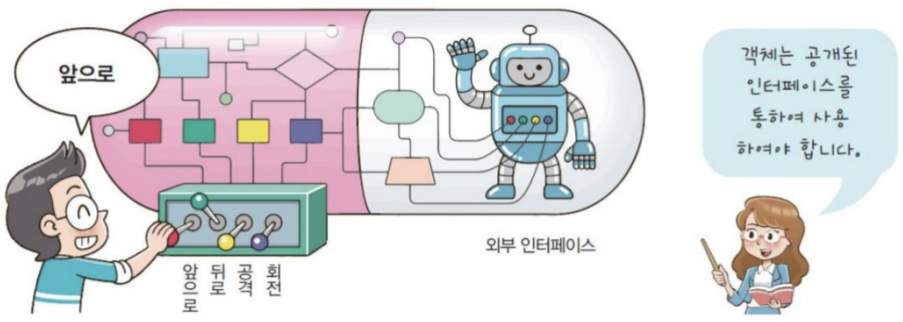
아이템 15 클래스와 멤버의 접근 권한을 최소화하라

💡 잘 설계된 컴포넌트

- 클래스 내부 데이터와 내부 구현 정보를 외부 컴포넌트로부터 얼마나 잘 숨겼는가
- 구현과 API를 깔끔하게 분리해야 한다 → 정보은닉/캡슐화 는 **소프트웨어 설계의 근간이 되는 원리**다.



이미지 출처 : https://radait.tistory.com/5

💡 캡슐화(정보 은닉)의 장점

1. **시스템 개발 속도 향상**

   → 여러 컴포넌트를 병렬로 개발할 수 있기 때문이다.

2. **시스템 관리 비용 절약** (컴포넌트 교체 비용 절감)

   → 컴포넌트 이해도가 높아져 디버깅할 수 있고, 컴포넌트 교체 부담도 적기 때문이다.

3. **성능 최적화에 도움**

   → 완성된 시스템을 프로파일링해 최적화할 컴포넌트를 정한 후 다른 컴포넌트에 영향을 주지 않고 해당 컴포넌트만 최적화할 수 있기 때문이다.

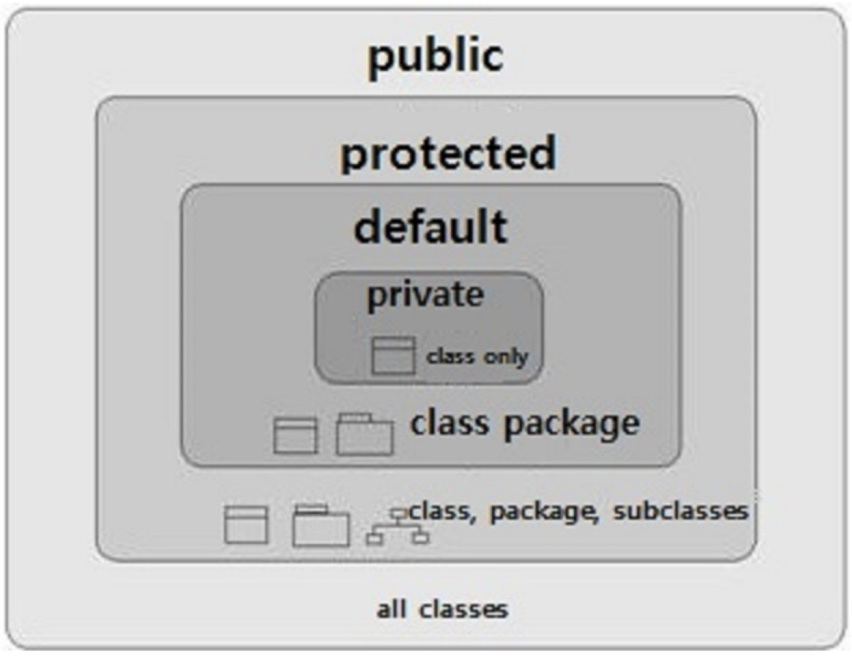
4. **소프트웨어 재사용성**을 높인다

   → 의존성이 낮은 컴포넌트라면 다른 환경에서도 유용하게 쓰일 가능성이 높기 때문이다.

5. **큰 시스템을 제작하는 난이도를 낮춰준다**

   → 개별 컴포넌트의 동작을 검증할 수 있기 때문이다.

> 자바에서 정보은닉의 핵심은 접근 제한자를 제대로 활용하는 것이다.



이미지 출처 : https://whatisthenext.tistory.com/32

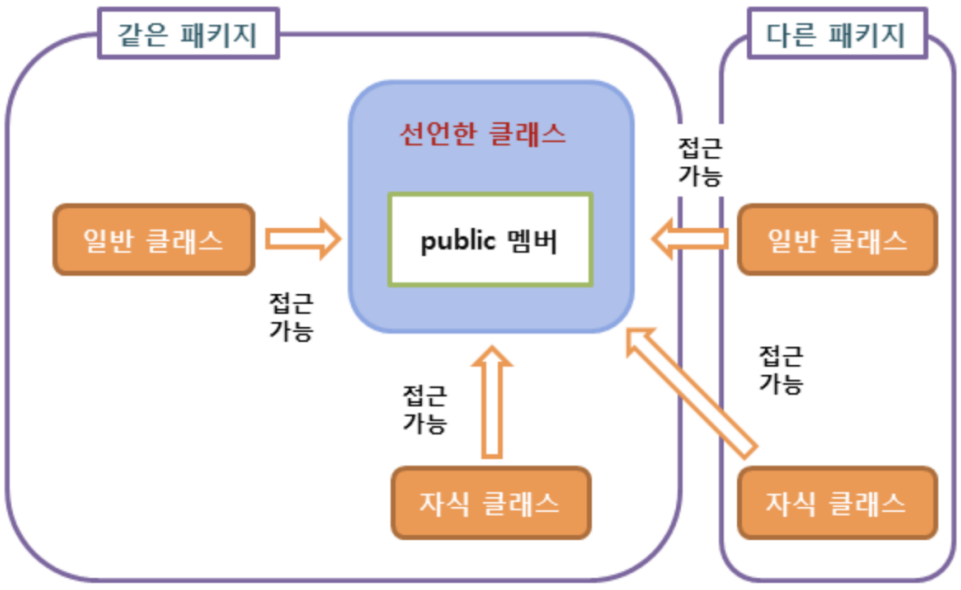
💡 캡슐화(정보 은닉)의 핵심 - 접근 제한자

- 멤버(필드, 메서드, 중첩 클래스, 중첩 인터페이스)에 부여할 수 있는 접근 수준은 4가지다
  1. **private** : 멤버를 선언한 톱 레벨 클래스에서만 접근 가능하다.
  2. **package-private**: 멤버가 소속된 패키지 안의 모든 클래스에서 접근할 수 있다. 접근 제한자를 명시하지 않았을 때 적용되는 패키지 접근 수준이다. (단, 인터페이스의 멤버는 기본적으로 public)
  3. **protected** : package-private 의 접근범위를 포함하며, 이 멤버를 선언한 클래스의 하위 클래스에서도 접근할 수 있다.
  4. **public** : 모든 곳에서 접근 가능하다.

> 기본 원칙은 모든 클래스와 멤버의 접근성을 가능한 좁히는 것이다.

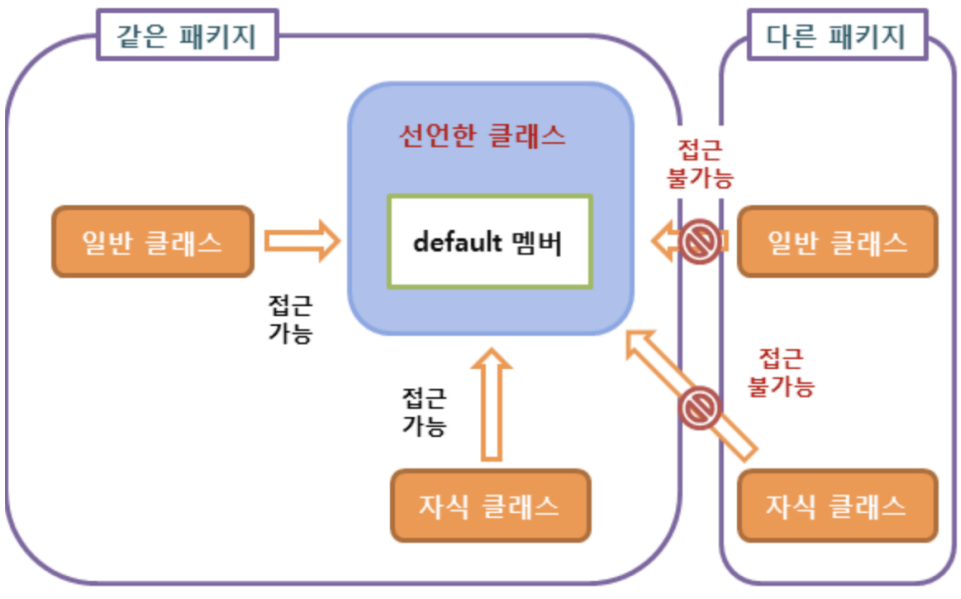
톱레벨 클래스, 인터페이스 → package-private / public

- 톱레벨 클래스(가장 바깥클래스)와 인터페이스에 부여할 수 있는 접근 수준은 package-private과 public 뿐이다.
  - **public** : **공개 API** 가 된다.



이미지 출처 : http://tcpschool.com/java/java_modifier_accessModifier

  - 하위 호환을 위해 영원히 관리해줘야 한다.
- **package-private** : 해당 패키지 안에서만 이용할 수 있다.



  - 패키지 외부에서 쓸 이유가 없다면 package-private으로 선언하자.

    → API가 아닌 **내부 구현이 되어 언제든 수정 가능**하다.

    ⇒ 클라이언트에 아무런 피해 없이 **다음 릴리즈에서 수정, 교체, 제거 할 수 있다**.

private static 중첩 클래스 (Static Nested Class)

- 한 클래스에서만 사용하는 **package-private 톱 레벨 클래스나 인터페이스**는 **private static 중첩 클래스**로 만들자

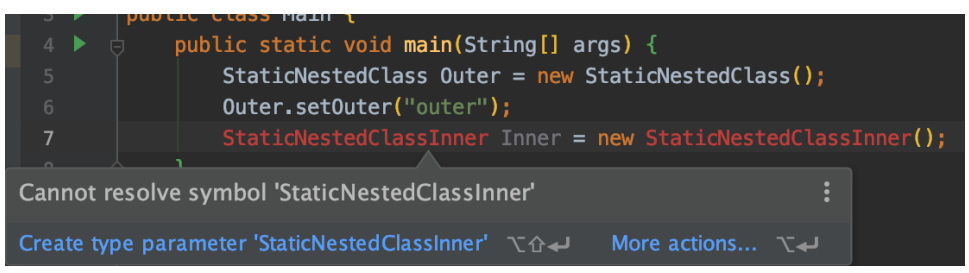
```
public class StaticNestedClass {
    /*
    중첩 클래스 (Static Nested Class)
    클래스 선언 안에 또 다른 클래스 선언이 있는 상태
    정적 멤버로 등록된 중첩 클래스
```

```
    */
    private String outer;

    public void setOuter(String outer) {
        this.outer = outer;
    }

    private static class InnerClass {
        private String inner;

        public void setInner(String inner) {
            this.inner = inner;
        }
    }
}
```



- 톱레벨로 두면 같은 패키지의 모든 클래스가 접근할 수 있지만, **private static**으로 중첩시키면 **바깥 클래스 하나에서만 접근**할 수 있다.
- 더 중요한 것은 public 일 필요 없는 클래스의 접근 수준을 **package-private 톱레벨 클래스로 좁히는 일**이다.
    - public 클래스는 그 패키지의 API 인 반면, package-private 톱 레벨 클래스는 내부 구현에 속하기 때문이다.

멤버 필드

- 클래스의 공개 API를 세심히 설계한 후, 그 외의 모든 멤버는 **private**으로 만들자

  그런 다음 **오직 같은 패키지의 다른 클래스가 접근해야 하는 멤버**에 대하여 private 제한자를 제거해 **package-private**으로 만들자.

  이때, 권한을 풀어주는 일을 자주 하게 되면 시스템에서 컴포넌트를 더 분해해야 하는건 아닌지 고민해 봐야 한다.

- **private과 package-private 멤버**는 모두 해당 클래스의 구현에 해당하므로 보통은 공개 API에 영향을 주지 않는다.
    - 단, **Serializable**을 구현한 클래스에서는 그 필드들도 의도치 않게 공개 API가 될 수도 있다.
- **public 클래스**에서는 멤버의 접근 수준을 **package-private** 에서 **protected**로 바꾸는 순간 그 멤버에 접근할 수 있는 대상 범위가 엄청나게 넓어진다.

  public 클래스의 **protected(상속) 멤버**는 공개 API이므로 영원히 지원돼야 하며 내부 동작 방식을 API 문서에 적어 사용자에게 공개해야 할 수도 있다. → **따라서 protected 멤버의 수는 적을 수록 좋다.**
- 상위 클래스의 메서드를 **재정의**할 때는 그 **접근 수준을 상위 클래스에서보다 좁게 설정할 수 없다.**

```
class Parent {
    protected void accessModifier() {
        System.out.println("Class Modifier " +this.getClass().getModifiers());
    }
}
```

상위 클래스 접근제한자가 protected 인데 하위 클래스가 package-private으로 변경할 수 없다

→ **상위 클래스의 인스턴스는 하위 클래스의 인스턴스로 대체해 사용할 수 있어야 한다**는 규칙(리스코프 치환 원칙, 아이템10)을 위반하기 때문이다.

단, 클래스가 인터페이스를 구현하는 건 예외로 이때 **클래스는 인터페이스가 정의한 모든 메서드를 public 으로 선언**해야 한다.

주의사항

1. **테스트만을 위해** 클래스, 인터페이스, 멤버를 *공개 API*로 만들어서는 안된다.
    - 테스트 코드는 테스트 대상과 같은 패키지에 두면 package-private 요소에 접근할 수 있기 때문이다.
2. **public 클래스의 인스턴스 필드는 되도록 public 이 아니어야 한다**.
    - 필드가 가변 객체(Collections 이나 배열)를 참조하거나 final이 아닌 인스턴스 필드를 public으로 선언하면 **불변식을 보장할 수 없게 된다.**

      필드가 수정될 때 (락 획득 같은) 다른 작업을 할 수 없으므로 public 가변 필드를 갖는 클래스는 일반적으로 스레드 안전(thread safe)하지 않다. 심지어 필드가 final이면서 불변 객체를 참조하더라도 문제는 여전히 남는다.

      내부 구현을 바꾸고 싶어도 그 public 필드를 없애는 방식으로는 리팩터링 할 수 없게된다.
    - 예외! **해당 클래스가 표현하는 추상 개념을 완성하는데 꼭 필요한 구성요소로써의 상수**라면 **public static final 필드로 공개해도** 좋다.

      관례상 이런 상수의 이름은 대문자 알파벳으로 쓰며, 각 단어 사이에 밑줄(_)을 넣는다.이런 필드는 반드시 기본 타입 값이나 불변 객체를 참조해야 한다.
    - **길이가 0이 아닌 배열은 모두 변경가능하니 주의**해야 한다.

      클래스에서 **public static final 배열 필드를 두거나 이 필드를 반환하는 접근자 메서드를 제공해서는 안된다**.
    - 클라이언트에서 그 배열의 내용을 수정할 수 있게 된다.

```
public static final Thing[] VALUES = {...};
```

- 어떤 IDE가 생성하는 접근자는 private 배열 필드의 참조를 반환해 위와 같은 문제를 똑같이 일으키니 주의해야 한다.

  (VALUES에 대한 참조를 변경할 수는 없지만, 배열내의 내용을 변경할 수는 있다.)
- 해결방법 1 : 앞 코드의 public 배열을 **private**으로 만들고 **public 불변 리스트**를 추가한다

```
private static final Thing[] PRIVATE_VALUES = {...};
public static final List<Thing> VALUES =
  Collections.unmodifiableList(Arrays.asList(PRIVATE_VALUES));
```

- 해결방법 2 : 배열을 private 으로 만들고 그 복사본을 반환하는 pulbic 메서드를 추가한다(방어적 복사)

```
private static final Thing[] PRIVATE_VALUES = {...};
public static final Thing[] valuse(){
  return PRIVATE_VALUES.clone();
}
```

💡 Java9 의 모듈 시스템

- 자바 9에서는 모듈 시스템이라는 개념이 도입되면서 암묵적 접근 수준이 추가되었다.
    - 자바 9 부터의 접근 제한자
        - private, package-private, protected 는 이전과 동일
        - 모듈 내부의 public : 모듈 내부의 모든 곳에서 접근가능하다.
        - required의 public : 모듈에 종속하는 모듈의 모든 패키지 내의 클래스에 접근할 수 있다.
        - export public : module-info.java에서 제공하는 모든 public에 접근할 수 있다.
    - 패키지가 클래스의 묶음이듯, 모듈은 패키지들의 묶음이다.
    - 모듈은 자신에 속하는 패키지 중 공개(export)할 것들을 (관례상 module-info.java에) 선언한다. protected 또는 public 멤버라도 해당 패키지를 공개하지 않았다면 모듈 외부에서는 접근할 수 없다.
    - 모듈 시스템을 활용하면 클래스를 외부에 공개하지 않으면서도 같은 모듈을 이루는 패키지 사이에 자유롭게 공유할 수 있다.
    - 대표적인 예는 JDK
        - 자바 라이브러리에서 공개하지 않은 패키지들은 해당 모듈 밖에서는 절대로 접근할 수 없다.

- 아직 이른 개념이기때문에 꼭 필요한 경우가 아니라면 당분간은 사용하지 않는게 좋다.

▼ 책에 나온 모듈 시스템의 내용 더보기

- 숨겨진 패키지 안에 있는 public 클래스의 **public 혹은 protected 멤버와 관련**있는 암묵적 접근 수준은 각각 public, protected 수준과 같으나 그 효과가 모듈 내부로 한정되는 변종이다. → 이런 형태로 공유해야 하는 상황은 흔치 않다. 그런 상황이더라도 패키지들 사이에서 클래스들을 재배치하면 대부분 해결된다.
- 모듈에 적용되는 새로운 두 접근 수준은 상당히 주의해서 사용해야한다. 개발자 모듈의 JAR 파일을 자신의 모듈경로가 아닌 애플리케이션의 classpath에 두면 그 모듈안의 모든 패키지는 마치 모듈이 없는 것처럼 작동된다.
- 모듈이 공개했는지 여부와 상관없이 public 클래스가 선언한 모든 public, protected 멤버를 모듈밖에서도 접근할 수 있게 한다.
- 모듈은 여러면에서 자바 프로그래밍에 영향을 준다.
    - 패키지들을 모듈 단위로 묶고, 모듈 선언에 패키지들의 모든 의존성을 명시한다. 소스트리를 재배치하고, 모듈 안으로부터 일반 패키지로로의 모든 접근에 특별한 조치를 취해야 한다.

핵심 정리

- 프로그램 요소의 접근성은 가능한 한 최소한으로 해야한다. 꼭 필요한 것만 골라 최소한의 public API를 설계하자.
- public 클래스는 상수용 public static final 필드 외에는 어떠한 public 필드도 가져서는 안 된다.
- public static final 필드가 참조하는 객체가 불변인지 확인하라.