VICTORIA UNIVERSITY OF WELLINGTON
*Te Whare Wananga o te Upoko o te Ika a Maui*

School of Mathematical and Computing Sciences
Computer Science

PO Box 600
Wellington
New Zealand

Tel: +64 4 471 5341, Fax: +64 4 495 5045
Email: Tech.Reports@mcs.vuw.ac.nz
Http://www.mcs.vuw.ac.nz/comp/Publications

# Surprises in Teaching CS1 with Java

Peter Andreae, Robert Biddle, Gill Dobbie, Amy Gale,
Linton Miller, Ewan Tempero

Technical Report CS-TR-98/9
September 1998

**Abstract**

We describe our experiences teaching our CS1 course with Java. Java has a number of features that complicate using it to teach introductory programming. Although we designed our course to deal with these features, there were some surprises in how the course worked out. We discuss the underlying cause of these surprises.

# Surprises in Teaching CS1 with Java

Peter Andreae, Robert Biddle, Gill Dobbie, Amy Gale, Linton Miller, Ewan Tempero
School of Mathematical and Computing Sciences
Victoria University of Wellington
New Zealand
+64 4 471 5341
{pondy,robert,gill,amyl,lmiller,ewan}@mcs.vuw.ac.nz

January 7, 1999

## ABSTRACT

**We describe our experiences teaching our CS1 course with Java. Java has a number of features that complicate using it to teach introductory programming. Although we designed our course to deal with these features, there were some surprises in how the course worked out. We discuss the underlying cause of these surprises.**

## 1 INTRODUCTION

Java [1] has become a popular choice as the language to teach in introductory programming courses, but it is not an easy language to teach with. It was designed as a language for developing applications in the real world, not as a teaching language. As a consequence, it has features with complex behavior that are difficult to avoid even in the simplest program. This complex behavior can create difficulties for students with no previous programming experience, since they must deal with the complexity while they are still coming to terms with more fundamental concepts. Thus, any course that uses Java must be carefully designed with this in mind.

Last year we taught our CS1 course in Java for the first time. We have been made aware of the difficulties Java imposed on such a course [2], and took care to deal with them. However, while our course was generally successful, there were some surprises — some happy and some unhappy. We believe that the underlying causes of these surprises give a good indication of the problems that students *really* had, as opposed to our (and others') guesses as to what the problems could be. We present our analysis of these underlying causes in the hope that others might benefit.

Our paper is organized as follows. We begin by presenting the context in which our course design took place. We then discuss the surprises that arose in the early part of the course, which covered the introduction of objects, definition and use of classes, and event-driven input. Section 4 discusses the problems caused by the distinction between primitive and object types in Java, and section 5 discusses what was the biggest surprise of the course — the unintended downplaying of types. Finally, we present our conclusions.

## 2 CONTEXT

Our CS1 course is a first course on programming that emphasizes good design of computer programs based on object-oriented programming. We do not require previous experience with computers but it is helpful. The course is 12 weeks long with 3 lectures a week. The students do weekly programming assignments that involve building or completing several programs relevant to the topics just covered in lectures.

We changed the programming language in our CS1 course to Java from Pascal. The decision to change was in part driven by the desire to use a language that was portable, better supported the principles of design that we wanted to teach, and was more useful to our students. In particular, we wanted better support for event driven input, encapsulation, object-oriented design, and generic container types.

The design of the course was impacted by the fact that we wanted to teach objects and classes early. Also, because most of the applications that students are familiar with, (e.g. games, paint programs, spreadsheets, and word processors) are event driven and have graphical output, we wanted them to build such applications themselves. In order to do this without having to introduce some of the more complex aspects of Java, we created a library of classes that provided an easy to use interface, and allowed students to write interesting applications from their first programming assignment.

The course consisted of three parts:

- The first part introduced the basic elements of programming in Java, including calling methods on objects, using primitive types (including strings), instantiating objects, defining classes and methods, selection and repetition, and dealing with input and output (but not files).
- The second part addressed simple algorithm design in the context of dealing with strings, text files, and arrays. It also introduced recursion.
- The third part addressed more complex design with multiple classes. In particular, it addressed composition, container classes, and genericity. It also introduced inheritance and some software engineering principles, such as system testing.

We used the textbook "Java Software Solutions" by Lewis and Loftus [3] because it followed an approach compatible with our own.

In the laboratories we had Macintoshes, and many of the students had PCs at home so we chose an environment that ran on both. MetroWerks CodeWarrior was the best solution at the time. Although CodeWarrior was functional, we were disappointed that it was missing at least two very valuable features: an easy-to-use debugging environment (with stepping, breakpoints, etc.) and automatic indentation of code.

# 3 PROGRAMMING WITH OBJECTS AND EVENTS

Teaching how to program with objects and event driven input involved more complex concepts than the previous version of the course, and so we used a different teaching sequence. There were several surprises in how the students coped with these differences.

## 3.1 Object instantiation

We had expected that the students would have difficulty with the notion of instantiating objects and calling methods on them. The set of concepts involved in this notion is significantly larger than the set of concepts needed for procedure invocation in Pascal, and some of the concepts, particularly dynamic allocation of objects and the use of references, are very subtle. Consistent with the "objects early" decision, this notion was introduced in the 4th lecture, and the second programming assignment involved writing a program that instantiated and called methods on objects from a class that had been described in lectures. To our surprise, the students had little difficulty with this assignment, and in later assignments, they appeared to continue to have little difficulty with instantiating and calling methods on objects. They seemed to be coping with the larger set of more complex concepts just as easily as previous students had coped with procedure invocation.

Although we would like to credit this all to our lecture presentations, we speculate that the students did not, in fact, grasp all the concepts involved, but were able to cope with a much shallower understanding and the use of "boilerplate" code, taken from the example programs given in lectures. The students' later difficulties with the distinctions between objects and non-objects (section 4) provides the basis for this speculation.

## 3.2 Method and Parameter declarations

Immediately after showing how to instantiate objects and call methods on them, we addressed the declaration of classes and methods. We did not introduce explicit constructors at this stage, so the class definitions consisted just of field and method declarations. We were surprised that so many students had difficulty with method definitions, particularly the distinction between arguments to a method call and parameters of a method declaration. We had hoped that the simpler parameter mechanism of Java (no VAR parameters) and the better motivation for method definitions provided by the class structure would make it easier for students to distinguish arguments from formal parameters than in the previous version of the course.

Dealing with the abstraction required for the "definition of the general case", in contrast to the more concrete "calling of a particular case", seems to be a difficult concept for many students to grasp. This difficulty is just as great in Java as it was in Pascal.

## 3.3 Event-driven input

As soon as the students had the basic blocks of class and method declarations and the conditional statement, we introduced programs with event driven input (buttons, text-fields, mouse, etc). This was at the end of the third week of the course. We anticipated that students would have considerable difficulties with event driven programs because they are conceptually more complex than the programs we had used in the Pascal version of the course. Furthermore the mechanism by which the Java machine executes an event driven program is more complex than the mechanism for a non-event driven program.

We were surprised that the students seemed to have very little difficulty with writing programs using this event-driven input model. Anecdotal comments from students during the fifth week suggested that they had far less difficulty with the programs that introduced event-driven input than they had had with the programs that introduced class and method declarations.

We think that there were several contributions to the smooth introduction of event-driven input:

1. The careful design of the event-handling model and the

library to reduce complexities and avoid potential confusions.

2. The presentation of a computer program from the first lecture as a collection of specifications of responses to requests. The first programs responded to only one request — "start"; event driven programs merely expanded the set of requests that could be responded to.

3. Consistent use of a single program design for handling event driven input that enabled the students to construct their own programs using the lecture examples as boilerplate code. This enabled the weaker students to build working programs and learn about other aspects of programming even when they did not understand the event-driven model well.

## 4 OBJECTS VS NON-OBJECTS

We knew from the beginning that the different semantics for object and primitive (non-object) types in Java were going to require significant teaching effort. Despite our planning, we were surprised at just how difficult this was to teach, and we believe that by the end of the CS1 course a reasonable proportion of students still did not have a good understanding of the distinction between primitive and object types and between an object reference and the object it refers to.

In our CS1 course, the underlying pointer structure used for object types was explained using an "object ID" metaphor. Object variables were presented as containing "IDs" of objects that were physically located elsewhere, while primitive variables were presented simply as boxes where values could be stored. Students were taught that dereferencing occurred when they "looked at" part of an object, or needed to "do something" to an object.

We do not believe we were entirely successful in teaching students how to determine when they were accessing and manipulating the object reference and when they were accessing and manipulating the object itself. This manifested itself in problems with parameter passing, in difficulties in determining whether to compare using == or `equals()` and in confusion over the behavior of Strings.

### 4.1 Syntactic Markers for Dereferencing

Through analyzing the problems we had in this area, we have come to believe that essential in making the distinction between references and the objects they refer to is the presence of an explicit syntactic marker. This marker serves as a mnemonic aid, reminding the programmer of what mode they are in.

In other teaching languages these markers have been large and explicit. In Java, there is such a marker: the dot ".". Unfortunately, we didn't emphasis the correspondence of the syntactic marker (".") with the concept (dereference) as much as we should have. We therefore did not provide our students with a marker as a clue to when dereferencing occurred, and this may have been behind some of our surprises (although it might not have helped with Strings). In particular, our standard style was to name data fields directly inside method definitions instead of always accessing via `this`, so the dot was not always a feature of dereferencing in the way we taught the course.

### 4.2 Passing Objects vs Primitive Types

When you pass an object to a method, any changes made to that object persist after the method has been exited. When you pass a primitive data value the same is not true because primitives are passed by value. The object pointer is also passed by value, but this has the effect of appearing to pass the object itself by reference.

In the past we had taught about passing by value and passing by reference in CS1, and students, by and large, had grasped the distinction by about halfway through the course. We were therefore fairly confident in our ability to teach the distinction between parameters whose value might change as the result of a method call and parameters whose value will not change. Despite this, we were much less successful than we had been in the past.

One of the big differences is the existence in Pascal of the keyword `VAR`, distinguishing cases where a parameter is passed by reference from cases when it is passed by value. No such visual clue is present in Java, programmers are simply supposed to remember the different behavior of primitives and objects.

For experienced programmers this will generally not cause too much difficulty. Our CS1 students are most definitely not experienced programmers. Without the existence of a visual reminder that an object will be permanently affected (and a primitive won't) many fall into difficulties very easily.

### 4.3 Testing Equality

We took great pains to emphasize to students that == was only to be used to compare primitives, that all comparisons between objects were to use the `equals()` method.

The key word here is "between". Many students had difficulty reconciling this rule with the mechanism for testing if an object is `null`, which is to use == to compare the object to `null` directly.

We had not seen this degree of difficulty in the past, and so were not fully prepared for the existence of these misunderstandings. That we had not seen it in CS1 when we used Pascal is not surprising, since we had never before taught about pointers in CS1. That we had not seen it to the same degree in CS2 after the introduction of pointers can perhaps be ascribed to the presence in Pascal of explicit syntactic markers distinguishing "the pointer" from "what is being pointed to".

Again, we can argue that the dot is the dereference, and indeed in in many cases focusing on the dot as a syntactic marker will help. This is not always the case. In particular, the String object type causes a lot of problems.

## 4.4 The String type

Strings, while objects types, have specific support in the language. In particular, whenever a program creates two literals with the same value, in fact only one String object is created with two references to that object. This has the consequence that == can be, in some situations, sensibly used to determine if two String objects have the same value.

This specialized behavior of == for Strings undermines the distinction between primitive and object types. We emphasize to students that == should only be used to compare primitive types, but if students use == to compare the values of Strings in error, they will nevertheless get the behavior they expect, which reinforces the wrong model about object vs primitive types.

# 5 TYPES

One of the biggest surprises we had from the course was something we did not realize until after the course itself was over. It was only in our review session that we realized that we had not stressed the concept of "type" as much as would have liked, nor indeed as much as we used to do when using Pascal. This seemed odd, because we regarded the user-defined types as better supported in OO languages, such as Java, than in Pascal. It is possible that our confidence in this support simply resulted in our lack of attention to the issue. Moreover, we now realize that there are several ways in which Java supports the type concept less well than Pascal.

## 5.1 Classes and Primitive Types

One feature of Java that necessarily appeared early, and was pervasive throughout, was the "class". A class name can be used as a type to declare object variables, and type-checking is an important topic when introducing many principles of program design. However, our introductory programs typically involved only the class with the "main" method, or other classes with only a single instance. In retrospect, it seems this early familiarity with classes for program structuring may have contributed to misunderstanding when we later promoted the important link between class and type. Java does not distinguish classes being used for program structuring and classes being used as types, and so this distinction is one that we as teachers must explain.

To introduce the concept of type, our experience with other languages leads us to draw parallels between built-in types and user-defined types. We discuss the role of both in declaring and storing data, and how both are used by operations specific to the kind of data that is stored. We did follow this approach in Java, but it worked less well than we hoped.

One factor was that it was easy to become involved in explaining how primitive types and object types differ, and easy to under-emphasize how they are similar. Some differences did not seem problematic, such as use of infix operators versus use of postfix method names. More subtle differences involved semantics of assignment, parameter passing, and comparison, as we discussed in the previous section. These required explanations repeated several times during the course, and may have undermined the concept of type.

Another factor was that the type concept is not strongly emphasized in what Java itself provides. In Pascal, for example, there is the enumerated type to represent simple sets, and this is a useful place to begin teaching about user-defined types. In introductory Java programs we represented simple sets, such as colors or days, with integers or strings. This did not assist understanding of the type concept at all, and the alternative of special classes and static fields seemed far too complex. In Pascal, enumerated types also play a role in the structure of arrays, further assisting emphasis of the type concept. In Java, the array does not provide such assistance. Moreover, the Java array involves aspects of both primitive types (specific syntax) and object types (constructors and null), but isn't really either.

## 5.2 Genericity

A concept Java supports, but Pascal doesn't, is genericity. We were pleased this would allow us to enrich the later part of the course, by introducing generic sets or lists as more reusable components. We were familiar with introducing this topic using C++ and its "template" feature in another course. In that approach, a generic class is a parameterized class, where typically the type of the contained element is the parameter: so a generic set is used as a set of integers, a set of account objects, and so on. However, the Java approach is that a generic class is a class with elements of type Object, so the class may be used to hold any kind of object.

Compared with C++, the Java approach did seem simpler and easier to teach and learn. However, the difference also means that the type concept is less emphasized in Java, for several reasons. Firstly, the generic class can in fact hold a mixed set of objects. Secondly, to extract objects from the set, a "cast" is necessary, which must be explained with great care to avoid the type issue seeming a nuisance. Thirdly, the type checking involved in the Java approach is at run-time, instead of compile-time as in C++, and run-time checking blurs the type concept. Finally, Object is not compatible with primitive types, and so Java generic classes cannot contain primitive type data without the complication of "wrapper" objects.

## 5.3 Encapsulation and Abstraction

One principle of program design that we stressed at several points was encapsulation. We discussed this early, when we explained about the access control labels "private" and "public", and we reviewed it in detail later on, when we began to discuss larger design issues. We emphasized the role of the public interface of a class, and the importance of type checking in creating a program by using components. We also discussed how encapsulation allows the implementation of a class to be changed without affecting any users of the class. In Java, the public interface to a class is obscured by being textually interleaved with the implementation, so care is needed to emphasize the importance of the distinction.

We felt satisfied with our approach at the time, but now realize we could have gone a step further. When we reviewed the course, we realized that in the past we had explicitly introduced and used the "abstract data type" (ADT) concept, and now did not mention the term at all. In Pascal, the lack of support for type encapsulation made stressing this concept very important, and our feeling had been that Java support for class encapsulation made this stress unnecessary. However, we now feel that we should return to the ADT concept. This might have the effect of underlining our discussion of encapsulation, and at the same time increasing our emphasis on the type concept. Furthermore, if we were to increase emphasis on the ADT, we could support this by using Java "interfaces" as a syntactic specification for ADTs.

## 6 CONCLUSIONS

Java is not designed as a teaching language, and so care must be taken when using it to teach introductory programming. Ouerall, we believe the course was successful. Anecdotal feedback suggests the students enjoyed the course, student evaluations of the course were similar to those for the previous version of the course, and performance in programming assignments and exam results was also in line with previous years.

Although succssful, there were surprises — students coped with some parts of the course better than we hoped, but did not cope with other parts as well, even though some of the other parts involved concepts closely related to those that the students appeared to understand. Our analysis of what happened leads us to believe that students did better than we expected in situations where there was good syntactic support for the concept, and poorly when there wasn't. This analysis suggests that, by following coding conventions that better support the concepts we are trying to teach, we can improve the students' experience of the course, and we hope that this will ultimately lead to improving their understanding.

## References


[1] Ken Arnold and James Gosling. *The Java Programming Language*. Addison Wesley, 1996.

[2] Robert Biddle and Ewan Tempero. Java pitfalls for beginners. *SIGCSE Bulletin*, 1998.

[3] John Lewis and William Loftus. *Java Software Solutions*. Addison Wesley, 1998.