을 살펴보십시오.

```
public interface Supplier<T>
    Represents a supplier of results of type T.
    The functional method is get().
```

이 인터페이스의 유일한 메소드는 get()입니다. get()은 입력 매개 변수를 가지지 않고 T 유형의 값을 리턴합니다. 이에 따라 함수를 작성할 수 있습니다.

```
Supplier<String> ourSupplier = new Supplier<String>() {
    public String get() {
        String res = "Success";
        //Do something and return result – Success or Error.
        return res;
    }
};
```

get() 메서드는 무언가를 수행 한 다음 String 유형의 값을 반환하므로 다음을 작성할 수 있습니다.

```
System.out.println(ourSupplier.get());
```

이 결과는 다음과 같습니다.

```
Success
```

1. 함수형 인터페이스 Predicate (이름은 이 인터페이스의 메소드가 부울을 반환한다는 것을 기억하는 데 도움이 됩니다.

```
@FunctionalInterface
public interface Predicate<T>
```

JavaDoc 상태 : T형의 하나의 인수에 대한 술어(부울 값 함수)를 나타냅니다. 함수 메소드는 test(Object)입니다. 즉, 이 인터페이스의 유일한 메소드는 T 유형의 입력 매개 변수를 허용하고 유형이 boolean 인 값을 리턴하는 test(Object)입니다. 함수를 만들어 봅시다 :

```
Predicate<Double> ourPredicate = new Predicate<Double>() {
    public boolean test(Double num) {
        System.out.println("Test if " + num + " is smaller than 20");
        return num < 20;
    }
};
```

test() 메서드는 Double 형식의 값을 매개 변수로 받아들이고 형식 boolean 형식의 값을 반환하므로 다음을 작성할 수 있습니다.

```
System.out.println(ourPredicate.test(10.0) ? "10 is smaller" : "10 is bigger");
```

이 결과는 다음과 같습니다.

```
Test if 10.0 is smaller than 20
10 is smaller
```

1. java.util.function 패키지의 다른 39개의 함수형 인터페이스를 살펴보십시오. 그것들은 앞서 논의한 네 개의 인터페이스의 변형입니다. 이러한 변형은 다음과 같은 이유로 생성됩니다.
2. int, double 또는 long 프리미티브를 명시 적으로 사용하여 autoboxing 및 unboxing을 방지하여 성능 향상
3. 2개의 입력 파라미터를 수용하기 위해
4. 더 짧은 표기법

//TODO

4 3 Understanding lambda expressions 람다 식의 이해

이 레서피에서는 Java 8부터 지원되는 람다식에 대해 배우게 됩니다.

Getting ready

이전 레시피 (함수형 인터페이스의 구현을 위해 익명 클래스를 사용)의 예제는 너무 커 보이고 너무 지나치게 느꼈습니다. 이미 객체 참조의 유형으로 선언했기 때문에 인터페이스 이름을 반복할 필요가 없었습니다. 둘째, 함수형 인터페이스의 경우 (하나의 추상 메서드 만 사용) 구현할 메서드 이름을 지정할 필요가 없었습니다. 컴파일러와 Java 런타임은 어쨌든 그것을 파악할 수 있습니다. 우리가 필요한 것은 새로운 기능을 제공하는 것이었습니다. 이것은 람다식이 구조에 오는 곳입니다.

The examples in the previous recipe (that used anonymous classes for functional interfaces' implementation) looked bulky and felt excessively verbose. For one, there was no need to repeat the interface name because we had declared it already as the type for the object reference. Second, in the case of a functional interface (that had only one abstract method), there was no need to specify the method name to be implemented. The compiler and Java runtime can figure it out anyway. All we needed was to provide the new functionality. This is where a lambda expression comes to the rescue.

TODO

h1. Chapter 5 Stream Operations and Pipelines

최신 Java 릴리스 (8 및 9)에서 콜렉션 API는 스트림 도입과 람다 표현식을 이용한 내부 반복에 대한 주요 개혁을 실시했습니다. 이 장에서는 파이프 라인을 만들기 위해 스트림을 활용하고 컬렉션에서 여러 작업을 연결하는 방법을 보여줍니다. 또한 이러한 작업을 병렬로 수행하는 방법을 알려 드리고자합니다. 우리는 다음과 같은 조리법을 다룰 것입니다 :

- 새 팩토리 메서드를 사용하여 컬렉션 객체 만들기
- 스트림에서 생성 및 작동
- 스트림에서 연산 파이프 라인 만들기
- 스트림의 병렬 계산

[[5_1 Introduction]] [[5_2 Using the new factory methods to create collection objects]]