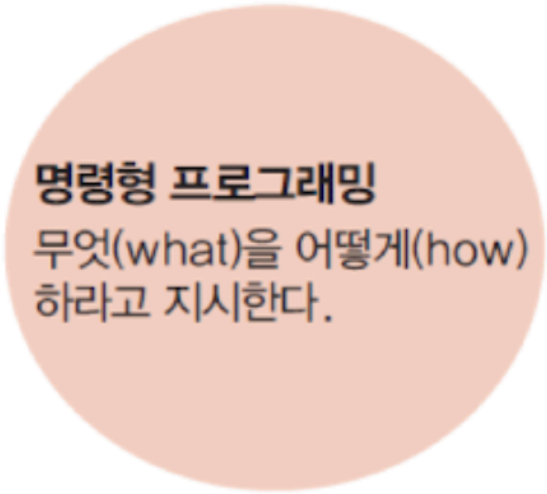
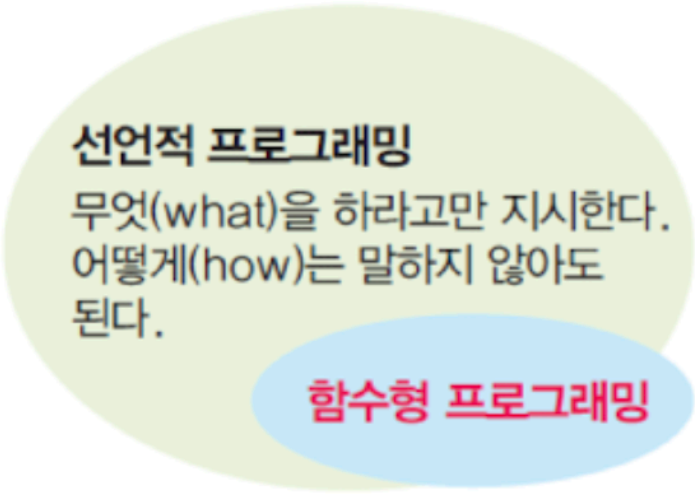
람다식과 함수형 인터페이스

함수형 프로그래밍의 소개

함수형 프로그래밍 지원은 java 8부터 시작 됨.
java 8부터 람다식, Stream API, 컬렉션 API향상, 디폴트 메소드와 정적 메소드, forEACH() apthem, Concurrency API향상, Java Time API 등이 지원 된다.

프로그래밍 패러다임 분류





명령형 VS 선언적 프로그래밍

선언적 프로그래밍은 해야 할 일에 초점을 맞춰 함수들이 계속 적용이 되면서 작업이 진행된다.
그리고 명령문이 아닌 수식이나 함수 호출로 이루어진다. (ex) 샌드위치 주세요.)
명령형 프로그래밍은 하나하나 다 명령 해줘야됨. (ex) 서브웨이에서 샌드위치 주문할 때)

람다식 기초

미국 수학자 알론조 처치가 함수를 분명하고 간결한 방법으로 설명하기 위해 고안.
나중에 실행할 목적으로 다른 곳에 전달할 수 있는 코드이다.
익명의 클래스를 단순화해 그 표현식을 메서드의 인수로 전달하거나 인터페이스의 객체를 생성할 수 있는 기능을 제공한다.
"동적 매개 변수화"를 사용해 코드를 전달하면 코드의 빈번한 요구 사항을 변경에 대처가 유용하다.

의미: 메소드를 포함하는 익명 구현 객체를 전달할 수 있는 코드

특징

- 메소드와 달리 이름이 없다.
- 메소드와 달리 특정 클래스에 종속되지 않지만, 매개변수, 반환타입, 본체를가지며, 예외도 처리할 수 있다.
- 메소드의 인수로 전달될 수 있고 변수에 대입될 수 있다.
- 익명 구현 객체와 달리 메서드의 핵심 부분만 포함한다.

문법

(선언부) -> {구현부}

```
(타입 매개변수) -> {실행문; 실행문; ...}
```

메소드 참조

전달할 동작을 수행하는 메소드가 이미 정의된 경우에 표현할 수 있는 람다식의 축약형 메소드 참조의 종류와 표현 방식
정적 메소드 참조 => 클래스이름 :: 정적메서드
인스턴스 메소드 => 객체이름 :: 인스턴스메소드 // 클래스이름 :: 인스턴스메소드 생성자 참조 => 클래스이름 :: new // 배열타입이름 :: new

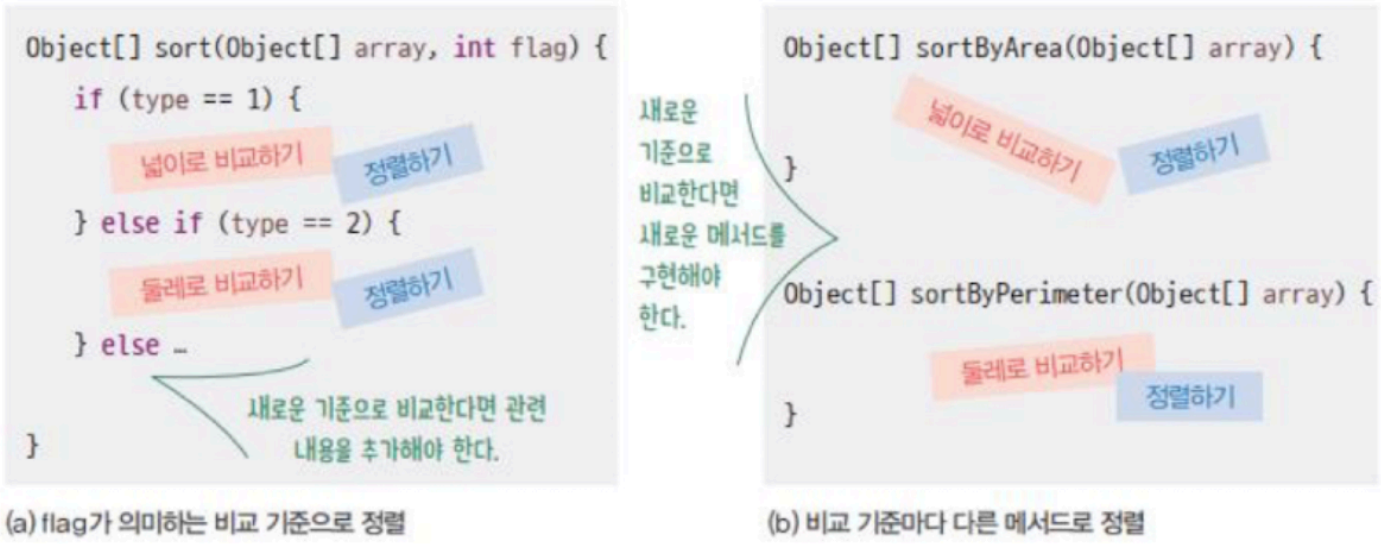
람다식의 필요성

하나의 Rectangle 클래스를 정의하면 사각형 객체끼리 비교할 수 없다.
문제점: 복잡하고 가독성이 떨어진다. Rectangle 클래스에 색상, 사각형 번호와 같은 다른 속성도 있다면 정렬 메서드를 수정하거나 새로운 메소드를 추가하는 것이 필요하다.

그러나 객체끼리 비교할 기준이 여러가지라면 비교 기준마다 Comparable 구현 클래스를 따로 정의해야된다.

비교 기준을 포함할 클래스가 최종 클래스라면 Comparable 구현 클래스를 정의할 수 없다.

정렬 메서드 구현



(a) flag가 의미하는 비교 기준으로 정렬

(b) 비교 기준마다 다른 메서드로 정렬

객체 비교 및 정렬

자바는 비교할 수 있는 객체를 생성하기 위해 Comparable 인터페이스를 제공한다.

```
public interface Comparable <T> {
    int compareTo(T o);
}
```

java.util 패키지의 Arrays 클래스는 sort()라는 정적 메서드를 제공한다.

```
static void Arrays.sort(Object[] a);
```

Object[] a -> 배열 월소가 Comparable 타입이어야한다.

동작 매개 변수화(behavior parameterization)

고객의 빈번한 요구 사항 변경을 처리할 수 있는 소프트웨어 개발 패턴.
이 방법에서는 사용자의 요구를 담은 코드 블록을 생성하고 이것을 프로그램의 다른 부분에 전달하는 것.

예시

영업사원: 자동차 재고를 저장하고 검색할 수 있는 것을 원함.
개발자: 이러한 변화하는 요구 사항에 부응하면서 최소한의 노력으로 구현 및 유지 관리가 간단한 방법을 사용해야됨.

1) 흰색자동차를찾는기능
2) 자동차가격이5000만원이하자동차를찾는기능
3) 색상이흰색으로5000만원이하인자동차를찾는기능

버전 1)
매개변수 없음: 매개변수가 없이 흰색 자동차만 추려서 리스트로 만들어 반환

버전2)
값 매개 변수화: 색상을 매개 변수화하고 메소드에 색상을 나타내는 매개 변수를 추가

버전3)
동작 매개 변수화 : 자동차의 속성을 검사해 true, false로 반환하는 함수를 작성해 메소드로 전달

버전4)
익명 클래스 사용:

```
List<Car> whiteCars = filterCars(carList, new CarPredicate() {
    public boolean test (Car car) {
        return "WHITE".equals(car.getColor());
    }
}
```

버전5)
람다식 사용:

```
List<Car> whiteCars = filterCars(carList, (Car car) -> "WHITE".equals(car.getColor()));
```

람다 표현식으로 전환

- 기존에 만들어 놓은 익명 클래스에 구현해 놓은 메서드를 변경하여 람다 만들기

```
List<Car> whiteCars = filterCars(inventory, new CarPredicate() {
    public boolean test(Car car){
        return "WHITE".equals(car.getColor());
    }
});
```

- 1단계 : 익명클래스 선언부분 제거

```
List<Car> whiteCars = filterCars(inventory,
    public boolean test(Car car){
        return "WHITE".equals(car.getColor());
    }
);
```

```
List<Car> whiteCars = filterCars(inventory,
    public boolean test(Car car){
        return "WHITE".equals(car.getColor());
    }
);
```

■ 2단계 : 메서드 선언 부분 제거

```
List<Car> whiteCars = filterCars(inventory,
    (Car car){
        return "WHITE".equals(car.getColor());
    }
);
```

■ 3단계 : 람다 문법으로 정리

```
List<Car> whiteCars = filterCars(inventory,
    (Car car) -> "WHITE".equals(car.getColor());
);
```

함수형 인터페이스

의미: 하나의 추상 메소드만을 가진 인터페이스
분류: 매개 값 -> 함수형 인터페이스 -> 반환 값

함수형 인터페이스 종류

종류	매개 값	반환 값	메서드	의미
Predicate	있음	boolean	test( )	매개 값을 조사하여 논릿값으로 보낸다.
Consumer	있음	void	accept( )	매개 값을 소비한다.
Supplier	없음	있음	get( )	반환 값을 공급한다.
Function	있음	있음	apply( )	매개 값을 반환 값으로 매핑한다.
Operator	있음	있음	apply( )	매개 값을 연산하여 반환 값으로 보낸다.

● Predicate (Bi, Double, Int, Long을 접두어로 붙인 변종이 있음.)

```
Predicate<T> p = t -> { T 타입 t 객체를 조사하여 논릿값으로 반환하는 실행문; };
```

● Consumer (Bi, Double, Int, Long, ObjDouble, ObjInt, ObLong를 접두어로 붙인 변종이 있음.)

```
Consumer<T> c = t -> { T 타입 t 객체를 사용한 후 void를 반환하는 실행문; };
```

● Supplier (Double, Int 등을 접두어로 붙인 변종이 있음.)

```
Supplier<T> s = () -> { T 타입 t 객체를 사용한 후 void를 반환하는 실행문; };
```

● Function (Bi, Double, IntToDouble, ToDoubleBi 등을 접두어로 붙인 변종이 있음.)

```
Function<T, R> f = t -> { T 타입 t 객체를 사용한 후 void를 반환하는 실행문; };
```

● Operator (Operator라는 인터페이스는 없고 Binary, Unary, Double, Int, Long을 접두어로 붙인 변종만 있다.)

```
BinaryOperator<T> o = (x, y) -> { T 타입 t 객체를 사용한 후 void를 반환하는 실행문; };
```

● Comparator 인터페이스
객체에 순서를 정하기 위하여 사용되는 함수형 인터페이스
compare()라는 추상 메소드 외에도 유용한 정적 메서드와 디폴트 메서드를 제공하며, 메서드의 반환 타입은 모두 `Comparator<T>` 타입

정적 메서드	의미
comparing( )	Comparable 타입의 정렬 키로 비교하는 Comparator를 반환한다.
naturalOrder( )	Comparable 객체에 자연 순서로 비교하는 Comparator를 반환한다.
nullsFirst( )	null을 객체보다 작은 값으로 취급하는 Comparator를 반환한다.
nullsLast( )	null을 객체보다 큰 값으로 취급하는 Comparator를 반환한다.
reverseOrder( )	자연 반대 순서로 비교하는 Comparator를 반환한다.

디폴트 메서드	의미
reversed( )	현재 Comparator의 역순으로 비교하는 Comparator를 반환한다.
thenComparing( )	다중 키를 사용하여 정렬하려고 새로운 Comparator를 반환한다.

함수형 인터페이스와 람다식

람다식과 함수형 인터페이스는 분리할 수 없는 관계이다.
컴파일러는 람다식을 올바르게 컴파일하려면 함수형 인터페이스가 정의되어야 한다.

=> 람다식을 사용하려면 누군가가 먼저 람다식을 위한 함수형 인터페이스를 정의해야된다.