

Item 31.

한정적 와일드카드를 사용해 API 유연성을 높이라





INDEX

목차

1. API 유연성 중요
2. 제네릭 타입 유연함의 한계
3. 한정적 와일드 카드
4. 한정적 와일드 카드 사용 주의
5. 한정적 와일드 카드 심화



01

API 유연성 중요



API 유연성

다양한 입력을 처리 가능하도록 설계
호출자의 사용 제약을 줄이고 재사용 가능성을 높이는 성질



```
1  class NumberProcessor {  
2      public void printNumbers(List<Number> numbers) {  
3          for (Number n : numbers) {  
4              System.out.println(n);  
5          }  
6      }  
7  }
```



```
1  public static void main(String[] args) {  
2      List<Integer> integers = List.of(1, 2, 3);  
3  
4      NumberProcessor processor = new NumberProcessor();  
5      processor.printNumbers(integers);  
6  }
```

```
public static void main(String[] args) {  
    List<Integer> integers = List.of(1, 2, 3);  
    NumberProcessor processor = new NumberProcessor();  
    processor.printNumbers(integers);  
}
```

Required type: List <Number>

Provided: List <Integer>

매개변수 타입 불공변

- Integer은 Number의 하위 타입 O
- List<Integer>는 List<Number>의 하위 타입 X



```
1  public void printNumbers(List<Number> numbers) {  
2      for (Number n : numbers) {  
3          System.out.println(n);  
4      }  
5  }  
6  
7  public void printIntegers(List<Integer> numbers) {  
8      for (Number n : numbers) {  
9          System.out.println(n);  
10     }  
11 }
```



```
1 public <T> void print(List<T> objects) {  
2     for (T o : objects) {  
3         System.out.println(o);  
4     }  
5 }
```



```
1 public static void main(String[] args) {  
2     List<Number> numbers = List.of(1, 2, 3);  
3     List<Double> doubles = List.of(1.1, 2.2, 3.3);  
4     List<Integer> integers = List.of(1, 2, 3);  
5  
6     NumberProcessor processor = new NumberProcessor();  
7     processor.print(numbers);  
8     processor.print(doubles);  
9     processor.print(integers);  
10 }
```

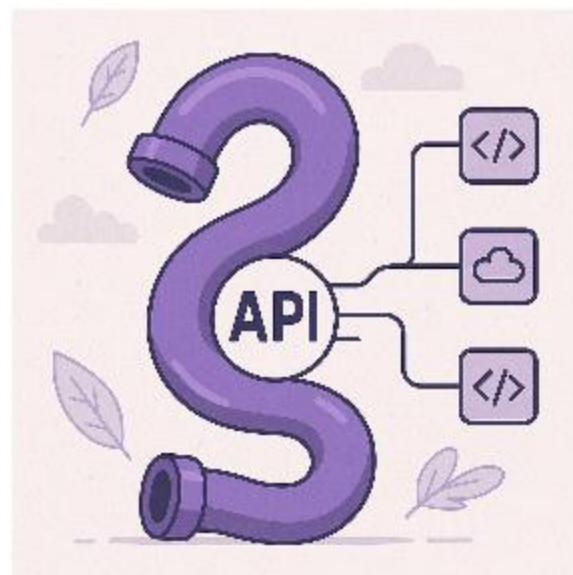
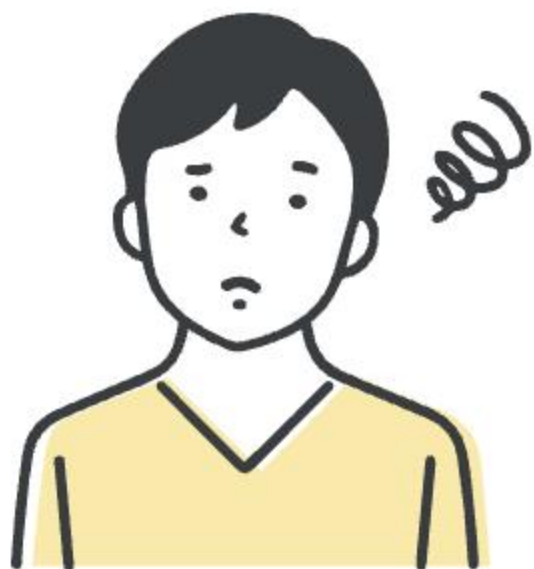


유연한 API 설계

- 모든 타입 유연한 처리 가능
- 중복 메서드 제거
- 코드 간결성 & 유지보수성
- 명확한 의도 표현

02

제네릭 타입 유연함의 한계



```
1 public class Stack<E> {
2
3     public void push(E item) { ...}
4
5     public E pop() {...}
6
7     public boolean isEmpty() { ... }
8
9     public void pushAll(Collection<E> src) {
10         for (E e : src) {
11             push(e);
12         }
13     }
14 }
```

```
1 public static void main(String[] args) {
2     List<Integer> integers = new ArrayList<>();
3
4     Stack<Number> stack = new Stack<>();
5     stack.pushAll(integers);
6 }
```

데이터 생산자

외부에서 전달받은

Collection<E>의 데이터를 읽어

Stack에 넣는 역할



```
1 public class Stack<E> {  
2  
3     public void push(E item) { ...}  
4  
5     public E pop() {...}  
6  
7     public boolean isEmpty() { ... }  
8  
9     public void popAll(Collection<E> dst) {  
10         while (!isEmpty()) {  
11             dst.add(pop());  
12         }  
13     }  
14 }
```



```
1 public static void main(String[] args) {  
2     List<Object> objects = new ArrayList<>();  
3  
4     Stack<Number> stack = new Stack<>();  
5     stack.popAll(objects);  
6 }
```



데이터 소비자

Stack 데이터를 꺼내 전달받은
Collection<E>에 추가함으로써
전달받은 컬렉션이 데이터를 소비

```
public static void main(String[] args) {  
    List<Integer> integers = new ArrayList<>();  
  
    Stack<Number> stack = new Stack<>();  
    stack.pushAll(integers);  
}
```

Required type: Collection <Number>

Provided: List <Integer>

```
public static void main(String[] args) {  
    List<Object> objects = new ArrayList<>();  
  
    Stack<Number> stack = new Stack<>();  
    stack.popAll(objects);  
}
```

Required type: Collection <Number>

Provided: List <Object>

제네릭 타입 불공변성

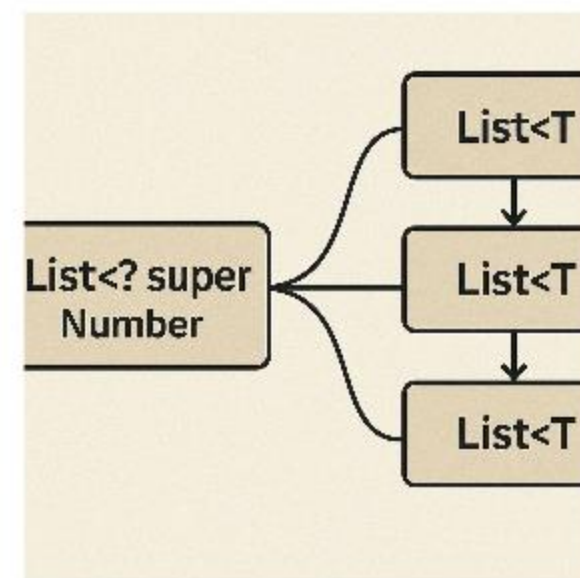
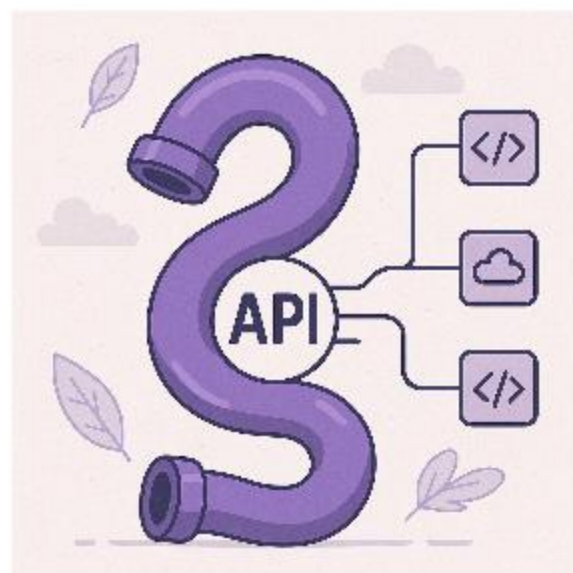
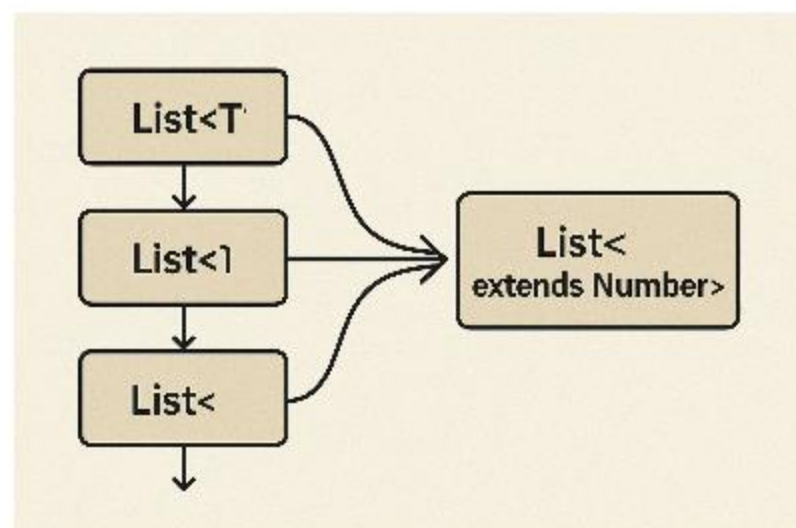
- 자바는 제네릭 타입의 자동 형변환을 허용 X



**다른 호환되는 타입도
push, pop 할 수 있을까?**

03

한정적 와일드 카드




한정적 와일드카드

제네릭의 유연성을 높이면서도 타입 안정성을 유지하기 위한 핵심 개념
자바 제네릭의 불공변성을 극복하고자 등장

상한 한정

(Upper Bounded Wildcard)




```
1 <? extends T>
```

T 이거나 T의 하위 타입만 가능

하한 한정

(Lower Bounded Wildcard)



```
1 <? super T>
```

T 이거나 T의 상위 타입만 가능



```
1 public void pushAll(Collection<E> src) {  
2     for (E e : src) {  
3         push(e);  
4     }  
5 }  
6  
7 public void popAll(Collection<E> dst) {  
8     while (!isEmpty()) {  
9         dst.add(pop());  
10    }  
11 }
```



```
1 public void pushAll(Collection<? extends E> src) {  
2     for (E e : src) {  
3         push(e);  
4     }  
5 }  
6  
7 public void popAll(Collection<? super E> dst) {  
8     while (!isEmpty()) {  
9         dst.add(pop());  
10    }  
11 }
```



**언제 extends, super
키워드 사용하지?**



```
1 public void pushAll(Collection<? extends E> src) {  
2     for (E e : src) {  
3         push(e);  
4     }  
5 }  
6  
7 public void popAll(Collection<? super E> dst) {  
8     while (!isEmpty()) {  
9         dst.add(pop());  
10    }  
11 }
```



데이터 생산자

외부에서 전달받은

Collection<E>의 데이터를 읽어

Stack에 넣는 역할



데이터 소비자

Stack 데이터를 꺼내 전달받은

Collection<E>에 추가함으로써

전달받은 컬렉션이 데이터를 소비

PECS 공식

- Producer **E**xtends, Consumer **S**uper
- 생산자(Producer) → ? **extends** T
- 소비자(Consumer) → ? **super** T



1 <? extends T>



1 <? super T>



```
1 public void pushAll(Collection<? extends E> src) {  
2     for (E e : src) {  
3         push(e);  
4     }  
5 }  
6  
7 public void popAll(Collection<? super E> dst) {  
8     while (!isEmpty()) {  
9         dst.add(pop());  
10    }  
11 }
```

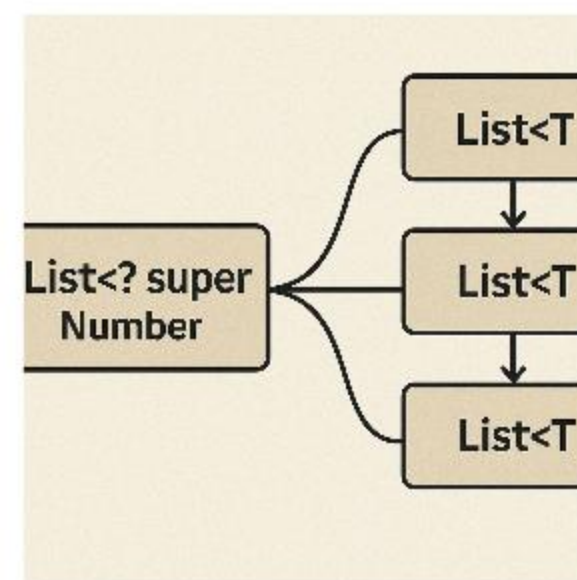
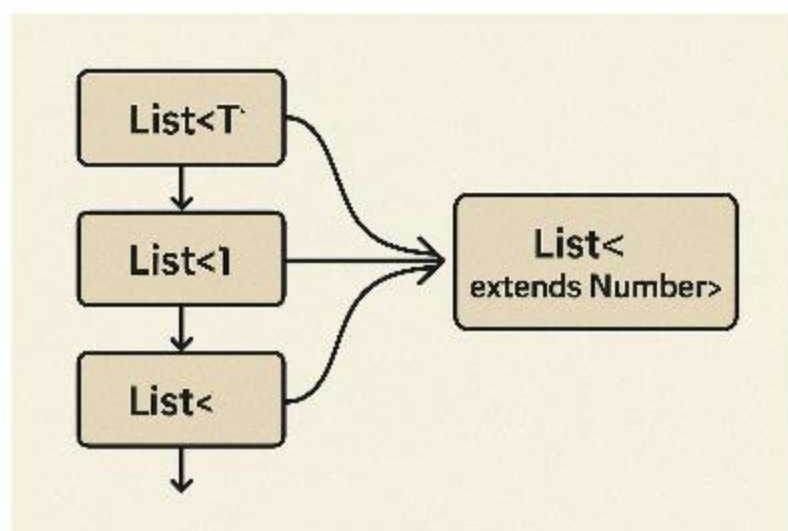


한정적 와일드카드 장점

1. 유연성 : 불공변 극복의 상하위 타입 지원
2. 안전성 : 타입 체크로 오류 방지
3. 재사용성 : 코드 중복 해소
4. 명확성 : 파라미타의 목적(읽기/쓰기) 표현

04

한정적 와일드 카드 사용 주의



한정적 와일드카드 사용 주의

1. 입력 매개변수가 생산자와 소비자 역할이 동시인 경우 금지
2. 반환 타입에는 한정적 와일드카드 타입 사용 금지


```
public <T> void swapEx1(List<? extends T> list, int i, int j) {  
    T temp = list.get(i);  
    list.set(i, list.get(j));  
    list.set(j, temp);  
}
```

```
public <T> void swapEx2(List<? super T> list, int i, int j) {  
    T temp = list.get(i);  
    list.set(i, list.get(j));  
    list.set(j, temp);  
}
```



```
1 public <T> void swapEx3(List<T> list, int i, int j) {  
2     T temp = list.get(i);  
3     list.set(i, list.get(j));  
4     list.set(j, temp);  
5 }
```

생산자와 소비자

- list.get(i) 생산자 역할(읽기)
- list.set(j, temp) 소비자 역할(쓰기)
- 타입의 안정성 불가능

즉, 생산자 + 소비자 역할을 동시에 하는 매개변수는 일반적인 제네릭 타입<T> 명시적 선언



```
1 public static List<? extends CharSequence> getTexts() {  
2     return List.of("hello", "world");  
3 }  
4  
5 public static void main(String[] args) {  
6     List<? extends CharSequence> texts = getTexts();  
7 }
```



반환 타입 사용 금지

- 호출자가 반환값을 자유롭게 사용 불가
- 타입 **안정성과 추론**의 문제 발생
- 호출자에게 변환의 **책임 전파**

05

한정적 와일드 카드 심화





```
1 public static <E extends Comparable<E>> E max(List<E> list)
```



```
1 public static <E extends Comparable<E>> E max(List<E> list) {  
2     E max = list.get(0);  
3     for (E item : list) {  
4         if (item.compareTo(max) > 0) {  
5             max = item;  
6         }  
7     }  
8     return max;  
9 }
```



```
1 public static <E extends Comparable<E>> E max(List<E> list)
```



```
1 public static <E extends Comparable<E>> E max(List<? extends E> list)
```



Step 1 매개변수 적용

list 변수는 생산자 역할

- 매개변수의 list는 값을 꺼내는 작업(read)
- add() 같은 쓰기 작업 X
- Producer → Extends 적용



```
1 public static <E extends Comparable<E>> E max(List<? extends E> list)
```



```
1 public static <E extends Comparable<? super E>> E max(List<? extends E> list)
```

Step 2 타입 매개변수 선언 개선

Comparable<E>은 소비자 역할

- 타입 설계 관점에서 비교 대상을 소비
- E를 받아서 의미 있는 판단을 내릴 수 있는 타입
즉, E를 소비할 수 있는 타입
- Consumer → Super 적용

언제나 소비자
SUPER

Comparable과 Comparator는 항상 소비자 역할



```
1 public static <T extends Comparable<? super T>> T max(List<? extends T> list)
```

Item 31.

**한정적 와일드카드를 사용해
API 유연성을 높이라**

