Java Wildcards

1. ? extends

"? extends E"

- M이 E의 subtype이면 Pair<E>와 Pair<M>은 Pair<? extends E>의 subtype이다.

  Pair<? extends E> p = new Pair<E>
  Pair<? extends E> p = new Pair<M>

- 변수 p는 Pair<E의 subtype>을 저장할 수 있다.
- p변수가 가르키는 Pair의 원소값을 변경시키려고 하면 오류가 발생한다.

  왜냐하면, p변수가 가르키는 Pair의 원소의 타입을 컴파일타임에 결정할 수 없다. 위에서 p변수가 가르키는 Pair는 항상 저장하려고 하는 객체의 타입보다 subtype을 원소로 가지는 Pair를 가르킬 수 있다.

  - 가령,
    p.setFirst(e) // e는 E타입
    p가 Pair<M>이라면 오류 (p는 Pair<? extends E>의 subtype은 모두 가질 수 있으니까)

    p.setFirst(m) // m은 M타입
    p가 Pair<S> (S를 M의 subtype이라고 가정, 뒤에라도 S를 추가 가능함)라면 오류

  즉, 어떤 E의 subtype 객체를 set하려고 해도 p는 그보다 더 subtype 객체의 Pair를 가르킬 수 있다.

  - 강의노트에서의 해석방식 (실제로는 이렇게 컴파일 되지 않음, 개념적인 해석)

    "? extends E" getFirst() - getFisrt가 "? extends E"타입의 객체를 반환. 따라서 E타입으로 결과를 저장하면 됨
    void setFirst("? extends E") - setFirst가 "? extends E"를 요구함. "? extends E"는 E의 subtype 중 어떤 것일 수 있음(미지의 타입. X라고 해도 좋고 Java 컴파일러에서 표현하는 CAP#1과 같은 타입이라해도 좋음). 하지만 컴파일 타임에는 모른다는 것을 의미함. 따라서 setFirst를 적용할 대상보다 더 supertype 객체가 전달될 가능성이 항상 있음.

  - Java compiler에서는 이런 wildcard type을 CAP#n과 같은 방식으로 표현(wildcard capture)하고 특정 type이 CAP#n으로 변환될 수 없다는 규칙을 만듬

    p.setFirst(e); // E cannot be converted to CAP#1

- upper bounded wildcards(? extends)는 값을 간단하게 꺼집어 내어 사용하는 경우에 활용함. 데이터를 꺼집어내어 저정하는 변수는 bound type(가령, E)으로 선언함.

```
void printBuddies(Pair<? extends E> p)
{
    E first = pair.getFrist();
    …
}
```

2. ? extends E vs <T extends E>

- "? extends E"는 type parameter <T extend E>로 교체 가능함 (역은 성립 안함)
- "? extends E"가 더 유용한 경우가 있음.

1) Case 1

```
public static <T extends E> double totalEarnings(List<T> emps)
public static double totalEarnings(List<? extends E> emps)
```

- 각 원소를 읽어내어 총 급여만 계산한다면 두 method 사이에는 차이가 없다.
- 둘 다 사용가능하나 후자가 더 적합함 (왜냐하면 type parameter를 굳이 사용하지 않아도 됨)
- 단순히 bound type(여기서는 E)의 관점에서 값을 읽어내어 polymorphic하게 처리한다면 wildcard가 적합함

2) Case 2

```
public static <T extends E> T min(List<T>)
public static E min(List<? extends E>)
```

- 전자는 compiler가 T에 instantiation되는 타입(가령, M)에 따라 정확하게 type checking을 할 수 있기 때문에 type safe한 코드를 만들기 쉬움
- 후자는 E type으로만 처리가능한 코드 작성가능
- 둘 다 사용가능하지만 전자가 가능한 작업이 더 많음

전자)

```
M  m = min(new List<M>);
m.mMethod();  // mMethod는 M에 존재하는 method
```

후자)

```
M m = min(new List<M>); // 오류: min의 return type은 E

E e = min(mew List<M>);
e.mMethod();  // 오류
```

```
E e = min(mew List<M>);
e.eMethod();    // OK
```

- type parameter T가 다른 곳에서 reference된다면(위에서는 return type) bounded type parameter를 사용함

3) Case 3

```
public static <T extends E> void replace(List<T> emps, T e)
public static void replace(List<? extends E> emps, E e)
```

- List에서 하나의 Employee를 두 번째 매개변수의 e로 교체하고자 한다면 후자는 불가능함.
  ? extends Employee와 Employee가 일반적으로 타입이 일치하지 않음
- 즉, type parameter가 매개변수에 두 군데 이상 사용된다면 wildcard의 사용은 불가능함

3. ? super M

- M이 E의 subtype이면 Pair<? super M>은 Pair<M>, Pair<E>, Pair<Object> 의 super type이다.
- 따라서 Pair<? super M> 변수에 setFisrt(m)을 한다면 항상 가능함(m은 M타입)

- 사용예1: 강의노트 minMaxBonus

```
public static void minMaxBonus(M[] a, Pair<? super M> result)
{
    M min;
    M max;
      ....
    result.setFirst(min);    // result는 Pair<M> Pair<E> Pair<Object> 등이 될 수 있음
    result.setSecond(max);
}
```

사용예 1은 wildcard type을 통해 값을 반환하거나 매개변수를 통해 바깥(out)으로 전달하는 경우임.

- 사용예2: Collections.sort

```
static <T> void sort(List<T> list, Comparator<? super T> c)
```

T 타입의 객체를 정렬할 때 비교자로는 T의 supertype을 비교하는 어떤 비교자도 가능하다는 뜻. 가령, List<M>를 정렬하는데 M의 bonus를 가지고 비교하는 비교자로 좋고, E의 이름으로 비교하는 비교자(가령, Comparator<E>)도 좋다. 심지어 Comparator<Object>도 가능하다. Comparator<M>, Comparator<E>, Comparator<Object>를 모두 포함하는 타입이 바로 Comparator<? super M>이다.