11. 제네릭(Generics)

제네릭 클래스, 메서드 정의

🟦 1. 제네릭(Generic) 이란?

> 클래스나 메서드에서 **데이터 타입을 매개변수처럼 일반화**해주는 기능

```
<T> : 타입 매개변수(Type Parameter)
```

❓ 왜 필요한가?

- 컴파일 시 **타입 안정성 보장**
- **캐스팅 제거**
- **재사용성 증가**

🟧 2. 제네릭 클래스 정의

```
class Box<T> {
    private T item;

    public void setItem(T item) {
        this.item = item;
    }

    public T getItem() {
        return item;
    }
}
```

🔧 사용 예

```
Box<String> stringBox = new Box<>();
stringBox.setItem("Hello");
System.out.println(stringBox.getItem()); // Hello

Box<Integer> intBox = new Box<>();
intBox.setItem(100);
System.out.println(intBox.getItem()); // 100
```

🎯 제네릭 클래스의 형식

```
class 클래스이름<타입매개변수> {
    ...
}
```

✅ 타입 매개변수 관례 이름

이름	의미
T	Type
E	Element (컬렉션에서 사용)
K	Key (맵에서 사용)
V	Value (맵에서 사용)
N	Number

🟩 3. 제네릭 메서드 정의

> 메서드에만 제네릭을 적용 (클래스 전체가 제네릭일 필요는 없음)

```
public class Util {
    public static <T> void printArray(T[] array) {
        for (T item : array) {
            System.out.print(item + " ");
        }
        System.out.println();
    }
}
```

🔧 사용 예

```
Integer[] intArr = {1, 2, 3};
String[] strArr = {"a", "b", "c"};

Util.printArray(intArr);  // 1 2 3
Util.printArray(strArr);  // a b c
```

🟥 4. 제네릭 클래스 vs 메서드 비교

항목	제네릭 클래스	제네릭 메서드
적용 범위	클래스 전체	해당 메서드 하나
선언 위치	`class Box<T>`	`<T> void method(T param)`
예시	`Box<T>`	`Util.<T>printArray()`
사용 시	`Box<String>`, `Box<Integer>`	`Util.printArray(...)`

🟪 5. 타입 추론

- 제네릭 메서드는 **타입을 생략**해도 컴파일러가 추론해줌

```
public static <T> T pick(T a, T b) {
    return a;
}

// 사용 시
String s = pick("A", "B");   // T = String
Integer i = pick(1, 2);      // T = Integer
```

🟫 6. 제네릭 생성자 정의

```
class Sample {
    public <T> Sample(T item) {
        System.out.println("Created with: " + item);
    }
}
```

```
Sample s1 = new Sample("hello");
Sample s2 = new Sample(123);
```

✅ 마무리 요약

기능	예시
제네릭 클래스	`class Box<T> { T item; ... }`
제네릭 메서드	`public static <T> void print(T t)`
타입 매개변수	`T`, `E`, `K`, `V` 등의 의미
장점	타입 안정성, 코드 재사용성, 캐스팅 제거

와일드카드 (`<?>`, `<? extends T>`, `<? super T>`)

🟦 1. 와일드카드란?

> `?` 는 **어떤 타입이든 올 수 있다**는 의미를 가진 *타입 매개 변수의 대체자*

기본 형태

구문	의미
`<?>`	모든 타입 허용 (Unbounded wildcard)

구문	의미
`<? extends T>`	T 또는 T의 하위 타입만 허용 (Upper Bounded)
`<? super T>`	T 또는 T의 상위 타입만 허용 (Lower Bounded)

🟧 2. `<?>` (Unbounded Wildcard)

```
public void printList(List<?> list) {
    for (Object obj : list) {
        System.out.println(obj);
    }
}
```

✅ 특징

- **읽기 전용**으로 사용해야 안전
- 어떤 타입의 List든 전달 가능: `List<Integer>`, `List<String>` 등

```
List<String> strList = List.of("a", "b");
printList(strList);  // OK
```

❌ 쓰기 제한

```
list.add("hello"); // 컴파일 에러!
```

왜? → `<?>`로는 구체적인 타입을 알 수 없기 때문.

🟩 3. `<? extends T>` (Upper Bounded Wildcard)

> T와 그 하위 클래스만 받을 수 있음

```
public void process(List<? extends Number> list) {
    Number n = list.get(0); // OK
}
```

✅ 특징

- **읽기는 가능**
- **쓰기 불가**

```
List<Integer> intList = List.of(1, 2, 3);
process(intList); // OK

list.add(3); // ❌ 컴파일 에러
```

→ 하위 타입이 너무 다양해서 안전하게 쓸 수 없음

🟥 4. `<? super T>` (Lower Bounded Wildcard)

> T와 그 **상위 클래스만** 받을 수 있음

```
public void insert(List<? super Integer> list) {
    list.add(10);  // OK
    // Integer는 항상 하위 타입이므로 안전
}
```

✅ 특징

- **쓰기 가능 (T 또는 하위 타입만)**
- **읽기 불가 (Object로만 읽힘)**

```
Object obj = list.get(0); // Only as Object
```

🟪 5. PECS 원칙

> **"Producer Extends, Consumer Super"**

역할	와일드카드	이유
Producer	`<? extends T>`	데이터를 꺼내기만 함 (read only)
Consumer	`<? super T>`	데이터를 넣기만 함 (write only)

```
void readItems(List<? extends Item> items);   // Produces Item
void addItems(List<? super Item> items);      // Consumes Item
```

🟫 6. 정리 비교

와일드카드	읽기 가능성	쓰기 가능성	예시 타입
`<?>`	✅ 가능 (Object로)	❌ 불가	`List<?>`
`<? extends T>`	✅ 가능 (T로)	❌ 불가	`List<? extends Number>`
`<? super T>`	✅ 가능 (Object로)	✅ 가능 (T 또는 하위)	`List<? super Integer>`

🧠 예제: 숫자 총합 계산기

```
public static double sum(List<? extends Number> list) {
    double sum = 0;
    for (Number n : list) {
        sum += n.doubleValue();
    }
    return sum;
}
```

```
List<Integer> intList = List.of(1, 2, 3);
List<Double> dblList = List.of(1.1, 2.2, 3.3);

System.out.println(sum(intList));  // 6.0
System.out.println(sum(dblList));  // 6.6
```

🔚 마무리 요약

와일드카드	읽기	쓰기	대표 상황
`<?>`	✅	❌	읽기만 필요할 때
`<? extends T>`	✅	❌	데이터를 "생산"만 할 때
`<? super T>`	❌	✅	데이터를 "소비"만 할 때

제네릭 제한 (`T extends`, `T super`)

🟦 1. 타입 제한(Type Bounds)이란?

> 제네릭 타입 매개변수에 **상위나 하위 타입 제한을 명시**하는 기능.

```
<T extends Number>
<T super Integer>
```

이걸 쓰면:

- `T`는 특정 타입의 자식이거나 (extends)
- `T`는 특정 타입의 부모이거나 (super)

🟧 2. `T extends ...` (상한 제한, Upper Bound)

```
public class Calculator<T extends Number> {
    public double add(T a, T b) {
        return a.doubleValue() + b.doubleValue();
    }
}
```

✅ 특징

- `T`는 `Number` 혹은 그 하위 타입만 허용
- `Integer`, `Double`, `Float` 등 OK
- `String`, `Object` 등은 ❌

```
Calculator<Integer> calc1 = new Calculator<>();
Calculator<Double> calc2 = new Calculator<>();
// Calculator<String> ❌ 컴파일 에러
```

✅ 제네릭 메서드도 사용 가능

```
public static <T extends Comparable<T>> T max(T a, T b) {
    return (a.compareTo(b) > 0) ? a : b;
}
```

🟥 3. `T super ...` (하한 제한, Lower Bound)

※ 주의: `super`는 클래스 정의에서는 안 되고, **와일드카드에서만 사용 가능**

✅ 예시: 소비자 메서드

```
public void addIntegers(List<? super Integer> list) {
    list.add(100);  // OK
}
```

- `list`는 `Integer` 이상의 타입이면 OK (즉, Integer, Number, Object 등)
- → `Integer` 객체를 안전하게 넣을 수 있음

🟩 4. 다중 상한 제한

```
<T extends Number & Comparable<T>>
```

- `T`는 `Number`를 상속하면서 `Comparable`도 구현해야 함
- 인터페이스는 여러 개 가능, 클래스는 하나만 가능

```
public <T extends Number & Comparable<T>> T min(T a, T b) {
    return a.compareTo(b) <= 0 ? a : b;
}
```

5. 타입 제한 없이 정의한 경우와 비교

선언	의미
`<T>`	아무 타입이나 가능
`<T extends Number>`	Number 및 하위 타입만 가능
`<? super Integer>`	Integer 및 상위 타입만 가능 (소비자 용)
`<T extends A & B & C>`	A를 상속하며 B, C 인터페이스 구현해야 함

실전 응용 예시: 필터 함수

```
public static <T extends Comparable<T>> List<T> filterGreaterThan(List<T> list, T threshold) {
    List<T> result = new ArrayList<>();
    for (T item : list) {
        if (item.compareTo(threshold) > 0) {
            result.add(item);
        }
    }
    return result;
}
```

```
List<Integer> nums = List.of(1, 5, 3, 7);
System.out.println(filterGreaterThan(nums, 4)); // [5, 7]
```

마무리 요약

구문	설명
`<T>`	아무 타입이나 허용
`<T extends Number>`	T는 Number나 그 하위 타입이어야 함
`<T extends A & B>`	A 상속 + B 인터페이스 구현
`<? super T>`	T나 그 상위 타입만 허용 (소비자용)

타입 소거(Type Erasure)

🧨 1. 타입 소거란?

> **제네릭 타입은 컴파일 타임에만 존재**하고,
> **런타임에는 모든 타입 정보가 제거(Erasured)** 되는 Java의 제네릭 처리 방식.

📌 즉, JVM은 제네릭을 *알지 못함*.
모든 제네릭은 컴파일 후 `Object` 또는 경계 타입으로 **대체됨**.

🛠️ 예시

```
List<String> list1 = new ArrayList<>();
List<Integer> list2 = new ArrayList<>();
```

> 컴파일 후에는 둘 다 `List`로 변함.
> 즉, **런타임에는 동일한 타입으로 간주됨**.

```
System.out.println(list1.getClass() == list2.getClass());  // true
```

🧪 2. 타입 소거가 일어나는 이유

- Java는 **하위 호환성**을 유지하기 위해
  → 기존 JVM에서도 동작하도록 설계됨 (1.5 이전 코드와 호환)
- 따라서 제네릭 정보는 **컴파일 타임에만 유효**하고,
  `.class` 파일에는 들어가지 않음

🧬 3. 타입 소거의 방식

1) 경계(bound)가 없는 경우

```
class Box<T> {
    T value;
}
```

> → 컴파일 후

```
class Box {
    Object value;
}
```

2) 경계(bound)가 있는 경우 (`<T extends Number>`)

```
class Calculator<T extends Number> {
    T value;
}
```

> → 컴파일 후

```
class Calculator {
    Number value;
}
```

🚫 4. 타입 소거로 인해 생기는 제약

❌ 1. 제네릭 배열 생성 불가

```
T[] arr = new T[10]; // 컴파일 에러
```

📌 이유: 런타임에 T가 뭔지 모르기 때문에 배열을 만들 수 없음

❌ 2. instanceof 사용 제한

```
if (obj instanceof List<String>) // 컴파일 에러
```

📌 이유: 런타임엔 `List<String>` 정보가 없기 때문
✔️ 대신엔 다음처럼 사용:

```
if (obj instanceof List<?>) { ... }
```

❌ 3. 타입 캐스팅 시 경고 발생

```
List<String> list = (List<String>) someRawList; // Unchecked warning
```

🔐 5. 타입 소거와 리플렉션

```
List<String> list = new ArrayList<>();
Field field = list.getClass().getDeclaredField("elementData");

System.out.println(field.getType());  // → Object[], ArrayList 내부 구조
```

📌 `list`의 타입 파라미터인 `<String>` 정보는 **리플렉션에서도 확인 불가**

단, `Method.getGenericReturnType()` 같은 API를 쓰면 **제한적으로** 접근 가능

🧱 6. 와일드카드와도 밀접한 관계

타입 소거는 다음 구조로 변환함:

제네릭 선언	컴파일 후 타입
`T`	`Object` 또는 Bound Type
`<? extends Number>`	`Number`
`<? super Integer>`	`Object`
`List<?>`	`List<Object>`

🧭 7. 대안은 없을까? → `Class<T>` 활용

```
public class TypeSafeBox<T> {
    private Class<T> type;

    public TypeSafeBox(Class<T> type) {
        this.type = type;
    }

    public T cast(Object obj) {
        return type.cast(obj);
    }
}
```

> `Class<T>`를 넘겨받아 *타입을 추적*하는 방식으로 보완 가능

🔚 정리 요약

항목	설명
정의	제네릭 타입 정보는 컴파일 후 제거됨
이유	JVM의 하위 호환성 유지
런타임 영향	`List<String>`과 `List<Integer>`는 런타임에 같은 타입
제약	제네릭 배열 생성 불가, instanceof 제한, 타입 캐스팅 경고 등
대응	`Class<T>`를 파라미터로 받아 추적, `@SuppressWarnings` 등 활용