

10장. 람다를 이용한 도메인 전용 언어

DSL(domain-specific languages)은 특정 비즈니스 도메인 문제를 해결하려고 만든 언어다.

ex) 회계 전용 소프트웨어 애플리케이션을 개발한다고 가정 통장 입출금 내역서, 계좌 통합 등 표현할 수 있는 DSL을 만들 수 있다.

→ 자바에서는 도메인을 표현할 수 있는 클래스와 메서드 집합이 필요하다. DSL이란 특정 비즈니스 도메인을 인터페이스로 만든 API라고 생각할 수 있다.

다음의 두 가지 필요성을 생각하면서 DSL을 개발해야 한다.

- 의사 소통의 왕: 우리의 코드의 의도가 명확히 전달되어야 하며 프로그래머가 아닌 사람도 이해할 수 있어야 한다. 이런 방식으로 코드가 비즈니스 요구사항에 부합하는지 확인 할 수 있다.
- 한 번 코드를 구현하지만 여러 번 읽는다: 가독성은 유지보수의 핵심이다. 즉 항상 우리의 동료가 쉽게 이해할 수 있도록 코드를 구현해야 한다.

DSL 장점

- 간결함: API는 비즈니스 로직을 간편하게 캡슐화하므로 반복을 피할 수 있고 코드를 간결하게 만들 수 있다.
- 가독성: 도메인 영역의 용어를 사용하므로 비 도메인 전문가도 코드를 쉽게 이해할 수 있다. 결과적으로 다양한 조직 구성원 간에 코드와 도메인 영역이 공유될 수 있다.
- 유지보수: 잘 설계된 DSL로 구현한 코드는 쉽게 유지보수하고 바꿀 수 있다. 유지보수는 비즈니스 관련 코드 즉 가장 빈번히 바뀌는 애플리케이션 부분에 특히 중요하다.
- 높은 수준의 추상화: DSL은 도메인과 같은 추상화 수준에서 동작하므로 도메인의 문제와 직접적으로 관련되지 않은 세부 사항을 숨긴다.
- 집중: 비즈니스 도메인의 규칙을 표현할 목적으로 설계된 언어이므로 프로그래머가 특정 코드에 집중할 수 있다. 결과적으로 생산성이 좋아진다.
- 관심사분리: 지정된 언어로 비즈니스 로직을 표현함으로 애플리케이션의 인프라구조와 관련된 문제와 독립적으로 비즈니스 관련된 코드에서 집중하기가 용이하다. 결과적으로 유지보수가 쉬운 코드를 구현한다.

DSL 단점

- DSL 설계의 어려움: 간결하게 제한적인 언어에 도메인 지식을 담는 것이 쉬운 작업은 아니다.
- 개발 비용: 코드에 DSL을 추가하는 작업은 초기 프로젝트에 많은 비용과 시간이 소모되는 작업이다. 또한 DSL 유지보수와 변경은 프로젝트에 부담을 주는 요소다.
- 추가 우회 계층: DSL 추가적인 계층으로 도메인 모델을 감싸며 이 때 계층을 최대한 작게 만 들어 성능 문제를 회피한다.
- 새로 배워야 하는 언어: 요즘에는 한 프로젝트에도 여러가지 언어를 사용하는 추세다. 하지만 DSL을 프로젝트에 추가하면서 팀이 배워야 하는 언어가 한 개 더 늘어난다는 부담이 있다. 여러 비즈니스 도메인을 다루는 개별 DSL을 사용하는 상황이라면 이들을 유기적으로 동작하도록 합치는 일은 쉬운 일이 아니다.
- 호스팅 언어 한계: 일부 자바 같은 범용 프로그래밍 언어는 장황하고 엄격한 문법을 가졌다. 이런 언어로는 사용자 친화적 DSL을 만들기가 힘들다. 자바 8의 람다 표현식은 이 문제를 해결 할 강력한 새 도구다.

DSL 카테고리를 구분하는 가장 흔한 방법은 마틴 파울러(Martin Fowler)가 소개한 방법으로 내부 DSL과 외부 DSL을 나누는 것이다.

- **내부 DSL:** 순수 자바 코드같은 기존 호스팅 언어를 기반으로 구현
- **외부 DSL:** 호스팅 언어와는 독립적으로 자체의 문법을 가진다.
- **다중 DSL**

빨간 블록 안에 있는 코드가 코드의 잡음이다.

```
1 List<String> numbers = Arrays.asList("one", "two", "three");
2 numbers.forEach( new Consumer<String>() {
3     @Override
4     public void accept( String s ) {
5         System.out.println(s);
6     }
7 });

```

자바 8에서는 이런 잡음이 많이 줄어든다.



```
1 numbers.forEach(s → System.out.println(s));
```



```
1 numbers.forEach(System.out :: println);
```

자바 문법이 큰 문제가 아니라면 순수 자바로 DSL을 구현함으로 다음과 같은 장점을 얻을 수 있다.

- 기존 자바 언어를 이용하면 외부 DSL에 비해 새로운 패턴과 기술을 배워 DSL을 구현하는 노력이 현저하게 줄어든다.
- 순수 자바로 DSL을 구현하면 나머지 코드와 함께 DSL을 컴파일할 수 있다. 따라서 다른 언어의 컴파일러를 이용하거나 외부 DSL을 만드는 도구를 사용할 필요가 없으므로 추가로 비용이 들지 않는다.
- 여러분의 개발 팀이 새로운 언어를 배우거나 또는 익숙하지 않고 복잡한 외부 도구를 배울 필요가 없다.
- DSL 사용자는 기존의 자바 IDE를 이용해 자동 완성, 자동 리팩터링 같은 기능을 그대로 즐길 수 있다.
- 한 개의 언어로 한 개의 도메인 또는 여러 도메인을 대응하지 못해 추가로 DSL을 개발해야 하는 상황에서 자바를 이용한다면 추가 DSL을 쉽게 합칠 수 있다.

요즘 JVM에서 실행되는 언어는 100개 넘는다.

ex) 스칼라, 루비, 코틀린, 실론, JRuby, Jython

스칼라 문법

```
● ● ●  
1 def times(i: Int, f: ⇒ Unit): Unit = {  
2   f // f 함수 실행  
3   if (i > 1) times(i -1, f) // 횟수가 양수면 횟수를 감소시켜  
4 } // 재귀적으로 times를 실행한다.
```

```
● ● ●
```

```
1 times(3, println("Hello World"))
```

- 스칼라에서는 i가 아주 큰 숫자라 하더라도 자바에서처럼 스택 오버플로 문제가 발생하지 않는다.
- 스칼라는 꼬리 호출 최적화를 통해 times 함수 호출을 스택에 추가하지 않기 때문이다.

- 예제에서 확인했듯이 결과적으로 문법적 잡음이 전혀없으며 개발자가 아닌 사람도 코드를 쉽게 이해할 수 있다. (잡음은 없어 보이긴 하는데.. 코드가 쉽게 이해가 안되었다.)
- 새로운 프로그래밍 언어를 배우거나 또는 팀의 누군가가 이미 해당 기술을 가지고 있어야한다. 멋진 DSL을 만들려면 이미 기존 언어의 고급 기능을 사용할 수 있는 충분한 지식이 필요하기 때문이다.
- 두 개 이상의 언어가 혼재하므로 여러 컴파일러로 소스를 빌드하도록 빌드 과정을 개선해야 한다.
- 마지막으로 JVM에서 실행되는 거의 모든 언어가 자바와 백 퍼센트 호환을 주장하고 있지만 자바와 호환성이 완벽하지 않을 때가 많다.

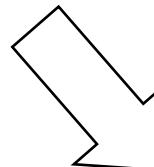
- 자신만의 문법과 구문으로 새 언어를 설계해야 한다.
- 새 언어를 파싱하고, 파서의 결과를 분석하고, 외부 DSL을 실행할 코드를 만들어야 한다.
- 하지만 구현만 한다면 우리에게 필요한 특성을 완벽하게 제공하는 언어를 설계할 수 있다는 것이 장점이다.
- 인프라구조 코드와 외부 DSL로 구현한 비즈니스 코드를 명확하게 분리한다는 것도 장점이다. 하지만 이 분리로 인해 DSL과 호스트 언어 사이에 인공 계층이 생기므로 이는 양날의 검과 같다.

자바의 새로운 기능의 장점을 적용한 첫 API는 네이티브 자바 API 자신이다.

자바 8의 Comparator 인터페이스가 네이티브 자바 API의 재사용성과 메서드 결합도를 어떻게 높였는지 확인해보자.

사람(Persons)을 가리키는 객체 목록을 가지고 있는데 사람의 나이를 기준으로 객체를 정렬한다고 가정하자.

```
● ● ●  
1 Collections.sort(persons, new Comparator<Person>() {  
2     public int compare(Person p1, Person p2) {  
3         return p1.getAge() - p2.getAge();  
4     }  
5 });
```



```
● ● ●  
1 Collections.sort(people, (p1, p2) → p1.getAge() - p2.getAge());
```

정적으로 Comparator.comparing 메서드를 임포트해 위 예제를 다음처럼 구현할 수 있다.



```
1 Collections.sort(people, comparing(p → p.getAge()));
```



```
1 Collections.sort(people, comparing(Person::getAge));
```



```
1 public class Stock {  
2  
3     private String symbol;  
4     private String market;  
5  
6     public String getSymbol() {  
7         return symbol;  
8     }  
9  
10    public void setSymbol( String symbol ) {  
11        this.symbol = symbol;  
12    }  
13  
14    public String getMarket() {  
15        return market;  
16    }  
17  
18    public void setMarket( String market ) {  
19        this.market = market;  
20    }  
21  
22    @Override  
23    public String toString() {  
24        return String.format("Stock[symbol=%s, market=%s]", symbol, market);  
25    }  
26  
27 }
```

DSL은 특정 도메인 모델에 적용할 친화적이고 가독성 높은 API를 제공한다.

예제 도메인 모델은 세 가지로 구성된다. 첫 번째는 주어진 시장에 주식 가격을 모델링하는 순수 자바 빈즈다.

자바로 DSL을 만드는 패턴과 기법



```
1 public class Trade {  
2  
3     public enum Type {  
4         BUY,  
5         SELL  
6     }  
7  
8     private Type type;  
9     private Stock stock;  
10    private int quantity;  
11    private double price;  
12  
13    public Type getType() {  
14        return type;  
15    }  
16  
17    public void setType(Type type) {  
18        this.type = type;  
19    }  
20  
21    public int getQuantity() {  
22        return quantity;  
23    }  
24  
25}
```



```
1     public void setQuantity(int quantity) {  
2         this.quantity = quantity;  
3     }  
4  
5     public double getPrice() {  
6         return price;  
7     }  
8  
9     public void setPrice(double price) {  
10        this.price = price;  
11    }  
12  
13    public Stock getStock() {  
14        return stock;  
15    }  
16  
17    public void setStock(Stock stock) {  
18        this.stock = stock;  
19    }  
20  
21    public double getValue() {  
22        return quantity * price;  
23    }  
24  
25    @Override  
26    public String toString() {  
27        return String.format("Trade[type=%s, stock=%s, quantity=%d, price=%.2f]", type, stock, quantity, price);  
28    }  
29  
30 }  
31
```

두 번째는 주어진 가격에서 주어진 양의 주식을 사거나 파는 거래다.

자바로 DSL을 만드는 패턴과 기법



```
1 public class Order {  
2  
3     private String customer;  
4     private List<Trade> trades = new ArrayList<>();  
5  
6     public void addTrade( Trade trade ) {  
7         trades.add( trade );  
8     }  
9  
10    public String getCustomer() {  
11        return customer;  
12    }  
13  
14    public void setCustomer( String customer ) {  
15        this.customer = customer;  
16    }  
17  
18    public double getValue() {  
19        return trades.stream().mapToDouble( Trade ::getValue ).sum();  
20    }  
21  
22    @Override  
23    public String toString() {  
24        String strTrades = trades.stream().map(t → " " + t).collect(Collectors.joining("\n", "[\n", "\n]"));  
25        return String.format("Order[customer=%s, trades=%s]", customer, strTrades);  
26    }  
27  
28 }  
29 }
```

마지막으로 고객이 요청한 한 개 이상의 거래의 주문이다.

자바로 DSL을 만드는 패턴과 기법



```
1 Order order = new Order();
2     order.setCustomer("BigBank");
3
4     Trade trade1 = new Trade();
5     trade1.setType(Trade.Type.BUY);
6
7     Stock stock1 = new Stock();
8     stock1.setSymbol("IBM");
9     stock1.setMarket("NYSE");
10
11    trade1.setStock(stock1);
12    trade1.setPrice(125.00);
13    trade1.setQuantity(80);
14    order.addTrade(trade1);
15
16    Trade trade2 = new Trade();
17    trade2.setType(Trade.Type.BUY);
18
19    Stock stock2 = new Stock();
20    stock2.setSymbol("GOOGLE");
21    stock2.setMarket("NASDAQ");
22
23    trade2.setStock(stock2);
24    trade2.setPrice(375.00);
25    trade2.setQuantity(50);
26    order.addTrade(trade2);
27
28    System.out.println("Plain:");
29    System.out.println(order);
30
```

위 코드는 상당히 장황한 편이다. 비개발자인 도메인 전문가가 위 코드를 이해하고 검증하기를 기대할 수 없기 때문이다.

조금 더 직접적이고, 직관적으로 도메인 모델을 반영할 수 있는 DSL이 필요하다.



```
1 Order order = forCustomer("BigBank")
2         .buy(80)
3         .stock("IBM")
4         .on("NYSE")
5         .at(125.00)
6         .sell(50)
7         .stock("GOOGLE")
8         .on("NASDAQ")
9         .at(375.00)
10        .end();
```

결과를 달성하려면 어떻게 DSL을 구현해야 할까?

플루언트 API로 도메인 객체를 만드는 몇 개의 빌더를 구현해야 한다.

플루언트 API

- 메소드 체이닝을 지원하는 디자인 패턴
- 가독성 높은 객체지향 API 구현 가능

메소드 체이닝

- OOP에서 여러 메소드를 이어서 호출하는 문법
- 메소드가 객체(주로 this)를 반환함으로써 가능하게 됨

```
● ● ●  
1 public class MethodChainingOrderBuilder {  
2  
3     public final Order order = new Order(); // 빌더로 감싼 주문  
4  
5     private MethodChainingOrderBuilder(String customer) {  
6         order.setCustomer(customer);  
7     }  
8  
9     public static MethodChainingOrderBuilder forCustomer(String customer) {  
10        return new MethodChainingOrderBuilder(customer); // 고객의 주문을 만드는 정적  
11        // 팩토리 메서드  
12    }  
13  
14    public TradeBuilder buy(int quantity) {  
15        return new TradeBuilder(this, Trade.Type.BUY, quantity); // 주식을 사는 TradeBuilder 만들기  
16    }  
17  
18    public TradeBuilder sell(int quantity) {  
19        return new TradeBuilder(this, Trade.Type.SELL, quantity); // 주식을 파는 TradeBuilder 만들기  
20    }  
21  
22    private MethodChainingOrderBuilder addTrade(Trade trade) {  
23        order.addTrade(trade); // 주문에 주식 추가  
24        return this; // 유연하게 추가 주문을 만들어 추가할 수 있도록 주문 빌더 자체를 반환  
25    }  
26  
27    public Order end() {  
28        return order; // 주문 만들기를 종료하고 반환  
29    }  
30}
```

주문 빌더의 `buy()`, `sell()` 메서드는 다른 주문을 만들어 추가할 수 있도록 자신을 만들어 반환한다.



```
1 public static class TradeBuilder {  
2  
3     private final MethodChainingOrderBuilder builder;  
4     public final Trade trade = new Trade();  
5  
6     private TradeBuilder(MethodChainingOrderBuilder builder, Trade.Type type, int quantity) {  
7         this.builder = builder;  
8         trade.setType(type);  
9         trade.setQuantity(quantity);  
10    }  
11  
12    public StockBuilder stock(String symbol) {  
13        return new StockBuilder(builder, trade, symbol);  
14    }  
15  
16}  
17
```



```
1 public static class StockBuilder {
2
3     private final MethodChainingOrderBuilder builder;
4     private final Trade trade;
5     private final Stock stock = new Stock();
6
7     private StockBuilder(MethodChainingOrderBuilder builder, Trade trade, String symbol) {
8         this.builder = builder;
9         this.trade = trade;
10        stock.setSymbol(symbol);
11    }
12
13    public TradeBuilderWithStock on(String market) {
14        stock.setMarket(market);
15        trade.setStock(stock);
16        return new TradeBuilderWithStock(builder, trade);
17    }
18
19 }
20
```

빌더를 계속 이어가려면 Stock 클래스의 인스턴스를 만드는 TradeBuilder의 공개 메서드를 이용해야 한다.

StockBuilder는 주식의 시장을 지정하고, 거래에 주식을 추가하고, 최종 빌더를 반환하는 on() 메서드 한 개를 정의한다.



```
1 public static class TradeBuilderWithStock {  
2  
3     private final MethodChainingOrderBuilder builder;  
4     private final Trade trade;  
5  
6     public TradeBuilderWithStock(MethodChainingOrderBuilder builder, Trade trade) {  
7         this.builder = builder;  
8         this.trade = trade;  
9     }  
10  
11    public MethodChainingOrderBuilder at(double price) {  
12        trade.setPrice(price);  
13        return builder.addTrade(trade);  
14    }  
15  
16 }  
17
```

거래되는 주식의 단위 가격을 설정
한 다음 원래 주문 빌더를 반환한다.

여러 빌드 클래스 특히 두 개의 거래 빌더를 따로 만들었으므로 사용자가 미리 지정된 절차에 따라 플루언트 API의 메서드를 호출하도록 강제한다. 덕분에 사용자가 다음 거래를 설정하기 전에 기존 거래를 올바로 설정하게 된다.

중첩된 함수 DSL 패턴은 이름에서 알 수 있듯이 다른 함수 안에 함수를 이용해 도메인 모델을 만든다.



```
1 Order order = order("BigBank",
2                 buy(80,
3                     stock("IBM", on("NYSE")),
4                     at(125.00)),
5                 sell(50,
6                     stock("GOOGLE", on("NASDAQ")),
7                     at(375.00))
8 );
```

중첩된 함수 이용

```

1 public class NestedFunctionOrderBuilder {
2
3     public static Order order(String customer, Trade... trades) {
4         Order order = new Order(); // 해당 고객의 주문 만들기
5         order.setCustomer(customer);
6         Stream.of(trades).forEach(order::addTrade); // 주문에 모든 거래 추가
7         return order;
8     }
9
10    public static Trade buy(int quantity, Stock stock, double price) {
11        return buildTrade(quantity, stock, price, Trade.Type.BUY); // 주식 매수 거래 만들기
12    }
13
14    public static Trade sell(int quantity, Stock stock, double price) {
15        return buildTrade(quantity, stock, price, Trade.Type.SELL); // 주식 매도 거래 만들기
16    }
17
18    private static Trade buildTrade(int quantity, Stock stock, double price, Trade.Type buy) {
19        Trade trade = new Trade();
20        trade.setQuantity(quantity);
21        trade.setType(buy);
22        trade.setStock(stock);
23        trade.setPrice(price);
24        return trade;
25    }

```

```

1     public static double at(double price) {
2         return price; // 거래된 주식의 단가를 정의하는 더미 메서드
3     }
4
5     public static Stock stock(String symbol, String market) {
6         Stock stock = new Stock(); // 거래된 주식 만들기
7         stock.setSymbol(symbol);
8         stock.setMarket(market);
9         return stock;
10    }
11
12    public static String on(String market) {
13        return market; // 주식이 거래된 시장을 정의하는 더미 메서드 정의
14    }
15
16 }

```



```
1 Order order = LambdaOrderBuilder.order( o → {  
2     o.forCustomer( "BigBank" );  
3     o.buy( t → {  
4         t.quantity(80);  
5         t.price(125.00);  
6         t.stock(s → {  
7             s.symbol("IBM");  
8             s.market("NYSE");  
9         });  
10    });  
11    o.sell( t → {  
12        t.quantity(50);  
13        t.price(375.00);  
14        t.stock(s → {  
15            s.symbol("GOOGLE");  
16            s.market("NASDAQ");  
17        });  
18    });  
19});
```

다음 DSL 패턴은 람다 표현식으로 정의한 함수 시퀀스를 사용한다.

이런 DSL을 만들려면 람다 표현식을 받아 실행해 도메인 모델을 만들어 내는 여러 빌더를 구현해야 한다.

메서드 체인 패턴에는 주문을 만드는 최상위 수준의 빌더를 가졌지만 이번에는 Consumer 객체를 빌더가 인수로 받음으로 DSL 사용자가 람다 표현식으로 인수를 구현할 수 있게 했다.

람다 표현식을 이용한 함수 시퀀싱



```

1 public class LambdaOrderBuilder {
2
3     private Order order = new Order(); // 빌더로 주문을 감쌈
4
5     public static Order order(Consumer<LambdaOrderBuilder> consumer) {
6         LambdaOrderBuilder builder = new LambdaOrderBuilder();
7         consumer.accept(builder); // 주문 빌더로 전달된 람다 표현식 실행
8         return builder.order; // OrderBuilder의 Consumer를 실행해
9     } // 만들어진 주문을 반환
10
11    public void forCustomer(String customer) {
12        order.setCustomer(customer); // 주문을 요청한 고객 설정
13    }
14

```



```

1 public void buy(Consumer<TradeBuilder> consumer) {
2     trade(consumer, Trade.Type.BUY); // 주식 매수 주문을 만들도록 TradeBuilder 소비
3 }
4
5 public void sell(Consumer<TradeBuilder> consumer) {
6     trade(consumer, Trade.Type.SELL); // 주식 매도 주문을 만들도록 TradeBuilder 소비
7 }
8
9 private void trade(Consumer<TradeBuilder> consumer, Trade.Type type) {
10    TradeBuilder builder = new TradeBuilder();
11    builder.trade.setType(type);
12    consumer.accept(builder); // TradeBuilder로 전달할 람다 표현식 실행
13    order.addTrade(builder.trade); // TradeBuilder의 Consumer를 실행해
14 } // 만든 거래를 주문에 추가
15
16

```

주문 빌더의 `buy()`, `sell` 메서드는 두 개의 `Consumer<TradeBuilder>` 람다 표현식을 받는다.
이 람다 표현식을 실행하면 다음처럼 주식 매수, 주식 매도 거래가 만들어진다.

```
● ● ●  
1 public static class TradeBuilder {  
2  
3     private Trade trade = new Trade();  
4  
5     public void quantity(int quantity) {  
6         trade.setQuantity(quantity);  
7     }  
8  
9     public void price(double price) {  
10        trade.setPrice(price);  
11    }  
12  
13    public void stock(Consumer<StockBuilder> consumer) {  
14        StockBuilder builder = new StockBuilder();  
15        consumer.accept(builder);  
16        trade.setStock(builder.stock);  
17    }  
18  
19}
```

```
● ● ●  
1 public static class StockBuilder {  
2  
3     private Stock stock = new Stock();  
4  
5     public void symbol(String symbol) {  
6         stock.setSymbol(symbol);  
7     }  
8  
9     public void market(String market) {  
10        stock.setMarket(market);  
11    }  
12  
13}
```

마지막으로 TradeBuilder는 세 번째 빌더의 Consumer 즉 거래된 주식을 받는다.

지금까지 살펴본 것처럼 세가지 DSL 패턴 각자가 장단점을 갖고 있다. 하지만 한 DSL에 한 개의 패턴만 사용하라는 법은 없다.

```
1 Order order =
2     forCustomer("BigBank", // 최상위 수준 주문의 속성을 지정하는 중첩 함수
3                 buy(t → t.quantity(80) // 한 개의 주문을 만드는 람다 표현식
4                     .stock("IBM") // 거래 객체를 만드는 람다 표현식 바디의 메서드 체인
5                     .on("NYSE")
6                     .at(125.00)),
7                 sell(t → t.quantity(50)
8                     .stock("GOOGLE")
9                     .on("NASDAQ")
10                    .at(375.00)));
```

```
1 public class MixedBuilder {  
2  
3     public static Order forCustomer(String customer, TradeBuilder... builders) { // 중첩된 함수 패턴  
4         Order order = new Order();  
5         order.setCustomer(customer);  
6         Stream.of(builders).forEach(b → order.addTrade(b.trade));  
7         return order;  
8     }  
9  
10    public static TradeBuilder buy(Consumer<TradeBuilder> consumer) { // 람다 표현식  
11        return buildTrade(consumer, Trade.Type.BUY);  
12    }  
13  
14    public static TradeBuilder sell(Consumer<TradeBuilder> consumer) { // 람다 표현식  
15        return buildTrade(consumer, Trade.Type.SELL);  
16    }  
17  
18    private static TradeBuilder buildTrade(Consumer<TradeBuilder> consumer, Trade.Type buy) { // 람다 표현식  
19        TradeBuilder builder = new TradeBuilder();  
20        builder.trade.setType(buy);  
21        consumer.accept(builder);  
22        return builder;  
23    }
```

마지막으로 헬퍼 클래스 TradeBuilder와 StockBuilder는 내부적으로
메서드 체인 패턴을 구현해 플루언트 API를 제공한다.

```
1 public static class TradeBuilder {  
2  
3     private Trade trade = new Trade();  
4  
5     public TradeBuilder quantity(int quantity) {  
6         trade.setQuantity(quantity);  
7         return this;  
8     }  
9  
10    public TradeBuilder at(double price) {  
11        trade.setPrice(price);  
12        return this;  
13    }  
14  
15    public StockBuilder stock(String symbol) {  
16        return new StockBuilder(this, trade, symbol);  
17    }  
18  
19 }
```

```
1 public static class StockBuilder {  
2  
3     private final TradeBuilder builder;  
4     private final Trade trade;  
5     private final Stock stock = new Stock();  
6  
7     private StockBuilder(TradeBuilder builder, Trade trade, String symbol) {  
8         this.builder = builder;  
9         this.trade = trade;  
10        stock.setSymbol(symbol);  
11    }  
12  
13    public TradeBuilder on(String market) {  
14        stock.setMarket(market);  
15        trade.setStock(stock);  
16        return builder;  
17    }  
18  
19 }
```

DSL에 메서드 참조 사용하기

지금까지는 람다 표현식을 사용했지만 Comparator와 스트림 API에서 확인했듯이 메서드 참조를 이용하면 많은 DSL의 가독성을 높일 수 있다.



```
1 public class Tax {  
2  
3     public static double regional(double value) {  
4         return value * 1.1;  
5     }  
6  
7     public static double general(double value) {  
8         return value * 1.3;  
9     }  
10  
11    public static double surcharge(double value) {  
12        return value * 1.05;  
13    }  
14  
15 }
```



```
1 public static double calculate(Order order, boolean useRegional,  
                                boolean useGeneral, boolean useSurcharge) {  
2  
3     double value = order.getValue();  
4  
5     if (useRegional) {  
6         value = Tax.regional(value);  
7     }  
8     if (useGeneral) {  
9         value = Tax.general(value);  
10    }  
11    if (useSurcharge) {  
12        value = Tax.surcharge(value);  
13    }  
14    return value;  
15 }
```

주문의 총 합에 0개 이상의 세금을 추가해 최종값을 계산하는 기능

세금을 적용할 것인지 결정하는 불리언 플래그를 인수로 받는 정적 메서드를 이용해 간단하게 해결할 수 있다.

이제 다음 코드처럼 지역 세금과 추가 요금을 적용하고 일반 세금을 뺀 주문의 최종값을 계산 할 수 있다.



```
1 double value = TaxCalculator.calculate(order, true, false, true);
```

문제점

- 불리언 변수의 올바른 순서를 기억하기 어렵다.
- 어떤 세금이 적용되었는지 파악하기 어렵다.

```
1 private boolean useRegional;
2 private boolean useGeneral;
3 private boolean useSurcharge;
4
5 public TaxCalculator withTaxRegional() {
6     useRegional = true;
7     return this;
8 }
9
10 public TaxCalculator withTaxGeneral() {
11     useGeneral= true;
12     return this;
13 }
14
15 public TaxCalculator withTaxSurcharge() {
16     useSurcharge = true;
17     return this;
18 }
19
20 public double calculate(Order order) {
21     return calculate(order, useRegional, useGeneral, useSurcharge);
22 }
```

```
1 value = new TaxCalculator()
2
3
4     .withTaxRegional()
5     .withTaxSurcharge()
6     .calculate(order);
```

다음 코드처럼 TaxCalculator는 지역 세금과 추가 요금은 주문에 추가하고 싶다는 점을 명확하게 보여준다.

DSL에 메서드 참조 사용하기



```
1 private boolean useRegional;
2 private boolean useGeneral;
3 private boolean useSurcharge;
4
5 public TaxCalculator withTaxRegional() {
6     useRegional = true;
7     return this;
8 }
9
10 public TaxCalculator withTaxGeneral() {
11    useGeneral= true;
12    return this;
13 }
14
15 public TaxCalculator withTaxSurcharge() {
16     useSurcharge = true;
17     return this;
18 }
19
20 public double calculate(Order order) {
21     return calculate(order, useRegional, useGeneral, useSurcharge);
22 }
```

리팩토링 전



```
1 // 주문값에 적용된 모든 세금을 계산하는 함수
2 public DoubleUnaryOperator taxFunction = d → d;
3
4 public TaxCalculator with(DoubleUnaryOperator f) {
5     // 새로운 세금 계산 함수를 얹어서 인수로 전달된 함수와 현재 함수를 합침
6     taxFunction = taxFunction.andThen(f);
7     return this;
8 }
9
10 public double calculateF(Order order) {
11     // 주문의 총 합에 세금 계산 함수를 적용해 최종 주문값을 계산
12     return taxFunction.applyAsDouble(order.getValue());
13 }
```

리팩토링 후



```
1 value = new TaxCalculator().with(Tax::regional)
2                               .with(Tax::surcharge)
3                               .calculateF(order);
```

패턴 이름	장점	단점
메서드 체인	<ul style="list-style-type: none"> • 메서드 이름이 키워드 인수 역할을 한다. • 선택형 파라미터와 잘 동작한다. • DSL 사용자가 정해진 순서로 메서드를 호출하도록 강제할 수 있다. • 정적 메서드를 최소화하거나 없앨 수 있다. • 문법적 잡음을 최소화한다. 	<ul style="list-style-type: none"> • 구현이 장황하다. • 빌드를 연결하는 접착 코드가 필요하다. • 들여쓰기 규칙으로만 도메인 객체 계층을 정의한다.
중첩 함수	<ul style="list-style-type: none"> • 구현의 장황함을 줄일 수 있다. • 함수 중첩으로 도메인 객체 계층을 반영한다. 	<ul style="list-style-type: none"> • 정적 메서드의 사용이 빈번하다. • 이름이 아닌 위치로 인수를 정의한다. • 선택형 파라미터를 처리할 메서드 오버로딩이 필요하다.
람다를 이용한 함수	<ul style="list-style-type: none"> • 선택형 파라미터와 잘 동작한다. 	<ul style="list-style-type: none"> • 구현이 장황하다.
시퀀싱	<ul style="list-style-type: none"> • 정적 메서드를 최소화하거나 없앨 수 있다. • 람다 중첩으로 도메인 객체 계층을 반영한다. • 빌더의 접착 코드가 없다. 	<ul style="list-style-type: none"> • 람다 표현식으로 인한 문법적 잡음이 DSL에 존재한다.

패턴 이름	장점	단점
메서드 체인	<ul style="list-style-type: none"> • 메서드 이름이 키워드 인수 역할을 한다. • 선택형 파라미터와 잘 동작한다. • DSL 사용자가 정해진 순서로 메서드를 호출하도록 강제할 수 있다. • 정적 메서드를 최소화하거나 없앨 수 있다. • 문법적 잡음을 최소화한다. 	<ul style="list-style-type: none"> • 구현이 장황하다. • 빌드를 연결하는 접착 코드가 필요하다. • 들여쓰기 규칙으로만 도메인 객체 계층을 정의한다.
중첩 함수	<ul style="list-style-type: none"> • 구현의 장황함을 줄일 수 있다. • 함수 중첩으로 도메인 객체 계층을 반영한다. 	<ul style="list-style-type: none"> • 정적 메서드의 사용이 빈번하다. • 이름이 아닌 위치로 인수를 정의한다. • 선택형 파라미터를 처리할 메서드 오버로 딩이 필요하다.
람다를 이용한 함수	<ul style="list-style-type: none"> • 선택형 파라미터와 잘 동작한다. 	<ul style="list-style-type: none"> • 구현이 장황하다.
시퀀싱	<ul style="list-style-type: none"> • 정적 메서드를 최소화하거나 없앨 수 있다. • 람다 중첩으로 도메인 객체 계층을 반영한다. • 빌더의 접착 코드가 없다. 	<ul style="list-style-type: none"> • 람다 표현식으로 인한 문법적 잡음이 DSL에 존재한다.

세 가지 유명한 자바 라이브러리에 지금까지 살펴본 패턴이 얼마나 사용되고 있는지 살펴보자.

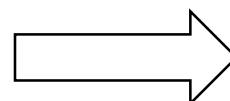
SQL 매팅 도구 동작 주도(behavior-driven) 개발 프레임워크

엔터프라이즈 통합 패턴(Enterprise Integration Patterns)

JOOQ는 SQL을 구현하는 내부적 DSL로 자바에 직접 내장된 형식 언어다.



```
1 SELECT * FROM BOOK  
2 WHERE BOOK.PUBLISHED_IN = 2016  
3 ORDER BY BOOK.TITLE
```



```
1 create.selectFrom(BOOK)  
2       .where(BOOK.PUBLISHED_IN.eq(2016))  
3       .orderBy(BOOK.TITLE)
```

SQL 질의

JOOQ DSL을 이용한 질의

스트림 API와 조합해 사용할 수 있다는 것이 JOOQ DSL의 또 다른 장점이다.

```
1 Class.forName("org.h2.Driver");
2 try (Connection c = // SQL 데이터베이스 연결 만들기
3       getConnection("jdbc:h2:~/sql-goodies-with-mapping", "sa", ""));
4     DSL.using(c) // 만들어진 데이터베이스 연결을 이용해 JOOQ SQL문 시작
5       .select(BOOK.AUTHOR, BOOK.TITLE
6             .where(BOOK.PUBLISHDATE.eq(2016))
7             .orderBy(BOOK.TITLE)
8       .fetch() // JOOQ DSL로 SQL문 정의
9       .stream() // 데이터베이스에서 데이터 가져오기 JOOQ문은 여기서 종료
10      .collect(groupingBy( // 스트림 API로 데이터베이스에서 가져온 데이터 처리 시작
11        r → r.getValue(BOOK.AUTHOR),
12        LinkedHashMap::new,
13        mapping(r → r.getValue(BOOK.TITLE), toList())))
14          .forEach((author, titles) → // 저자의 이름 목록과 각 저자가 집필한 책들을 출력
15        System.out.println(author + " is author of " + titles));
16    }
```

큐컴버(Cucumber)

동작 주도 개발(Behavior-driven development (BDD))은 테스트 주도 개발의 확장으로 다양한 비즈니스 시나리오를 구조적으로 서술하는 간단한 도메인 전용 스크립팅 언어를 사용한다.

큐컴버(Cucumber)는 다른 BDD 프레임워크와 마찬가지로 이들 명령문을 실행할 수 있는 테스트 케이스로 변환한다.



```
1 Feature: Buy stock
2   Scenario: Buy 10 IBM stocks
3     Given the price of a "IBM" stock is 125$
4     When I buy 10 "IBM"
5     Then the order value should be 1250$
```

큐컴버는 세 가지로 구분되는 개념을 사용한다.

- 전제 조건 정의(Given)
- 시험하려는 도메인 객체의 실질 호출(When)
- 테스트 케이스의 결과를 확인하는 어설션(Then)

큐컴버(Cucumber)



```

1 public class BuyStocksSteps {
2     private Map<String, Integer> stockUnitPrices = new HashMap();
3     private Order order = new Order();
4
5     // 시나리오의 전제 조건인 주식 단가 정의
6     @Given("^the price of a \"(.*)\" stock is (\\d+) \\$")
7     public void setUnitPrice(String stockName, int unitPrice) {
8         stockUnitValues.put(stockName, unitPrice);
9     }
10    // 테스트 대상인 도메이 모델에 행할 액션 정의
11    @When("^I buy (\\d+) \"(.*)\"$")
12    public void buyStocks(int quantity, String stockName) {
13        Trade trade = new Trade(); // 적절하게 도메인 모델 도출
14        trade.setType(Trade.Type.BUY);
15
16        Stock stock = new Stock();
17        stock.setSymbol(stockName);
18
19        trade.setStock(stock);
20        trade.setPrice(stockUnitPrices.get(stockName));
21        trade.setQuantity(quantity);
22        order.addTrade(trade);
23    }
24
25    @Then("^the order value should be (\\d+)\\$")
26    public void checkOrderValue(int expectedValue) {
27        assertEquals(expectedValue, order.getValue());
28    }
29}

```

10.3절의 앞부분에서 소개했던 주식 거래 도메인 모델을 이용해 큐컴버로 주식 거래 주문의 값이 제대로 계산되었는지 확인하는 테스트 케이스를 개발할 수 있다.

자바 8이 람다 표현식을 지원하면서 두 개의 인수 메서드(기존에 어노테이션 값을 포함한 정규 표현식과 테스트 메서드를 구현하는 람다)를 이용해 어노테이션을 제거하는 다른 문법을 큐컴버로 개발할 수 있다.



```

1 public class BuyStocksSteps implements cucumber.api.java8.en {
2     private Map<String, Integer> stockUnitPrices = new HashMap();
3     private Order order = new Order();
4
5     // 시나리오의 전제 조건인 주식 단가 정의
6     public buyStocksSteps() {
7         @Given("the price of a \"(.*)\" stock is (\\d+) \\$",
8               String stockName,int unitPrice) → {
9             stockUnitValues.put(stockName, unit Price);
10            });
11        // ... When과 Then라는 편의상 생략
12    }
13 }
14

```

다음과 같은 표기법을 이용해 테스트 시나리오를 다음처럼 다시 구현할 수 있다.