Refactoring

객체의 생성 과 삭제

- 생성자 대신 정적 팩토리 메서드를 사용할 수 없는지 생각해 보라
  - ✓ 클래스를 통해 객체를 만드는 일반적인 방법은 public으로 선언된 생성자 (constructor)를 이용하는 것이지만 프로그래머가 반드시 알고 있 어야 하는 방법이 하나 더 있는데 클래스에 public으로 선언된 정적 팩토리 메서드(static factory method)를 추가하는 것
  - ✓ 클래스를 정의할 때 생성자 대신(아니면 생성자 와는 별도로) 정적 팩토리 메서드를 제공할 수 있음

객체의 생성 과 삭제

- ❖ 생성자 대신 정적 팩토리 메서드를 사용할 수 없는지 생각
  - ✓ public으로 선언된 생성자 대신 정적 팩토리 메서드를 제공하는 방법의 장단점
    - 생성자 와는 달리 정적 팩터리 메서드에는 이름(name)이 있다 는 것
      - ◆ 생성자에 전달되는 인자(parameter)들은 어떤 객체가 생성되는지를 설명하지 못하지만 정적 팩토리 메서드는 이름을 잘 짓기만 한다면 사용하기도 쉽고 클라이언트 코드의 가독성(readability)도 높아지는데 소수(prime)일 가능성이 높은 Biginteger 객체를 생성하는 생성자 BigInteger(int, int, Random)는 Biginteger.probablePrime과 같은 이름의 정적 팩토리 메서드로 표현했으면 더 이해하기 쉬웠을 것
    - 생성자 와는 달리 호출할 때마다 새로운 객체를 생성할 필요는 없다는 것
      - ◆ 변경 불가능 클래스라면 이미 만들어 둔 객체를 활용할 수도 있고 만든 객체를 캐시(cache) 해놓고 재사용하여 같은 객체가 불필요하게 거듭 생성되는 일을 피할 수도 있음
    - 생성자 와는 달리 반환값 자료형의 하위 자료형 객체를 반환할 수 있다는 것
    - 형인자 자료형(parameterized type) 객체를 만들 때 편리
    - 정적 팩터리 메서드만 있는 클래스를 만들면 생기는 가장 큰 문제는 public이나 protected로 선언된 생성자가 없으므로 하위 클래스를 만들 수 없음
    - 정적 팩터리 메서드가 다른 정적 메서드와 확연히 구분되지 않음

객체의 생성 과 삭제

- 생성자 인자가 많을 때는 Builder 패턴 적용을 고려
  - ✓ 점충적 생성자 패턴은 잘 동작하지만 인자 수가 늘어나면 클라이언트 코드를 작성하기가 어려워지고 무엇보다 읽기 어려운 코드가 됨
  - ✓ 생성자에 전달되는 인자 수가 늘어날 때 생각할 수 있는 대안 중 하나는 Java Bean 패턴
    - 1회의 함수 호출로 객체 생성을 끝낼 수 없으므로, 객체 일관성(consistency)이 일시적으로 깨질 수 있음
    - 자바빈 패턴으로는 변경 불가능(immutable) 클래스를 만들 수 없음
  - ✓ 자바빈 패턴에 점층적 생성자 패턴의 안정성을 결합한 것이 빌더 패턴

객체의 생성 과 삭제

- 생성자 인자가 많을 때는 Builder 패턴 적용을 고려
  - ✓ 빌더 패턴을 적용한 DTO 클래스

```
public class NutritionFacts {
    private final int servingSize;
    private final int servings;
    private final int calories;
    private final int fat;
    private final int sodium;
    private final int carbohydrate;
```

객체의 생성 과 삭제

- 생성자 인자가 많을 때는 Builder 패턴 적용을 고려
  - ✓ 빌더 패턴을 적용한 DTO 클래스

```
public static class Builder {
        // 필수 인자
        private final int servingSize;
        private final int servings;

        // 선택적 인자 - 기본값으로 초기화
        private int calories = 0;
        private int fat = 0;
        private int carbohydrate = 0;
        private int sodium = 0;

        public Builder(int servingSize, int servings) {
                this.servingSize = servingSize;
                this.servings = servings;
        }

        public Builder calories(int val){
                calories = val;
                return this;
        }
```

객체의 생성 과 삭제

- 생성자 인자가 많을 때는 Builder 패턴 적용을 고려
  - ✓ 빌더 패턴을 적용한 DTO 클래스

```
        public Builder fat(int val){
                fat = val;
                return this;
        }

        public Builder carbohydrate(int val){
                carbohydrate = val;
                return this;
        }

        public Builder sodium(int val){
                sodium = val;
                return this;
        }

        public NutritionFacts build() {
                return new NutritionFacts(this);
        }
}
```

객체의 생성 과 삭제

- 생성자 인자가 많을 때는 Builder 패턴 적용을 고려
  - ✓ 빌더 패턴을 적용한 DTO 클래스

```
        private NutritionFacts(Builder builder) {
                servingSize = builder.servingSize;
                servings = builder.servings;

                calories = builder.calories;
                fat = builder.fat;
                sodium = builder.sodium;
                carbohydrate = builder.carbohydrate;
        }
    }
```

객체의 생성 과 삭제

- 생성자 인자가 많을 때는 Builder 패턴 적용을 고려
  - ✓ 인스턴스 생성

```
NutritionFacts cocaCola = new NutritionFacts.Builder(240, 8)
                                            .calories(100)
                                            .sodium(35)
                                            .carbohydrate(27)
                                            .build();
System.out.println(cocaCola);
```

객체의 생성 과 삭제

- 객체 생성을 막을 때는 private 생성자를 사용
  - ✓ 때로는 정적 메서드나 필드만 모은 클래스를 만들고 싶을 때가 있는데 이런 클래스들은 악명이 높은데 객체 지향적으로 생각하지 않으려는 사람들이 남용하는 경향이 있기 때문인데 하지만 이런 클래스들도 분명 필요할 때가 있음
  - ✓ 자바의 기본 자료형 값(primitive value) 또는 배열에 적용되는 메서드를 한군데 모아둘 때 유용한데 java.lang.Math나 java.util.Arrays가 좋은 예
  - ✓ 특정 인터페이스를 구현하는 객체를 만드는 팩토리 메서드 등의 정적 메서드를 모아놓을 때도 사용할 수 있는데 java.util.Collections는 그 좋은 예
  - ✓ final 클래스에 적용할 메서드들을 모아놓을 때도 활용할 수 있는데 클래스를 계승하여 메서드를 추가할 수 없는데 그런 유틸리티 클래스(utility class)들은 객체를 만들 목적의 클래스가 아니므로 객체를 만들면 오히려 이상한데 생성자를 생략하면 컴파일러는 자동으로 인자 없는 public 기본 생성자(default constructor) 만들어 버림
  - ✓ 사용자 는 이 생성자를 일반 생성자와 구별할 수 없기 때문에 원래 의도와는 달리 객체 생성이 가능한 클래스가 공개 API에 포함되는 일도 드물지 않음
  - ✓ 객체를 만들 수 없도록 하려고 클래스를 abstract로 선언해 봤자 소용없는데 하위 클래스를 정의하는 순간 객체 생성이 가능해지기 때문이며 게다가 abstract 클래스니까 계승해서 사용하는 것이 맞다고 착각하는 사용자도 있을 수 있기 때문
  - ✓ 기본 생성자는 클래스에 생성자가 없을 때 만들어지니까 private 생성자를 클래스에 넣어서 객체 생성을 방지

객체의 생성 과 삭제

- Singleton
  - ✓ 인스턴스를 1개만 만들 수 있는 클래스 패턴
  - ✓ 클래스를 싱 글턴으로 만들면 클라이언트를 테스트하기가 어려워질 수가 있는데 싱글턴이 어떤 인터페이스를 구현하는 것이 아니면 가짜 구현으로 대체할 수 없기 때문
  - ✓ Singleton을 구현하는 방법
    - public final 필드를 이용하는 방법

```
public class Singleton {
    private Singleton() {}

    public static final Singleton INSTANCE = new Singleton();
}
```

    - 정적 팩토리 메소드를 이용하는 방법

```
public class Singleton {
    private Singleton() {}

    private static final Singleton INSTANCE = new Singleton();


    public static Singleton getInstance() {
       return INSTANCE;
    }
}
```

객체의 생성 과 삭제

- ❖ Singleton
  - ✓ Singleton을 구현하는 방법
    - ● Enum을 이용하는 방법

```
public enum Singleton {
    INSTANCE;

    public void method() {
      System.out.println("Singleton");
    }
}
```

객체의 생성 과 삭제

- 불필요한 객체는 생성하지 않도록 하기
  - ✓ 기능적으로 동일한 객체는 필요할 때마다 만드는 것보다 재사용하는 편이 나음
  - ✓ 객체를 재사용하는 프로그램은 더 빠르고 더 우아
  - ✓ 변경 불가능(immutable) 객체는 언제나 재사용할 수 있음
  - ✓ 극단적인 예

    String s = new String("stringette");

    - 위의 문장은 실행될 때마다 String 객체를 만드는데 쓸데없는 짓
    - String 생성자에 전달되는 stringette는 그 자체로 String 객체
    - 생성자 호출로 만들어지는 모든 객체와 기능적으로 같기 때문에 만일 위의 문장이 순환문(loop)이나 자주 호출되는 메서드 안에 있다면 수백만 개의 String 객체가 쓸데없이 만들어질 것
    - 아래처럼 수정

    String s = "stringette";

객체의 생성 과 삭제

- 불필요한 객체는 생성하지 않도록 하기
  - ✓ 불필요한 객체 생성

```
import java.util.Calendar;
import java.util.Date;
import java.util.TimeZone;
public class Person {
    private final Date birthDate;
    public Person(Date birthDate) {
            this.birthDate = birthDate;
    }
    // 다른 필드와 메서드, 생성자는 생략

    // 이렇게 하면 안 된다!
    public boolean isBabyBoomer() {
            // 생성 비용이 높은 객체를 쓸데없이 생성한다
            Calendar gmtCal = Calendar.getInstance(TimeZone.getTimeZone("GMT"));
            gmtCal.set(1946, Calendar.JANUARY, 1, 0, 0, 0);
            Date boomStart = gmtCal.getTime();
            gmtCal.set(1965, Calendar.JANUARY, 1, 0, 0, 0);
            Date boomEnd = gmtCal.getTime();
            return birthDate.compareTo(boomStart) >= 0 &&
    birthDate.compareTo(boomEnd) < 0;
    }
}
```

객체의 생성 과 삭제

- 불필요한 객체는 생성하지 않도록 하기
  - ✓ 정적 초기화 블록으로 수정

```
public class Person {
    private final Date birthDate;

    public Person(Date birthDate) {
            this.birthDate = birthDate;
    }

    private static final Date BOOM_START;
    private static final Date BOOM_END;

    static {
            Calendar gmtCal = Calendar.getInstance(TimeZone.getTimeZone("GMT"));
            gmtCal.set(1946, Calendar.JANUARY, 1, 0, 0, 0);
            BOOM_START = gmtCal.getTime();
            gmtCal.set(1965, Calendar.JANUARY, 1, 0, 0, 0);
            BOOM_END = gmtCal.getTime();
    }
    public boolean isBabyBoomer() {
            return birthDate.compareTo(BOOM_START) >= 0 &&
    birthDate.compareTo(BOOM_END) < 0;
    }
}
```

객체의 생성 과 삭제

- 불필요한 객체는 생성하지 않도록 하기
  - ✓ 불필요한 객체 생성으로 인한 시간 낭비: Long -> long(자동 객체화로 인한 객체 생성 낭비)

```
public class Main {
    public static void main(String[] args) {
        long start = System.currentTimeMillis();
        Long sum = 0L;
        for (long i = 0; i < Integer.MAX_VALUE; i++) {
            sum += i;
        }
        long end = System.currentTimeMillis();
        System.out.println(sum);
        System.out.println(end-start);
    }
}
```

객체의 생성 과 삭제

- 유효 기간이 지난 객체 참조는 폐기
  - ✓ 스택의 구현에서의 유효 기간이 지난 객체의 참조를 제거하기

```
public class Stack {
    private Object[] elements;
    private int size = 0;
    private static final int DEFAULT_INITIAL_CAPACITY = 16;

    public Stack() {
        elements = new Object[DEFAULT_INITIAL_CAPACITY];
    }

    void push(Object e) {
        ensureCapacity();
        elements[size++] = e;
    }
```

객체의 생성 과 삭제

- 유효 기간이 지난 객체 참조는 폐기
  - ✓ 스택의 구현에서의 유효 기간이 지난 객체의 참조를 제거하기

```
public Object pop() {
    /*
    if (size == 0)
        throw new EmptyStackException();
    return elements[--size];
     */


    if ( size == 0 )
        throw new EmptyStackException();
    Object result = elements[--size];
    elements [size] = null; //만기 참조 제거
    return result;
}
```

객체의 생성 과 삭제

- 유효 기간이 지난 객체 참조는 폐기
  - ✓ 스택의 구현에서의 유효 기간이 지난 객체의 참조를 제거하기

```
    /**
     * 적어도 하나 이상의 원소를 담을 공간을 보장한다.
     * 배열의 길이를 늘려야 할 때마다 대략 두 배씩 늘인다.
     */
    private void ensureCapacity() {
        if (elements.length == size)
            elements = Arrays.copyOf(elements, 2 * size + 1);
    }
}
```

객체의 생성 과 삭제

- 유효 기간이 지난 객체 참조는 폐기
  - ✓ 객체 참조를 null 처리하는 것은 규범(norm)이라기 보단 예외적인 조치가 되어야 함
  - ✓ 자체적으로 관리하는 메모리가 있는 클래스를 만들 때는 메모리 누수가 발생하지 않도록 주의
  - ✓ 캐시(cache)도 메모리 누수가 흔히 발생하는 장소
    - 객체 참조를 캐시 안에 넣어 놓고 잊어버리는 일이 많기 때문인데 이 문제에는 몇 가지 해결책이 있음
      - 첫 번째 해결책은 WeakHashMap을 가지고 캐시를 구현하는 것으로 캐시 바깥에서 키(key)를 참조하고 있을 때만 값(value)을 보관하면 될 때 쓸 수 있는 전략으로 키에 대한 참조가 만기 참조가 되는 순간 캐시 안에 보관된 키값 쌍은 자동으로 삭제되기 때문인데 WeakHashMap은 캐시 안에 보관되는 항목의 수명이 키에 대한 외부 참조의 수명에 따라 결정되는 상황에만 적용 가능하다는 것을 기억
      - 일반적으로 캐시에 보관되는 항목의 수명은 캐시에 보관된 기간에 따라 결정되는 것이 보통인데 그런 캐시는 이따금씩 사용하지 않는 항목들을 비워야 하며 이런 작업은 후면 스레드(background thread)를 사용해 처리할 수 도 있고(Timer나 ScheduledThreadPoolExecutor 사용) 캐시에 새로운 항목을 추가할 때 처리할 수도 있는데 LinkeclHashMap 클래스를 사용하면 두 번째 접근법을 구현하기 좋은데 removeEldestEntiy 메서드가 제공되기 때문인데 좀 더 복잡한 캐시의 경우에는 java.lang.ref를 직접 사용해야 할 수도 있음

객체의 생성 과 삭제

- ❖ 유효 기간이 지난 객체 참조는 폐기
  - ✓ 메모리 누수가 흔히 발견되는 또 한 곳은 리스너(listener) 등의 역호출자 (callback)
    - 역호출자 등록 기능을 제공하는 API를 사용하는 클라이언트가 역호출자를 명시적으로 제거하지 않을 경우 적절한 조치를 취하기 전까지 메모리는 점유된 상태로 남아있게 되는데 쓰레기 수집기가 역호출자를 즉시 처리하도록 할 가장 좋은 방법은 역호출자에 대한 약한 참조(weak reference)만 저장 하는 것으로 WeakHashMap의 키로 저장하는 것이 그 예
  - ✓ 메모리 누수는 보통 뚜렷한 오류로 이어지지 않기 때문에 수 년간 시스템에 남아 있는 경우도 있는데 이런 문제는 보통 주의 깊게 코드를 검토하다 발견되거나 힙 프로파일러(heap proper) 같은 도구를 통해 검증하다가 발견되기 때문에 이런 문제가 생길 수 있다는 것을 사전에 인지하고 방지 대책을 세우는 것이 바람직

객체의 생성 과 삭제

- 자원을 직접 명시하지 말고 의존 객체 주입을 사용
  - ✓ 클래스 내부에서 다른 클래스의 객체를 사용하는 경우 내부에서 직접 생성하지 않고 외부에서 주입받아서 생성하는 것을 권장
  - ✓ 주입을 받고자 할 때는 생성자를 이용한 주입 과 setter를 이용하는 주입 방식이 있음

객체의 생성 과 삭제

- Avoid finalizers and cleaners
  - ✓ finalizer: finalize 메서드를 오버라이드하여 Garbage Collect 대상이 될 때 수행
  - ✓ cleaner: 더 이상 사용되지 않을 때 등록된 스레드에서 정의된 클린 작업을 수행
    - Cleaner 구현

```
public class CleaningRequiredObject implements AutoCloseable {

    private static final Cleaner cleaner = Cleaner.create();

    private static class CleanData implements Runnable {

        @Override
        public void run() {
            // 여기서 클린 작업 수행
        }
    }
```

객체의 생성 과 삭제

- Avoid finalizers and cleaners
  - ✓ cleaner: 더 이상 사용되지 않을 때 등록된 스레드에서 정의된 클린 작업을 수행
    - Cleaner 구현

```
    private final CleanData cleanData;
    private final Cleaner.Cleanable cleanable;

    public CleaningRequiredObject() {
        this.cleanData = new CleanData();

        // 등록
        this.cleanable = cleaner.register(this, cleanData);
    }

    @Override
    public void close() {
        cleanable.clean();
    }
}
```

객체의 생성 과 삭제

- Avoid finalizers and cleaners
  - ✓ 활용
    - 클라이언트가 close 메소드를 호출하지 않을 경우를 대비한 안전망 역할
      - finalizer 또는 cleaner가 동작할거란 보장은 없지만 아무것도 안하는 것 보단 나을 수 있음
      - FileInputStream, FileOutputStream, ThreadPoolExecutor는 안전망 역할의 finalizer를 제공하는 라이브러리 중 하나
    - 네이티브 피어와 연결된 객체에서 사용
      - 네이티브 피어는 일반 자바 객체가 네이티브 메서드를 통해 기능을 위임한 네이티브 객체
      - 자바 객체가 아니므로 GC는 네이티브 객체의 존재조차 모르는데 이런 상황이라면 finalizer 또는 cleaner가 처리할 수 있지만 성능 저하를 감수할 수 있고 네이티브 피어가 그다지 중요하지 않은 자원을 가지고 있을 때에 해당
      - 그렇지 않다면 close 메서드를 활용

객체의 생성 과 삭제

- Avoid finalizers and cleaners
  - ✓ 문제점
    - 자바는 finalizer의 수행 시점은 물론 수행 여부조차 보장해주지 않음
    - try-with-resources 문에 비해 심각한 성능 저하가 있음
    - finalizer attack에 노출

```
public class Zombie {
    static Zombie zombie;

    public void finalize() {
       zombie = this;
    }
}
```

객체의 생성 과 삭제

- Avoid finalizers and cleaners
  - ✓ 권장
    - 명시적인 종료 메서드(termination method}를 하나 정의하고 내부에서 객체 종료를 보장하기 위해서 try-finally 문과 함께 작성
    - 종료 메서드의 예로 OutputStream이나 InputStream, java.sql. Connection에 정의된 close 메서드

객체의 생성 과 삭제

- try-finally보다는 try-with-resources를 사용
  - ✓ 자바 라이브러리의 직접 닫아줘야 하는 자원
    - 자바 라이브러리에는 close 메서드를 호출해 직접 닫아줘야 하는 자원이 많음
    - InputStream, OutputStream, java.sql.Connection 등이 좋은 예
  - ✓ 전통적인 자원 회수 방식
    - 전통적으로 자원이 제대로 닫힘을 보장하는 수단으로 try-finally를 사용했는데 이는 예외가 발생하거나 메서드에서 반환되는 경우까지도 포함하여 사용
    - 이 경우 자원이 둘 이상이면 너무 코드가 지저분

객체의 생성 과 삭제

- try-finally보다는 try-with-resources를 사용
  - ✓ 전통적인 자원 회수 방식

```
public class Item {
    private static final int BUFFER_SIZE = 0; // 편의상 0으로 지정

    static void copy(String src, String dst) throws IOException {
        InputStream in = new FileInputStream(src);
        try {
            OutputStream out = new FileOutputStream(dst);
            try {
                byte[] buf = new byte[BUFFER_SIZE];
                int n;
                while ((n = in.read(buf)) >= 0) {
                    out.write(buf, 0, n);
                }
            } finally {
                out.close();
            }
        } finally {
            in.close();
        }
    }
}
```

객체의 생성 과 삭제

- try-finally보다는 try-with-resources를 사용
  - ✓ 전통적인 자원 회수 방식
    - 문제점
      - 예외는 try 블록과 finally 블록 모두에서 발생할 수 있는데 예컨대 기기에 물리적인 문제가 생긴다면 firstLineOfFile 메서드 안의 readLine 메서드가 예외를 던지고 같은 이유로 close 메서드도 실패할 것인데 이런 상황이라면 두 번째 예외가 첫 번째 예외를 완전히 집어삼켜 버리고 스택 추적 내역에 첫 번째 예외에 관한 정보는 남지 않게 되어 실제 시스템에서의 디버깅을 몹시 어렵게 함
    - 해결책
      - try-with-resources 구조를 사용
      - 단 이 경우 자원이 AutoCloseable 인터페이스를 구현해야 함
      - 닫아야 하는 자원을 뜻하는 클래스를 작성한다면 AutoCloseable을 반드시 구현

객체의 생성 과 삭제

- try-finally보다는 try-with-resources를 사용
  - ✓ try-with-resources

```
public class Item {
    private static final int BUFFER_SIZE = 0; // 편의상 0으로 지정

    static void copy(String src, String dst) throws IOException {
        try (InputStream in = new FileInputStream(src);
             OutputStream out = new FileOutputStream(dst)) {
            byte[] buf = new byte[BUFFER_SIZE];
            int n;
            while ((n = in.read(buf)) >= 0) {
                out.write(buf, 0, n);
            }
        }
    }
}
```

객체의 생성 과 삭제

- try-finally보다는 try-with-resources를 사용
  - ✓ try-with-resources
    - readLine 과 (코드에는 나타나지 않는) close 호출 양쪽에서 예외가 발생하면 close에서 발생한 예외는 숨겨지고 readLine에서 발생한 예외가 기록
    - 숨겨진 예외들도 그냥 버려지지는 않고 스택 추적 내역에 숨겨졌다(suppressed)는 꼬리표를 달고 출력
    - 자바 7에서 Throwable에 추가된 getSuppressed 메서드를 이용하면 프로그램 코드에서 가져올 수도 있음

객체의 공통 메서드

- equals를 재정의할 때는 일반 규약을 따르라
  - ✓ equals를 재정의하지 않아도 되는 경우
    - 각각의 객체가 고유한 경우: 값(value) 대신 활성 개체(active entity)를 나타내는 Thread 같은 클래스가 이 조건에 부합하는데 이런 클래스는 Object의 equals 메서드를 그대로 사용해도 됨
    - 클래스에 논리적 동일성(logical equality) 검사 방법이 있건 없건 상관없는 경우: java.util.Random 클래스는 두 Random 객체가 같은 난수열(sequence of random numbers}을 만드는지 검사하는 equals 메서드를 재정의할 수도 있었지만 이 클래스를 설계한 사람들은 클라이언트가 그런 기능을 원할 거라 생각하지 않았기 때문에 Object에서 계승한 equals만으로 충분
    - 상위 클래스에서 재정의한 equals가 하위 클래스에서 사용하기에도 적당한 경우: 대부분의 Set 클래스는 AbstractSet의 equals 메서드를 그대로 사용하며 대부분의 List 클래스는 AbstractList의 equals 메서드를 그대로 사용하고 Map 클래스들은 대체로 AbstractMap의 equals 메서드를 그대로 사용
    - 클래스가 private 또는 package-private로 선언되었고 equals 메서드를 호출할 일이 없는 경우

객체의 공통 메서드

- equals를 재정의할 때는 일반 규약을 따르라
  - ✓ equals를 재정의하는 경우
    - 객체 동일성(object equality)이 아닌 논리적 동일성(logical equality)의 개념을 지원하는 클래스일 때
    - 상위 클래스의 equlas 가 하위 클래스의 필요를 충족 시키지 못할 때
    - 일반적으로 데이터를 저장하는 클래스는 동일한 객체인지 확인하는 경우보다는 같은 값을 나타내는지를 확인하기 때문에 필요
    - 값을 나타내지만 싱글톤 패턴의 형태로 디자인되는 enum의 경우는 equals를 재정의하지 않음
  - ✓ equals를 재정의 시 지켜야 할 내용
    - 반사성(reflexive): null이 아닌 참조가 있을 때 x.equals(x)는 true를 반환
    - 대칭성(symmetric): null 아닌 참조 x 와 y가 있을 때 x.equals(y)는 y.equals(x)가 true일 때만 true를 반환
    - 추이성(transitive): null 아닌 참조 x, y, z가 있을 때 x.equals(y)가 true이고 y.equals(z)가 true이면 x.equals(z)도 true
    - 일관성(consistent): null 아닌 참조 x 와 y가 있을 때 equals를 통해 비교되는 정보에 아무 변화가 없다면 x.equals(y) 호출 결과는 호출 횟수에 상관없이 항상 같아야 함
    - null 아닌 참조 x에 대해서 x.equals(null)은 항상 false
    - hashCode 도 재정의

객체의 공통 메서드

- equals를 재정의할 때는 일반 규약을 따르라
  - ✓ equals를 재정의하는 경우
    - 객체 동일성(object equality)이 아닌 논리적 동일성(logical equality)의 개념을 지원하는 클래스일 때
    - 상위 클래스의 equlas 가 하위 클래스의 필요를 충족 시키지 못할 때
    - 일반적으로 데이터를 저장하는 클래스는 동일한 객체인지 확인하는 경우보다는 같은 값을 나타내는지를 확인하기 때문에 필요
    - 값을 나타내지만 싱글톤 패턴의 형태로 디자인되는 enum의 경우는 equals를 재정의하지 않음
  - ✓ equals를 재정의 시 지켜야 할 내용
    - 반사성(reflexive): null이 아닌 참조가 있을 때 x.equals(x)는 true를 반환
    - 대칭성(symmetric): null 아닌 참조 x 와 y가 있을 때 x.equals(y)는 y.equals(x)가 true일 때만 true를 반환
    - 추이성(transitive): null 아닌 참조 x, y, z가 있을 때 x.equals(y)가 true이고 y.equals(z)가 true이면 x.equals(z)도 true
    - 일관성(consistent): null 아닌 참조 x 와 y가 있을 때 equals를 통해 비교되는 정보에 아무 변화가 없다면 x.equals(y) 호출 결과는 호출 횟수에 상관없이 항상 같아야 함
    - null 아닌 참조 x에 대해서 x.equals(null)은 항상 false

객체의 공통 메서드

- equals를 재정의하려거든 hashCode도 재정의
  - ✓ equals 메서드를 재정의하는 클래스는 반드시 hashCode 메서드도 재정의 해야 하는데 그러지 않으면 Object.hashCode의 일반 규약을 어기게 되므로 HashMap, HashSet, Hashtable 같은 해시(hash) 기반 컬렉션과 함께 사용하면 오동작

객체의 공통 메서드

- equals를 재정의하려거든 hashCode도 재정의
  - ✓ 문제 발생 – null 리턴
    - PhoneNumber.java

```
public final class PhoneNumber {
    private final short areaCode;
    private final short prefix;
    private final short lineNumber;

    public PhoneNumber(int areaCode, int prefix, int lineNumber) {
        rangeCheck(areaCode, 999, "area code");
        rangeCheck(prefix, 999, "prefix");
        rangeCheck(lineNumber, 9999, "line number");

        this.areaCode = (short) areaCode;
        this.prefix = (short) prefix;
        this.lineNumber = (short) lineNumber;
    }

    private static void rangeCheck(int arg, int max, String name) {
        if (arg < 0 || arg > max)
            throw new IllegalArgumentException(name + ": " + arg);
    }
```

객체의 공통 메서드

- equals를 재정의하려거든 hashCode도 재정의
  - ✓ 문제 발생
    - PhoneNumber.java

```
@Override
public boolean equals(Object o) {
    if (o == this) return true;
    if (!(o instanceof PhoneNumber))
        return false;
    PhoneNumber pn = (PhoneNumber) o;
    return pn.lineNumber == lineNumber && pn.prefix == prefix &&
 pn.areaCode == areaCode;
}

// hashCode 메서드가 없으므로 문제가 발생
}
```

객체의 공통 메서드

- equals를 재정의하려거든 hashCode도 재정의
  - ✓ 문제 발생
    - Main.java

```
public class Main {
    public static void main(String[] args) {
        Map<PhoneNumber, String> m = new HashMap<PhoneNumber, String>();
        m.put(new PhoneNumber(707, 867, 5309), "Jenny");
        System.out.println(m.get(new PhoneNumber(707, 867, 5309)));
    }
}
```

객체의 공통 메서드

- equals를 재정의하려거든 hashCode도 재정의
  - ✓ 문제 해결
    - PhoneNumber에 메서드 추가

```
@Override
public int hashCode() {
        int result = 17;
        result = 31 * result + areaCode;
        result = 31 * result + prefix;
        result = 31 * result + lineNumber;
        return result;
}
```

객체의 공통 메서드

- toString은 항상 재정의
  - ✓ java.lang.Object 클래스가 toString 메서드를 제공하긴 하지만 이 메서드가 반환하는 문자열은 일반적으로는 사용자가 보려는 문자열이 아니며 클래스 이름 다음에 @ 기호와 16진수로 표현된 해시 코드가 붙은 문자열로 PhoneNumber@163b91 과 같은 형태
  - ✓ toString의 일반 규약을 보면 toString 이 반환하는 문자열은 사람이 읽기 쉽도록 간략하지만 유용한 정보를 제공해야 한다 고 되어 있음
  - ✓ PhoneNumter@163b91 와 같은 문자열이 간략하지만 유용한 정보인지에는 논쟁의 여지가 있지만 (707) 867-5309 같은 문자열에 비해서는 그다지 유용하지 못한 것이 사실
  - ✓ toString의 일반 규약에는 이런 구절도 있는데 모든 하위 클래스는 이 메서드를 재정의함이 바람직하다. 라는 것
  - ✓ equals와 hashCode의 일반 규약을 지키는 것보다는 덜 중요하지만 toString을 잘 만들어 놓으면 클래스를 좀 더 쾌적하게 사용할 수 있는데 toString 메서드는 println이나 printf 같은 함수, 문자열 연결 연산자(string concatenation operator), assert, 디버거(debugger) 등에 객체가 전달되면 자동으로 호출

객체의 공통 메서드

- toString은 항상 재정의
  - ✓ 재정의 규칙
    - 가능하다면 toString 메서드는 객체 내의 중요 정보를 전부 담아 반환
    - toString이 반환하는 문자열의 형식을 명시하건 그렇지 않건 간에, 어떤 의도 인지는 문서에 분명하게 남김
    - toString이 반환하는 문자열에 포함되는 정보들은 전부 프로그래밍을 통해서 가져올 수 있도록(programmatic access) 작성

객체의 공통 메서드

- clone을 재정의할 때는 신중
  - ✓ Cloneable은 어떤 객체가 복제(clone)를 허용한다는 사실을 알리는 데 쓰려고 고안된 믹스인(mixin) 인터페이스인데 불행히도 당초 목적에는 부합하지 못하고 있음
  - ✓ 기본적인 문제는 이 인터페이스에는 clone 메서드가' 없으며 Object 의 clone 메서드는 protected로 선언되어 있다는 것
  - ✓ 리플렉션(reflection)을 사용하지 않고는 Cloneable을 구현한 객체라 해도 clone 메서드를 호출할 방법이 없는데 리플렉션을 사용한 호출도 실패할 가능성이 있는데 해당 객체에 호출 가능한 clone 메서드가 구현되어 있으리라는 보장이 없기 때문
  - ✓ 이런저런 단점에도 불구하고 Cloneable은 널리 사용되고 있음
  - ✓ 어떤 클래스가 Cloneable을 구현하면 Object의 clone 메서드는 해당 객체를 필드 단위로 복사한 객체를 반환
  - ✓ Clone 메서드 재정의 시 권장 사항
    - Non-Final 클래스에 clone을 재정의할 때는 반드시 super.clone을 호출해 얻은 객체를 반환
    - Cloneable 인터페이스를 구현하는 클래스는 제대로 동작하는 pubUc clone 메서드를 제공

객체의 공통 메서드

- clone을 재정의할 때는 신중
  - ✓ PhoneNumber에서의 Clone

```
@Override
public PhoneNumber clone() {
        try {
            return (PhoneNumber) super.clone();
        } catch(CloneNotSupportedException e ) {
            throw new AssertionError();
        }
}
```

객체의 공통 메서드

- clone을 재정의할 때는 신중
  - ✓ 내부에 데이터의 모임이 있는 경우 – 내부 구조도 복사

```
@Override
public Stack clone() {
        try {
            Stack result = (Stack) super.clone();
            result.elements = elements.clone();
            return result;
        } catch ( CloneNotSupportedException e ) {
            throw new AssertionError();
        }
}
```

객체의 공통 메서드

- clone을 재정의할 때는 신중
  - ✓ 내부에 데이터의 모임이 있는 경우 – 내부 구조도 복사

```
@Override
public Stack clone() {
        try {
            Stack result = (Stack) super.clone();
            result.elements = elements.clone();
            return result;
        } catch ( CloneNotSupportedException e ) {
            throw new AssertionError();
        }
}
```

객체의 공통 메서드

- clone을 재정의할 때는 신중
  - ✓ 객체를 지원하는 좋은 방법은 복사 생성자 나 복사 팩토리 메서드를 제공하는 것

```
public Yum(Yum yum);

public static Yum newInstance(Yum yum);
```

객체의 공통 메서드

- Comparable 구현을 고려
  - ✓ Comparable 인터페이스를 구현한 객체들은 검색하거나 최대/최소치를 계산하기도 간단하며 그 컬렉션을 정렬된 상태로 유지하기도 쉬움
  - ✓ Comparable을 구현한 클래스는 다양한 제네릭 알고리즘 및 Comparable 인터페이스를 이용하도록 작성된 컬렉션 구현체 와도 전부 연동할 수 있음
  - ✓ 적은 노력으로 엄청난 결실을 거둘 수 있는 것인데 자바 플랫폼 라이브러리에 포함 된 거의 모든 값 클래스(value class)는 Comparable 인터페이스를 구현
  - ✓ 알파벳 순서나 값의 크기, 또는 시간적 선후 관계처럼 명확한 자연적 순서를 따르는 값 클래스를 구현할 때는 Comparable 인터페이스를 구현할 것을 반드시 고려

클래스 와 인터페이스

- 클래스와 멤버의 접근 권한은 최소화
  - ✓ 잘 설계된 모듈과 그렇지 못한 모듈을 구별짓는 가장 중요한 속성 하나는 모듈 내부의 데이터를 비롯한 구현 세부 사항을 다른 모듈에 잘 감추느냐 의 여부
  - ✓ 잘 설계된 모듈은 구현 세부 사항을 전부 API 뒤쪽에 감추는데 모듈들은 이 API를 통해서만 서로 통신하며 각자 내부적으로 무슨 짓을 하는지는 신경 쓰지 않음
  - ✓ 이 개념은 정보 은닉(information hidiing) 또는 캡슐화(encapsulation) 라는 용어로 알려져 있으며 소프트웨어 설계의 기본적인 원칙 가운데 하나
  - ✓ 정보 은닉은 여러 가지 이유로 중요한데 그 대부분은 정보 은닉이 시스템을 구성하는 모듈 사이의 의존성을 낮춰서(decouple) 각자 개별적으로 개발하고 시험하고 최적화하고 이해하고 변경할 수 있도록 한다는 사실에 기초하며 시스템 개발 속도가 올라가는데 각각의 모듈을 병렬적으로 개발할 수 있기 때문
  - ✓ 유지 보수의 부담도 낮아지는데 모듈 각각을 좀 더 빨리 이해할 수 있을 뿐 아니라 다른 모듈에 영향을 끼칠 걱정 없이 디버깅을 진행할 수 있기 때문이며 정보 은닉 원칙이 좋은 성능을 자동적으로 보장하는 것은 아니지만 효과적인 성능 튜닝(tuning)을 가능하게 하는 것은 사실인데 시스템이 완성된 다음에 어떤 모듈이 성능 문제를 일으키는지 프로파일링(profiling) 하기 용이하기 때문
  - ✓ 해당 모듈은 다른 모듈과는 관계없이 최적화할 수 있으며 정보 은닉 원칙은 소프트웨어의 재사용 가능성을 높이는데 모듈 간 의존성이 낮으므로 각 모듈은 다른 소프트웨어 개발에도 유용하게 쓰일 수 있으며 아울러 정보 은닉 원칙은 대규모 시스템 개발 과정의 위험성(risk)도 낮추는데 전체 시스템은 성공적이지 않더라도 각각의 모듈은 성공적으로 구현 될 수 있기 때문

클래스 와 인터페이스

- 클래스와 멤버의 접근 권한은 최소화
  - ✓ 원칙
    - 클래스와 멤버는 가능한 한 접근 불가능하도록 생성
    - 객체 필드(instance field)는 절대로 public 으로 선언하면 안되는데 변경 가능 public 필드를 가진 클래스는 다중 스레드에 안전하지 않음
    - 어떤 상수들이 클래스로 추상화된 결과물의 핵심적 부분을 구성한다고 판단되는 경우 해당 상수들을 public static final 필드들로 선언하여 공개할 수 있는데 이런 필드들은 관습적으로 대문자로 구성된 이름을 가지며 이름을 구성하는 단 어들은 밑줄(underecore)기호로 구분하며 반드시 기본 자료형 값들을 갖거나 변경 불가능 객체를 참조해야 하는데 변경 가능 객체를 참조하는 final 필드는 참조 자체는 변경할 수 없지만 참조 대상 객체는 변경할 수 있기 때문
    - public 클래스 안에는 public 필드를 만들지 말고 접근자 메서드를 사용

클래스 와 인터페이스

- 변경 가능성을 최소화
  - ✓ 변경 불가능(immutable) 클래스는 그 객체를 수정할 수 없는 클래스로 객체 내부의 정보는 객체가 생성될 때 주어진 것이며 객체가 살아 있는 동안 그대로 보존
  - ✓ 자바 플랫폼 라이브러리에는 String, Wrapper Class, Biginteger, BigDecimal 등이 변경 불가능 클래스
  - ✓ 변경 불가능 클래스를 만드 는 이유는 다양한데 변경 불가능 클래스는 변경 가능 클래스보다 설계하기 쉽고 구현하기 쉬우며 사용하기도 쉽고 오류 가능성도 적고 더 안전
  - ✓ 변경 불가능한 클래스를 만들 때 지켜야 할 규칙
    - 객체 상태를 변경하는 메서드(mutator 메서드 등)를 제공하지 않음
    - 상속할 수 없도록 생성
    - 모든 필드를 final 로 선언
    - 모든 필드를 private 로 선언
    - 변경 가능 컴포넌트에 대한 독점적 접근권을 보장

클래스 와 인터페이스

- ❖ 변경 가능성을 최소화
  - ✓ 장점
    - 변경 불가능 객체는 단순
    - 변경 불가능 객체는 스레드에 안전하기 때문에 동기화가 필요 없음
    - 변경 불가능한 객체는 공유 가능
    - 변경 불가능한 객체는 그 내부도 공유할 수 있음
    - 다른 객체의 구성 요소로도 훌륭
  - ✓ 단점
    - 값 마다 별도의 객체를 생성해야 함

클래스 와 인터페이스

- ❖ 상속보다는 컴포지션 사용
  - ✓ 상속은 코드 재사용을 돕는 강력한 도구지만 항상 최선이라고는 할 수 없음
  - ✓ 상속을 적절히 사용하지 못한 소프트웨어는 깨지기 쉬운데 상속은 상위 클래스 와 하위 클래스 구현을 같은 프로그래머가 통제하는 단일 패키지 안에서 사용하면 안전
  - ✓ 상속을 고려해 설계되고 그에 맞는 문서를 갖춘 클래스에 사용하는 것도 안전
  - ✓ 일반적인 객체 생성 가능 클래스(concrete class) 라면 해당 클래스가 속한 패키지 밖에서 상속을 시도하는 것은 위험
  - ✓ 메서드 호출과 달리 상속은 캡슐화(encapsulation) 원칙을 위반
    - 하위 클래스가 정상 동작하기 위해서는 상위 클래스의 구현에 의존할 수 밖 에 없음
    - 상위 클래스의 구현은 릴리스(release)가 거듭되면서 바뀔 수 있는데 그러다 보면 하위 클래스 코드는 수정된 적이 없어도 망가질 수 있기 때문에 상위 클래스 작성자가 상속을 고려해 클래스를 설계하고 문서까지 만들어 놓지 않았다면 하위 클래스는 상위 클래스의 변화에 발맞춰 진화해야 함

클래스 와 인터페이스

- 상속보다는 컴포지션 사용
  - ✓ 잘못된 상속
    - HashSet을 상속하는 클래스

```
import java.util.Collection;
import java.util.HashSet;

public class InstrumentedHashSet<E> extends HashSet<E> {
    // 요소를 삽입하려 한 횟수
    private int addCount = 0;
    public InstrumentedHashSet() { }

    public InstrumentedHashSet(int initCap, float loadFactor) {
        super(initCap, loadFactor);
    }

    @Override
    public boolean add(E e) {
        addCount++;
        return super.add(e);
    }
```

클래스 와 인터페이스

- ❖ 상속보다는 컴포지션 사용
  - ✓ 잘못된 상속
    - ● HashSet을 상속하는 클래스

```
    @Override
    public boolean addAll(Collection<? extends E> c) {
        addCount += c.size();
        return super.addAll(c);
    }

    public int getAddCount () {
        return addCount;
    }
}
```

클래스 와 인터페이스

- ❖ 상속보다는 컴포지션 사용
  - ✓ 잘못된 상속
    - ● 실행 클래스

```
public class Main {
    public static void main(String[] args) {
        InstrumentedHashSet<String> s = new InstrumentedHashSet<String>();
        s.addAll(Arrays.asList("Java", "Python", "C++"));
        System.out.println("데이터 개수:" + s.getAddCount());
    }
}
```

클래스 와 인터페이스

- ❖ 상속보다는 컴포지션 사용
  - ✓ 잘못된 상속
    - ● 실행 클래스
      - ◆ getAddCount 메서드가 3을 반환할 것이라 기대하겠지만 실제로는 6을 리턴
      - ◆ InstrumentedHashSet에 정의된 addAll 메서드는 addCount에 3을 더하고 상위 클래스인 HashSet의 addAll 메서드를 super.addAll과 같이 호출하는데 이 메서드는 InstrumentedHashSet에서 재정의한 add 메서드를 삽입할 원소마다 호출하게 되서 각각의 add 메서드가 호출될 때마다 addCount는 1씩 증가할 것 이므로 총 6만큼 증가해서 addAll 메서드로 추가한 요소는 중복 계산하게 되는 것
  - ✓ 기존 클래스를 상속하는 대신 새로운 클래스에 기존 클래스 객체를 참조하는 private 필드를 하나 두는 것으로 해결할 수 있는데 이런 설계 기법을 구성(composition)이라고 부르는데 기존 클래스가 새 클래스의 일부(component)가 되기 때문
  - ✓ 새로운 클래스에 포함된 각각의 메서드는 기존 클래스에 있는 메서드 가운데 필요한 것을 호출해서 그 결과를 반환하면 되는데 이런 구현 기법을 전달(forwarding)이라고 하고 전달 기법을 사용해 구현된 메서드를 전달 메서드(forwarding method)라고 부름
  - ✓ 구성 기법을 통해 구현된 클래스는 견고한데 기존 클래스의 구현 세부 사항에 종속되지 않기 때문
  - ✓ 기존 클래스에 또 다른 메서드가 추가되더라도 새로 만든 클래스에는 영향이 없음

클래스 와 인터페이스

- ❖ 상속보다는 컴포지션 사용
  - ✓ 상속 대신 컴포지션 사용
    - ● 재사용 가능한 전달 클래스 – FowardingSet

```
public class ForwardingSet<E> implements Set<E> {
    private final Set<E> s;

    public ForwardingSet(Set<E> s) {
        this.s = s;
    }

    public void clear() {
        s.clear();
    }

    public boolean contains(Object o) {
        return s.contains(o);
    }

    public boolean isEmpty() {
        return s.isEmpty();
    }
```

클래스 와 인터페이스

- ❖ 상속보다는 컴포지션 사용
  - ✓ 상속 대신 컴포지션 사용
    - ● 재사용 가능한 전달 클래스 – FowardingSet

```
public int size() {
    return s.size();
}

public Iterator<E> iterator() {
    return s.iterator();
}

public boolean add(E e) {
    return s.add(e);
}

public boolean remove(Object o) {
    return s.remove(o);
}

public boolean containsAll(Collection<?> c) {
    return s.containsAll(c);
}
```

클래스 와 인터페이스

- ❖ 상속보다는 컴포지션 사용
  - ✓ 상속 대신 컴포지션 사용
    - ● 재사용 가능한 전달 클래스 – FowardingSet

```
public boolean addAll(Collection<? extends E> c) {
    return s.addAll(c);
}

public boolean removeAll(Collection<?> c) {
    return s.removeAll(c);
}

public boolean retainAll(Collection<?> c) {
    return s.retainAll(c);
}

public Object[] toArray() {
    return s.toArray();
}

public <T> T[] toArray(T[] a) {
    return s.toArray(a);
}
```

클래스 와 인터페이스

- ❖ 상속보다는 컴포지션 사용
  - ✓ 상속 대신 컴포지션 사용
    - ● 재사용 가능한 전달 클래스 – FowardingSet

```
    @Override
    public boolean equals(Object o) {
        return s.equals(o);
    }

    @Override
    public int hashCode() {
        return s.hashCode();
    }

    @Override
    public String toString() {
        return s.toString();
    }
}
```

클래스 와 인터페이스

- ❖ 상속보다는 컴포지션 사용
  - ✓ 상속 대신 컴포지션 사용
    - ● 포장 클래스 – InstrumentedSet

```
public class InstrumentedSet<E> extends ForwardingSet<E> {
    private int addCount = 0;

    public InstrumentedSet(Set<E> s) {
        super(s);
    }

    @Override
    public boolean add(E e) {
        addCount++;
        return super.add(e);
    }
```

클래스 와 인터페이스

- 상속보다는 컴포지션 사용
  - ✓ 상속 대신 컴포지션 사용
    - 포장 클래스 – InstrumentedSet

```
    @Override
    public boolean addAll(Collection<? extends E> c) {
        addCount += c.size();
        return super.addAll(c);
    }

    public int getAddCount() {
        return addCount;
    }
}
```

클래스 와 인터페이스

- 상속을 고려해 설계하고 문서화 – 그러지 않았다면 상속을 금지
  - ✓ 재정의 가능 메서드를 내부적으로 어떻게 사용하는지(self-use) 반드시 문서에 남김
  - ✓ 클래스 내부 동작에 개입할 수 있는 훅(hooks)을 신중하게 고른 protected 메서드 형태로 제공
  - ✓ 상속을 위해 설계한 클래스를 테스트할 유일한 방법은 하위 클래스를 직접 만들어 보는 것 – 릴리스에 포함시키기 전에 반드시 하위 클래스를 만들어서 테스트
  - ✓ 생성자는 직접적이건 간접적이건 재정의 가능 메서드를 호출해서는 안됨
  - ✓ clone이나 readObject 메서드 안에서 직접적이건 간접적이건 재정의 가능한 메서드를 호출하지 않도록 주의

클래스 와 인터페이스

- 추상 클래스보다는 인터페이스를 우선
  - ✓ 자바 언어에는 여러 가지 구현을 허용하는 자료형을 만드는 방법이 두 가지 포함되어 있는데 인터페이스와 추상 클래스(abstract class)
  - ✓ 추상 클래스는 구현된 메서드를 포함할 수 있지만 인터페이스는 아니라는 것이며 추상 클래스가 규정하는 자료형을 구현하기 위해서는 추상 클래스를 반드시 상속해야 한다는 것인데 인터페이스의 경우에는 인터페이스에 포함된 모든 메서드를 정의하고 인터페이스가 규정하는 일반 규약(general contract)을 지키기만 하면 되며 그렇게 만든 클래스는 클래스 계층(class hierarchy)에 속할 필요가 없음
  - ✓ 자바는 다중 상속(multiple inheritance)를 허용하지 않기 때문에 추상 클래스를 사용하게 되면 자료형으로 사용하는 데 많은 제약이 발생
  - ✓ 인터페이스는 mixin을 정의하는데 이상적
    - 믹스인은 클래스가 주 자료형(primary type) 이외에 추가로 구현할 수 있는 자료형으로 어떤 선택적 기능을 제공한다는 사실을 선언하기 위해 사용하는데 예를 들어 Comparable은 어떤 클래스가 자기 객체는 다른 객체와의 비교 결과에 따른 순서를 갖는다고 선언할 때 쓰는 믹스인 인터페이스로 이런 인터페이스가 믹스인
    - 자료형의 주된 기능에 선택적인 기능을 혼합(mix in)할 수 있도록 하기 때문인데 추상 클래스는 믹스인 정의에는 사용할 수 없는데 클래스가 가질 수 있는 상위 클래스는 하나뿐이며 클래스 계층에는 믹스인을 넣기 좋은 곳이 없음

클래스 와 인터페이스

- 추상 클래스보다는 인터페이스를 우선
  - ✓ 인터페이스는 비 계층적인(nonhierarchical) 자료형 프레임워크(type framework)를 만들 수 있음
  - ✓ 인터페이스를 사용하면 포장 클래스 숙어(wrapper class idiom)을 통해 안전하면서도 강력한 기능 개선이 가능
  - ✓ 추상 골격 구현(abstract skeletal implementation) 클래스를 중요 인터페이스마다 두면 인터페이스의 장점과 추상 클래스의 장점을 결합할 수 있음
  - ✓ 인터페이스보다는 추상 클래스가 발전시키기 쉬움
  - ✓ 인터페이스가 공개되고 널리 구현된 다음에는 인터페이스 수정이 거의 불가능

클래스 와 인터페이스

- ❖ 인터페이스는 구현하는 쪽을 생각해 설계
  - ✓ Default Method
    - 자바 8 이전에는 기존 구현체를 깨뜨리지 않고는 인터페이스에 메서드를 추가할 방법이 없었음
    - 인터페이스에 메서드를 추가하면 보통은 컴파일 오류가 나는데 추가된 메서드가 기존 구현체에 존재할 가능성이 아주 낮기 때문
    - 자바 8 이후부터 기존 인터페이스에 메서드를 추가할 수 있도록 디폴트 메서드 추가
    - 디폴트 메서드를 선언하면 그 인터페이스를 구현한 후 디폴트 메서드를 재정의하지 않은 모든 클래스에서 디폴트 구현이 쓰이게 됨
    - 디폴트 메서드는 구현 클래스에 대해 아무것도 모른 채 합의없이 무작정 삽입될 뿐이므로 주의
    - 자바 8에서는 핵심 컬렉션 인터페이스들에 다수의 디폴트 메서드가 추가되었는데 이는 주로 람다를 활용하기 위해서
    - 자바 라이브러리의 디폴트 메서드는 코드 품질이 높고 범용적이라 대부분의 상황에서 잘 작동하지만 생각할 수 있는 모든 상황에서 불변식을 해치지 않는 디폴트 메서드를 작성하는 것은 어려움

클래스 와 인터페이스

- 인터페이스는 구현하는 쪽을 생각해 설계
  - ✓ 주의 사항
    - 디폴트 메서드는 (컴파일에 성공하더라도) 기존 구현체에 런타임 오류를 일으킬 수 있음
    - 기존 인터페이스에 디폴트 메서드로 새 메서드를 추가하는 일은 꼭 필요한 경우가 아니면 피해야 함
    - 새로운 인터페이스를 만드는 경우라면 표준적인 메서드 구현을 제공하는 데 아주 유용한 수단이며 그 인터페이스를 더 쉽게 구현해 활용할 수 있게끔 해줌
    - 디폴트 메서드는 인터페이스로부터 메서드를 제거하거나 기존 메서드의 시그니처를 수정하는 용도가 아님을 명심

클래스 와 인터페이스

- 인터페이스는 자료형을 정의할 때 만 사용
  - ✓ 상수 인터페이스 패턴은 인터페이스를 잘못 사용한 것
  - ✓ 상수 인터페이스

```
public interface PhysicalConstants {
    // 아보가드로 수(1/mol)
    static final double AVOGADROS_NUMBER = 6.02214199e23;
    // 볼쯔만 상수(J/K)
    static final double BOLTZMANN_CONSTANT = 1.3806503e-23;
    // 전자 질량(kg)
    static final double ELECTRON_MASS = 9.10938188e-31;
}
```

클래스 와 인터페이스

- 태그 달린 클래스보다는 클래스 계층 구조를 활용
  - ✓ 태그 달린 클래스: 두 가지 이상의 의미를 표현할 수 있으며 그중 현재 표현하는 의미를 태그 값으로 알려주는 클래스
  - ✓ 일반적으로 태그 달린 클래스는 클래스 계층 구조보다 나쁨

클래스 와 인터페이스

❖ 태그 달린 클래스보다는 클래스 계층 구조를 활용

✓ 태그 달린 클래스

```
public class FigureWithTag {
    enum Shape {RECTANGLE, CIRCLE};

    // 태그 필드 - 현재 모양을 나타낸다.
    final Shape shape;

    // 다음 필드들은 모양이 사각형(RECTANGLE)일 때만 쓰인다.
    double length;
    double width;

    // 다음 필드는 모양이 원(CIRCLE)일 때만 쓰인다.
    double radius;
```

클래스 와 인터페이스

- 태그 달린 클래스보다는 클래스 계층 구조를 활용
  - ✓ 태그 달린 클래스

```
// 원 생성자
public FigureWithTag(double radius) {
    this.shape = Shape.CIRCLE;
    this.radius = radius;
}

// 사각형 생성자
public FigureWithTag(double length, double width) {
    this.shape = Shape.RECTANGLE;
    this.length = length;
    this.width = width;
}
```

클래스 와 인터페이스

- 태그 달린 클래스보다는 클래스 계층 구조를 활용
  - ✓ 태그 달린 클래스

```
double area() {
    switch (shape) {
        case RECTANGLE:
            return length * width;
        case CIRCLE:
            return Math.PI * (radius * radius);
        default:
            throw new AssertionError(shape);
    }
}
```

클래스 와 인터페이스

- ❖ 태그 달린 클래스보다는 클래스 계층 구조를 활용
  - ✓ 태그 달린 클래스 단점
    - 태그 달린 클래스에는 쓸데없는 코드가 많음
    - 여러 구현이 한 클래스에 혼합돼 있어서 가독성도 나쁨
    - 다른 의미를 위한 코드도 언제나 함께 하므로 메모리도 많이 사용
    - 필드들을 final로 선언하려면 해당 의미에 쓰이지 않는 필드들까지 생성자에서 초기화해야 하기 때문에 불필요한 코드가 늘어남
    - 생성자가 태그 필드를 설정하고 해당 의미에 쓰이는 데이터 필드들을 초기화하는 데 컴파일러가 도와줄 수 있는 게 별로 없고 엉뚱한 필드를 초기화해도 런타임에서 문제가 드러날 뿐
    - 다른 의미를 추가하려면 코드를 수정해야 함
    - 인스턴스의 타입만으로는 현재 나타내는 의미를 알 방법이 없음

클래스 와 인터페이스

- ❖ 태그 달린 클래스보다는 클래스 계층 구조를 활용
  - ✓ 태그 달린 클래스를 계층 구조로 만드는 방법
    - 계층구조의 루트(root)가 될 추상 클래스를 정의
    - 태그 값에 따라 동작이 달라지는 메서드들을 루트 클래스의 추상 메서드로 선언
    - 태그 값에 상관없이 동작이 일정한 메서드들을 루트 클래스에 일반 메서드로 추가
    - 모든 하위 클래스에서 공통으로 사용하는 데이터 필드들도 전부 루트 클래스에 작성
    - 루트 클래스를 확장한 구체 클래스를 의미 별로 하나씩 정의
    - 각 하위 클래스에는 각자의 의미에 해당하는 데이터 필드를 작성
    - 루트 클래스가 정의한 추상 메서드를 각자의 의미에 맞게 구현

클래스 와 인터페이스

- 태그 달린 클래스보다는 클래스 계층 구조를 활용
  - ✓ 태그 달린 클래스를 계층 구조로 변경

```
public abstract class Figure {
    abstract double area();
}
```

클래스 와 인터페이스

- 태그 달린 클래스보다는 클래스 계층 구조를 활용
  - ✓ 태그 달린 클래스를 계층 구조로 변경

```
public class Circle extends Figure {
    final double radius;

    public Circle(double radius) {
        this.radius = radius;
    }

    @Override
    double area() {
        return Math.PI * (radius * radius);
    }
}
```

클래스 와 인터페이스

- 태그 달린 클래스보다는 클래스 계층 구조를 활용
  - ✓ 태그 달린 클래스를 계층 구조로 변경

```
public class Rectangle extends Figure {
    final double length;
    final double width;

    public Rectangle(double length, double width) {
        this.length = length;
        this.width = width;
    }

    @Override
    double area() {
        return length * width;
    }
}
```

클래스 와 인터페이스

- 멤버 클래스는 되도록 static 으로
  - ✓ Nested Class
    - 중첩 클래스(nested class)란 다른 클래스 안에 정의된 클래스
    - 중첩 클래스는 자신을 감싼 바깥 클래스에서만 쓰여야 하며 그 외의 쓰임새가 있다면 톱레벨 클래스로 만들어야 함
    - 중첩 클래스의 종류는 정적 멤버 클래스, (비정적) 멤버 클래스, 익명 클래스, 지역 클래스 네 가지가 있는데 정적 멤버 클래스를 제외한 나머지는 내부 클래스(inner class)에 해당
  - ✓ 정적 멤버 클래스
    - 정적 멤버 클래스는 다른 클래스 안에 선언되고 바깥 클래스의 private 멤버에도 접근할 수 있음
    - 정적 멤버 클래스는 다른 정적 멤버와 똑같은 접근 규칙을 적용받음
    - 정적 멤버 클래스는 보통 바깥 클래스와 함께 쓰일 때만 유용한 public 도우미 클래스로 사용

클래스 와 인터페이스

- ❖ 멤버 클래스는 되도록 static 으로
  - ✓ 비정적 멤버 클래스
    - 정적 멤버 클래스와의 구문상 차이는 static의 유무
    - 의미상 차이는 바깥 클래스의 인스턴스와 연결되는지 아닌지 여부
    - 비정적 멤버 클래스의 인스턴스는 바깥 클래스의 인스턴스와 암묵적으로 연결
    - 비정적 멤버 클래스의 인스턴스 메서드에서 정규화된 this를 사용해 바깥 인스턴스의 메서드를 호출하거나 바깥 인스턴스의 참조를 가져올 수 있음
    - 정규화된 this란 클래스명.this 형태로 바깥 클래스의 이름을 명시하는 용법
    - 개념상 중첩 클래스의 인스턴스가 바깥 인스턴스와 독립적으로 존재할 수 있다면 정적 멤버 클래스로 만들어야 하는데 비정적 멤버 클래스는 바깥 인스턴스 없이는 생성할 수 없기 때문
    - 비정적 멤버 클래스의 인스턴스와 바깥 인스턴스 사이의 관계는 멤버 클래스가 인스턴스화될 때 확립되며 더 이상 변경할 수 없음
    - 이 관계 정보는 비정적 멤버 클래스의 인스턴스 안에 만들어져 메모리 공간을 차지하며 생성 시간도 더 걸림
    - 비정적 멤버 클래스는 어댑터를 정의할 때 자주 사용
    - 어떤 클래스의 인스턴스를 감싸 마치 다른 클래스의 인스턴스처럼 보이게 하는 뷰로 사용하는 것

클래스 와 인터페이스

- ❖ 멤버 클래스는 되도록 static 으로
  - ✓ 멤버 클래스에서 바깥 인스턴스에 접근할 일이 없다면 무조건 static을 붙여서 정적 멤버 클래스로 생성
    - static을 생략하면 바깥 인스턴스로의 숨은 외부 참조를 갖게 되는데 이 참조를 저장하려면 시간과 공간이 소비
    - 숨은 참조로 인해 바깥 클래스의 인스턴스를 가비지 컬렉션이 수거하지 못하는 메모리 누수가 발생할 수 있게 됨
    - 숨은 참조는 눈에 보이지 않기 때문에 문제의 원인을 찾기 어려움
  - ✓ private 정적 멤버 클래스
    - private 정적 멤버 클래스는 흔히 바깥 클래스가 표현하는 객체의 한 부분(구성 요소)을 나타낼 때 사용 (ex. Map.Entry)
    - 엔트리(Entry) 객체들의 메서드들(getKey, getValue, setValue)은 맵을 직접 사용하지는 않음
    - 엔트리를 비정적 멤버 클래스로 표현하는 것은 낭비고 private 정적 멤버 클래스가 가장 적절
    - 멤버 클래스가 공개된 클래스의 public이나 protected 멤버라면 멤버 클래스 역시 공개 API가 되니 향후 릴리스에서 static을 붙이면 하위 호환성이 깨지게 됨

클래스 와 인터페이스

- ❖ 멤버 클래스는 되도록 static 으로
  - ✓ 익명 클래스
    - 익명 클래스에는 이름이 없고 바깥 클래스의 멤버도 아님
    - 멤버와 달리 쓰이는 시점에 선언과 동시에 인스턴스가 만들어지고 코드의 어디서든 만들 수 있음
    - 비정적인 문맥에서 사용될 때만 바깥 클래스의 인스턴스를 참조할 수 있음
    - 정적 문맥에서라도 상수 변수 이외의 정적 멤버는 가질 수 없는데 상수 표현을 위해 초기화된 final 기본 타입과 문자열 필드만 가질 수 있음
    - 선언한 지점에서만 인스턴스를 만들 수 있고 instanceof 검사나 클래스의 이름이 필요한 작업은 수행할 수 없음
    - 여러 인터페이스를 구현할 수 없고 인터페이스를 구현하는 동시에 다른 클래스를 상속할 수도 없음
    - 익명 클래스를 사용하는 클라이언트는 그 익명 클래스가 상위 타입에서 상속한 멤버 외에는 호출할 수 없음
    - 익명 클래스는 로직의 중간에 등장하므로 짧지 않으면 가독성이 떨어짐
    - 람다를 지원하기 전에는 즉석에서 작은 함수 객체나 처리 객체(process object)를 만드는 데 익명 클래스를 주로 사용 (지금은 람다에게 빼앗긴 자리)
    - 익명 클래스의 또 다른 주 쓰임은 정적 팩토리 메서드를 구현할 때

클래스 와 인터페이스

- ❖ 멤버 클래스는 되도록 static 으로
  - ✓ 지역 클래스
    - 지역변수를 선언할 수 있는 곳이면 실질적으로 어디서든 선언할 수 있고 유효범위도 지역변수와 동일
    - 멤버 클래스처럼 이름이 있고 반복해서 사용할 수 있음
    - 익명 클래스처럼 비정적 문맥에서 사용될 때만 바깥 인스턴스를 참조할 수 있음
    - 정적 멤버를 가질 수 없고 가독성을 위해 짧게 작성

클래스 와 인터페이스

- 톱 레벨 클래스는 한 파일에 하나
  - ✓ 하나의 파일에 여러 개 클래스 작성 – Utensil.java

```
public class Main {
    public static void main(String[] args) {
        System.out.println(Utensil.NAME + Dessert.NAME);
    }
}

class Utensil {
    static final String NAME = "pan";
}

class Dessert {
    static final String NAME = "cake";
}
```

클래스 와 인터페이스

- 톱 레벨 클래스는 한 파일에 하나
  - ✓ 하나의 파일에 여러 개 클래스 작성 – Dessert.java

```
class Utensil {
    static final String NAME = "pot";
}

class Dessert {
    static final String NAME = "pie";
}
```

클래스 와 인터페이스

- ❖ 톱 레벨 클래스는 한 파일에 하나
  - ✓ 하나의 파일에 여러 개 클래스 작성
    - javac Main.java Dessert.java 명령으로 컴파일한다면 컴파일 오류가 나고 Utensil과 Dessert 클래스를 중복 정의했다고 알려줄 것
    - 컴파일러는 가장 먼저 Main.java를 컴파일하고 그 안에서(Dessert 참조보다 먼저 나오는) Utensil 참조를 만나면 Utensil.java 파일을 살펴 Utensil과 Dessert를 모두 찾아내게 되고 컴파일러가 두 번째 명령줄 인수로 넘어온 Dessert.java를 처리하려 할 때 같은 클래스의 정의가 이미 있음을 알게 됨
    - javac Main.java나 javac Main.java Utensil.java 명령으로 컴파일하면 Dessert.java 파일을 작성하기 전처럼 pancake를 출력
    - javac Dessert.java Main.java 명령으로 컴파일하면 potpie를 출력
    - 컴파일러에 어느 소스 파일을 먼저 건네느냐에 따라 동작이 달라지는 문제가 발생
    - 해결책으로는 톱 레벨 클래스들(Utensil과 Dessert)을 서로 다른 소스 파일로 분리하면 됨
    - 굳이 여러 톱 레벨 클래스를 한 파일에 담고 싶다면 정적 멤버 클래스를 사용하는 방법을 고려해볼 수 있음
    - 다른 클래스에 딸린 부차적인 클래스라면 정적 멤버 클래스로 만드는 쪽이 일반적으로 더 나을 것인데 읽기 좋고 private로 선언하면 접근 범위도 최소로 관리할 수 있기 때문

Generic

- Raw type 사용 금지
  - ✓ 제네릭 클래스, 제네릭 인터페이스
    - 클래스와 인터페이스 선언에 타입 매개변수(type parameter)가 쓰이면 이를 제네릭 클래스 혹은 제네릭 인터페이스라 함 – List 는 원소의 타입을 나타내는 매개변수 E를 받은 List
    - 제네릭 클래스와 제네릭 인터페이스를 통틀어 제네릭 타입(generic type)이라 함
    - 각각의 제네릭 타입은 일련의 매개변수화 타입(parameterized type)을 정의 – 클래스(혹은 인터페이스) 이름이 나오고 이어서 꺾쇠 괄호 안에 실제 타입 매개변수들을 나열
    - 제네릭 타입을 하나 정의하면 그에 딸린 로 타입(raw type)도 함께 정의 – 로 타입이란 제네릭 타입에서 타입 매개변수를 전혀 사용하지 않을 때
    - 로 타입은 타입 선언에서 제네릭 타입 정보가 전부 지워진 것처럼 동작하는데 이는 제네릭 도입 이전 코드와의 호환을 위한 것

Generic

- Raw type 사용 금지
  - ✓ 로 타입 vs 제네릭 타입
    - List 같은 로 타입은 사용해서는 안되지만 List<Object>처럼 임의 객체를 허용하는 매개변수화 타입은 괜찮음
    - List는 제네릭 타입에서 완전히 발을 뺀 것이고 List<Object>는 모든 타입을 허용한다는 의사를 컴파일러에 명확히 전달한 것
    - 매개변수로 List를 받는 메서드에 List을 넘길 수는 있지만 List<Object>를 받는 메서드에는 넘길 수 없는데 이는 제네릭의 하위 타입 규칙 때문
    - List<String>은 로 타입인 List의 하위 타입이지만 List<Object>의 하위 타입은 아님
    - List<Object> 같은 매개변수화 타입을 사용할 때와 달리 List 같은 로 타입을 사용하면 타입 안정성을 잃게 됨
  - ✓ 제네릭 타입을 쓰고 싶지만 실제 타입 매개변수가 무엇인지 신경 쓰고 싶지 않다면 물음표(?)를 사용 - 제네릭 타입인 Set의 비한정적 와일드카드 타입은 Set<?>인데 이는 어떤 타입이라도 담을 수 있는 가장 범용적인 매개변수화 Set 타입
  - ✓ class 리터럴에는 로 타입을 써야 하는데 자바 명세는 class 리터럴에 매개변수화 타입을 사용하지 못하게 함(배열과 기본 타입은 허용) - List.class, String[].class, int.class는 허용 / List.class, List<?>.class는 허용하지 않음

Generic

- 비 검사 경고를 제거
  - ✓ 컴파일러 경고는 다양한데 비검사 형변환 경고, 비검사 메서드 호출 경고, 비검사 매개변수화 가변인수 타입 경고, 비검사 변환 경고 등이 있음
  - ✓ 할 수 있는 한 모든 비검사 경고를 제거
  - ✓ 비검사 경고를 모두 제거한다면 그 코드는 타입 안전성이 보장되는데 런타임에 ClassCastException이 발생할 일이 없고 의도한대로 잘 동작하리라 확신할 수 있음
  - ✓ 경고를 제거할 수는 없지만 타입 안전하다고 확신할 수 있다면 @SuppressWarnings("unchecked") 애너테이션을 달아서 경고를 숨기도록 하는데 타입 안전함을 검증하지 않은 채 경고를 숨기면 이후에 더 큰 문제가 발생할 수 있으니 확실하게 검증해야 함
  - ✓ @SuppressWarnings
    - @SuppressWarnings 애너테이션은 개별 지역변수 선언부터 클래스 전체까지 어떤 선언에도 추가할 수 있음
    - @SuppressWarnings 애너테이션은 항상 가능한 한 좁은 범위에 적용 – 보통은 변수선언, 아주 짧은 메서드, 생성자
    - 심각한 경고를 놓치는 일이 없도록 해야하며 따라서 절대로 클래스 전체에 적용해서는 안됨

Generic

- ❖ 배열 보다는 리스트를 사용
  - ✓ 공변 과 불공변
    - 배열은 공변(covariant)
      - ◆ Sub가 Super의 하위 타입이라면 배열 Sub[]는 배열 Super[]의 하위 타입이 됨
      - ◆ 공변은 함께 변한다는 뜻
    - 제네릭은 불공변(invariant) - 서로 다른 타입 Type1과 Type2가 있을 때 List<Type1>은 List<Type2>의 하위 타입도 아니고 상위 타입도 아님
    - 타입이 맞지 않으면 배열은 예외를 던지지만 List는 컴파일 오류

```
public class Main {
    public static void main(String[] args) {
        Object[] objectArray = new Long[1];
        objectArray[0] = "타입이 달라 넣을 수 없다."; // ArrayStoreException을 던진다.

        List<Object> ol = new ArrayList<Long>();
        ol.add("타입이 달라 넣을 수 없다.");
    }
}
```

Generic

- 배열 보다는 리스트를 사용
  - ✓ 실체화
    - 배열은 실체화 됨
      - 배열은 런타임에도 자신이 담기로 한 원소의 타입을 인지하고 확인
      - Long 배열에 String을 넣으려 하면 ArrayStoreException이 발생
    - 제네릭은 타입 정보가 런타임에는 소거(erasure)됨 – 원소 타입을 컴파일에만 검사하며 런타임에는 알 수조차 없다는 의미
    - 배열은 제네릭 타입, 매개변수화 타입, 타입 매개변수로 사용할 수 없음 – new List<E>[], new List<String[]>, new E[]와 같이 사용하려고 하면 컴파일할 때 제네릭 배열 생성 오류를 일으킴

Generic

- 이왕이면 제네릭 타입
  - ✓ Object 기반 스택

```
public class StackBasedObject {
    private Object[] elements;
    private int size = 0;
    private static final int DEFAULT_INITIAL_CAPACITY = 16;

    public StackBasedObject() {
        elements = new Object[DEFAULT_INITIAL_CAPACITY];
    }

    public void push(Object e) {
        ensureCapacity();
        elements[size++] = e;
    }

    public Object pop() {
        if (size == 0) {
            throw new EmptyStackException();
        }
        Object result = elements[--size];
        elements[size] = null; // 다 쓴 참조 해제
        return result;
    }
```

Generic

- 이왕이면 제네릭 타입
  - ✓ Object 기반 스택

```
    public boolean isEmpty() {
        return size == 0;
    }

    private void ensureCapacity() {
        if (elements.length == size) {
            elements = Arrays.copyOf(elements, 2 * size + 1);
        }
    }
}
```

Generic

- 이왕이면 제네릭 타입
  - ✓ Generic 기반 스택

```
public class StackBasedGeneric<E> {
    private Object[] elements; // Obejct[] 타입이다!
    private int size = 0;
    private static final int DEFAULT_INITIAL_CAPACITY = 16;

    public StackBasedGeneric() {
        elements = new Object[DEFAULT_INITIAL_CAPACITY];
    }

    public E pop() {
        if (size == 0) {
            throw new EmptyStackException();
        }

        @SuppressWarnings("unchecked") // push에서 E 타입만 허용하므로 이 형변환은
     안전하다.
        E result = (E) elements[--size];

        elements[size] = null; // 다 쓴 참조 해제
        return result;
    }
```

Generic

- 이왕이면 제네릭 타입
  - ✓ Generic 기반 스택

```
    public boolean isEmpty() {
        return size == 0;
    }

    private void ensureCapacity() {
        if (elements.length == size) {
            elements = Arrays.copyOf(elements, 2 * size + 1);
        }
    }
}
```

Generic

- 이왕이면 제네릭 타입
  - ✓ 클라이언트에서 직접 형변환해야 하는 타입보다 제네릭 타입이 더 안전하고 쓰기 편함
  - ✓ 새로운 타입을 설계할 때는 형변환 없이도 사용할 수 있도록 하기 위해서는 제네릭 타입으로 만들어야 할 경우가 많음
  - ✓ 기존 타입 중 제네릭이었어야 하는 것이 있다면 제네릭 타입으로 변경
  - ✓ 제네릭 타입을 사용하면 기존 클라이언트에는 아무 영향을 주지 않으면서 새로운 사용자에게 타입 안전성이라는 편의를 제공해줄 수 있음

Generic

- 이왕이면 제네릭 메서드
  - ✓ 경고

```
public static Set union(Set s1, Set s2) {
        Set result = new HashSet<>();
        result.addAll(s2);
        return result;
}
```

  - ✓ 수정

```
public static <E> Set<E> union(Set<E> s1, Set<E> s2) {
        Set<E> result = new HashSet<>(s1);
        result.addAll(s2);
        return result;
}
```

ENUM

- int 상수 대신 ENUM 사용
  - ✓ 정수 열거 패턴
    - 정수 열거 패턴(int enum pattern)에는 단점이 많은데 타입 안전을 보장할 수 없고 표현력도 좋지 않음
    - 자바는 정수 열거 패턴을 위한 별도 이름공간(namespace)을 지원하지 않음
    - 정수 열거 패턴을 사용한 프로그램은 깨지기 쉬움
    - 평범한 상수를 나열한 것뿐이라 컴파일하면 그 값이 클라이언트 파일에 그대로 새겨지므로 상수의 값이 바뀌면 클라이언트도 반드시 다시 컴파일해야 함
    - 정수 상수는 문자열로 출력하기가 다소 까다로움
    - 디버거로 살펴봐도 단지 숫자로만 보여서 썩 도움이 되지 않음
    - 정수 대신 문자열 상수를 사용하는 변형 패턴도 있음
    - 문자열 열거 패턴(string enum pattern)이라 하는 이 변형은 문자열 상수의 이름 대신 문자열 값을 그대로 하드코딩하게 만들기 때문에 더 좋지 않음

ENUM

- int 상수 대신 ENUM 사용
  - ✓ 열거 타입
    - 자바의 열거 타입은 완전한 형태의 클래스
    - 상수 하나당 자신의 인스턴스를 하나씩 만들어 public static final 필드로 공개
    - 열거 타입은 밖에서 접근할 수 있는 생성자를 제공하지 않으므로 사실상 final로 클라이언트가 인스턴스를 직접 생성하거나 확장할 수 없기 때문에 열거 타입 선언으로 만들어진 인스턴스들은 딱 하나씩만 존재함이 보장됨
    - 싱글턴은 원소가 하나뿐인 열거 타입이라 할 수 있고 열거 타입은 싱글턴을 일반화한 형태라고 볼 수 있음
    - 열거 타입은 컴파일타임 타입 안전성을 제공 - 열거 타입에 다른 타입의 값을 넘기려 하면 컴파일 오류가 발생하는데 타입이 다른 열거 타입 변수에 할당하려 하거나 다른 열거 타입의 값끼리 == 연산자로 비교하려는 것과 같기 때문
    - 열거 타입에는 각자의 이름 공간이 있으므로 이름이 같은 상수도 평화롭게 공존
    - 열거 타입에 새로운 상수를 추가하거나 순서를 바꿔도 다시 컴파일하지 않아도 되는데 공개되는 것이 오직 필드의 이름뿐이라, 상수 값이 클라이언트로 컴파일되어 각인되지 않기 때문
    - 열거 타입은 출력하기 좋은데 열거 타입의 toString 메서드는 출력하기에 적합한 문자열을 내어줌
    - 열거 타입 상수 각각을 특정 데이터와 연결지으려면 생성자에서 데이터를 받아 인스턴스 필드에 저장하면 됨

ENUM

- int 상수 대신 ENUM 사용
  - ✓ 열거 타입
    - 열거 타입은 근본적으로 불변이라 모든 필드는 final
    - 필드를 public으로 선언해도 되지만 private로 두고 별도의 public 접근자 메서드를 두는 게 나음
    - 열거 타입은 자신 안에 정의된 상수들의 값을 배열에 담아 반환하는 정적 메서드인 values를 제공
    - 값들은 순서대로 저장되며 각 열거 타입 값의 toString 메서드는 상수 이름을 문자열로 반환
    - 열거 타입에서 상수를 제거해도 해당 상수를 참조하지 않는 클라이언트에는 아무 영향이 없음
    - 널리 쓰이는 열거 타입은 톱레벨 클래스로 만들고 특정 톱레벨 클래스에서만 쓰인다면 해당 클래스의 멤버 클래스로 생성

ENUM

- ordinal 메서드 대신 인스턴스 필드를 사용
  - ✓ ordinal을 잘못 사용한 예

```
enum Ensemble {
    SOLO, DUET, TRIO, QUARTET, QUINTET, SEXTET, SEPTET, OCTET, NONET,
     DECTET;

    public int numberOfMusicians() {
        return ordinal() + 1;
    }
}
public class Main {
    public static void main(String[] args) {
        System.out.println(Ensemble.DUET.numberOfMusicians());
    }
}
```

ENUM

- ordinal 메서드 대신 인스턴스 필드를 사용
  - ✓ 인스턴스 필드 사용

```
enum Ensemble {
    SOLO(1),
    DUET(2),
    TRIO(3),
    QUARTET(4),
    QUINTET(5),
    SEXTET(6),
    SEPTET(7),
    OCTET(8),
    DOUBLE_QUARTET(8),
    NONET(9),
    DECTET(10),
    TRIPLE_QUARTET(12);

    private final int numberOfMusicians;
    Ensemble(int numberOfMusicians) {
        this.numberOfMusicians = numberOfMusicians;
    }
    public int numberOfMusicians() {
        return numberOfMusicians;
    }
}
```

ENUM

- 비트 필드 대신 EnumSet 사용
  - ✓ 비트 필드 사용

```
class Text {
    public static final int STYLE_BOLD = 1 << 0; // 1
    public static final int STYLE_ITALIC = 1 << 1; // 2
    public static final int STYLE_UNDERLINE = 1 << 2; // 4
    public static final int STYLE_STRIKETHROUGH = 1 << 3; // 8

    // 매개변수 styles는 0개 이상의 STYLE_ 상수를 비트별 OR한 값이다.
    public void applyStyles(int styles) {
        System.out.println(styles);
    }
}

public class Main {
    public static void main(String[] args) {
        Text text = new Text();
        text.applyStyles(Text.STYLE_BOLD | Text.STYLE_ITALIC);
    }
}
```

ENUM

- ❖ 비트 필드 대신 EnumSet 사용
  - ✓ 비트 필드 사용
    - 비트 필드를 사용하면 비트별 연산을 사용해 합집합과 교집합 같은 집합 연산을 효율적으로 수행할 수 있음
    - 비트 필드는 정수 열거 상수의 단점을 그대로 지님
    - 추가로 비트 필드 값이 그대로 출력되면 단순한 정수 열거 상수를 출력할 때보다 해석하기가 훨씬 어려움
    - 비트 필드 하나에 녹아 있는 모든 원소를 순회하기도 까다로움
    - 최대 몇 비트가 필요한지를 API 작성 시 미리 예측하여 적절한 타입(보통은 int나 long)을 선택해야 하는데 API를 수정하지 않고는 비트 수(32비트 or 64비트)를 더 늘릴 수 없기 때문

ENUM

- 비트 필드 대신 EnumSet 사용
  - ✓ 현대적 기법

```
class Text {
    public enum Style {BOLD, ITALIC, UNDERLINE, STRIKETHROUGH}

    // 매개변수 styles는 0개 이상의 STYLE_ 상수를 비트별 OR한 값이다.
    public void applyStyles(Set <Style> styles) {
        System.out.println(styles);
    }
}
public class Main {
    public static void main(String[] args) {
        Text text = new Text();
        text.applyStyles(EnumSet.of(Text.Style.BOLD, Text.Style.ITALIC));
    }
}
```

ENUM

- ❖ 비트 필드 대신 EnumSet 사용
  - ✓ 현대적 기법
    - 비트 필드보다 더 나은 대안으로 EnumSet 클래스가 있음
    - java.util 패키지의 EnumSet 클래스는 열거 타입 상수의 값으로 구성된 집합을 효과적으로 표현
    - Set 인터페이스를 완벽히 구현하며 타입 안전하고 다른 어떤 Set 구현체 와도 함께 사용 가능
    - EnumSet의 내부는 비트 벡터로 구현되어있는데 원소가 총 64개 이하라면 대부분의 경우에 EnumSet 전체를 long 변수 하나로 표현하여 비트 필드에 비견되는 성능을 보여줌
    - removeAll과 retainAll 같은 대량 작업은 (비트 필드를 사용할 때 쓰는 것과 같은) 비트를 효율적으로 처리할 수 있는 산술 연산을 써서 구현
    - 난해한 작업들은 EnumSet이 다 처리해주기 때문에 비트를 직접 다룰 때 겪는 흔한 오류들로부터 해방

Method

- 매개변수가 유효한지 검사
  - ✓ 메서드와 생성자 대부분은 입력 매개변수의 값이 특정 조건(불변식)을 만족했을 때 제대로 동작해야 하고 이런 제약은 반드시 문서화해야 하며 메서드 몸체가 시작되기 전에 검사해야 함
  - ✓ 오류는 가능한 한 빨리 (발생한 곳에서) 잡아야 한다 는 일반 원칙의 한 사례이기도 하고 오류를 발생한 즉시 잡지 못하면 해당 오류를 감지하기 어려워지고 감지하더라도 오류의 발생 지점을 찾기 어려워짐
  - ✓ 매개변수 검사를 제대로 하지 못하면 문제가 생길 수 있음
    - 메서드가 수행되는 중간에 모호한 예외를 던지며 실패할 수 있음
      - 메서드는 잘 수행되었지만 잘못된 결과를 반환하는 경우도 있는데 미래에 이 메서드와는 관련 없는 오류를 낼 위험이 있음
      - 매개변수 검사에 실패하면 실패 원자성(failure atomicity)을 어기는 결과를 낳을 수 있음
    - public과 protected 메서드는 매개변수 값이 잘못됐을 때 던지는 예외를 문서화
      - 매개변수의 제약을 문서화한다면 그 제약을 어겼을 때 발생하는 예외도 함께 기술해야 함

Method

- 매개변수가 유효한지 검사
  - ✓ 문서화 예시

```
/**
 * (현재 값 mod m) 값을 반환한다. 이 메서드는
 * 항상 음이 아닌 BigInteger를 반환한다는 점에서 remainder 메서드와 다르다.
 * @param m 계수(양수여야 한다)
 * @return 현재 값 mod m
 * @throws ArithmeticException m이 0보다 작거나 같으면 발생한다.
 */
public BigInteger mod(BigInteger m) {
    if (m.signum() <= 0) {
        throw new ArithmeticException("계수(m)는 양수여야 합니다. " + m);
    }
    // 계산 수행
    return null; // 편의상 null을 리턴했다.
}
```

Method

- 매개변수가 유효한지 검사
  - ✓ 단언을 이용한 매개변수 유효성 검증

```
private static void sort(long a[], int offset, int length) {
        assert a != null;
        assert offset >= 0 && offset <= a.length;
        assert length >= 0 && length <= a.length - offset;
        // 계산 수행
}
```

- 단언문들은 자신이 단언한 조건이 무조건 참이라고 선언하는데 참이 아닐 경우 AssertionError가 발생
- 단언문은 일반적인 유효성 검사와 다름
  - 실패하면 AssertionError를 던짐
  - 런타임에 아무런 효과도 아무런 성능 저하도 없음(단 java를 실행할 때 명령줄에서 -ea 혹은 --enableassertions 플래그 설정하면 런타임에 영향)

Method

- ❖ 매개변수가 유효한지 검사
  - ✓ 나중에 쓰기 위해 저장하는 매개변수의 유효성을 검사
    - 메서드가 직접 사용하지는 않으나 나중에 쓰기 위해 저장하는 매개변수는 특히 더 신경 써서 검사해야 해야 하는데 한참 뒤에서야 문제가 발생할 수 있기 때문
    - 생성자는 나중에 쓰려고 저장하는 매개변수의 유효성을 검사하라는 원칙의 특수한 사례로 생성자 매개변수의 유효성 검사는 클래스 불변식을 어기는 객체가 만들어지지 않게 하는 데 꼭 필요
    - 메서드 몸체 실행 전에 매개변수 유효성을 검사해야 한다 는 규칙에도 예외가 있는데 유효성 검사 비용이 지나치게 높거나 상용적이지 않을 때 혹은 계산 과정에서 암묵적으로 검사가 수행되는 경우
  - ✓ 암묵적 유효성 검사에 너무 의존하면 실패 원자성을 해칠 수 있으니 주의
    - API 문서에 정의된 예외와 다른 예외가 발생할 경우
    - 계산 중 잘못된 매개변수 값을 사용해 발생한 예외가 API 문서에서 던지기로 한 예외와 다를 수 있는데 이럴 때는 예외 번역(exception translate, 또는 예외 전환) 관용구를 사용하여 API 문서에 기재된 예외로 번역

Method

- 적절한 시점에 방어적 복사본을 생성
  - ✓ 자바는 안전한 언어인데 네이티브 메서드를 사용하지 않으니 C, C++ 같이 안전하지 않은 언어에서 흔히 보는 버퍼 오버런, 배열 오버런, 와일드 포인터 같은 메모리 충돌 오류에서 안
  - ✓ 자바로 작성한 클래스는 시스템의 다른 부분에서 무슨 짓을 하든 그 불변식이 지켜지는데 메모리 전체를 하나의 거대한 배열로 다루는 언어에서는 누릴 수 없는 강점
  - ✓ 방어적 프로그래밍
    - 자바라 해도 다른 클래스로부터의 침범을 아무런 노력 없이 다 막을 수 있는 것은 아님
    - 클라이언트가 불변식을 깨뜨리려한다고 가정하고 방어적으로 프로그래밍
    - 악의적인 공격으로부터 방어하지 못하거나 또는 실수로 클래스를 오작동하게 만들 수도 있음
    - 어떤 객체든 그 객체의 허락 없이는 외부에서 내부를 수정하는 일은 불가능하지만 이는 주의를 기울여야 보장할 수 있음

Method

- 적절한 시점에 방어적 복사본을 생성
  - ✓ Period 내부 공격

```
final class Period {
    private final Date start;
    private final Date end;

    public Period(Date start, Date end) {
        if (start.compareTo(end) > 0) {
            throw new IllegalArgumentException(start + "가 " + end + "보다 늦다.");
        }
        this.start = start;
        this.end = end;
    }

    public Date start() {
        return start;
    }

    public Date end() {
        return end;
    }
}
```

Method

- 적절한 시점에 방어적 복사본을 생성
  - ✓ Period 내부 공격

```
public class Main{
    public static void main(String [] args){
        Date start = new Date();
        Date end = new Date();
        Period period = new Period(start, end);
        end.setYear(78); // period의 내부를 수정했다!
    }
}
```

Method

- 적절한 시점에 방어적 복사본을 생성
  - ✓ Period 내부 공격
    - 자바 8 이후로는 쉽게 해결할 수 있는데 Date 대신 불변인 Instant를 사용하면 됨(혹은 LocalDateTime이나 ZonedDateTime을 사용해도 가능)
    - Date는 낡은 API이니 새로운 코드를 작성할 때는 더 이상 사용하면 안됨
    - 외부 공격으로부터 Period 인스턴스의 내부를 보호하려면 생성자에서 받은 가변 매개변수 각각을 방어적으로 복사(defensive copy)해야 한 다음 Period 인스턴스 안에서는 원본이 아닌 복사본을 사용

Method

- 적절한 시점에 방어적 복사본을 생성
  - ✓ Period 방어적 복사본을 이용하는 클래스로 수정

```
final class Period {
    private final Date start;
    private final Date end;

    public Period(Date start, Date end) {
        this.start = new Date(start.getTime());
        this.end = new Date(end.getTime());
        if (start.compareTo(end) > 0) {
            throw new IllegalArgumentException(start + "가 " + end + "보다 늦다.");
        }
    }

    public Date start() {
        return start;
    }

    public Date end() {
        return end;
    }
}
```

Method

- 적절한 시점에 방어적 복사본을 생성
  - ✓ Period 방어적 복사본을 이용하는 클래스로 수정
    - 매개변수의 유효성을 검사하기 전에 방어적 복사본을 만들고 이 복사본으로 유효성을 검사
    - 순서가 부자연스러워 보일 수 있으나 반드시 이렇게 작성
    - 멀티 스레딩 환경이라면 원본 객체의 유효성을 검사한 후 복사본을 만드는 그 찰나의 취약한 순간에 다른 스레드가 원본 객체를 수정할 위험이 있기 때문으로 방어적 복사를 매개변수 유효성 검사 전에 수행하면 이런 위험에서 해방될 수 있으며 이를 검사시점/사용시점(time-of-check/time-of-use) 공격 혹은 TOCTOU 공격이라 함
    - Date는 final이 아니므로 clone이 Date가 정의한 게 아닐 수 있는데 clone이 악의를 가진 하위 클래스의 인스턴스를 반환할 수도 있기 때문에 Date의 clone 메서드를 사용하지 않음
    - 매개변수가 제3자에 의해 확장될 수 있는 타입이라면 방어적 복사본을 만들 때 clone을 사용해서는 안됨

Method

- 적절한 시점에 방어적 복사본을 생성
  - ✓ 생성자가 아닌 경우
    - 생성자와 달리 접근자 메서드에서는 방어적 복사에 clone을 사용해도 되는데 Period가 가지고 있는 Date 객체는 java.util.Date임이 확실하기 때문 (신뢰할 수 없는 하위 클래스가 아님)
    - 그렇더라도 인스턴스를 복사하는 데는 일반적으로 생성자나 정적 팩터리를 쓰는 것을 권장
    - 매개변수를 방어적으로 복사하는 목적에는 불변 객체를 만들기 위해서만은 아님
    - 클라이언트가 제공한 객체의 참조를 내부의 자료구조에 보관해야 할 때면 항시 그 객체가 잠재적으로 변경될 수 있는지를 생각해야함
    - 변경될 수 있는 객체라면 그 객체가 클래스에 넘겨진 뒤 임의로 변경되어도 그 클래스가 문제없이 동작할지를 따지고 확신할 수 없다면 복사본을 만들어 저장
    - 클래스가 불변이든 가변이든 가변인 내부 객체를 클라이언트에 반환할 때는 반드시 심사숙고해야 하며 안심할 수 없다면 방어적 복사본을 반환
    - 길이가 1 이상인 배열은 무조건 가변이므로 내부에서 사용하는 배열을 클라이언트에 반환할 때는 항상 방어적 복사를 수행
    - 자바 8 이상에서는 Instant(혹은 LocalDateTime이나 ZonedDateTime)를 사용하도록 권장
    - 이전 버전의 자바를 사용한다면 Date 참조 대신 Date.getTime()이 반환하는 long 정수를 사용하는 방법을 써도 됨

Method

- 적절한 시점에 방어적 복사본을 생성
  - ✓ 방어적 복사의 생략
    - 방어적 복사에는 성능 저하가 따르고 항상 쓸 수 있는 것도 아니며(같은 패키지에 속하는 등의 이유로) 호출자가 컴포넌트 내부를 수정하지 않으리라 확신하면 방어적 복사를 생략할 수 있음
    - 호출자에서 해당 매개변수나 반환값을 수정하지 말아야 함을 명확히 문서화하는 게 좋음
    - 다른 패키지에서 사용한다고 해서 넘겨받은 가변 매개변수를 항상 방어적으로 복사해 저장해야 하는 것은 아님
    - 메서드나 생성자의 매개변수로 넘기는 행위가 그 객체의 통제권을 명백히 이전함을 뜻하기도 함
    - 통제권을 이전하는 메서드를 호출하는 클라이언트는 해당 객체를 더 이상 직접 수정하는 일이 없다고 약속해야 함
    - 클라이언트가 건네주는 가변 객체의 통제권을 넘겨받는다고 기대하는 메서드나 생성자에서도 그 사실을 확실히 문서에 기재
    - 통제권을 넘겨 받기로 한 메서드, 생성자를 가진 클래스들은 취약하기 때문에 해당 클래스와 그 클라이언트가 상호 신뢰할 수 있을 때 혹은 불변식이 깨지더라도 그 영향이 오직 호출한 클라이언트로 국한될 때로 한정해서 방어적 복사를 생략해도 됨

Method

- ❖ 메서드 시그니처를 신중히 설계
  - ✓ 메서드 이름을 신중히
    - 항상 표준 명명 규칙을 따라야 하는데 이해할 수 있고 같은 패키지에 속한 다른 이름들과 일관되게 짓는 게 최우선 목표
    - 긴 이름은 피하도록 하는데 애매하면 자바 라이브러리의 API 가이드를 참조
  - ✓ 편의 메서드를 너무 많이 만들면 안됨
    - 메서드가 너무 많은 클래스는 익히고 사용하고 문서화하고 테스트하고 유지보수하기 어려움
    - 클래스나 인터페이스는 자신의 각 기능을 완벽히 수행하는 메서드로 제공
    - 자주 쓰일 경우에만 별도의 약칭 메서드를 두도록 하는데 확신이 서지 않으면 만들지 않도록
  - ✓ 매개변수 목록은 짧게 유지
    - 4개가 넘어가면 매개변수를 전부 기억하기가 쉽지 않음
    - 같은 타입의 매개변수 여러 개가 연달아 나오는 경우가 특히 해로운데 사용자가 매개변수 순서를 기억하기도 어렵고 실수로 순서를 바꿔 입력해도 그대로 컴파일되고 실행되고 의도와 다르게 동작
    - 짧게 유지하는 방법
      - ◆ 여러 메서드로 분할
      - ◆ 매개변수를 묶어주는 DTO 클래스를 활용
      - ◆ 객체 생성에 사용한 빌더 패턴을 메서드 호출에 응용

Method

- 메서드 시그니처를 신중히 설계
  - ✓ 매개변수의 타입으로는 클래스보다는 인터페이스
    - 매개변수로 적합한 인터페이스가 있다면 (이를 구현한 클래스가 아닌) 그 인터페이스를 직접 사용
    - 메서드에 HashMap을 넘기지 않고 Map을 사용하면 다른 Map 구현체도 인수로 건넬 수 있음
    - 인터페이스 대신 클래스를 사용하면 클라이언트에게 특정 구현체만 사용하도록 제한하는 꼴이며 혹시라도 입력 데이터가 다른 형태로 존재한다면 명시한 특정 구현체의 객체로 옮겨 담느라 비싼 복사 비용을 치러야 함
  - ✓ boolean보다는 원소 2개짜리 열거 타입
    - 메서드 이름상 boolean을 받아야 의미가 더 명확할 때는 예외
    - 열거 타입을 사용하면 코드를 읽고 쓰기가 더 쉬워지며 나중에 선택지를 추가하기도 쉬움
    - 열거 타입을 사용하면 개별 열거 타입 상수 각각에 특정 동작을 수행하는 메서드를 정의해 둘 수도 있음

Method

- Overloading
  - ✓ 잘못된 Overloading

```
class CollectionClassifier{
    public static String classify(Set<?> set){
        return "집합";
    }

    public static String classify(List<?> list){
        return "리스트";
    }

    public static String classify(Collection<?> collection){
        return "기타";
    }
}
```

Method

- Overloading
  - ✓ 잘못된 Overloading

```
public class Main{
    public static void main(String [] args){
        Collection <?> [] collections = {
                new HashSet<String>(),
                new ArrayList<String>(),
                new HashMap<String, String>().values()
        };

        for(Collection <?> c : collections){
            System.out.println(CollectionClassifier.classify(c));
        }
    }
}
```

Method

- 가변 인수는 신중히 사용
  - ✓ 인수를 0개만 넣어 호출했을 때 런타임에 실패
  - ✓ args 유효성 검사를 명시적으로 해야하고 min의 초깃값을 Integer.MAX_VALUE로 설정하지 않고는 더 명료한 for-each문도 사용할 수 없음
  - ✓ 성능에 민감한 상황이라면 가변 인수가 걸림돌이 될 수도 있는데 가변 인수 메서드는 호출될 때마다 배열을 새로 하나 할당하고 초기화
  - ✓ 최소값을 찾아주는 메서드

```
static int min(int... args) {
        if (args.length == 0)
            throw new IllegalArgumentException("인수가 1개 이상 필요합니다.");
        int min = args[0];
        for (int i = 1; i < args.length; i++)
            if (args[i] < min)
                min = args[i];
        return min;
}
```

Method

- null이 아닌 빈 컬렉션이나 배열을 반환
  - ✓ 빈 컨테이너를 할당하는 데도 비용이 드니 null을 반환하는 쪽이 낫다는 주장도 있지만 대부분 이 비용은 무시 가능

Method

- ❖ 옵셔널 반환은 신중히
  - ✓ 자바 8 이전의 메서드가 특정 조건을 반환할 수 없을 때
    - 예외 반환
    - null 반환
  - ✓ 이전 방식의 문제점
    - 예외는 진짜 예외적인 상황에서만 사용해야 하는데 예외를 생성할 때 스택 추적 전체를 캡처하므로 비용이 비쌈
    - 별도의 null 처리 코드를 추가해야 하는데 null을 반환하게 한 실제 원인과는 상관 없는 코드에서
  - ✓ 자바 8부터는 Optional<T>가 생겼는데 null이 아닌 T타입 참조 하나를 담거나 혹은 아무것도 담지 않을 수도 있는데 옵셔널은 원소를 최대 1개 가질 수 있는 불변 컬렉션
  - ✓ 보통은 T를 반환해야 하지만 특정 조건에서는 아무것도 반환하지 않아야 할 때 T 대신 Optional<T>를 반환하도록 선언하면 되는데 옵셔널을 반환하는 메서드는 예외를 던지는 메서드보다 유연하고 사용하기 쉬우며 null을 반환하는 메서드보다 오류 가능성이 작음

Method

- ❖ 옵셔널 반환은 신중히
  - ✓ 컬렉션에서 최대값을 구해주는 메서드

```
public static <E extends Comparable<E>> Optional<E> max(Collection<E> c) {
        if (c.isEmpty())
            return Optional.empty();
        E result = null;
        for (E e : c)
            if (result == null || e.compareTo(result) > 0)
                result = Objects.requireNonNull(e);
        return Optional.of(result);
}
```

- 빈 옵셔널은 Optional.empty() 로 만들고 값이 든 옵셔널은 Optional.of(value)로 생성
- Optional.of(value)에 null을 넣으면 NullPointerException을 던지니 주의
- null 값도 허용하는 옵셔널을 만들려면 Optional.ofNullable(value)를 사용하면 되는데 옵셔널을 반환하는 메서드에서는 절대 null을 반환을 하면 안됨
- null 반환이나 예외 반환 대신 옵셔널 쓰는 이유는 반환 값이 없을 수도 있음을 API 사용자에게 명확히 알려준다라는 점
- 메서드가 옵셔널을 반환한다면 클라이언트는 값을 받지 못했을 때 취할 행동을 선택해야 하는데 그 중 하나는 기본값을 설정하는 방법

Method

- 옵셔널 반환은 신중히
  - ✓ 항상 옵셔널을 써야하는건 아니며 컬렉션, 스트림, 배열, 옵셔널 같은 컨테이너 타입은 옵셔널로 감싸면 안되는데 빈 Optional<List<T>>>를 반환하기보다는 빈 List<T>를 반환하는게 좋음
  - ✓ 결과가 없을 수 있으며 클라이언트가 이 상황을 특별하게 처리해야 한다면 Optional<T>를 반환
  - ✓ Optional도 엄연히 새로 할당 및 초기화해야 하는 객체이고 그 안에서 값을 꺼내려면 메서드를 호출해야 하니 성능이 중요한 상황이면 안 맞을 수도 있음
  - ✓ 박싱된 기본 타입을 담는 옵셔널은 기본 타입 자체보다 무겁기 때문에 OptionalInt, OptionalLong, OptionalDouble과 같은 것들이 있으니 박싱된 기본 타입을 담은 옵셔널을 반환하면 안됨
  - ✓ 옵셔널은 맵의 값으로 사용하면 안되는데 그러면 키가 없다는 사실을 나타내는 경우가 두 가지가 되는데 하나는 키 자체가 없거나 다른 하나는 키는 있지만 속이 빈 옵셔널 인 경우
  - ✓ 옵셔널을 컬렉션의 키, 값, 원소내 배열의 원소로 사용하는게 적절한 상황은 거의 없음

Method

- 공개된 API 요소에는 항상 문서화 주석을 작성
  - ✓ 자바독은 소스코드 파일에서 문서화 주석이라는 특수한 형태로 기술된 설명을 추려 API 문서로 변환
  - ✓ 자바 버전에 따라 자바독 태그도 발전
    - 자바 5 → @literal, @code
    - 자바 6 → @implSpec
    - 자바 9 → @index
  - ✓ API를 올바로 문서화하려면 공개된 모든 클래스, 인터페이스, 메서드, 필드 선언에 문서화 주석을 달아야 하며 직렬화할 수 있는 클래스라면 직렬화 형태에 관해서도 적어야 함
  - ✓ 메서드용 문서화 주석에는 해당 메서드와 클라이언트 사이의 규약을 명료하게 기술해야 함
    - how가 아닌 what을 기술
    - 전제 조건(precondition)을 모두 나열
    - 메서드가 성공적으로 수행된 후에 만족해야 하는 사후 조건도 모두 나열
    - 전제조건은 @throws 태그로 비검사 예외를 선언하여 암시적으로 기술
    - @param 태그를 이용해 그 조건에 영향 받는 매개변수에 기술
    - 관례상 @param 태그와 @return 태그의 설명은 해당 매개변수가 뜻하는 값이나 반환값을 설명하는 명사구를 사용하는데 드물게는 명사구 대신 산술 표현식을 쓰기도 함

Method

- 공개된 API 요소에는 항상 문서화 주석을 작성
  - ✓ BigInteger의 메서드 주석

```
/**
 * Returns a BigInteger whose value is {@code (this * val)}.
 *
 * @implNote An implementation may offer better algorithmic
 * performance when {@code val == this}.
 *
 * @param  val value to be multiplied by this BigInteger.
 * @return {@code this * val}
 */
public BigInteger multiply(BigInteger val) {
    return multiply(val, false);
}
```

Method

- 공개된 API 요소에는 항상 문서화 주석을 작성
  - ✓ 메서드와 생성자의 요약 설명은 해당 메서드와 생성자의 동작을 설명하는 (주어가 없는) 동사구
  - ✓ 클래스, 인터페이스, 필드의 요약 설명은 대상을 설명하는 명사절
  - ✓ 자바 9부터는 자바독이 생성한 HTML 문서에 검색(색인) 기능이 추가되어 광대한 API 문서들을 한결 수월
  - ✓ 제네릭 타입이나 제네릭 메서드를 문서화할 때는 모든 타입 매개변수에 주석을 달아야 함
  - ✓ 열거 타입을 문서화할 때는 상수들에도 주석을 달아야 함
  - ✓ 애너테이션 타입을 문서화할 때는 멤버들에도 모두 주석을 달아야 함
  - ✓ API 문서화에서 자주 누락되는 설명이 두 가지 있으니 바로 스레드 안전성과 직렬화 가능성인데 클래스 혹은 정적 메서드가 스레드 안전하든 그렇지 않든 스레드 안전 수준을 반드시 API 설명에 포함해야 함

Method

- 가변 인수는 신중히 사용
  - ✓ 최소값을 찾아주는 메서드 – 수정

```
static int min(int firstArg, int... remainingArgs) {
        int min = firstArg;
        for (int arg : remainingArgs)
            if (arg < min)
                min = arg;
        return min;
}
```

Lambda & Stream

- 익명 클래스 보다는 람다
  - ✓ 익명 클래스를 이용한 List 정렬

```
public class Main {
    public static void main(String[] args) {
        List<String> words = new ArrayList<>();
        words.add("Javascript");
        words.add("Java");
        words.add("C++");
        words.add("Python");
        words.add("Go");

        Collections.sort(words, new Comparator<String>() {
            @Override
            public int compare(String o1, String o2) {
                return Integer.compare(o1.length(), o2.length());
            }
        });
        System.out.println(words);
    }
}
```

Lambda & Stream

- 익명 클래스 보다는 람다
  - ✓ 람다를 이용한 List 정렬

```
public class Main {
    public static void main(String[] args) {
        List<String> words = new ArrayList<>();
        words.add("Javascript");
        words.add("Java");
        words.add("C++");
        words.add("Python");
        words.add("Go");

        Collections.sort(words, (s1, s2) -> Integer.compare(s1.length(), s2.length()));
        System.out.println(words);
    }
}
```

Lambda & Stream

- ❖ 익명 클래스 보다는 람다
  - ✓ 람다 사용시 고려할 점
    - 메서드나 클래스와 달리 람다는 이름이 없고 문서화도 할 수 없기 때문에 코드 자체로 동작이 명확히 설명되지 않거나 코드 줄 수가 많아지면 람다를 쓰지 말아야 함
    - 람다는 한 줄일 때 가장 좋고 길어야 세 줄안에 끝내는 게 좋은데 세 줄을 넘어가면 가독성이 심하게 나빠짐
    - 람다가 길거나 읽기 어렵다면 더 간단히 줄여보거나 람다를 쓰지 않는 쪽으로 리팩터링
    - 람다는 함수형 인터페이스에서만 쓰임
    - 추상 클래스의 인스턴스를 만들 때 람다를 쓸 수 없으니 익명 클래스를 사용
    - 추상 메서드가 여러 개인 인터페이스의 인스턴스를 만들 때도 익명 클래스를 사용
    - 람다는 자신을 참조할 수 없음
    - 람다에서의 this 키워드는 바깥 인스턴스를 가리킴
    - 익명 클래스에서의 this는 익명 클래스의 인스턴스 자신
    - 함수 객체가 자신을 참조해야 한다면 반드시 익명 클래스를 사용

Lambda & Stream

- 람다보다는 메서드 참조 사용
  - ✓ 메서드 참조 와 람다

```
public class Main {
    public static void main(String[] args) {
        List<String> words = new ArrayList<>();
        words.add("Javascript");
        words.add("Java");
        words.add("C++");
        words.add("Python");
        words.add("Go");

        //람다
        words.stream().forEach(word -> System.out.println(word));

        //메서드 참조
        words.stream().forEach(System.out::println);
    }
}
```

Lambda & Stream

- ❖ 람다보다는 메서드 참조 사용
  - ✓ 메서드 참조 유형
    - 정적 메서드를 가리키는 메서드 참조하는 유형
    - 수신 객체(receiving object; 참조 대상 인스턴스)를 특정하는 한정적(bound) 인스턴스 메서드 참조: 한정적 참조는 근본적으로 정적 참조와 비슷한데 함수 객체가 받는 인수와 참조되는 메서드가 받는 인수가 동일
    - 수신 객체를 특정하지 않는 비한정적(unbound) 인스턴스 메서드 참조.
      - ◆ 비한정적 참조에서는 함수 객체를 적용하는 시점에 수신 객체를 알려줌
      - ◆ 이를 위해 수신 객체 전달용 매개변수가 매개변수 목록의 첫 번째로 추가되며 그 뒤로는 참조되는 메서드 선언에 정의된 매개변수들이 뒤따름
      - ◆ 비한정적 참조는 주로 스트림 파이프라인에서의 매핑과 필터 함수에 사용
    - 클래스 생성자를 가리키는 메서드 참조 – 생성자 참조는 팩터리 객체로 사용
    - 배열 생성자를 가리키는 메서드 참조

Lambda & Stream

- ❖ 람다보다는 메서드 참조 사용
  - ✓ 메서드 참조 유형
    - 정적
      - Integer::parseInt
      - str -> Integer.parseInt(str)
    - 한정적(인스턴스)
      - Instant.now::isAfter
      - Instant then = Instant.now(); t -> then.isAfter(t)
    - 비한정적(인스턴스)
      - String::toLowerCase
      - str -> str.toLowerCase()
    - 클래스 생성자
      - TreeMap<K, V>::new
      - () -> new TreeMap<K, V>()
    - 배열 생성자
      - int[]:new
      - len -> new int[len]

Lambda & Stream

- 표준 함수형 인터페이스 사용
  - ✓ 자바가 람다를 지원하면서 상위 클래스의 기본 메서드를 재정의해 원하는 동작을 구현하는 템플릿 메서드 패턴의 매력이 크게 줄어들었는데 같은 효과의 함수 객체를 받는 정적 팩터리나 생성자를 제공하는 것으로 대체할 수 있기 때문
  - ✓ 표준 함수형 인터페이스
    - Operator 인터페이스는 인수가 1개인 UnaryOperator와 2개인 BinaryOperator로 나뉘며 반환값과 인수의 타입이 같은 함수
    - Predicate 인터페이스는 인수 하나를 받아 boolean을 반환하는 함수
    - Function 인터페이스는 인수와 반환 타입이 다른 함수
    - Supplier 인터페이스는 인수를 받지 않고 값을 반환(혹은 제공)하는 함수
    - Consumer 인터페이스는 인수를 하나 받고 반환값은 없는 인수를 소비하는 함수

Lambda & Stream

- ❖ 스트림은 주의해서 사용
  - ✓ 스트림 API
    - 스트림 API는 다량의 데이터 처리 작업을 위해 자바 8에 추가됨
    - 스트림(stream)은 데이터 원소의 유한 혹은 무한 시퀀스(sequence)를 의미
    - 스트림 파이프라인은 이 원소들로 수행하는 연산 단계를 표현하는 개념
    - 스트림 안의 데이터 원소들은 객체 참조나 기본 타입 값
    - 기본 타입 값으로는 int, long, double 이렇게 세 가지를 지원

Lambda & Stream

- ❖ 스트림은 주의해서 사용
  - ✓ 스트림 파이프라인
    - 스트림 파이프라인은 소스 스트림에서 시작해 종단 연산(terminal operation)으로 끝
    - 스트림의 시작과 종단 연산의 사이에 하나 이상의 중간 연산(intermediate operation)이 있을 수 있음
    - 중간 연산은 스트림을 어떠한 방식으로 변환(transform)
    - 스트림 파이프라인은 지연 평가(lazy evaluation)
    - 평가는 종단 연산이 호출될 때 이뤄지며 종단 연산에 쓰이지 않는 데이터 원소는 계산에 쓰이지 않음
    - 종단 연산이 없는 스트림 파이프라인은 아무 일도 하지 않는 명령어인 no-op과 같으므로 종단 연산은 필수
    - 스트림 API는 메서드 연쇄를 지원하는 플루언트 API(fluent API)
    - 파이프라인 하나를 구성하는 모든 호출을 연결하여 단 하나의 표현식으로 완성할 수 있으며 파이프라인 여러 개를 연결해 표현식 하나로 만들 수도 있음
    - 기본적으로 스트림 파이프라인은 순차적으로 수행
    - 파이프라인을 병렬로 실행하려면 파이프라인을 구성하는 스트림 중 하나에서 parallel 메서드를 호출해주기만 하면 되는데 다만 효과를 볼 수 있는 상황은 많지 않음

Lambda & Stream

- ❖ 스트림은 주의해서 사용
  - ✓ 스트림이 안성맞춤인 일
    - 원소들의 시퀀스를 일관되게 변환하는 경우
    - 원소들의 시퀀스를 필터링
    - 원소들의 시퀀스를 하나의 연산을 사용해 결합(더하기, 연결하기, 최솟값 구하기 등)
    - 원소들의 시퀀스를 컬렉션에 모으기(아마도 공통된 속성을 기준으로 묶어가며)
    - 원소들의 시퀀스에서 특정 조건을 만족하는 원소를 찾는 경우
  - ✓ 스트림으로 처리하기 어려운 일
    - 한 데이터가 파이프라인의 여러 단계(stage)를 통과할 때 이 데이터의 각 단계에서의 값들에 동시에 접근하는 것은 처리하기 어려운데 스트림 파이프라인은 일단 한 값을 다른 값에 매핑하고 나면 원래의 값은 잃는 구조이기 때문

Lambda & Stream

- 스트림에서는 순수 함수를 사용
  - ✓ 스트림 패러다임의 핵심은 계산을 일련의 변환(transformation)으로 재구성하는 부분
  - ✓ 각 변환 단계는 가능한 한 이전 단계의 결과를 받아 처리하는 순수 함수여야 함
  - ✓ 순수 함수란 오직 입력만이 결과에 영향을 주는 함수
  - ✓ 다른 가변 상태를 참조하지 않고, 함수 스스로도 다른 상태를 변경하지 않음
  - ✓ 스트림 연산에 건네는 함수 객체는 모두 부작용(side effect)이 없어야 함

Lambda & Stream

- 스트림 병렬화는 주의해서 적용
  - ✓ 메르센 소수 구하기

```
public class Main {
    public static void main(String[] args) {
        primes().map(prime -> TWO.pow(prime.intValue()).subtract(ONE))
                .filter(mersenne -> mersenne.isProbablePrime(50))
                .limit(20)
                .forEach(System.out::println);
    }

    static Stream<BigInteger> primes() {
        return Stream.iterate(TWO, BigInteger::nextProbablePrime);
    }
}
```

Lambda & Stream

- ❖ 스트림 병렬화는 주의해서 적용
  - ✓ 메르센 소수 구하기
    - 위 코드는 새로 메르센 소수를 찾을 때마다 그 전 소수를 찾을 때보다 두 배 정도 더 오래 걸림
    - 19번째 계산까지 마치고 마지막 20번째 계산이 수행되는 시점에 한가한 CPU 코어들이 병렬로 21번째 ... 등의 메르센 소수를 찾는 작업을 시작하는데 20번째 계산이 끝나도 이 계산들은 끝나지 않아서 결과적으로 자동 병렬화 알고리즘을 마비시킴
    - 환경이 아무리 좋더라도 데이터 소스가 Stream.iterate거나 중간 연산으로 limit를 쓰면 파이프라인 병렬화로는 성능 개선을 기대할 수 없음
    - 파이프라인 병렬화는 limit를 다룰 때 CPU 코어가 남는다면 원소를 몇 개 더 처리한 후 제한된 개수 이후의 결과를 버려도 아무런 해가 없다고 가정
    - 스트림 파이프라인을 마구잡이로 병렬화하면 안되며 성능이 오히려 나빠질 수도 있음
    - 대체로 스트림의 소스가 ArrayList, HashMap, HashSet, ConcurrentHashMap의 인스턴스거나 배열, int 범위, long 범위일 때 병렬화의 효과가 가장 좋은데 이 자료구조들은 모두 데이터를 원하는 크기로 정확하고 손쉽게 나눌 수 있어서 일을 다수의 스레드에 분배하기 좋다는 특징이 있음
    - 나누는 작업은 Spliterator가 담당하며 Spliterator 객체는 Stream이나 Iterable의 spliterator 메서드로 얻어올 수 있음
    - 이 자료 구조들은 원소들을 순차적으로 실행할 때의 참조 지역성(locality of reference)이 뛰어난데 이웃한 원소의 참조들이 메모리에 연속해서 저장되어 있다는의미

Lambda & Stream

- ❖ 스트림 병렬화는 주의해서 적용
  - ✓ 최적화
    - 스트림 병렬화는 오직 성능 최적화의 수단일 뿐인데 다른 최적화와 마찬가지로 변경 전후로 반드시 성능을 테스트하여 병렬화를 사용할 가치가 있는지 확인
    - 보통은 병렬 스트림 파이프라인도 공통의 포크 조인 풀에서 수행되므로(같은 스레드 풀을 사용하므로) 잘못된 파이프라인 하나가 시스템의 다른 부분의 성능에까지 악영향을 줄 수 있음
    - 스트림 병렬화가 효과를 보는 경우는 많지 않으나 조건이 잘 갖춰지면 parallel 메서드 호출 하나로 거의 프로세서 코어 수에 비례하는 성능 향상을 얻을 수 있음

Lambda & Stream

- 스트림 병렬화는 주의해서 적용
  - ✓ 최적화

```
ublic class Main {
    public static void main(String[] args) {
        long start = System.currentTimeMillis();
        long count = LongStream.rangeClosed(2, 10000000L)
                .mapToObj(BigInteger::valueOf)
                .filter(i -> i.isProbablePrime(50))
                .count();
        long end = System.currentTimeMillis();
        System.out.println(count);
        System.out.println(end - start);

        start = System.currentTimeMillis();
        count = LongStream.rangeClosed(2, 10000000L)
                .parallel()
                .mapToObj(BigInteger::valueOf)
                .filter(i -> i.isProbablePrime(50))
                .count();
        end = System.currentTimeMillis();
        System.out.println(count);
        System.out.println(end - start);
    }
}
```

프로그래밍 원칙

- 지역 변수의 범위를 최소화
  - ✓ 지역 변수의 유효 범위를 최소화하면 가독성과 유지보수성이 좋아지고 오류 가능성이 낮아짐
  - ✓ 지역 변수의 범위를 줄이는 가장 강력한 기법은 처음 쓰일 때 선언하는 것
  - ✓ 실제 사용하는 블록 바깥에 선언한 변수는 그 블록이 끝난 뒤에도 살아있음
  - ✓ 거의 모든 지역 변수는 선언과 동시에 초기화해야 하는데 try-catch 문은 여기서 예외 - 변수를 초기화할 때 예외를 던질 가능성이 있으면 try 블록 안에서 초기화해야 하므로 try 블록을 넘어서도 사용할 것 같으면 try 블록 앞에서 선언해야 함
  - ✓ 의도하지 않은 동작을 수행하는 코드

```
Iterator<Element> i = c.iterator();
while (i.hasNext()) {
    doSomething(i.next());
}

...

Iterator<Element> i2 = c2.iterator();
while (i.hasNext()) {
    doSomethingElse(is.next());
}
```

프로그래밍 원칙

- 전통적인 for 문보다는 for-each 문을 사용
  - ✓ 컬렉션이나 배열 순회

```
for (Iterator<Element> i = c.iterator(); i.hasNext(); ) {
    Element e = i.next();
    …
}


for (int i = 0; i < a.length; i++) {
    …
}
```

  - 반복자와 인덱스 변수는 코드를 지저분하게 하고 오류가 생길 가능성이 높아짐
  - 1회 반복에서 반복자는 세 번 등장하고 인덱스는 네 번 등장하고 있는데 혹시라도 잘못된 변수를 사용했을 때 컴파일러가 잡아주지 못할 수도 있고 컬렉션이나 배열이냐에 따라 코드 형태도 달라지고 있음

프로그래밍 원칙

- 전통적인 for 문보다는 for-each 문을 사용
  - ✓ 향상된 for-each

```
for (Element e:  elements) {
    ...
}
```

  - ✓ for-each 문 사용 불가 상황
    - 파괴적인 필터링: 컬렉션을 순회하면서 선택된 원소를 제거해야 한다면 remove 메서드를 호출해야 하는데 자바 8부터는 Collection의 removeIf 메서드를 사용해 컬렉션을 명시적으로 순회하는 일을 피할 수 있음
    - 변형: 리스트나 배열을 순회하면서 그 원소의 값 일부 혹은 전체를 교체해야 한다면 리스트의 반복자나 배열의 인덱스를 사용해야 함
    - 병렬 반복: 여러 컬렉션을 병렬로 순회해야 한다면 각각의 반복자와 인덱스 변수를 사용해 명시적으로 제어해야 함

프로그래밍 원칙

- 라이브러리를 익히고 사용
  - ✓ 장점
    - 표준 라이브러리는 검증된 코드
    - 표준 라이브러리르 쓰면 핵심적인 일에 집중할 수 있음
    - 따로 노력하지 않아도 성능이 지속해서 개선되는데 표준 라이브러리는 꾸준히 개선됨
    - 기능이 점점 추가되는데 부족한 부분은 다음 릴리스에 추가됨
  - ✓ 자바 프로그래머라면 java.lang, java.util, java.io와 하위 패키지들은 익숙해져야 하며 컬렉션 프레임워크와 스트림 라이브러리 및 java.util.concurrent도 중요
  - ✓ 필요한 라이브러리가 표준에 없으면 고품질의 서드파티 라이브러리를 사용 – 바퀴를 다시 구현할 필요 없음

프로그래밍 원칙

- 정확한 답이 필요하다면 float과 double은 회피
  - ✓ float과 double 타입은 과학과 공학 계산용으로 설계되었기 때문에 정확한 결과가 필요한 금융 관련 계산과는 맞지 않음
  - ✓ 금융 계산에는 BigDecimal, int, long 을 사용
  - ✓ BigDecimal은 기본타입보다 훨씬 느리고 불편한데 BigDecimal 대신 int 혹은 long을 쓸 경우 값의 크기가 제한되고 소수점을 직접 관리해야 함
  - ✓ BigDecimal에는 문자열을 넘겨야하며 double을 넘기면 안되는데 double형을 넘기면서 그 순간 정확도를 보장하지 못함

프로그래밍 원칙

- ❖ 박싱된 기본 타입보다는 기본 타입을 사용
  - ✓ 기본 타입 과 박싱된 기본 타입의 차이
    - 박싱된 기본 타입은 값에 더해 식별성을 가지기 때문에 그 박싱된 기본 타입의 두 인스턴스는 값이 같아도 서로 다르다고 식별될 수 있음 - -128 ~ 128 까지는 캐싱되어 있음
    - 기본 타입의 값은 언제나 유효하나 박싱된 기본 타입은 유효하지 않은 값인 null을 가질 수 있음
    - 기본 타입이 시간과 메모리면에서 더 효율적
  - ✓ 사용해야 하는 경우
    - 컬렉션의 원소, 키 값. 매개변수화 타입 이나 매개변수화 메서드의 타입 매개변수를 사용하는 경우
    - 리플렉션를 통해 메서드를 호출할 때
    - null 값 처리가 용이하기 때문에 SQL과 연동할 경우에 처리를 원활하게 할 수 있는데 DB에서 자료형이 정수형이지만 null 값이 필요한 경우 VO에서 Integer를 사용할 수 있음

프로그래밍 원칙

- 다른 타입이 적절하다면 문자열 사용을 회피
  - ✓ 문자열은 다른 값 타입을 대신하기에 적합하지 않음
  - ✓ 받은 데이터가 수치형이라면 int, float, BigInteger등 적당한 수치 타입으로 변환해야 하며 예/아니오 질문의 답이라면 적절한 열거 타입이나 boolean으로 변환
  - ✓ 문자열은 열거 타입을 대신하기에 적합하지 않은데 상수를 열거할 때는 문자열보다 열거 타입이 월등히 나음
  - ✓ 문자열은 혼합 타입을 대신하기에 적합하지 않은데 여러 요소가 혼합된 데이터를 하나의 문자열로 표현하는 것은 대체로 좋지 않은 생각

```
String compoundKey = className + "#" + i.next();
```

  - ✓ 개별 요소에 접근하려면 파싱해야해서 느리고 오류 가능성도 커지는데 equals, toString, compareTo 메서드도 제공할 수 없기 때문에 전용 클래스를 새로 만드는 편이 나음
  - ✓ 문자열은 권한을 표현하기에 적합하지 않음

프로그래밍 원칙

- 문자열 연결은 느리니 주의
  - ✓ 문자열 연결 연산자로 문자열 n개를 잇는 시간은 n^2에 비례
  - ✓ 문자열은 불변이라서 두 문자열을 연결할 경우 양쪽의 내용을 모두 복사해야 함
  - ✓ 성능을 포기하고 싶지 않다면 String 대신 StringBuilder를 사용
  - ✓ JDK 5이상부터는 String으로 더하기 연산을 할 경우 컴파일할 때 자동으로 StringBuilder로 변환을 해주지만 for 루프와 같이 반복 연산을 할 때에는 자동으로 변환을 해주지 않으므로 꼭 필요

프로그래밍 원칙

- 객체는 인터페이스를 사용해 참조
  - ✓ 적합한 인터페이스만 있다면 매개변수뿐 아니라 반환값, 변수, 필드를 전부 인터페이스 타입으로 선언

```
// 좋은 예
Set<Son> sonSet = new LinkedHashSet<>();

// 나쁜 예
LinkedHashSet<Son> sonSet = new LinkedHashSet<>();
```

  - ✓ 인터페이스를 타입으로 사용하면 프로그램이 훨씬 유연해지는데 나중에 구현 클래스를 교체할 때 새 클래스의 생성자를 호출해주기만 하면 됨

```
Set<Son> sonSet = new HashSet<>();
```

  - ✓ 적합한 인터페이스가 없다면 당연히 클래스로 참조해야 하는데 인터페이스에는 없는 특별한 메서드를 제공하는 클래스들이다.

```
List<Integer> list = new ArrayList<>();
ArrayList<Integer> list2 = new ArrayList<>();

// list.ensureCapacity
list2.ensureCapacity(3);
```

프로그래밍 원칙

- 일반적으로 통용되는 명명 규칙을 사용
  - ✓ 패키지
    - 패키지의 각 요소는 일반적으로 8자 이하 짧은 단어로 설정
    - utilities 보다는 util 처럼 약어를 추천
    - 여러 단어로 구성된 이름이라면 awt 처럼 첫 글자만 따기도 함
  - ✓ 클래스
    - 객체를 생성할 수 있는 클래스는 보통 단순 명사나 명사구를 사용 (Thread, PriorityQueue, ChessPiece...)
    - 객체를 생성할 수 없는 클래스는 보통 복수형 명사로(Collectors, Collections...)
    - 인터페이스 이름은 클래스(Collection, Comparator)와 똑같이 짓거나 able 또는 ible로 끝나게(Runnable, Iterable, Accessible)
  - ✓ 메서드
    - boolean 값을 반환하면 보통 is로 시작 드물게 has로 시작
    - 해당 인스턴스 속성 반환하는 메서드는 보통 명사, 명사구 또는 get으로 시작하게 (size, hashCode, getTime 등)
    - 객체의 타입을 바꿔서 반환하는 메서드는 보통 toType으로

예외

- 예외는 진짜 예외 상황에만 사용하라
  - ✓ 예외는 예외 상황에 쓸 용도로 설계되었으므로 JVM 구현자가 최적화를 하지 않았을 수 있음
  - ✓ try-catch 블록 안에 넣으면 JVM이 적용할 수 있는 최적화가 제한됨

예외

- 복구할 수 있는 상황에는 검사 예외를, 프로그래밍 오류에는 런타임 예외를 사용
  - ✓ 검사 예외 (Checked Exception)
    - 호출하는 쪽에서 복구하리라 여겨지는 상황이라면 검사 예외(checked exception)를 사용해야하는데 checked exception은 호출자가 그 예외를 catch로 잡아 처리하거나, 위로 전파하도록 강제하므로 메서드 선언에 포함된 checked exception 각각은 그 메서드를 호출했을 때 발생할 수 있는 유력한 결과임을 클라이언트에게 알려주는 것
    - unchecked throwable은 런타임 예외이거나 에러인데 이 둘은 프로그램에서 잡을 필요 없거나 혹은 통상적으로는 잡지 말아야 하는데 프로그램에서 비검사 예외나 에러를 던졌다는 것은 복구가 불가능하거나 더 실행해봐야 득보다는 실이 많다는 의미
    - 검사 예외는 일반적으로 복구할 수 있는 조건일 때 발생하므로 호출자가 예외 상황에서 벗어나는데 필요한 정보를 알려주는 메서드를 함께 제공하는 것이 중요
  - ✓ 런타임 예외 (Unchecked Exception)
    - 프로그래밍 오류를 나타낼 때는 런타임 예외를 사용하는데 런타임 예외의 대부분은 전제조건을 만족하지 못했을 때 발생 (ex. 배열 인덱스 크기. API 명세에 기록된 제약을 지키지 못했다는 것)
    - 물론 복구할 수 있는 상황인지 프로그래밍 오류인지 항상 명확히 구분되는 것은 아닌데 말도 안되는 크기의 배열을 할당해 생긴 프로그래밍 오류일 수도 있고 진짜로 자원이 부족해서 발생한 문제일 수도 있음
  - ✓ 에러
    - 에러는 보통 더 이상 수행할 수 없는 상황을 나타내기 때문에 구현하는 비검사 throwable은 모두 RuntimeException의 하위 클래스여야 하는데 Error는 상속하지도 말고 throw문도 던지지 않아야 함(AssertionError는 예외)

예외

- 필요 없는 검사 예외 사용은 피하라
  - ✓ 검사 예외를 잘 사용하면 발생한 문제를 프로그래머가 처리하여 안전성을 높여주는데 하지만 과하게 쓰면 오히려 불편하게 됨
  - ✓ 어떤 메서드가 검사 예외를 던질 수 있게 선언됐다면 이를 호출하는 쪽에서 catch 블록을 두거나 바깥으로 던져서 전파해야 하는데 이는 클라이언트에게 부담을 주고 예외를 던지는 메서드는 스트림 안에서 직접 사용할 수 없기 때문에 자바 8부터 부담이 더 커짐
  - ✓ 검사 예외를 회피하는 가장 쉬운 방법은 적절한 결과 타입을 담은 옵셔널을 반환하는 것이고 다른 방법은 검사 예외를 던지는 메서드를 2개로 쪼개 비검사 예외로 바꿀 수 있음

예외

- 표준 예외를 사용
  - ✓ 표준 예외를 사용해야 하는 이유
    - 많은 프로그래머가 익숙하기 때문에 가독성이 좋음
    - 클래스 수가 적으면 메모리 사용량도 줄고 클래스 적재 시간도 적게 걸림
  - ✓ 많이 쓰이는 예외
    - IllegalArgumentException: 호출자가 인수로 부적절한 값을 넘길 때 던지는 예외.
    - IllegalStateException: 대상 객체의 상태가 호출된 메서드를 수행하기에 적절하지 않을 때
    - NullPointerException: null 값을 허용하지 않는 메서드에 null을 건낼 때
    - IndexOutOfBoundsException: 시퀀스의 허용 범위 넘을 때
    - ConcurrentModificationException: 단일 스레드에서 사용하려고 설계한 객체를 여러 스레드가 동시에 수정하려고 할 때
    - UnsupportedOperationException: 클라이언트가 요청한 동작을 대상 객체가 지원하지 않을 때

예외

- 표준 예외를 사용
  - ✓ 커스텀 예외 사용
    - 예외 이름 자체가 정보를 전달할 수 있는데 NoSuchElementException 보다는 PostNotFoundException이 더 정확
    - 더 상세하게 예외 정보를 제공할 수 있음
    - 예외의 응집도 향상 - 예외에 필요한 메시지, 전달할 정보의 데이터 등등을 한 곳에서 관리가 가능
    - 정확하게 위치 파악이 가능 - 커스텀화를 시켰기 때문에 어떤 상황에서 발생했는지 특정할 수 있음
    - 예외 생성 비용을 절감 - 예외를 생성할 때 stackTrace 때문에 비용이 많이 발생하는데 이 stackTrace는 상위 클래스 중 Throwable.fillInStackTrace()에서 발생하는데 이걸 Override해서 간략하게 하거나 아예 생성하지 않을 수 있음

예외

- 메서드가 던지는 모든 예외를 문서화하라
  - ✓ 예외는 아주 중요한 정보
  - ✓ 검사 예외는 항상 따로 따로 선언하고 각 예외 발생 상황을 자바독의 @throws 태그로 정확히 문서화해야 하는데 유일한 예외는 main메서드로 main은 오직 JVM만이 호출하므로 Exception을 던져도 됨
  - ✓ 비검사 예외도 문서화해두면 좋은데 public 메서드라면 필요한 조건을 문서화 해야하고 그 수단으로 좋은게 비검사 예외 문서화인데 비검사 예외는 메서드 선언의 throws 목록에 넣지 않아야 함
  - ✓ 자바독 유틸리티가 @throws 태그에도 명시한 예외와 @throws 태그에만 명시한 예외를 구분해주기 때문에 프로그래머가 어느 것이 비검사 예외인지 바로 알 수 있음
  - ✓ 한 클래스에 정의된 많은 메서드가 같은 이유로 같은 예외를 던지면 클래스 차원에 설명을 붙일 수도 있는데 NPE가 대표적인 예로 클래스의 문서화 주석에 인수로 null이 오면 NPE 반환 이런식으로 적어도 됨

예외

- 예외의 상세 메시지에 실패 관련 정보를 담으라
  - ✓ 예외를 잡지 못해 프로그램이 실패하면 스택 추적(stack trace)가 출력되는데 스택 추적은 예외 객체의 toString 메서드를 호출해 얻는 문자열
  - ✓ 만약 실패를 재현하기 어려우면 자세한 정보를 얻기가 어렵기 때문에 실패 원인에 관한 정보를 가능한 한 많이 담아야 함
  - ✓ IndexOutOfBoundsException의 상세 메시지에는 범위의 최소값과 최댓값, 그리고 인덱스 값을 담아야 하는데 인덱스가 잘못 됐을 수도 있고, 최소 값이 최대값보다 클 수도 있음
  - ✓ 실패 순간을 포착하려면 발생한 예외에 관여된 모든 매개변수와 필드의 값을 실패 메시지에 담아야 함
  - ✓ 예외의 상세 메시지와 최종 사용자에게 보여줄 오류 메시지를 혼동해서는 안되는데 최종 사용자에게는 친절한 메시지를 보여주고 예외 메시지는 가독성보다 담긴 내용이 중요
  - ✓ 예외는 실패와 관련한 정보를 얻을 수 있는 접근자 메서드를 적절히 제공하는 것이 좋은데 포착한 실패 정보로 예외 상황을 복구할 수 있으므로 접근자 메서드는 unchecked exception보다는 checked exception에서 더 유용할 것이지만 toString이 반환한 값에 포함된 정보를 얻어올 수 있는 API를 제공하는 원칙에 따라 unchecked exception도 접근자 메서드를 제공하면 좋을 것

예외

- 가능한 한 실패 원자적으로 만들라
  - ✓ 실패 원자적(failure-atomic): 호출된 메서드가 실패하더라도 해당 객체는 메서드 호출 전 상태를 유지해야 한다는 것인데 예를 들어 작업 도중 예외가 발생하도 그 객체는 여전히 정상적으로 사용할 수 있는 상태
  - ✓ 메서드를 실패 원자적으로 만드는 방법은 다양
    - 불변 객체화: 불변 객체로 설계하는 것으로 메서드가 실패하면 새로운 객체가 만들어지지 않을 수는 있으나 기존 객체가 불안정한 상태에 빠지는 일은 없는데 불변 객체의 상태는 생성 시점에 고정되어 절대 변하지 않기 때문
    - 매개변수 유효성 검사: 작업 수행에 앞서 매개변수의 유효성 검사를 해서 객체의 내부 상태를 변경하기 전에 잠재적 예외의 가능성 대부분을 걸러낼 수 있음
    - 임시 복사본에서 작업 후 교체
    - 복구 코드 작성: 작업 도중 발생하는 실패를 가로채는 복구 코드를 작성하여 작업 전 상태로 되돌리는 방법으로 자주 쓰이는 것은 아님
  - ✓ 예외 사항: 실패 원자성은 항상 달성할 수 있는 건 아닌데 두 스레드가 동기화 없이 같은 객체를 수정하면 그 객체의 일관성이 깨질 수 있기 때문에 ConcurrentModificationException은 잡아냈다고 해도 그 객체가 쓸 수 있는 상황이라고 생각해서는 안됨
  - ✓ 메서드 명세에 기술한 예외라면 예외가 발생하더라도 객체의 상태는 메서드 호출 전과 똑같이 유지돼야 하는 것이 기본 규칙인데 지키지 못하면 실패 시의 객체 상태를 API 설명에 명시해야 함

예외

❖ 예외를 무시하지 말라

✓ 예외 무시

```
try {
    …
} catch (SomeException e) {
}
```

- 위에 처럼 catch 블록을 비워두면 예외가 존재할 이유가 없어짐
- 예외를 무시하기로 했다면 catch 블록안에 그렇게 결정한 이유를 주석으로 남기고 예외 변수의 이름도 ignored로 변경

✓ 예외 전달

```
try {
    numColors = f.get(1L, TimeUnit.SECONDS);
} catch (TimeoutException | ExecutionException ignored) {
    …
}
```

- 예외를 무시하지 않고 바깥으로만 전파하게 놔둬도 최소한 디버깅 정보를 남긴채 프로그램이 신속히 중단되게는 할 수 있음