아이템 44) 표준 함수형 인터페이스를 사용하라

- **필요한 용도에 맞는 게 있다면, 직접 구현하지 말고** 표준 함수형 인터페이스를 사용하라
  - 유용한 디폴트 메소드를 많이 제공하므로, 다른 코드와의 상호운용성도 크게 좋아진다
- 표준 함수형 인터페이스 대부분은 `기본 타입` 만 지원
  - **기본 함수형 인터페이스에 박싱된 기본타입을 넣어 사용하지는 말자!**
  - 계산량이 많을 때는 `박싱/ 언박싱으로 인한 성능 저하` 이슈가 있음
- java.util.function 패키지의 인터페이스 예시들

인터페이스	함수 시그니처	예	설명
UnaryOperator<T>	T apply(T t)	String::toLowerCase	반환값과 인수의 타입이 같은 함수, 인수 1개
BinaryOperator<T>	T apply(T t1, T t2)	BigInteger::add	반환값과 인수의 타입이 같은 함수, 인수 2개
Predicate<T>	boolean test(T t)	Collection::isEmpty	한 개의 인수를 받아서 boolean을 반환하는 함수
Function<T,R>	R apply(T t)	Instant::now	인수와 반환 타입이 다른 함수
Supplier<T>	T get()	Instant::now	인수를 받지 않고 값을 반환, 제공하는 함수
Consumer<T>	void accept(T t)	System.out::println	한 개의 인수를 받고 반환값이 없는 함수

  - 기본 인터페이스는 기본 타입인 int, long, double 용으로 3개씩 변형이 생겨남
    ex) IntPredicate, LongBinaryOperator

    → 기본 타입에 대한 특화된 인터페이스 사용시 박싱, 언박싱 오버헤드를 피할 수 있음

```
// 기본 함수형 인터페이스 사용 (int 를 Integer 로 박싱한 형태)
Predicate<Integer> predicate = i -> i > 0;

// 특화된 기본 함수형 인터페이스 사용
IntPredicate intPredicate = i -> i > 0;
```

  - `IntPredicate`를 사용하면 박된 형태로의 변환 없이 기본 타입을 직접 다룰 수 있음
    - **박싱 회피:** `IntPredicate` 등의 특화된 인터페이스를 사용하면 기본 타입에 대한 객체화를 피할 수 있음
    - **언박싱 회피:** 특화된 인터페이스를 사용하면 함수형 인터페이스를 통해 전달되는 값이 기본 타입이므로, 메소드 내부에서 언박싱 과정이 필요 없음

```
public interface Stream<T> extends BaseStream<T, Stream<T>> {
    Stream<T> filter(Predicate<? super T> predicate);
}
```

  - **Java 8 stream API 에서 사용되는 Predicate 예시**

```
// A.
Predicate<Employee> isMale = p -> p.getGender().equalsIgnoreCase("M");
List<Employee> maleEmployeeList = employeeList
                                    .stream()
                                    .filter(isMale)
                                    .toList();

// B.
List<Employee> maleEmployeeList = employeeList
                                    .stream()
                                    .filter(employee -> employee.getGender().equalsIgnoreCase("M")
                                    .toList();
```

1. **가독성과 재사용성**
   - A : `isMale`이라는 Predicate를 정의하여 재사용이 가능 → Predicate 함수형 인터페이스를 “일급 객체” 로 정의하고 사용할 수 있음

        - **변수에 할당할 수 있고,**
        - **인자로 전달할 수 있고,**
        - 반환값으로 사용될 수 있으며,
        - 자료구조에 저장할 수 있다
    - B : 람다 표현식을 직접 사용하므로 코드가 간단하지만, 재사용성이 떨어질 수 있습니다.
2. **코드 구조**
    - A : Predicate를 변수에 할당하여 코드의 구조를 명확하게 함
        - Predicate의 역할이 분명하게 드러나며, 필요한 경우 Predicate를 변경하기도 쉽습니다.
    - B : 한 줄에 람다 표현식을 사용하므로 코드가 짧아지지만, 이로 인해 코드의 의도를 파악하기가 어려울 수 있음

표준 함수형 인터페이스가 아니라 직접 작성하는 것이 좋은 경우

```
// Comparator
@FunctionInterface
public interface Comparator<T> {
    int compare(T o1, T o2);
}

// ToIntBiFunction
@FunctionalInterface
public interface ToIntBiFunction<T, U> {
    int applyAsInt(T t, U u);
}
```

- `Comparator<T>` 인터페이스는 구조적으로는 `ToIntBiFunction<T,U>`와 동일
- 심지어 자바 라이브러리에 `Comparator<T>`를 추가할 당시 `ToIntBiFunction<T,U>`가 이미 존재했더라도 `ToIntBiFunction<T,U>`를 사용하면 안 됐다.
- 전용 함수형 인터페이스를 구현해야 하는 지 고민해야 할 조건
    1. 자주 쓰이며, **이름 자체가 용도를 명확히 설명**해준다 / 비교
    2. 반드시 **따라야 하는 규약**이 있다 / compare 메소드 구현 (정렬규칙 정의)
    3. 유용한 디폴트 메소드를 제공할 수 있다 / **reversed()** → 역순 Comparator 반환, **thenComparing()** 메소드는 두 번째로 비교할 **Comparator**를 지정
- **직접 만든 함수형 인터페이스에는 항상 @FunctionalInterface 애너테이션을 사용하라.**

@FunctionalInterface의 세 가지 목적

- 해당 클래스의 코드나 설명 문서를 읽을 이에게 그 인터페이스가 람다용으로 설계된 것임을 알려준다.
- 해당 인터페이스가 추상 메서드를 오직 하나만 가지고 있어야 컴파일되게 해준다.
- 유지보수 과정에서 누군가 실수로 메서드를 추가하지 못하게 막아준다.

함수형 인터페이스 사용 시 주의점

- 서로 다른 함수형 인터페이스를 같은 위치의 인수로 받는 메서드들을 다중정의해서는 안 된다.
- 클라이언트에게 모호함을 주며 문제가 발생할 소지가 많다.

```
public interface ExecutorService extends Executor {
    // Callable<T>와 Runnable을 각각 인수로 하여 다중정의했다.
    // submit 메서드를 사용할 때마다 형변환이 필요해진다.
    <T> Future<T> submit(Callback<T> task);
    Future<?> submit(Runnable task);
}
```

결론

- 입력값과 반환값에 함수형 인터페이스 타입을 활용하라
- 표준 함수형 인터페이스를 사용하는 것이 대부분의 경우에 좋다