[아이템 19] 상속을 고려해 설계하고 문서화하라. 그러지 않았다면 상속을 금지하라.

상속이 가능한 클래스를 설계 시 고려해야 할 점

- 제대로 된 문서화를 할 것

  (부모 클래스에서) 오버라이딩 가능한 메소드들의 자체 사용(self-use) 즉, 그 메소드들이 같은 클래스의 다른 메소드를 호출하는지에 대해 문서화 할 것

  반대로, public이나 protected 메소드 및 생성자에서 오버라이드 가능한 메소드를 호출하는건 아닌지, 호출한다면 어떤 순서로 하는지, 호출한 결과가 다음 처리에 어떤 영향을 주는지에 대해 문서화 할 것
- 잘 선정된 protected 메소드를 제공하여 클래스 내부의 다른 메소드와 연결되도록 할 것

1. [AbstractCollection] public boolean remove(Object o)
2. [AbstractList] protected void removeRange(int fromIndex, int toIndex)

오버라이드 가능하다? final 이 아니면서 public 이나 protected 인 경우 오버라이딩 가능하다

상속을 위한 클래스를 제대로 만들었는지에 대한 테스트 - 서브 클래스를 만들어 보기

상속을 위한 클래스를 제대로 만들었는지 테스트 하는 유일한 방법은 서브 클래스를 직접 만들어 보는 것인데

1. 중요한 protected 멤버를 빼먹어서 서브 클래스 작성이 어려운 것은 아닌지 느낄 수 있고

2. 여러 서브 클래스를 만들때 사용하지 않았던 protected 메소드가 있다면 그 멤버를 private 으로 선언할지를 고려할 수 있다.

생성자에서는 오버라이드 가능한 메소드를 호출해서는 안된다.

상속을 허용하기 위해서는 클래스의 생성자에서 직접 또는 간접의 어떤 형태로든 오버라이드 가능한 메소드를 호출하지 않도록 하자.

1. Super.overrideMe(), Sub.overrideMe()
2. Cloneable, Serializable 인터페이스를 구현한 클래스를 상속하는 경우 비슷한 일이 생길 수 있다.

제대로 문서화하지 않을 생각이라면 클래스의 상속을 금지해라

클래스의 상속을 금지시키는 방법?

1. 클래스를 final 로 선언하기
2. 클래스의 모든 생성자를 private 이나 패키지 전용(그 패키지 안에서만 사용 가능한 클래스)으로 하고 생성자 대신 public static 팩토리 메소드를 추가하는 것

클래스를 상속 가능하게 할 때, 오버라이딩 가능한 메소드들의 경우 정확한 문서화가 필요하다. 관례적으로 오버라이드 가능한 메소드를 호출하는 메소드들의 주석은 이번 구현 버전에서는 ... 하다 라는 구문으로 시작한다.

AbstractCollection 클래스의 remove() 메소드를 보면 파라매터로 받은 요소가 컬렉션에 있는지 찾고 찾으면 이를 Iterator.remove() 를 이용해서 삭제한다.만일 이 컬렉션의 iterator() 로 반환한 순환자 객체가 remove() 를 구현하고 있지 않으면 현재 버전에서는 UnsuppertedOperationException 예외가 발생한다.

```
public abstract class AbstractCollection<E> implements Collection<E> {

  /**
   * {@inheritDoc}
   *
   * @implSpec
```

```
    * This implementation iterates over the collection looking for the
    * specified element.  If it finds the element, it removes the element
    * from the collection using the iterator's remove method.
    *
    * <p>Note that this implementation throws an
    * {@code UnsupportedOperationException} if the iterator returned by this
    * collection's iterator method does not implement the {@code remove}
    * method and this collection contains the specified object.
    *
    * @throws UnsupportedOperationException {@inheritDoc}
    * @throws ClassCastException            {@inheritDoc}
    * @throws NullPointerException          {@inheritDoc}
    */
   public boolean remove(Object o) {
       Iterator<E> it = iterator();
       if (o==null) {
           while (it.hasNext()) {
               if (it.next()==null) {
                   it.remove();
                   return true;
               }
           }
       } else {
           while (it.hasNext()) {
               if (o.equals(it.next())) {
                   it.remove();
                   return true;
               }
           }
       }
       return false;
   }

   // ...

}
```

위는 좋은 문서화의 예이다.

오버라이딩 할 수 있는 메소드들이 같은 클래스의 다른 메소드들을 내부적으로 어떻게 이용하고 있는지 (self-use) 서술하고 있기 때문이다.

- iterator() 를 오버라이딩 하면 해당 메소드-remove()-에 영향을 준다
- iterator() 에서 반환된 Iterator 객체가 해당 메소드에 영향을 준다

💡 잘된 API 문서는 what 을 기술 하고 how 를 설명해서는 안된다는 통념이 있다. 무슨 일을 하는지 기술해야 하고 어떻게 하는지는 설명하지 말라는 건데, 상속과 관련해 이러한 케이스들 같은 경우는 어쩔 수 없이 이 통념을 어기더라도 서브 클래스가 안전하게끔 클래스의 상세 구현 내역을 기술해야 한다.

이외에도 클래스를 상속 가능하게 할 때 잘 선정된 protected 메소드를 제공해야 한다. 이런 protected 메소드들에서 클래스 내부의 다른 메소드를 잘 호출할 수 있도록 해야 한다는 것

```
public abstract class AbstractList<E> extends AbstractCollection<E> implements Li
st<E> {
  // ...

  /**
   * Removes from this list all of the elements whose index is between
   * {@code fromIndex}, inclusive, and {@code toIndex}, exclusive.
   * Shifts any succeeding elements to the left (reduces their index).
   * This call shortens the list by {@code (toIndex - fromIndex)} elements.
   * (If {@code toIndex==fromIndex}, this operation has no effect.)
   *
   * <p>This method is called by the {@code clear} operation on this list
   * and its subLists.  Overriding this method to take advantage of
   * the internals of the list implementation can <i>substantially</i>
   * improve the performance of the {@code clear} operation on this list
   * and its subLists.
   *
   * @implSpec
   * This implementation gets a list iterator positioned before
   * {@code fromIndex}, and repeatedly calls {@code ListIterator.next}
   * followed by {@code ListIterator.remove} until the entire range has
   * been removed.  <b>Note: if {@code ListIterator.remove} requires linear
   * time, this implementation requires quadratic time.</b>
   *
   * @param fromIndex index of first element to be removed
   * @param toIndex index after last element to be removed
   */
  protected void removeRange(int fromIndex, int toIndex) {
      ListIterator<E> it = listIterator(fromIndex);
      for (int i=0, n=toIndex-fromIndex; i<n; i++) {
          it.next();
          it.remove();
      }

   /**
   * Removes all of the elements from this list (optional operation).
   * The list will be empty after this call returns.
   *
```

```
     * @implSpec
     * This implementation calls {@code removeRange(0, size())}.
     *
     * <p>Note that this implementation throws an
     * {@code UnsupportedOperationException} unless {@code remove(int
     * index)} or {@code removeRange(int fromIndex, int toIndex)} is
     * overridden.
     *
     * @throws UnsupportedOperationException if the {@code clear} operation
     *         is not supported by this list
     */
    public void clear() {
        removeRange(0, size());
    }
}
```

List 구현 클래스(ArrayList, LinkedList, ...) 를 사용할 때 우리는 removeRange() 를 고려하지 않고 그저 clear() 와 같은 public 한 메소드들의 사용만 고려하면 된다. 내부 구현은 사용자 입장에서 고려할 것이 아니다.

만약 우리가 ArrayList 나 LinkedList 같이 List 를 구현하는 클래스를 새로 작성하는 입장이라고 생각해보자. protected 로 정의된 removeRange 메소드를 재정의 할 수 있고 이로써 clear() 메소드의 연산 성능을 크게 향상 시킬 수 있다.

상속을 위한 클래스를 작성할 때 적절한 메소드를 protected 로 선언하여 밖에서는 숨기고 안에서는 적당한 메소드를 호출해야 잘 한것이다

이렇게 상속을 위한 클래스를 만들었는데 과연 얘네를 잘 만든게 맞는지 테스트해 보는 방법은 서브 클래스를 직접 만들어 보는 것이다. (1) 만약 서브 클래스의 구현이 어렵다면 너무 멤버나 메소드를 private 으로 숨겨서 작성히 힘든지를 생각해보고 이럴 때 적당히 protected 로 돌릴 수 있고, (2) 처음에 부모 클래스를 설계했을 때는 서브 클래스에서 사용할 것을 고려해서 protected 로 선언 했지만 사용되지 않는 메소드가 있다면 이를 private 으로 돌릴 수 있다.

또한 상속을 허용하기 위해서는 클래스의 생성자에서 직접 또는 간접의 어떤 형태로든 오버라이드 가능한 메소드를 호출하지 않아야 한다.

다음 예시를 보자. 부모 클래스 Super 의 생성자에서 overrideMe() 메소드를 호출하는데 이는 public 으로 자식 클래스에서 오버라이딩 가능하게 설계되었다. 만약 Sub 클래스에서 overrideMe() 를 재정의 하는 경우 어떤 일이 생기는지 보자

```java
// Class whose constructor invokes an overridable method.
// NEVER DO THIS! (Page 95)
public class Super {
    // Broken - constructor invokes an overridable method
    public Super() {
        overrideMe();
    }

    public void overrideMe() {
    }
}
```

```java
// Demonstration of what can go wrong when you override a method
// called from constructor (Page 96)
public final class Sub extends Super {
    // Blank final, set by constructor
    private final Instant instant;

    Sub() {
        instant = Instant.now();
    }

    // Overriding method invoked by superclass constructor
    @Override public void overrideMe() {
        System.out.println(instant);
    }
}
```

```java
public static void main(String[] args) {
    Sub sub = new Sub();
    sub.overrideMe();
}
```

▼ 위의 실행결과는 어떻게 될까?

```
null
2021-01-24T12:27:48.596171Z
```

자식 타입의 객체를 생성할 때 부모 클래스의 생성자가 먼저 불리게 되고 자식 클래스의 생성자가 차례로 호출된다. 이처럼 null 이 찍히면서 예상대로 동작하지 않을 수 있다는 것이다. 잘못하면 NullPointerException 이 날 수도 있는 상황임.

Cloneable 이나 Serializable 인터페이스를 구현하는 클래스를 상속하게 설계하고 싶은 경우, 생성자에서 오버라이딩 가능한 메소드를 호출하면 안 되듯, clone() 이나 readObject() 메소드에서 오버라이딩 가능한 메소드를 호출하지 않게끔 하자.

이처럼 상속 가능한 클래스를 설계하려면 제대로 된 문서화, 적당히 잘 protected 로 빼놓기, 생성자에서 오버라이딩 가능한 메소드 호출 안하도록 해놓기 등 많은 귀찮음이 생긴다. 이런 처리를 잘 해놓지 못할 바에는 클래스의 상속을 금지시키는 게 좋다.

문서화의 잘 된 예

```
public abstract class AbstractQueue<E>
    extends AbstractCollection<E>
    implements Queue<E> {

  /**
   * Inserts the specified element into this queue if it is possible to do so
   * immediately without violating capacity restrictions, returning
   * {@code true} upon success and throwing an {@code IllegalStateException}
   * if no space is currently available.
   *
   * <p>This implementation returns {@code true} if {@code offer} succeeds,
   * else throws an {@code IllegalStateException}.
   *
   * @param e the element to add
   * @return {@code true} (as specified by {@link Collection#add})
   * @throws IllegalStateException if the element cannot be added at this
   *         time due to capacity restrictions
   * @throws ClassCastException if the class of the specified element
   *         prevents it from being added to this queue
   * @throws NullPointerException if the specified element is null and
   *         this queue does not permit null elements
   * @throws IllegalArgumentException if some property of this element
   *         prevents it from being added to this queue
   */
  public boolean add(E e) {
      if (offer(e))
          return true;
      else
          throw new IllegalStateException("Queue full");
  }
}
```