Generic Programming - Part 3

Chapter 8, Core Java Volume I

Contents

- Why Generic Programming?
- Generic Classes
- Type Variables
- Advantages of Generics
- Generic Interfaces and Classes in Java Collection Framework
- Using a Generic Class
- Defining Generic Methods
- Bounds for Type Variables
- Type Erasure
- Cast Insertion
- Inheritance Rules for Generic Types
- Wildcard Types (Upper Bounded Wildcards)
- Supertype Bounds (Lower Bounded Wildcards)
- Unbounded Wildcards

Wildcard Types (Upper Bounded Wildcards)

- Write a method that print all the subclasses of Employee

```
public static void printBuddies(Pair<Employee> pair) { … }
```

⇒ cannot hold Pair<Manager> as the parameter. Why?

- Wildcard types allow type variance:

```
Pair<? extends Employee> pair;
```

  - Can hold a Pair<Employee> or Pair<Manager>.

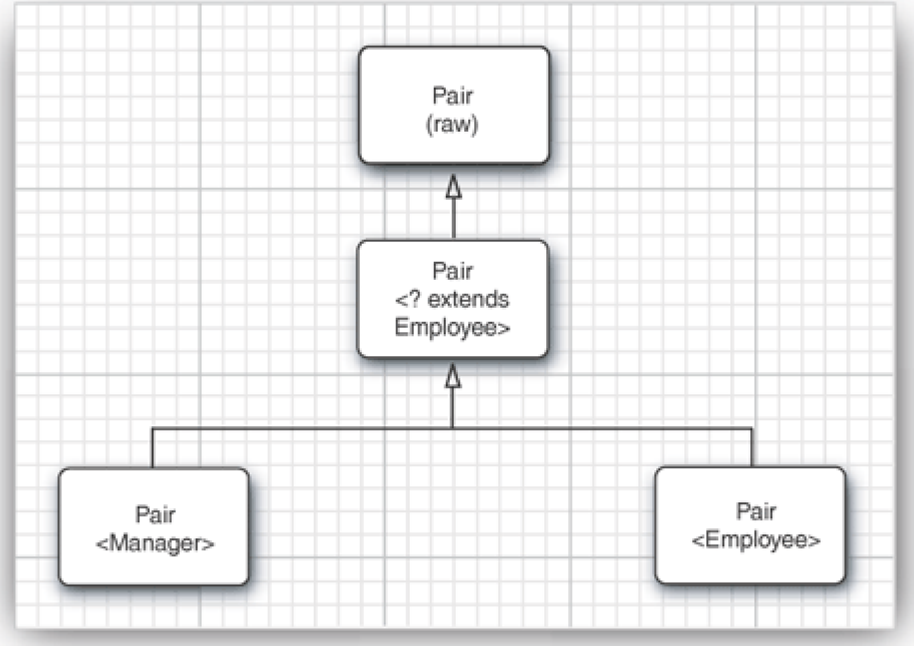
```
public static void printBuddies(Pair<? extends Employee> pair)
{
   Employee first = pair.getFirst();
   Employee second = pair.getSecond();
   System.out.println(first.getName() + " and " + second.getName() + " are buddies.");
}
```

```
Pair<Employee> pe = new Pair<>();
Pair<Manager> pm = new Pair<>();
…
printBuddies(pe);
printBuddies(pm);
…
```

Wildcard Types (Upper Bounded Wildcards)

- Can we use wildcards to corrupt a Pair<Manager> through a Pair<? extends Employee> reference?

```
Pair<Manager> managerBuddies   = new Pair<>(ceo, cfo);
Pair<? extends Employee> wildcardBuddies = managerBuddies;
    // OK
wildcardBuddies.setFirst(lowlyEmployee);
    // compile-time error
```



- The Pair<? extends Employee> methods look like this:

```
? extends Employee getFirst()
void setFirst(? extends Employee)
```

  - it is impossible to call the setFirst method

Supertype Bounds (Lower Bounded Wildcards)

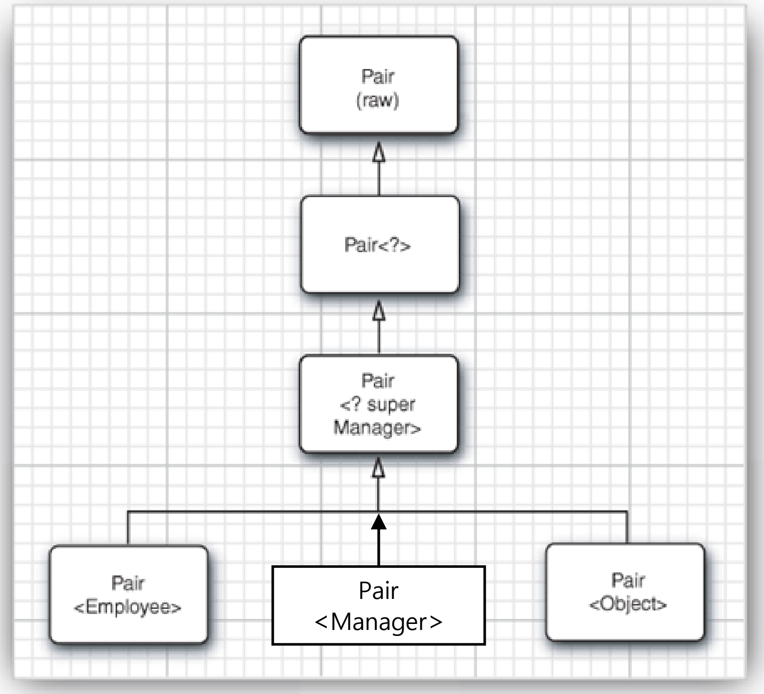
- Supertype bound:

  ```
  ? super Manager
  ```

  - The wildcard is restricted to all supertypes of Manager

- Example

```
public static void minmaxBonus
     (Manager[] a, Pair<? super Manager> result)
{
  if(a.length==0) return;
  Manager min = a[0]; Manager max = a[0];
  for(int i=0; i<a.length; i++) {
     if(min.getBonus() > a[i].getBonus()) min = a[i];
     if(max.getBonus() > a[i].getBonus()) max = a[i];
  }
  result.setFirst(min);
  result.setSecond(max);
}
```



Supertype Bounds (Lower Bounded Wildcards)

- The opposite behavior to the upper bounded wildcards:
  - You can supply parameters to the methods.
  - But you cannot use the return value.

- Pair<? super Manager> has methods that look like:

```
? super Manaer getFirst()
void setFirst(? super Manager)
```

- Intuitively speaking, wildcards with supertype bounds let you write to a generic object, while wildcards with subtype bounds let you read from a generic object.

Supertype Bounds (Lower Bounded Wildcards)

- Another use of supertype bounds: Comparable interface

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

- minmax method in the modified ArrayAlg class

```
public static <T extends Comparable<T>> Pair<T> minmax(T[] a)  {…}
```

  - This method works for String class because String is a subtype of Comparable<String>
- How about LocalDate? It does not work!
  - LocalDate implements ChronoLocalDate interface and ChronoLocalDate extends Comparable<ChronoLocalDate>, thus LocalDate is a subtype Comparable<ChronoLocalDate>, but not a subtype of Comparable<LocalDate>
- supertype bound solves the problem:

```
public static <T extends Comparable<? super T>> Pair<T> minmax(T[] a)  {…}
```

Supertype Bounds (Lower Bounded Wildcards)

- Java Collection Framework API extensively use wildcards
- Example: Collections.sort

```
static <T extends Comparable<? super T>> void sort(List<T> list)

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

Unbounded Wildcards

- Can you do anything with the `Pair<?>` type?
- It looks identical to the raw `Pair` type; But actually, they are very different
- The `Pair<?>` has the methods such as:

```
? getFirst()
void setFirst(?)
```

  - The return value of `getFirst` can only be assigned to an `Object`.
  - The `setFirst` method can never be called.
- Is it useful?
  - When the code is using methods in the generic class that don't depend on the type parameter.

```
public static void printList(List<?> list) {
    for (Object elem: list)
        System.out.print(elem + " ");
    System.out.println();
}
```

```
public static boolean hasNulls(Pair<?> p) {
   return p.getFirst() == null || p.getSecond() == null;
}
```