Quiz yourself: Using subclasses and covariant return types

When overloading a Java method, be careful to specify the appropriate return types.

by Mikalai Zaikin and Simon Roberts

February 23, 2021

If you have worked on our quiz questions in the past, you know none of them is easy. They model the difficult questions from certification examinations. We write questions for the certification exams, and we intend that the same rules apply: Take words at their face value and trust that the questions are not intended to deceive you but to straightforwardly test your knowledge of the ins and outs of the language.

Given the following two classes and the task of creating a subclass of `BirdsFarm`:

```java
class Bird {}

class BirdsFarm {
    public List<Bird> getAllBirds() { /* ... */ }
}
```

Which of the following methods will be allowed in the new class? Choose one.

A. `public List<? extends Bird> getAllBirds() { /* ... */ }`

The answer is A.
B. 
public List<? super Bird> getAllBirds() { /* ...
... */ } 

The answer is B.

C. 
public ArrayList<Bird> getAllBirds() { /* ...
... */ } 

The answer is C.

D. 
public ArrayList<Object> getAllBirds() { /* ...
... */ } 

The answer is D.

**Answer.** This question proposes four methods with the goal of adding one of them to the subclass. All four methods have the same name and the same (empty) argument type sequence as the parent class method, but they differ from each other and the parent method in their return types.

An overloading method is not permitted to differ solely by return type, so if one of these methods is acceptable, it must correctly override the parent method. Therefore, the question must hinge on the rules constraining return types in overriding methods.

Since Java 5, the language has permitted a *covariant return type* for overriding methods with reference type returns. This means that an overriding method can be declared to return a subtype of the return type of the parent class. This makes sense since the subtype “is a” instance of the parent anyway; the rule simply allows the overriding method to be more specific about what it returns, which might be helpful information to a client. Here’s a simple illustration:

```java
class Parent {}
class Child extends Parent {}
class A {
    public Parent getSomething() {...}
}
class B extends class A {
    @Override
    public Child getSomething() {...} // covari
}
```

To answer this question, you need to determine which of the four proposed return types is a subtype of (or, if you prefer, is assignment-compatible with, which amounts to the same thing) `List<Bird>`.

Option A proposes a return type of `List<? extends Bird>`. This syntax means a “list of anything that is assignment-
compatible with Bird. Of course, a List<Bird> "is a" List of something (Bird) that's assignment-compatible with Bird. This means that List<Bird> is a subtype of List<? extends Bird>. That in turn means List<? extends Bird> is a supertype for List<Bird>. This is the exact opposite of what's needed for the covariant return, which means option A is incorrect.

Let's look at another perspective of this problem. List<? extends Bird> means that at this point in the code you don't know exactly what the List is intended to hold, but whatever that intended type is, it must be assignment-compatible to Bird. Potentially the list might actually be a List<Chicken>, a List<Duck>, or simply a List<Bird>, but you don't know for sure. Now, since you don't know exactly what this List is intended to contain, you cannot add anything to this type of List. Why not? Well, if it's actually a List<Chicken>, you mustn't add ducks or sparrows to it. If it's a List<Duck>, you mustn't add chickens or penguins to it.

Option B proposes a return type of List<? super Bird>. This syntax means that whatever type of contents the List was created to accept (which, again, is unknown at this point in the code), Bird is assignment-compatible to that type. Again, a List<Bird> satisfies that requirement, so List<Bird> must be the subtype and List<? super Bird> must be the supertype. This makes option B incorrect.

In the declaration List<? super Bird>, you see a lower-bounded wildcard. This means that at this point in the code, you can safely add Bird (or any subclass of Bird) to the List, but if you perform a get operation, you can't know what type you might retrieve. After all, List<Object> is a List of something that will accept a Bird, but such a List might already contain String, LocalDate, or anything else that's not of primitive type.

It's worth thinking about why you cannot make the following assignment:

ArrayList<Parent> pl = new ArrayList<Child>();

If this assignment were permitted, you could also do that assignment in two steps, potentially leading to this code:

```java
ArrayList<Child> cl = new ArrayList<Child>();
ArrayList<Parent> pl = cl; // mimics the above
// now, pl accepts parent, so this must be an
pl.add(parentInstance);
// but now you can get Parent unexpectedly from
Child c = cl.get(0); // blows up: expects Child
```

On the other hand, this assignment is valid:
**Mikalai Zaikin**

Mikalai Zaikin is a lead Java developer at IBA IT Park in Minsk, Belarus. During his career, he has helped Oracle with development of Java certification exams, and he has been a technical reviewer of several Java certification books, including three editions of the famous *Sun Certified Programmer for Java* study guides by Kathy Sierra and Bert Bates.

**Simon Roberts**

Simon Roberts joined Sun Microsystems in time to teach Sun's first Java classes in the UK. He created the Sun Certified Java Programmer and Sun Certified Java Developer exams. He wrote several Java certification guides and is currently a freelance educator who publishes recorded and live video training through Pearson InformIT (available direct and through the O'Reilly Safari Books Online service). He remains involved with Oracle’s Java certification projects.

---

List<Parent> pl = new ArrayList<Parent>(); // OK

If you have this situation, `pl` is a List that's constrained to contain only Parent items. That's also all true of the ArrayList that's created and, therefore, ArrayList<Parent> is assignment-compatible with, and a subtype of, List<Parent>. In view of this, option C is correct.

For option D to be true, ArrayList<Object> would need to be a subclass of List<Bird>. However, a List<Bird> must not contain anything that's not assignment-compatible with Bird and, of course, a List<Object> can contain any reference type. This means that List<Object> cannot be a subtype of List<Bird>, and compilation must fail. From this you can see that option D is incorrect.

**Conclusion:** The correct answer is option C.