Java and C++
A critical comparison

Robert C. Martin

March 9, 1997

Software engineers are being forced to become multi-lingual. Any of you who have had the pleasure of creating a web site know this. Reading this article will be something like that. I freely swap between C++ and Java throughout. Those of you who are not familiar with Java need not fear, you will find the syntax to be very similar to C++. However, for serious readers, a Java reference guide might be a good thing to have nearby. Fortunately, the entire API for the current release of Java is on-line at `www.javasoft.com/products/JDK/CurrentRelease/api/`.

I want to make it very clear this is not a diatribe against one language or another. I happen to like both Java and C++ quite a bit. I have been a long time programmer in C++, and have just begun to program in Java. I find programming in Java to be a joy. But then, I find programming in any language to be a joy, even JCL `;-)`.

This paper is simply a discussion of the differences in the two languages. I will not be commenting heavily upon deficiencies in C++. These are already very well documented (see Ian Joyner's famous critique of C++. For a copy write to: ian@syacus.acus.oz.au) I will, however, be commenting about both the good and bad points that I perceive in Java. When I have good things to say, this should not be taken as a recommendation of Java. By the same token, when I have bad things to say, this should not be taken as a admonition against the use of Java. In both cases, it is just me venting my opinion. Nothing more.

Up front, I'll say that I am looking forward to writing lots of neat Java applications and applets. But that I am not going to give up C++ any time soon either.

Multiple Inheritance

The designers of Java avoided multiple inheritance. Replacing it is multiple conformance to *interfaces*. In Java, there is a structure called an "Interface". A Java interface is almost identical to a C++ class that has nothing but pure virtual functions. In Java, you cannot inherit from more than one base class; even if the base classes have nothing but abstract methods (pure virtual functions). However you *can* “implement” more than one “interface”; which amounts to the same thing.

For example. In Java you can create the interface for a Stack as follows:

```
public interface Stack
{
  public void Push(Object o);
  public Object Pop();
};
```

This structure is roughly identical to the following C++ code:

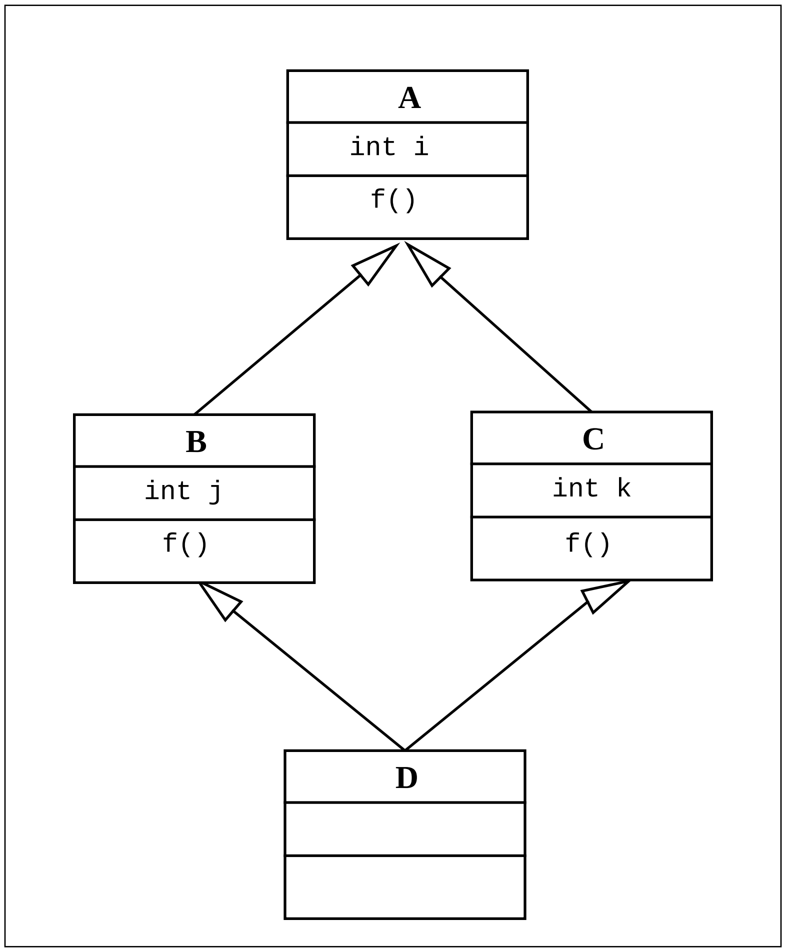
```
class Stack
{
  public:
    virtual void Push(Object&) = 0;
    virtual Object& Pop() = 0;
};
```

However, a Java interface is *not* a class. The functions declared within a Java interface cannot be implemented within that interface. Moreover, a Java interface cannot have any member variables.

Because interfaces cannot have function implementations or data members, multiple implementation of interfaces does not lead to the problems that caused “virtual” inheritance to be added to C++. That is, in Java there is no need for virtual inheritance since it is impossible to inherit the same member variable from more than one path.

In C++, these such situations arise from the so-called *deadly diamond of death*. See Figure 1.

Figure 1: Deadly Diamond of Death



This UML 9.1 diagram shows four classes arranged in the diamond structure that creates the need for virtual inheritance. Both of the classes B and C inherit from class A. D multiply inherits from both B and C. Two problem arise from this. First, which implementation of the ’f’ function does D inherit? Should it inherit **`B::f()`** or **`C::f()`**. In C++ the answer turns out to be *neither*. **`D::f()`** must be declared and implemented. This eliminates the ambiguity, and certainly this simple rule could have been adopted in Java.

The second problem, however, is quite a bit more complicated. The class A has a member variable named **`i`**. Both classes B and C inherit this member variable. Since D inherits from both B and C we face an ambiguity. On the one hand, we might want **`B::i`** and **`C::i`** to be separate variables in D; thus creating two copies of A in D. On the other hand we might want a single copy of A in D so that only **`A::i`** exists in D.

In C++ we choose between these two options by varying the *kind* if inheritance we use. If we want two copies of A in D, then we use regular inheritance from B to A and from C to A. If, however, we want only one copy of A in D, then we use *virtual* inheritance from B to A and from C to A.

Virtual inheritance is a complex feature and creates problems for compiler implementers and application programmers alike. The designers of Java did not want to have to deal with this issue. So by disallowing multiple inheritance of classes, and only allowing the multiple implementation of interfaces, they set the language up such that *the deadly diamond of death* (DDD) cannot be created.

This was probably a good trade-off. In all likelihood it has simplified the language appreciably. However, it has left a problem. It prevents us from inheriting implementation from more than one class in cases where the DDD does not appear. This is unfortunate since it is often the case that we want to inherit from more than one base class which has functions and data.

For example, consider the following C++ program that uses the OBSERVER pattern[1].

```
class Clock
{
   public:
     virtual void Tick() // called once a second to maintain the time.
     {itsTime++;}
   private:
     Time itsTime;
};
```

We have a Clock class that understands the mathematics of time, and which receives a ’Tick’ event once per second. We would like to create a version of this class that is *observed*. That is, we want to be able to inform other classes when the state of the Clock class has changed. For this, we use the OBSERVER pattern.

The OBSERVER pattern involves two base classes. One is called **`Observer`**; it is an abstract base class with one pure virtual function: **`Update()`**. Any class that wants to be informed when the state of an observed class changes must inherit from **`Observer`**.

```
class Observer
```

1. Design Patterns: Elements of Reusable Object Oriented Software, Gamma, et. al., Addison Wesley, 1995

```
{
   public:
     virtual void Update() = 0;
};
```

The class that forms the other half of the OBSERVER pattern is named **`Subject`**. The **`Subject`** class maintains a list of **`Observer`** instances and has two concrete functions. The first function, **`Register`**, allows instances of **`Observer`** to be added to the list. The second function, **`Notify`**, is called when there is a state change that needs reporting. This function calls **`Update`** on all the registered **`Observer`** instances.

```
class Subject
{
   public:
     void Register(Objserver& o)
     {itsObservers.Add(&o);}
     void Notify()
     {
       for (vector<Observer*>::iterator i = itsObservers.begin(); i; i++)
         (*i)->Update();
     }
   private:
     vector<Observer*> itsObservers;
};
```

If you want to create a class that observes state changes in another class, the observing class must inherit from Observer. The observed class must inherit from Subject. The observer class must be registered with the subject class. And the subject class must call Notify when its state is changed.

Back to the clock problem. Not all applications are interested in observing our **`Clock`** class; so we don't want to have **`Clock`** inheriting from **`Subject`**. If we did, then every application would be forced to include the **`Subject`** class, even though it didn't need it.

To prevent **`Subject`** from existing in every application that uses **`Clock`**, we employ multiple inheritance. We create an new class called **`ObservedClock`** that inherits from both **`Subject`** and **`Clock`**.

```
class ObservedClock : public Clock, public Subject
{
    public:
      virtual void Tick()
      {Clock::Tick(); Notify();}
};
```

This use of multiple inheritance in C++ yields a simple and elegant solution to our problem. **`Clock`** is still free to be used in applications that don't need **`Subject`**. Those applications that want to observe clock objects can use **`ObservedClock`**.

In Java one cannot do this. Rather one must do the following:

```
public class Clock
{
   public void Tick() {itsTime.Add(1);}
   private Time itsTime;
}

public interface Observer
{
    public void Update();
}

public interface Subject
{
   public void Register(Observer o);
   public void Notify();
}
```

The **`Clock`** class is not surprising. The **`Observer`** is very similar to the C++ version. In Java it is an interface, whereas in C++ it was a class with nothing but pure virtual functions. However the **`Subject`** class is quite different. In Java it too is an interface. But in C++ it was a concrete class with member variables and implemented functions.   The implementation of the **`Subject`** interface is accomplished in a class named **`SubjectImpl`**

```
public class SubjectImpl implements Subject
{
   public void Register(Observer o)
   {itsObservers.addElement(o);   }

   public void Notify()
   {
     Enumeration i = itsObservers.elements();
     while(i.hasMoreElements())
     {
       Observer o = (Observer) (i.nextElement());
       o.Update();
     }
   }
   private Vector itsObservers;
};
```

We can see that this is very similar to the C++ version. However Java does not allow multiple inheritance like C++ does, so we have a problem We cannot inherit from both **`Clock`** and **`SubjectImpl`**. Instead, we must inherit from **`Clock`** and implement the **`Subject`** interface. This amounts to multiple inheritance of interface since the **`ObservedClock`** class has the union of the **`Clock`** and **`Subject`** interface

```
class ObservedClock extends Clock implements Subject
{
   public void Tick() {super.Tick(); Notify();}
   public void Notify() {itsSubjectImpl.Notify();}
   public void Register(Observer o) {itsSubjectImpl.Register(o);}
   private SubjectImpl itsSubjectImpl;
}
```

Notice what we had to do. We had to implement the **`Subject`** interface by creating a member that points to a **`SubjectImpl`**. We then had to delegate every **`Subject`** interface to that contained **`SubjectImpl`**.

This use of aggregation instead of multiple inheritance is inconvenient to say the least, especially when it must be used with a pattern that is a prevalent as OBSERVER.

This leads me to believe that the Java language will either need a more complete form of multiple inheritance, *or* it will need some new syntax that allows the compiler to automatically delegate. e.g.

```
class ObservedClock extends Clock delegatesTo SubjectImpl....
```

This proposed **`delegatesTo`** syntax would automatically cause **`ObservedClock`** to implement the interface of **`SubjectImpl`** as well as automatically forwarding all calls to that interface to an automatically contained instance of **`SubjectImpl`**.

Memory Management

Java uses garbage collection. Garbage collection is a scheme of memory management that automatically frees blocks of memory sometime after all references to that memory have been redirected. For example, consider the following snippet of Java:

```
Clock c = new Clock(); // c refers to the new clock.
// ... use c for awhile.
c = null;  // done with that clock.  System will clean up later.
```

In this example, we create a new **`Clock`** object using the keyword: **`new`**. The new object is referred to by the variable '**`c`**'. Note that '**`c`**' is rather like a reference variable in C++; however in Java it is possible to reassign references. We use the new **`Clock`** object through its reference variable '**`c`**' for awhile. Then, when we are done with it, we redirect '**`c`**' to **`null`**. When the Java runtime system detects that there are no more reference variables referring to the **`Clock`** object, it classifies that object as “garbage”. At some later time, the Java runtime system will clean up that “garbage” returning its memory to the heap.

Garbage collection makes certain kinds of applications much easier to program. The designers of those programs need not worry as much about cleaning up after “dead” memory. As a result, C++ is often criticized for its lack of GC. However, many people have added garbage collectors to C++. Some of these are available as third party products, or as shareware on the net. These collectors are far from perfect, but they can be used when convenient.

The corresponding statement cannot be made for Java. There is *no way* that this humble writer could discover to manage memory manually. Apparently, you cannot write your own memory manager and construct objects within the memory that it controls.

There is a sound reason for this. Any memory management scheme that allows a program to hold pointers or references to unused space allows certain security violations. For example, consider the following program in C++:

```
void f()
```

```
{
  char* p = new char[1000000];
  delete [] p;
  SeachForPasswordsInSeparateThread(p);
  return;
}
```

The last function called: **`SearchForPasswordsInSeparateThread`** returns immediately. However it also starts a new thread that continuously scans the megabyte pointed to by p. Since this megabyte has been returned to the heap already, it will be used by lots of other functions in the system. It is just possible that it might be used to hold a password, or some other security sensitive material. This material is open to examination by any function that holds a dead pointer into the heap.

Thus, any form of manual memory management that involves holding on to dead pointers or references could result in a security breach. In the typical Java environment, security is a serious concern. Java applets are often downloaded and run in web browsers. The users may have no idea what applets are running because of their browsing activities. If manual memory management were allowed, it might be possible for unscrupulous people to put up web pages that contained insecure applets. These applets would be downloaded into the systems of unsuspecting users who happened to browse that page. Once downloaded those applets could then transmit private information back to the author of the web page.

Is the lack of manual memory management in Java a problem? In most cases no. However, the lack of manual memory management makes Java a difficult language to use in applications that have hard real-time constraints. The problem is that it is very difficult to predict when the garbage collector will run. When it does run, it can use of significant amounts of CPU time. Consider the following Java snippet:

```
public class RealTime
{
  public void Do()  // must complete in 500µs
  {
    Clock c = new Clock;  // might collect!
    // diddle with clock for 100µs
  }
}
```

Here we have a function that must complete in 500µs. This is a typical constraint in a hard real time system. Those functions that call **`RealTime.Do()`** depend on the fact that it will take no longer than 500µs to execute. In many cases, this is a hard constraint that must be met every time **`Do()`** is called. However, every once in awhile the Java runtime system will be unable to allocate the new **`Clock`** object until it has collected garbage and returned unused memory to the heap. There is no telling how long this collection will take. And so, under these circumstances, **`RealTime.Do()`** cannot meet its real-time constraints.

One cannot simply follow James Gosling's and Ken Arnold’s advice when they say: “[Garbage Collection] can interfere with time critical applications. You should design systems to be judicious in the number of objects they create.”[1] Instead, in time critical applications, you must design ways in which the memory you need can be made available without the possibility of incurring a garbage collection. One simple strategy is as follows:

```
public class ClockPool
{
  ClockPool()
  {
    for (int i=0; i<10; i++)
      itsClocks.push(new Clock());
  }
  public Clock GetClock()
  {return (Clock)(itsClocks.pop());}

  public void FreeClock(Clock c) {itsClocks.push(c);}

  private Stack itsClocks;
}
```

Here we have created a simple memory manager. It manages **`Clock`** objects. If you need a **`Clock`** object you simply call **`GetClock()`**. When you are done with it you call **`FreeClock()`**. It creates 10 **`Clock`** objects for this purpose and holds them in reserve. If more than 10 **`Clock`** objects are needed, an exception will be thrown by the **`Stack`** when it underflows.

One might think that this solves the problem. Now the RealTime class could be written as follows:

```
public class RealTime
{
  public void Do(ClockPool p)  // must complete in 500µs
  {
    Clock c = p.GetClock();
    // diddle with clock for 100µs
    p.FreeClock(c);
  }
}
```

However, we are fiddling with the **`java.util.Stack`** class within **`ClockPool`**! It is possible that its activities might force a garbage collection? This train of thought will quickly convince us that using any of the Java standard library within time critical applications can lead to garbage collection.

It should be clear that using Java in hard real-time applications presents some interesting challenges. Caveat Emptor.

Finalize

The **`finalize`** function in Java, roughly corresponds to the destructor in C++. When an object is collected by the garbage collector, its **`finalize`** method is called. This allows objects to clean up after themselves. However, it should be noted that in most cases **`finalize`** is not a

1. *The Java Programming Language*, Ken Arnold, James Gosling, Addison Wesley, 1996, p. 46

good place to release resources held by the object. It may be a very long time before such objects get collected by the garbage collector. Thus, any resources they release in **finalize** may be held for a very long time.

Upon normal exit of a Java application, the garbage collector sweeps up all the uncollected objects, and their **finalize** methods are called at that time. (This implies that exiting can take a bit of time). Thus, barring abnormal exits, all **finalize** methods will eventually be called.

However, there is a convention that one must adopt:

```
class D extends B
{
  protected void finalize() throws Throwable
  {
     super.finalize(); // finalize B.
     // now take care of finalizing D.
  }
}
```

The **finalize** of the derived class must *explicitly* call the **finalize** of the base class.   If you forget to do this, then base class **finalize** functions simply don't get called. Gosling and Arnold recommend: "Train your fingers so that you always do so in any **finalize** method you write."[1] Unfortunately, this is very error prone. In my opinion such "training" belongs to the compiler. The calling of base class **finalize** methods should have been taken care of by the compiler in the manner of destructors in C++.

toString()

Any class that has the **toString** method implemented as follows can be used in some special contexts that expect a **String**.

```
class MyClass
{
   public String toString()
   {
     return ("MyClass as a String");
   }
}

class Test
{
  public static void main(String[] args)
  {
    MyClass o = new MyClass();
    System.out.println("I just created a Myclass: " + o);
  }
}
```

1. *ibid*, p47.

The use of ’`o`’ as an argument of the ’`+`’ operator in a `String` context automatically invokes the `MyClass.toString()` method. The returned `String` is then used as the argument of the ’`+`’ operator to concatenate the strings. Thus: "`I just created a Myclass: Myclass as a String`" is printed.

This automatic use of `toString()` seems to be an immature version of the automatic conversion system of C++. This feature (among others) makes the `String` class something more special than any other class.

I think it would be wise for the Java designers to work on a generic conversion system. (e.g. a to<xxx> method template)

Exceptions and '`finally`'

I am very pleased with the exception mechanism in Java. Although modeled after the C++ mechanism, it avoids some of C++s more severe problems by using the '`finally`' clause.

In C++, when an exception leaves the scope of a function, all objects that are allocated on the stack are reclaimed, and their destructors are called. Thus, if you want to free a resource or otherwise clean something up when an exception passes by, you must put that code in the destructor of an object that was allocated on the stack. For example:

```
template <class T>
class Deallocator
{
  public:
    Deallocator(T* o) : itsObject(o) {}
    ~Deallocator() {delete itsObject;}
  private:
    T* itsObject;
};

void f() throw (int)
{
  Deallocator<Clock> dc = new Clock;
  //....things happen, exceptions may be thrown.
}
```

In this example, the `Deallocator<Clock>` object is responsible for deleting the instance of `Clock` that was allocated on the heap. Whenever ’`dc`’ goes out of scope, either because ’`f`’ returns, or because an exception is thrown, the destructor for ’`dc`’ will be called and the `Clock` instance will be returned to the heap.

This is artificial, error prone, and inconvenient. Moreover there are some really nasty issues having to do with throwing exceptions from constructors and destructors that make exceptions in C++ a difficult feature to use well. For more details on the traps and pitfalls of C++ exceptions, I recommend that you read the excellent series of Leading Edge columns by Jack Reeves that have appeared over the past year in C++ Report.

Now I am not going to claim that Java has fixed all these things. However, I like their solution better than the C++ solution. Every **`try`** block can have a '**`finally`**' clause. Any time such a block is exited, regardless of the reason for that exit (e.g. execution could proceed out of the block, or an exception could pass through it) the code in the **`finally`** clause is executed.

```
public class Thing
{
  public void Exclusive()
  {
    itsSemphore.Acquire();
    try
    {
      // Code that executes while semaphore is acquired.
      // exceptions may be thrown.
    }
    finally
    {
      itsSemaphore.Release();
    }
  }
  private Semaphore itsSemaphore;
};
```

The above Java snippet shows an example of the **`finally`** clause. The method **`Exclusive`** sets a semaphore and then continues to execute. The **`finally`** clause frees the semaphore either when the **`try`** block exists, or if an exception is thrown.

In many ways, this scheme seems superior to the C++ mechanism. Cleanup code can be directly specified in the **`finally`** clause rather than artificially put into some destructor. Also, the cleanup code can be kept in the same scope as the variables being cleaned up. In my opinion, this often makes Java exceptions easier to use than C++ exceptions. The main downside to the Java approach is that it forces the application programmer to (1) know about the release protocol for every resource allocated in a block and to (2) explicitly handle all cleanup operations in the finally block. Nevertheless, I think the C++ community ought to take a good hard look at the Java solution.

Threads

An in-depth discussion of Java threads is beyond the scope of this article. For more information on Java threads I recommend that you read Doug Lea's new book[1] [Editors note: the article "Experiences Converting C++ Communication Software Frameworks to Java" by Jain and Schmidt in this issue also discusses Java threading in more detail.] I will return to this topic in a subsequent article. Suffice it to say that I am overjoyed with the way that threads have been implemented in Java. The implementation is minimal and elegant. The simple way that methods can be protected from concurrent update, the equally simple semaphore and critical code mechanisms, the very easy way of creating a rendezvous between two threads, all combine to make this a good language feature.

1. Doug Lea, *Concurrent Programming in Java: Design Principles and Patterns.* Addison Wesley, 1997

Operator Overloading

While writing code in Java I have to say that I miss being able to overload operators as in C++. This is not a critical issue, but I am disappointed.

Templates

Templates are a wonderful feature of C++. The fact that Java does not have them is of some concern to me. In Java, one cannot create a type-safe container. All containers in Java can hold any kind of object. This can lead to some ugly problems.

Mitigating these problems is the fact that all casts in Java *are* type safe. That is, Java casts are roughly equivalent to **`dynamic_cast`** of references in C++. An incorrect cast results in an exception being thrown. Since all objects coming out of a Java container must be downcast, and since such casts are relatively safe, the need for type safe containers is somewhat lessened.

```
   public void Notify()
   {
     Enumeration i = itsObservers.elements();
     while(i.hasMoreElements())
     {
       Observer o = (Observer) (i.nextElement());
       // the cast above will throw an exception if
       // nextElement returns something other than
       // an Observer.
       o.Update();
     }
   }
```

However, type safe containers are not the only good thing about templates. Templates in C++ are a very nice way of achieving static polymorphism. Consider the following C++ code:

```
template <class Modem>
void SendString(char* s, Modem& m)
{
    m.Dial("5551212"); // call my system.
    if (m.IsConnected())
    {
       while (*s)
        (m.Send(*s++);)
       m.Hangup();
    }
    else // not connected
      cerr << "could not connect" << endl;
};
```

Here we see a C++ template function that employs static polymorphism. The **`SendString`** function can work with any class that has the methods: **`Dial`**, **`IsConnected`**, **`Send`**, and **`Hangup`**. Although it is more typical in both C++ and Java to gain this kind of polymorphism using abstract base classes; there are some distinct advantages to using templates. For example, there is no virtual overhead. i.e. no extra time or memory is spent managing the dynamic binding of normal C++ virtual functions.

For these reasons, I think that templates should be considered for later releases of Java.

Labeled Breaks and Continues

OK, it's time for a minor rant. The language designers pride themselves on creating a language that does not have a **`goto`** statement.[1] Yet, wonder of wonders, they added *labeled* **`break`** and **`continue`**. This is a sore point of mine. The tenets of *structured programming* do not disallow the use of **`goto`**. Rather they disallow its use in any context that destroys the single-entry/single-exit (SE/SE) paradigm. For example, using **`goto`** to create the equivalent of a **`for`** loop or a **`while`** loop does not violate structured programming. Of course in a language like C++ or Java it would be a silly thing to do since those statements already exist.

The SE/SE paradigm says that every block of code should have a single entry point and a single exit point. There should be no way to enter such a block in the middle, and no way to exit such a block from the middle. Entry is at the top, and exit is at the bottom.

The use of **`break`** and **`continue`** in C, C++, or Java constitute a minor violation of SE/SE. We use them to transfer control out of the middle of a loop. The use of *labeled* break and *labeled* continue are a much more serious violation. These can be used to exit deeply nested blocks from their middles. Indeed, some of the enclosing blocks may not know that they are being exited and may be written to assume that they are not. This can lead to errors that are very hard to identify.

```
outer:
for (int i=0; i<99; i++)
{
  for (int j=0; j<99; j++)
  {
    if (AreWeDone(i,j) == true)
      break outer; // get all the way out.
    else
      DoSomethingCool(i,j);
  }
  DoSomethingElse();
}
```

The code above shows a typical scenario. There are two nested loops, one iterating on i, and the other on j. In the inner loop we test to see if some condition having to do with i and j will allow us to terminate the loop early. If so, we use a labeled **`break`** to exit both loops.

This example also demonstrates the danger of violating SE/SE. The outer loop expects to call **`DoSomethingElse`** every time the inner loop is finished. Yet the labeled break thwarts this. To correct this, we could use **`finally`** as follows:

```
outer:
for (int i=0; i<99; i++)
{
```

1. *ibid*, p. 131.

```
  try
  {
    for (int j=0; j<99; j++)
    {
      if (AreWeDone(i,j) == true)
        break outer; // get all the way out.
      else
        DoSomethingCool(i,j);
    }
  }
  finally
  {
    DoSomethingElse();
  }
}
```

Or we could put the actual loop conditions in the inner **`for`** statement.

```
for (int i=0; i<99; i++)
{
  for (int j=0; j<99 && AreWeDone(i,j) == false; j++)
  {
    DoSomethingCool(i,j);
  }
  DoSomethingElse();
}
```

I make it one of my personal rules to avoid the use of **`break`** and **`continue`** as a way of managing loops. So I will probably not make much use of labeled **`break`** and labeled **`continue`**.

Conclusion

Java is a fun language. C++ programmers should have a relatively easy time learning it, and will find that they enjoy using it. I have noted a few problems with the language in the above paper, but I don't consider these to be very critical issues. Moreover, I don't know of any language that doesn't have such problems. Language design always involves trade-offs that displease someone.

I look forward to writing lots of interesting Java applications. I also look forward to watching how the language evolves from this point onward. I expect to see some changes in the next few years.