이팩티브 자바 완벽 공략 2부

“클래스와 인터페이스” 그리고 “제네릭”

인프런 / 백기선 (AKA, WHITESHIP)

이팩티브 자바 (Effective Java)

조류와 블로크 지음 / 개앞맵시(이복연) 번역

- 모든 자바 개발자의 필독서
- 하지만, 대부분의 자바 개발자가 소화하기 어려운 내용이 많다.
- 그래서 이 강의를 준비했다.

클래스와 인터페이스

클래스와 인터페이스를 쓰기 편하고, 견고하며, 유연하게 만드는 방법

아이템 15. 클래스와 멤버의 접근 권한을 최소화하라.

핵심 정리 1: 구현과 API를 분리하는 “정보 은닉”의 장점

- 시스템 개발 속도를 높인다. (여러 컴포넌트를 병렬로 개발할 수 있기 때문에)
- 시스템 관리 비용을 낮춘다. (컴퍼넌트를 더 빨리 파악할 수 있기 때문에)
- 성능 최적화에 도움을 준다. (프로파일링을 통해 최적화할 컴포넌트를 찾고 다른 컴포넌트에 영향을 주지 않고 해당 컴포넌트만 개선할 수 있기 때문에)
- 소프트웨어 재사용성을 높인다. (독자적인 컴포넌트라면)
- 시스템 개발 난이도를 낮춘다. (전체를 만들기 전에 개별 컴포넌트를 검증할 수 있기 때문에)

아이템 15. 클래스와 멤버의 접근 권한을 최소화하라.

핵심 정리 2: 클래스와 인터페이스의 접근 제한자 사용 원칙

- 모든 클래스와 멤버의 접근성을 가능한 한 좁혀야 한다.
- 톱레벨 클래스와 인터페이스에 package-private 또는 public을 쓸 수 있다.
  - public으로 선언하면 API가 되므로 하위 호환성을 유지하려면 영원히 관리해야 한다.
  - 패키지 외부에서 쓰지 않을 클래스나 인터페이스라면 package-private으로 선언한다.
- 한 클래스에서만 사용하는 package-private 클래스나 인터페이스는 해당 클래스에 private static으로 중첩 시키자. (아이템 24)

아이템 15. 클래스와 멤버의 접근 권한을 최소화하라.

핵심 정리 3: 멤버(필드, 메서드, 중첩 클래스/인터페이스)의 접근 제한자 원칙

- private과 package-private은 내부 구현.
- public 클래스의 protected와 public은 공개 API.
- 코드를 테스트 하는 목적으로 private을 package-private으로 풀어주는 것은 허용할 수 있다. 하지만 테스트만을 위해서 멤버를 공개 API로 만들어서는 안 된다. (테스트를 같은 패키지에 만든다면 그럴 필요도 없다.)
- public 클래스의 인스턴스 필드는 되도록 public이 아니어야 한다. (아이템16)
- 클래스에서 public static final 배열 필드를 두거나 이 필드를 반환하는 접근자 메서드를 제공해서는 안 된다.

아이템 15. 클래스와 멤버의 접근 권한을 최소화하라.

완벽 공략

- p98, Serializable, “완벽 공략 13, 객체 직렬화” 참고
- p98, 리스코프 치환 원칙, “완벽 공략 26” 참고
- p99, 스레드 안전 (Thread Safe), “완벽 공략 28” 참고
- p99, 불변 객체, “완벽 공략 24, Value 기반의 클래스” 참고
- p100, 자바 9 모듈 시스템

완벽 공략 31. 자바 9 모듈

Java Platform Module System

- JSR-376 스팩으로 정의한 자바의 모듈 시스템
- 안정성 - 순환 참조 허용하지 않음, 실행시 필요한 모듈 확인, 한 패키지는 한 모듈에서만 공개할 수 있음.
- 캡슐화 - public 인터페이스나 클래스라 하더라도, 공개된 패키지만 사용할 수 있다. 내부 구현을 보호하는 수단으로 사용할 수 있다. (하지만 모듈이 아닌 곳에서 참조한다면…)
- 확장성 - 필요한 자바 플랫폼 모듈만 모아서 최적의 JRE를 구성할 수 있다. 작은 기기에서 구동할 애플리케이션을 개발할 때 유용하다.

완벽 공략 31. 자바 9 모듈

Java Platform Module System

아이템 16. public 클래스에서는 public 필드가 아닌 접근자 메서드를 사용하라.

핵심 정리

```
public class Point {
    public double x;
    public double y;
}
```

- 클라이언트 코드가 필드를 직접 사용하면 캡슐화의 장점을 제공하지 못한다.
- 필드를 변경하려면 API를 변경해야 한다.
- 필드에 접근할 때 부수 작업을 할 수 없다.
- package-private 클래스 또는 private 중첩 클래스라면 데이터 필드를 노출해도 문제가 없다.

아이템 16. public 클래스에서는 public 필드가 아닌 접근자 메서드를 사용하라.

완벽 공략

- p103, 아이템 67에서 설명하듯, 내부를 노출한 Dimesion 클래스의 심각한 성능 문제는 오늘날까지도 해결되지 못했다.

아이템 17. 변경 가능성을 최소화 하라.

핵심 정리 1: 불변 클래스

- 불변 클래스는 가변 클래스보다 설계하고 구현하고 사용하기 쉬우며, 오류가 생길 여지도 적고 훨씬 안전하다.
- 불변 클래스를 만드는 다섯 가지 규칙
  - 객체의 상태를 변경하는 메서드를 제공하지 않는다.
  - 클래스를 확장할 수 없도록 한다.
  - 모든 필드를 final로 선언한다. (JLS 17.5)
  - 모든 필드를 private으로 선언한다. (아이템 15, 16)
  - 자신 외에는 내부의 가변 컴포넌트에 접근할 수 없도록 한다.

아이템 17. 변경 가능성을 최소화 하라.

핵심 정리 2: 불변 클래스의 장점과 단점

- 함수형 프로그래밍에 적합하다. (피연산자에 함수를 적용한 결과를 반환하지만 피연산자가 바뀌지는 않는다.)
- 불변 객체는 단순하다.
- 불변 객체는 근본적으로 스레드 안전하여 따로 동기화할 필요 없다.
- 불변 객체는 안심하고 공유할 수 있다. (상수, public static final)
- 불변 객체 끼리는 내부 데이터를 공유할 수 있다.
- 객체를 만들 때 불변 객체로 구성하면 이점이 많다.
- 실패 원자성을 제공한다. (아이템 76, p407)
- 단점) 값이 다르다면 반드시 별도의 객체로 만들어야 한다.
  - “다단계 연산”을 제공하거나, “가변 동반 클래스”를 제공하여 대처할 수 있다.

아이템 17. 변경 가능성을 최소화 하라.

핵심 정리 3: 불변 클래스 만들 때 고려할 것

- 상속을 막을 수 있는 또 다른 방법
  - private 또는 package-private 생성자 + 정적 팩터리
  - 확장이 가능하다. 다수의 package-private 구현 클래스를 만들 수 있다.
  - 정적 팩터리를 통해 여러 구현 클래스중 하나를 활용할 수 있는 유연성을 제공하고 객체 캐싱 기능으로 성능을 향상 시킬 수도 있다.
- 재정의가 가능한 클래스는 방어적인 복사를 사용해야 한다.
- 모든 “외부에 공개하는” 필드가 final이어야 한다.
  - 계산 비용이 큰 값은 해당 값이 필요로 할 때 (나중에) 계산하여 final이 아닌 필드에 캐시해서 쓸 수도 있다.

아이템 17. 변경 가능성을 최소화 하라.

완벽 공략

- p105, 새로 생성된 불변 인스턴스를 동기화 없이 다른 스레드로 건네도 문제없이 동작 (JLS 17.5)
- p106, readObject 메서드 (아이템 88)에서 방어적 복사를 수행하라.
- p112, 불변 클래스의 내부에 가변 객체를 참조하는 필드가 있다면… (아이템 88)
- p113, java.util.concurrnet 패키지의 CountDownLatch 클래스

완벽 공략 32. final과 자바 메모리 모델(JMM)

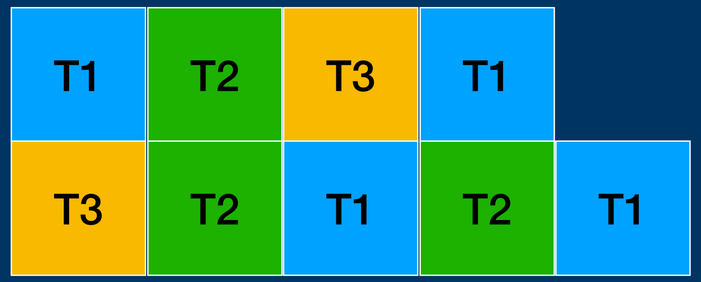
final을 사용하면 **안전하게** 초기화 할 수 있다.

- JMM과 final을 완벽히 이해하려면 JLS 17.4와 JLS 17.5를 참고하세요.
- JMM:
  - 자바 메모리 모델은 JVM의 메모리 구조가 아닙니다.
  - 적법한 (legal) 프로그램을 실행 규칙.
  - 메모리 모델이 허용하는 범위내에서 프로그램을 어떻게 실행하든 구현체(JVM)의 자유다. (이 과정에서 실행 순서가 바뀔 수도 있다.)
- 어떤 인스턴스의 final 변수를 초기화 하기 전까지 해당 인스턴스를 참조하는 모든 쓰레스는 기다려야 한다. (freeze)

완벽 공략 33. java.util.concurrnet 패키지

병행(concurrency) 프로그래밍에 유용하게 사용할 수 있는 유틸리티 묶음

- 병행(Concurrency)과 병렬(Parallelism)의 차이



- 병행은 여러 작업을 번갈아 가며 실행해 마치 동시에 여러 작업을 동시에 처리하듯 보이지만, 실제로는 한번에 오직 한 작업만 실행한다. CPU가 한개여도 가능하다.
- 병렬은 여러 작업을 동시에 처리한다. CPU가 여러개 있어야 가능하다.
- 자바의 concurrent 패키지는 병행 애플리케이션에 유용한 다양한 툴을 제공한다.
  - BlockingQueue, Callable, ConcurrrentMap, Executor, ExecutorService, Future, …

완벽 공략 33. CountDownLatch

다른 여러 스레드로 실행하는 여러 오퍼레이션이 마칠 때까지 기다릴 때 사용할 수 있는 유틸리티

- 초기화 할 때 숫자를 입력하고, await() 메서드를 사용해서 숫자가 0이 될때까지 기다린다.
- 숫자를 셀 때는 countDown() 메서드를 사용한다.
- 재사용할 수 있는 인스턴스가 아니다. 숫자를 리셋해서 재사용하려면 CyclicBarrier를 사용해야 한다.
- **시작** 또는 **종료** 신호로 사용할 수 있다.

아이템 18. 상속보다는 컴포지션을 사용하라
핵심 정리

- 패키지 경계를 넘어 다른 패키지의 구체 클래스를 상속하는 일은 위험하다.
  - 상위 클래스에서 제공하는 메서드 구현이 바뀐다면…
  - 상위 클래스에서 새로운 메서드가 생긴다면…
- 컴포지션 (Composition)
  - 새로운 클래스를 만들고 private 필드로 기존 클래스의 인스턴스를 참조.
  - 새 클래의 인스턴스 메서드들은 기존 클래스에 대응하는 메서드를 호출해 그 결과를 반환한다.
  - **기존 클래스의 구현이 바뀌거나, 새로운 메서드가 생기더라도 아무런 영향을 받지 않는다.**

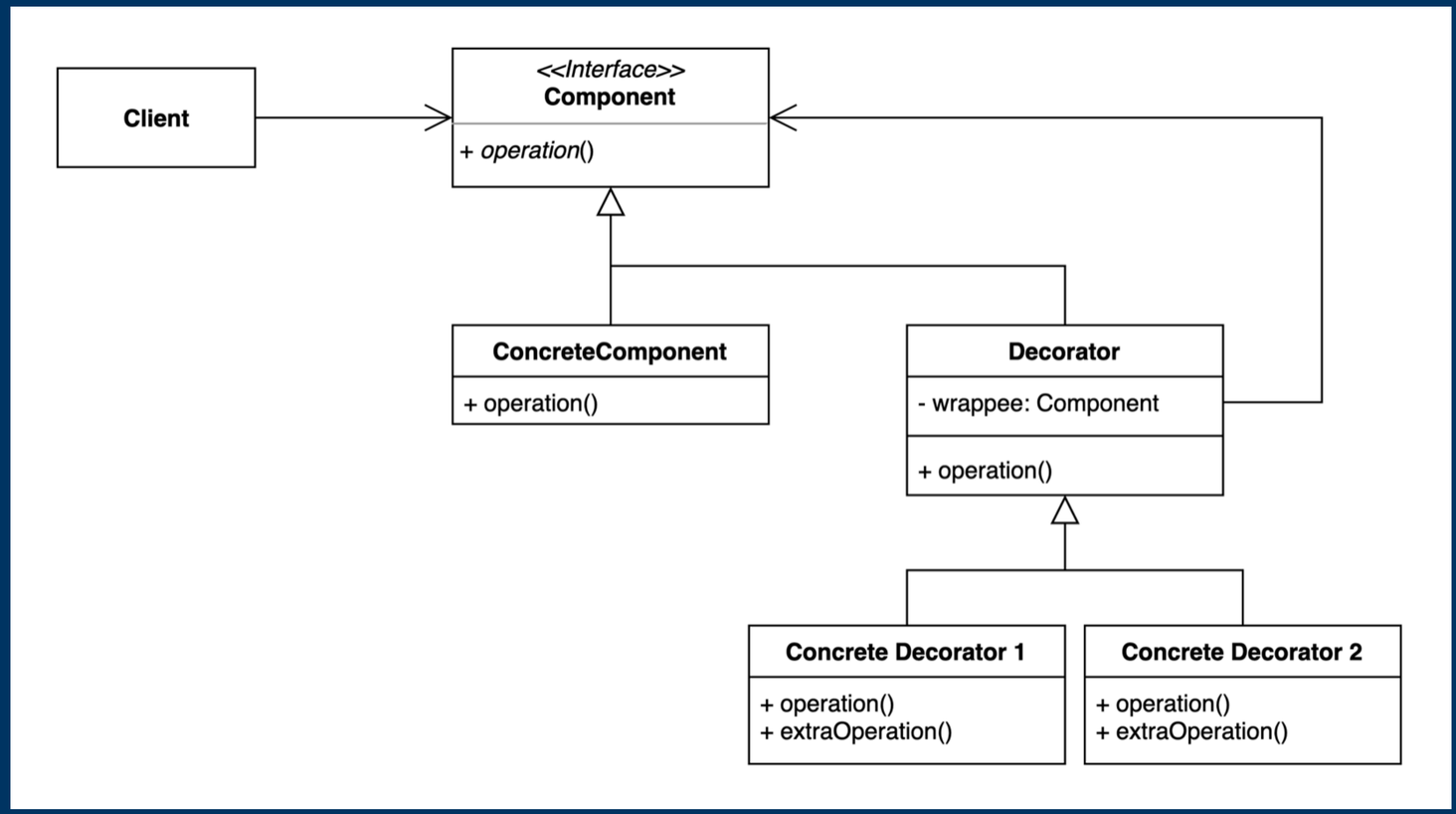
아이템 18. 상속보다는 컴포지션을 사용하라
완벽 공략

- p119, 데코레이터 패턴
- p119, 컴포지션과 전달 조합은 넓은 의미로 위임(delegation)이라고 부른다.
- p119, 콜백 프레임워크와 셀프 문제

완벽 공략 34. 데코레이터(Decorator) 패턴

기존 코드를 변경하지 않고 부가 기능을 추가하는 패턴

- 상속이 아닌 위임을 사용해서 보다 유연하게(런타임에) 부가 기능을 추가하는 것도 가능하다.



완벽 공략 35. 콜백 프레임워크와 셀프 문제

콜백 프레임워크와 래퍼를 같이 사용했을 때 발생할 수 있는 문제

- 콜백 함수: 다른 함수(A)의 인자로 전달된 함수(B)로, 해당 함수(A) 내부에서 필요한 시점에 호출 될 수는 함수 (B)를 말한다.
- 래퍼로 감싸고 있는 내부 객체가 어떤 클래스(A)의 콜백으로(B) 사용되는 경우에 this를 전달한다면, 해당 클래스(A)는 래퍼가 아닌 내부 객체를 호출한다. (SELF 문제)

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

핵심 정리

- 상속용 클래스는 내부 구현을 문서로 남겨야 한다.
  - @implSpec을 사용할 수 있다.
- 내부 동작 중간에 끼어들 수 있는 훅(hook)을 잘 선별하여 protected 메서드로 공개해야 한다.
- 상속용으로 설계한 클래스는 배포 전에 반드시 하위 클래스를 만들어 검증해야 한다.
- 상속용 클래스의 생성자는 재정의 가능한 메서드를 호출해서는 안 된다.
  - Cloneable(아이템 13)과 Serializable(아이템 86)을 구현할 때 조심해야 한다.
- 상속용으로 설계한 클래스가 아니라면 상속을 금지한다.
  - final 클래스 또는 private 생성자

아이템 20. 추상 클래스보다 인터페이스를 우선하라.

핵심 정리: 인터페이스의 장점

- 자바 8부터 인터페이스도 디폴트 메서드를 제공할 수 있다. (완벽 공략 3)
- 기존 클래스도 손쉽게 새로운 인터페이스를 구현해 넣을 수 있다.
- 인터페이스는 믹스인(mixtin) 정의에 안성맞춤이다. (선택적인 기능 추가)
- 계층구조가 없는 타입 프레임워크를 만들 수 있다.
- 래퍼 클래스와 함께 사용하면 인터페이스는 기능을 향상 시키는 안전하고 강력한 수단이 된다. (아이템 18)
- 구현이 명백한 것은 인터페이스의 디폴트 메서드를 사용해 프로그래머의 일감을 덜어 줄 수 있다.

아이템 20. 추상 클래스보다 인터페이스를 우선하라.

핵심 정리: 인터페이스와 추상 골격(skeletal) 클래스

- 인터페이스와 추상 클래스의 장점을 모두 취할 수 있다.
  - 인터페이스 – 디폴트 메서드 구현
  - 추상 골격 클래스 – 나머지 메서드 구현
  - 템플릿 메서드 패턴
- 다중 상속을 시뮬레이트할 수 있다.
- 골격 구현은 상속용 클래스이기 때문에 아이템 19를 따라야 한다.

아이템 20. 추상 클래스보다 인터페이스를 우선하라.

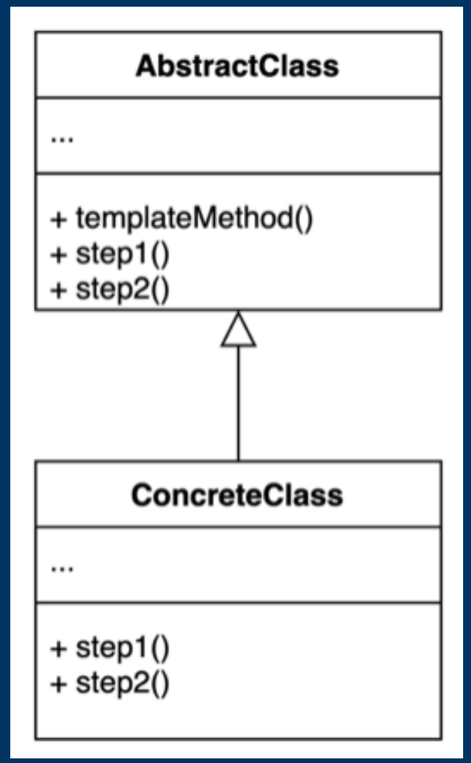
완벽 공략

- p132, 템플릿 메서드 패턴
- p135, 디폴트 메서드는 equals, hashCode, toString 같은 Object 메서드를 재정의할 수 없기 때문이다.

완벽 공략 36. 템플릿 메서드 패턴

알고리듬 구조를 서브 클래스가 확장할 수 있도록 템플릿으로 제공하는 방법.

- 추상 클래스는 템플릿을 제공하고 하위 클래스는 구체적인 알고리듬을 제공한다.



완벽 공략 37. 디폴트 메서드와 Object 메서드

인터페이스의 디폴트 메서드로 Object 메서드를 재정의 할 수 없는 이유

- 디폴트 메서드 핵심 목적은 “인터페이스의 진화”.
- 두 가지 규칙만 유지한다.
  - “클래스가 인터페이스를 이긴다.”
  - “더 구체적인 인터페이스가 이긴다.”
- 토이 예제에나 어울리는 기능이다. 실용적이지 않다.
- 불안정하다.
- 관련 문서
  - https://mail.openjdk.org/pipermail/lambda-dev/2013-March/008435.html

아이템 21. 인터페이스는 구현하는 쪽을 생각해 설계하라.

핵심 정리

- 기존 인터페이스에 디폴트 메서드 구현을 추가하는 것은 위험한 일이다.
  - 디폴트 메서드는 구현 클래스에 대해 아무것도 모른 채 합의 없이 무작정 “삽입” 될 뿐이다.
  - 디폴트 메서드는 기존 구현체에 런타임 오류를 일으킬 수 있다.
- 인터페이스를 설계할 때는 세심한 주의를 기울여야 한다.
  - 서로 다른 방식으로 최소한 세 가지는 구현을 해보자.

완벽 공략 38. ConcurrentModificationException

현재 바뀌면 안되는 것을 수정할 때 발생하는 예외

- 멀티 스레드가 아니라 싱글 스레드 상황에서도 발생할 수 있다. 가령, fail-fast 이터레이터를 사용해 콜렉션을 순회하는 중에 콜렉션을 변경하는 경우.

```
public class ConcurrentModificationException
extends RuntimeException
```

This exception may be thrown by methods that have detected concurrent modification of an object when such modification is not permissible.

For example, it is not generally permissible for one thread to modify a Collection while another thread is iterating over it. In general, the results of the iteration are undefined under these circumstances. Some Iterator implementations (including those of all the general purpose collection implementations provided by the JRE) may choose to throw this exception if this behavior is detected. Iterators that do this are known as *fail-fast* iterators, as they fail quickly and cleanly, rather that risking arbitrary, non-deterministic behavior at an undetermined time in the future.

아이템 22. 인터페이스는 타입을 정의하는 용도로만 사용하라.

핵심 정리

- 상수를 정의하는 용도로 인터페이스를 사용하지 말 것!
  - 클래스 내부에서 사용할 상수는 내부 구현에 해당한다.
  - 내부 구현을 클래스의 API로 노출하는 행위가 된다.
  - 클라이언트에 혼란을 준다.
- 상수를 정의하는 방법
  - 특정 클래스나 인터페이스
  - 열거형
  - 인스턴스화 할 수 없는 유틸리티 클래스

아이템 23. 태그 달린 클래스보다는 클래스 계층 구조를 활용하라

핵심 정리

- 태그 달린 클래스의 단점
  - 쓸데없는 코드가 많다.
  - 가독성이 나쁘다.
  - 메모리도 많이 사용한다.
  - 필드를 final로 선언하려면 불필요한 필드까지 초기화해야 한다.
  - 인스턴스 타입만으로는 현재 나타내는 의미를 알 길이 없다.
- 클래스 계층 구조로 바꾸면 모든 단점을 해결할 수 있다.

아이템 24. 멤버 클래스는 되도록 static으로 만들라.

핵심 정리: 네 종류의 중첩 클래스와 각각의 쓰임

- 정적 멤버 클래스
  - 바깥 클래스와 함께 쓰일 때만 유용한 public 도우미 클래스. 예) Calculator.**Operation**.PLUS
- 비정적 멤버 클래스
  - 바깥 클래스의 인스턴스와 암묵적으로 연결된다.
  - 어댑터를 정의할 때 자주 쓰인다.
  - **멤버 클래스에서 바깥 인스턴스를 참조할 필요가 없다면 무조건 정적 멤버 클래스로 만들자.**
- 익명 클래스
  - 바깥 클래스의 멤버가 아니며, 쓰이는 시점과 동시에 인스턴스가 만들어진다.
  - 비정적인 문맥에서 사용될 때만 바깥 클래스의 인스턴스를 참조할 수 있다.
  - 자바에서 람다를 지원하기 전에 즉석에서 작은 함수 객체나 처리 객체를 만들 때 사용했다.
  - 정적 팩터리 메서드를 만들 때 사용할 수도 있다.
- 지역 클래스
  - 가장 드물게 사용된다.
  - 지역 변수를 선언하는 곳이면 어디든 지역 클래스를 정의해 사용할 수 있다.
  - 가독성을 위해 짧게 작성해야 한다.

아이템 24. 멤버 클래스는 되도록 static으로 만들라.

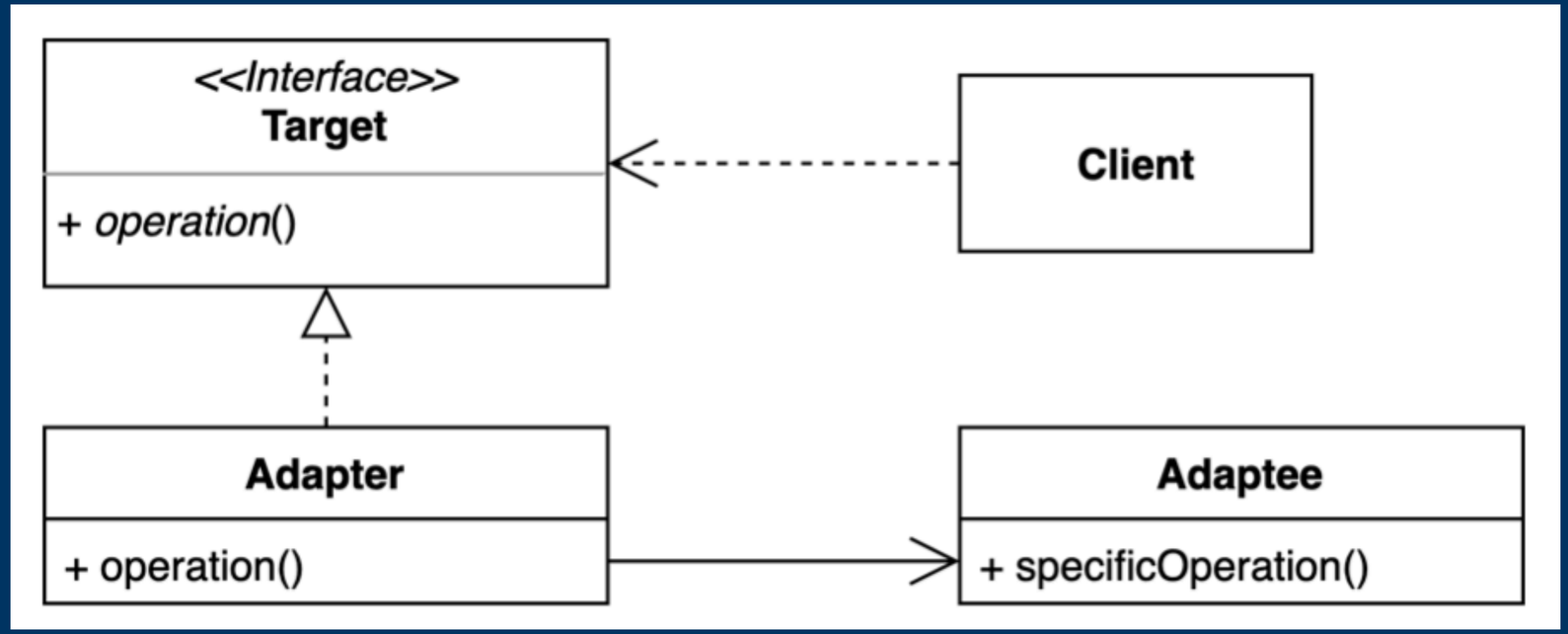
완벽 공략

- p147, 어댑터

완벽 공략 39. 어댑터 패턴

기존 코드를 클라이언트가 사용하는 인터페이스의 구현체로 바꿔주는 패턴

- 클라이언트가 사용하는 인터페이스를 따르지 않는 기존 코드를 재사용할 수 있게 해준다.



아이템 25. 톱 레벨 클래스는 한 파일에 하나만 담으라

핵심 정리

- 한 소스 파일에 톱 레벨 클래스를 여러 개 선언하면 컴파일 순서에 따라 결과가 달라질 수 있다.
- 다른 클래스에 딸린 부차적인 클래스는 정적 멤버 클래스로 만드는 것이 낫다. 읽기 좋으며 private으로 선언해서 접근 범위도 최소한으로 관리할 수 있다.

제네릭

제네릭의 장점을 살리고 단점을 최소화하는 방법

아이템 26. 로 타입은 사용하지 말라.

핵심 정리: 용어 정리

- 로 타입: List
- 제네릭 타입: List<E>
- 매개변수화 타입: List<String>
- 타입 매개변수: E
- 실제 타입 매개변수: String
- 한정적 타입 매개변수: List<E extends Number>
- 비한정적 와일드카드 타입: Class<?>
- 한정적 와일드카드 타입: Class<? extends Annotation>

아이템 26. 로 타입은 사용하지 말라.

핵심 정리: 매개변수화 타입을 사용해야 하는 이유

- 런타임이 아닌 컴파일 타임에 문제를 찾을 수 있다. (안정성)
- 제네릭을 활용하면 이 정보가 주석이 아닌 타입 선언 자체에 녹아든다. (표현력)
- “로 타입”을 사용하면 안정성과 표현력을 잃는다.
- 그렇다면 자바는 “로 타입”을 왜 지원하는가?
- List와 List<Object>의 차이는?
- Set과 Set<?>의 차이는?
- 예외: class 리터럴과 instanceof 연산자

아이템 26. 로 타입은 사용하지 말라.
완벽 공략

- p156, 마이그레이션 호환성을 위해 로 타입을 지원하고 제네릭 구현에는 소거 방식을 사용하기로 했다. (아이템 28)
- p158, 제네릭 메서드 (아이템 30)
- p158, 한정적 와일드카드 타입 (아이템 31)
- Generic DAO 만들기

완벽 공략 40. GenericRepository

자바 Generic을 활용한 중복 코드 제거 예제

```java
public class AccountRepository {

    private Set<Account> accounts;

    public AccountRepository() {
        this.accounts = new HashSet<>();
    }

    public Optional<Account> findById(Long id) {
        return accounts.stream().filter(a -> a.getId().equals(id)).findAny();
    }

    public void add(Account account) {
        this.accounts.add(account);
    }
}
```

```java
public class MessageRepository {

    private Set<Message> messages;

    public MessageRepository() {
        this.messages = new HashSet<>();
    }

    public Optional<Message> findById(Long id) {
        return messages.stream().filter(a -> a.getId().equals(id)).findAny();
    }

    public void add(Message message) {
        this.messages.add(message);
    }
}
```

```java
public class AccountRepository extends GenericRepository<Account> {

}
```

```java
public class MessageRepository extends GenericRepository<Message> {

}
```

아이템 27. 비검사 경고를 제거하라.

핵심 정리

- “비검사 (unchecked) 경고”란?
    - 컴파일러가 타입 안정성을 확인하는데 필요한 정보가 충분치 않을 때 발생시키는 경고.
- 할 수 있는 한 모든 비검사 경고를 제거하라.
- 경고를 제거할 수 없지만 안전하다고 확신한다면 @SuppressWarnings(“unchecked”) 애노테이션을 달아 경고를 숨기자.
- @SuppressWarnings 애너테이션은 항상 가능한 한 좁은 범위에 적용하자.
- @SuppressWarnings(“unchecked”) 애너테이션을 사용할 때면 그 경고를 무시해도 안전한 이유를 항상 주석으로 남겨야 한다.

완벽 공략 41. 애너테이션

자바 애너테이션을 정의하는 방법

- @Retention: 애노테이션의 정보를 얼마나 오래 유지할 것인가.
  - Runtime, Source, Class
- @Target: 애노테이션을 사용할 수 있는 위치.
  - Type, Field, Method, Parameter, …

아이템 28. 배열보다는 리스트를 사용하라.

핵심 정리: 배열과 제네릭은 잘 어울리지 않는다.

- 배열은 공변 (covariant), 제네릭은 불공변
- 배열은 실체화(reify) 되지만, 제네릭은 실체화 되지 않는다. (**소거**)
- new Generic<타입>[배열] 은 컴파일 할 수 없다.
- 제네릭 소거: 원소의 타입을 컴파일 타임에만 검사하며 런타임에는 알 수 없다.

완벽 공략 42. @SafeVaragas

생성자와 메서드의 제네릭 가변인자에 사용할 수 있는 애노테이션

- 제네릭 가변인자는 근본적으로 타입 안전하지 않다. (가변인자가 배열이니까, 제네릭 배열과 같은 문제)
- 가변 인자 (배열)의 내부 데이터가 오염될 가능성이 있다.
- @SafeVarargs를 사용하면 가변 인자에 대한 해당 오염에 대한 경고를 숨길 수 있다.
- 아이템 32. 제네릭과 가변인수를 함께 쓸 때는 신중하라.

아이템 29. 이왕이면 제네릭 타입으로 만들라.

핵심 정리

- 배열을 사용하는 코드를 제네릭으로 만들 때 해결 책 두 가지.
- 첫번째 방법: 제네릭 배열 (E[]) 대신에 Object 배열을 생성한 뒤에 제네릭 배열로 형변환 한다.
  - 형변환을 배열 생성시 한 번만 한다.
  - 가독성이 좋다.
  - 힙 오염이 발생할 수 있다.
- 두번째 방법: 제네릭 배열 대신에 Object 배열을 사용하고, 배열이 반환한 원소를 E로 형변환 한다.
  - 원소를 읽을 때 마다 형변환을 해줘야 한다.

완벽 공략 43. 한정적 타입 매개변수

Bounded Type Parameters

- 매개변수화 타입을 특정한 타입으로 한정짓고 싶을 때 사용할 수 있다.
  - 〈E extends Number〉, 선언할 수 있는 제네릭 타입을 Number를 상속(extnds)했거나 구현한(implements)한 클래스로 제한한다.
- 제한한 타입의 인스턴스를 만들거나, 메서드를 호출할 수도 있다.
  - 〈E extedns Number〉, Number 타입이 제공하는 메서드를 사용할 수 있다.
- 다수의 타입으로 한정 할 수 있다. 이 때 클래스 타입을 가장 먼저 선언해야 한다.
  - 〈E extedns Numebr & Serializable〉, 선언할 제네릭 타입은 Integer와 Number를 모두 상속 또는 구현한 타입이어야 한다.

아이템 30. 이왕이면 제네릭 메서드로 만들라.

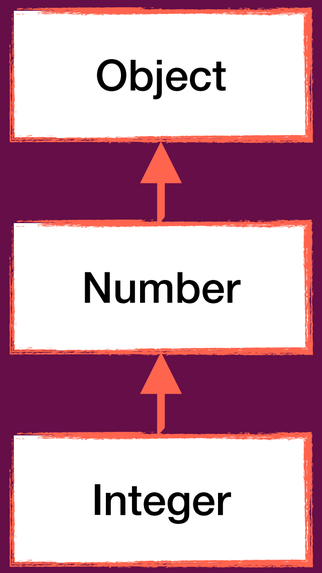
핵심 정리

- 매개변수화 타입을 받는 정적 유틸리티 메서드
  - 한정적 와일드카드 타입(아이템 31)을 사용하면 더 유연하게 개선할 수 있다.
- 제네릭 싱글턴 팩터리
  - (소거 방식이기 때문에) 불변 객체 하나를 어떤 타입으로든 매개변수화 할 수 있다.
- 재귀적 타입 한정
  - 자기 자신이 들어간 표현식을 사용하여 타입 매개변수의 허용 범위를 한정한다.

아이템 31. 한정적 와일드카드를 사용해 API 유연성을 높이라.

핵심 정리 1: Chooser와 Union API 개선

- PECS: Producer-Extends, Consumer-Super

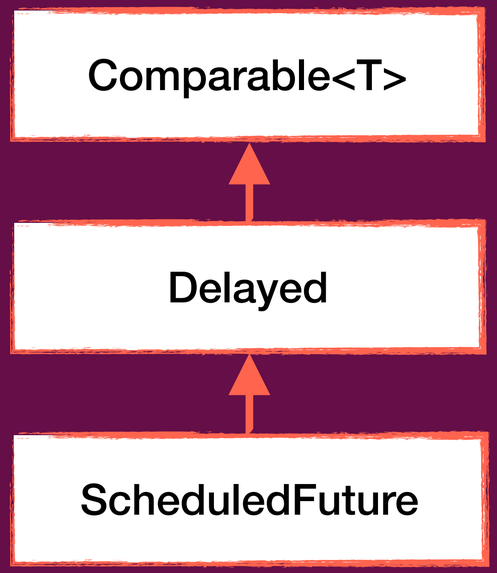


- Producer-Extends
  - Object의 컬렉션 Number나 Integer를 넣을 수 있다.
  - Number의 컬렉션에 Integer를 넣을 수 있다.
- Consumer-Super
  - Integer의 컬렉션의 객체를 꺼내서 Number의 컬렉션에 담을 수 있다.
  - Number나 Integer의 컬렉션의 객체를 꺼내서 Object의 컬렉션에 담을 수 있다.

아이템 31. 한정적 와일드카드를 사용해 API 유연성을 높이라

핵심 정리 2: Comparator와 Comparable은 소비자

- Comparable을 직접 구현하지 않고, 직접 구현한 다른 타입을 확장한 타입을 지원하려면 와일드카드가 필요하다.



- ScheduledFuture는 Comparable을 직접 구현하지 않았지만, 그 **상위 타입 (Delayed)**이 구현하고 있다.

아이템 31. 한정적 와일드카드를 사용해 API 유연성을 높이라

핵심 정리 3: 와일드카드 활용 팁

- 메서드 선언에 타입 매개변수가 한 번만 나오면 와일드카드로 대체하라.
  - 한정적 타입이라면 한정적 와일드카드로
  - 비한정적 타입이라면 비한정적 와일드카드로
- 주의!
  - 비한정적 와일드카드(?)로 정의한 타입에는 null을 제외한 아무것도 넣을 수 없다.

완벽 공략 44. 타입 추론

Type Inference

- 타입을 추론하는 컴파일러의 기능
- 모든 인자의 가장 구체적인 공통 타입 (most specific type)
- 제네릭 메서드와 타입 추론: 메서드 매개변수를 기반으로 타입 매개변수를 추론할 수 있다.
- 제네릭 클래스 생성자를 호출할 때 다이아몬드 연산자 〈〉를 사용하면 타입을 추론한다.
- 자바 컴파일러는 "타겟 타입"을 기반으로 호출하는 제네릭 메서드의 타입 매개볓수를 추론한다.
  - 자바 8에서 "타겟 타입"이 "메서드의 인자"까지 확장되면서 이전에 비해 타입 추론이 강화되었다.

아이템 32. 제네릭과 가변인수를 함께 쓸 때는 신중하라.

핵심 정리

- 제네릭 가변인수 배열에 값을 저장하는 것은 안전하지 않다.
  - 힙 오염이 발생할 수 있다. (컴파일 경고 발생)
  - 자바7에 추가된 @SafeVarargs 애노테이션을 사용할 수 있다.
- 제네릭 가변인수 배열의 참조를 밖으로 노출하면 힙 오염을 전달할 수 있다.
  - 예외적으로, @SafeVarargs를 사용한 메서드에 넘기는 것은 안전하다.
  - 예외적으로, 배열의 내용의 일부 함수를 호출하는 일반 메서드로 넘기는 것은 안전하다.
- 아이템 28의 조언에 따라 가변인수를 List로 바꾼다면
  - 배열없이 제니릭만 사용하므로 컴파일러가 타입 안정성을 보장할 수 있다.
  - @SafeVarargs 애너테이션을 사용할 필요가 없다.
  - 실수로 안전하다고 판단할 걱정도 없다.

완벽 공략 45. ThreadLocal

쓰레드 지역 변수

- 모든 멤버 변수는 기본적으로 여러 쓰레드에서 공유해서 쓰일 수 있다. 이때 쓰레드 안전성과 관련된 여러 문제가 발생할 수 있다.
  - 경합 또는 경쟁조건 (Race-Condition)
  - 교착상태 (deadlock)
  - Livelock
- 쓰레드 지역 변수를 사용하면 동기화를 하지 않아도 한 쓰레드에서만 접근 가능한 값이기 때문에 안전하게 사용할 수 있다.
- 한 쓰레드 내에서 공유하는 데이터로, 메서드 매개변수에 매번 전달하지 않고 전역 변수처럼 사용할 수 있다.

완벽 공략 46. ThreadLocalRandom

스레드 지역 랜덤값 생성기

- java.util.Random은 멀티 스레드 환경에서 CAS(CompareAndSet)로 인해 실패 할 가능성이 있기 때문에 성능이 좋지 않다.

```
protected int next(int bits) {
    long oldseed, nextseed;
    AtomicLong seed = this.seed;
    do {
        oldseed = seed.get();
        nextseed = (oldseed * multiplier + addend) & mask;
    } while (!seed.compareAndSet(oldseed, nextseed));
    return (int)(nextseed >>> (48 - bits));
}
```

```
public synchronized int compareAndSwap(int expectedValue, int newValue)
{
    int readValue = value;
    if (readValue == expectedValue)
        value = newValue;
    return readValue;
}
```

- Random 대신 ThreadLocalRandom을 사용하면 해당 스레드 전용 Random이라 간섭이 발생하지 않는다.

아이템 33. 타입 안전 이종 컨테이너를 고려하라.

핵심 정리: 타입 토큰을 사용한 타입 안전 이종 컨테이너

- 타입 안전 이종 컨테이너: 한 타입의 객체만 담을 수 있는 컨테이너가 아니라 여러 다른 타입 (이종)을 담을 수 있는 타입 안전한 컨테이너.
- 타입 토큰: String.class 또는 Class〈String〉
- 타입 안전 이종 컨테이너 구현 방법: 컨테이너가 아니라 “키”를 매개변수화 하라!

```
public class Favorites {

    private Map<Class<?>, Object> favorites = new HashMap<>();

    public <T> void putFavorite(Class<T> type, T instance) {
        favorites.put(Objects.requireNonNull(type), instance);
    }

    public <T> T getFavorite(Class<T> type) {
        return type.cast(favorites.get(type));
    }

}
```

완벽 공략 47. 수퍼 타입 토큰

익명 클래스와 제네릭 클래스 **상속**을 사용한 **타입 토큰**

- 닐 게프터의 슈퍼 타입 토큰
  - https://gafter.blogspot.com/2006/12/super-type-tokens.html
  - https://gafter.blogspot.com/2007/05/limitation-of-super-type-tokens.html
- 상속을 사용한 경우 제네릭 타입을 알아낼 수 있다. 이 경우에는 제네릭 타입이 제거되지 않기 때문에…

아이템 33. 타입 안전 이종 컨테이너를 고려하라.

핵심 정리: 한정적 타입 토큰

- 한정적 타입 토큰을 사용한다면, 이종 컨테이너에 사용할 수 있는 타입을 제한할 수 있다.
  - AnnotatedElement.**<T extends Annotation>** T getAnnotation(Class<T> annotationClass);
- asSubclass 메서드
  - 메서드를 호출하는 Class 인스턴스를 인수로 명시한 클래스로 형변환 한다.

감사합니다.

3부(열거 타입과 애너테이션, 람다와 스트림, 메서드)를 기대해 주세요.

백기선 올림