REPORT

HW 5

자바프로그래밍2 1분반

제출일 2023. 10.28

소속 컴퓨터공학과

학번 32183520

이름 이 주성

목표

- Factory Pattern과 Builder Pattern을 적용해 모든 Period Element에 대해 Phase 타입에 따른 Period Element 객체를 생성한다.
- 모든 Period를 타입에 따른 Element 객체로 생성해두고 조회하고 싶은 Phase 타입을 사용자 입력으로 받아서 출력시켜준다.

Factory Pattern

- 객체 생성 로직을 캡슐화한 후 하위 클래스에서 구체화하는 방법
- new 연산자로 직접 객체를 생성하는 것을 회피하므로 객체 생성 로직을 런타임에 변경할 수 있어 유연하다.

Builder Pattern

- 복잡한 객체 생성 과정을 Builder 클래스에게 맡기고 Builder는 객체 생성을 단계적으로 구성해 마지막 단계(build)에서 객체를 반환
- 객체 생성 과정을 추상화하고 생성 중 해당 객체에 접근하지 못하도록 해 불변 객체 생성에 유용하다.

Factory Pattern 적용

main

- 객체 생성 로직을 PhaseElementFactory에 넘긴다.
- PhaseElementFactory에 getInstance()로 객체를 반환받는다

```
// PeriodicElements.csv 파일을 load해서 PeriodicElement 리스트로 저장
List<PeriodicElement> peList =
```

```
PeriodicElementImporter.loadCSV("PeriodicElements.csv");

// peList를 for 문으로 돌면서 Element를 생성한 후 elements에 add
List<Element> elements = new ArrayList<>();
for (PeriodicElement element : peList) {
   // Element 생성 부분은 팩토리 패턴을 적용해 캡슐화
   elements.add(PhaseElementFactory.getInstance(element.getNumber(),
element.getName(), element.getSymbol(), element.getWeight(), element.getPhase()));
}
```

PhaseElementFactory

- Element 객체 생성 로직 담당 - getInstance()
- Phase에 따라 Element를 생성한다.
- 실제 생성 부분은 각 Pahse 타입에 따른 Builder를 사용한다.

```
public class PhaseElementFactory {
   public static Element getInstance(int number, String name, String symbol, double
weight, Phase phase) {

       // phase 타입에 따라 구체 클래스 객체 생성
       // - 객체 생성은 빌더 패턴 사용
       switch (phase) {
           case gas:
               return new GasElement.GasElementBuilder(number, name)
                       .phase(phase)
                       .symbol(symbol)
                       .weight(weight)
                       .build();
           case liq:
               return new LiquidElement.LiquidElementBuilder(number, name)
                       .phase(phase)
                       .symbol(symbol)
                       .weight(weight)
                       .build();
           case solid:
               return new SolidElement.SolidElementBuilder(number, name)
                       .phase(phase)
                       .symbol(symbol)
                       .weight(weight)
                       .build();
           case artificial:
               return new ArtificialElement.ArtificialElementBuilder(number, name)
```

```
                        .phase(phase)
                        .symbol(symbol)
                        .weight(weight)
                        .build();
            default:
                return null;
        }
    }
}
```

Builder Pattern 적용

Element 추상클래스

- number, name을 필수 필드로 가지고 있고 symbol, weight 필드를 optional 필드로 가지고 있다. 이 필드들은 전부 Builder 패턴으로 초기화할 것이므로 private final로 선언해준다.
- 생성 부분은 Builder가 전부 맡을 것이므로 Element의 생성자는 protected로 막아준다.
    - private이 아닌 protected로 해주는 이유는 Element는 추상클래스이고 나중에 상속받은 Element 구체 클래스들이 접근할 수 있도록 하기 위함이다.
- inner클래스로 abstract static class Builder를 선언해준다.
- 단계별로 각 필드를 설정하는 메서드를 생성한 뒤 마지막 단계인 build()를 abstract 메서드로 선언해 Builder를 구현한 클래스에서 오버라이드할 수 있도록 한다.

```
public abstract class Element {
    private final int number;       // required
    private final String name;      // required
    private final String symbol;    // optional
    private final double weight;    // optional

    // 추상 클래스를 구현한 클래스만 접근 가능하도록 접근 제한자 protected로 설정
    protected Element(ElementBuilder elementBuilder) {
        this.number = elementBuilder.number;
        this.name = elementBuilder.name;
```

```
        this.symbol = elementBuilder.symbol;
        this.weight = elementBuilder.weight;
    }

    // Getter 메서드 ...

    public abstract static class ElementBuilder {
        // required 필드만 final로 선언
        private final int number;       // required
        private final String name;      // required
        private String symbol;          // optional
        private double weight;          // optional

        // final 필드들을 받아 생성하는 생성자 선언
        public ElementBuilder(int number, String name) {
            this.number = number;
            this.name = name;
        }

        // symbol 필드 설정 (Optional)
        public ElementBuilder symbol(String symbol) {
            this.symbol = symbol;
            return this;
        }

        // weight 필드 설정 (Optional)
        public ElementBuilder weight(double weight) {
            this.weight = weight;
            return this;
        }

        // ElementBuilder 구현 클래스에서 구현하도록 abstract 메소드로 선언
        public abstract Element build();
    }
}
```

GasElement

- Phase 필드값 보유
- Element를 상속받아 구현하고 ElementBuilder를 상속받은 GasElementBuilder를 inner 클래스로 보유

```
import element.Element;
```

```java
public class GasElement extends Element {
    private final Phase phase; // required

    public Phase getPhase() {
        return phase;
    }

    private GasElement(GasElementBuilder gasElementBuilder) {
        super(gasElementBuilder);
        this.phase = gasElementBuilder.phase;
    }

    public static class GasElementBuilder extends ElementBuilder{
        private Phase phase;

        // 필수 필드인 number, name을 받아 생성
        public GasElementBuilder(int number, String name) {
            super(number, name);
        }

        // phase 설정
        public GasElementBuilder phase(Phase phase) {
            this.phase = phase;
            return this;
        }

        // build 추상 메서드 오버라이드
        @Override
        public Element build() {
            // 최종 객체 생성
            return new GasElement(this);
        }
    }
}
```

나머지 **LiqElement, SolidElemnt, ArtificialElement** 동일하게 코드 구현

실제 사용자 입력을 받아 조회하는 Main

- quit을 입력받을 때까지 무한반복해서 사용자 입력을 받음
- 입력받은 phase 값의 class와 동일한 class를 가진 element를 필터링해서 출력

```
// quit로 종료 요청시까지 반복
while (true) {
    // 유저에게 String 값을 입력받아 Phase 값으로 변환
    Phase phase = Phase.names(UserInput.getString());

    // quit 입력시 무한 루프 탈출
    if (phase == null) {
        System.out.println("검색을 종료합니다.\n");
        break;
    }

    System.out.println("===== Start Searching =====");
     // 입력받은 phase 값에 따른 class와 elements의 클래스가 동일한 값 필터링한 후 새로운 배열로 생성
    Element[] array = elements.stream()
            .filter(e -> e != null && e.getClass().equals(getClass(phase)))
            .toArray(Element[]::new);

    for (Element e : array) {
        System.out.println(e.toString());
    }

    System.out.println("===== End Searching =====\n");
}
```

UserInput

- 사용자입력을 받아 Phase 타입인지 확인
- 아니라면 다시 입력하도록

```
public class UserInput {
   static BufferedReader br = new BufferedReader(new InputStreamReader(System.in));

   // 조회하고 싶은 phase 값 입력받기
   public static String getString() throws IOException {
       // 입력받은 값이 Phase 타입에 존재할 때까지 무한 루프
       while (true) {
           System.out.print("조회하고 싶은 Phase를 입력하세요. (gas, liq, solid,
artificial, quit(종료)) : ");
           String s = br.readLine();

           // 유효한 Phase 값이 아니면 null 반환
           Phase phase = names(s);

           if (phase != null) {
               return s;
           } else {
               if (s.equals("quit")) {
                   return null;
               }
               System.out.println("gas, liq, solid, artificial 중에서 하나를
입력해주세요. ");
           }
       }
   }
}
```