Chapter 1.Birth

- 객체는 **살아있는 유기체** 이다.
- 객체는 자신의 **가시성 범위** 안에서 살아간다.
- 예제) if 블록 내부가 **extra 객체**의 **가시성 범위**이다.
    - 객체의 가시성 범위가 중요한 이유는 객체는 살아있는 유기체이기 때문이다.

```
- 객체 내부에 존재 : 5
- 객체 외부에 존재 : price
if(price < 100){
  Cash extra = new Cash(5);
  price.add(extra);
}
```

- **책의 목표**는 코드의 **유지보수성(maintainability)을 향상**
    - 다른 사람이 코드를 이해할 수 없다면 **문제의 원인은 여러분**에게 있다.
    - 객체와 객체의 역할을 이해함으로써 코드 유지보수성 향상 시킬 수 있다.
        - 코드는 짧아지고, 모듈성 및 응집도 향상
    - 코드 품질 향상은 프로젝트에서 비용 절감을 의미한다.
        - 버그 감소

        - 확장성 & 유연성 향상

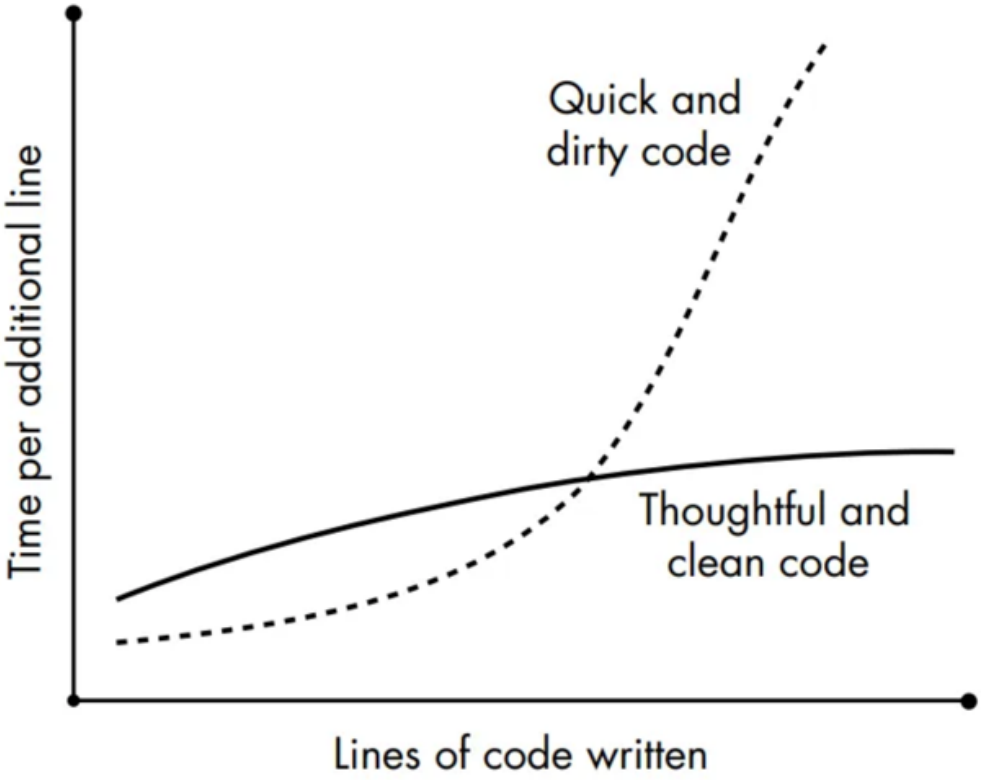


Figure 4-1: Clean code improves the scalability and maintainability of your codebase.

The Art of Clean Code by Christian Mayer

1. -er로 끝나는 이름을 사용하지 마세요.

[클래스]

- **클래스는** 객체의 **팩토리**이다.
- 클래스는 객체를 생성한다.(클래스가 객체를 **인스턴스화한다** 라고 표현)
- **클래스는 객체의 템플릿이 아니다.**
- **클래스를 객체의 능동적인 관리자로 생각**해야 한다.
    - 클래스에서 객체를 꺼내거나 반환할 수 있다.
        - **클래스를 저장소** 또는 **웨어하우스**라고 불러야 한다.
        - **클래스는 객체의 어머니**라고 할 수 있다.

[클래스 이름을 짓는 적절한 방법]

- **잘못된 방법**은 객체들이 무엇을 하고 있는(doing)지 살펴본 후 **기능(functionality)에 기반해서 이름을 짓는것**이다.
    - CashFormatter : 존중할 만한 객체가 아니다! 의인화 불가
        - Cash, USDCash, CashInUSD 같은 이름으로 변경 되어야 한다.
    - **클래스의 이름은 기능에 기반해서는 안된다.**
- 객체는 그의 **역량(capability)**으로 특징 지어야 한다.
    - ex) 내가 어떤 사람인지 키, 몸무게, 피부색과 같은 속성이 아니라, **내가 할수 있는 일(What I can do)**로 설명해야 한다.

- `-er, -or` 로 끝나는 이름을 가진 수많은 잘못 지어진 클래스들이 존재한다.
  - Manager, Controller, Helper, Encoder, Decoder, Dispatcher, Util, Utils, Target, source ...
- **객체는 캡슐화된 데이터의 대표자(representative)이다.**
  - 객체는 객체 외부 세계와 내부 세계를 이어주는 연결장치가 아니다.
  - **대표자는 스스로 결정을 내리고 행동할 수 있는 자립적인 엔티티이다.**
- 클래스의 이름은 **무엇을 하는지(what he does)**가 아니라 **무엇인지(what he is)**에 기반해야 한다.
- 예시) 오직 소수만으로 구성된 리스트를 얻는 클래스 네이밍
  - Primer, PrimeFinder, PrimeChooser, PrimeHelper 등으로 지으면 안된다.

```
class PrimeNumbers{
  List<Integer> origin;
  PrimeNumbers(List<Integer> origin){
      this.origin = origin;
  }
  ...
}
```

2. 생성자 하나를 주 생성자로 만드세요.

- 생성자(ctor)는 새로운 **객체에 대한 진입점** 이다.
  - 몇 개의 인자들을 전달받아, 어떤 일을 수행한 후, 임무를 수행할 수 있도록 **객체를 준비시킨다**.

```
//생성자 예시
class Cash{
  private int dollars;
  Cash(int dlr){
      this.dollars = dlr;
  }
}
```

[생성자]

- 보통 생성자(ctor)의 수가 메서드의 수보다 많아진다.
  - 2~3개의 메서드, **5 ~ 10개의 생성자를 포함하는 것이 적당**하다.
- **응집도가 높고 견고한 클래스**에는 적은 수의 메서드와 더 **많은 수의 생성자가 존재**한다.
- 생성자가 많을수록, 클라이언트가 **클래스를 유연하게 사용**할 수 있다.

```
new Cash(30);
new Cash("$29.95");
new Cash(29.95d);
new Cash(29.95, "USD");
```

- **퍼블릭 메서드를 많이 제공하면 유연성이 저하**된다.
- 메서드가 많아지면 클래스의 초점이 흐려지고, **단일 책임 원칙(SRP)를 위반**한다.

- 생성자의 주된 작업은 **제공된 인자를 사용해서 캡슐화하고 있는 프로퍼티를 초기화**하는 일이다.
- **초기화 로직을 단 하나의 생성자에만 위치**시키고, **부 생성자들이 주 생성자를 호출하도록 만드는것을 권장**한다.
- 하나의 주 생성자 다수의 부 생성자 원칙 (one primary, many secondary)의 핵심
    - **중복코드 방지, 설계 간결, 유지보수성 향상**

```
# GOOD : 유연함, 생성자에 사용되는 dlr이 음수불가 조건이 추가될때
         주 생성자만 수정하면 된다.
class Cash{
  private int dollars;
  Cash(float dlr){
      //부 생성자
      this((int) dlr);
  }
  Cash(String dlr){
      //부 생성자
      this(Cash.parse(dlr));
  }
  Cash(int dlr){
      //주 생성자 - 최 하단에 위치시키는것이 추후 찾기에 편하다.
      this.dollars = dlr;
  }
  private static int parse(String str){
      return Integer.valueOf(str);
  }
}
```

```
# BAD : 만약 클래스에 들어오는 dlr이 음수가 될 수 없으면 각 생성자마다 로직을 추가
해야 한다.
class Cash{
  private int dollars;
  Cash(float dlr){
      this.dollars = (int)dlr;
  }
  Cash(String dlr){
      this.dollars = Cash.parse(dlr);
  }
  Cash(int dlr){
      this.dollars = dlr;
  }
  private static int parse(String str){
      return Integer.valueOf(str);
  }
}
```

[메서드 오버로딩]

- **이름은 동일**하지만 **서로 다른 인자를 가지는 메서드 또는 생성자를 정의**할 수 있는 기법
- 객체지향 프로그래밍에서 반드시 제공되어야 하는 중요한 기능이다.

- **가독성을 극적으로 향상**시킬 수 있다.

```
content(File), contentInCharset(File, Charset)

//메서드 오버로딩을 통해 코드가 간결해진다.
content(File), content(File, Charset)
```

3. 생성자에 코드를 넣지 마세요

- **생성자는 객체 초기화 프로세스를 시작하는 유일한 장소**이기 때문에 **제공되는 인자들은 완전**해야 한다.
  - 완전하다는 의미?
    - 누락된 정보가 없어야 한다.(객체를 올바르게 초기화하기 위해 필요한 **모든 매개변수가 전달**)
    - 중복되는 정보가 없어야 한다.
- **'인자에 손대지 말라', 객체 초기화에는 코드가 없어야 하고 인자를 건드려서는 안된다.**
  - 필요하다면, 인자들을 **다른 타입의 객체로 감싸거나, 가공하지 않은 형식으로 캡슐화** 해야 한다.

```
# BAD : 객체를 초기화하는데, 인자를 건드림.
class Cash{
  private int dollars;
  Cash(String dlr){
      this.dollars = Integer.parseInt(dlr); //객체 초기화시 형변환을 함.
  }
}
```

```
# GOOD : 인자를 건드리지 않으며, 객체를 사용하는 시점까지 객체의 변환 작업을 연기한다.
1. 진정한 객체지향에서 인스턴스화란 더 작은 객체들을 조합해서 더 큰 객체를 만드는 것을 의미한다.
2. 생성자 안에서 인자에게 어떠한 작업을 하도록 요청해서는 안된다.
3. 요청이 있을 때 파싱하도록 하면, 클래스의 사용자들이 파싱 시점을 자유롭게 결정할 수 있게 된다.
4. 객체를 인스턴스화하는 동안에는 객체를 '만드는(build)'일 외에는 어떠한 일도 수행하지 않는다.
    실제 작업은 객체의 메서드가 수행하도록 한다.
5. 클래스가 하는 역할이 언제 어떻게 변경될지 모르므로, 생성자에서 어떤일을 처리하는것은 추후 유지보수에도 좋다.

class Cash{
  private Number dollars;

  Cash(String dlr){
      this(new StringAsInteger(dlr));
  }
  Cash(Number dlr){
      this.dollars = dlr; //주 생성자
```

```
  }
}

class StringAsInteger implements Number {
  private String source;
  StringAsInteger(String src){
      this.source = src;
  }
  int intValue(){
    return Integer.parseInt(this.source);
  }
}

Number num = new StringAsInteger("123");
num.intValue(); //실제작업은 객체의 메서드가 실행
```

- 표면적으로 두 Cash 클래스로부터 인스턴스 생성 과정은 동일해 보인다.

```
Cash five = new Cash("5");
```

- 첫 번째 예시는 숫자 5를 캡슐화, 두 번째는 인스턴스(StringAsInteger) 를 캡슐화한다.
- 진정한 객체지향에서 **인스턴스화**란 **더 작은 객체들을 조합해서 더 큰 객체를 만드는 것을 의미**한다.
  - 객체들을 조합해야하는 이유는 **새로운 계약을 준수하는 새로운 엔티티**가 필요하기 때문이다.
    - 기존의 계약은 Number만 허용하였으나, 새로운 계약을 준수하는 새로운 엔티티 (StringAsInteger)가 추가 되었다.

- 생성자에 코드를 넣으면 안되는 이유
  - 성능 최적화가 더 쉽다.
  - 인자를 전달된 상태 그대로 캡슐화하고 나중에 요청이 있을때 파싱하도록 하면, **클래스의 사용자들이 파싱 시점을 자유롭게 결정**할 수 있다.

```
class StringAsInteger implements Number{
  private String text;
  public StringAsInteger(String txt){
      this.text = txt;
  }
  public int intValue(){
      return Integer.parseInt(this.text);
  }
}

//코드 사용
Number five = new StringAsInteger("5");
...
if(/* 오류 */){
  throw new Exception("오류 발생");
}
five.intValue();
```

- 파싱이 여러번 수행되지 않도록 하고 싶다면 **데코레이터를 추가하여, 최초 파싱결과를 캐싱**

```
class CachedNumber implements Number{
  private Number origin;
  private Collection<Integer> cached = new ArrayList<>(1);
  public CachedNumber(Number num){
      this.origin = num;
  }
  public int intValue(){
      if(this.cached.isEmpty()){
          this.cached.add(this.origin.intValue());
      }
      return this.cached.get(0);
  }
}

//코드 사용
Number num = new CachedNumber(
  new StringAsInteger("123")
);
num.intValue();//처음에는 파싱처리
num.intValue();//캐싱된 결과 전달
```

- **객체를 인스턴스화하는 동안에는 객체를 만드는(build) 일 이외에는 어떤 일도 수행하지 않는 것이 좋다.**

Q&A

- 중복되는 정보가 없어야 한다.

Chapter 2.Education

- 객체를 다른 객체와 상호작용할 수 있도록 준비시키기 위해 필요한 몇 가지 원칙을 설명
- 이번 챕터에서 배우게 될 교훈은 **객체는 작아야 한다** 이다.
  - **작은 객체**란 우아한 동시에 유지보수 가능한 객체이다.

1. 가능하면 적게 캡슐화 하세요.

- 복잡성은 직접적으로 유지보수성에 영향을 미친다.
  - 복잡성이 높을수록 유지보수성 저하, 시간&돈 낭비, 고객 만족도 감소
- 클래스내 객체수를 **4개** 또는 그 이하의 객체를 캡슐화할 것을 권장한다.
  - 4개이상의 객체수가 캡슐화되어 있을때는, 리팩터링이 필요한 시점이다.
  - 예외는 없다. **항상 4개 이하로 유지**하도록 노력해야 한다.
- 캡슐화된 객체 전체를 가리켜 객체의 상태 또는 식별자라고 부른다.

- Java 언어의 설계적 결함으로, **객체의 식별자와 상태는 서로 분리**되어 있다.
    - 객체 지향 프로그래밍에서 **객체의 동일성은 객체를 식별하는 고유한 식별자(identifier)에 의해 결정**되고, **상태는 객체가 가지는 데이터의 값으로 결정**
    - equals를 별도로 구현해야 한다.
    - 끔찍하게 잘못된 방식이다.

```
Cash x = new Cash(29,95, "USD");
Cash y = new Cash(29,95, "USD");
x == y //동일한 식별자(오류)
x.equals(y) //객체의 상태 비교(오류)
```

- OOP의 객체는 고수준의 행동을 낳기 위해 함께 동작하는 객체들의 집합체(**aggregation**) 이다.
    - 자동차는 바퀴, 엔진, 유리 의 집합체...
- **상태 없는 객체는 존재해서는 안되고, 상태는 객체의 식별자여야 한다**
    - 4개 이상의 좌표는 직관에 위배된다.
    - **최댓값은 4**이므로, 더 많은 객체가 필요하다면 **더 작은 클래스로 분해**해야 한다.
- Java의 결함을 해결하기 위해 == 연산자가 아니라 항상 **equlas() 메서드를 오버라이드** 해야 한다!

```
//필자는 다음과 같은 방식으로 자바가 구현돼야 한다고 생각
Cash x = new Cash(29,95, "USD");
Cash y = new Cash(29,95, "USD");
assert x.equals(y);
assert x==y;
```

```
1. 4개 이하로 충분히 Cash 클래스의 객체를 식별할 수 있다.
2. == 대신, equals() 메서드를 오버라이드해서 객체의 동일성을 체크하자.
class Cash{
   private Integer digits;
   private Integer cents;
   private String currency;
}

class CashTest{

  @Test
  @DisplayName("Cash constructor without Equals and Hashcode")
  public void cashConstructorWithout(){
    CashWithoutEqualsAndHashCode x = new
CashWithoutEqualsAndHashCode(29,95,"USD");
    CashWithoutEqualsAndHashCode y = new
CashWithoutEqualsAndHashCode(29,95,"USD");
    Assertions.assertNotEquals(x,y);
```

```
    }

    @Test
    @DisplayName("Cash constructor")
    public void cashConstructor(){
       Cash x = new Cash(29,95,"USD");
       Cash y = new Cash(29,95,"USD");
       Assertions.assertEquals(x,y);
    }

    @Test
    @DisplayName("withoutHashCode")
    public void withoutHashCode(){
        HashSet<Cash> set = new HashSet<>();

        Cash x = new Cash(29,95,"USD");
        Cash y = new Cash(29,95,"USD");

        set.add(x);
        set.add(y);

        Assertions.assertEquals(1,set.size());
        Assertions.assertNotEquals(2,set.size());
    }
 }
```

2. 최소한 뭔가는 캡슐화하세요.

- 너무 많이 캡슐화하는 방식도 좋지 않지만, **아무것도 캡슐화하지 않는 방식 또한 바람직하지 않다**.
    - 프로퍼티가 없는 클래스는 객체지향 프로그래밍에서 **악명이 높은 정적 메서드(static method) 와 유사**하다.
        - 객체 지향 원칙 위배(캡슐화), 유지보수의 어려움(의존성 증가)
- 아래 예제는 **아무런 상태와 식별자도 가지지 않고 오직 행동만을 포함**한다.
    - 정적 메서드가 존재하지 않고, **객체의 생성과 실행을 엄격하게 분리하는 순수한 OOP** 에서는 프로퍼티가 없는 클래스를 만들 수 없다.

```
//프로퍼티가 없고, 객체를 생성하지 않으면서 기능을 제공한다.
class Year{
    int read(){
        return System.currentTimeMillis()/ (1000*60*60*24*30*12) - 1970;
    }
 }
```

- **실행으로부터 인스턴스 생성을 고립시켜야 한다**.
    - **오직 생성자에서만 new 연산자를 허용**해야 한다는 의미이다.

```
//프로퍼티를 제공하며, 객체를 생성된 이후에 기능을 제공한다.
class Year{
    private Number num;
```

```
    Year(final Millis mesc){
        this.num = msec.div(
            1000.mul(60).mul(60).mul(24).mul(30).mul(12)
        ).min(1970);
    }
    int read(){
        return this.num.intValue();
    }
}
```

- 객체가 '무(nothing)'와 아주 비슷한 어떤 것이 아니라면 **무언가를 캡슐화**해야 한다.
  - 자신의 좌표를 표현하기 위해 다른 엔티티를 필요로 하지 않은것
    - ex) String, ArrayList
- 객체가 어떤 것도 캡슐화하지 않는다면 객체의 좌표는 무엇일까?
  - 객체 자신이 세계 전체가 되어야 한다.
  - ```
    class Universe{
    }
    ```

3. 항상 인터페이스를 사용하세요.

- 객체는 살아있는 유기체이다.
- 객체는 다른 객체와 유기적으로 협동한다.
  - 객체는 다른 객체들과 의사소통하면서 그들의 작업을 지원, 다른 객체들 역시 이 객체에 도움을 제공한다.
- 따라서, **객체들은 서로를 필요로 하기 때문에 결합**된다.(coupled)
- 애플리케이션이 성장하면 **객체 사이의 강한 결합도(tight coupling)이 문제로 떠오른다**.
  - 강한 결합도는 **유지보수에 영향**을 미친다.
- **이 책의 목표는 유지보수성에 초점**을 맞춘다.
- 애플리케이션 전체를 유지보수 가능하도록 만들기 위해서는 최선을 다해서 객체를 분리해야(decouple) 한다.
  - **응집도를 높이고, 결합도를 낮춘다.**
- 이를 가능하게 하는것이 바로 **인터페이스**(interface)이다.

- 인터페이스에는 다른 객체와 의사소통하기 위해 따라야 하는 **계약(contract)이 명시**되어 있다.

```
interface Cash{
    Cash multiply(float factor);
}

class DefaultCash implements Cash{
    private int dollars;
    DefaultCash(int dlr){
        this.dollars = dlr;
    }
    @Override
    Cash multiply(float factor){
        return new DefaultCash(this.dollars * factor);
```
```

```
    }
}
```

- **Cash 인터페이스를 이용**하면 Employee 클래스와 DefaultCash 클래스의 **결합도를 낮출 수 있다**.
- Employee 클래스는 Cash 인터페이스의 구현 방법에 아무런 관심이 없다.

```
class Employee{
    private Cash salary;
}
```

- 클래스 안의 **모든 퍼블릭 메서드가 인터페이스를 구현하도록 만들어야 한다**.
- 올바르게 설계된 클래스는 **최소한 하나의 인터페이스를 구현**하고, 해당 **인터페이스에 정의된 메서드를 퍼블릭으로 노출**시켜야 한다.

```
# 잘못 설계된 케이스
1. cents() 는 어떤 것도 오버라이드하지 않는다.
2. Cash 클래스 사용자에게 강하게 결합되도록 조장한다.
3. 다른 클래스가 Cash.cents()를 강제로 사용하게 하므로, 새로운 메서드를 이용해서 구현 을 대체할 수 없다.
class Cash {
    public int cents() {
    // 어떤 작업을 수행한다
    }
}
```

- **클래스의 존재이유**는 다른 객체가 해당 클래스의 서비스를 필요로 하기 때문이다.
- 동일한 인터페이스를 구현하는 여러 클래스들이 존재한다
    - **각각의 클래스는 서로 다른 클래스를 쉽게 대체할 수 있어야 한다**. -> **느슨한 결합도(loose coupling)**
- 여전히 인터페이스를 통해 결합되는것은 마찬가지이다.
    - 부정할 수 없는 사실이다.
    - 하지만, 요소들 사이의 계약으로서 **인터페이스는 우리가 전체적인 환경을 구조화된 상태로 유지할 수 있도록 도와**준다.

```
interface Cash{
    int cents();
}

class DefaultCash implements Cash{

    @Override
    public int cents() {
        return Integer.MAX_VALUE;
    }
}

class UsdCash implements Cash{
```

```
    @Override
    public int cents() {
        return Integer.MIN_VALUE;
    }
}

public class Employee {
  private Cash cash;

  public Employee(Cash cash) {
    this.cash = cash;
  }

  public int toCents() {
    return cash.cents();
  }
}
```

> **각각의 클래스는 서로 다른 클래스를 쉽게 대체할 수 있어야 한다**
> **Employee 객체는, Cash 객체가 변경되거나 또는 구현체가 변경이 되어도 문제가 생기지 않는다.**

```
class CashInterfaceTest{
    @Test
    @DisplayName("Cash Interface")
    public void cashInterface(){
        Employee employee = new Employee(new DefaultCash());
        Assertions.assertEquals(Integer.MAX_VALUE,employee.toCents());

        Employee employeeUsd = new Employee(new UsdCash());
        Assertions.assertEquals(Integer.MIN_VALUE,employeeUsd.toCents());
    }
}
```

```
  interface Cash{
      Cash multiply(float factor);
  }

  class DefaultCash implements Cash{
      private int dollars;
      DefaultCash(int dlr){
          this.dollars = dlr;
      }
      @Override
      Cash multiply(float factor){
          return new DefaultCash(this.dollars * factor); //TODO 객체요소 변경
시, 새로운 객체로
      }
  }
```

```
class Employee{
    private Cash salary;
}
```

4. 메서드 이름을 신중하게 선택하세요

- 메서드 이름을 올바르게 짓는 방법
  - 빌더(builder)의 이름은 **명사**, 조정자(manipulator)의 이름은 **동사**로 짓는다.
    - 빌더 : **새로 만들고 새로운 객체를 반환하는 메서드 이며**, **반환타입은 절대 void 가 되지 않도록** 하며, 이름은 **명사**로 작성해야 한다.

```
int pow(int base, int power);
float speed();
Employee employee(int id);
String parsedCell(int x, int y);
```

    - 조정자 : **객체로 추상화한 실세계 엔티티를 수정하는 메서드 이며**, **반환 타입은 void가 되도록** 하며, 이름은 **동사**로 작성해야 한다.

```
void save(String content);
void put(String key. Float value);
void remove(Employee emp);
void quicklyPrint(int id);
```

- 빌더는 어떤 것을 만들고, 조정자는 뭔가를 조작한다.
- 빌더와 조정자 사이에는 **어떤 메서드도 존재해서는 안된다**.
  - 뭔가를 조작한 후 반환하거나, 뭔가를 만드는 동시에 조작하는 메서드가 있어서는 안된다

```
//잘못된 예시
// 저장된 전체 바이트를 반환
int save(String content);
// map이 변경된 경우 TRUE를 반환
boolean put(String key. Float value);
// speed를 저장한 후 이전 값을 반환
float speed(float val);
```

  - save() 메서드는 조정자지만 빌더처럼 int 값을 반환한다.
    - void로 변경하거나 bytesSaved() 이름으로 바꿔야 한다.
  - put() 메서드는 조정자지만 빌더처럼 boolean을 반환한다.
    - 성공/실패 여부를 반환하는 빌더인 success() 메서드를 생성해야한다.
  - speed() 메서드는 빌더인 동시에 조정자이다.
    - 스피드를 저장 메서드, 스피드를 구하는 메서드를 나눠야 한다.
- **getter**는 근본적으로 **어떤 값을 반환하는 빌더**이다. -> get은 동사이기 때문에, 잘못된 네이밍이다.

4.1 빌더는 명사다

- 어떤 것을 반환하는 메서드의 이름을 **동사로 짓는 것은 잘못된 것**이다.
  - 커피 한 잔 끓여 주세요 라고 말하지 않고, 커피 한 잔 주세요 라고 말한다.
  - 커피를 끓이는 정확한 방법은 관심사가 아니다
  - 단지 **객체에게 커피를 요구**할 뿐이다.

```
class Bakery{
  Food cookBriwnie();
  Drink brewCupOfCoffee(String flavor);
}

class Bakery{
  Food food;
  Drink drink(String flavor);

}
class Food{
}
class Drink{
}

Bakery bakery = new Bakery(new Brownie(), new LatteeCoffe("헤이즐넛"));
```

- 객체는 자신의 의무를 수행하는 방법을 알고 있고 존중 받기를 원하는 **살아있는 유기체**이다.
  - 단순히 지시에 따르는 것이 아니라 계약에 기반해 일하고 싶어한다.
  - 메서드의 이름을 동사로 지을 때에는 객체에게 무엇을 할 지(what to do) 알려 주어야 한다.
  - **객채에게 무엇을 만들라고(build) 요청**하는 것은 **협력자에 대한 존중이 결여되어 있고 예의에도 어긋나는 행동**이다.
- 객체에게 **무엇을 만들라고 요청하는 것이 아니라**, **무엇을 할지** 알려준다.

```
Inputstream load(URL uri); -> InputStream stream(URL uri);
String read(File file); -> String content(File file);
int add(int x, int y); -> int sum(int x, int y);
```

```
int add(int x, int y) // bad : 객체에게 더하라고 요청하지 않는다.
int sum(int x, int y) // good : 합을 계산하고, 객체를 반환해 달라고 요청한다.
```

4.2 조정자는 동사다

- 객체가 무언가를 조정해야 한다면 **이름은 동사**이고 **반환값은 없다**.
- 객체는 **실세계의 엔티티(entity)를 대표** 한다.
- 무언가가 만들어질(build) 것이라고 기대하지 않는다.
  - paint 라는 이름은 동사이고, 기본적으로 **객체에게 어떤 일을 하도록 지시**한다.
    - **paint() 메서드는 값을 반환하지 않는다.**

```
class Pixel {
    private int x;
    private int y;
    Pixel(int x, int y){
        this.x = x;
        this.y = y;
    }
    void paint(Color color);
}
Pixel center = new Pixel(50,50);
center.paint(new Color("red")); //'동사'형태로, 그려라 라고 지시를 하며, 반환값은
void 이다.
```

- 빌더들은 이름은 명사이고 이름은 명사이다.
    - 빌더 패턴은 유지보수성이 낮고 응집도가 떨어진다. -> 더 커다란 객체를 만들도록 조장한다.
    - 빌더 패턴은 생성자에 많은 이자를 전달하고 싶지 않을 때 유용하게 사용할 수 있다.
    - 애초에 인자의 수가 많다는 것 자체가 문제이다.
    - **빌더 패턴 대신 복잡한 객체를 더 작은 객체들로 나눠야 한다.**

```
class Book {
    Book withAuthor(String author);
    Book withTitle(String title);
    Book withPage(Page page);
}
```

4.3 빌더와 조정자 혼합하기

- write 메서드는 원칙상 return 타입이 void로 바꿔야 한다.
    - 하지만 우리는 실제로 저장된 바이트 수를 알 필요가 있다.
- write 메서드는 2가지 역할을 하고 있다.
    - **데이터를 쓰고, 바이트 수를 카운트**

```
//메서드는 데이터를 쓰는 동시에, 쓰여진 바이트 수를 카운트 한다. (2개 이상의 역할을 한다)
class Document{
    int write(InputStream contents);
}
```

- output() 은 빌더이다.
- 데이터를 쓸 준비를 하는 OutputPipe 객체를 통해 write() 를 쓰고, bytes()를 통해 바이트 수를 카운트 할 수 있다.

```
//빌더와 조정자를 함께쓴다.OutputPipe 에서 역할이 완벽하게 분리되었다.(write : 쓰기, bytes
: 쓰여진 바이트 수)
class Document{
    OutputPipe output(); //빌더
```

```
}
class OutputPipe{
    void write(InputStream contents); //조정자
    int bytes(); //빌더
}
```

- **OOP의 전체 목적은 개념을 고립시켜 복잡성을 낮추는 것**이다.
- 하나 이상의 값을 반환하는 메서드는 코드를 지저분하게 하고, 유지보수성을 저하 시킨다.(Go 언어는 2개 이상의 데이터를 리턴할 수 있다.)

4.4 Bollean 값을 결과로 반환하는 경우

- **Boolean 값을 반환하는 메서드**는 규칙에 있어 **예외적인 경우**이다.
- 이 **메서드들은 빌더에 속하지만**, 가독성 측면에서 이름은 명사가 아닌 **형용사**로 짓는다.

```
boolean empty();
boolean readable();
boolean negative();
```

- is는 중복이기 때문에, 메서드 이름에는 포함시키지 않는다.
- 그렇다면 Boolean을 반환하는 메서드를 특별하게 취급해야 하는 이유는 무엇일까?
    - 메서드의 이름 을 형용사로 지어야 더 자연스럽게 읽히는 문장이 되기 때문이다.

```
if (name.empty) { // 만약 이름이 공백이라면
}
```

> **요약**
>
> 1. 메서드는 빌더나 조정자, 둘 중 하나여야 한다.
> 2. 결코 빌더인 동시에 조정자여서는 안된다.
> 3. 빌더라면 이름을 명사로, 조정자 라면 이름을 동사로 지어야 한다.
> 4. Boolean 값을 반환하는 빌더는 예외에 속한다. 이름을 형용사로 한다.

5. 퍼블릭 상수(Public Constant)를 사용하지 마세요

- 상수라고 불리는 public static final 프로퍼티는 **객체 사이에 데이터를 공유하기 위해 사용하는 매커니즘**
    - **데이터를 공유(다른 객체를 공유)하기 때문에 퍼블릭 상수를 강력하게 반대**한다.
- **객체들은 어떤 것도 공유해서는 안된다. 독립적**이여야 하고 **닫혀** 있어야 한다..
- 즉, 상수를 이용한 공유 메커니즘은 **캡슐화와 객체지향적인 사고 전체를 부정하는 일**이다.

- Writer 객체에 기록, 각 줄을 개행(new line) 문자로 종료하는 메서드
- EOL 벨류 중복 발생, Constatns 로 중복 문제를 해결
    - OOP에는 객체가 존재하기 때문에 **퍼블릭 상수를 이용해서 중복 문제를 해결하는 것은 매우 잘못된 접근방법**이다.

```
class Records {
  private static final String EOL = "\r\n";

  void write(Writer out) {
    for (Record rec : this.all) {
      out.write(rec.toString());
      out.write(Records.EOL);
    }
  }
}

class Rows{
    private static final String EOL = "\r\n";

    void print(PrintStream pnt) {
      for (Row row : this.fetch()) {
        pnt.printf("{ %s }%s'\ row. Rows.EOL);
      }
    }
}
```

[결합도 증가, 응집도 저하]

- 코드 중복을 해결하기 위해 **두 개의 더 큰 문제가 추가** 되었다.
- **Constants, Records, Rows 가 강하게 결합**되어 있다.
- **Constants.EOL이 변경되면 참조하는 나머지 클래스의 일이 달라진다**.

1. **결합도 증가**

- 두 클래스는 모두 같은 객체에 의존, 이 **의존성은 하드 코딩**되어 있다.
  - Records.write(), Rows.print(), Constants.EOL
- Constants.EOL 내용을 수정하면 참조하는 나머지 클래스의 일이 달라진다.
- **사용자들(Records, Rows 객체를 사용하는 객체 또는 사람)이 Constants.EOL에 결합될 수 밖에 없도록 만들었고,**
  유지보수성은 크게 저하되었다.
- 많은 객체들이 **다른 객체를 사용하는 상황에서 서로를 어떻게 사용하는지 알 수 없다면,**
  이 객체들은 매우 강하게 결합되어 있는 것이다.
- **Constants.EOL 객체는 사용 방법과 관련된 어떤 정보도 제공하지 않는다.**
  - 객체의 변경으로 인해 사용자가 어떤 영향을 받는지 파악 불가

2. **응집도 저하**

- 퍼블릭 상수를 사용하면 객체의 **응집도는 낮아진다**.
  - 낮은 응집도는 **객체가 자신의 문제를 해결하는데 덜 집중**한다는 것을 의미한다.
- Constants.EOL은 **자신에 관해 아무 것도 알 지 못하며, 자신의 존재 이유를 이해하지 못하는 하나의 텍스트 덩어리**에 불과하다.
  - **상수는 삶의 의미가 명확하지 않다.**
- Records와 Rows의 목적은 한 줄의 마지막을 처리하는 것이 아니라 레코드나 로우 자체를 처리 하는 것이다.
  - 한 줄을 **종료하는 작업을 다른 객체에게 위임**한다면 각 **객체의 응집도 를 향상**할 수 있다.

```
public class Constants{
    public static final String EOL = "\r\n";
}

class Records {
    void write(Writer out){
        for(Records rec : this.all){
            out.write(rec.toString());
            out.write(Constants.EOL);
        }
    }
}

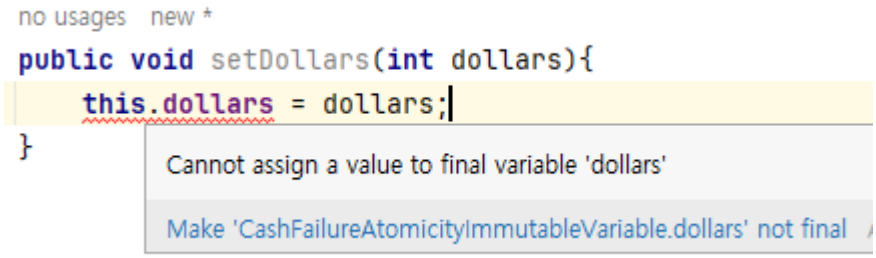
class Rows{
    void print(PrintStream pnt){
        for(Row row : this.fecch()){
            pnt.printf("{%s}",row, Constants.EOL);
        }
    }
}
```

[refactor]

- **객체 사이에 데이터를 중복해서는 안된다.**
  - 기능을 공유할 수 있도록 EOLString **새로운 클래스 생성**한다.
- **데이터가 아니라 기능을 공유**해야 한다.
- EOLString과의 결합은 **계약(contract)을 통해 추가된 것**이다.
  - **계약에 의한 결합은 언제라도 분리가 가능하기 때문에 유지보수성 향상**된다.
  - EOLString은 계약에 따라 행동하며 내부에 **계약의 의미를 캡슐화**합니다
- 예를들어, 윈도우일경우 예외를 던지도록 코드를 수정하는 요청이 들어올때도 **다른 객체와의 계약은 동일하게 유지**하면서 동작은 변경할 수 있다.
  - public static 의 경우 이러한 변경이 불가능하다.
- 퍼블릭 상수마다 계약의 의미를 캡슐화하는 새로운 클래스를 만들어야 한다는 것을 의미하는 것일까? -> **맞다**
- 수백 개의 단순한 상수 문자열 리터럴 대신 수백 개의 마이크로 클래스를 만들어야 한다는 것을 의미하는 것일까? -> **맞다**
- **애플리케이션을 구성하는 클래스의 수가 많을수록 설계가 더 좋아지고 유지보수하기도 쉬워진다.**

```
class EOLString{
    private final String origin;
    EOLString(String src){
        this.origin = src;
    }
    @Override
    String toString(){
        if (/* Windows의 경우 */) {
          throw new IllegalStateException("현재 Windows에서 실행 중이기 때문에
EOL을 사용할 수 없습니다. 죄송합니다." );
        }
```

```
            return String.format("%s\r\n", origin);
        }
    }

    class Records {
        void write(Writer out){
            for(Records rec : this.all){
                out.write(new EOLString(rec.toString()));
            }
        }
    }

    class Rows{
        void print(PrintStream pnt){
            for(Row row : this.fecch()){
                pnt.printf(
                    new EOLString(String.format("{%s}"),row)
                );
            }
        }
    }
```

[Http 요청 메서드 예제]

- POST 리터럴의 의미를 기억할 필요가 없다.
- PostRequest는 **POST 리터럴의 의미론(semantic)인 설정 로직을 내부에 캡슐화**한다.

```
    //BAD : 하드코딩
    String body = new HttpRequest().method("POST").fetch();

    //BAD : 퍼블릭 상수를 사용하는 예제
    String body = new HttpRequest()
        .method(HttpMethods.POST) //퍼블릭 상수
        .fetch();

    //GOOD : post 리터럴의 의미를 기억할 필요 없이, Post 방식으로 요청을 전송하기만 하면 된
다.
    String body = new PostRequest(new HttpRequest()).fetch();
```

[요약]

- OOP에서 **퍼블릭 상수를 절대로 사용해서는 안된다.**
- 아무리 사소해 보이는 상수라도 **항상 작은 클래스를 이용해서 대체**해야 한다.
- 퍼블릭 상수를 이용해서 코드 중복 문제를 해결하지 말고 대신 **클래스를 사용**해야 한다.
- 열거형(enum)과 퍼블릭 상수 사이에 는 아무런 차이도 없기 때문에 **열거형 역시 사용해서는 안된다.**

6. 불변 객체로 만드세요.

- **모든 클래스를 상태변경이 불가능한 불변 클래스(immutable class)로 구현**하면 **유지보수성을 크게 향상**시킬 수 있다.
    - **불변성** 역시 크기가 작고, 응집력이 높으며, 느슨하게 결합되고, 유지보수하기 쉬운 클래스를 만들 수 있도록 한다.
- **불변 객체를 기반으로 사고**하면 **더 깔끔하고, 더 작고, 더 쉽게 이해할 수 있는 코드**를 만들 수 있다.
- **인스턴스 생성한 후**, **상태를 변경할 수 없는 객체**를 **불변 객체**라고 부른다.

- **to-be** 부분에 private dollars 프로퍼티에 **final** 이 추가 되었다.

```
//as-is (bad)
class Cash{
    private int dollars;

    public void setDollars(int val){
        this.dollars = val;
    }
}
//to-be (good)
class Cash{
    private final int dollars;
    Cash(int val){
        this.dollars = val;
    }
}
```

- final 키워드는 생성자 외부에서 값 수정시 **컴파일 에러가 발생**한다.



    - **불변 객체**는 **필요한 모든 것을 내부에 캡슐화** 하고 **변경 할 수 없도록 통제**한다.
    - **불변 객체를 수정해야 한다면 새로운 객체를 생성해야 한다.**

```
//bad 프로퍼티가 변경 가능한 코드 (가변객체)
class Cash {
    private int dollars;
    public void mul(int factor) {
        this.dollars *= factor;
    }
}

//가변객체의 예제 - 사용
Cash five = new Cash(5);
five.mul(10);
```

```
System.out.println(five); // "$50"이 출력

//good 프로퍼티가 변경 불가능한 코드 (불변객체)
class Cash {
  private final int dollars; public Cash mul(int factor) {
    return new Cash(this.dollars * factor);
  }
}

//불변객체의 예제 – 사용
Cash five = new Cash(5);
Cash fifty = five.mul(10);
System.out.println(fifty); // "$50"이 출력
```

- **가변 객체**는 **전반적인 객체 패러다임의 오용**이다.
  - **가변 객체의 사용을 엄격하게 금지**해야 한다.
  - **모든 클래스**는 상태를 절대로 변경하지 않는 **불변 객체를 인스턴스화 해야 한다.**
- 가변객체의 예제는 five 객체가 **'5'달러가 아닌 '50'달러 처럼 행동**하고 있다.

- 기술적으로 불변 객체를 사용해서 지연 로딩을 구현하는 것은 불가능 하다.(java, ruby, c++)
  - 자바에서는 유사한 기능을 현재 제공하고 있진 않다.
- 필자는 **언어 차원에서 지연 로딩을 제공해야 한다고 생각**한다.

```
class Page {
  private final String uri;
  private String html; Page(String address) {
    this.uri = address;
    this.html = null;
  }
  @OnlyOnce //필자가 생성하는 필요로 하는 지연로딩 어노테이션
  public String content() {
    if (this.html == null) {
        this.html = /* 네트워크로부터 로드한다 */
    }
    return this.html;
  }
}
```

1) 식별자 가변성(Identity Mutability)

- **불변 객체**에는 **식별자 가변성(identity mutability) 문제가 없다.**
- 불변 객체를 사용하면 **객체를 map에 추가한 후에는 상태 변경이 불가능**하기 때문에 **식별자 가변성 문제가 발생하지 않는다.**

[가변객체]

- 장황한 소스 어딘가에서, five.mul(2) 를 했을경우 **식별자 문제가 생긴다.**

```
public class CashIdentityMutabilityChangeVariable {
    private int dollars;

    public CashIdentityMutabilityChangeVariable(int dollars) {
        this.dollars = dollars;
    }

    public void mul(int factor) {
        this.dollars *= factor;
    }
}

@Test
@DisplayName("CashIdentityMutabilityChangeVariable")
public void CashIdentityMutabilityChangeVariable(){
  Map<CashIdentityMutabilityChangeVariable, String> map = new HashMap<>();

  CashIdentityMutabilityChangeVariable five = new
CashIdentityMutabilityChangeVariable(5);
  CashIdentityMutabilityChangeVariable ten = new
CashIdentityMutabilityChangeVariable(10);

  map.put(five,"five");
  map.put(ten,"ten");
  //~~~ 장황한 소스 어딘가에서
  five.mul(2);

  Assertions.assertNotEquals(map.get(five), "five");//five가 아니라고 나옴
  Assertions.assertEquals(map.get(five), "ten");//five를 get 했는데 ten 이라고
나옴
}
```

[불변객체]

- 장황한 소스 어딘가에서, five.mul(2) 를 했음에도 **식별자 문제 생기지 않는다.**

```
public class CashIdentityMutability {
    private final int dollars;

    public CashIdentityMutability(int dollars) {
        this.dollars = dollars;
    }

    public CashIdentityMutability mul(int factor) {
        return new CashIdentityMutability(this.dollars * factor);
    }
}

@Test
@DisplayName("CashIdentityMutability")
```

```
public void CashIdentityMutability(){
  Map<CashIdentityMutability, String> map = new HashMap<>();

  CashIdentityMutability five = new CashIdentityMutability(5);
  CashIdentityMutability ten = new CashIdentityMutability(10);

  map.put(five,"five");
  map.put(ten,"ten");
  //~~~ 장황한 소스 어딘가에서
  five.mul(2);
  //map.put(five.mul(2),"five");

  Assertions.assertEquals(map.get(five), "five");
}
```

[Map, List 는??]

- Map과 List는 불변객체가 아니지 않은가?

```
    @Test
    @DisplayName("CashIdentityMutabilityMap")
    public void CashIdentityMutabilityMap(){
        CashIdentityMutability five = new CashIdentityMutability(5);
        CashIdentityMutability ten = new CashIdentityMutability(10);
        Map<CashIdentityMutability, String> immutableMap = Map.of(five,
"five", ten, "ten");
        Assertions.assertThrows(UnsupportedOperationException.class,()-> {
            immutableMap.put(new CashIdentityMutability(15),"fifteen");
        });
    }

    @Test
    @DisplayName("CashIdentityMutabilityArray")
    public void CashIdentityMutabilityArray(){
        CashIdentityMutability five = new CashIdentityMutability(5);
        CashIdentityMutability ten = new CashIdentityMutability(10);

        CashIdentityMutabilitys immutableArray = new
CashIdentityMutabilitys(List.of(five,ten));
        Assertions.assertThrows(UnsupportedOperationException.class,()-> {
            immutableArray.getCashIdentityMutability().add(new
CashIdentityMutability(15));
        });
    }
```

2) 실패 원자성(Failure Atomicity)

- **실패 원자성**이란 **완전하고 견고한 상태의 객체를 가지거나** 아니면 **실패**하거나 둘 중 하나만 가능 한 특성이다.
  - 중간은 없다.

[가변객체]

- 객체의 **유지보수성** 역시 심각한 영향을 받게 된다.
- 객체의 **복잡성이 훨씬 더 높아지고**, 그 때문에 **실수할 가능성**이 더 커진다.

```
//가변 객체
class CashFailureAtomicityChangeVariable {
    private int dollars;
    private int cents;

    public CashFailureAtomicityChangeVariable(int dollars, int cents) {
        this.dollars = dollars;
        this.cents = cents;
    }
    public void mul(int factor) {
        this.dollars *= factor;
        if (1 == 1) {
            throw new RuntimeException("oops...");
        }
        this.cents *= factor;
    }
}

@Test
@DisplayName("CashFailureAtomicityChangeVariable")
public void CashFailureAtomicityChangeVariable(){

  CashFailureAtomicityChangeVariable cash = new
CashFailureAtomicityChangeVariable(10,50);
  Assertions.assertThrows(RuntimeException.class,()-> {
    cash.mul(2);
  });
  Assertions.assertEquals(20, cash.getDollars()); //예상되는 dollars 값
  Assertions.assertNotEquals(100, cash.getCents());//예상되는 cents 값
}
```

[불변객체(실패 원자성)]

- **불변 객체**는 **원자적**이기 때문에 원자성을 걱정할 필요가 없다.

```
//불변객체(실패 원자성)
class CashFailureAtomicityImmutableVariable {
    private final int dollars;
    private final int cents;

    public CashFailureAtomicityImmutableVariable(int dollars, int cents) {
        this.dollars = dollars;
        this.cents = cents;
    }
```

```
    public CashFailureAtomicityImmutableVariable mul(int factor) {
        if (1 == 1) {
            throw new RuntimeException("oops...");
        }
        return new CashFailureAtomicityImmutableVariable(
                this.dollars * factor,
                this.cents * factor
        );
    }
}

@Test
@DisplayName("CashFailureAtomicityImmutableVariable")
public void CashFailureAtomicityImmutableVariable(){

  CashFailureAtomicityImmutableVariable cash = new
CashFailureAtomicityImmutableVariable(10,50);
  Assertions.assertThrows(RuntimeException.class,()-> {
    cash.mul(2);
  });
  Assertions.assertEquals(10, cash.getDollars()); // 예상되는 dollars 값
  Assertions.assertEquals(50, cash.getCents());   // 예상되는 cents 값
}
```

[자바에서 해시충돌 처리는?]

```
class Student {
    private int id;

    public Student(int id) {
        this.id = id;
    }

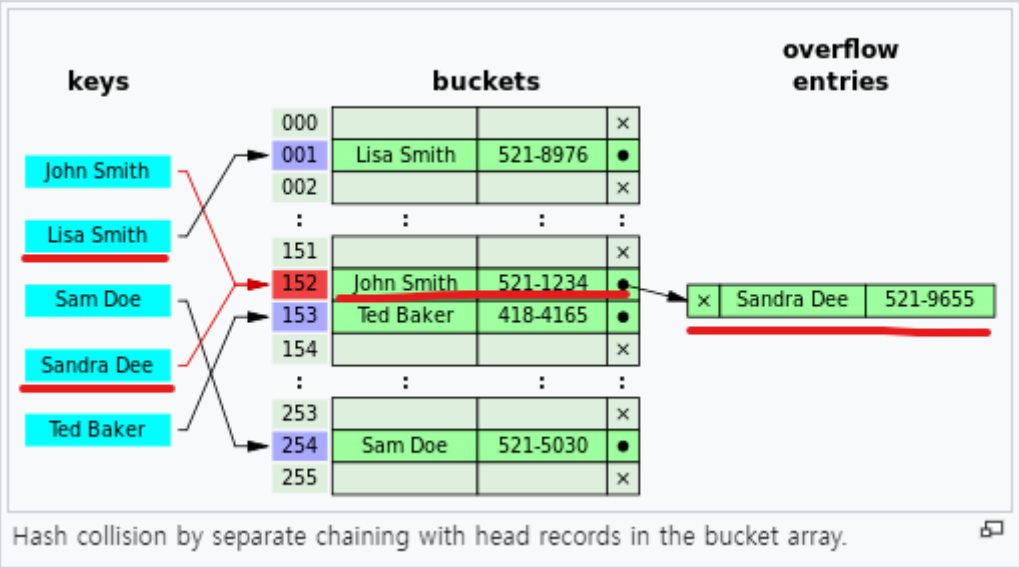
    @Override
    public int hashCode() {
        // 모든 학생의 hashCode를 1로 설정하여 충돌을 일으킴
        return 1;
    }

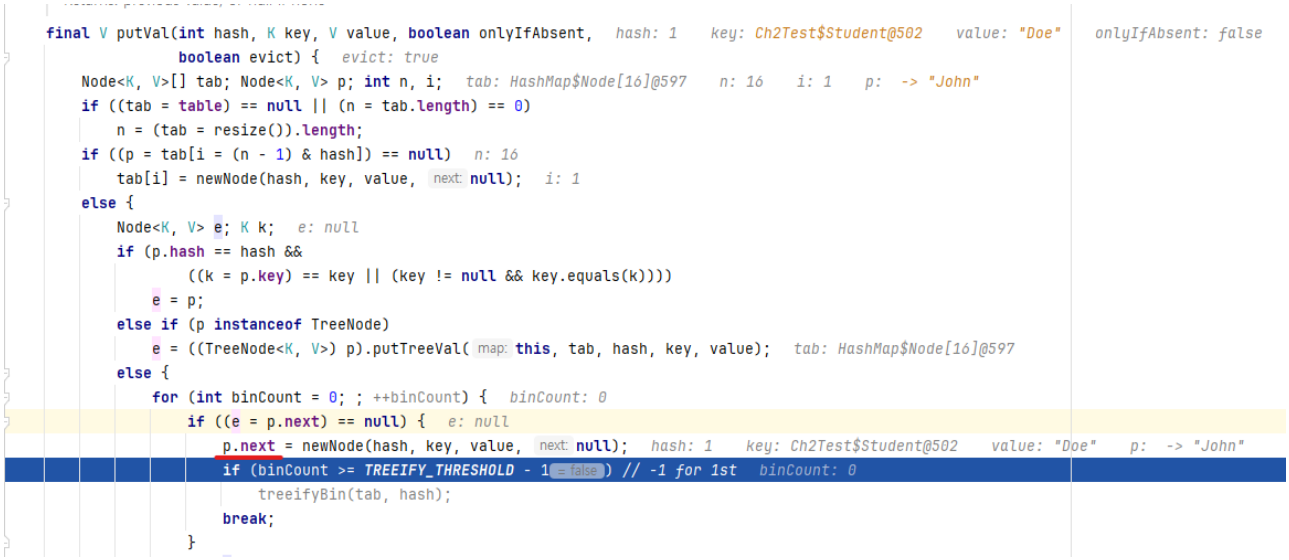
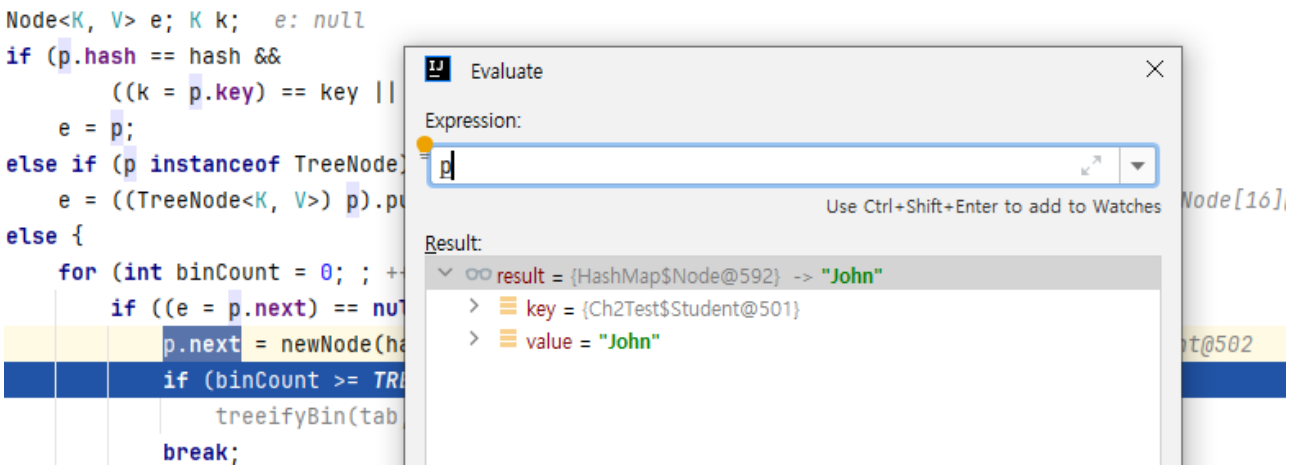
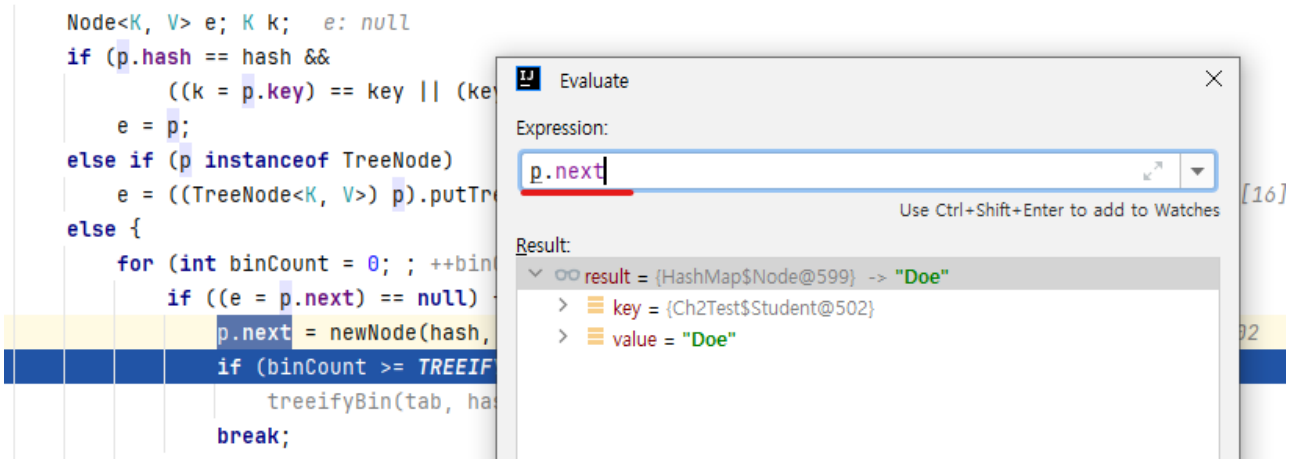
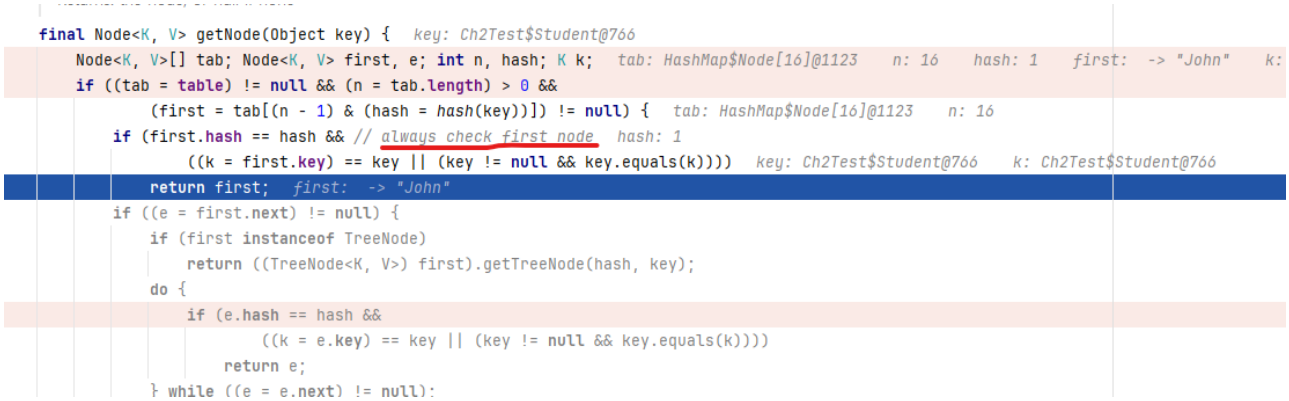
    @Override
    public boolean equals(Object obj) {
        if (this == obj) return true;
        if (obj == null || getClass() != obj.getClass()) return false;
        Student student = (Student) obj;
        return id == student.id;
    }
}

// 사용
HashMap<Student, String> studentMap = new HashMap<>();
```

```
Student student1 = new Student(1);
Student student2 = new Student(2);

// 두 학생이 같은 해시 코드를 가지도록 함
System.out.println("HashCode of student1: " + student1.hashCode());
System.out.println("HashCode of student2: " + student2.hashCode());

studentMap.put(student1, "John");
studentMap.put(student2, "Doe");

// 해시 충돌 발생으로 두 학생이 같은 해시 버킷에 매핑되었을 것
System.out.println("Student 1's name: " + studentMap.get(student1));
System.out.println("Student 2's name: " + studentMap.get(student2));
```

- 해쉬코드



Hash collision by separate chaining with head records in the bucket array.

출처 : https://en.wikipedia.org/wiki/Hash_table

[hash put]

- John 넣을때

```
static class Node<K, V> implements Map.Entry<K, V> {
    final int hash;
    final K key;
    V value;
    Node<K, V> next;

    Node(int hash, K key, V value, Node<K, V> next) {  hash: 1   key: Ch2Test$Student@602   value: "John"   next: null
        this.hash = hash;  hash: 1
        this.key = key;  key: Ch2Test$Student@602
        this.value = value;  value: "John"
        this.next = next;  next: null
    }
```

- Doe 넣을때







[hash get]

- John을 찾을때



- Doe를 찾을때

3) 시간적 결합(Temporal Coupling)

- 불변 객체를 사용하면 , **시간적 결합**(temporal coupling)을 **제거** 할 수 있다.

[시간적 결합 발생 예제]

- 프로퍼티를 null로 선언한 이후, setter를 이용해 프로퍼티 값을 설정
  - 다양한 표준(JavaBeans, JPA, JAXB)에서 Java 객체를 조작하기 위해 권장하는 방식이다.
  - 하지만, 진정한 **객체 사고**(object thinking)의 관점에서는 **완전히 잘못된 방식 이다.**
- 불변객체 아닌 객체의 코드를 수정하기로 했다면, 코드상의 **시간적인 결합**을 이해해야 한다.
- 가변 객체들을 처리하는 연산들의 순서를 일일이 기억해야 한다면 **코드 유지보수성이 떨어진다.**

```
Cash price = new Cash();
//~ x를 구하기위한 코드
price.setDollars(x);
//~ y를 구하기위한 코드
price.setCents(y);
```

[불변객체를 사용한 시간적 결합 제거]

- 객체의 **인스턴스화와 초기화가 한번에** 이루어지며, **분리시킬 수 없다**.
- 따라서, **시간적 결합이 제거**가 된다.

```
Cash price = new Cash(29, 95);
```

- 객체를 사용해서 어떤 일을 수행하려면, **객체를 완전한 상태로 초기화**해야 한다.

```
// 시간적 결합 발생
Order order = new Order();

// 시간적 결합 발생: Order 객체 생성 후에 상품을 추가해야 함
Product laptop = new Product("Laptop", 1500);
Product mouse = new Product("Mouse", 30);

order.addProduct(laptop);
order.addProduct(mouse);
-----------------------------
//시간적 결합 제거
Order order = new Order(
        List.of(new Product("Laptop", 1500)
                , new Product("Mouse", 30))
        );
```

4) 부수효과 제거(Side effect-free)

[가변객체 부수효과]

- 객체가 가변적일 때는 누구든 객체를 수정할 수 있다.
- 개발자가 실수로 cash.mul(2)를 넣을 수 있다.
- **실수로 넣은 코드로 인해, 데이터의 정합성이 깨질 수 있다.**

```
class Cash {
    private int dollars;
    private int cents;
    Cash(int dollars, int cents) {
        this.dollars = dollars;
        this.cents = cents;
    }
    public void mul(int factor){
        this.dollars *= factor;
        this.cents *= factor;
    }
}

//사용시
Cash cash = new Cash(10,0);
//~몇만줄의 로직
cash.mul(2); //실수로 넣은 코드
//~몇만줄의 로직
UsdCash usdCash =  toUsd(cash);
//실수로 넣은 코드로 인해 Cash는 20을 계산한다.
```

[불변객체 부수효과 제거]

- 반면에 Cash **클래스를 불변으로 만들면 누구도 객체를 수정할 수 없다.**
- **Cash는 불변**이기 때문에, 실수로 넣은 코드와 상관없이 데이터 정합성은 깨지지 않는다.

```
class Cash {
    private final int dollars;
    private final int cents;
    Cash(int dollars, int cents) {
        this.dollars = dollars;
        this.cents = cents;
    }
    //불변
    public Cash mul(int factor){
        return new Cash(
            this.dollars * factor,
            this.cents * factor
        );
    }
}
//사용시
Cash cash = new Cash(10,0);
//~몇만줄의 로직
cash.mul(2); //실수로 넣은 코드
//~몇만줄의 로직
UsdCash usdCash = toUsd(cash);
//실수로 넣은 코드와 상관없이 Cash는 10으로 불변이다.
```

5) NULL 참조 없애기

- **설정되지 않은**(unset) **프로퍼티**에 대한 문제점
  - null 체크 로직 증가
  - null 체크 누락으로 인한 NP 가능성 증가
  - **코드 유지보수성 감소**

```
class User{
    private final int id;
    private String name = null; //unset 프로퍼티
    public User(int num){
        this.id = num;
    }
    public void setName(String txt){
        this.name = txt;
    }
}
```

- **이러한 설정되지 않은 프로퍼티가 만들어지는 이유**
  - 다른 클래스가 필요하지만 새로운 클래스를 생성하는 작업이 귀찮게 여기기 때문이다.
  - 클래스를 어떻게 만들어야 할지 몰라서이다.
  - OOP에서 클래스가 무엇을 의미하는 지를 제대로 이해하지 못해서 일 수 있다.
- 결국에는 결과는 **매우 큰 클래스가 만들어 진다.**
  - 사용자인 동시에 고객이고, 사원인 동시에 데이터베이스 레코드인 **여러가지 역할을 담당하는 클래스**가 된다.
- **NULL의 존재 자체가 이런 형편없는 프랙티스를 따르도록 조장**한다.
- **모든 객체를 불변으로 만들면** 객체 안에 **NULL을 포함시키는 것이 애초에 불가능**하다.
- **불변 객체를 사용**하면 **작고, 견고하며, 응집도 높은 객체를 생성할 수 밖에 없도록 강제**되기 때문에 **유지보수성이 높은 쉬운 객체를 만든다**.

6) 스레드 안전성

- 스레드 안정성(Thread Safety)?
  - 객체가 여러 스레드에서 동시에(concurrently) 사용될 수 있으며 **그 결과를 항상 예측가능하도록 유지할 수 있는 객체의 품질**을 의미한다.
- 병행성 이슈는 발견하고, 디버깅하고, 해결하기 가장 어려운 문제 중 하나이다.
- **불변 객체는 실행 시점에 상태를 수정할 수 없게 금지함으로써 이 문제를 완벽하게 해결한다.**
- 명시적인 동기화를 이용하면 가변 클래스 역시 스레드에 안전하게 만들 수는 있다.
  - 성능상의 비용을 초래한다.
  - 각각의 스레드가 객체를 사용하기 위해서는 객체가 다른 스레드로 부터 해방될 때까지 기다려야 한다.

```
class Cash {
    private int dollars;
    private int cents;
    public void mul(int factor) {
      synchronized (this) {
          this.dollars *= factor;
          this.cents *= factor;
      }
    }
}
```

```
    @Test
    @DisplayName("가변객체는 스레드 안정성이 보장되지 않는다.")
    public void threadSafety() throws InterruptedException {
        final int DOLLARS = 10;
        final int CENTS = 50;
        final CashThreadSafe cash = new CashThreadSafe(DOLLARS, CENTS);
        Thread thread1 = new Thread(() -> {
            cash.mul(2);
        });
        Thread thread2 = new Thread(() -> {
            cash.mul(3);
        });
        thread1.start();
        thread2.start();
```

```
        thread1.join();
        thread2.join();
        Assertions.assertEquals(DOLLARS*2*3,cash.getDollars());
        Assertions.assertEquals(CENTS *2*3,cash.getCents());
    }

    @Test
    @DisplayName("불변객체는 스레드 안정성이 보장된다.")
    public void threadSafety2() throws InterruptedException {
        final int DOLLARS = 10;
        final int CENTS = 50;
        final Cash cash = new Cash(DOLLARS, CENTS,"USD");
        Thread thread1 = new Thread(() -> {
            cash.mul(2);
        });
        Thread thread2 = new Thread(() -> {
            cash.mul(3);
        });
        thread1.start();
        thread2.start();
        thread1.join();
        thread2.join();
        Assertions.assertEquals(DOLLARS,cash.getDigits());
        Assertions.assertEquals(CENTS,cash.getCents());
    }
```

[싱글턴 구현]

```
public class Singleton {
    private static Singleton singleton;

    public static Singleton getInstance() {
        if(singleton == null) {
            singleton = new Singleton();
        }
        return singleton;
    }
}

public class Singleton {
  private static Singleton singleton;

  //synchronized - 특정 쓰레드가 해당 메소드를 실행시키는 중이면 접근불가
  public static synchronized Singleton getInstance() {
    if(singleton == null) {
      singleton = new Singleton();
    }
    return singleton;
  }
}
```

```
public class Singleton {
  private static Singleton singleton = new SingleTon();

  public static Singleton getInstance() {
    return singleton;
  }
}

public class Singleton {
  private volatile static Singleton singleton;

  public static Singleton getInstance() {
    if(singleton == null) {
      synchronized (Singleton.class) {
        if(singleton == null) {
          singleton = new Singleton();
        }
      }
    }
    return singleton;
  }
}
```

7) 더 작고 더 단순한 객체

- 불변성은 **단순성**(simplicity)의 장점을 갖는다.
- **객체가 단순해질 수록 응집도는 더 높아지고, 유지보수하기 쉬워진다.**
- **최고의 소프트웨어는 단순하다.**
  - 이해하고, 수정하고, 문서화하고, 지원하고, 리팩토링하기 쉽다.
- Java에서 허용 가능한 클래스의 최대 크기는 주석+공백을 포함해서 **250줄** 정도이다.(프로덕션 코드, 테스트 코드 각각)
  - **250줄을 넘는 클래스는 리팩토링이 필요**하다.
- 불변 객체를 크게 만드는 일은 불가능하다.
  - 불변 객체가 작은 이유는 **생성자 안에서만 상태를 초기화할 수 있기 때문**이다.
- 불변객체로 클래스를 생성하려는 노력을 하면, 클래스의 크기는 자연적으로 작아진다.
- **불변성은 클래스를 더 깔끔하고 더 짧게 만든다.**
- **진정한 객체지향 소프트웨어에는 오직 불변 객체만이 존재해야 한다.**
  - 가변성은 절차 적인 프로그래밍의 끔직한 유산이다.

> **어떤 클래스도 가변 클래스로 만들지 마세요.**

7. 문서를 작성하는 대신 테스트를 만드세요

- **문서화**는 유지보수에 있어 중요한 구성요소 이다.
- **코드를 읽게 될 사람이 비즈니스 도메인, 프로그래밍 언 어, 디자인 패턴, 알고리즘을 거의 이해하지 못하는 주니어 프로그래머라고 가정해야 한다.**
- **코드를 읽을 사람이 자신보다 훨씬 더 멍청하다고 가정**해야 한다.
- 나쁜 프로그래머는 복잡한 코드를 짜고, **훌륭한 프로그래머는 단순한 코드**를 짠다.
- **이상적인 코드는 스스로를 설명**하기 때문에 어떤 추가 문서도 필요하지 않다.

```
//코드 스스로 설명을 해준다.
Employee jeff = department.employee("Jeff");
jeff.giveRaise(new Cash("$5,000"));
if (jeff.performance() < 3.5) {
  jeff.fire();
}
```

- **나쁜 설계는 문서를 작성하도록 강요한다.**

```
class Helper {
  int saveAndCheck(float x) { .. }
  float extract(String text) { .. }
  boolean convert(int value, boolean extra) { .. }
}
```

- **좋은 클래스**는 **목적이 명확하고, 작고, 설계가 우아**하다.
- 따라서, **코드를 문서화하는 대신 코드를 깔끔하게(clean) 만들어야 한다.**
  - **단위 테스트**도 함께 만들어야 한다는 의미이다.
  - 단위 테스트 역시 클래스의 일부로 취급해야 한다.
- **단위 테스트는 클래스의 일부**이지 독립적인 개체(entity)가 아니다.
- 깔끔하고 유지보수 가능한 단위 테스트를 추가하면, **클래스를 더 깔끔하게 만들 수 있고 유지 보수성을 향상**시킨다.
  - 더 훌륭한 단위 테스트를 작성시, 더 적은 문서화가 요구된다.
  - **단위 테스트가 바로 문서화 이다.**
  - 하나의 단위 테스트는 한 페이지 분량의 문서 만큼이나 가치가 있다.
- **단위 테스트는 클래스의 사용 방법을 보여주는데** 반해, 문서는 이해하고 해석하기 어려운 이야기를 들려준다.
- 단위 테스트를 올바르게 관리한다면, 실제 클래스보다 **단위 테스트를 훨씬 더 자주 읽게 될 것**이다.

```
@Test
@DisplayName("Cash 5와 Cash 6을 더하면, Cash 11을 리턴한다.")
public void plusTest(){
    Assertions.assertEquals(
            new Cash(5,0,"USD")
                    .plus(new Cash(6,0,"USD"))
            ,new Cash(11,0,"USD")
    );
}

@Test
@DisplayName("Cash 5에 6을 곱하면, Cash 30을 리턴한다.")
public void mulTest(){
    Assertions.assertEquals(
            new Cash(5,0,"USD")
                    .mul(6)
            ,new Cash(30,0,"USD")
    );
}
```

8. 모의 객체(Mock) 대신 페이크 객체(Fake)를 사용하세요.

```
interface Exchange{
  double rate(String source, String target);
}
class NYSE implements Exchange {
  private final String secret;
  public NYSE(String secret) {
      this.secret = secret;
  }

  @Override
  public double rate(String source, String target) {
    //뉴욕거래소의 http 결과값을 가정한다.
    return 2L;
  }
}
class Cash {
  private final Exchange exchange;
  private final int cents;

  public Cash(Exchange exchange, int cents) {
    this.exchange = exchange;
    this.cents = cents;
  }

  public Cash in(String currency) {
    return new Cash(
            this.exchange,
            (int) (this.cents * this.exchange.rate("USD", currency))
    );
  }
}
```

```
//실제 사용시
Cash dollar = new Cash(new NYSE("secret"), 100);
Cash euro = dotlar.in("EUR");//in 사용시 뉴욕거래소와 http 통신을 진행한다.
```

- 우리에게는 **NYSE 서버가 개입하지 않은 상황에서도 Cash 클래스를 테스트**할 수 있는 방법이 필요하다.
  - 테스트가 작동하지 않는다면, 오류로 판단되기 때문이다.
- NYSE서버 대신 **모킹(mocking)** 통해 테스트를 진행할 수 있다.

```
Exchange exchange = Mockito.mock(Exchange.class);
Mockito.doReturn(1.15)
       .when(exchange)
       .rate("USD", "EUR")
Cash dollar = new Cash(exchange, 500);
```

```
Cash euro = dollar.in("EUR");
assert "5.75".equals(euro.toString());
```

- **모킹은 나쁜 프랙티스이며, 최후의 수단으로만 사용해야 한다.**
- 모킹 대신 **페이크 객체**(fake object)를 사용해야 한다.

```
interface Exchange{
  double rate(String source, String target);

  final class Fake implements Exchange{
      @Override
      public double rate(String source, String target) {
          return 3L;
      }
  }
}

@Test
@DisplayName("페이크객체 NYSE에서 달러와 유료 비율은 1:3 이다.")
public void fakeNYSE(){
    Cash dollar = new Cash(new Exchange.Fake(), 500);
    Cash euro = dollar.in("EUR");
    Assertions.assertEquals(1500,euro.getCents());
}
```

- 페이크 클래스는 더 똑똑하게 만들어져야 하고, 강력해져야 한다.
  - 페이크 클래스가 실제 클래스보다 더 복잡한 경우도 있다.
- 페이크 클래스를 만족하도록 테스트를 작성하지 말고, **페이크 클래스가 테스트를 올바르게 지원**하도록 만들어야 한다.
- 페이크 클래스를 사용하면 **테스트를 더 짧게** 만들 수 있기 때문에 **유지보수성이 향상된다.**
  - **모킹**의 경우, 테스트가 매우 장황하고 **리팩터링하기 어려워진다.**
- **모킹은 가정** (assumption)을 **사실**(facts)로 전환시키기 때문에 단위 테스트를 유지보수하기 어렵게 만든다.
  - dollar.in 의 구현체를 테스트 할 수 없다. 이미 1.15 라고 fix 해버림, **가정을 사실로 전환해버린다**.
  - 불확실한 가정을 세우고 이 가정을 중심으로 전체 테스트를 구축할 수 있다.
- 클래스의 행동이 변경되면 단위 테스트가 실패하기 때문에, 단위 테스트는 코드 리팩토링에 큰 도움이 된다. 또한, 단위 테스트는 **행동이 변경되지 않을 경우에는 실패해서는 안된다**.

```
  @Test
  @DisplayName("페이크객체 NYSE에서 달러와 유료 비율은 1:4 이다.")
  public void fakeNYSE(){
      Cash dollar = new Cash(new Exchange.Fake(), 500);
      Cash euro = dollar.in();
      Assertions.assertEquals(2000,euro.getCents());
  }

  @Test
  @DisplayName("mock 객체")
  public void mokito(){
      Exchange exchange = Mockito.mock(Exchange.class);
```

```
        Mockito.doReturn(1.15)
                .when(exchange)
                .rate("USD", "EUR");

        Cash cash = new Cash(exchange, 300);
        Cash euro = cash.in("EUR");
        Assertions.assertEquals(345,euro.getCents());
    }
```

- 요점은 **모킹이 나쁜 프랙티스라는 사실이다.**
- 반대로 **페이크 클래스를 사용하면 테스트를 충분히 유지보수 가능**하게 만들 수 있다.
- Cash 클래스와 Exchange 클래스 사이의 **의사소통 방식에 대해서는 신경 쓸 필요가 없다.**
- 우리의 관심사는 **Cash와 우리 사이의 상호작용 방법**이지 Cash가 다른 클래스와 상호작 용하는 방법이 아니다.
- 테스트가 객체 내부의 구현 세부사항을 알면 테스트가 취약해지고 유지보수하기도 어려워지고, **문제의 근본 원인은 모킹**이다.
- 모킹은 또한 상호작용 횟수를 검증할 수 있는 기능을 제공하지만, 나쁜 아이디어 이다.
    - 단위 테스트를 상호작용에 의존하도록 만듦으로써, 리팩토링을 고통스럽고 때로는 불가능하게 만들기 때문이다.
- 페이크 클래스는 **인터페이스의 설계에 관해 더 깊이 고민하도록 해준다.**
    - 사용자의 관점에서도 고민하도록 도와준다.

> **객체와 의존 대상 사이의 상호작용 방식을 확인하거나 테스트해서는 안된다.**
> 이것은 객체가 캡슐화해야 하는 정보이다. 다시 말해서 **객체가 숨겨야 하는 비밀**입니다.

//TODO

9. 인터페이스를 짧게 유지하고 스마트(smart)를 사용하세요

- 올바로 설계된 견고하고, 응집도가 높은 클래스는 적은 수의 public 메서드만 포함한다
- 클래스를 작게 만드는 것이 중요하다면 **인터페이스를 작게 만드는 것은 훨씬 더 중요**하다.
    - **클래스가 다수의 인터페이스를 구현**하기 때문이다.

> 두 개의 인터페이스가 각각 5개의 메서드를 선언하고 있다면, 두 인터페이스를 모두 구현하는 클래스는 10개의 public 메서드를 가진다.
> **이 클 래스를 우아하다고 말하기는 어렵다.**

[형편 없는 인터페이스]

```
interface ExchangeBad {
    float rate(String target);
    float rate(String source, String target);
}

class NYSE_BAD implements ExchangeBad{
  @Override
  public float rate(String target) {
    return 1;
  }
  @Override
```

```
  public float rate(String source, String target) {
    return 1;
  }
}
class YAHOO_BAD implements ExchangeBad{
  @Override
  public float rate(String target) {
    return 2;
  }
  @Override
  public float rate(String source, String target) {
    return 2;
  }
}
```

- **무조건 override 강제한다.**
- ExchangeBad 인터페이스는 2개의 public 메서드를 구현 계약에 **강제한다.**
- 해당 인터페이스는 너무 많은 것을 요구하기 때문에 설계 관점에서는 **형편 없는 인터페이스** 이다.
  - **Exchange 인터페이스는 구현자에게 너무 많은 것을 요구한다.**
  - **단일 책임 원칙**(Single Responsibility Principle)을 **위반하는 클래스를 만들도록 부추긴다**.(응집도가 낮은 클래스)
  - 이 인터페이스 계약은 **거래소가 환율을 계산하도록 요구**하는 동시에 **클라이언트가 환율을 제공하지 않을 경우 기본 환율을 사용하도록 강제**한다.

[좋은 인터페이스]

- 이를 해결하기 위해 또 다른 인터페이스를 정의해야 할까요? 그럴 필요는 없다.
  - **인터페이스 안에 스마트**(smart) **클래스를 추가해서 해결**할 수 있습니다.

```
interface Exchange {
  float rate(String source, String target);

  final class Smart {
    private final Exchange origin;
    public Smart(Exchange origin) {
      this.origin = origin;
    }
    public float toUsd(String source) {
      return this.origin.rate(source, "USD");
    }
    public float eurToUsd() {
      return this.toUsd("EUR");
    }
  }
}

//사용
float rate = new Exchange.Smart(new YAHOO())
        .toUsd("EUR");
```

```
float rate = new Exchange.Smart(new YAHOO())
        .eurToUsd();
```

- 이 스마트 클래스는 Exchange 인터페이스가 어떻게 구현되고 환율이 어떻게 계산되는지는 모르지만, **인터페이스 위에 특별한 기능을 적용**한다.
    - Exchange의 서로 다 른 구현 사이에 공유될 수 있다.
- **스마트 클래스를 인터페이스와 함께 제공**해야 하는 또 다른 이유는 인터페이스를 구현하는 **서로 다른 클래스 안에 동일한 기능을 반복해서 구현할 필요가 없다**.
- 스마트 클래스의 크기는 점점 더 커지겠지만, **Exchange 인터페이스는 작고, 높은 응집도를 유지**된다.
    - 인터페이스에는 오직 하나의 메서드만 선언하고, NYSE, XE, Yahoo 등의 환율 제공자는 이 메서드를 구현한다.
    - 든 거래소들이 기능을 공유할 수 있기 때문에 **각 거래소들이 해당 기능을 개별적으로 구현할 필요가 없다.**
- 기본적으로 인터페이스를 짧게 만들고 스마트 클래스를 인터페이스와 함께 사용 함으로써 **공통 기능을 추출**하고 **코드 중복을 피할 수 있다.**

[테스트 코드]

```
@Test
@DisplayName("나쁜 인터페이스 : 여러개의 public 메서드를 노출시킨다")
public void badExchangeTest(){
    ExchangeBad bad = new YAHOO_BAD();
    Assertions.assertEquals(1.0F,bad.rate("USD"));
    Assertions.assertEquals(1.3F,bad.rate("EUR","USD"));
}

@Test
@DisplayName("나쁜 인터페이스 : override 강제성이 생긴다." +
        "각 구현체마다 특정 기능이 필요 없을수도 있기 때문이다.")
public void badExchangeTest2(){
    ExchangeBad bad = new NYSE_BAD();
    Assertions.assertThrows(RuntimeException.class,()-> {
        bad.rate("EUR","USD");
    });
}

@Test
@DisplayName("좋은 인터페이스 : 최대한 응집도를 높인다.")
public void exchangeTest(){
    Exchange.Smart good = new Exchange.Smart(new YAHOO());
    Assertions.assertEquals(1.3F,good.eurToUsd());
}

@Test
@DisplayName("좋은 인터페이스 : override를 강제하지 않는다.")
public void exchangeTest2(){
    Exchange.Smart good = new Exchange.Smart(new NYSE());
    Assertions.assertThrows(RuntimeException.class,()->{
        good.eurToUsd();
```

```
    });
}
```

[인터페이스 - 스마트 데코레이터]

- 인터페이스는 **우리 자신과 거래소의 사용자, NYSE 클래스 구현자 사이의 계약**이다.
- NYSE 클래스는 네트워크를 경유해 뉴욕 증권 거래소를 호출해서 어떤 작업을 수행한다고 **가정**
    - **스마트 클래스**는 네트워크 호출에 대한 어떤 것도 알아서 는 안됩니다
- 데코레이터가 스마트 클래스와 다른 점은 **스마트 클래스**가 **객체에 새로운 메서드를 추가**하는데 비해 데코레이터는 **이미 존재하는 메서드를 좀 더 강력하게 만든다는 점**이다.

```
interface Exchange {
    float rate(String origin, String target);
    final class Fast implements Exchange {
      private final Exchange origin;
      @Override
      public float rate(String source, String target) {
        final float rate;
        if (source.equals(target)) {
            rate = 1.0f;
        } else {
            rate = this.origin.rate(source, target);
        }
        return rate;
      }
      public float toUsd(String source) {
          return this.origin.rate(source, "USD");
      }
    }
}
```

- 중첩 클래스인 Exchange.Fast는 데코레이터인 동시에 스마트 클래스이다.
    - 첫째, Exchange.Fast는 rate() 메서드를 오버라이드해서 더 강력하게 만든다.
        - 이 클래스는 source와 target이 동일한 통화를 가리킬 경우에는 네트워크 호출을 피할 수 있도록 해준다.
    - 둘째, Exchange.Fast는 새로운 메서드인 toUsd()를 추가해서 USD로 쉽게 환율을 변환할 수 있도록 해준다.

[테스트코드]

- **각각 구현체마다** `if (source.equals(target)) { rate = 1.0f }` 을 넣어줘야하는 **귀찮음을 피할 수 있다.**(응집도 강화)

```
@Test
@DisplayName("공통 기능을 추출하고 코드 중복을 피할 수 있다.")
public void exchangeTestDecorator(){
    Exchange.Smart good = new Exchange.Smart(new NYSE());
    Exchange.Fast fast = new Exchange.Fast(new NYSE());
```

```
    Assertions.assertEquals(1F,fast.rate("USD","USD"));
}
```

[추상클래스 vs 인터페이스]

상속이 필요한 순간인지를 확실히 판단하는 것이 중요

- 추상클래스
  - 다른 클래스에서 **상속받아 확장하여 사용**한다.

```
abstract class Animal {
    private String name;
    public Animal(String name) {
        this.name = name;
    }
    public abstract String makeSound();  // 추상 메서드
    public String eat() {
        return name;
    }
}
public class Dog extends Animal{
  public Dog(String name) {
    super(name);
  }
  @Override
  public String makeSound() {
    return "멍멍";
  }
}
```

- 인터페이스
  - 인터페이스는 **기능을 추가할때 사용**한다.

```
interface Vehicle {
    int start();
    int accelerate();
    //Java 8 부터 추가되었다
    default String commonVehicle(){
        return "공통이닷";
    }
}
public class Tico implements Vehicle{
  @Override
  public int start() {
    return 0;
  }

  @Override
  public int accelerate() {
```

```
    return 0;
  }
}
```

[테스트코드]

```
@Test
@DisplayName("abstarct test")
public void abstractTest(){
    Animal animal = new Dog("개");
    Assertions.assertEquals("개",animal.eat());
    Assertions.assertEquals("멍멍",animal.makeSound());
}

@Test
@DisplayName("interface test")
public void interfaceTest(){
    Vehicle vehicle = new Tico();
    //java 8 부터 추가
    Assertions.assertEquals("공통이닷",vehicle.commonVehicle());
}
```

[Sealed Class - java 17]

- Sealed Class/Interface 는 상속하거나(extends), 구현(implements) 할 **클래스를 지정**해두고, **해당 클래스들만 상속/구현이 가능하도록 제한한다.**

[클래스]

```
public sealed class Shape permits Circle {
  private String name;
  public Shape(String name) {
    this.name = name;
  }
  public String getName() {
    return name;
  }
}

final class Circle extends Shape {
  private double radius;
  public Circle(double radius) {
    super("똥글이");
    this.radius = radius;
  }
  public double getArea() {
    return Math.PI * radius * radius;
```

```
    }
}
```

[인터페이스]

```
public sealed interface Exchange permits NYSE {
    int rate();
}
public final class NYSE implements Exchange {
  @Override
  public int rate() {
    return 0;
  }
}
```

[테스트]

```
@Test
@DisplayName("sealed class")
public void sealedClass(){
    Shape shape = new Circle(123);
    Assertions.assertAll(
            () -> assertEquals("똥글이",shape.getName()),
            () -> assertEquals(Math.PI * 123 * 123,((Circle)
shape).getArea())
    );
}

@Test
@DisplayName("sealed interface")
public void sealedInterface(){
    Shape shape = new Circle(123);
    Assertions.assertAll(
            () -> assertEquals("똥글이",shape.getName()),
            () -> assertEquals(Math.PI * 123 * 123,((Circle)
shape).getArea())
    );
}
```

Chapter 3.Employment

- 절차적인 프로그래밍과 **OOP의 중요한 차이점**은 **책임을 지는 주체가 무엇인가** 이다.
  - 절차적인 프로그래밍에서는 연산자, 명령문이 책임을 진다.
  - **OOP 에서는 객체에게 작업을 수행하도록 위임한다.**
- 이번챕터는 거대한 객체, 정적 메서드, NULL 참조, get/set, new 연산자에 반대하는 내용이다.

1. 5개 이하의 public 메서드만 노출하세요.

- 가장 우아하고, 유지보수가 가능하고, 응집력이 높으면서, 테스트하기도 용이한 객체는 **작은 객체**이다.
  - 모든 클래스의 크기를 250줄 이하로 유지하라!(섹션 2.6.7)
- **작은 객체란 무엇일까??**
  - 20개의 메서드를 가진 클래스(50줄) : **큰객체임**
  - **하나의 public 메서드와 20개의 private 메서드를 가진 클래스 : 작은객체**
- 따라서, 클래스의 크기를 정하는 기준으로 public 메서드(protected 메서드 포함)의 개수를 사용하기를 권장한다.
- public 메서드가 많을수록 클래스도 커지기 때문에, **5개보다 많다면 클래스를 리팩토링해야할 필요가 있는것이다.**
  - 4번째 메서드를 추가하고 나서 5번째 메서드를 추가하기 전에 잠시 숨을 고르고 클래스의 크기에 관해 고민 해야 한다.

> **클래스를 작게 만들어서 얻는 장점 으로는 무엇일까?**
>
> - 우아함, 유지보수성, 응집도, 테스트 용이성 향상
> - **클래스가 더 작을수록 실수할 가능성이 줄어들기 때문에** 작은 클래스는 더 우아하다.
> - **더 작은 클래스는 유지보수하기도 쉽다.**
>   - **코드양이 더 적고, 메소드의 수가 더 적고, 에러를 찾기 더 쉽고, 수정하기도 더 쉽다.**
> - **각각의 메서드가 객체의 진입점이고, 진입점의 수가 적다면 문제를 더 쉽게 고립**시킬 수 있다.
>   - **클래스가 작으면 메서드와 프로퍼티가 더 가까이 있을 수 있기 때문에 응집도가 높아진다**.
> - **작은 클래스는 테스트하기도 쉽고, 모든 사용 시나리오를 쉽게 재현**할 수 있다.
>   - 클래스가 작기 때문에 사용 시나리오의 수도 많지 않다.
>   - 클래스가 1개의 public 메서드만 포함하고 있 다면, 중요한 모든 테스트를 아주 쉽게 작성할 수 있다.

[큰 클래스]

```
public class OrderProcessor {
  public boolean placeOrder(int orderId, String productName, int quantity)
{
    // 주문을 처리하는 로직: 예시로 간단히 구현
    if (quantity <= 0) {
      throw new IllegalArgumentException("주문 수량은 양수여야 합니다.");
    }
    // 실제로는 데이터베이스에 주문을 저장하는 로직
    System.out.println("주문이 접수되었습니다 - 주문번호: " + orderId);
    return true;
  }

  public boolean cancelOrder(int orderId) {
    // 주문을 취소하는 로직: 예시로 간단히 구현
    // 실제로는 데이터베이스에서 주문을 삭제하는 로직
    System.out.println("주문이 취소되었습니다 - 주문번호: " + orderId);
    return true;
  }
}

public class OrderProcessorTest {

    @Test
```

```
    public void testPlaceOrder_validOrder() {
        // 주문이 제대로 처리되는지 확인
        OrderProcessor processor = new OrderProcessor();
        int orderId = 1001;
        String productName = "Product A";
        int quantity = 5;
        boolean result = processor.placeOrder(orderId, productName,
quantity);
        Assertions.assertTrue(result); // 주문 처리가 성공했는지 확인
    }

    @Test
    public void testPlaceOrder_invalidQuantity() {
        // 잘못된 주문 수량일 때 IllegalArgumentException이 발생하는지 확인
        OrderProcessor processor = new OrderProcessor();
        int orderId = 1002;
        String productName = "Product B";
        int quantity = -1; // 잘못된 수량
        Assertions.assertThrows(IllegalArgumentException.class,
                () -> processor.placeOrder(orderId, productName,
quantity));
    }

    @Test
    public void testCancelOrder_validOrder() {
        // 주문 취소가 제대로 처리되는지 확인
        OrderProcessor processor = new OrderProcessor();
        int orderId = 1003;
        boolean result = processor.cancelOrder(orderId);
        Assertions.assertTrue(result); // 주문 취소가 성공했는지 확인
    }
}
```

[작은 클래스]

- 주문 클래스

```
public class Order {
    private int orderId;
    private String productName;
    private int quantity;
    Order(int orderId, String productName, int quantity) {
        this.orderId = orderId;
        this.productName = productName;
        this.quantity = quantity;
    }

    public boolean placeOrder() {
        // 주문을 처리하는 로직: 예시로 간단히 구현
        if (quantity <= 0) {
            throw new IllegalArgumentException("주문 수량은 양수여야 합니다.");
```

```
        }
        // 실제로는 데이터베이스에 주문을 저장하는 로직
        System.out.println("주문이 접수되었습니다 - 주문번호: " + orderId);
        return true;
    }
}

public class OrderTest {
  @Test
  public void testPlaceOrder_validOrder() {
    // 유효한 주문을 처리하는지 확인
    int orderId = 1001;
    String productName = "Product A";
    int quantity = 5;
    Order order = new Order(orderId, productName, quantity);
    boolean result = order.placeOrder();
    Assertions.assertTrue(result); // 주문 처리가 성공했는지 확인
  }

  @Test
  public void testPlaceOrder_invalidQuantity() {
    // 잘못된 주문 수량일 때 IllegalArgumentException이 발생하는지 확인
    int orderId = 1002;
    String productName = "Product B";
    int quantity = -1; // 잘못된 수량
    Order order = new Order(orderId, productName, quantity);
    Assertions.assertThrows(IllegalArgumentException.class, () ->
order.placeOrder());
  }
}
```

- 주문 취소 클래스

```
public class OrderCancel {
    private int orderId;
    public boolean cancelOrder(int orderId) {
        // 주문을 취소하는 로직: 예시로 간단히 구현
        // 실제로는 데이터베이스에서 주문을 삭제하는 로직
        System.out.println("주문이 취소되었습니다 - 주문번호: " + orderId);
        return true;
    }
}

public class OrderCancelTest {
  @Test
  public void testCancelOrder_validOrder() {
    // 유효한 주문을 취소하는지 확인
    int orderId = 1001;
    OrderCancel orderCancel = new OrderCancel();
    boolean result = orderCancel.cancelOrder(orderId);
    Assertions.assertTrue(result); // 주문 취소가 성공했는지 확인
```

```
    }
}
```

2. 정적 메서드를 사용하지 마세요.

- **OOP에 static은 순수한 악이다.**
- 정적 메서드를 사용하고 있는지 여부는 OOP를 제대로 이해하지 못한 형편없는 프로그래머를 구별하기 위해 사용할 수 있는 최적의 지표
    - 어떤 상황에서도 **정적 메서드**에 대해서는 **변명의 여지가 없다.**
    - **성능 역시 중요한 요소가 아니다.**
- **정적 메서드는 객체 패러다임의 남용이다.**
- **정적 메서드는 소프트웨어를 유지보수하기 어렵게 만든다.**

[정적 메서드 사용]

```
class WebPage {
    public static String read(String uri) {
    // HTTP 요청을 만들고
    // UTF-8 문자열로 변환한다
    }
}
//사용
String html = WebPage.read("http://www.java.com");
```

[정적 메서드 사용 X]

```
class WebPage {
    private final String uri;
    public String content() {
    // HTTP 요청을 만들고
    // UTF-8 문자열로 변환한다
    }
}
//사용
String html = new WebPage("http://www.java.com").content();
```

1) 객체 대 컴퓨터 사고(object vs computer thinking)

- 우리는 어셈블리어, C, COBOL, Basic, Pascal 등의 초창기 프로그래밍 언어로부터 **컴퓨터처럼 생각하는 방법을 물려받았다.**
    - 우리는 명시적인 명령어를 제공해 서 컴퓨터에게 지시를 내린다
    - 프로그래머가 CPU와 유사한 방식으로 수행될 작업을 CPU에게 직접 지시한다.

```
//비교할 두 정수를 AX와 BX 레지스터에 넣고, 최종 결과는 CX 레지스터에 저장하는 로직
 CMP AX, BX
 JNAE greater
 MOV CX, BX
 RET
greater:
 MOV CX, AX
 RET

//최대값 리턴
int max(int a, int b) {
   if (a > b) {
     return a;
   }
   return b;
}
```

- **순차적인** 사고 방식을 가리켜 **컴퓨터 입장**(thinking as a computer)에서 **생각하기** 라고 부른다.
    - 규모가 더 커지면 **순차적인 사고방식은 한계에 직면**한다.
- 우리는 CPU로부터 분리되어 있디. **우리는 지금 컴퓨터가 아니라, 함수처럼 사고**하고 있습니다. 새로운 무엇(thing)이 필요하다면 그 무엇을 정의한다.
    - (def x (max 5 9))
    - 단순히 x는 이 두 수의 최댓 값이다(is a)라고 이야기 한다.
- 우리는 CPU에게 할일을 지시하는 것이 아니라 **정의**한다.
    - x는 최댓값이다(is a)라고 정의하는 것이 핵심이다.
- 함수형, 논리형, **객체지향 프로그래밍이 절차적 프로그래밍과 차별화되는 점**이 바로 이 is a이다.
- **컴퓨터**처럼 생각하기에서는 명령의 실행 흐름을 **제어할 책임이 우리**에게 있다.
- **객체지향적**으로 생각하기에서 **우리는 누가 누구인지만 정의**하고 **객체들이 필요할 때 스스로 상호작용하도록 위임**한다.

[객체지향에서 최대값을 구하는 코드]

- 코드는 최댓값을 계산하지 않는다.
- 그저 X가 5와 9의 최댓값 이라는(is a) **사실을 정의**한다.
    - CPU에게 계산과 관련된 어떤 지시도 내리지 않고, **단순히 객체를 생성**한다.
    - 반대로, OOP의 정적 메서드는 정확하게 C와 어셈블리어의 서브루틴과 동일하다.
        - int x = Math.max(5, 9);는 잘못된 방식이다.

```
class Max implements Number {
  private final Number a;
  private final Number b;
  public Max(Number left, Number right) {
    this.a = left;
```

```
    this.b = right;
  }
  public Number getMax() {
    // a와 b 중에서 큰 값을 반환
    if (a.doubleValue() > b.doubleValue())
        return a;
    return b;
  }
}
//사용
Number x = new Max(5, 9);
```

2) 선언형 스타일 대 명령형 스타일(declarative vs imperative style)

- 명령형 프로그래밍(imperative programming)
  - **프로그램의 상태를 변경하는 문장을 사용해서 계산 방식을 서술**한다.
  - 명령형 프로그래밍은 **컴퓨터처럼 연산을 차례대로 실행**한다.
- 선언형 프로그래밍(declarative programing)
  - **제어 흐름을 서술하지 않고 계산 로직을 표현**한다.
  - 선언형 프로그래밍은 엔티티와 엔티티 사이의 관계로 구성되는 자연스러운 사고 패러다임에 가깝다.
- **명령형 프로그래밍과 선언형 프로그래밍이 정적 메서드와 무슨 상관이 있을까?**
  - 정적 메서드를 사용하든, 객체를 사용하든, 여전히 어딘가에서는 if(a>b)가 참인지를 확인한다.
  - 둘 사이의 차이점은 다른 클래스, 객체, 메서드가 이 기능을 **사용하는 방법**에 있다.

[명령형 프로그래밍]

- max()가 정 적 메서드라면 다음과 같이 구현해야만 한다.
- **계산은 즉시 수행**되기 때문에 **between()은 명령형 스타일의 연산자**이다.
- 메서드를 호출한 시점에 **CPU가 즉시 결과를 계산**한다. (**명령형**)

```
public static int between(int I, int r, int x) {
    return Math.min(Math.max(l, x), r);
}

//사용
int y = Math.between(5, 9, 13); // 9를 메소드 호출과 동시에 반환
```

[선언형 프로그래밍]

- **CPU에게 숫자를 계산하라고 말하지 않았기 때문에, 이 방식은 선언형 스타일** 이다.
- **Between이 무엇인지만 정의**하고, 변수 y의 사용자가 intValue()의 값을 **계산하는 시점을 결정**한다.
- **오직 선언만 했다는 점이 중요하며, CPU에게 어떤 일을 하라고 지시하지 않았다.**
- **제어를 서술하지 않고, 로직만 표현했다.**

```
class Between implements Number {
    private final Number num;
```

```
    Between(Number left, Number right, Number x) {
        this(new Min(new Max(left, x), right));
    }
    @Override
    public int intValue() {
        return this.num.intValue();
    }
}
//사용
Number y = new Between(5, 9, 13); // 메서드 호출 전에는 호출하지 않는다.
```

[왜 선언형 프로그래밍이 더 강력할까?]

- **첫째, 먼저 선언형 방식은 더 빠르다.**
  - 선언형 방식에서는 **우리가 직접 성능 최적화를 제어할 수 있기 때문에 더 빠르다.**
  - 선언형 프로그래밍은 **CPU에게 결과가 실제로 필요한 시점과 위치를 결정하도록 위임**하고, **CPU는 요청이 있을 경우에만 계산을 실행 한다.**

```
# 명령형
public void dolt() {
  int x = Math.between(5, 9, 13);
  if (/* x가 필요한가? */) {
      System.out.println("x=" + x);
  }
}

# 선언형
public void dolt() {
  Integer x = new Between(5, 9, 13);
  if (/* x가 필요한가? */) {
    System.out.println("x=" + x);
  }
}
```

  - **최적화 관점**에서 **직접 통제할 수 있는 코드가 많을수록 유지보수하기도 더 쉽다.**
- **둘째, 다형성**(polymorphism) 때문이다.
  - **다형성** 이란 코드 블록 사이의 **의존성을 끊을 수 있는 능력**을 말한다.
  - **클래스를 다른 알고리즘과 함께 조합해서 사용할 수 있다.**
  - 객체지향 프로그래밍에서 객체는 **일급 시민**(first- class citizen) 이다.
  - 선언형 프로그래밍을 이용 하면 객체 사이의 **결합도를 낮출 수 있다.**
  - 정적 메서드를 사용한다면(Math.max() , Math.min()) 리팩터링 하기 어려워진다.

```
public interface OwnAlgorithm {
}

class Between{
private final OwnAlgorithm own;
    public Between(OwnAlgorithm own) {
```

```
            this.own = own;
        }
    }
    //사용
    Integer x = new Between(
      new IntegerWithMyOwnAlgorithm(5, 9, 13)
    );

    Integer x = new Between(
      new IntegerWithMyOwnAlgorithm2(5, 9, 13)
    );
```

  - Between과 Max, Min은 모두 클래스이기 때문에 Max와' Min으로부터 Between을 아주 쉽게 분리할 수 있다.
  - 명령형 코드에서도 위와 동일한 방법으로는 **연산을 분리할 수 없다. (리팩토링 불가능).**
    `int y = Math.between(5, 9, 13);`
    - 새로운 정적 메서드를 네 번째 파라미터로 전달하는것이 유일하다.
- **셋째**, 선언형 방식이 **표현력**(expressiveness) 때문 이다.
  - 선언형 방식은 **결과**를 이야기하는데 반해, 명령형 방식은 수행 가능한 한 가지 **방법**을 이야기한다.
  - 알고리즘(algorithm)과 실행(execution) 대신 **객체**(object)와 **행동**(behavior)의 **관점에서 사고해야한다.**
    - 명령형 은 알고리즘과 실행을 다룬다.
    - **선언형은 객체와 행동에 관한 방법이다.**

```
  //명령형 - 방법을 이야기한다.
  Collection<Integer> evens = new LinkedList<>();
  for(int number : numbers){
      if(number % 2 == 0){
          evens.add(number);
      }
  }

  //선언형 : Filtered 가 어떤 역할을 하는지 알수 없고, 단순히 필터링을 해준다는 사실만
알 수 있다.
  Collection<Integer> evens = new Filtered(
      numbers,
      new Predicate<Integer>(){
          @Override
          public boolean suitable(Integer number){
              return number %2 == 0;
          }
      }
  );
```

- **넷째**, **코드 응집도**(cohesion) 때문이다.
  - 선언형은 컬렉션의 계산을 책임지는 모든 코드들은 한 곳에 뭉쳐 있기 때문에, 실수로라도 **분리할 수 없습니다**
    - evens = new Filtered(...)
  - 명령형 코드에서는 코드의 각 줄을 이어주는 접착제가 없다.

    - 실수로 코드의 순서를 쉽게 변경할 수 있으며 그로 인해 알고리즘에 오류 가 발생할 가능성이 높다.
    - 선언형 프로그래밍 스타일 역시 **시간적인 결합 문제를 제거**할 수 있다.

> 정적 메서드를 재사용하면 순수하고 깔끔한 객체지향 코드를 작성하기란 불가능하다.
> 정적 메서드는 객체지향 소프트웨어의 **암적인 존재**이다.
> 오픈소스에 정적 메서드를 사용하는것이 많다.
>
> - 가장 좋은 방법은 여러분의 코드가 객체를 직접 처리할 수 있도록 **정적 메서드를 감싸는 클래스를 만들어 고립시키는 것**입니다

```
public static List<String> readLines(final File file, final Charset
encoding) throws IOException {
    try (InputStream in = openInputStream(file)) {
        return IOUtils.readLines(in, Charsets.toCharset(encoding));
    }
}

class FileLines implements Iterable<String> {
    private final File file;
    public Iterator<String> iterator() {
        return Arrays.asList(FileUtils.readLines(this.file)).iterator();
    }
}
```

3 유틸리티 클래스(Utility classes)

- 유틸리티 클래스를 클래스라고 부르기 어려운 이유는 **인스턴스를 생성하지 않기 때문**이다.
    - 객체와 클래스의 차이점을 설명하면서 **클래스를** 객체의 팩토리라고 정의하엿다.
    - **유틸리티 클래스는 어떤 것의 팩토리가 아니기 때문에 진짜 클래스라고 부를 수 없다.**
- 유틸리티 클래스는 편의를 위해 정적 메서드들을 모아 놓은 정적 메서드들의 컬렉션이다.
- 절차적인 프로그래머들이 OOP라는 영토에서 거둔 승리의 상징이다.
- 유틸리티 클래스는 최대한 지양하자.(정적 메서드의 단점을 가지기 때문)
    - **유틸리티 클래스는 끔찍한 안티패턴이다.**

```
class Math {
  private Math() {
    //...
  }
  public static int max(int a, int b) {
  if (a < b) {
    return b;
    return a;
  }
}
```

4 싱글톤(Singleton) 패턴

- 싱글톤 패턴은 정적 메서드 대신 사용할 수 있는 매우 유명한 개념이다.
- 싱글톤 패턴은 유명하지만, 사실 끔찍한 **안티 패턴**이다.

```
#싱글톤
class Math(){
  private static Math INSTANCE = new Math();
  private Math(){}
  public static Math getInstance(){
      return Math.INSTANCE;
  }
  public static int max(int a, int b){
      if(a < b){
          return b;
      }
      return a;
  }
}
#정적메서드
class Math(){
  private Math(){}
  public static int max(int a, int b){
      if(a < b){
          return b;
      }
      return a;
  }
}

Math.max(5,9);//유틸리티 클래스
Math.getInstance().max(5,9)//싱글톤

> 앞서 말했던 정적메서드와 싱글톤의 사용이 동일하기 때문에 싱글톤은 좋지 않다.
```

```
# 싱글턴
class User {
  private static User INSTANCE = new User();
  private String name;
  private User() {}
  public static User getInstance() {
    return User.INSTANCE;
  }
  public String getName() {
    return this.name;
  }
  public String setName(String txt) {
    this.name = txt;
  }
}

# 유틸
```

```
class User {
  private static String name;
  private User() {}
  public static String getName() {
    return User.name;
  }
  public static String setName(String txt) {
    User.name = txt;
  }
}
```

- **싱글톤 패턴**과 **유틸클래스**의 핵심적인 차이는 무엇일까?
    - 싱글턴 패턴은, **getInstance()** , **setInstance()** 를 추가 할 수 있다는 것이다.
    - 싱글턴 패턴은 fake 객체를 만들어 인스턴스를 주입하며, 테스트 코드에 쉽게 사용할 수 있다.
    - ```
      Math math = new FakeMath();
      Math.setInstance(math);
      ```
- **캡슐화된 객체를 변경할 수 있기 때문에 싱글톤이 유틸리티 클래스보다는 상대적으로 더 좋다.**
- 하지만 싱글턴도 여전히 안티패턴이다.
- 전역 변수 그 이상도 그 이하도 아니기 때문이다.
- **OOP에는 전역 범위(global scope)가 없다.**
- **유틸리티 클래스는 분리할 수 없는 하드코딩된 의존성**입니다
- Java에서 전역 변수를 사용할수 있는 방법을 발견했고, 그 결과로 만들어진 것이 바로 싱글톤 이다.
    - 싱글톤은 **전역변수** 그 이상도 그 이하도 아니다.
- **클래스가 작업을 수행하는데 필요한 모든 요소들이 생성자에 제공되고 내부에 캡슐화**돼야 한다.
- **절대로 싱글톤을 사용하지 말자.**

5 함수형 프로그래밍

```
class Max implements Number {
  private final int a;
  private final int b;
  public Max(int left, int right) {
    this.a = left;
    this.b = right;
  }
  @Override
  public int intValue() {
    return this.a > this.b ? this.a : this.b;
  }
```
```

```
}

//사용
Number x = new Max(5, 9);
```

- 이 책에서 권장하는 방식에 따라 객체를 우아하게 만들었다면, **함수와 객체 사이에 는 많은 부분이 유사**해진다.
- **그렇다면 왜 객체가 필요할까?**
- **FP보다 OOP의 표현력이 더 뛰어나고 강력하기 때문이다.**
  - **FP에서는 오직 함수만 사용할 수 있지만 OOP에서는 객체와 메서드를 조합할 수 있다.**
- FP를 향한 일환으로 Java에서는 람다표현식이 생겼고, 객체지향 스타일로부터 우리를 멀어지게 만든다.
- FP도 훌륭한 패러다임이지만, OOP가 더 낫다. 특히 제대로 사용할 경우에는 더욱 확실해진다.
- **이상적인 OOP 언어에는 클래스와 함께 함수가 포함되어야 한다.**

6 조합 가능한 데코레이터 (다른 객체를 감싸는 객체)

- 조합 가능한 데코레이터(composable decorator) : 필자가 고안한 용어이다.
- **조합 가능한 데코레이터는 그저 다른 객체를 감싸는 객체일 뿐이다.**
  - 데코레이터 객체 들을 다중계층 구조(multi-layer structures)로 구성하기 시작하면 다음 예제처럼 조합 가능(composable)해진다.
- **단순히 선언**만 했을 뿐인데, 객체가 생성되었기 때문에 이상적이다.(내부에서 어떻게 생성되는지 알 필요가 없음)
  - 객체를 어떻게 만들었는지를 전혀 설명하지 않고도 **이 객체가 무엇인지를 설명**한다.
  - Unique, Capitalized 등등이 하나의 데코레이터이다.
- **각 데코레이터는 내부에 캡슐화하고 있는 객체에 별도의 행동을 추가한다.**
- **데코레이터의 상태는 내부에 캡슐화하고 있는 객체들의 상태와 동일하다.**
- **프로그래머는 데코레이터를 조합하는 일을 제외한 다른 일은 하지 말아야 한다.**
  - if, for, switch, while과 같은 절차적인 문장(statements)이 포함되어 있어서는 안된다.

```
# 매우 깔끔하면서도 객체지향적
# 복잡해 보이지만, 이상적인 코드이며, 객체지향적이다.
# 또한 순수하게 선언형이다.
names = new Sorted(
    new Unique(
      new Capitalized(
          new Replaced(
              ...
          )
      )
    )
  )
```

- 순수한 객체지향에서는 if, for, switch while 연산자는 필요없다. **클래스로 구현된 If, For, While** 이 필요 할 뿐이다.
- 작으면서도 조합 가능한 클래스들을 설계하고, 더 큰 객체를 조합하기 위해 작은 클래스들을 재사용할 수 있도록 만들어야 한다.
- **객체지향 프로그래밍이란 더 작은 객체들을 기반으로 더 큰 객체들을 조합하는 작업이다.**
- 하지만 지금까지 설명한 모든 내용들이 정적 메서드와 어떤 관련이 있는 걸까?
  - 이미 예상하고 있겠지만 **정적 메서드는 조합이 불가능하다.**
  - **정적 메서드를 포함하는 작은 객체들을 조합해서 더 큰 객체를 만들수가 없다.**

```
float rate;
if(client.age() > 65){
  rate = 2.5;
}else{
  rate = 3.0;
}

float rate = new If(
  client.age() > 65, 2.5, 3.0
)

float rate = new If(
  new Greater(client.age(), 65), 2.5, 3.0)
)

float rate = new If(
  new GreaterThan(new AgeOf(client),65),
  2.5, 3.0
)
```

```
> **최종 정리**
> - 객체지향 프로그래밍이란 더 작은 객체들을 기반으로 더 큰 객체들을 조합하는 작업이다.
> - 정적 메서드는 조합이 불가능하다. OOP에서 정적(static) 메서드를 사용해서는 안된다.
```

3. 인자의 값으로 NULL을 절대 허용하지 마세요

- **NULL은 객체지향 세계의 커다란 골치거리 중 하나**이다.
  - 코드 어딘가에 NULL이 존재한다면 여러분은 커다란 실수를 저지르고 있는것이다.
- NULL을 허용하는 find() 메서드를 구현예제이다.
  - 이코드의 문제는 mask == null 이다. mask 객체에게 **이야기하는** 대신 **객체를 피하고 무시**한다.
  - 객체의 면전에 대고 "당신과 이야기 나눌 가치가 있나요?". "그 객체에게 말할 가치가 있나요?"라고 묻는 것과 동일하다.
  - 두 말할 필요 없이 예의바른 의사소통 방식이 아니다.
  - NULL 여부를 체크함으로써 객체가 맡아야 하는 상당량의 책임을 빼앗는다.
  - 우리는 **외부에서 자신의 데이터를 다뤄주기만을 기대하고 스스로를 책임질 수 없는멍청한 자료구조로 객체를 퇴화시키고 있는 것**이다.

```
# 객체를 무시하는 코드
public Iterable<File> find(String mask){
  if(mask==null){
  //모든 파일을 찾음
  }else{
  //마스크를 이용해 파일을 찾음
  }
}
```

```
# 객체를 존중하는 코드
public Iterable<File> find(Mask mask){
  if(mask.empty()){
  //모든 파일을 찾음
  }else{
  //마스크를 이용해 파일을 찾음
  }
}
```

- 대부분의 언어들이 더 이상 포인터를 제공하지 않음에도 불구하고, 불행하게도 객체지향 세계에서는 이 null의 개념을 물려받았다.
  - **Java에는 포인터가 없기 때문에 역참조(de-reference)가있어야 할 이유가 없다.**
- 여전히 Java에 null이 존재하는 이유는 무엇일까?
- 필자는 이것이 Java 언어가 안고 있는 설계 상의 커다란 실수라고 생각한다.
- **객체를 존중**했다면 조건의 존재 여부를 **객체 스스로 결정**하게 해야한다.
- **NULL인지를 확인하는 짐을 메소드 구현자에게 떠넘겨서는 안된다.**
  - 대신 항상 객체를 전달하되, 전달한 객체에게 무리한 요청을 한다면 **응답을 거부하도록 객체를 구현**해야 합니다.
- 클라이언트가 여전히 NULL을 전달한다면 어떻게 대응을 해야 할까?
  - **null을 체크하여 예외를 던진다.** if(mask == null) throw new Expception("object can't be null");
  - **null을 무시한다**. NullpointException이 나오면, 자신이 실수 했다는 것을 인지 할 수 있다.
- **중요하지 않은 NULL 확인 로직으로 코드를 오염시켜서는 안된다.**
- **올바른 방식으로 설계된 소프트웨어에는NULL 참조가 존재해서는 안된다.**
- **NULL을 절대 허용하지 말자. 예외는 없다.**

4. 충성스러우면서 불변이거나, 아니면 상수이거나

- 불변성에 반대하는 사람들은 세상은 본질적으로 가변적이기 때문에 불변 객체만으로 세상을 표현 불가능하다고 주장한다.
  - 이런 관점이 많은 사람들의 상식에는 부합할지 몰라도 **동의할 수 없다.**
  - 세상은 가변적이지만, **불변 객체로 세상을 모델링할 수 있다.**
- 많은 사람들이 혼란스러워 하는 이유는 서로 다른 개념인 **상태(state)와 데이터(data)에 관해 오해하고 있기 때문이다.**
  - 객체의 행동이나 메서드의 반환값은 중요하지 않다.
  - 핵심은 객체가 살아 있는 동안 상태가 변하지 않는다는 사실이다.
  - content() 메서드를 호출할 때마다 서로 다른값이 반환되더라도 이 객체는 **불변**이다.
  - **객체가 대표하는 엔티티에 충성하기(loyal) 때문 에불변 객체로 분류된다.**

```
class WebPage{
  private final URI uri;
  WebPage(URI path){
      this.uri = path;
  }

  public String content(){
      //HTTP GET 요청을 전송해서, 웹컨텐츠를 읽는다.
```

```
    }
}
```

- **불변 객체의 메서드를 호출할 때마다 상수(constant)처럼 매번 동일한 데이터가 반환되리라 기대한다.**
    - 불변 객체의 정의에 따르면 적절해 보일 수도 있지만, **결론적으로 이러한 사고 방식은 틀렸다.**
    - **상수처럼 동작하는 객체는 단지 불변성의 특별한 경우(corner case)일 뿐이다.**
        - **불변 객체는 그 이상이다.**
- **객체의 상태**
    - 객체란 디스크에 있는 파일, 웹페이지, 해시맵, 달력의 월과 같은 **실제 엔티티(real-life entity)의 대표자** 이다.
    - 실제라는 말은 **객체의 가시성 범위(scope of visibility) 밖에 존재하는 모든 것을 의미**한다.
        - **f의 가시성 범위는 echo() 메서드이다.**
        - **객체 f는 '/tmp/test.txt'파일의 대표자**이다.
        - 우리의 관점에서 **객체 f는 echo() 메서드 안에서만큼은 파일**이다.
        - 디스크에 저장된 파일을 다루기 위해 객체는 파일의 좌표를 알아야한다.
            - 이 좌표를 다른 말로 **객체의 상태(state)라고 부른다.**
            - 객체 f의 좌표(상태)는 `/tmp/test.txt` 이다.

```
public void echo(){
  File f = new File("/tmp/test.txt");
  System.out.println("File size: %d", file.lengthQ);
}
```

- 모든 객체는 식별자(identity), 상태(state), 행동(behavior)을 포함한다.
    - **식별자**는 f를 다른 객체와 구별한다.
    - **상태**는 f가 디스크 상의 파일에 대해 알고 있는 것이다.
    - **행동**은 요청을 수신했을 때 f가 할 수 있는 작업을 나타낸다.
- 불변 객체와 가변 객체의 중요한 차이는 **불변 객체에는 식별자가 존재하지 않으며, 절대로 상태를 변경할 수 없다.**
    - **불변 객체의 식별자는 객체의 상태와 완전히 동일하다.**
    - ex) 동일한 URI(상태 이면서 식별자)를 가진 두 개의 WebPage 인스턴스를 생성할 경우 이들은 서로 다른 같은 객체이다.
- equals()와 hashCode() 메서드 모두 **캡슐화된 uri 프로퍼티에 기반**하며, WebPage 클래스의 **객체들을 투명하게(transparent) 만든다.**
    - **투명하다** 라는 말은 객체들이 더 이상 자기 자신만의 식별자를 가지지 않는다는 뜻이다.(**외부 세계에서 대표자이다**)
    - 객체들은 **웹 상의 페이지를 대표**하며, **객체들이 포함하는 유일한 상태는 URI 형태의 페이지 좌표뿐이다.**

```
class WebPage {
    private final URI uri; WebPage(URI path) {
        this.uri = path;
```

```
    }
    @Override
    public void equals(Object obj) {
        return this.uri.equals( WebPage.class.cast(obj).uri);
    }
    @0verride
    public int hashCode() {
        return this.uri.hashCodeQ;
    }
}
```

- 반면에 **가변객체**는 완전히 다른 방식으로 동작한다.
    - **가변 객체의 상태는 변경이 가능**하기 때문에, 상태에 독립적인 식별자를 별도로 포함해야 한다.
    - **가변 객체**는 대표하는 **엔티티의 좌표를 변경**할 수 있기 때문에 충성스럽지 않다.
- 완벽한 객체지향 세계에는 불변 객체만 존재하기 때문에 equals()와 hashCode() 메서드가 필요하지 않다.
    - **캡슐화된 상태**만으로도 **불변 클래스의 모든 객체들을 식별**할 수 있다.
    - **상태**는 불변 객체를 식별하기 위한 필요충분조건이다.
- **불변 객체는 좌표를 알고, 우리는 이 좌표를 상태(state)라고 부른다.**
- **불변 객체**는 자신이 대표하는 실세계의 엔티티에게 충성(loyal)한다.
- **불변 객체**는 엔티티의 좌표를 절대로 변경하지 않는다.
- **불변 객체**는 항상 동일한 엔티티를 대표한다.
- 객체가 대표하는 **실제 엔티티와 객체의 상태가 동일한 경우**에는 불변 대신 상수(constant)라는 용어를 사용하길 권장 한다.
    - this.array는 ConstantList의 상태인 동시에 ConstantList 객체가 대표하는 엔티티와 동일(same)하다.
    - **ConstantList에서는 대표하는 엔티티가 바로 상태이다.**
    - 즉, **상수 객체는 불변 객체의 특별한 경우일 뿐이다.**

[WebPage 새로운 메서드 추가]

- **page 객체의 상태가 바뀌었나? 그렇지 않다.**
- **page는 여전히 불변 객체일까? 틀림없다.**
- 객체가 대표하는 실제 웹 페이지는 불변일까? **불변이 아닐 확률 이 높다.**

```
class WebPage {
  private final URI uri;
  WebPage(URI path) {
      this.uri = path;
  }
  public void modify(String content) { // HTTP PUT 요청을 보내서
  // 웹 페이지 내용을 수정한다.
  }
}
```

> **불변 객체의 식별자는 객체의 상태와 완전히 동일하다.**
> **엔티티 : 식별자를 갖는다**

5. 절대 getter와 setter를 사용하지 마세요

- 가변 클래스
- 잘못된 메서드 명칭
- **Cash가 진짜 클래스가 아니라 단순한 자료 구조(data structure)라는 사실**이다.(용서 불가)

```
class Cash {
    private int dollars;
    public int getDollars() {
        return this.dollars;
    }
    public void setDollars(int value) {
        this.dollars = value;
    }
}
```

1. 객체 대 자료구조

- 자료구조 : 어떤 개성도 지니고 않은 단순한 **데이터 가방**일 뿐이다.
- 객체 : **멤버에게 접근하는 것을 허용하지 않는다.** 또한 **캡슐화된 어떤 멤버가 어떤 작업에 개입하는지도 알 수 없다.**
    - **캡슐화(encapsㄴlation)이며, OOP가 지향하는 가장 중요한 설계 원칙 중 하나**이다.
- 자료구조는 투명하지만, **객체**는 **불투명**하다.
- 자료구조는 수동적이지만, **객체는 능동적** 이다.
- 자료구조는 죽어있지만, **객체는 살아 있다.**
- 자료구조가 나쁜 이유는 무엇일까?
    - 유지보수성과 관련이 있다.
- 모든 프로그래밍 스타일의 핵심 목표는 **가시성의 범위를 축소해서 사물을 단순화**시키는것이다.
    - 특정한 시점에 이해해야 하는 범위가 작을수록, **소프트웨어의 유지보수성이 향상되고 이해하고 수정하기도 쉬워진다.**
- 데이터는 더 이상 앉아서 기다리지 않는다. **데이터는 객체 안에 캡슐화되어 있고, 객체는 살아 있다.**
    - 객체들은 서로 연결되고, **어떤 일을 수행해야 할 때는 메시지 (message)를 전송해서 작업을 실행**한다.
- OOP에서는 **코드가 데이터를 지배하지 않는다.** 대신 필요한 시점에 객 체가 자신의 코드를 실행시킨다.
- 객체가 일급 시민이며, 생성자를 통한 객체 초기화가 곧 소프트웨어 이다.

> **일급 시민**
>
> - 변수에 담을 수 있다.
> - 함수의 인자로 전달할 수 있다.
> - 함수의 반환값으로 전달할 수 있다.

- 객체지향적이고 선언형 스타일을 유지하기 위해서는 **데이터를 객체 안에 감추고 절대로 외부에 노출해서는 안된다.**
- 데이터를 **발가벗겨진** 상태로 두어서는 안된다. 항상 제대로 된 옷을 입혀줘야 한다!.
- **OOP에서는 어떤 희생을 치르더 라도 절차적인 프로그래밍 스타일을 피해야 한다.**

2. 좋은 의도, 나쁜 결과

- java에서 **getter/setter는 클래스를 자료구조로 바꾸기 위해 도입**되었다.
- getter/setter 는 **캡슐화 원칙을 위반**하기 위해 설계되었다.
- getter/setter 는 **행동이 아닌 데이터를 표현할 뿐이다.**
- getter와 setter를 사용하면 **OOP의 캡슐화 원칙을 손쉽게 위반할 수 있다는 점이다.**
- getter/setter 는 **행동이 아닌 데이터를 표현할 뿐이다.**

3. 접두사에 관한 모든 것

- **객체는 대화를 원하지 않는다**. 그저 우리가 **어떤 데이터를 객체 안에 넣어주 거나 다시 꺼내주기를 원할 뿐이다.**
- get / set 이라는 접두사는 진짜 객체가아니고, 어떤 존중도 받을 가치가 없는 자료구조라는 사실을 명확하게 전달한다.
- **dollars()는 객체를 데이터의 저장소로 취급하지 않고, 객체를 존중한다.**
- **getter와 setter가 OOP에서 끔찍한 안티 패턴**이다. 메서드의 이름을 절대 이 런 방식으로 지어서는 안된다.

```
# good : 적절한 메서드명 (객체에게 물음, 얼마나 많은 달러가 필요하세요?)
class Cash{
  private final int value;
  public int dollars(){
      return this.value;
  }
}

# bad : 적절하지 않는 메서드명 (데이터 중에 dollars를 찾은 후 반환하세요.)
class Cash{
  private final int value;
  public int getDollars(){
      return this.value;
  }
}
```

6. 부 actor 밖에서는 new를 사용하지 마세요

- 주 생성자를 사용할 경우, 객체와 협력하는 모든 의존성을 우리 스스로 완전히 제어할 수 있게된다.
- **new를 합법적으로 사용할 수 있는 유일한 곳은 부 ctor일 뿐이다.**
- Cash 클래스는 Exchange 클래스에 직접 연결되어 있기 때문에, **의존성을 끊기 위해서는 Cash 클래스 의 내부 코드를 변경할 수 밖에 없다.**

[의존성]

```
# bad:하드코딩된 의존성
- Cash가 NYSE 서버와 통신하지 않게 만들 수 없다.
- 이 문제의 근본 원인은 new 연산자 이다.
class Cash {
    private final int dollars;

    public int euro() {
        //뉴욕에 있는 서버와 통신하는 메서드
        return new Exchange().rate("USD", "EUR") * this.dollars;
    }
}
```

[의존성 제거]

```
# good:Cash 클래스는 더이상 Exchange 클래스에 의존하지 않으며,'협력'만 할 뿐이다.
    필요한 의존성 전체를 생성자를 통해 전달받아,부 생성자 여러개를 추가할 수도 있는 형태이다.
class Cash {

    private final int dollars;
    private final Exchange exchange;

    Cash() { //부 생성자
        this(0);
    }

    Cash(int value) { //부 생성자
        this(value, new NYSE());
    }

    Cash(int value, Exchange exch) { //주 생성자
        this.dollars = value;
        this.exchange = exch;
    }

    public int euro() {
        //뉴욕에 있는 서버와 통신하는 메서드
        return this.exchange.rate("USD", "EUR") * this.dollars;
    }
}
```

- 주 ctor을 사용할 경우에는 객체와 협력하는 모든 의존성을 우리 스스로 완전히 제어할 수 있다.
- 부 ctor을 제외한 어떤 곳에서도 new를 사용하지 마세요
    - 부 ctor을 제외한 어떤 곳에서도 new를 사용할 수 없도록 금지시킨다면, **객체들은 상호간에 충분히 분리되고 테스트 용이성과 유지보수성을 크게 향상시킬 수 있다**.

[의존성]

```
class Requests {
    private final Socket socket;
    public Requests(Socket skt) {
        this.socket = skt;
    }
    public Request next() {
        return new SimpleRequest(/* 소켓에서 데이터를 읽는다 */ );
    }
}
```

[의존성 제거]

- **new 연산자는 오직 부 ctor 내부에서만 사용한다.**
- 개선된 설계에서는 더 쉽게 **Requests 클래스의 설정을 변경**할 수 있으며 **하드코딩된 의존성도 존재하지 않는다.**

```
class Requests {
    private final Socket socket;
    private final Mapping<String, Request> mapping;
    public Requests(Socket skt) {
        this(skt, new Mapping<String, Request>() {
            @Override
            public Request map(String data) {
                return new SimpleRequest(data);
            }
        });
    }
    public Requests(Socket skt, Mapping<String, Request> mpg) {
        this.socket = skt;
        this.mapping = mpg;
    }
    public Request next() {
        return this.mapping.map( /* 소켓에서 데이터를 읽는다 */ );
    }
}
```

- 생성자 안에서 new를 사용하는 매순간마다 여러분이 뭔가 잘못하고 있다는 사실을 떠올려야 한다.
- **new를 합법적으로 사용할 수 있는 유일한 곳은 부 ctor뿐이다.**
- 의존성 주입과 제어 역전(inversion of control)에 관해 알아야 하는 전부이다.
- 부 ctor에서만 new를 사용해야 한다는 간단한 규칙을 불변 객체와 조합하면, **코드는 깔끔해지고 언제라도 의존성을 주입할 수 있게 된다.**

7. 인트로스펙션과 캐스팅을 피하세요

- **타입 인트로스펙션**(introspection)과 **캐스팅**(casting)을 사용하고 싶은 유혹에 빠지더라도 **절대로 사용해서는 안된다.**
    - 프로그래머는 이 연산 자들을 사용해서 런타임에 객체의 타입을 확인할 수 있다.
    - java의 instanceof 연산자와 Class.cast() 메서드 도 포함된다.
- 타입 인트로스펙션은 **리플렉션(reflection)** 이라는 더 포괄적인 용어로 불리는 기법중 하나이다.

    - 클래스, 메서드, 속성, 객체 등에 대한 정보를 동적으로 탐색하고 조작할 수 있다.
- 리플렉션은 강력하지만, 동시에 코드를 **유지보수하기 어렵게 만든다.**
    - 코드가 **런타임에** 다른 코드에 의해 수정된다는 사실을 항상 기억해야 한다면, 코드를 읽기가 매우 어려울 것이다.

[BAD]

```
public static <T> int size(Iterable<T> items) {
    if (items instanceof Collection) {
        return Collection.class.cast(items).size();
    }
    int size = 0;
    for (T item : items) {
        ++size;
    }
    return size;
}
```

- 이 접근방법은 **타입에 따라 객체를 차별하기 때문에 OOP의 기본 사상을 심각하게 훼손**한다.
    - items 객체의 타입에 따라(Iterable, Collection) 타입을 차별화 한다.
    - 요청을 어떤 식으로 처리할 지 객체가 결정할 수 있도록 하는 대신, 객체를 배재한 상태에서 결정을 내리고, 이를 바탕으로 좋은 객체와 나쁜 객체를 차별한다.
    - **우리는 차별하지 말고 객체가 누구건 상관없이 자신의 일을 할 수 있도록 해야 한다.**
- **런타임에 객체의 타입을 조사(introspect)하는 것**은 클래스 사이의 **결합도가 높아지기 때문에** 기술적인 관점에서도 좋지 않다.
    - size() 메서드는 Iterable과 Collection이라는 두 개의 인터페이스에 의존한다.
    - 의존하는 대상이 늘어날수록 결합도는 높아지기 때문에 유지보수성의 측면에서 좋지 않다.
- **소스 코드를 확인하지 않고서는 인자가 Collection일 경우 다르게 동작한다는 사실을 알 수 없다.**

[GOOD]

- 메서드 오버로딩(method overloading)을 통해 개선할 수 있다.

```
public <T> int size(Collection<T> items) {
  return items.size();
}
public <T> int size(Iterable<T> items) {
  int size = 0;
  for (T item : items) {
      ++size;
  }
  return size;
}
```

- 사전에 약속하지 않았던 새로운 계약을 따르도록 강제하는 **클래스 캐스팅(class casting)에도** 지금 까지 이야기한 모든 내용이 **동일하게 적용**된다.

    - `return Collection.class.cast(items).size();`, `return ((Collection) items).size();`

```
Object obj = "Hello, World!";
String str = (String) obj; // 성공적인 캐스팅

Object obj = 42; // Integer 타입
String str = (String) obj; // 실패한 캐스팅, ClassCastException 발생
```

- 아래와 같은 코드는 만약 당신이 컴퓨터 전문가이기도 하다면, 이 프린터를 고쳐 주세요라고 말하는 것이다.
    - **방문한 객체에 대한 기대(expectation)를 문서에 명시적으로 기록하지 않은 채로 외부에 노출한 것**이다.
    - 클라이언트와 객체 사이의 불명확하고, 은폐되고, 암시적인 관계는 유지보수성에 심각한 영향을 끼친다.

```
if(items instanceof Collection) {
    return ((Collection) items).size();
}
```

- **instanceof 연산자**를 사용하거나 **클래스를 캐스팅하는 일은 안티패턴**이기 때문에 사용해서는 안된다.

> 리플렉션을 사용하는 코드는 컴파일 타임에 정확한 타입 정보를 알 수 없기 때문에,
> JVM의 최적화 기능을 충분히 활용하지 못합니다. 이는 **JIT(Just-In-Time) 컴파일러가 성능 최적화를 수행하는 데 제한을 받는다.**

Chapter 4.Retirement

- 이번 장에는 메서드 결과로 **반환되는 NULL, 예외 처리, 리소스 획득에 관해 살펴본다.**

1. 절대 NULL을 반환하지 마세요

```
public String title() {
    if (/* title이 없다면 */) {
        return null;
    }
    return "Elegant Objects";
}

String title = x.title();
print(title.length());
```

- **title() 메서드가 반환하는 객체는 신뢰할 수 없다.**
- 이 객체는 장애를 안고 있기 때문에 특별하게 대우할 필요가 있습니다.
- title.length()를 호출할 때마다 항상 **NullPointerException 예외가 던져질지 모른다는 사실에 불안**할 수 밖에 없다.
- **객체에 대한 신뢰(trust)가 무너졌다.**

```
String title = x.title();
if (title == null) {
    print("Can’t print; it’s not a title.");
    return;
}
print(title.length());
```

- 객체라는 사상에는 **우리가 신뢰하는 엔티티라는 개념이 담겨져 있다.**
- 객체는 우리의 의도에 관해 전혀 알지 못하는 **데이터 조각이 아니다.**
- 객체는 **자신만의 생명주기, 자신만의 행동, 자신만의 상태를 가지는 살아있는 유기체**이다.
- 우리는 객체를 신뢰하기 때문에 객체와 완벽하게 동일한 의미를 가지는 **변수(t) 역시 신뢰**한다.
    - String t = x.title();
    - 신뢰라는 말에는 **객체가 자신의 행동을 전적으로 책임지고(responsible) 우리가 어떤 식으로든 간섭하지 않는다** 는 의미이다.
- 객체는 **자신이 맡은 일을 수행하는 방법을 스스로 결정**한다.
- 객체가 이름을 출력하는 대신 예외를 던지고 싶다면, 그것도 괜찮다.
    - 하지만 다음 코드에서 볼 수 있는 것처럼 객체 에게 아무런 말도 하지 않은 채 **우리 마음대로 예외를 던져서는 안된다.**
    - 이런 방식은 완전히 잘못됐고 **무례하다.**

```
if (title == null) {
  print("Can’t print; it’s not a title.");
  return;
}
```

- **반환값을 검사하는 방식**은 **애플리케이션에 대한 신뢰가 부족하다는 명백한 신호**이다.
- **NULL을 사용하면 전체 소프트웨어에 대한 신뢰가 크게 손상**되고, 소프트웨어가 **유지보수 불가능할 정 도로 엉망**이 되고 말 것이다.
    - **결과가 NULL이 아닌지를 중복으로 체크**해야한다.
- **코드를 읽을 때** 신뢰할 수 있는 메서드 호출이 어떤 것이고 NULL을 반환하는 메서드 호출이 어떤 것인지 파악하는데 **많은 시간이 걸린다.**
- 우리에게는 **신뢰가 필요하지만 NULL은 우리에게서 신뢰를 앗아간다.**
    - 작업 속도 저하
    - 모든 반환값 체크
    - 코드 유지보수성 저하
- **NULL을 반환하는 방식은 잘못됐으며 무례하다.**
- 왜 그렇게 NULL을 반환하는 코드가 흔하고 널리 사용되는 것일까?
    - 필자는 JDK를 설계하던 시점에 **빠르게 실패하기 원칙(fail fast principle)을 몰랐기 때문**이라고 추측한다.
    - 필자는 **NULL을 반환해서 사용자가 필요에 따라 예외를 던질 수 있도록 하는 방식이 더 좋은 설계라고 생각**했다고 추측한다.

1. 빠르게 실패하기 vs 안전하게 실패하기

- 소프트웨어 견고성(software robustness)과 실패 탄력회복성(failure resilience) 관련해서 **상반되는 두 가지 철학이 존재**한다.
- 빠르게 실패하기(fail fast)
  - 필자는 **강하게 찬성하는 입장.**
  - **문제가 발생하면 곧바로 실행을 중단 하고 최대한 빨리 예외를 던진다.**
  - 만약 소프트웨어가 부서지기 쉽고 **모든 단일 제어 지점(single control point)에서 중단되록 설계됐다면**, 단위 테스트 에서 **실패 상황을 손쉽게 재현할 수 있을 것이다.**
  - 프로덕션 환경에서 실패한다고 해도, **모든 실패 지점이 명확하고 훌륭하게 문서화되어 있기 때문에 상황을 재생하는 테스트를 쉽게 추가할 수 있다.**
  - **실패를 감추는 대신 강조한다. 실패를 눈에 잘 띄게 만들고 추적하기 쉽게 만든다.**

```
void list(File dir) {
  File[] files = dir.listFiles();
  if (files == null) {
     throw new IOException("Directory is absent.");
  }
  for (File file : files) {
   System.out.println(file.getName());
  }
}
```

- 안전하게 실패하기(fail safe)
  - 필자는 **강하게 반대하는 입장.**
  - **안전하게 실패하기**는 버그, 입출력 문제, 메모리 오버플로우 등이 발생한 상황에서도 **소프트웨어가 계속 실행될 수 있도록 최대한 많은 노력을 기울일 것을 권장**한다.
  - 어떤 상황이 닥치더라도 **소프트웨어는 생존하기 위해 노력**한다.
  - IOException을 던지는 대신, **누군가 이 상황을 처리할 수 있도록 NULL을 반환**한다.

```
void list(File dir) {
   for (File file : dir.listFiles()) {
      System.out.println(file.getName());
   }
}
```

> **빠르게 실패하기**
>
> - 에러를 발견한 즉시 보고하는 경우에만 안전성과 견고함을 얻을 수 있다.
> - 더 빠르게 실패하고, 결과적으로 전체적인 품질이 향상된다. 반대로 더 오래 숨 길수록 문제는 더 커진다.
> - 버그는 분명히 존재한다. 이 사실을 감춤으로써, 스스로에게 죄를 짓고 있는 것이다.
> - 상처를 드러내어 치료하는 대신, **상처를 숨기고 모든 일이 순조롭게 진행될 것이라고 환자에게 거짓말**을 하고 있는 것이다.
> - **문제를 더 빨리 발견할수록 문제를 수정하는 시간 역시 빨라진다.**
> - 많은 Java 메서드가 예외를 던지지 않고 NULL을 반환하는 것일까?
>   - 대부분 의 개발자들이 안전하게 실패하기 철학을 믿기 때문이 아닌가 싶다.
>   - **하나의 메서드가 아닌 전체 애플리케이션의 품질을 고려 한다면 빠르게 실패하기 원칙을 따르는 편이 좋다.**

2. NULL의 대안

- **실제 객체 대신 NULL을 반환하는 가장 흔한 경우이다.**
  - 안전하게 실패하기 철학과 매우 유사한 형태이다.

```
public User user(String name) {
  if (/* 데이터베이스에서 발견하지 못했다면 */) {
      return null;
  }
  return /* 데이터베이스로부터 */;
}
```

- NULL을 반환하는 방식의 대안으로는 어떤 것이 있을까?
  - **첫 번째 방법은 메서드를 두 개로 나누는 것이다.**
    - exists() 메서드는 객체의 존재를 확인
    - user(name) 메서드는 예외를 던진다.
    - 요청을 두 번 전송하기 때문에 비효율적이라는 단점이 있다.

```
public boolean exists(String name) {
  if (/* 데이터베이스에서 발견하지 못했다면 */) {
    return false;
  }
  return true;
}
public User user(String name) {
  return /* 데이터베이스로부터 */;
}
```

  - **두 번째 방법은 NULL을 반환하거나 예외를 던지는 대신 객체 컬렉션을 반환하는 것이다.**
    - 기술적으로 이 방법은 **NULL과 크게 다르지는 않지만 더 깔끔하다.**

```
public Collection<User> users(String name) {
  if (/* 데이터베이스에서 발견하지 못했다면 */) {
     return new ArrayList(0);
  }
  return Collections.singleton(/* 데이터베이스로부터 */);
}
```

  - **세 번째 방법은 널 객체(null object) 디자인 패턴이다.**
    - 겉으로 보기에는 원래의 객체처럼 보이지만 **실제로는 다르게 행동하는 객체를 반환한다.**
    - **널 객체는 일부 작업은 정상적으로 처리하지만, 나머지 작업은 처리하지 않는다.**
    - 객체지향적인 사고방식과도 잘 어울리지만, **제한된 상황에서만 사용 가능하다는 단점**이 있다.
    - 반환된 객체의 타입을 동일하게 유지해야 한다는 사실에도 주의해야 한다.

```
class NullUser implements User {
  private final String label;
  NullUser(String name) {
    this.label = name;
  }
  @Override
  public String name() {
    return this.label;
  }
  @Override
  public void raise(Cash salary) {
     throw new IllegalStateException("제 봉급을 인상할 수 없습니다. 저는
스텁(stub)입니다.");
  }
}
```

- **Optional**
  - (필자)의미론적으로 부정확하기 때문에 OOP와 대립한다고 생각하며 사용을 권하지 않는다.
  - 실제로 반환하는 객체는 사용자가 아니라 사용자를 포함하는 일종의 봉투이다.
  - 이방법은 오해의 여지가 있으며 객체지향적인 사고방식과도 거리가 멀다.
  - NULL 참조와 매 우 비슷하다.

> **요약**
>
> - **절대로 NULL을 반환하지 마세요.**
> - OOP에서 NULL을 사용하는 상황에 대해서는 **어떤 변명도 있을 수 없다.**
> - 찾지 못한 뭔가를 반환할 필요가 있다면 NULL 대신 **예외를 던지거나**, **컬렉션을 반환**하거나, **널 객체를 반환**해야 한다.

2. 체크 예외(checked exception)만 던지세요

- **체크 예외**(checked exception)와 언체크 예외(unchecked exception)에 관해 이야기 한다.
  - java는 두 종류의 예외를 모두 제공한다.
- 언체크 예외를 사용하는 것은 실수이며, 모든 **예외는 체크 예외**여야 한다.
  - 다양한 예외 타입을 만드는 것도 좋지 않은 생각이다.
- 대부분의 언어들에서 언체크 예외만 사용할 수 있다.

[체크 예외]

- 무슨 일이 있어도 **content()를 호출하는 쪽에서 IOException 예외를 잡아야 한다는 것을 의미한다.**
- **예외를 저 위로 띄워 올린다.**
- 관리를 할 때 흔히 그러는 것처럼, **문제를 더 높은 레벨로 확대시킨다.**
- IOException은 catch 구문을 이용해서 반드시 잡아야 하기 때문에 **체크(checked) 예외에 속한다.**
- **체크 예외가 항상 가시적인(visible)이다.**
  - length() 메서드를 이용하는 동안에는 content()라는 해롭고 안전하지 않은 메서드를 다루고 있다는 사실을 상기 시킨다.

```
public byte[] content(File file) throws IOException {
  byte[] array = new byte[1000];
  new FileInputStream(file).read(array);
  return array;
}

//호출부
public int length(File file){
  try{
    return content(file).length();
  } catch(IOExpceion e){
    //여기서 예외처리 또는 상위레벨로 전달
  }
}

public int length(File file) throws IOException{
    return content(file).length();
}
```

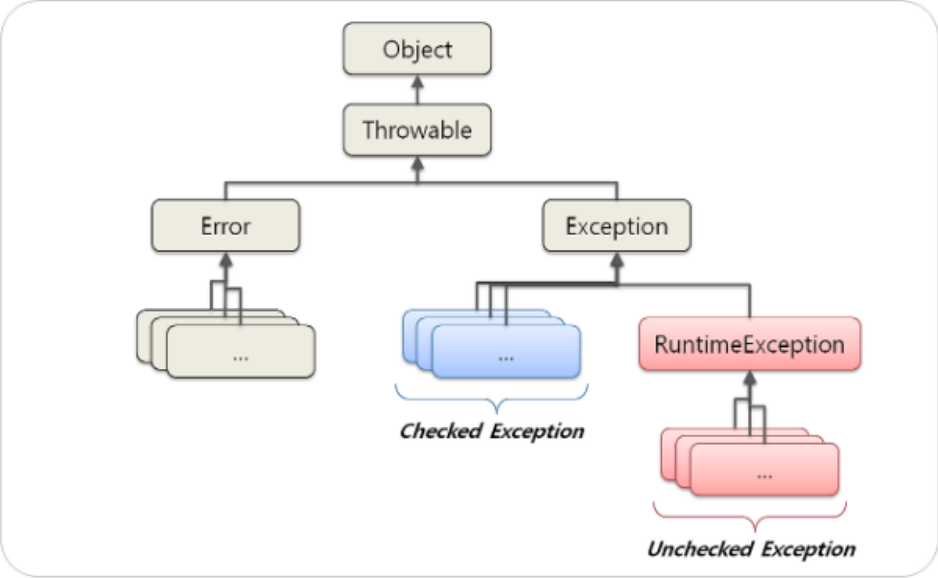
[언체크 예외]

- **언체크 예외는 무시할 수 있으며, 예외를 잡지 않아도 무방하다.**
- 일단 **언체크 예외를 던지면, 누군가 예외를 잡기 전까지는 자동으로 상위로 전파**된다.
- **호출부가 어떤 예외가 던져질지 예상할 수가 없다.**
    - **IllegalArgumentException에 대한 정보는 숨겨져 있다.**

```
public int length(File file) throws IOException{
  if(!file.exists()){
      throw new IllegalArgumentException("오류야");
  }
  return content(file).length();
}
```

- **언체크 예외의 경우 예외의 타입을 선언하지 않아도 무방한 반면에 체크 예외는 항상 예외의 타입을 공개해야 한다.**
    - RuntimeException 클래스를 상속받는 예외 클래스들은 복구 가능성이 없는 예외들이므로 컴파일러가 **예외처리를 강제하지 않는다.**
        - ex) NullPointerException, IllegalArgumentException

- 언체크 예외는 Error와 마찬가지로 **에러를 처리하지 않아도 컴파일 에러가 발생하지 않는다**



1. 꼭 필요한 경우가 아니라면 예외를 잡지마세요

- **상위로 예외를 전파하는 방법이 훨씬 좋다. 가능하면 예외를 더 높은 레벨로 전파하자.**
- **catch 문에는 납득할 수 있는 이유**가 있어야한다.
- 이상적인 어플리케이션은 **각 진입점 별로 오직 하나의 catch 문만 존재**해야 한다.
- **예외를 잡는것은 안전하게 실패하기 방법(비추천) 과 동일하다.**
  - 파일 시스템에 어떤 일이 발생하더라도 메서드는 종료되지 않는다.
    - 흐름 제어를 위한 예외 사용(using exceptions for flow control)

```
public int length(File file) {
    try {
        return content(file).length();
    } catch (IOException ex) {
        return 0;
    }
}
```

2. 항상 예외를 체이닝하세요

- **예외 되던지기(rethrowing)에 관해 살펴보자**
- **예외 체이닝(exception chaining)은 훌륭한 프랙티스 이다.**
- **원래의 문제를 새로운 문제로 감싸서 함께 상위로 던진다.**
- **여기에서 핵심은 문제를 발생시켰던 낮은 수준의 근본 원인(root cause)을 소프트웨어의 더 높은 수준으로 이동시켰다는 것이다.**

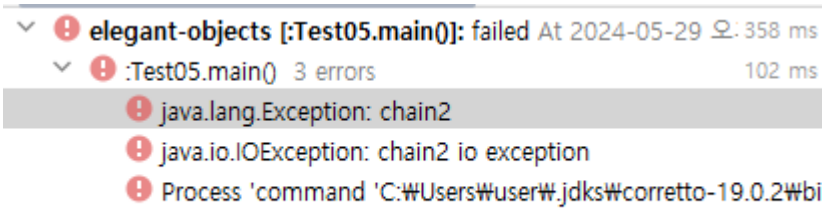
```
# GOOD : 문제를 발생시킨 근본 Exception ex 를 상위레벨로 던진다.
public int length(File file) throws Exception{
```

```
    try {
        return content(file).length();
    } catch (IOException ex){
        //원래의 문제를 새로운 문제로 감싸서 함께 상위로 던진다.
        throw new Exception("길이를 계산할 수 없다.",ex);
    }
  }

  # BAD : 문제를 발생시킨 근본 원인에 대한 가치있는 정보가 손실된다.
  public int length(File file) throws Exception{
    try {
        return content(file).length();
    } catch (IOException ex){
        //상위로 근본 에러 원인을 던지지 않는다.
        throw new Exception("길이를 계산할 수 없다.");
    }
  }
```

- **항상 예외를 체이닝하고 절대로 원래 예외를 무시하면 안된다.**
- **모든 예외를 잡아 체이닝한 후, 다시 던지는것이 예외를 처리할 수 있는 최선의 방법이다.**
- 예외 체이닝은 왜 필요한 것일까?
    - **예외 체이닝은 의미론적으로 문제와 관련된 문맥을 풍부하게 만들기 위해 필요하다.**
- 이상적으로 각각의 메서드는 발생할 수 있는 모든 예외를 잡은 후, 예외를 체이닝해서 다시 던져야 한다.
    - **한번 더 강조하지만 모든 예외를 잡아 체이닝한 후, 즉시 다시 던져야 한다.**

```
public class Member {
    private String name;

    public Member(String name) throws Exception {

        if(1==1){
            chain1();
        }
        this.name = name;
    }

    private void chain1() throws Exception {

        try {
            chain2();
        }catch (Exception e){
            throw new Exception("chain1", e);
        }
    }

    private void chain2() throws Exception {
        try {
            throw new IOException("chain2 io exception");
        }catch (IOException e){
            throw new Exception("chain2",e);
        }
```

```
    }
}
```



3. 단 한번만 복구하세요

- **필요시, 가장 최상위 수준에서 오직 한번만 복구해라**
- 만약 **빠르게 실패하기 진영에 속한다면 예외 후 복구를 적용할 수 없다.**
    - 예외 후 복구는 `흐름 제어를 위한 예외 사용(using exceptions for flow control)`으로 알려진 **안티패턴의 또 다른 이름일뿐이다.**

```
int age;
try {
    age = Integer, parseInt(text);
} catch (NumberFormatException ex) {
    // 여기에서 발생한 예외를 복구한다
    age = -1;
}
```

- 하지만 무조건 예외를 잡아서는 안된다는 주장은 전적으로 옳지 않다. **딱 한번은 복구해야 한다.**
    - **복구지점은 사용자가 애플리케이션과 커뮤니케이션하는 진입점(entry point)을 의미한다.**
    - **이곳이 복구하기에 적합한 유일한 장소이다.**

```
public class App {
    public static void main(String... args) {
      try {
          System.out.println(new App().run());
      } catch (Exception ex) {
      System.err.println("죄송하지만 문제가 발생했습니다:" +
ex.getLocalizedMessage());
      }
    }
}
```

> **요약**
>
> - **항상 예외를 잡고, 체이닝하고, 다시 던지세 요. 가장 최상위 수준에서 오직 한번만 복구하세요. 이게 전부이다.**

4. 관점-지향 프로그래밍을 사용하세요

- 예외를 꼭 복구해야하는경우, **AOP를 통한 실패재시도는 OOP의 코드를 깔끔한 상태를 유지**할 수 있다는 현실적이며, 실용적인 예이다.
- **AOP는 단순하면서도 강력한 프로그래밍 패러다임으로 OOP와도 궁합이 잘 맞는다.**

```
public String conetnet() throws IOException{
    int attempt = 0;
    while(true){
        try{
          return http();
        }catch (IOException ex){
            if(attempt >=2) throw ex;
        }
    }
}
//Java 6에서 AOP를 사용한다면 재시도 메커니즘을 다음과 같이 구현할 수 있다.
@RetryOnFailure(attempts =3)
public String content() throws IOException{
  return http();
}
```

5 하나의 예외 타입만으로도 충분합니다.

- 사실 단 한번만 복구한다면 어떤 예외라도 담을 수 있는 예외 객체만 있으면 된다.
  - 절대 복구하지 않기 / 항상 체이닝하기
- **잡은 예외의 실제 타입에 대해서는 신경 쓸 필요가 없습니다. 어차피 다시 던질 것이기 때문이다.**
- **최상위에서, 어떤 예외라도 담을 수 있는 예외 객체만 있으면 해당 객체로 최상위 레벨에서 복구하거나 또는 오류를 처리할 수 있다.**

3. final이나 abstact이거나

- **상속(inheritance)을 올바르게 사용하는 방법에대해 설명**한다.
- 사실 우리가 원하는 것은 상 속을 완전히 제거하는 것이 아니라 **올바르게 사용하는 것**이다.
- **상속이 해로운 이유**와 **상속으로 인해 발생하는 문제**를 방지하기 위해 무엇을 할 수 있는지 확인해보자.
  - 상속에 반대하는 가장 강력한 주장은 **상속이 객체들의 관계를 너무 복잡하게 만든다**는 것이다.
  - 근본적인 원인은 상속 그 자체가 아니고, **문제를 일으키는 주범은 가상 메서드(virtual method)이다.**

[상속]

- Document content 의 length 길이를 원하는 거였지만, EncryptedDocument의 length를 반환해 **오류를 야기할 수 있다.**
- **content length 가 객체별로 어떤행동을 하는지 명확하게 알 수가 없는 코드이다.**
- 자식이 부모의 유산에 접근하는 일반적인 상속과 달리, **메서드 오버라이딩은 부모가 자식의 코드에 접근하는 것을 가능하게 한다.**
  - 상속이 OOP를 지탱하는 편리한 도구에서 **유지보수성을 해치는 골치덩어리로 추락하는 곳이 바로 이 지점이다.**
  - **복잡성이 상승하고, 코드를 읽고 이해하기가 매우 어려워진다.**

```
class Document {
  public int length(){
      return this.content().length();
  }
  public byte[] content() {
      //문서를 읽어 바이트 배열로 로드한다.
  }
}

class EncryptedDocument extends Document {
  @Override
  public byte[] content(){
      //문서를 읽어, 복호화 하여 반환
  }
}

Document document = new Document("1");
EncryptedDocument encryptedDocument = new EncryptedDocument("2");
Document parentDocument = new EncryptedDocument("3");
```

- 클래스와 메서드를 `final`이나 `abstract` **둘 중 하나로만 제한한다면 문제가 발생할 수 있는 가능성을 없앨 수 있다.**
- 클래스 앞 final **수정자** 는 이 클래스 안의 **어떤 메서드도 자식 클래스에서 오버라이딩 할 수 없다는 사실을 컴파일러에 알려준다.**

[final]

```
public class FinalDocument {
    private String content;
    public FinalDocument(String content) {
        this.content = content;
    }
    // final 메서드는 오버라이딩될 수 없다.
    public final int length() {
        return this.content.length();
    }
    // final 메서드는 오버라이딩될 수 없다.
    public final byte[] content() {
        return content.getBytes();
    }
}
```

```java
no usages  new *
public class FinalEncryptedDocument extends FinalDocument{

    no usages  new *
    public FinalEncryptedDocument(String content) {
        super(content);
    }

    no usages  new *
    @Override
    public byte[] content() {
        // 문서를 읽어, 복호화하여 반환
        return new byte[0];
    }
}
```

[final 2]

- final1을 해결하기 위해 interface 를 추가해야한다.
- DefaultDocument와 EncryptedDocument **모두 final이기 때문에 확장이 불가능하다.**

```java
interface Document {
  int length();
  byte[] content();
}

final class DefaultDocument implements Document {
  @Override
  public int length() {...}
  @Override
  public byte[] content() {...}
}

final class EncryptedDocument implements Document {
  private final Document plain;
  EncryptedDocument(Document doc){
      this.plain = doc;
  }

  @Override
  public int length(){
      return this.plain.length();
  }

  @Override
  public byte[] content() {
    byte[] raw = this.plain.content();
```

```
    return /* 원래 내용을 복호화*/
  }
}
```

[abstract]

- 스스로 행동할 수 없기 때문에 누군가의 도움이 필요하며 일부 요소가 누락되어 있다.
- 기술적인 관점에서 abstract 클래스 는 아직 클래스가 아니다.

```
public abstract class AbstractDocument {
    private String content;

    public AbstractDocument(String content) {
        this.content = content;
    }

    // abstract 메서드는 하위 클래스에서 반드시 구현해야 합니다.
    public abstract byte[] content();

    //공통 메서드
    public int length() {
        return this.content.length();
    }
}

public class AbstractEncryptedDocument extends AbstractDocument{
  public AbstractEncryptedDocument(String content) {
    super(content);
  }

  //무조건 구현!
  @Override
  public byte[] content() {
    return new byte[0];
  }
}
```

- **상속이 적절한 경우는 언제일까?**
  - **클래스의 행동을 확장(extend)하지 않고 정제 (refine)할 때이다.**
  - 확장이란 새로운 행동을 추가해서 기존의 행동을 부분적으로 보완하는 일을 의미한다.
  - 정제란 부분적으로 불완전한 행동을 완전하게 만드는 일을 의미한다.
- OOP에서는 어떤 것도 **확장 할 수 없어야** 한다, 클래스 확장은 곧 **침범**(intrusion)을 의미한다.
- abstract 추상클래스 올바른 설계
  - 추상클래스를 사용함으로써, 두 클래스 모두 length() 메서드가 자신들의 메서드 사용 방법을 명확하게 알고 있다.

```
abstract class Document {
  public abstract byte[] content();
  public final int length() {
      return this.content().length;
  }
}

final class DefaultDocument extends Document {
  @Override
  public byte[] content(){
      //내용을 로드한다.
  }
}

final class EncryptedDocument extends Document {
  @Override
  public byte[] content(){
      //내용을 로드한다.
  }
}
```

4. RAII를 사용하세요

- RAII (Resource Acquisition Is Initialization)
  - **리소스 획득이 초기화**라는 개념, 객체가 살아있는 동안에만 리소스를 확보하는 것
- 가비지 컬렉션을 이용해 객체를 제거하는 java에서는 사라진 개념이다.
- Java7에서도 RAII와 유사한 처리를 할 수 있는 기능이 있다.

```
- Text 클래스가 Closable 인터페이스를 구현하도록 되어 있다.
try(Text t = new Text("/tmp/test.txt")){
  t.content();
}
```

- 파일, 스트림, 데이터베이스 컨넥션 등 실제 리소스를 사용하는 모든곳에서 RAII를 사용할 것을 적극 추천한다.
  - AutoClosable 사용