Variance in the Java Programming Language

A Whitepaper

Sun Microsystems, Aarhus University and the Alexandra Institute

May 22, 2003

1 Introduction

This paper describes a proposed extension of The Java Programming Language, adding *variance* annotations to generic types and array types. The reader is assumed to be familiar with the JSR-14 proposal, and we refer to the language proposed therein as Generic Java.

There are a number of reasons for adding variance annotations to the Java programming language:

Increased abstraction: Generic classes suffer from a certain rigidity, precluding code that mixes generic instances with closely related argument types. Arrays circumvent this problem, but at the cost of static type safety. Variance annotations allow the flexibility of traditional arrays with the static type guarantees of generic classes. The result is a potential for increased code reuse and improved static type safety of libraries and frameworks.

Improved type inference: Polymorphic methods in Generic Java have their type arguments inferred from the call context. The presence of variant types often allow the inference mechanism to do a much better job.

Language integration: To a certain degree, Generic Java alienates arrays and generic classes, because the former require runtime type checks that the latter do not support. Variance introduces a class of arrays that are fully checked at compile time, thereby allowing a smoother integration of generic classes and array types.

This white paper describes the variance mechanism from a usage perspective, highlighting its consequences to the programmer if distributed as part of a future release of the Java platform.

2 Variance on generic classes

In Generic Java, as described in JSR-14, there is no type relationship between e.g. `Set<Integer>` and `Set<Number>`. To declare a variable that can alternately hold instances of both, we have two options for its type: We can use `Object`, which is safe, but hides all their common properties. Or we can use the *raw type* `Set`, which may result in unsafe code, and still hides the element type of the sets.

Variance is designed to provide better types for this kind of situation. In the above example, there might be three reasons why we wish to hold on to common properties of the two `Set` types:

Read-only: We want to read from the `Set` (e.g. using its `iterator()` method), knowing that what we get out are certainly instances of `Number`

Write-only: We want to fill entries into the `Set`, knowing that it will certainly hold `Integer` instances

Other manipulation: We do not care about the element type, but only wish to operate on the `Set` using element type independent operations, such as `size()` or `clear()`.

These three situations correspond to the three kinds of variance annotation proposed:

Covariance

To accommodate the read-only situation, we may declare and use our variable as follows:

```
Set<+Number> aReadOnlySet;
...
aReadOnlySet = new HashSet<Integer>();
Iterator<+Number> i = aReadOnlySet.iterator();
double sum = .0;
while(i.hasNext()) sum += i.next().doubleValue();
```

The type `Set<+Number>` is said to be *covariant* in `Number`. The covariance annotation '`+`' signals that the type ranges over all instances of `Set<T>`, where `T` is a specialization (e.g. a subclass) of `Number`. Thus it can be used e.g. for an instance of `Set<Integer>`, as in the example above. The downside of this flexibility is that we cannot call methods on it, which take arguments of the element type. If we allowed

```
aReadOnlySet.add(new Double(.75); // illegal
```

in the situation where `aReadOnlySet` contains an instance of `Set<Integer>` we would get a type error.

The term *co*-variance refers to the direction of specialization: `Set<+B>` specializes `Set<+A>` if `B` specializes `A`: they vary in the same direction.

Contravariance

The write-only situation is quite symmetrical to covariance, and is handled with the `contravariance` annotation '`-`':

```
Set<-Integer> aWriteOnlySet;
...
aWriteOnlySet = new HashSet<Number>();
aWriteOnlySet.add(new Integer(10));
```

Thus, `Set<-Integer>` ranges over all instances of `Set<T>`, where `Integer` is a specialization of `T`. An instance of e.g. `Set<Number>` may be assigned to it. On the downside, if we call methods on it which return results of the element type, we know nothing about the type of the returned element.

Again, the term *contra*-variance refers to the direction of specialization: `Set<-A>` specializes `Set<-B>` if `B` specializes `A`: they vary in the opposite direction.

Bivariance

Finally, if we do not care about the element type, we may make use of the *bivariance* type argument '`*`':

```
Set<*> anUnkownSet;
...
anUnkownSet = new HashSet<Number>();
if (!anUnknownSet.isEmpty()) anUnknownSet.clear();
```

The type `Set<*>` ranges over all instances of `Set<T>` for any `T`. It is called *bi*-variant because it is the combination of the two above variances: one may neither read from nor write to instances.

Invariance

In light of the new forms of type arguments introduced above, one may refer to the plain type arguments of Generic Java as *in*-variant: `Set<A>` specializes `Set<B>` only if `A` equals `B`. Invariant `Set`s of different element type do not mingle, and may therefore be both read from and written to.

For clarity, the invariance may be explicitly denoted by '`=`' as in `Set<=Integer>`.

A specific instance of `Set` must always have a particular element type. For this reason, type arguments to the classes/interfaces of `new` expressions as well as `extends` and `implements` clauses must always be invariant.

2.1 Library classes

Variance annotations are useful for loosening up the types of methods that do not both read from and write to their arguments. This is commonly the case in Java platform libraries, such as the `Collection` classes. As an example, take the `addAll()` method of the `Collection` interface: it only needs to *read* from its argument `Collection`, which can therefore be covariant:

```
public interface Collection<E> {
  ...
  public boolean addAll(Collection<+E> c);
}
```

As an example of a contravariant library method, consider the polymorphic `fill()` and `copy()` methods of the `Collections` class:

```
  public static <T> void fill(List<-T> list, T o) { ... }
  public static <T> void copy(List<-T> dest, List<+T> src) { ... }
```

Variance may also be useful in constructor declarations. Consider the three nontrivial constructors of `TreeSet`:

```
public class TreeSet<E> extends ... {
  public TreeSet(Comparator<-E> c) { ... }
  public TreeSet(Collection<+E> c) { ... }
  public TreeSet(SortedSet<E> s) { ... }
  ...
}
```

When taking a `Comparator`, the `TreeSet` only needs to write to it, in order to determine the ordering of its elements. Thus, it may accept a more *general* `Comparator` than `Comparator<E>`.

On the other hand, when taking a `Collection`, the `TreeSet` will only read from it, adding its elements to itself. Therefore it may accept a `Collection` of a more *special* element type than `Collection<E>`.

Finally, when taking a `SortedSet`, the `TreeSet` reuses both its elements and its `Comparator`. Thus it combines the needs of the two above constructors to both read and write its elements to the element type of the `SortedSet`, which must therefore be exactly `E`.

2.2 Applied variance: a large example

The usefulness of variance is best appreciated in larger examples. In order to promote decoupling and reuse, large scale applications often employ a number of design patterns giving rise to complex structures of mutually dependent objects. Genericity is an appealing approach to increase the abstraction of such systems, but the invariant requirements of Generic Java often become to restrictive.

An example is an event handling library consisting of listeners and providers. Both concepts are represented as interfaces parameterized with an `Event` type, on the grounds that specific `EventProvider`s generate specific kinds of `Event`s just as different `EventListener`s may be able to handle only certain kinds of `Event`s:

```
public interface Event {
  /** Return the original provider of the event */
   EventProvider<*> source();
}

public interface EventProvider<E extends Event> {
  /** register a listener */
  void addListener(<EventListener<-E>> listener);

  /** unregister a listener */
  void removeListener(<EventListener<-E>> listener);
}

public interface EventListener<E extends Event> {
  /** handle an event */
  void eventHappened(E event);
}
```

There may also be a supporting partial implementation of the `EventProvider`

interface:

```
public abstract class AbstractEventProvider<E extends Event>
  implements EventProvider<E>
{
  private List<EventListener<-E>> listeners
    = new ArrayList<EventListener<-E>>();
  public void addListener(EventListener<-E> listener) {
    listeners.add(listener);
  }
  public void removeListener(EventListener<-E> listener) {
    listeners.remove(listener);
  }
  protected void fireEvent(E event) {
    Iterator<EventListener<-E>> i = listeners.iterator();
    while(i.hasNext()) i.next().eventHappened(event);
  }
}
```

The class implements the register of `EventListener`s, and provides a `fireEvent()` method to its subclasses to handle the distribution of events. Because of variance, the `listeners` register may contain a variety of `EventListener`s at any given time, all with the common property that they accept events of type `E`.

In a given application there may be a hierarchy of event types:

```
public class GUIEvent implements Event {
  public EventProvider<*> source() { ... }
  public Object getComponent() { ... }
}
public class MouseEvent extends GUIEvent {
  public int getX() { ... }
  public int getY() { ... }
}
```

For these, a number of providers and listeners exist, e.g.:

```
public class Window extends AbstractEventProvider<GUIEvent> { ... }
public class Mouse extends AbstractEventProvider<MouseEvent> { ... }
public class GUILogger implements EventListener<GUIEvent> {
  public void eventHappened(GUIEvent event) {
    System.out.println("GUIEvent from "+e.getComponent());
```

```
  }
}
public class MouseLogger implements EventListener<MouseEvent> {
  public void eventHappened(MouseEvent event) {
    System.out.println("MouseEvent at "+e.getX()+","+e.getY());
  }
}
```

These may be combined in the following ways:

```
Window window;
Mouse mouse;
GUILogger gl;
MouseLogger ml;
window.addListener(gl);
mouse.addListener(gl);
mouse.addListener(ml);
```

But not:

```
window.addListener(ml); // illegal
```

because `window.add()` expects an `EventListener<-GUIEvent>` which can handle *all* `GUIevents`, while a `MouseLogger` is an `EventListener<MouseEvent>`, which can only handle `MouseEvents`.

In real life, a GUI application like this contains many kinds of events, providers and listeners, which must be dynamically configured and reconfigured in complex structures. With standard invariant generic classes, they would all have to be declared with the same event type in order to fit together. In other words all classes, however specialized, would "pose" as dealing with events in general, causing a widespread need for runtime casting to reestablish specific event information. Thus, the possibility of a reliable compile time check of the configuration operations is lost. With variance, components can declare their "real" event type without risk of not fitting in with the others, thereby holding on to type information that would otherwise have to be reestablished by the use of casts.

2.3 Improved type inference

Polymorphic methods in Generic Java can be called without explicitly giving type arguments to the method: they are inferred on the basis of the call site type information. As an example, given the declarations

```
<T> T choose(T t1, T t2);
Set<Integer> is;
List<String> sl;
```

what is the type of `choose(is,sl)`? Although both arguments are `Collection`s, in Generic Java the inference mechanism will have to disregard this information, because the most specific type that ranges over both arguments rather disappointingly is `Object`. With variant types, although the element types of the two collections differ, we may still infer `T` to be a `COllection` of *something*, i.e., `Collection<*>`.

3 Variance on array types

Variance annotations may also be applied to array types, denoting variance of their element type. The annotation is placed within the square brackets as in:

```
Number[+] nums = new Integer[10]; // Covariant/read-only
Integer[-] ints = new Number[20]; // Contravariant/write-only
String[=] strings = new String[30]; // Invariant/read-write
```

As with generic classes, `new` expressions on arrays always create invariant arrays, so when a dimension is specified, no variance annotation is needed, avoiding the potential syntactic conflict of having both within the same pair of brackets. The notation extends to multidimensional arrays, allowing expressions such as `new Number[10][+]`, which allocates a size 10 array with element type `Number[+]`.

For syntactic reasons, there is no bivariance on arrays. The type `Object[+]` has the same property of ranging over all possible array types, and may be used instead.

Contrary to generic classes, the explicit '`[=]`' annotation of invariance is not optional, because traditional array types with '`[]`' are not invariant: by allowing assignments such as

```
Number[] numbers = new Integer[10];
```

they are closer to being covariant, but unlike explicitly covariant array types they do allow write operations. These are checked at runtime with the risk of throwing an `ArrayStoreException`, as in

```
numbers[0] = new Double(.0); // Runtime store check
```

For this reason, we refer to the "old" style of array types as *dynamic arrays*, whereas the new explicitly variance-annotated array types, with `[=]`,`[+]` and `[-]`, may be called *statically safe* arrays, because their runtime store checks do not fail.

3.1 Arrays of generic classes

Generic classes are implemented by means of a technique called *erasure*, which removes information about type arguments from the compiled version of the code. This means that generic classes cannot be expected to behave right in runtime type checks, and hence not in the store checks of dynamic arrays. The following example illustrates the problem:

```
Object[] objects = new Comparable<Integer>[10]; // Warning
objects[0] = new Double(.0);      // Runtime check succeeds
```

This should really fail at runtime, but the store check does not have the information at hand to distinguish `Comparable<Integer>` from `Comparable<Double>`, so the assignment is wrongfully allowed. To prevent this kind of situation, a warning is issued whenever an array of generic element type is assigned to a dynamic array type. For compatibility reasons, it is not a compile time error: combinations of old and new code should be allowed. However, if the warning is not the result of integration with old code, programmers should regard it as an error and rearrange their code to avoid it, typically by shifting to statically safe arrays.

For similar reasons, dynamic arrays of generic classes and of type variables are not allowed:

```
class C<T> {
  T[] ts;         // Rejected
  C<Object>[] cs; // Rejected
}
```

Such examples cannot occur as a result of integration with old code, as both generic classes and type variables are new features of Generic Java. Hence, the declarations cause compile time errors.

3.2 Library classes

All array-related methods of the standard API are updated to exclusively use statically safe array types. This is to ensure that the methods apply to all array objects, including those that involve generic element types.

A few examples from `java.util.Arrays`:

```
public static <T> void sort(T[=] a, Comparator<-T> c) { ... }
public static <T> void fill(T[-] a, T val) { ... }
```

The `sort()` method needs to both read and write the array elements, and hence invariance on its array argument, whereas the `fill()` method only needs to write to its array argument, and is therefore declared to be contravariant.

The changes may cause old code to emit warnings, as in

```
Number[] nums = new Integer[20];   // dynamic array
Arrays.fill(nums, new Double(.0)); // warning
```

This code abuses the assumption of the `fill()` method that input arrays will always accept the filling object passed in, but cannot be made illegal for compatibility reasons.

4 Implementation of variance

Variance annotations are implemented as an extension to `javac` which compiles source code into JVM byte code. The extension involves passing the new syntax, checking the augmented types and transforming the code to a version without variance. Also, method and constructor signatures in `java.lang` and `java.util` have been modified to include variance.

The extension is closely integrated with the generic extension proposed under JSR-14, and shares most of its advantages and limitations. It remains compatible with old code, and the output bytecode runs on unmodified JVM platforms. The downside, lack of runtime type information, also remains, although statically safe array types eliminate a particular problem in Generic Java by allowing arrays of generic classes.

The current experimental implementation is the result of a joint research project of Sun Microsystems, Aarhus University and the Alexandra Institute.

References

[BOSW98] Gilad Bracha, Martin Odersky, David Stoutamire, and Philip Wadler. Making the future safe for the past: Adding genericity to the Java programming language. In *Proc. OOPSLA'98*, October 1998.

[GJSB00] James Gosling, Bill Joy, Guy Steele and Gilad Bracha. *The Java Language Specification, Second Edition*. Java Series, Sun Microsystems, 2000. ISBN 0-201-31008-2.

[IgVi02] Atsushi Igarashi and Mirko Viroli. *On Variance-based Subtyping for Parametric Types*. ECOOP 2002

[ThoTor99] Kresten Krab Thorup and Mads Torgersen *Unifying Genericity: Combining the benefits of virtual types and parameterized classes*. ECOOP 1999