Bounded wildcards made simple

The topic of bounded wildcards is often cloaked in mystery. We attempt to introduce the topic in a way that uncloaks the mystery, making the topic as simple as possible. We will present two methods to help explain bounded wildcards. You will understand this topic more easily if you copy-and-paste the methods into a Java program and play with them as we suggest, so that you see for yourself what happens.

We will be using classes `Integer` and `Double`, which are both subclasses of type `Number`.

To the right is method `m1`. The type of parameter `p` contains an upper-bounded wildcard:

```java
public static void m1(ArrayList<? extends Number> p) {
    Number x = p.get(5);
    // p.add(new Integer(5));
}
```

This means that the corresponding argument of a call on `m1` can be an `ArrayList` whose elements are of class `Number` or of any class that extends class `Number`, including `Integer` and `Double`.

Within the body of `m1`, when an element of `ArrayList p` is retrieved, it is viewed as a `Number`.

Copy method `m1` into a Java class and note that it compiles. Now, uncomment the call `p.add(…). This call is syntactically incorrect; it does not compile; it violates the typing rules. This is good! Method `m1` should not be allowed to change `p`. For example, if the argument of a call is of type `ArrayList<Double>`, adding an `Integer` to `p` would be a mistake.

**SUMMARY.** Use an upper-bounded wild card only when values are to be retrieved and processed and the data structure (`ArrayList`) won't be changed.

To the right is method `m2`. The type of parameter `p` contains a lower-bounded wildcard:

```java
public static void m2(ArrayList<? super Integer> p) {
    p.add(5);
    // Integer x = p.get(5);
}
```

This means that the corresponding argument of a call on `m2` can be an `ArrayList` whose elements are of any class that is `Integer` or a superclass of `Integer`, including `Number` and `Object`. But an element of the `ArrayList` is viewed as an `Integer`.

Within the body of `m2`, it’s OK to add `Integer`s to `ArrayList p` because the possible types of the `ArrayList` elements are `Integer` and any superclass of `Integer`. For example, an `Integer` is a `Number` and also an `Object`.

Copy method `m2` into a Java class and note that it compiles. Now uncomment the assignment to `x`. This assignment is syntactically incorrect; it does not compile; it violates the typing rules. This is good! Method `m2` should not be allowed to retrieve values from the `ArrayList` because it views them all as of type `Integer` and they need not be `Integers`. For example, if the argument of a call has type `ArrayList<Number>`, an element might be a `Double`, which can’t be stored in an `Integer` variable.

**SUMMARY.** Use a lower-bounded wild card only when the data structure, in this case the `ArrayList`, is to be changed, but not to process its elements.

**Notes**

1. In the upper-bounded wildcard of the form `? extends C`, `C` can be a class or an interface. In this context, the word `extends` is used for both classes and interfaces.

2. In the lower-bounded wildcard of the form `? super C`, `C` can be a class or an interface. In this context, the word `super` is used for both classes and interfaces.

3. You may hear the terminology `in parameter` for a parameter that provides data to be processed. An upper-bounded wildcard can be used for this parameter. On the other hand, the terminology `out parameter` means a parameter that accepts data. A lower-bounded wildcard can be used for this parameter.

4. Don’t use wildcards in return types because that forces a user who is writing calls on the method to deal with them.