아이템 31) 한정적 와일드카드를 사용해 API 유연성을 높이라

- 매개변수화 타입은 불공변이다
  - Type1, Type2 가 있을 때 List<Type1> 은 List<Type2> 의 하위, 상위타입도 아님!
  - ex) **Type1 : String / Type2 : Object**

PECS : 와일드 카드 타입 사용 기본 원칙

[Producer 예시]

```
public class Stack<E> { // Stack<Number>
  public void push(E e); // push(Integer e);
}
```

```
public void pushAll(Iterable<E> src){ // Iterable<Integer> src
  for(E e : src) // Integer e : src
    push(e);
}
```

- 입력 매개변수 (src) 로부터 Stack 으로 원소를 옮겨담아 E 인스턴스를 생산하기 때문에, src 를 `생산자` 라고 함

```
Stack<Number> s = new Stack<>();
Iterable<Integer> integers = ...;
s.pushAll(integers);
```

- Integer 는 Number 의 하위 타입이니까 잘 동작해야 할 것 같지만, 실제로는 Imcompatible types 에러 발생
  → 매개변수화 타입이 불공변이기 때문 !
- 이런 상황에 대처할 수 있는 방법 : `한정적 와일드카드 타입`
  - 한정적 와일드카드를 사용하면 **특정 타입을 기준으로 상한 범위와 하한 범위를 지정**함으로써 호출 범위를 확장 또는 제한할 수 있음
  - Iterable<E> 는 “E의 Iterable” 이 아니라 “**E의 하위타입의 Iterable**” (Iterable<? extends E>)
  - 하위타입에는 자기 자신도 포함

```
public void pushAll(Iterable<? extends E> src) {
  for(E e : src)
    push(e);
}
```

[Consumer 예시]

```
public class Stack<E> { // Stack<Number>
  public E pop();
}
```

```
public void popAll(Collection<E> dst){ // Collection<Object>
  while(!isEmpty())
    dst.add(pop()); // Collection<Object>.add(Number);
}
```

- 입력 매개변수 (dst) 에 Stack 의 원소를 옮겨담아서 E 인스턴스를 소비하기 때문에, dst 를 `소비자` 라고 함

```
Stack<Number> s = new Stack<>();
Collection<Object> objects = ...;
s.popAll(objects);
```

- Collection<E> 는 “E의 Collection” 이 아니라 “**E의 상위타입의 Collection**”
- **유연성을 극대화하려면 원소의 생산자, 소비자용 입력 매개변수에 와일드카드 타입을 사용하라**
- PECS (펙스) : **Producer-extends, Consumer-super** **/** 와일드카드 타입을 사용하는 기본 원칙

상한 / 하한 경계 와일드 카드

- `상한 경계 와일드카드` : 와일드카드 타입에 extends를 사용해서 와일드카드 타입의 최상위 타입을 정의함으로써 상한 경계를 설정

```
class MyGrandParent {}
class MyParent extends MyGrandParent {}
class MyChild extends MyParent {}
```

```
void printCollection(Collection<? extends MyParent> c) {
    // 컴파일 에러
    for (MyChild e : c) {
        System.out.println(e);
    }

    for (MyParent e : c) {
        System.out.println(e);
    }

    for (MyGrandParent e : c) {
        System.out.println(e);
    }

    for (Object e : c) {
        System.out.println(e);
    }
}
```

- **<? extends MyParent>**으로 가능한 타입은 MyParent와 모든 (미지의) MyParent 자식 클래스들
  - 미지의 MyParent 자식 클래스 == 자식이 어떤 타입인지 알 수 없다는 것!
  - 그 타입이 MyChild 일 수도 있지만, 아닐 수도 있다.
  - 또 다른 MyParent의 자식인 AnotherChild 라는 클래스가 있다면 MyChild 타입으로 꺼내려고 시도할 때 컴파일 에러가 발생

```
void addElement(Collection<? extends MyParent> c) {
    c.add(new MyChild());        // 불가능(컴파일 에러)
    c.add(new MyParent());       // 불가능(컴파일 에러)
    c.add(new MyGrandParent());  // 불가능(컴파일 에러)
    c.add(new Object());         // 불가능(컴파일 에러)
}
```

- 하위 타입으로는 MyChild가 올 지, AnotherChild 가 올 지 모름. 어떤 하위 타입이 들어올 지 몰라서 **하위 타입을 결정할 수 없어서** 컴파일 에러 발생
- MyParent 의 상한 타입은 당연히 추가 할 수 없음

- `하한 경계 와일드카드` : super를 사용해 와일드카드의 최하위 타입을 정의하여 하한 경계를 설정

```
void addElement(Collection<? super MyParent> c) {
    c.add(new MyChild());
    c.add(new MyParent());
    c.add(new MyGrandParent());  // 불가능(컴파일 에러)
    c.add(new Object());         // 불가능(컴파일 에러)
}
```

- 컬렉션 C가 갖는 타입은 적어도 MyParent의 부모 타입이니까, MyParent의 자식 타입이라면 안전하게 컬렉션에 추가할 수 있고, 부모 타입인 경우에만 컴파일 에러가 발생

- 반환타입에는 한정적 와일드카드 타입을 사용하면 안되는데, 이유는 클라이언트 코드에서도 와일드카드 타입을 사용해야 하기 때문

와일드카드 타입의 실제 타입을 알려주는 메서드를 private 도우미 메소드로 따로 작성하자

```
public static <E> void swap(List<E> list, int i, int j);
public static void swap(List<?> list, int i, int j);
```

- public API 라면 간단한 2번째가 나은데, 어떤 리스트든 메서드에 넘기면 명시한 인덱스 원소를 바꿔줄 것이기 때문 / 신경써야 할 타입 매개변수도 없다
- **메서드 선언에 타입 매개변수가 1번만 나오면 와일드 카드로 대체하라 !**
- `List<E>`은 타입을 명확하게 지정하는데 사용
  - 타입 매개변수가 실제 타입으로 대체되는 제네릭 클래스나 메서드를 나타내서 , 컴파일 시에 실제 타입으로 대체
- `List<?>`은 와일드카드 타입으로 “알 수 없는 타입"의 리스트를 나타내서, 다양한 타입을 다룰 때 사용
  - 모든 종류의 리스트를 처리할 수 있지만, 그 내부에서는 리스트의 요소에 대한 구체적인 타입을 알지 못해서, 메서드 내부에서는 리스트의 요소를 읽는 것은 가능하지만 값을 추가하거나 변경하는 것은 허용되지 않음

```
public static void swap(List<?> list, int i, int j) {
  swapHelper(list, i, j);
}

private static <E> void swapHelper(List<E> list, int i, int j){
  list.set(i, list.set(j, list,get(i)));
}
```

- 결국 한정적 와일드카드를 사용해 API 유연성을 높이라는 의미는,
  List<?> 를 통해서 다양한 타입을 받지만 내부에서는 와일드카드를 실제 타입으로 바꿔주는 형태로 구현함으로써 `깔끔한 컴파일 + 추상화`를 이뤄내자는 말인 것 같다

- 와일드카드 타입 vs. 타입 매개변수

https://stackoverflow.com/questions/18142009/type-parameter-vs-unbounded-wildcard

> Conclusion: In all these examples it is mostly a matter of taste and style whether you prefer the generic or the wildcard version. There is usually trade-off between **ease of implementation (the generic version is often easier** to implement) and **complexity of signature (the wildcard version has fewer type parameters or none at all).**

- 간단한 메소드 시그니처 → 이해하기 쉽고 → public API 에서 좋다 (tradeoff : 복잡한 구현)
- 사용하는 스타일보다는 전체적인 API에서 일관성을 유지하는 게 좋다