1996

Safe Structural Conformance for Java

Konstantin Laufer

Gerald Baumgartner

Vincent F. Russo

Report Number:
96-077

SAFE STRUCTURAL CONFORMANCE FOR JAVA

Konstantin Laufer
Gerald Baumgartner
Vincent F. Russo

CSD-TR 96-077
December 1996

Safe Structural Conformance for Java

Konstantin Läufer* Gerald Baumgartner** Vincent F. Russo**

* Department of Mathematical and Computer Sciences
Loyola University
Chicago, IL 60626, USA
laufer@math.luc.edu

** Department of Computer Sciences
Purdue University
West Lafayette, IN 47907, USA
{gb, russo}@cs.purdue.edu

December 6, 1996

Technical Report CSD-TR-96-077
Department of Computer Sciences, Purdue University

Abstract

In Java, an interface specifies public abstract methods and associated public constants. Conformance of a class to an interface is by name. We propose to allow structural conformance to interfaces as well: Any class that provides an implementation for each method in an interface conforms structurally to the interface, and any instance of the class can be used where a value of the interface type is expected. We argue that structural conformance results in a major gain in flexibility in situations that require retroactive abstraction over types.

Our extension comes almost for free. No additional syntax and only minor modifications to the Java compiler and virtual machine are required. Our extension is type-safe: a cast-free program that compiles without errors will not have any type errors at run time, and it is conservative: existing Java programs compile and run in the same manner as under the original language definition. Finally, our extension works well with recent extensions such as the Java Distributed Object Model.

We have implemented our extension of Java with structural interface conformance by modifying the Java Developers Kit 1.0.2 source release for Solaris. We have also created a test suite for the extension.

1 Introduction

Java [GJS96] differs from many statically typed object-oriented languages by offering two separate constructs, the *class* and the *interface*, for user-defined abstract types. An interface contains only public abstract methods and public final fields (constants), but no method implementations. This separation allows independent class and interface hierarchies and solves some of the shortcomings of object-oriented languages that use classes for defining both interfaces and implementations.

We argue that Java is still limited by the requirement that classes must be declared to conform to their interfaces. A class declared to conform to an interface must provide implementations for each method in the interface or leave the method abstract. We identify interfaces in Java with signatures, an extension of C++ [BR95], and propose allowing a class to conform structurally to an interface without explicitly associating the class with the interface.

The remainder of this paper is structured as follows. First, we describe the relevant aspects of the Java type system. Next, we present our extension of Java with structural conformance. Then we give examples in the extended language. Thereafter, we describe the implementation of our extension based on the Java Developers Kit [Sun96a]. Finally, we discuss the integration of structural conformance with the Java Distributed Object Model [Sun96b] and explore possible future changes to the language.

2 Classes and Interfaces in Java

In many statically typed object-oriented languages, for example C++ [ES90], a single construct, namely the *class*, is used to define and implement new types and to provide type abstraction, inheritance, and subtyping. This overuse of a single construct limits the expressiveness of the type system [BR95]. By contrast, Java [GJS96] offers two separate constructs, the *class* and the *interface*.

A class declaration introduces a new type with an implementation that *extends* the implementation of another class called the *immediate superclass*. A class that provides only a partial implementation is called *abstract*. This single implementation inheritance mechanism supports code reuse. The designers of Java have chosen to provide single implementation inheritance to avoid the semantic complexity associated with multiple inheritance [GJS96].

An interface declaration introduces a new type that *specifies* a set of method signatures and some associated named constants, but does not provide an implementation of the methods. An interface may extend one or more other interfaces, that is, it specifies its own method signatures and named constants in addition to all method signatures and named constants of the interfaces it extends. This mechanism provides multiple inheritance of interfaces.

A class may be declared to *implement* one or more interfaces, that is, the class must provide each method specified by the interfaces either explicitly or by inheriting it from a superclass. A class implicitly implements each interface already implemented by the class's superclass and each interface from which the implemented interfaces were extended. This mechanism supports abstraction and specification of common behaviors without code sharing.

Every class or interface has exactly one immediate superclass and zero or more immediate interfaces. The immediate superclass for every interface is `Object`. Since an interface is simply a fully abstract class without any method code, inheriting multiple interfaces poses no semantic difficulties.

Conflicting type and class hierarchies are possible

The separation of the class and interface constructs overcomes many of the problems caused by having a single construct [BR95]. In particular, it is possible in Java to build abstract type hierarchies separate from the corresponding implementation hierarchies. This capability is important because the two hierarchies often evolve in opposite directions.

As an example, consider two abstract types `Queue` and `Dequeue` for FIFO queues and double-ended queues, respectively (a similar example was presented by Snyder [Sny86]). The abstract type `Dequeue` provides the same operations as `Queue` as well as two additional operations for inserting at the head and for removing from the tail of the queue. Therefore, `Dequeue` is a subtype of `Queue`. On the other hand, a `Dequeue` is easily implemented in terms of the array class `java.util.Vector` [GYT96]. A `Queue` is trivially implemented by extending `Dequeue` and

ignoring the superfluous operations.

```
interface Queue {
    Object dequeueHead();
    void enqueueTail(Object x);
    boolean isEmpty();
}

interface Dequeue extends Queue {
    void enqueueHead(Object x);
    Object dequeueTail();
}

class DequeueImpl extends java.util.Vector implements Dequeue {
    public final void enqueueHead(Object x) { insertElementAt(x, 0); }
    public final Object dequeueHead() {
        Object x = firstElement();
        removeElementAt(0);
        return x;
    }
    public final void enqueueTail(Object x) { addElement(x); }
    public final Object dequeueTail() {
        Object x = lastElement();
        removeElementAt(size() - 1);
        return x;
    }
}

class QueueImpl extends DequeueImpl implements Queue {
    // We do not want enqueueHead() and dequeueTail() here,
    // but there is no way to avoid inheriting them.
}

public class Hierarchies {
    public static void main(String[] arg) {
        Queue q1 = new QueueImpl();
        Dequeue q2 = new DequeueImpl();

        q1.enqueueTail("Hello");
        q1.enqueueTail("World");
        System.out.println(q1.dequeueHead());

        q2.enqueueHead("World");
        q2.enqueueHead("Hello");
        System.out.println(q2.dequeueTail());
    }
}
```

The clause `implements Queue` in the declaration of the class `QueueImpl` is actually redundant by transitivity, since the class already extends the class `DequeueImpl`, which implements an interface derived from `Queue`. Furthermore, there are accidental conformance relationships `DequeueImpl implements Queue`, which is acceptable, and `QueueImpl implements Dequeue`, which is undesirable. These relationships are shown in Figure 1. This example illustrates that a true separation of the abstract type and implementation hierarchies is not possible in Java since the language does not allow code reuse without defining a subtype relationship. An example in Section 6 shows how structural conformance together with a renaming mechanism could be used without introducing unwanted conformance relationships.

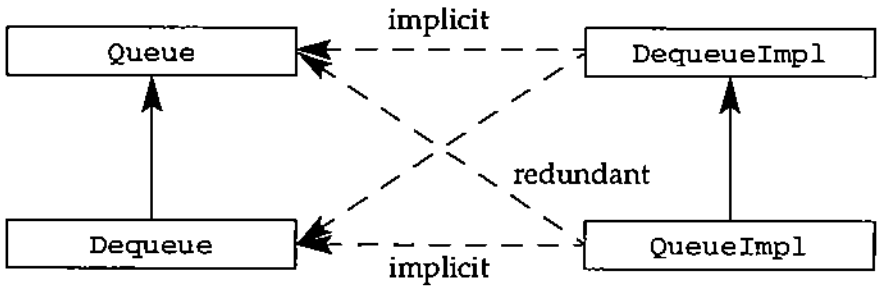

Figure 1: Explicit, implicit, and redundant conformance relationships

Retroactive type abstraction is hard

The integration of different existing class hierarchies, often available only in binary form, may require abstracting over the types of these class hierarchies [BR96]. This usually involves designing a new, common abstract type hierarchy that sits on top of the existing implementation hierarchies. The problem is that we would have to modify the sources of the existing hierarchies to declare their classes as implementations of the interfaces of the new abstract hierarchy.

As an example [GR91], consider two class libraries for X-Window widgets. One hierarchy is rooted at `OpenLookObject` and the other at `MotifObject`. Suppose that all widgets implement the operations `display()` and `move()`. Is it possible to construct a list containing widgets from both class libraries at the same time? The answer is yes, but the solution would involve either introducing a discriminated union for the widgets, or using multiple inheritance to implement a new set of leaf classes in each hierarchy, or building a hierarchy of forwarding classes. All three solutions are undesirable: the first one is inelegant, while the other two introduce many superfluous class names.

In another, perhaps more realistic, scenario that calls for retroactive type abstraction, we want to abstract over some types from an existing class hierarchy and provide an alternative implementation in the form of a new class hierarchy. If the existing class hierarchy was not designed to support this form of reuse or if the alternative implementation uses different data structures, then we are in the same situation as above.

Within the same scenario, assume that the desired abstractions are similar but less specific interfaces than the ones of the existing class hierarchy. We could then provide an alternative implementation with respect to these existing interfaces. For operations that are part of these interfaces, but not of our desired abstraction, we would provide dummy implementations that raise a runtime exception. However, this approach defeats strong typing because those operations are specified in the interface but not really understood by the objects that are supposed to conform to the interface.

For example, suppose we have an interface `File` with operations `read` and `write` but no interface for read-only files. If we write a device driver for a CD-ROM that implements the `File` interface, any application that uses the `File` interface to write to a device cannot detect statically if the device is actually a CD-ROM.

3 Structural Class-to-Interface Conformance

Most object-oriented languages provide some form of conformance or subtyping, allowing an instance of a class to be used wherever an instance of the class's superclass is expected. In Java, conformance is entailed by both the class-superclass relation and the class-interface relation and is called *assignment conversion*. An instance of a class can be used wherever an instance of its imme-

diate or of an indirect superclass is expected. Furthermore, an instance of a class can be used wherever a value of any immediate or indirect interface of the class is expected. By indirect interface we mean an interface of a superclass or an interface from which an immediate interface of the class was extended. Since the relationships that entail conformance are established by declaration, this form of conformance is called *conformance by name*.

We propose to allow structural conformance as well, but only between a class and an interface. Specifically, any class that provides an implementation or abstract declaration for each method in an interface *conforms structurally* to the interface, and any instance of this class can be used wherever a value of the interface type is expected. It is no longer required for the class to conform to the interface by name.

We limit structural conformance to assignments whose target types are interfaces because the purpose of interfaces is to specify behavior without giving an implementation. On the other hand, we make no attempt to allow structural inference of class-subclass relationships since subclassing provides code reuse for implementation purposes. Therefore, it would make no sense to try to infer such a relationship based on the public methods of a class.

This extension comes entirely for free from the programmer's viewpoint. No changes to the language syntax are necessary. Only minor modifications to the Java compiler and run time are required and are described below. Our extension is type-safe in the sense that any cast-free program that compiles without type errors will not cause any type errors at run time. The extension is conservative in the sense that existing Java programs still compile and execute in the same manner as under the original language definition.

Formal definition of conformance

In the following definition of conformance, each of `C` and `D` is either a class or an interface. Every class or interface has one immediate superclass and zero or more declared interfaces. The immediate superclass of every interface is `Object`, the root class of the Java class hierarchy.

`C` *conforms by name* to `D` if and only if

- `C` is identical to `D`, or
- `C`'s immediate superclass conforms by name to `D`, or
- some declared interface `I` of `C` conforms by name to `D`.

`C` *conforms structurally* to `D` if and only if

- `C` conforms by name to `D`, or
- `D` is an interface for which the compiler does not require conformance by name, and
- `C` overrides each method specified in `D`, and
- `C` conforms structurally to each declared interface of `D`

`C` *overrides* each method `f` specified in `D` if and only if for each method specified in `D`, there is a method `f` in `C` that is at least as accessible as `D.f`, has the same type signature as `D.f`, and throws only checked exceptions that are subclasses of the checked exceptions that `D.f` throws. This definition of overriding is equivalent to the one given in the Java language specification [GJS96] and includes implementing the method as well as leaving it abstract. Furthermore, since `D` must be an interface, all its methods (and constants) are public, and those methods in `C` that override methods in `D` must be public as well.

By-name-only interfaces

Certain interfaces are treated differently by the compiler and must be excluded from structural conformance; we call them *by-name-only* interfaces. For example, the interface `Cloneable` is empty, so every class would conform to it structurally. However, when a class chooses to implement `Cloneable`, it does so to indicate to the compiler that it actually supports the method `clone()`. This method is understood by every Java object, but raises the exception `CloneNotSupported` if the class does not implement the interface `Cloneable`. While `Cloneable` currently is the only by-name-only interface, other by-name-only interfaces are likely to be needed. For example, the proposed Java Distributed Object Model [Sun96b] provides the empty interface `Remote`, which all interfaces of remote objects extend directly or indirectly. By-name-only interfaces usually represent semantic properties satisfied by classes that implement those interfaces.

Type safety

For the purpose of this paper, we define type safety as follows:

> Any program without casts that compiles correctly will not raise a `NoSuchMethodError` or any other type-related error or exception.

The idea behind this definition is that the compiler is able to prove from the type information available at compile-time that the run-time behavior of a program is safe. Hence we allow an assignment conversion only when the assigned value provides at least the public methods required by the type of the assignment target. To be on the safe side, the Java[1] compiler verifies that a new class is a suitable implementation of an interface or a subclass of an existing class at the point where the new class is defined. This is the conformance-by-name approach.

However, this is not the only way to guarantee type-safe assignment conversion. Instead of requiring an `implements` declaration between classes and interfaces, it is equally safe but much more flexible to examine each intended assignment conversion individually. Since the compiler knows the types of both the target and the source of an assignment, it can check whether the class of the source provides at least the public methods required by the interface type of the target without requiring a prior declaration. This is the structural conformance approach.

Avoiding accidental conformance

The usual objection to structural conformance is that it creates the possibility of *accidental* conformance. The solution is to include *properties* (dummy methods with well-known *names*) in the interface to represent a semantic specification of the interface type. For example, a stack interface might include the property `LIFO`. Any stack implementation only conforms to the interface if it also includes the property `LIFO`. What we achieve in this way is exactly the same as name conformance with the names now being the names of the properties. A programmer cannot be prevented from putting the `LIFO` property into a queue implementation, but, on the other hand, a programmer could also write a queue implementation and say it "`implements Stack`".

4 Examples

In this section, we present examples illustrating the capabilities of structural conformance.

1. Actually, Java is not statically type-safe with respect to the assignment of arrays [GJS96, Coo95]. An modification that makes Java arrays type-safe is discussed in Section 6, and a corresponding patch for the JDK source is found in Appendix C.

A simple example: queues revisited

Our first example resembles the queue/double-ended queue example from above, but uses structural conformance. The interfaces and classes are defined as above, but the classes no longer have `implements` clauses.

```
interface Queue {
    Object dequeueHead();
    void enqueueTail(Object x);
    boolean isEmpty();
}

interface Dequeue extends Queue {
    void enqueueHead(Object x);
    Object dequeueTail();
}

class DequeueImpl extends java.util.Vector {
    // ...
}

class QueueImpl extends DequeueImpl {
}
```

Now instances of the implementation classes conform structurally to the corresponding interfaces:

```
Queue q3 = new QueueImpl();
Dequeue q4 = new DequeueImpl();
// ...
```

Alternate implementation of a full interface

A common scenario is that we already have an implementation of a class but would like to provide alternate implementations with the same interface as the original class. If we need the capability to switch implementations in the application, all implementation classes must conform to the same interface.

Consider the example of a class for a random-access file on a local disk, which is provided in the `java.io` package:

```
class RandomAccessFile implements DataOutput, DataInput {
    public void close() throws IOException { /* ... */ }
    public FileDescriptor getFD() throws IOException { /* ... */ }
    public long getFilePointer() throws IOException { /* ... */ }
    long length() throws IOException { /* ... */ }
    void seek(long pos) throws IOException { /* ... */ }
    // methods from the two interfaces
}
```

We would like to add an alternate implementation for a remote random-access file, perhaps using the Proxy pattern [GHJV95]. We proceed in two steps, as illustrated in Figure 2. First, we distill the full interface from the existing class. The existing class conforms structurally to this interface.

```
interface RandomAccess extends DataInput, DataOutput {
    void close() throws IOException;
    FileDescriptor getFD() throws IOException;
    long getFilePointer() throws IOException;
    long length() throws IOException;
    void seek(long pos) throws IOException;
```

```
}
```

Next, we implement the new alternate class. We also state that the new class implements the distilled interface. Besides serving documentary purposes, this results in early conformance checking at the time the class is defined instead of late checking when an instance of the class is assigned to an interface variable.

```
class RemoteRandomAccessFile implements RandomAccess {
    // same methods, but with different implementations suitable for remote use
}
```

Without structural conformance, we would have had to write a forwarding class that implements the interface `RandomAccess` by name and forwards requests to the class `RandomAccessFile`.

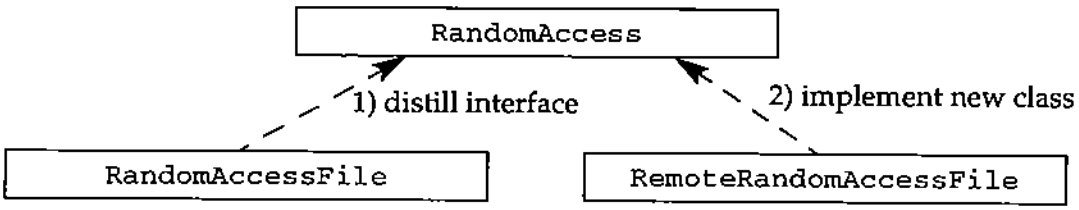

Figure 2: Alternate implementation of a full interface

Abstraction to a reduced interface

In another scenario, we want to deal uniformly with different existing implementations. This requires retroactive abstraction over those implementations to a possibly reduced interface.

For example, consider an application that performs read-only random access to databases located either on a disk drive or on a CD-ROM drive. Again, we access disk files using the class `java.io.RandomAccessFile`, but using only the methods for reading from the file, not the ones for writing to the file. We also provide a class for accessing information on CD-ROM drives. Since CD-ROM drives are read-only devices, the corresponding class implements only the `java.io.DataInput` interface and provides additional methods for random access:

```
class CDRomDrive implements DataInput {
    long length() throws IOException { /* ... */ }
    public void seek(long pos) throws IOException { /* ... */ }
    // implementation of methods from interface DataInput
}
```

Besides the methods for reading data, the application uses the random-access method `seek`. We therefore abstract retroactively over both classes, with a reduced interface as the result. This process is illustrated in Figure 3.

```
interface ReadOnlyRandomAccess extends DataInput {
    long length() throws IOException;
    void seek(long pos) throws IOException;
}
```

Again, without structural conformance, we would have had to write a forwarding class that implements the interface `ReadOnlyRandomAccess` by name and forwards requests to the class `RandomAccessFile`.

Structural subtyping is also useful in the context of persistent objects. Suppose that we have some objects of class `RandomAccessFile` stored on a disk. (Although the example applies to other persistent data structures equally well, files are an obvious choice because they are com-

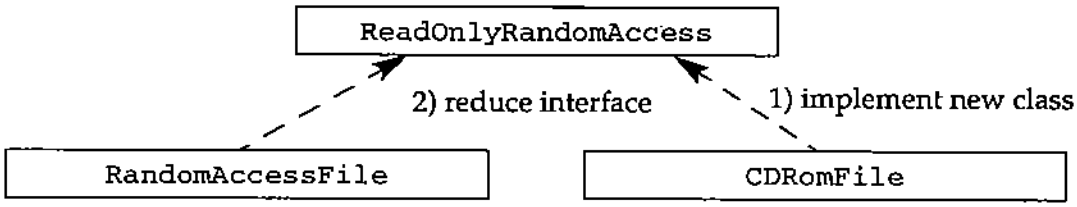

Figure 3: Abstraction to a reduced interface

monly stored on disks.) We later realize that the original class hierarchy was not well-designed and add an interface `ReadOnlyRandomAccess` and a class `ReadOnlyRandomAccessFile` as a superclass of `RandomAccessFile`. If conformance by name is used, we are no longer able to read the old files after the system is upgraded to the new class and interface hierarchies. By contrast, if the persistent data structure is type-checked structurally [CBC+90, MCKM96], evolving the system in this way does not cause any problems.

5 Implementation

While the proposed extension requires no changes to the Java syntax at all, it does require subtle changes to the Java compiler and virtual machine. This section describes the changes made to the Java Developers Kit 1.0.2 source release [Sun96a]. The full changes to the JDK 1.0.2 source release are presented in the form of context diffs in Appendices A and B. In addition, we have developed a test suite for structural conformance based on the test suite for signatures in the GNU C++ compiler [Bau95]. Changes and test suite are available from http://www.math.luc.edu/~laufer/papers/Java/.

Compiler

In the `javac` compiler, the conformance relation between classes or interfaces is represented by the method `ClassDefinition.implementedBy`. We modified this method to allow structural conformance.

`$JAVASRC/src/share/sun/sun/tools/java/ClassDefinition.java`

- The method `ClassDefinition.implementedBy` was modified to fall back to structural conformance if the receiver is an interface. However, by-name-only interfaces such as `Cloneable` must still be implemented by name and cannot participate in structural conformance.
- The following methods were added:

  `ClassDefinition.implementedByNameBy`

  This method checks for conformance by name and is identical to the original method `ClassDefinition.implementedBy`.

  `ClassDefinition.byNameInterfacesOK`

  This method checks that the receiver or any of its interfaces is not a by-name-only interface.

  `ClassDefinition.implementedStructurallyBy`

  This method checks that each public method of the receiver is provided by the argument. This requires first checking each public method specified in the receiver itself. Next, the method checks recursively that each interface of the receiver is also implemented structurally by the

argument.

`$JAVASRC/src/share/sun/sun/tools/javac/SourceClass.java`

- The method `SourceClass.check` was changed to use `ClassDefinition.implementedByNameBy` to check for cyclical interface definitions. This change simply reflects the fact that `ClassDefinition.implementedBy` now allows structural conformance.

Virtual machine

In principle, no changes to the Java virtual machine should be necessary. However, run-time checks are performed in various places when assigning array elements, including the system method `arraycopy`. Therefore, the changes we made to the method `ClassDefinition.implementedBy` must be duplicated in the function `ImplementsInterface` in the run-time system.

`$JAVASRC/src/share/java/runtime/interpreter.c`

- The function `ImplementsInterface` was changed to perform structural conformance.
- The following functions were added in analogy to the new methods in the class `ClassDefinition`:

  `ImplementsInterfaceByName`

  `ByNameInterfacesOK`

  `ImplementsInterfaceStructurally`

- The function `HasPublicMethod` was added to check whether a class has a method with a given type signature.

Adding by-name-only interfaces

Any by-name-only interface that is introduced to the language in the future must be made known to both the compiler and the run-time. In the compiler, we assume that there already is a global identifier for each new by-name-only interface, for example, `idSomePackageSomeInterface`. Then the class definition corresponding to each such interface is added to the array `byNameIntf` in the method `ClassDefinition.byNameInterfacesOK`.

In the run-time, we assume that there already are a definition and an initialization for each new by-name-only interface, for example, `interfaceSomePackageSomeInterface` in the file `$JAVASRC/src/share/java/runtime/classresolver.c` in the function `Locked_InitializeClass`, as well as a declaration in `$JAVASRC/src/share/java/include/interpreter.h`. It is then sufficient to add, for each by-name-only interface, an array element to the array `byNameIntf` in the file `java/runtime/interpreter.c` in the function `ByNameInterfacesOK`.

6 Possible Extensions

This section discusses several possible extensions and modifications to Java that would work well with structural conformance. Among these, method renaming and class import are conservative extensions, while type-safe arrays and deep subtyping are not.

Type-safe arrays

As pointed out by Cook [Coo95], the Java rules for assignment conversion between array types

[GJS96] are unsafe. Java allows assigning an array to another as long as the element type of the source array conforms to the element type of the target array. However, since individual array elements can be assigned to, an array type should conform to another only if their element types are identical. This problem is illustrated by the following example:

```
interface I {
    void f();
}

interface J extends I {
    void g();
}

class C implements I {
    public void f() { System.out.println("C.f"); }
}

class D implements J {
    public void f() { System.out.println("D.f"); }
    public void g() { System.out.println("D.g"); }
}

class ArrayUnsound {
    public static void main(String[] arg) {
        J[] j = new J[2];
        I[] i;
        i = j;          // unsound!
        i[0] = new C(); // ArrayStoreException
        j[0].g();
  }
}
```

This example is accepted by the Java compiler despite the unsound array assignment. As expected, the program raises an `ArrayStoreException`. In our modification, we require identical element types for array conformance. Therefore, our version of the compiler rejects this program, reporting incompatible types in the assignment. The code changes for this proposed extension are found in Appendix C.

Method renaming

Frequently, we would like to establish structural conformance to an interface when only the types, but not the names of corresponding methods match. The usual way to deal with this situation is to write a forwarding class according to the Adapter pattern [GHJV95]. We could avoid this unnecessary cluttering of the class name space if we had a mechanism for renaming methods while establishing structural conformance. We propose to include such a mechanism with a cast-like notation.

The following example shows a general container interface that could be implemented by any container class that allows inserting and removing an element and checking if the container is empty. If we wanted to use the `Stack` class as an implementation, we could then rename its methods to the names used in the interface:

```
interface Queue {
    Object dequeueHead();
    void enqueueTail(Object x);
    boolean isEmpty();
}
```

```
interface Dequeue extends Queue {
    void enqueueHead(Object x);
    Object dequeueTail();
}

class DoublyLinkedList extends Vector {
    public final void addFirst(Object x) { /* ... */ }
    public final Object removeFirst() { /* ... */ }
    public final void addLast(Object x) { /* ... */ }
    public final Object removeLast(Object x) { /* ... */ }
}

DoublyLinkedList d1, d2;
// ...
Queue q5 = (Queue { dequeueHead = removeFirst, enqueueTail = addLast }) d1;
Dequeue q6 = (Dequeue { enqueueHead = addFirst, dequeueHead = removeFirst,
                        enqueueTail = addLast, dequeueTail = removeLast}) d2;

q5.enqueueTail("This string will be added at the end of the queue");
```

This example shows how structural subtyping together with a renaming mechanism could be used without introducing an unwanted conformance relationship.

Importing classes for code reuse

In Java, multiple inheritance is supported for interfaces, but not for classes. Since all methods in an interface are abstract, a class inherits code only from its (single) superclass. This restriction makes it difficult to combine functionality provided by library classes with user-defined class hierarchies. The technique of incorporating an existing implementation into a new class is called *mixin* and is usually expressed through multiple inheritance. Workarounds for languages without multiple inheritance exist, but involve code duplication or forwarding methods.

For example, suppose that we want to provide a remote version of a class in an existing hierarchy. Java provides the class `UnicastRemoteObject` that implements remote objects. We could define a subclass of `UnicastRemoteObject` and duplicate the methods of the existing class. Alternatively, we could define a subclass of our class that contains an instance of `UnicastRemoteObject`, to which the relevant methods are forwarded. Using structural conformance, we can retroactively provide a common interface for the existing class and the remote class, but we cannot do without code duplication or forwarding methods.

However, these workarounds could be avoided if we had a language mechanism for combining library classes with user-defined class hierarchies. The ability to import code without establishing an explicit subclass relationship is sufficient to support the mixin programming style cleanly and directly. We propose extending the class construct in Java to allow *importing* zero or more classes.

```
class C extends D imports E, F, ... implements I, J, ... {
    // ...
}
```

Importing a class would have the same effect as inclusion together with forwarding methods for all the public methods provided by the included class, but not by the new class. For example,

```
class RemoteChatServer extends ChatServer imports UnicastRemoteObject {
    // methods and variables of RemoteChatServer
}
```

would be equivalent to

```
class RemoteChatServer extends ChatServer {
```

```
    private UnicastRemoteObject remote = new UnicastRemoteObject();
    // forwarding methods for each method in UnicastRemoteObject:
    public Object clone() { return remote.clone(); }
    public static void exportObject(Remote obj) { return remote.exportObject(obj); }
    // ...
    // methods and variables of RemoteChatServer
}
```

The proposed import mechanism might have the following properties, among others:

- A method defined in the new class takes precedence over imported methods with the same signature.
- An imported method may provide an implementation of an abstract method from the (abstract) superclass or from an interface.
- Java provides field access through a qualified name or by casting `this`, for example, when accessing a variable in the superclass shadowed by a variable in the subclass. The same mechanism could be used to disambiguate between methods and variables from imported classes.
- As in Java, the receiver `super` would provide *static access* to methods and variables in the immediate superclass.

Deep subtyping

Our definition of structural conformance requires an exact match between the signature of an overriding method and the signature of the corresponding method in the interface. This strict notion of structural conformance sometimes gets in the way of retroactive abstraction. For example, suppose a class over which we want to abstract has a binary method, that is, a method whose argument type is the class itself. In the interface distilled from this class, we would change the argument type of the method to be that interface. However, the class would not conform to the resulting interface because the signatures of the binary method do not match exactly. (A similar mismatch would occur if the method signature contained a different class that we also want to abstract over.)

The following example illustrates this problem. It discusses a common abstraction for menu-based applications providing both a graphical and a telephone interface. To provide this abstraction, we would like to reduce the interfaces of the menu-related Java Abstract Window Toolkit [GYT96] components to a least common denominator for the two platforms. Among others, we would define interfaces for menus and menu bars:

```
package common;

interface Menu {
    // ...
}

interface MenuBar {
    Menu add(Menu m);
    // ...
}
```

The problem is that neither of the corresponding AWT classes, `java.awt.Menu` and `java.awt.MenuBar` conform structurally to our `common.Menu` and `common.MenuBar` classes, respectively. The reason is that the argument type of `common.MenuBar.add` is `common.Menu`, while the argument type of `java.awt.MenuBar.add` is `java.awt.Menu`.

It is possible to fix this problem by using *deep structural conformance* instead of shallow structural conformance. This form of conformance would no longer require an exact match between

the type signatures of the overriding and overridden methods, but would require the type of the overriding method to be a subtype of the type of the overridden method [AC93].

7 Conclusion

We have presented an extension of the Java language that allows structural conformance between classes and interfaces. Any class that provides implementations for each method in an interface conforms structurally to the interface, and any value of the class can be used where a value of the interface type is expected, without having to declare conformance of the class to the interface.

We have argued that structural conformance makes the type system much more flexible in situations that require retroactive abstraction over types, and we have given examples to support this view.

Furthermore, our proposed extension comes almost for free. No additional syntax and only minor modifications to the Java compiler and virtual machine are required. Our extension is type-safe in the sense that a program without casts that compiles without errors will not have any type errors at run time. Our extension is conservative in the sense that existing Java programs compile and run in the same manner as under the original language definition. Finally, our extension works well with the proposed Java Distributed Object Model [Sun96b].

References

[AC93] Roberto M. Amadio and Luca Cardelli. Subtyping recursive types. *ACM Transactions on Programming Languages and Systems*, 15(4):575–631, September 1993.

[Bau95] G. Baumgartner. C++ signature and class-scoping tests. Included in the Cygnus GNU C++ test suite, March 1995. Available from ftp://ftp.cygnus.com/pub/g++/.

[BR95] Gerald Baumgartner and Vincent F. Russo. Signatures: A language extension for improving type abstraction and subtype polymorphism in C++. *Software—Practice & Experience*, 25(8):863–889, August 1995.

[BR96] Gerald Baumgartner and Vincent F. Russo. Implementing signatures for C++. *ACM Transactions on Programming Languages and Systems*, 1996. To appear.

[CBC+90] R. C. H. Connor, A. B. Brown, Q. I. Cutts, A. Dearle, R. Morrison, and J. Rosenberg. Type equivalence checking in persistent object systems. In A. Dearle, G. M. Shaw, and S. B. Zdonik, editors, *Implementing Persistent Object Bases, Principles and Practice*, pages 151–164. Morgan Kaufmann, 1990. Available from http://www-fide.dcs.st-and.ac.uk/Publications/1990.html#type.equiv.

[Coo95] William R. Cook. Array subclassing and IncompatibleTypeException. Posted to the java-interest@java.sun.com mailing list, June 1995. Available from http://java.sun.com/archives/java-interest/0463.html.

[ES90] M. Ellis and B. Stroustrup. *The Annotated C++ Reference Manual*. Addison-Wesley, 1990.

[GHJV95] Erich Gamma, Richard Helm, Ralph E. Johnson, and John M. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series. Addison-Wesley, Reading, Massachusetts, 1995.

[GJS96] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. Sun Microsystems, Mountain View, California, Version 1.0 edition, August 1996. Available from http://java.sun.com/doc/language_specification/.

[GR91] Elana D. Granston and Vincent F. Russo. Signature-based polymorphism for C++. In *Proceedings of the 1991 USENIX C++ Conference*, pages 65–79, Washington, D.C., 22–25 April 1991. USENIX Association.

[GYT96] James Gosling, Frank Yellin, and The Java Team. *Java API Documentation*. Sun Microsystems, Mountain View, California, Version 1.0.2 edition, April 1996. Available from http://java.sun.com/products/JDK/1.0.2/api/.

[MCKM96] Ron Morrison, Richard Connor, Graham Kirby, and David Munro. Can Java persist? In *Proc. Persistent Java Workshop*, Scotland, 1996.

[Sny86] Alan Snyder. Encapsulation and inheritance in object-oriented programming languages. In *Proceedings of the OOPSLA'86 Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 38–45, Portland, Oregon, 29 September - 2 October 1986. Association for Computing Machinery. *ACM SIGPLAN Notices*, 21(11), November 1986.

[Sun96a] Java Developers Kit Release 1.0.2, May 1996. Available from http://java.sun.com/products/JDK/.

[Sun96b] Java Remote Method Invocation Prebeta, November 1996. Available from http://chatsubo.javasoft.com/current/rmi/.

A Code Changes to the Compiler

This and the following appendix present the code changes that implement structural interface conformance. The changes are given in the form of context diffs for the Java Developers Kit 1.0.2 source release [Sun96a].

```
*** src/share/sun/sun/tools/java/ClassDefinition.java.orig Wed May  8 17:47:05 1996
--- src/share/sun/sun/tools/java/ClassDefinition.java      Sun Sep 29 03:01:14 1996
***************
*** 270,290 ****
      /**
       * Check if this class is implemented by another class
       */
!     public boolean implementedBy(Environment env, ClassDeclaration c) throws Clas
sNotFound {
!       for (; c != null ; c = c.getClassDefinition(env).getSuperClass()) {
            if (getClassDeclaration().equals(c)) {
                return true;
            }
            ClassDeclaration intf[] = c.getClassDefinition(env).getInterfaces();
            for (int i = 0 ; i < intf.length ; i++) {
!               if (implementedBy(env, intf[i])) {
                    return true;
                }
            }
        }
        return false;
      }


      /**
       * Check if a class can be accessed from another class
--- 270,402 ----
      /**
       * Check if this class is implemented by another class
```

```
         */
!       public boolean implementedBy(Environment env, ClassDeclaration decl)
!       throws ClassNotFound {
!         return
!               implementedByNameBy(env, decl)
!             || (isInterface()
!                 && byNameInterfacesOK(env, decl)
!                 && implementedStructurallyBy(env, decl));
!       }
!
!       /**
!        * Check if this class is implemented by name only by another class
!        */
!       public boolean implementedByNameBy(Environment env, ClassDeclaration decl)
!       throws ClassNotFound {
!         for (ClassDeclaration c = decl;
!              c != null;
!              c = c.getClassDefinition(env).getSuperClass()) {
              if (getClassDeclaration().equals(c)) {
                  return true;
              }
              ClassDeclaration intf[] = c.getClassDefinition(env).getInterfaces();
              for (int i = 0 ; i < intf.length ; i++) {
!                 if (implementedByNameBy(env, intf[i])) {
                      return true;
                  }
              }
         }
+        if (env.verbose()) {
+            env.output("[class " + this.getName() + " not implemented by name by "
+                       + decl.getName() + "]");
+        }
         return false;
       }

+      /**
+       * Check if all by-name-only interfaces implemented by this class
+       * are also implemented by the other class
+       */
+      private boolean byNameInterfacesOK(Environment env, ClassDeclaration decl)
+      throws ClassNotFound {
+        ClassDefinition byNameIntf[] = {
+            env.getClassDeclaration(idJavaLangCloneable).getClassDefinition(env)
+            // add other by-name-only interfaces here
+        };
+        for (int i = 0 ; i < byNameIntf.length ; i++) {
+            if (byNameIntf[i].implementedByNameBy(env, getClassDeclaration())
+                && !byNameIntf[i].implementedByNameBy(env, decl)) {
+                if (env.verbose()) {
+                    env.output("[interface " + byNameIntf[i].getName()
+                               + " not implemented by name by "
+                               + decl.getName() + "]");
+                }
+                return false;
+            }
+        }
+        return true;
+      }
+
```

```
+     /**
+      * Check if this class is implemented structurally by another class
+      */
+     private boolean implementedStructurallyBy(Environment env,
+                                               ClassDeclaration c)
+     throws ClassNotFound {
+         // check whether all public methods of this class are provided by c
+       FieldDefinition f;
+         for (f = getFirstField() ; f != null ; f = f.getNextField()) {
+                   if (f.isMethod() && f.isPublic()) {
+                 // find this method in class c
+               Type ftype = f.getType();
+                 FieldDefinition field =
+                     c.getClassDefinition().findMethod(env, f.getName(), ftype);
+
+               String sig = "";
+               if (env.verbose()) {
+                   sig = ftype.getReturnType().toString() + " " +
+                         Type.tMethod(ftype.getReturnType(),
+                                      ftype.getArgumentTypes())
+                             .typeString(f.getName().toString(), false, false);
+               }
+
+                 // check if found, access, and argument and return types
+                 if (field == null
+                   || !field.isPublic()
+                   || !field.getType().getReturnType()
+                       .equals(ftype.getReturnType())) {
+                   if (env.verbose()) {
+                       env.output("    [" + sig + " not found!]");
+                   }
+                     return false;
+                 }
+
+                 // check exceptions thrown
+                 ClassDeclaration[] e1 = field.getExceptions(env);
+                 ClassDeclaration[] e2 = f.getExceptions(env);
+                 for (int i = 0 ; i < e1.length ; i++) {
+                     ClassDefinition c1 = e1[i].getClassDefinition(env);
+                     boolean ok = false;
+                     for (int j = 0 ; j < e2.length ; j++) {
+                         if (c1.subClassOf(env, e2[j])) {
+                             ok = true;
+                             break;
+                         }
+                     }
+                     if (!ok) {
+                         env.error(field.getWhere(), "invalid.throws",
+                                   c1, f, f.getClassDeclaration());
+                         return false;
+                     }
+               }
+
+               if (env.verbose()) {
+                   env.output("    [" + sig + " found]");
+               }
+           }
+       }
+
```

```
+         // recursively look in the interfaces
+       ClassDeclaration intf[] = getInterfaces();
+       for (int i = 0 ; i < intf.length ; i++) {
+           if (!intf[i].getClassDefinition().implementedStructurallyBy(env, c)) {
+               return false;
+           }
+       }
+
+       return true;
+     }

      /**
       * Check if a class can be accessed from another class
*** src/share/sun/sun/tools/javac/SourceClass.java.orig Wed May  8 17:47:32 1996
--- src/share/sun/sun/tools/javac/SourceClass.java      Sun Sep 29 02:12:45 1996
***************
*** 377,383 ****
                      env.error(getWhere(), "cant.access.class", intf);
                  } else if (!intf.getClassDefinition(env).isInterface()) {
                      env.error(getWhere(), "not.intf", intf);
!                 } else if (isInterface() && implementedBy(env, intf)) {
                      env.error(getWhere(), "cyclic.intf", intf);
                  }
              } catch (ClassNotFound ee) {
--- 377,383 ----
                      env.error(getWhere(), "cant.access.class", intf);
                  } else if (!intf.getClassDefinition(env).isInterface()) {
                      env.error(getWhere(), "not.intf", intf);
!                 } else if (isInterface() && implementedByNameBy(env, intf)) {
                      env.error(getWhere(), "cyclic.intf", intf);
                  }
              } catch (ClassNotFound ee) {
```

B Code Changes to the Virtual Machine

This appendix presents the code changes to the Java virtual machine.

```
*** src/share/java/runtime/interpreter.c.orig   Wed May  8 17:39:29 1996
--- src/share/java/runtime/interpreter.c        Sun Sep 29 03:29:26 1996
***************
*** 809,845 ****
      }
  }

  /* Return TRUE if cb implements the interface icb */

  bool_t
  ImplementsInterface(ClassClass *cb, ClassClass *icb, ExecEnv *ee)
  {
      /* Let's avoid resolving the class unless we really have to.  So first,
!      * First, do a string comparison of icb against all the interfaces thant
       * cb implements.
       */
      char *icb_name = icb->name;
      cp_item_type *constant_pool = cb->constantpool;
      int i;
      for (i = 0; i < (int)(cb->implements_count); i++) {
        int interface_index = cb->implements[i];
```

```
        char *interface_name =
                GetClassConstantClassName(constant_pool, interface_index);
!       if (strcmp(icb_name, interface_name) == 0)
            return TRUE;
      }
      /* See if any of the interfaces that we implement possibly implement icb */
      for (i = 0; i < (int)(cb->implements_count); i++) {
        int interface_index = cb->implements[i];
        ClassClass *sub_cb;
        if (!ResolveClassConstantFromClass(cb, interface_index, ee,
!                                          1 << CONSTANT_Class))
            return FALSE;
        sub_cb = (ClassClass *)(constant_pool[interface_index].p);
!       if (ImplementsInterface(sub_cb, icb, ee))
            return TRUE;
      }
      return FALSE;
  }

  bool_t array_is_instance_of_array_type(JHandle * h, ClassClass *cb, ExecEnv *ee)
--- 809,967 ----
      }
  }

+ /* auxiliary functions for ImplementsInterface */
+
+ static bool_t ImplementsInterfaceByName(ClassClass *cb, ClassClass *icb, ExecEnv
*ee);
+ static bool_t ImplementsInterfaceStructurally(ClassClass *cb, ClassClass *icb, Ex
ecEnv *ee);
+ static bool_t ByNameInterfacesOK(ClassClass *cb, ClassClass *icb, ExecEnv *ee);
+ static bool_t HasPublicMethod(ClassClass *cb, char* nm, char* sig);
+
  /* Return TRUE if cb implements the interface icb */

  bool_t
  ImplementsInterface(ClassClass *cb, ClassClass *icb, ExecEnv *ee)
  {
+     return
+         ImplementsInterfaceByName(cb, icb, ee)
+         || (ByNameInterfacesOK(cb, icb, ee)
+             && ImplementsInterfaceStructurally(cb, icb, ee));
+ }
+
+ /* Return TRUE if cb implements the interface icb by name */
+
+ static bool_t
+ ImplementsInterfaceByName(ClassClass *cb, ClassClass *icb, ExecEnv *ee)
+ {
      /* Let's avoid resolving the class unless we really have to.  So first,
!      * First, do a string comparison of icb against all the interfaces that
       * cb implements.
       */
      char *icb_name = icb->name;
      cp_item_type *constant_pool = cb->constantpool;
      int i;
+
      for (i = 0; i < (int)(cb->implements_count); i++) {
        int interface_index = cb->implements[i];
```

```
          char *interface_name =
                  GetClassConstantClassName(constant_pool, interface_index);
!         if (strcmp(icb_name, interface_name) == 0) {
!               fprintf(stderr, "[ImplementsInterfaceByName(%s, %s, env) OK]\n",
!                       cb->name, icb->name);
!               fflush(stderr);
              return TRUE;
+           }
        }
        /* See if any of the interfaces that we implement possibly implement icb */
        for (i = 0; i < (int)(cb->implements_count); i++) {
          int interface_index = cb->implements[i];
          ClassClass *sub_cb;
          if (!ResolveClassConstantFromClass(cb, interface_index, ee,
!                                            1 << CONSTANT_Class)) {
!
!               fprintf(stderr, "[ImplementsInterfaceByName(%s, %s, env) failed]\n",
!                       cb->name, icb->name);
!               fflush(stderr);
              return FALSE;
+           }
          sub_cb = (ClassClass *)(constant_pool[interface_index].p);
!         if (ImplementsInterfaceByName(sub_cb, icb, ee)) {
!               fprintf(stderr, "[ImplementsInterfaceByName(%s, %s, env) OK]\n",
!                       cb->name, icb->name);
!               fflush(stderr);
              return TRUE;
+           }
        }
+       fprintf(stderr, "[ImplementsInterfaceByName(%s, %s, env) failed]\n",
+               cb->name, icb->name);
+       fflush(stderr);
        return FALSE;
+ }
+
+ static bool_t
+ ByNameInterfacesOK(ClassClass *cb, ClassClass *icb, ExecEnv *ee)
+ {
+       int byNameIntfCount = 1;
+       ClassClass* byNameIntf[1];
+       int i;
+
+       byNameIntf[0] = interfaceJavaLangCloneable;
+       /* add other by-name-only interfaces here */
+
+       for (i = 0; i < byNameIntfCount; i++) {
+           if (ImplementsInterfaceByName(icb, byNameIntf[i], ee)
+               && !ImplementsInterfaceByName(cb, byNameIntf[i], ee)) {
+               fprintf(stderr, "[%s NOT implemented by name by %s]\n",
+                       byNameIntf[i]->name, cb->name);
+               fflush(stderr);
+               return FALSE;
+           }
+       }
+       fprintf(stderr, "[ByNameInterfacesOK(%s, %s, env) OK]\n",
+               icb->name, cb->name);
+       fflush(stderr);
+       return TRUE;
+ }
```

```
+
+ /* Return TRUE if cb conforms structurally to the interface icb */
+
+ static bool_t
+ ImplementsInterfaceStructurally(ClassClass *cb, ClassClass *icb, ExecEnv *ee)
+ {
+     int i;
+     cp_item_type *constant_pool = cbConstantPool(icb);
+
+     /* check if cb provides all methods of icb */
+     fprintf(stderr, "[Methods in %s]\n", classname(icb));
+     for (i = 0; i < (int)(icb->methods_count); i++) {
+         fprintf(stderr, "    [%s %s",
+                 cbMethods(icb)[i].fb.name,
+                 cbMethods(icb)[i].fb.signature);
+         if (!HasPublicMethod(cb,
+                              cbMethods(icb)[i].fb.name,
+                              cbMethods(icb)[i].fb.signature)) {
+             fprintf(stderr, " not found!]\n");
+             fflush(stderr);
+             return FALSE;
+         }
+         fprintf(stderr, " found]\n");
+     }
+     fflush(stderr);
+
+     /* check if cb implements structurally all of the interfaces that
+      * icb implements
+      */
+     for (i = 0; i < (int)(icb->implements_count); i++) {
+       int interface_index = cbImplements(icb)[i];
+       ClassClass *sub_cb;
+       if (!ResolveClassConstantFromClass(icb, interface_index, ee,
+                                          1 << CONSTANT_Class))
+           return FALSE;
+       sub_cb = (ClassClass *)(constant_pool[interface_index].p);
+       if (!ImplementsInterfaceStructurally(cb, sub_cb, ee))
+           return FALSE;
+     }
+     fprintf(stderr, "[ImplementsInterfaceStructurally(%s, %s, env)] OK\n",
+             cb->name, icb->name);
+     fflush(stderr);
+     return TRUE;
+ }
+
+ /* Return TRUE if cb has method nm with signature sig */
+
+ static bool_t
+ HasPublicMethod(ClassClass *cb, char* nm, char* sig)
+ {
+     int i;
+     for (i = 0; i < (int)(cb->methods_count); i++)
+         if (strcmp(nm, cbMethods(cb)[i].fb.name) == 0 &&
+             strcmp(sig, cbMethods(cb)[i].fb.signature) == 0)
+             return TRUE;
+     /* constructors are not inherited */
+     if (strcmp(nm, "<init>") == 0)
+         return FALSE;
+     if (cbSuperclass(cb) == NULL)
```

```
+          return FALSE;
+      return HasPublicMethod(unhand(cbSuperclass(cb)), nm, sig);
  }

  bool_t array_is_instance_of_array_type(JHandle * h, ClassClass *cb, ExecEnv *ee)
```

C Code Changes for Type-Safe Arrays

In the Java compiler, the method `Environment.implicitCast` checks whether two types conform to each other. This method originally allowed unsound conformance between array types as long as the element type of the source array was assignable to the element type of the target array. By contrast, the modification for type-safe arrays proposed in Section 6 requires identical element types for array conformance. The following code change implements this modification.

```
*** src/share/sun/sun/tools/java/Environment.java.orig  Wed May  8 17:47:09 1996
--- src/share/sun/sun/tools/java/Environment.java       Sun Sep 29 03:35:11 1996
***************
*** 241,247 ****
                } while (from.isType(TC_ARRAY) && to.isType(TC_ARRAY));
                if (  from.inMask(TM_ARRAY|TM_CLASS)
                      && to.inMask(TM_ARRAY|TM_CLASS)) {
!                   return isMoreSpecific(from, to);
                } else {
                    return (from.getTypeCode() == to.getTypeCode());
                }
--- 241,247 ----
                } while (from.isType(TC_ARRAY) && to.isType(TC_ARRAY));
                if (  from.inMask(TM_ARRAY|TM_CLASS)
                      && to.inMask(TM_ARRAY|TM_CLASS)) {
!                   return from.equals(to);
                } else {
                    return (from.getTypeCode() == to.getTypeCode());
                }
```