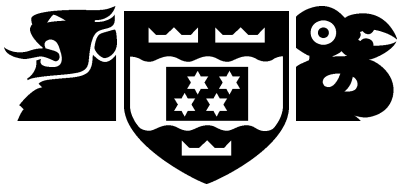
VICTORIA UNIVERSITY OF WELLINGTON
Te Whare Wananga o te Upoko o te Ika a Maui



School of Mathematical and Computing Sciences
Computer Science

PO Box 600
Wellington
New Zealand

Tel: +64 4 463 5341, Fax: +64 4 463 5045
Email: Tech.Reports@mcs.vuw.ac.nz
http://www.mcs.vuw.ac.nz/research

Experience Teaching CS1 with Java

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

Technical Report CS-TR-99/3
March 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 use when teaching 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 behaviour that are difficult to avoid even in the simplest program. This complex behaviour 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.

In this paper we outline our approach to teaching CS1 using Java, and discuss our experience. As the introductory course in programming, CS1 is the critical first step in a computer science degree. The choice of programming language is an important consideration, and affects much of the practical work for both students and teachers. For these reasons we believe it is important to document experience with a new language, both to work towards a better understanding of the impact of language choice on learning computer science, and to offer practical advice for other educators.

We were aware of the difficulties Java presented for beginners [2], and designed our course with these issues in mind. A major part of our preparation involved the development of a course library to allow us to present interesting applications to the students while avoiding many of the complexities of Java. The resulting course was generally successful, however 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 guesses as to what the problems here might be.

Our paper is organised as follows. We begin by summarising our course design, to provide context for the rest of our comments. We then describe the design of our course library, and demonstrate the kinds of applications that were built with it. We then present an evaluation of the course, in particular focusing on the surprises that arose. Finally, we present our conclusions.

2 Context

Our CS1 course is a first course on programming that emphasises 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 driven by a number of issues. We wanted 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 to teach object-oriented design and to teach objects and classes early. We also wanted better support for encapsulation, object-oriented design, and generic container types. Finally, we wanted to present applications that used interaction with graphical user interfaces (GUIs).

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. We discuss this further in the next section.

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, and so on) and automatic indentation of code.

3 Course Library

A significant part of the course preparation involved the development of the course library. This section discusses its motivation and design, and gives examples of the kinds of applications that can be built with the library.

3.1 Motivation

The problem we faced was that Java is not designed as a teaching language, but as a language for developing applications in the real world. As a consequence, it has many features with complex behaviour that must be used in even the simplest programs. We were concerned that when students were grappling with fundamental concepts of programming, these complex features would create confusion and anxiety for the students, and interfere with their learning. We particularly wanted to avoid making the students deal with exceptions or inheritance early in the course, or requiring them to understand large collections of predefined classes.

While we wanted to avoid the more complex features of Java, we still wanted to be able to present non-trivial programs to the students. For example, we wanted them to work on programs that used graphical user interfaces, with both text and event-based I/O. We also wanted them to work on programs that dealt with text files. Writing programs that deal with text, event-based, and file input in standard Java immediately introduces a large collection of predefined classes that have to be combined in complicated ways. Creating even simple graphical user interfaces requires nesting of containers, the use of non-default layout managers, and a variety of different classes and interfaces (for example, `Frame`, `Panel`, `GridLayout`, `Button`, `Graphics`, and `ActionListener` for a simple interface with buttons and graphical output). Programs with any kind of input require dealing with exceptions, and event-driven input is most neatly done with inner classes. We would consider all these features to be more complex than we want to introduce to beginning programmers.

We also wanted students to learn to program in standard Java so that what they learn would be portable to other courses or to other environments outside the university. Students did not like our use of THINK Pascal in the previous version of the course (especially given our dependence on its Macintosh specific features) because they perceived it to be irrelevant to future job prospects. The portability and perceived relevance of Java was a feature we wanted to build on.

One possible approach to Java's complex features is to provide students with almost complete programs in which all the complicated bits are written for them and they need only complete small sections of the program that avoid the complicated bits. A consequence of this approach is that students are not able to understand the programs that they have modified or completed, and never get to write a "complete" program that is totally their own work. We believe that these factors will lead to anxiety and reduced confidence for many students, and therefore will reduce their learning, and possibly cause

them to drop out of the course.

An alternative approach is to construct a library that provides the desired functionality, but hides the complex features inside the implementation of the library. The programs that students write are then "complete" programs in that they can understand all parts of the program, as long as they understand the functionality of the library. A consequence of this approach is that the students are not learning to write programs in "standard Java" so that what they learn is not as portable, and students may perceive the course as less relevant and useful to their goals.

Another alternative is to use Applets, rather than applications. This would allow us to have GUIs without creating extra code and unportable knowledge. However, creating Applets requires using inheritance, which we teach late in the course. We could just use inheritance without explaining it, but the resulting code would look very procedural, conflicting with our objects early approach. Furthermore, Applets would require use (if not explanation) of yet another concept, namely HTML, and would interfere with the use of file input.

There is a clear trade-off between reducing complexity and using "standard Java". We chose to minimise complexity by constructing a library to hide the complex features of Java. However, we were acutely aware of the trade-off, and we attempted to design our library to be as close to the standard Java as possible. In our choice of method names and functionality, and in our design of the event handling model, our goal was to be a simplification of the standard Java rather than a redesign.

A further goal in our design of the library was to provide support for several practical aspects of learning to program in a university course. For example, it was important that students could easily print the output of their programs (both text and graphics) in order to hand in with their assignments. It was also desirable that they could save the output to files to include in other documents. Another issue was that all of the Integrated Development Environments (IDEs) for Java that we were considering using in our labs had inadequate debugging facilities, both in terms of the functionality they provided, and in terms of their ease of use for beginners. We wanted the library to provide some additional support for debugging programs.

3.2 Design

With these motivations, we designed a library consisting of four main classes: one for interaction with the user, one for file reading, one for file writing, and one to support a trace facility for simple debugging.

Simple input/output

The central class of the library is the UI class, which provides methods for text and graphical output, prompted text input, and event based input. When the program creates a UI object, a window will appear on the screen that has 2 main areas: a bar down the left hand side in which graphical input components such as buttons or text fields can be placed, and a graphic/text area that takes up the rest of the window (see figure 3). The window also includes a menu that allows the user to quit, to turn the trace facility on or off, and to print or save the text area, graphics area, or trace window.

The UI provides methods for text and graphical output based on a subset of the methods for the `PrintStream` and `Graphics` classes. The `print` and `println` methods operate an identical fashion to `System.out.print` and `System.out.println` except that their output is displayed in the text area, which can be scrolled, printed, or saved to disk. There are methods for drawing simple graphical objects (lines, rectangles, circles, ovals, arcs, and strings) to the graphics area. Each of these objects can be drawn or erased, and for each of the 2D shapes there is the ability to draw the shape filled or outlined.

The UI provides methods for prompted text input that use dialog boxes to allow the user to enter `doubles`, `ints`, `Strings`, or `booleans`. The methods guarantee to return a value of the correct type. This means that the student

programs do not have to deal with invalid input or convert character input into numbers or other types.

Event-based input

The UI class also eases the task of doing event-based input. It provides a simple, but very constrained, way of building interfaces. Its event handling model is the same delegation model used in the AWT, but it provides simpler interfaces for event listeners.

The UI class provides methods for adding input components (buttons, text-fields, number-fields, and/or sliders) to the bar on the left-hand side of the UI window. Components are placed in the bar in the order that they are added to the window. The `addButton`, `addTextField` and `addNumberField` methods require a `String` that will be the name of the button or field, and an object that will handle events from the component. The `setMouseListener` and `setMouseMotionListener` methods allow an object to listen to mouse events (see figure 4.

In the AWT, an object that handles events from components must implement the `ActionListener` interface and provide an `actionPerformed` method whose argument is an event that the method must unpack to determine the details of the event. The UI uses the same model, but to reduce the complexity of manipulating `EventObjects`, it "unpacks" events and passes just the relevant values to the object that is handling the events. To make this work, the object must implement a different kind of listener interface (and define a different `...performed` method) for each kind of input component that it responds to. Although this involves more different interfaces than the AWT, the methods that the students must write are simpler than for the AWT. We believe that the UI model is sufficiently parallel to the AWT model that students will have little difficulty using the AWT in later courses.

In a typical program in the course, the `buttonPerformed` method (for example) will consist of a nested **if** statement that dispatches on the name of the button and does the appropriate action for each button. Similarly for the other `...Performed` methods.

In the AWT, the `MouseListener` interface is rather different from the `ActionListener` interface in that it involves five different methods to respond to the different kinds of mouse actions (and seven for the `MouseMotionListener`). The UI model for mouse events is more consistent with the model for events from input components. The UI's `MouseListener` interface requires just one method — `mousePerformed` — that takes a string specifying the kind of mouse event ("clicked", "pressed", *etc*), and four `doubles` specifying the current position of the mouse and the position of the mouse when it was last "pressed". The first argument is parallel to the name of the component in the other `...Performed` methods.

We note that the UI's event handling does require the use of Java interfaces, which we consider to be a complex feature, but there is no clean way to avoid it. In lectures, we present just the aspects of `interface` needed for these programs, but do not explain the details of interfaces until quite late in the course. This does not seem to cause the students difficulty.

File input and output

The other two major classes of the library are for reading and writing text files.

The `DataFileWriter` is a simple class that allows students to write text to files instead of the text output window. The `DataFileWriter` constructor uses a file dialog to allow the user to choose the file to be written. It has overloaded `print` and `println` methods, parallel to the UI text output, to write values to the file.

Input from files is more complicated, firstly because one has to deal with invalid input in some way, and secondly because different applications may need to read a file in different ways. The `DataFileReader` class provides methods for reading data from a file in several different ways, and hides all exceptions. It uses sentinel values to guarantee that the methods always return a value. Like

`DataFileWriter`, it has a constructor that uses a file dialog to allow the user to choose the file to be read. The `endOfFile` method indicates whether the reader is at the end of the file.

There are two modes for reading from a file: input can be character-based or token-based. In the character-based mode, `readChar` and `readLine` will read the data in the file one character or one line at a time. They return the null character or `null`, respectively, if at the end of the file. In the token-based mode, the file is viewed as a sequence of tokens. There are three kinds of tokens: numbers, words (alphanumeric), or punctuation symbols. `readToken` will read the next token in the file, returning `null` if at the end of the file. If the next token in the file is a number, `readNumber` will return that number; otherwise, it will return a sentinel value (Double.MAX_VALUE). Programs can switch between the two reading modes on the same file.

This simple tokenising enables a variety of student programs that deal with very differently structured files without the complications of explicitly using streams and tokenisers. The use of sentinel values enables the students to write programs that cope with invalid files without having to deal with exceptions.

Tracing

The last of the four public classes in the library is the Trace class, which supports simple debugging. The Trace class creates a separate window in which debugging statements can be printed using the `Trace.print` and `Trace.println` methods. The trace window can be turned on and off asynchronously from an option on the menu bar or from inside the program using the `setVisible` method. The contents of the trace window can also be printed from the menu. The `Trace.pause` method, which pops up a modal dialog box with a message and a continue button, provides a very limited sort of breakpoint facility.

The methods of the Trace class are all static, so there is just one Trace window for the whole program, and the methods can be called without having to pass a Trace object around the program. This makes it easy to add debugging statements without modifying the rest of the program. Because the trace output is in a separate window, the debugging statements also do not modify the behaviour of the program. Because the tracing can be turned off, there is no need to remove debugging statements in order to demonstrate the program.

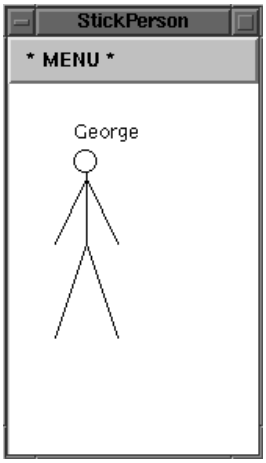


Figure 1: The `StickPerson` user interface for exercise 2.

Portability

We have successfully used the library in our CS1 course this year on the MetroWerks Mac Java VM, the Solaris JDK for PC, Solaris, and Digital Unix, and Apple's Mac Java VM.

3.3 Using `UI`

The most visible aspect of the course library is the `UI` class. In this section, we give a flavour of what it is like to use this class. Specifically we show how the `UI` class would be used in a typical class lab exercise.

Figure 1 shows the user interface for one of the exercises that the students work on. This is the first exercise that requires students to write Java code, and is given out in the second week of the course. The code the students produce is shown in figure 2. This application prompts the user for an integer representing a height and a string representing

a name, and then draws a stick figure of the specified height with the specified name. All the required operations are performed in an instance of the `UI` class, `window`. The methods that prompt for input, `getDouble` and `getString`, do so via dialog boxes.

As mentioned earlier, we wanted to show applications with event-based input. The first exercise where students see this kind of application is given out in the fourth week, with more complex applications provided as the course progresses. Figure 3 shows the interface for the exercise given out in the fifth week of the course. This particular application shows most of the capabilities of the graphics window. The text window isn't shown, but if the application were to print something on the text window, it would occupy the space currently occupied by the graphics window. As shown in the figure, the UI class forces all buttons, sliders, and textfields to be placed down the left-hand side.

Figure 4 shows the source code to the program shown in figure 3. The students are given, as a starting point, a file containing:

```
import mcs.comp100.*;

public class MiniPaint{
  public static void main(String[] args){
    new Painter();
  }
}
```

The students are responsible for everything else.

As discussed above, the design of the UI class is intended to allow students to write code similar to what they would write if they were using the AWT directly. As figure 4 shows, in the course we follow the convention of making the class that manages the user interface (`Painter` in this case) also implement the interfaces that handle the event input. This is not required by the UI class. Separate listener classes can be used.

Figure 3 also shows the standard menu provided in the UI class. As the figure indicates, students can print the contents of what can be viewed in the graphics, text, or trace windows, or can save the text or trace windows

```
// The package for the course library
import mcs.comp100.*;
public class StickPerson {
  public static void main(String[] args) {
    UI window = new UI("StickPerson");
    double x = 50;
    double y = 50;

    double height =
              window.getDouble("Height?");
    String name =
              window.getString("Name?");

    double headRadius = 0.06 * height;
    window.drawString(name, x - headRadius,
                      y - headRadius*2);
    window.drawCircle(x - headRadius,
                      y - headRadius,
                      headRadius * 2);
    double neckY =  y + headRadius;
    double hipY =  y + headRadius + 0.38 *
                   height;
    window.drawLine(x, neckY, x, hipY);
    double footY = hipY + height/2;
    double offset = height/6;
    window.drawLine(x, hipY, x-offset,
                    footY);
    window.drawLine(x, hipY, x+offset,
                    footY);
    double shoulderY =  neckY +
                        headRadius/2;
    window.drawLine(x, shoulderY, x-offset,
                    hipY);
    window.drawLine(x, shoulderY, x+offset,
                    hipY);
  }
}
```

Figure 2: The `StickPerson` source code

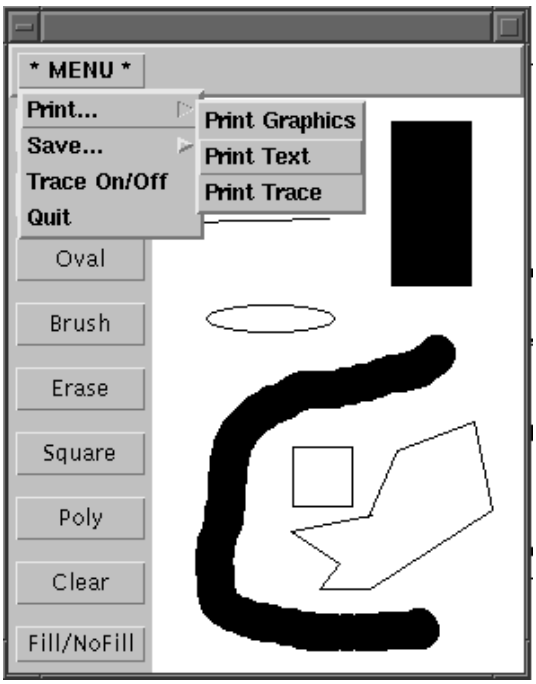


Figure 3: The Minipaint application user interface.

to a file, or turn tracing on or off. As we discuss later, having this standard behaviour in each application is of considerable benefit.

4 Evaluation

In this section we review the major issues involved in teaching CS1 with Java, discuss how our course tackled these issues, and evaluate the results. In particular, we focus on the surprises that arose. Section 4.1 covers the early part of the course, which covered the introduction of objects, definition and use of classes, and event-driven input, section 4.2 discusses the problems caused by the distinction between primitive and object types in Java, and section 4.3 discusses what was the biggest surprise of the course — the unintended downplaying of types.

Overall, we believe our course was successful. Anecdotal feedback suggests that 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.

4.1 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.

4.1.1 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 fourth 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 4.2) provides the basis for this speculation.

4.1.2 Method and Parameter declarations

Immediately after showing how to instantiate objects and call methods on them, we addressed the declaration of classes and meth-

```
import mcs.comp100.*;

class Painter implements MouseListener, ButtonListener{
  private UI gui = new UI();
  // ...... other declarations elided
  public Painter() {
    this.gui.addButton("Line", this);
    this.gui.addButton("Rect", this);
    // ...... similarly for the other buttons
    this.gui.setMouseMotionListener(this);
  }
  public void buttonPerformed(String button) {
    if (button.equals("Clear")) {
      this.gui.clear();
    } else if (button.equals("Fill/NoFill")) {
      this.filled = !this.filled;
    } else {
      this.currentShape = button;
      this.polyInProgress = false;
    }
  }
  public void mousePerformed (String action, double x, double y,
                              double lastx, double lasty) {
    if (action.equals("Released"))
      this.drawShape(lastx, lasty, x, y);
    if (action.equals("Dragged"))
      this.dragShape(x, y);
  }
  // ...... the other methods elided
}
```

Figure 4: Partial source code for the Minipaint application shown in figure 3

ods. 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.

4.1.3 Event-driven input

As soon as the students had the basic concepts of class and method declarations, and the conditional statement, we introduced programs with event driven input (buttons, textfields, mouse, and so on). 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.

4.2 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 pa-

rameter passing, in difficulties in determining whether to compare using `==` or `equals()` and in confusion over the behaviour of Strings.

4.2.1 Syntactic Markers for Dereferencing

We analysed the problems we were having with references and object. Our conclusion was that in order to make clear the distinction between references and the objects to which they refer, there has to be 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. This meant the students did not have a good understanding as 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.

4.2.2 Passing Objects vs Primitive Types

When an object is passed to a method, any changes made to that object persist after the method has completed executing. When a primitive data value is passed, 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 behaviour 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.

4.2.3 Testing Equality

We took great pains to emphasise 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, however, this is not always the case. In particular, the String object type causes a lot of problems.

4.2.4 The String type

Strings, while object types, have specific support in the language. In particular, whenever

a program creates two `String` literals with the same characters, only one String object is created with two references to that object. This has the consequence that `==` can, in some situations, be sensibly used to determine if two String objects have the same value.

This specialised behaviour of `==` for Strings undermines the distinction between primitive and object types. We emphasise 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 usually get the behaviour they expect, which reinforces the wrong model about object vs primitive types.

4.3 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 object-oriented languages, such as Java, than in Pascal. It is possible that our confidence in this support simply resulted in a lack of attention to the issue, but, we now realize that there are several ways in which Java supports the type concept less well than Pascal.

4.3.1 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-emphasise 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 emphasised 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 colours 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).

4.3.2 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 parameterised 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 emphasised 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.

4.3.3 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 emphasised 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 emphasise 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.

4.4 Library

Our use of a course-specific library achieved what we hoped for. In particular, it allowed us to show reasonably interesting programs to students very early, and more importantly, allowed the students themselves to write such applications.

There were two advantages to having our own library that we didn't anticipate: one minor and one major. The minor advantage was that it allowed us to mask inadequacies we encountered with the CodeWarrior implementation of some AWT components. The major advantage was that it solved a problem we didn't even realise we would have, namely allowing the students to easily get output from their programs to hand in for marking.

As shown in figure 3, the UI class has a standard menu that allows various parts of the application output to be sent to a printer, or saved to a file. Providing this functionality directly from AWT is very complex, and so we would have had to package it up in some form anyway. The fact that it was packaged up with the UI class meant that we were able to give a uniform presentation of applications.

Of course, had we had electronic submission, this would not have been so much of a problem. However, we did not have electronic submission, and furthermore students liked to be able to print their results for their own use. The importance of this functionality only became evident in our CS2 course, which

initially planned to use AWT directly. When the complexity of printing from AWT applications was realised, the CS2 labs were developed using the CS1 library instead, without any difficulty or limitations imposed.

In general, the UI class has been sufficiently successful that it is being used for non-course applications.

5 Conclusions

Java is not designed as a teaching language, and so care must be taken when using it to teach introductory programming. We were aware of the problems, and took care to address them in our course design, as we have explained above. In particular, we designed a course library design to conceal Java complexities but allow non-trivial programs, including graphics and GUI interaction, in a manner consistent with the more complex standard Java libraries.

Although our course was successful, 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*, Volume 30, Number 2, pages 48–52, June 1998.

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