Item 31: Use bounded wildcards to increase API flexibility

As noted in Item 28, parameterized types are *invariant*. In other words, for any two distinct types `Type1` and `Type2`, `List<Type1>` is neither a subtype nor a supertype of `List<Type2>`. Although it is counterintuitive that `List<String>` is not a subtype of `List<Object>`, it really does make sense. You can put any object into a `List<Object>`, but you can put only strings into a `List<String>`. Since a `List<String>` can't do everything a `List<Object>` can, it isn't a subtype (by the Liskov substitution principal, Item 10).

Sometimes you need more flexibility than invariant typing can provide. Consider the `Stack` class from Item 29. To refresh your memory, here is its public API:

```
public class Stack<E> {
    public Stack();
    public void push(E e);
    public E pop();
    public boolean isEmpty();
}
```

Suppose we want to add a method that takes a sequence of elements and pushes them all onto the stack. Here's a first attempt:

```
// pushAll method without wildcard type - deficient!
public void pushAll(Iterable<E> src) {
    for (E e : src)
        push(e);
}
```

This method compiles cleanly, but it isn't entirely satisfactory. If the element type of the `Iterable src` exactly matches that of the stack, it works fine. But suppose you have a `Stack<Number>` and you invoke `push(intVal)`, where `intVal` is of type `Integer`. This works because `Integer` is a subtype of `Number`. So logically, it seems that this should work, too:

```
Stack<Number> numberStack = new Stack<>();
Iterable<Integer> integers = ... ;
numberStack.pushAll(integers);
```

If you try it, however, you'll get this error message because parameterized types are invariant:

```
StackTest.java:7: error: incompatible types: Iterable<Integer>
cannot be converted to Iterable<Number>
        numberStack.pushAll(integers);
                            ^
```

Luckily, there's a way out. The language provides a special kind of parameterized type call a *bounded wildcard type* to deal with situations like this. The type of the input parameter to `pushAll` should not be "`Iterable` of `E`" but "`Iterable` of some subtype of `E`," and there is a wildcard type that means precisely that: `Iterable<? extends E>`. (The use of the keyword `extends` is slightly misleading: recall from Item 29 that *subtype* is defined so that every type is a subtype of itself, even though it does not extend itself.) Let's modify `pushAll` to use this type:

```
// Wildcard type for a parameter that serves as an E producer
public void pushAll(Iterable<? extends E> src) {
    for (E e : src)
        push(e);
}
```

With this change, not only does `Stack` compile cleanly, but so does the client code that wouldn't compile with the original `pushAll` declaration. Because `Stack` and its client compile cleanly, you know that everything is typesafe.

Now suppose you want to write a `popAll` method to go with `pushAll`. The `popAll` method pops each element off the stack and adds the elements to the given collection. Here's how a first attempt at writing the `popAll` method might look:

```
// popAll method without wildcard type - deficient!
public void popAll(Collection<E> dst) {
    while (!isEmpty())
        dst.add(pop());
}
```

Again, this compiles cleanly and works fine if the element type of the destination collection exactly matches that of the stack. But again, it isn't entirely satisfactory. Suppose you have a `Stack<Number>` and variable of type `Object`. If you pop an element from the stack and store it in the variable, it compiles and runs without error. So shouldn't you be able to do this, too?

```
Stack<Number> numberStack = new Stack<Number>();
Collection<Object> objects = ... ;
numberStack.popAll(objects);
```

If you try to compile this client code against the version of `popAll` shown earlier, you'll get an error very similar to the one that we got with our first version of `pushAll`: `Collection<Object>` is not a subtype of `Collection<Number>`. Once again, wildcard types provide a way out. The type of the input parameter to popAll should not be "collection of E" but "collection of some supertype of E" (where supertype is defined such that E is a supertype of itself [JLS, 4.10]). Again, there is a wildcard type that means precisely that: Collection<? super E>. Let's modify popAll to use it:

```
// Wildcard type for parameter that serves as an E consumer
public void popAll(Collection<? super E> dst) {
    while (!isEmpty())
        dst.add(pop());
}
```

With this change, both Stack and the client code compile cleanly.

The lesson is clear. **For maximum flexibility, use wildcard types on input parameters that represent producers or consumers.** If an input parameter is both a producer and a consumer, then wildcard types will do you no good: you need an exact type match, which is what you get without any wildcards.

Here is a mnemonic to help you remember which wildcard type to use:

PECS stands for producer-extends, consumer-super.

In other words, if a parameterized type represents a T producer, use <? extends T>; if it represents a T consumer, use <? super T>. In our Stack example, pushAll's src parameter produces E instances for use by the Stack, so the appropriate type for src is Iterable<? extends E>; popAll's dst parameter consumes E instances from the Stack, so the appropriate type for dst is Collection<? super E>. The PECS mnemonic captures the fundamental principle that guides the use of wildcard types. Naftalin and Wadler call it the *Get and Put Principle* [Naftalin07, 2.4].

With this mnemonic in mind, let's take a look at some method and constructor declarations from previous items in this chapter. The Chooser constructor in Item 28 has this declaration:

```
public Chooser(Collection<T> choices)
```

This constructor uses the collection choices only to **produce** values of type T (and stores them for later use), so its declaration should use a wildcard type that **extends** T. Here's the resulting constructor declaration:

```
// Wildcard type for parameter that serves as an T producer
public Chooser(Collection<? extends T> choices)
```

And would this change make any difference in practice? Yes, it would. Suppose you have a List<Integer>, and you want to pass it in to the constructor for a `Chooser<Number>`. This would not compile with the original declaration, but it does once you add the bounded wildcard type to the declaration.

Now let's look at the `union` method from Item 30. Here is the declaration:

```
public static <E> Set<E> union(Set<E> s1, Set<E> s2)
```

Both parameters, `s1` and `s2`, are `E` producers, so the PECS mnemonic tells us that the declaration should be as follows:

```
public static <E> Set<E> union(Set<? extends E> s1,
                               Set<? extends E> s2)
```

Note that the return type is still `Set<E>`. **Do not use bounded wildcard types as return types.** Rather than providing additional flexibility for your users, it would force them to use wildcard types in client code. With the revised declaration, this code will compile cleanly:

```
    Set<Integer> integers = Set.of(1, 3, 5);
    Set<Double>  doubles  = Set.of(2.0, 4.0, 6.0);
    Set<Number>  numbers  = union(integers, doubles);
```

Properly used, wildcard types are nearly invisible to the users of a class. They cause methods to accept the parameters they should accept and reject those they should reject. **If the user of a class has to think about wildcard types, there is probably something wrong with its API.**

Prior to Java 8, the type inference rules were not clever enough to handle the previous code fragment, which requires the compiler to use the contextually specified return type (or *target type*) to infer the type of `E`. The target type of the `union` invocation shown earlier is `Set<Number>`. If you try to compile the fragment in an earlier version of Java (with an appropriate replacement for the `Set.of` factory), you'll get a long, convoluted error message like this:

```
Union.java:14: error: incompatible types
        Set<Number> numbers = union(integers, doubles);
                                   ^
  required: Set<Number>
  found:    Set<INT#1>
  where INT#1,INT#2 are intersection types:
    INT#1 extends Number,Comparable<? extends INT#2>
    INT#2 extends Number,Comparable<?>
```

Luckily there is a way to deal with this sort of error. If the compiler doesn't infer the correct type, you can always tell it what type to use with an *explicit type* *argument* [JLS, 15.12]. Even prior to the introduction of target typing in Java 8, this isn't something that you had to do often, which is good because explicit type arguments aren't very pretty. With the addition of an explicit type argument, as shown here, the code fragment compiles cleanly in versions prior to Java 8:

```
// Explicit type parameter - required prior to Java 8
Set<Number> numbers = Union.<Number>union(integers, doubles);
```

Next let's turn our attention to the max method in Item 30. Here is the original declaration:

```
public static <T extends Comparable<T>> T max(List<T> list)
```

Here is a revised declaration that uses wildcard types:

```
public static <T extends Comparable<? super T>> T max(
        List<? extends T> list)
```

To get the revised declaration from the original, we applied the PECS heuristic twice. The straightforward application is to the parameter `list`. It produces `T` instances, so we change the type from `List<T>` to `List<? extends T>`. The tricky application is to the type parameter `T`. This is the first time we've seen a wildcard applied to a type parameter. Originally, `T` was specified to extend `Comparable<T>`, but a comparable of `T` consumes `T` instances (and produces integers indicating order relations). Therefore, the parameterized type `Comparable<T>` is replaced by the bounded wildcard type `Comparable<? super T>`. Comparables are always consumers, so you should generally **use `Comparable<? super T>` in preference to `Comparable<T>`.** The same is true of comparators; therefore, you should generally **use `Comparator<? super T>` in preference to `Comparator<T>`.**

The revised `max` declaration is probably the most complex method declaration in this book. Does the added complexity really buy you anything? Again, it does. Here is a simple example of a list that would be excluded by the original declaration but is permitted by the revised one:

```
List<ScheduledFuture<?>> scheduledFutures = ... ;
```

The reason that you can't apply the original method declaration to this list is that `ScheduledFuture` does not implement `Comparable<ScheduledFuture>`. Instead, it is a subinterface of `Delayed`, which extends `Comparable<Delayed>`. In other words, a `ScheduledFuture` instance isn't merely comparable to other

ScheduledFuture instances; it is comparable to any Delayed instance, and that's enough to cause the original declaration to reject it. More generally, the wildcard is required to support types that do not implement Comparable (or Comparator) directly but extend a type that does.

There is one more wildcard-related topic that bears discussing. There is a duality between type parameters and wildcards, and many methods can be declared using one or the other. For example, here are two possible declarations for a static method to swap two indexed items in a list. The first uses an unbounded type parameter (Item 30) and the second an unbounded wildcard:

```
// Two possible declarations for the swap method
public static <E> void swap(List<E> list, int i, int j);
public static void swap(List<?> list, int i, int j);
```

Which of these two declarations is preferable, and why? In a public API, the second is better because it's simpler. You pass in a list—any list—and the method swaps the indexed elements. There is no type parameter to worry about. As a rule, **if a type parameter appears only once in a method declaration, replace it with a wildcard.** If it's an unbounded type parameter, replace it with an unbounded wildcard; if it's a bounded type parameter, replace it with a bounded wildcard.

There's one problem with the second declaration for swap. The straightforward implementation won't compile:

```
public static void swap(List<?> list, int i, int j) {
    list.set(i, list.set(j, list.get(i)));
}
```

Trying to compile it produces this less-than-helpful error message:

```
Swap.java:5: error: incompatible types: Object cannot be
converted to CAP#1
        list.set(i, list.set(j, list.get(i)));
                                        ^
  where CAP#1 is a fresh type-variable:
    CAP#1 extends Object from capture of ?
```

It doesn't seem right that we can't put an element back into the list that we just took it out of. The problem is that the type of list is List<?>, and you can't put any value except null into a List<?>. Fortunately, there is a way to implement this method without resorting to an unsafe cast or a raw type. The idea is to write a private helper method to *capture* the wildcard type. The helper method must be a generic method in order to capture the type. Here's how it looks:

```
public static void swap(List<?> list, int i, int j) {
    swapHelper(list, i, j);
}

// Private helper method for wildcard capture
private static <E> void swapHelper(List<E> list, int i, int j) {
    list.set(i, list.set(j, list.get(i)));
}
```

The `swapHelper` method knows that `list` is a `List<E>`. Therefore, it knows that any value it gets out of this list is of type `E` and that it's safe to put any value of type `E` into the list. This slightly convoluted implementation of `swap` compiles cleanly. It allows us to export the nice wildcard-based declaration, while taking advantage of the more complex generic method internally. Clients of the `swap` method don't have to confront the more complex `swapHelper` declaration, but they do benefit from it. It is worth noting that the helper method has precisely the signature that we dismissed as too complex for the public method.

In summary, using wildcard types in your APIs, while tricky, makes the APIs far more flexible. If you write a library that will be widely used, the proper use of wildcard types should be considered mandatory. Remember the basic rule: producer-`extends`, consumer-`super` (PECS). Also remember that all comparables and comparators are consumers.