9장. 리팩터링, 테스트, 디버깅

- 기존에 람다, 스트림 API를 사용하지 않던 레거시 코드를 리팩터링할 방법을 알아보자.
- 람다 표현식으로 전략, 템플릿 메소드, 옵저버, 의무 체인, 팩토리 등의 객체지향 디자인 패턴을 어떻게 간소화할 수 있는지도 살펴보자.

코드 가독성 개선

코드 가독성은 다른 사람이 보더라도 쉽게 이해할 수 있음을 의미한다.

다음 리팩터링 예제를 통해 배워보자

- 익명 클래스를 람다 표현식으로 리팩터링하기
- 람다 표현식을 메서드 참조로 리팩터링 하기
- 명령형 데이터 처리를 스트림으로 리팩터링하기

익명 클래스를 람다 표현식으로 리팩터링하기

익명 클래스는 람다 표현식으로 리팩터링을 할 수 있는데 익명 클래스는 그 장황함과 쉽게 에러를 발생시킨다는 점에서 단점이 있기 때문이다.

하지만 변환이 불가한 케이스도 있는데 아래와 같다.

1. 익명 클래스에서 사용한 `this` 와 `super` 는 다른 의미를 가진다.
   익명 클래스에서 this는 자기 자신을 가르킨다.
   람다는 람다를 감싸는 클래스를 가리킨다.
2. 익명 클래스는 감싸고 있는 클래스의 변수를 가릴 수 있다.
   람다는 변수를 가릴 수 없다.

```
int a = 123;
Runnable run = () -> {
      a = 456;
};
```

3. 마지막으로 익명 클래스를 람다 표현식으로 바꾸면 컨텍스트 오버로딩에 따른 모호함이 초래 될 수 있다. 익명 클래스는 인스턴트화 할때 명시적으로 형식이 정해지는 반면 람다의 형식은 컨텍스트에 따라 달라지기 때문이다.
   하지만 명시적 형변환으로 타입을 명시해주면 이 문제도 해결이 된다.

```
public class Example {
    public static void main(String[] args) {
        //Compile 에러 발생
        //Task.subProcess(() -> System.out.println("##"));
        Task.subProcess((Task)() -> System.out.println("##"));
    }
}

interface Task {
    void process();

    static void subProcess(Runnable runnable) {
        runnable.run();
    }

    static void subProcess(Task task) {
        task.process();
    }
}
```

람다 표현식을 메서드 참조로 리팩터링하기

1. 코드의 나열보다는 메소드 선언을 한 후 메소드 참조를 거는 것이 더 가독성이 좋다.

- 원본

```
Map<CaloricLevel, List<Dish>> dishesByCaloricLevel =
    menu.stream()
        .collect(
              groupingBy(dish -> {
                  if (dish.getCalories() <= 400) return CaloricLevel.DIET;
                  else if (dish.getCalories() <= 700) return CaloricLevel.NORMAL;
                  else return CaloricLevel.FAT;
}));
```

- 칼로리 레벨 추출 메소드 분리

```
Map<CaloricLevel, List<Dish>> dishesByCaloricLevel =
    menu.stream().collect(groupingBy(Dish::getCaloricLevel));
```

람다 표현식을 메서드로 추출했다.

```
public class Dish{
    ...
    public CaloricLevel getCaloricLevel() {
        if (this.getCalories() <= 400) return CaloricLevel.DIET;
        else if (this.getCalories() <= 700) return CaloricLevel.NORMAL;
        else return CaloricLevel.FAT;
    }
}
```

2. 내장 컬렉터를 사용하면 구현해야하는 코드의 양을 줄일 수 있다.

- 원본

```
int totalCalories =
    menu.stream().map(Dish::getCalories)
                 .reduce(0, (c1, c2) -> c1 + c2);
```

- 내장 컬렉터 `summingInt` 활용

위의 reduce의 경우 현재는 간단하지만 연산이 복잡할 경우 해석을 해야 할 수도 있다.

하지만 summingInt와 같이 내장 컬렉터를 사용하면 그 자체로 의도를 들어낼 수 있다.

```
int totalCalories = menu.stream().collect(summingInt(Dish::getCalories));
```

명령형 데이터 처리를 리팩터링 하기

기존에 for-loop를 활용한 내부 반복의 경우 코드 전반에 대한 이해가 수반되어야만 완전한 이해가 가능하다.

```
List<String> dishNames = new ArrayList<>();
for(Dish dish: menu) {
    if(dish.getCalories() > 300) {
        dishNames.add(dish.getName());
    }
}
```

반면에 스트림은 의도를 들어내기가 용이하며, 병렬 처리 또한 쉽다.

```
menu.parallelStream()
    .filter(d -> d.getCalories() > 300)
    .map(Dish::getName)
    .collect(toList());
```

코드 유연성 개선

람다 표현식을 이용하면 동작 파라미터화를 쉽게 구현할 수 있다. 다시 말해, 람다를 전달해서 다양한 동작을 표현할 수 있고, 변화하는 요구사항에 대응하는 코드도 쉽게 작성할 수 있다.

조건부 연기 실행과 **실행 어라운드** 란 많이 사용되는 패턴으로 람다 표현식 리팩터링을 살펴본다.

조건부 연기 실행

제어 흐름문이 얽힌 코드를 흔히 볼 수 가 있다. 다음은 내장 자바 Logger 클래스를 사용하는 예제다.

```
if (logger.isLoggable(Log.FINER)) {
    logger.finer("Problem: " + generateDiagnostic());
}
```

위 코드의 단점은 아래와 같다.

- logger의 상태가 isLoggable이란 메서드에 의해 클라이언트 코드로 노출된다.
- 메세지를 로깅할때 마다 logger 객체의 상태를 매번 확인해야 할까? 이들의 코드를 어지럽힐 뿐이다.

객체지향적인 구조는 객체 간의 메시지를 통한 소통을 통해 조화를 이루며 요구사항이 처리되는 것을 의미한다. 최대한 캡슐화하여 상태는 숨기고 메시지로 처리를 하는 것이 유지보수성과 가독성 그리고 클라이언트 입장에서의 사용성, 결론적으로 사이드 이펙트도 최소화 할 수 있다.

그래서 레벨과 메시지를 받아서 Logger 내부에서 로깅 가능 여부를 확인하는 메소드를 사용하는 것이 바람직 하다.

```java
public void log(Level level, String msg) {
    if (!isLoggable(level)) {
        return;
    }
    LogRecord lr = new LogRecord(level, msg);
    doLog(lr);
}
```

실행 어라운드

매번 같은 준비, 종료 과정을 반복하는 코드가 있다면 람다로 변환할 수 있다. 준비, 종료 과정을 처리하는 로직을 재사용함으로써 코드 중복을 줄일 수 있다.

아래와 같이 for-loop, stream, parallelStream의 성능을 비교하는 코드가 있다고 했을 때 시작 시점 시간과 종료 시점 시간 측정, 그리고 콘솔 출력을 하는 로직은 반복된다. 그럴 때 아래와 같이 동작 파라미터를 받으면 해당 코드는 재사용하고 테스트 내용만 교체하는 것이 가능하다.

```
class StreamPerformanceTest {

    private static final int TEST_MAX_RANGE= 30_000_000;
    private static List<Integer>list= new ArrayList<Integer>();

    public static void main(String[] args) {
        list = IntStream.rangeClosed(1,TEST_MAX_RANGE).boxed().collect(toList());

        test("for-loop", (List<Integer> list) -> {
            int result = 0;
            for (int each : list) {
                result += each;
            }
        });

        test("stream", (List<Integer> list) -> {
            list.stream().mapToInt(i -> i.intValue()).sum();
        });

        test("parallel-stream", (List<Integer> list) -> {
            list.parallelStream().mapToInt(Integer::intValue).sum();
        });

    }

    private static void test(String label, Consumer<List<Integer>> function) {
        long current =currentTimeMillis();
        function.accept(list);
```

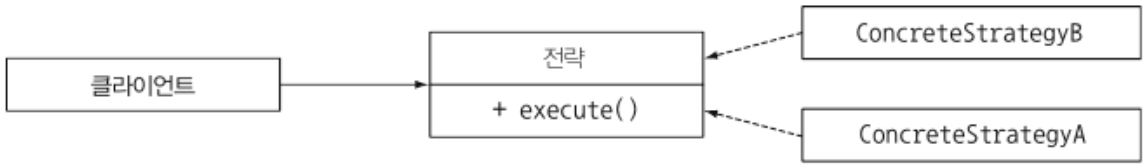
```
        long end =currentTimeMillis();
        System.out.println(label + "    " + (end - current));
    }
}
```

람다로 객체지향 디자인 패턴 리팩터링하기

디자인 패턴은 공통적인 소프트웨어 문제를 설계할 때 재사용할 수 있는, 검증된 청사진을 제공한다. 디자인 패턴은 재사용할 수 있는 부품으로 여러 가지 다리(현수교, 아치교 등)를 건설하는 엔지니어링에 비유할 수 있다.

다음 다섯 가지 패턴을 살펴보자.

전략 패턴



- 알고리즘을 나타내는 인터페이스(Strategy 인터페이스)
- 다양한 알고리즘을 나타내는 한 개 이상의 인터페이스 구현(ConcreteStrategyA, ConcreteStrategyB 같은 구체적인 구현 클래스)
- 전략 객체를 사용하는 한 개 이상의 클라이언트

템플릿 메소드

알고리즘의 개요를 제시한 다음에 일부를 고칠 수 있는 유연함을 제공해야할 때 유용한 패턴이다.

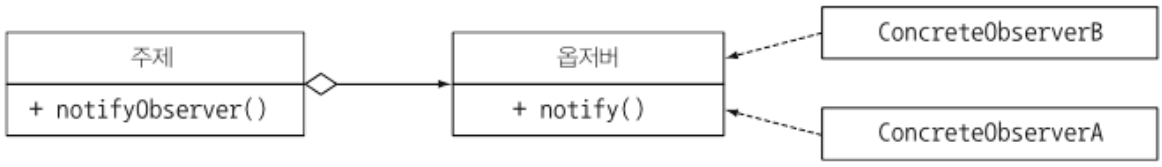
다시 말해 전반적인 아웃라인은 동일하지만 일부분에 대해서만 달리해야할 때 좋다.

```
class GitCommitter {
    public void commit(List<File> files, Runnable runnable) {
        /* commit logic */
        runnable.run();
    }
}
```

옵저버

어떤 이벤트가 발생했을 때 한 객체(주체, subject)가 다른 객체 리스트(옵저버, observer)에 자동으로 알림을 보내야 하는 상황에서 옵저버 디자인 패턴을 사용한다.

그림 9-2 옵저버 디자인 패턴



계좌 이체를 한 이후에 문자 알림과 이메일 알림을 보내야 하는 상황이고, 향후에 푸쉬 알림도 추가될 수 있는 상황을 생각해보자

```
interface Observer {
    void notify(String message);
}

class MailObserver implements Observer {
    public void notify(String message) {
        if(message != null) {
            /* detail Logic */
        }
    }
}

class SmsObserver implements Observer {
    public void notify(String message) {
        if(message != null) {
            /* detail Logic */
        }
    }
}

interface Subject {
    void registerObserver(Observer observer);
    void notifyObservers(String message);
}

class SendMoneySubject implements Subject {
    private final List<Observer> observers = new ArrayList<>();

    @Override
    public void registerObserver(Observer observer) {
        this.observers.add(observer);
    }
```

```
    @Override
    public void notifyObservers(String message) {
        observers.forEach(observer -> observer.notify(message));
    }
}

class Main {
    public static void main(String[] args) {
        SendMoneySubject subject = new SendMoneySubject();
        Stream.of(new SmsObserver(), new MailObserver())
                .forEach(subject::registerObserver);
        subject.notifyObservers("30,000원이 이체되었습니다.\n잔액: 123,000원");
    }
}
```

작업 체인

작업 처리 객체의 체인을 만들 때는 의무 체인 패턴을 사용한다. 한 객체가 어떤 작업을 처리한 다음에 다른 객체로 결과를 전달하고, 다른 객체도 해야할 작업을 처리한 다음 또 다른 객체로 전달하는 식이다.

```
UnaryOperator<String> introduce = text -> "소개할게요! " + text;
UnaryOperator<String> jordyHeight = text -> text + "키가 102cm 입니다.";
UnaryOperator<String> jordyHobby = text -> text + "취미는 코딩입니다.";
Function<String, String> jordyPipeline = introduce
        .andThen(jordyHeight)
        .andThen(jordyHobby);

String result = jordyPipeline.apply("죠르디는 말이죠.");
```

팩토리

인스턴스화 로직을 클라이언트에 노출하지 않고 객체를 만들 때 팩토리 디자인 패턴을 사용한다.

```
class Main {

    private static Supplier<Jordy> jordySupplier = Jordy::new;

    static class Jordy {
        String characteristics;
    }

    public static void main(String[] args) {
        IntStream.rangeClosed(1,10).forEach(num -> {
```

```
            System.out.println(jordySupplier.get());
        });
    }
}

/* console example */
Main$Jordy@f1e4fa
Main$Jordy@6a30c7
Main$Jordy@181eda8
Main$Jordy@8de145
Main$Jordy@1fa135a
Main$Jordy@f7f36a
Main$Jordy@1faab1
Main$Jordy@171fc7e
Main$Jordy@e65a89
Main$Jordy@1f38edc
```

람다 테스팅

테스트 없는 로직은 믿을 수 없다!

보이는 람다 표현식의 동작 테스팅

람다는 익명이므로 테스트 코드 이름을 호출할 수 없다. 그래서 테스트를 위해서는 람다를 필드에 저장해서 재사용할 수 있으며, 람다의 로직을 테스트할 수 있다.

람다를 사용하는 메서드의 종작에 집중하라

람다의 목표는 정해진 동작을 다른 메서드에서 사용할 수 있도록 하나의 조각으로 캡슐화하는 것이다. 그러려면 세부 구현을 포함하는 람다 표현식을 공개하지 말아야 한다. 람다 표현식을 사용하는 메서드의 동작을 테스트함으로써 람다를 공개하지 않으면서도 람다 표현식을 검증할수 있다.

복잡한 람다를 개별 메소드로 분할하기

복잡한 람다 표현식을 일반 메소드로 선언함으로써 메서드 참조를 통해 호출해서 처리할 수 있다.

고차원 함수 테스팅

함수를 인수로 받거나 다른 함수를 반환하는 메서드를 고차원 함수라고 한다. 이러한 함수는 테스트와 디버깅을 통해 검증이 필요하다.

디버깅

문제가 발생한 코드를 디버깅할 때 개발자는 두 가지를 가장 먼저 확인한다.

- 스택 트레이스

- 로깅

람다 표현식과 스트림 디버깅 방법을 알아보자

스택 트레이스

람다의 경우 여러 라이브러리가 연계되어 처리되므로 스택 트레이스가 다소 복잡하게 보일 수 있다.

그런데 람다를 메소드로 캡슐화를 하면 해당 메소드 명으로 스택 트레이스가 출력되므로 디버깅이 유용해진다.

AS-IS

```
Exception in thread "main" java.lang.NullPointerException
    at Debugging.lambda$main$0(Debugging.java:6)  <-- 여기 $0은 무슨 의미일까?
    at Debugging$$Lambda$5/284720968.apply(Unknown Source)
    at java.util.stream.ReferencePipeline$3$1.accept(ReferencePipeline
        .java:193)
    at java.util.Spliterators$ArraySpliterator.forEachRemaining(Spliterators
        .java:948)
...
```

TO-BE

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
    at Debugging.divideByZero(Debugging.java:10)<-- 스택 트레이스에 divideByZero가 보인다.
    at Debugging$$Lambda$1/999966131.apply(Unknown Source)
    at java.util.stream.ReferencePipeline$3$1.accept(ReferencePipeline
        .java:193)
...
```

로깅

파이프라인의 흐름에 따라 요소의 상태가 어떤지 궁금하면 peek 중간 연산을 사용하면 된다.

```
IntStream.rangeClosed(1, 10)
                .map(num -> num * 2)
                .peek(num -> System.out.println("#1   " + num))
                .map(num -> num + 1)
                .peek(num -> System.out.println("#2   " + num))
                .boxed().collect(Collectors.toList());
```