Introducing Variance into the Java Programming Language

A Quick Tutorial

DRAFT

Christian Plesner Hansen
Peter von der Ahé
Erik Ernst
Mads Torgersen
Gilad Bracha

June 3, 2003

1 Introduction

Notice: This tutorial is a work in progress. Stay tuned for further refinements. This tutorial describes an experimental extension of the Java programming language, known as *variance*, that adds to the generics features proposed in JSR-014 . At the time of this writing, the features described here are under consideration by the JSR-014 expert group.

The paper introduces the concepts of variant generic classes and statically safe arrays, in terms of read-only and write-only classes. It briefly explains generic classes in general, but otherwise assumes familiarity with GJ.

In the first part, read-only generic classes are introduced. Then, before the other kinds of variance on classes are described, read-only, invariant and write-only arrays are described. Then write-only and bivariant generic classes are introduced.

2 An Example

The running example in this introduction is a set of graphic classes.

Consider a simple drawing application that can draw simple shapes such as rectangles and circles. To represent these shapes within the program, you could define a class hierarchy such as the one in figure 1.

These classes can be drawn on a canvas:

```
public abstract class Shape {
    public abstract void drawOn(Canvas c);
}

public class Circle extends Shape {
    private int x, y, radius;
    public void drawOn(Canvas c) { ... }
}

public class Line extends Shape {
    private int x1, y1, x2, y2;
    public void drawOn(Canvas c) { ... }
}
```

Figure 1: Simple graphics classes

```
public class Canvas {
    ...
    public void draw(Shape s) {
        s.drawOn(this);
    }
    ...
}
```

Figure 2: The draw method in Canvas

3 Generic Classes

3.1 Read-only generic classes

A drawing will typically contain a number of shapes. If we represent those as a list, it would be convenient to have a utility method in Canvas that draws them all:

```
public void drawShapes(List⟨Shape⟩ shapes) {
    for (Shape s: shapes) {
        s.drawOn(this);
    }
}
```

Figure 3: First attempt to define drawAll.

At first glance, this method may seem reasonable enough: it takes a list of shapes and the draws them. There is, however, a problem. Suppose that we had a third kind of graphical element apart from Circle and Line, as in figure 4.

It could be reasonable to expect that a method in Polygon would be able to call drawShapes with its list of lines, since a list of lines could certainly also be

```
public class Polygon extends Shape {
    private List⟨Line⟩ lines;
    public void drawOn(Canvas c) { ... }
}
```

Figure 4: The Polygon class

considered as a list of shapes. That is, however, not the case.

One property of ordinary generic classes such as List⟨Shape⟩ is that it is *invariant*. This means exactly that you cannot, for instance, call a method expecting an argument of type List⟨Shape⟩ with something of type List⟨Line⟩, even though Line is a subclass of Shape. It can *only* be called with lists of Shapes. This is actually a very reasonable choice, as the example in figure 5 demonstrates.

```
List⟨Line⟩ lines = new LinkedList⟨Line⟩();
List⟨Shape⟩ shapes = lines; // Error: assigning List⟨Line⟩ to List⟨Shape⟩
shapes.put(0, new Circle(0, 0, 1));
```

Figure 5: The reason for invariance

If this program was allowed, the result would be that a circle would be stored in a list of lines, causing an error at runtime. The basic conflict here is this: if we want to allow a List⟨Line⟩ to be assigned to a List⟨Shape⟩, we cannot also allow shapes to be written into it – if we do, we get the problem from figure 5. Since it makes a big difference whether or not we intend to write into a class, it would definitely be convenient to be able to express this in the language.

Returning to the drawShapes example from figure 3, it is clear that this method really does not write into the list it receives as an argument – so in that case there would not be a problem in letting it accept a List⟨Line⟩ so that the Polygon could call it. This can be expressed by letting the argument be *read-only* list of shapes, List⟨+Shape⟩, rather than invariant. This method is shown in figure 6.

```
public void drawShapes(List⟨+Shape⟩ shapes) {
    for (Shape s: shapes) {
        s.drawOn(this);
    }
}
```

Figure 6: Correct signature for drawAll.

The syntax List⟨+Shape⟩ suggests that the argument does not have to be a list of exactly Shape but can also be something “more”, for instance List⟨Line⟩ or List⟨Circle⟩. The “price” for this flexibility is that it is now illegal to write into the list in the body of the method . For instance, if we tried to rewrite the problematic code from figure 5 we would still get an error:

```
List⟨Line⟩ lines = new LinkedList⟨Line⟩();
List⟨+Shape⟩ shapes = lines; // This is now allowed
shapes.put(0, new Circle(0, 0, 1)); // Error: shapes is read-only
```

Figure 7: This code is still illegal

Another example that demonstrates the usefulness of read-only classes is factory objects. A factory object is one that encapsulates the creation of objects. Encapsulating object creation can be very convenient, as the next example shows. The interface in figure 8 specifies the signature of a factory.

```
public interface Factory⟨T⟩ {
   T create();
}
```

Figure 8: Factory interface

Calling the create() method on a Factory⟨T⟩ should give an instance of the class T, much like calling **new** T(). For our graphics application, we would write factories for Circles and Lines as shown in figure 9.

```
class LineFactory implements Factory⟨Line⟩ {
   public Line create() { ... }
}
class CircleFactory implements Factory⟨Circle⟩ {
   public Circle create() { ... }
}
```

Figure 9: Line and Circle factories

Most graphics programs lets you select a "current shape" that is the one that is currently being drawn. For instance, that could be the shape to be drawn when the user clicks on the painting canvas. This could be implemented using a factory as described below and shown in figure 10.

When the user selects the current shape, the setFactory is called with a factory object that can create the selected type of shape. If we choose the current shape to be Line, for instance, we would call it with a LineFactory that is a Factory⟨Line⟩. This is legal exactly because this method accepts a Factory⟨+Shape⟩ rather than a Factory⟨Shape⟩. When the user wants to draw the shape, the newShape() method is called. We know that the factory variable contains a factory that can create instances of the currently selected shape so we can just "read out" a new instance of that shape by calling factory.create(). This is very convenient because that means that the newShape() method does not need to be concerned with which shape is currently selected, but can just expect that to be set up in advance. One consequence of this is that, since this part of the program is now independent of the available shapes, we can add new shapes and factories without ever having to change the body of the Controller.

```
public class Controller {
    ...
    /* Factory of the current shape */
    private Factory⟨+Shape⟩ factory;

    /* Selects the current shape */
    void setFactory(Factory⟨+Shape⟩ factory) {
        this.factory = factory;
    }

    /* Makes a new object the selected shape */
    void newShape() {
        Shape shape = factory.create();
        ... /* Add shape to the drawing */ ...
    }
    ...
}
```

Figure 10: Class using a factory

This example is another situation where we 1) want to be able to use not just Factory⟨Shape⟩ but a Factory of any kind of shape and 2) are not interested in writing into the class but only read out of it. This is exactly the situation where read-only classes are useful. To sum up read-only classes:

Read-only classes
A parameterized class with read-only arguments can reference classes parameterized with subclasses of the read-only arguments (see figure 6). .

4 Statically Checked Arrays

4.1 Read-only Arrays

In the previous section, we saw that allowing a List⟨Circle⟩ in the place of a List⟨Shape⟩ can get you into trouble if you intend to write into it. We also saw that in each of the cases where we tried to do that, in figure 5 and figure 7, the program was not allowed to compile, but generated an error. In the world of arrays, however, we are actually allowed to do this:

This program does exactly the same thing as the ones in figure 5 and figure 7. The major difference here is that the program in figure 11 is actually a legal program in the Java programming language and will compile without errors or warnings, whereas the others cause errors when compiled. Because of this, the programs involving generic classes are said to be *statically safe*. Actually, for programs involving generic classes, the language gives you an ironclad guarantee:

```
Line[] lines = new Line[1];
Shape[] shapes = lines;
shapes[0] = new Circle(0, 0, 1); // Compiles without errors, but throws
                                  // an exception at runtime
```

Figure 11: Legal but dangerous code!

if writing to a generic class (here the List) can cause errors at runtime, the program will generate errors when compiled . Such a guarantee simply does not hold for ordinary arrays, as the example in figure 11 shows.

In order to be able to write safer code with arrays, *statically safe arrays* have been introduced. These work much the same way as generic classes, and can be used much like ordinary arrays. However, the language gives the same guarantee as with generic classes: if a program that does not use ordinary arrays (*statically unchecked arrays*) can be compiled, it cannot cause runtime errors related to writing into arrays.

As an example, suppose that you wanted to define a drawShapes method like the one in figure 6, only taking an array of Shape rather than a List. Ordinarily, this could be defined as

```
public void drawShapes(Shape[] shapes) { ... }
```

Figure 12: drawShapes with ordinary arrays

Writing it this way is not recommended since the use of ordinary, statically unchecked, arrays involves the risk of runtime errors. Instead, we would really like to express making the same "deal" with the type system for Shape[] as with the List⟨+Shape⟩: the type system should allow you to call the method with arrays of subclasses, such as Line array, but disallow writing into the array. Previously, we used the read-only List⟨+Shape⟩ to express this. With statically safe arrays, we can express that in a similar way using a *read-only array*, written Shape[+]:

```
public void drawShapes(Shape[+] shapes) { ... }
```

Figure 13: drawShapes with statically safe arrays

The rules for read-only arrays are completely similar to the rules for read-only classes: you are allowed to call the method with arrays of any subclass of Shape, but trying to write into the array will cause an error at compile-time. This means that trying to rewrite the dangerous program from figure 11 will cause a compile-time error, just as we would expect:

Read-only arrays
A read-only array of type *Type*, written *Type*[+], can contain arrays of any subtype of *Type*. However, it is illegal to write into a read-only array.

```
Line[] lines = new Line[1];
Shape[+] shapes = lines;
shapes[0] = new Circle(0, 0, 1); // Error: shapes is read-only
```

Figure 14: Dangerous code that is now illegal.

4.2 Invariant Arrays

Of course, static safety is not of much use if it only deals with reading elements out of arrays, not writing elements in. It should be possible to do anything you would usually do with arrays and still have the static safety.

One thing we would of course want to do is both write into and read out of an array. The construct that allows this is is the *invariant array*, which corresponds to invariant generic classes described above. An invariant array of type *Type*, written *Type*[=], allows you to both read and write, but restricts the types that can be assigned to it the, same way as invariant classes. The intuition behind the syntax Shape[=] is that this is an array of *exactly* Shape, no more or less – so you cannot, for instance, call a method expecting a Shape[=] with something of type Line[=]. Examples of this is given in figure 15.

```
Shape[=] shapes = new Shape[1]; // Declaring an invariant array
shapes[0] = new Circle(0, 0, 1); // OK: writing allowed
Shape shape = shapes[0]; // OK: reading allowed
Circle[=] circles = shapes; // Error: Circle[=] and Shape[=] are incompatible
shapes = circles; // Error: Shape[=] and Circle[=] are incompatible
```

Figure 15: Invariant array examples

One of the consequences of the introduction of invariant arrays is that the usual signature for the main method is no longer the one in figure 16 but rather the one in figure 17.

```
public static void main(String[] args) { ... }
```

Figure 16: Old signature of main method

```
public static void main(String[=] args) { ... }
```

Figure 17: New signature for main method

4.3 Write-only Arrays

The third (and last) syntactically safe array construct is the *write-only array*, written *Type*[–]. In some cases you are only interested in writing into an array, not reading out of it. An example is a method initializing the elements of an array with a given element:

```
public void fillArray(Shape[–] array, Shape shape) {
    for (int i = 0; i < array.length; i++) {
        array[i] = shape;
    }
}
```

Figure 18: Method for filling array

This method can be called with any array you can put shapes into, which in this case means Shape[=] and Object[=]. Calling the method with a Shape array is legal, since we can write shapes into a shape array. Calling it with an Object array is legal since an array that can hold any kind of object can definitely also hold Shape objects. Calling it with a Circle, however, array will not work because we are allowed to write any kind of shapes into the array, including Circles for instance, and allowing that involves the risk of writing a Circle into an array of Lines.

Similar to the intuition behind Shape[+] meaning "an array of Shape or a subclass of Shape", the intuition behind Shape[–] is "an array of Shape or a *super*class of Shape". The code in figure 19 shows some of the things you can and cannot do with write-only arrays in relation to the fillArray method from figure 18.

```
Circle rectangle = new Circle();
Shape[=] shapes = new Shape[10];
fillArray(shapes, circle); // Allowed
Object[=] objects = new Object[10];
fillArray(objects, circle); // Allowed
Line[=] lines = new Line[10];
fillArray(lines, circle); // Error
```

Figure 19: Write-only arrays

Another example, given in figure 20, that illustrates both read-only and write-only arrays, is copying elements from one array to another of the same length.

```
public void copy(Shape[+] from, Shape[–] to) {
    assert from.length == to.length;
    for (int i = 0; i < from.length; i++) {
        Shape shape = from[i]; // OK: reading from a read-only array
        to[i] = shape; // OK: writing to a write-only array
    }
}
```

Figure 20: Copying from one array to another

This method is very flexible in the range of arguments it accepts: it can copy from any array that contains Shapes into any array that can hold Shapes, for instance from a Circle[=] to an Object[=].

Actually, saying that a Shape[–] is write-only is not completely true. We can actually read from it, we just do not get Shapes but Objects. That is safe to allow because even though we may not know exactly what kind of object we get when we read from the array, it will definitely be an Object:

```
public void readFromWriteOnly(Shape[–] shapes) {
    Object object = shapes[0]; // OK to read an Object
    Shape shape = shapes[0]; // Error
}
```

5 Write-only Classes

As you would probably expect, there is a concept of a write-only generic class corresponding to write-only arrays. For instance, you could write a method similar to copy from figure 20, using write-only lists:

```
void copy(List⟨+Shape⟩ from, List⟨–Shape⟩ to) { ... }
```

Write-only classes work almost exactly the same way as write-only arrays: You are allowed to use a list of Shape or a superclass of Shape in place of a List⟨–Shape⟩ and are allowed to write Shapes into it but only to read out Objects, not Shapes.

Another example where write-only classes are convenient are Comparators. A Comparator is an object that can compare two objects; the signature is given in figure 21.

```
public interface Comparator⟨T⟩ {
    int compare(T a, T b);
}
```

Figure 21: The Comparator interface.

A comparator can take two objects, a and b of type T, as arguments and compare them. If $a < b$, compare() returns -1, if $a > b$ it returns 1 and if $a = b$ it returns 0. An example could be a StringComparator, which is a Comparator⟨String⟩, that compares two strings so that $a < b$ if the string a came before b in lexicographical order.

```
Comparator⟨String⟩ strcomp = new StringComparator();
strcomp.compare("abbrev", "zorkmid"); /* –1 */
strcomp.compare("kremvax", "kremvax"); /* 0 */
```

Figure 22: Comparator example: StringComparator

Suppose that we wanted to define a utility method for finding the largest element in a collection. This method actually already exists as max(Collection, Comparator) in java.util.Collections. If we wanted to define it as a polymorphic method, it would look something like the method in figure 23.

```
static ⟨T⟩ T max(Collection⟨T⟩ coll, Comparator⟨T⟩ comp) {
  ... /* Find the max element */ ...
}
```

Figure 23: First attempt on the max method

This definition of max would allow it to be called with Collection⟨String⟩ and the StringComparator and the result would be the lexicographically first string in the collection. It would not, however, be legal to call it with a Collection⟨String⟩ and a Comparator⟨Object⟩. That means that we have a comparator that can compare *any* two objects, and hence also strings, we will not be allowed to call the max method as defined in figure 23 because it expects a Comparator⟨String⟩ when we call it with a Collection⟨String⟩. The solution is to declare the comparator as write-only as in figure 24:

```
static ⟨T⟩ T max(Collection⟨T⟩ coll, Comparator⟨–T⟩ comp) {
  ... /* Find the max element */ ...
}
```

Figure 24: Correct signature for max method

6 Bivariant Classes

Now we've seen three forms of generic classes: invariant, read-only and write-only. The last form is the *bivariant* generic class, written List⟨*⟩. A bivariant class can be considered a combination of read-only and write-only in that you can neither write not read anything but objects from the class. On the other hand, a bivariant list can contain *any* kind of list, as the code in figure 25 shows.

```
Vector⟨*⟩ someVector;
someVector = new Vector⟨Object⟩(); // Allowed
someVector = new Vector⟨String⟩(); // Allowed
someVector = new Vector⟨Circle⟩(); // Allowed
```

Figure 25: Bivariant assignments

The * in the syntax Vector⟨*⟩ signifies that this can be a list of anything at all. Even though it the operations that can be done on a bivariant class are very restricted, it is still possible to call methods that neither read nor write, such as size() on List. Also, since it is possible to read Objects from a bivariant list as it is with a write-only list, a method that only wants to read Objects from a bivariant class is still allowed to do that. For instance, a utility method that prints out all the elements in a list could be declared to take a bivariant argument:

```
public void printAll(List⟨*⟩ list) {
    for (Object obj : list) {
        System.out.println(obj);
    }
}
```

Another example could be a method that takes a list of lists and returns the total sum of their lengths:

```
public int totalSize(List⟨+List⟨*⟩⟩ lists) {
    int size = 0;
    for (List⟨*⟩ list : lists) {
        size += list.size();
    }
    return size;
}
```

This method takes a list that you can read List⟨*⟩s from. It uses the size() method to get their size, calculates the total sum, and returns it. The signature of this method allows it to be called with any list of lists, for instance a List⟨List⟨String⟩⟩ or a List⟨LinkedList⟨Shape⟩⟩.