SOFTWARE—PRACTICE AND EXPERIENCE
Softw. Pract. Exper., **29**(5), 457–478 (1999)

A Critique of Java

HAROLD THIMBLEBY*
Middlesex University, Bounds Green Road, London N11 2NQ, UK
(email: harold@mdx.ac.uk)

> "Java's ease of programming and safety features help you quickly produce working code."
>
> K. Arnold and J. Gosling in *Java Programming Language* [1]

> "I was eventually persuaded of the need to design programming notations so as to maximise the number of errors which cannot be made, or if made, can be reliably detected at compile time."
>
> C. A. R. Hoare [2]

SUMMARY

Our experience of using Java is disappointing: as a programming language (irrespective of its implementations and libraries), Java itself leaves much to be desired. This paper discusses a few serious problems with Java's design, which leads us to suggest that the language definition should have been an integral part of the design process rather than, as appears, a retrospective commentary. Copyright © 1999 John Wiley & Sons, Ltd.

KEY WORDS: Java; programming language design; documentation and explanation

INTRODUCTION

To be successful, a system must have a user base. The users must get enough of their tasks achieved within reasonable limits of performance, and given reasonable limits of training. From its first public release in 1994, Java has rapidly become a very popular programming language. Java is associated with the World Wide Web – and the Web's global scale – and it has applications near and far, from smart cards to the 2001 Mars Lander. There are hundreds of books on Java, specialist magazines and web sites. Java is now taught in hundreds of universities. Java is clearly a mainstream phenomenon.

In many ways, Java is a classic computer application, with its design and introduction requiring trade-offs. Its design had to balance being different and being 'better'. It had to successfully draw on enough users to make it a viable product. Java closely resembles C and C++, so existing programmers find it familiar. However, these languages have many problems and ambiguities, so Java made changes in order to have advantages over them.

We shall argue that Java has unfortunate and avoidable weaknesses. We will not discuss what Java might have been, because object orientation is a vast subject: instead, we try to

*Correspondence to: H. Thimbleby, Middlesex University, Bounds Green Road, London N11 2NQ, UK.
Contract/grant sponsor: EPSRC; Contract/grant number: GR/K79376.

CCC 0038–0644/99/050457–22$17.50

Received 3 August 1998
Revised 23 December 1998
Accepted 31 December 1998

critique Java specifically in the way it does what it does. We will take Java on its own terms. The reader interested in further exploring alternatives is referred to Abadi and Cardelli [3]; a dated but more accessible book is Tennent [4], who provides a set of programming language design principles – many of which are implicit in our discussion here.

Unfortunately, we are arguing from hindsight: any changes now to Java would compromise it, and almost certainly reduce its massive following. Practically, this paper will warn Java programmers of certain weaknesses in the language: they may therefore be able to take suitable precautions. Also, this paper will encourage future programming language designers to consider their language design more carefully: Java has so many users, even small improvements – at small cost to the designers – would have had enormous, indeed world-wide, benefits for its huge numbers of programmers.

For brevity, and sufficiently to make our point, we only discuss Java as a programming language; we do not discuss security, exceptions, concurrency, and package details (such as Beans, Swing, and so forth). This paper does not discuss Java's inefficiency or unreliability, although this is partly a consequence of the language being badly designed, and therefore unnecessarily hard to implement well. Interesting as the issues are, politics, standardisation, and the various flavours of Java (e.g. for embedded systems) are beyond the scope of this paper. As we will make few positive comments about Java: it should be said, then, at the outset, that Java is a major accomplishment, and an amazing success in marketing.

NOTATION, BARRIERS AND TRAPS

In any critique, it is easy to concentrate on disagreements in conventions, and particularly over notation.

Discussion about notation can be tedious. However, tedium itself can encourage errors – at its simplest, just having to do more typing increases the chance of making an error. We will give just four very brief examples of notational inefficiencies in Java, before turning to more central issues. Such inefficiencies have to be weighed up against the prior expectations of programmers. Just like the Qwerty keyboard: there may be more efficient keyboards, but they aren't popular enough to catch on. Being efficient is no good if nobody uses your system!

Here are some illustrative notational issues:

1. In Java the 32 bit numeral `71` can be made into a 64 bit numeral by appending a letter `l`, as in `71l`. Arguably, the notation should make this sort of difference clearer. Or it would have been easier if Java had been designed so numerals took just as many bits (`char`, `int`, `long`) as they required, and the compiler complained when there was numeric overflow.
2. Java uses Unicode, so a Java letter (for use in an identifier name) can be from almost any language, so A (Latin alphabet) and Α (Greek alphabet) are different.
3. Casts in Java are prefix (as in C). If they were postfix, fewer brackets would be needed and the code would be more readable. For example, Java's `((AClass) a.elementAt(n)).action()` could be more conveniently written, without having to balance nested brackets across arbitrarily long pieces of code, as `a.elementAt(n) (AClass).action()`.
4. Java's statement syntax 'taken over' from C allows compound statements wherever simple statements are permitted. Thus, the conditional statement in an `if` can either be a simple statement, or a compound statement grouped by curly brackets. Java introduced a new control structure (`throw`, `catch`, `finally`) but the statements guarded by

these structures *must* be ‘compound’. For example, it is not permitted to write `try a = b/c;` instead one has to write `try { a = b/c; }`

Perhaps one should live with such thoughts; a language has to make *some* decisions, and that we might not take to them is not an objective critique of the language. In this paper, we want to take a higher level view. The programmer faces two quite different sorts of more serious problem: *barriers*, which are explicit limitations to desired expressiveness, and *traps*, which are unknown and unexpected problems. Typically, a barrier reveals itself as a compile time error, or in the programmer being unable to find any way to conveniently express themselves. A trap, however, is much more dangerous: typically, a program fails for an unknown reason, and the reason is not visible in the program itself.

Java has become very popular largely because its of its improved type checking, its run time array bound checks, and its removal of explicit pointers: all these improvements can be understood as converting traps in C and C++ into barriers in Java: thus helping programmers write more robust programs.

In contrast, Java’s rules for variable initialisation form a trap. There are several different sorts of initialisation (that apply differently to local variables, class variables, parameters, etc., and to different types – primitive, class and array), and Java provides protection against only one sort of initialisation error. But with the apparent guarantee, a programmer might accidentally rely on a class variable or an array element being correctly initialised. Unfortunately, they are not. And since Java has no ‘undefined’ value any such incorrect assumption is unlikely to be easily discovered.

JAVA’S IMPROVEMENTS OVER C AND C++

C is a very popular systems programming language. It is concise, efficient and provides close control over hardware. The language has several idiomatic expressions (such as `while( *p++ = *q++ )`) that at first appear very obscure; their effective use gives expert C programmers a deep sense of mastery. C programs, however, are notoriously unreliable. Indeed, the C idiom shown above will generally leave both pointers, `p` and `q`, pointing outside the original data structures, and any use of their values will then lead to run-time errors. In general, C allows pointers to point to anywhere, and it provides type casts so that whatever a pointer points at can be converted into something that can be manipulated. In particular, C has an operator (`&`) that allows a program to obtain pointers to *anything*. Such features are extremely useful for directly controlling hardware – say, to send bytes to a peripheral – but unfortunately the same features *ensure* a C compiler cannot provide any protection, as when a program makes accidental changes to itself in arbitrary places.

Java overcomes these problems by two restrictions. Java simply has no pointers, so it is not possible to access arbitrary parts of a program or to get ‘inside’ private data structures, whose actual structure may not quite be what the programmer planned. Secondly, Java has much stricter type checking. For example, it is not possible to change the exponent of a floating point number by treating it as an array of bytes – whether deliberately or accidentally.

Both C and Java have arrays. In C, arrays are defined to be equivalent to pointer expressions, so they have exactly the same liabilities pointers do, perhaps with the added danger that they look innocuous. The legal C assignment `a[5]=6` looks perfectly reasonable, but if `a` is a pointer to an array declared to have only four elements, then the result of the assignment would be undefined. In fact, `a` is a pointer and could be pointing *anywhere* (say, to a C function) when the attempted assignment occurs! Quite possibly the assignment would

change the C program's return stack, and cause the current function to return to an arbitrary point in the program, and then anything could happen. In Java, however, the subscript range is part of the array's type, and attempting to access an element outside the appropriate range is an error that is always detected. In other words, the common traps of C have been converted to explicit barriers in Java.

Despite these improvements, Java nevertheless remains very close to C. A simple C program can be converted to Java more-or-less by just saying it is a class and fixing the errors the Java compiler reports. Doing so will, in many cases, produce a much more reliable program.

C++ is an object oriented extension to C, and while introducing object oriented concepts, it retains all the features of C that make it both efficient and unreliable, as described above. C++ is a complex language, and Java effectively uses the 'core' object oriented features without many of C++'s complications, such as multiple inheritance. C++'s design principles were not to create reliable or portable programs, instead it was designed to be useful and enjoyable for serious programmers [5]. C++ certainly succeeded, and Java builds on that enthusiasm – trading the fun of C++ with the fun of the Internet and the opportunity to run Java programs world wide.

THE IDEA OF PARADIGMS

Computers are faster than humans, and the purpose of programming is to define the behaviour of computers as effectively as possible, so they operate as intended under future circumstances that have not been predicted in complete detail. In an accounting program, we might write `balance=income-expenditure`, intending this to enable a computer to calculate an account's balance without further human intervention. The program fragment is intended to specify the general behaviour of the computer, that it should subtract two unspecified numbers, whose actual values will be provided in the future. Naturally it follows that the semantics of programming languages must be well-defined, otherwise programmers have to continually interfere with the operation of the computer to check it is behaving as intended, or at least they would have to undertake extensive checking before a program could be relied on to work without supervision. Badly defined semantics therefore lose the leverage programming languages are intended to provide.

Very few applications of computers conform nicely to the limitations of computers. Thus, if the example above was used to calculate the national debt, there may be problems with the representation of large numbers. Although being explicit about such limitations may make the behaviour of a program better defined, being explicit can be cumbersome and make the program harder to write and read – it also forces the programmer into considering exceptional behaviour that may itself be a distraction from considering more usual, and hence more salient, behaviour. What is gained in precision is lost in the human factors consequences of making programming harder to do.

In practice, programming languages take certain sorts of semantics as given. This defines their *paradigm* [6]. Since all programming languages share certain common assumptions (e.g. that numbers have finite representations) the term paradigm is usually taken to be what hidden semantics are *distinctive* of a particular language. Java, for instance, has an object oriented paradigm. In Java, objects can be created and manipulated *without* specifying the semantics of objects, in a way that would not be possible in a language such as C. In C, any manipulation of objects would need to be specified by the programmer explicitly: thus, C is not object oriented.

Because any object orientation done in C would have to be done explicitly by the programmers (and if several programmers are involved, they would all have to agree on how to do it) there is a strong possibility things would go wrong. Doing inheritance in C would require some explicit pointer manipulation, and programmers would have to remember to call the appropriate routines at the appropriate times. Paradigms, because they hide semantics, ensure that programmers cannot get them wrong.

In Java, there is no way to negotiate inheritance: programmers get what Java gives them, and there is no need to call routines at appropriate times to get things to work. Another example: in Java, programmers cannot do garbage collection, and they cannot make Java's garbage collection go wrong: thus, Java programs are more reliable than equivalent C programs, which would have to rely on specially written code and programmers *explicitly* adhering to appropriate conventions to avoid breaking their garbage collector.

There is little agreement exactly what many paradigms should mean. Different object oriented languages provide different semantics for objects – for example, C++ does not do garbage collection fully. Given there are differences, the main problem is that programming language designers and programming language users may not agree; moreover, because a paradigm is not explicit in the programming language notation, there may be no simple way to uncover this disagreement. The differences may be very subtle. The consequences are that programs written in such languages will be unreliable.

A misunderstanding about a paradigm may result in the entire structure of a program being misconceived. When – if ever! – the flaws in the program are noticed the changes needed to correct the program from the incorrect to the correct use of the paradigm will require a complete redesign and rewrite. In such circumstances, the paradigm becomes a major trap. In turn, attempts to avoid potential problems may encourage some programmers not to use the paradigm effectively.

In a sense, a programming language designer cannot be responsible for the ignorance of a programmer. Programming languages typically make explicit certain distinctions intended to guide the programmer in making appropriate programming decisions. Unfortunately, what may seem explicit to a designer may not be so obvious to a user. A case in point in Java is that the *language* distinguishes between 16, 32 and 64 bit numbers, but it is possible to write programs where such a distinction is accidentally overlooked, because the distinction is not sufficiently obvious to the programmer. The result will be a program that works correctly until circumstances offer it a number that is at an unintended precision; subsequent calculations will go awry, and the semantics of the program will be quite other than intended.

There will always be human error, against which the programming language designer has to make trade-offs. Some sorts of error can be anticipated, and made harder to commit – or, at least, harder to commit without knowing. Yet each barrier against accidental error can make deliberate exploitation of a programming language feature harder. For example, if Java had required some explicit and obvious statement when precisions less than 64 bits were required, then (as many numbers are 32 bit) programmer productivity could be reduced or their programs would be less efficient.

It is an empirical question what sorts of errors programmers make, and with what sorts of frequencies. Yet an appeal to empirical evidence may just delay designing (or critiquing) a language. There are errors that, while infrequently made, are very easily detected, or whose consequences may be so severe that the recruitment of empirical evidence is secondary. We put Java's object orientation in this latter category.

IMPORTING PACKAGES: A BRIEF EXAMPLE OF DESIGN ISSUES

This paper criticises Java in its own terms so far as possible. To illustrate the approach, we take a simple example of a barrier.

Java requires `import` statements to be placed at the beginning of source files, immediately after any `package` statement and before any other code. When the `import` statement is first mentioned in the *Java Programming Language* book [1], no restrictions on its use are mentioned. However, later the book gives a more detailed example:

> "A programmer who wants to use the `attr` package could put the following line near the top of a source file (after any package declaration but before anything else)." p. 210

Note the phrase in brackets. It describes a limitation, and one that need not have been designed into Java. Java could have been defined so that `import` statements could appear anywhere without restriction (this might also have permitted scoping of imported packages, to avoid name conflicts). To do so would have saved a restrictive phrase in the explanation.

Elsewhere in the book there is example code (p. 327):

```
// We have imported java.io.* and java.util.*
public String[] ls(String dir, String opts)
...
```

This code requires an `import` statement, but because of the restriction on the placement of such statements, there has to be a comment in English that there *would have been* an `import` statement in the program from which the example is exerpted. Thus, this code cannot be tested as it stands. In short, here is a second illustration of the consequences of a simple design choice – restricting the locations of `import` statements – being visible in the explanation of the programming language. In this case, the design choice has a negative effect on the explanation of Java (and, by implication, on the learnability of the language).

Restrictions on where `import` statements are placed is a barrier, as the compiler prohibits incorrect use. As `import` statements only make naming more convenient, where they go does not restrict expressiveness. Anyone can live with having to write `import` statements near the beginning of files – indeed, what are editors for? – so it is a trivial issue. But as the restriction is arbitrary, why not design the language without the restriction, for the description of the restricted language including providing examples is unnecessarily tedious and error prone.

Overall there are many improvements that can be made to Java (many more interesting than the `import` suggestion!), which can be determined by reading Java's own explanation, looking for verbosity, avoidable hedges, restrictive clauses, omissions or direct admission of design problems.

OBJECT ORIENTATION IN JAVA

The object orientated programming paradigm is appealing for many reasons. The real world contains objects, and object oriented programs are intended to model aspects of real world objects readily. Polymorphism is a central concept: it allows objects to substitute for each other – for example, you could substitute a duck object where an animal is required. Object orientation includes inheritance, which is a mechanism for extending functionality by reusing existing code. In particular, inheritance lets a programmer get new functionality for free,

inheriting most of what is needed from existing classes, and therefore is itself a sort-of paradigm-maker in its own right (inheritance provides programming features that need not be explicit to the inheriting object). It is therefore essential that object orientation works well.

This section describes some of the obfuscation of Java's inheritance and polymorphism mechanisms. It is possible to show the problems with very simple examples: we will define a class of ducks and a class of lame ducks, as a subclass of ducks. The meanings of even such simple classes are far from obvious: hence raising the spectre that really serious problems will lurk in more sophisticated Java programs. Although the confusing examples here are simplistic, real programs will be much larger, and it will be *very much* harder to correctly identify and fix their bugs.

We define a duck to have two feet. Ducks calculate how many legs they have by counting their feet. (In a realistic application, methods will do something more sophisticated; but maybe just returning the number of feet is enough for a duck!)

```
class Duck
{ int feet = 2;
  public int legs() { return feet; }
}
```

If we ask a duck, it has 2 feet, and 2 legs, as we would expect. Now a lame duck is a duck with only one foot, and as lame ducks are ducks, we define lame ducks as a subclass of ducks:

```
class LameDuck extends Duck
{ int feet = 1;
  public int legs() { return feet; }
}
```

Note that `LameDuck` has its own definitions of `feet` and `legs`, so things should be straight forward. Indeed, if we ask a lame duck, it has 1 foot, and 1 leg, as we would expect.

Now, since lame ducks are ducks, we can assign a lame duck object to a duck variable:

```
Duck d = new LameDuck();
```

If we ask a duck that is a lame duck, we find it has one leg but two feet! Why?

At this point, you probably fall into one of two sorts of programmer:

(a) For you, the reason why Java behaves like this is completely trivial, and you wonder why this paper mentions the example; or
(b) You are confused.

In fact most people are confused, and those who aren't yet confused might like to wait until we quote what the Java designers themselves say, or wonder why a 'popular' programming language confuses many programmers.[†] Few Java programming language books discuss the issue, so the trap is baited for inexperienced programmers.

Since all ducks are the same, and all lame ducks are the same, we may as well make the feet and legs `static` (`static` methods and fields are intended to be the same for all objects in a class). And if we do so, we find ducks that are lame ducks have two `static` legs and two `static` feet. In other words, `static` members of classes work differently!

[†]The lame duck's *own* method `legs` is called (which accesses the lame duck's field), however because `d` is a duck, the lame duck's field is hidden, so the lame duck seems to have two feet.

This seems curious. (It means that a method cannot guarantee what fields it accesses, because it depends on how they are declared – as class or object variables.) It is very easy to confuse the different behaviour of fields and methods. This is a point that is almost made in the *Java Programming Language* book [1]:

> "You've already seen that method overriding enables you to extend existing code by reusing it with objects of expanded, specialized functionality not forseen by the inventor of the original code. But where fields are concerned, it is hard to think of cases in which hiding them is a useful feature." p. 69

It might be hard to think of a reason, but on the very next page, the authors explain why:

> "Hiding fields is allowed in Java because implementors of existing super-classes must be free to add new `public` or `protected` fields without breaking subclasses." p. 70

This is a spurious reason. For example, simply changing a method to `final` (see below for a discussion of `final`) breaks all subclasses that override it. Ironically, one purpose of `final` is to improve efficiency, and efficiency is almost certainly the reason for Java's treatment of fields – they do not need dynamic lookup, and are therefore more efficient than methods.

> "Purists might well argue that classes should only have `private` data, but Java lets you decide on your style." p. 70

Careful Java programmers – or purists – will therefore define all fields to be `private`, and will provide accessor methods if the field's values are needed outside of the class body. (On the other hand, if Java required all fields to be `private`, then introducing a new field could *not* break a subclass, as the designers of Java evidently worry.) It is a shame that the design of the language opposes clarity (rhetorically dismissed as purity) to efficiency. In my view, being confused and fast, as Java's design explicitly encourages over this issue, only means you unwittingly do the wrong things faster. It is an approach that has little to commend it.

From classes to inner classes

Now suppose we allow that ducks lay eggs. Both `Duck` and `LameDuck` can have inner classes that define `Egg` objects. Inner classes are contained within their enclosing class, and are always `private`; so fields and methods of an inner class are hidden and do not override anything defined by another class. Since inner classes hide, their methods also hide rather than override. Thus,

```
new Duck().new Egg().hatch() // hatch a duck's egg
new LameDuck().new Egg().hatch() // hatch a lame duck's egg
((Duck) new LameDuck()).new Egg().hatch() // hatch a duck's egg
```

If the `hatch` method checks that legs and feet correspond – they don't in the last case – we're going to have some confused ducklings. (Although it is a duck's egg, the object referring to it is a lame duck, so the duck's method `legs` is overridden.)

This is one example where a method invoked (lexically) inside a class need not be a member of that class, but there is nothing unreasonable about this. The following class definitions have a call to `legs` inside a class which may be an invocation for a subclass member, as the fact that `((Duck) new LameDuck()).checkLegs()` is 1 proves:

```
class Duck
{ int legs() { return 2; }
  int checkLegs() { return legs(); }
}

class LameDuck extends Duck
{ int legs() { return 1; }
}
```

Java provides two ways to avoid overriding: a method can be declared `private` (so it is not known in a subclass) or it can be `final` (so it can be used but not overridden in a subclass). It follows that a method that is `private final` is, in this sense, the same as one that is just `private`.

Now consider this code:

```
class Duck
{ int legs() { return 2; }
  class Egg
  { int checkEggLegs() { return legs(); }
  }
}

class LameDuck extends Duck
{ int legs() { return 1; }
}
```

In the context of these definitions, a call `((Duck) new LameDuck()).new Egg().checkEggLegs()` returns 1. The method `legs` called in `checkEggLegs` is the method from `LameDuck` – even though no instance of `Egg` has a `legs` method (`Egg` is a direct subclass of `Object`, not `Duck`). Moreover, making `legs` `private` in `Duck` makes `Egg` throw a run time error (an `IllegalAccessException`). If a use of `legs` in `Egg` should be overridable, *despite* being `private`, this would make sense, because both ducks and lame ducks could use `Egg`, but only duck's eggs could access their own private legs. Thus it would be an error for a lame duck to invoke a duck's legs through an egg, and in principle this restriction cannot be detected at compile time. Time to read the book?

> "A nested class can use other members of its enclosing class – including private fields – without qualification because it is part of the enclosing class's implementation. An inner class can simply name the members of its enclosing object to use them [. . .]" pp. 52–53

Nevertheless, simply naming a member without qualification *can* get an overridden member. What are we to make of the claim:

> "Nesting is needed to make local code robust. If hiding outer variables were not allowed, adding a new field to a class or interface could break existing code that

> used variables of the same name. Scoping is meant as protection for the system as a whole rather than support for reusing identifiers." p. 114

Can ducks hide nested eggs?

At this point, some readers will be shouting that the code is rather artificial and convoluted, and that it does not represent production code. Certainly, the very brief code we used to illustrate inner classes is contrived. Yet the techniques are in principle perfectly straightforward, and could easily be used in production code, say, in a financial package. If this paper had written out in full a really good example based on a financial package, the chances are that the reader would not have clearly spotted the problems at all – which is exactly the trap facing real programmers: that the details of how the paradigms work are lost in details of what the program is supposed to be doing. If you found a few lines of ducks and eggs mysterious, that probably means that exploiting inner classes in larger programs will be even more obscure.

Method parameters

So far overriding methods have had no parameters; we'll now see that introducing parameters considerably complicates matters. We shall make lame ducks be 'politically correct' so they consider themselves the equal of *any* duck, lame or otherwise. Lame ducks are given a method to achieve this:[‡]

```
public boolean equals(Duck d) { return true; }
```

Straightforwardly, we want lame ducks to implement a different version of `equals`; indeed,

> "[. . .] `equals` methods should be overridden if you want to provide a notion of equality different from the default implementation provided in the `Object` class. The default is that any two different objects are not `equal` [. . .]" p. 73

And, as planned, we find that a duck thinks it is not the equal of any other duck, lame or otherwise (using the default `equals`), and that a lame duck thinks it is the equal of any duck (using its overriding `equals`). For example,

```
LameDuck ld = new LameDuck();
ld.equals(new Duck()) // returns true, as expected
```

The trap is that we might think from this example that the definition works correctly. Unfortunately, if we ask a duck that is a lame duck, it thinks it is *not* the equal of another duck!

```
Duck d = new LameDuck();
d.equals(new Duck()) // returns false
```

Here, one might have expected the `LameDuck`'s `equals` method to override any such method inherited from the `Duck` class, but as it returns `false`, it is clear that the lame duck's `equals` (which always returns `true`) has not overriden anything. It is worth pausing

[‡]Wise lame ducks would also override `hashCode` so that nobody could use it to tell they were not the same, though `System.identityHashCode` would still be available (it is not clear why `identityHashCode` is a member of `System` and not `Object`, where it belongs as a `final` method).

to wonder why, as this is a very common issue in Java programs – it is what overriding is all about. The duck classes represent any case where we wish to invoke any method in a superclass; the technique can be illustrated with only a few lines of code, though here the few lines of code are surprising.

The explanation for the behaviour is that the `equals` method for lame ducks does not override the `equals` method for ducks, because ducks inherit from `Object`, whose `equals` method takes an `Object` parameter. Even though a duck's `legs` method can be a lame duck's `legs` method, its `equals` method is inherited from `Object` (which makes different objects unequal). In short, a lame duck's `equals` method does *not* override duck's `equals`.

The trap is that a simple mistake, using a wrong parameter type in a method, will go unreported by a compiler and the class will work correctly except in certain cases. The nature of inheritance is that the class will probably work fine when it is tested but not when it is part of a production program. Ironically, the mistake can arise when a programmer tries to use types carefully – the requirements only refer to ducks; superficially, there is no reason to over-generalise the lame duck's `equals` method to take `Object` parameters. Even political correctness does not drive lame ducks to consider themselves equal to *any* object.

One 'correct' implementation of `equals` for lame ducks is:

```
public boolean equals(Object d)
{ if( d instanceof Duck ) return true;
  else return super.equals(d);
}
```

This approach always incurs a run time penalty to perform type checking. It is poignant that in some cases, in principle, no run time checking would have been required at all, as the following alternative approach suggests:

```
public boolean equals(Object d)
{ // we never want this method to be called
  throw new IllegalArgumentException
  ("Lame duck's equals(Object) invoked");
}
public boolean equals(Duck d)
{ return true;
}
```

In summary, we cannot define an `equals` method where its type-incorrect use would be a compile-time error. Since `Object`'s `equals` takes `Object` parameters, we *cannot* get the compiler to detect any occasions where we erroneously compare a duck with, say, a `Vector`. Instead, this programming error has to be treated as a run time error (with further error-prone overheads in handling the propagation of the error at run time).

> "Look what the *computer* did to *me* this time!" is a continual refrain [. . .] we really don't want to accept this responsibility and the easiest way to duck it [*sic*] is to not see the errors when they occur. [. . .] This psychological problem sits at the center of all error detection difficulties in serious computing. We just don't want to think that *we* were wrong. And when an error that trips us up turns out to be a system fault, that merely reinforces our reluctance to accept *any* errors – even tho we weren't checking very well." F. S. Acton [7].

OTHER FEATURES OF JAVA

Strings in Java

The preceding example was about ducks, hardly a serious area of concern for most Java programmers. Our critique is therefore open to the accusation of setting up sitting ducks as an easy target. Let us now look more closely at Java's own use of object orientation, as exemplified in its string handling. We shall argue that strings are handled in an anomalous manner; they are not true objects. Had Java gone through a full design process, the anomalous behaviour of strings might have been refused – alternatively, the same decisions might have been taken, but at least the rationale would have been explicit.

`String`s and `StringBuffer`s are fundamental to many programs, yet the Java classes that implement them are (arguably)[§] incomplete, but they are defined as `final`. As `final` classes they cannot be subclassed, so their incompleteness cannot be rectified. So much for inheritance, when useful classes cannot be inherited from!

For example, `StringBuffer`s do not support deletion directly (except by explicitly constructing a new `StringBuffer` out of two substrings). It is not possible to add a method `deleteChar` without creating a whole new, independent, class.

`StringBuffer.equals` may not do what the programmer wants – two `StringBuffer`s are unequal even if they both represent the same string – and it is not possible to override this meaning of `equals` should you want a sort of `StringBuffer` where `equals` is more conventional. If the `StringBuffer`s are `a` and `b`, testing for equal contents has to be done by

```
a.toString().equals(b.toString())
```

It is likely that one might wish to do this often, and therefore would like to subclass `StringBuffer` so methods can be defined to do it. But as `StringBuffer` is `final` it cannot be subclassed. Since `StringBuffer` cannot be subclassed, the temptation is to write out explicit code for `equals`, as above, each time it is required. Sadly, since `equals` takes an `Object` parameter, the following typos are all compilable code, but *always* return `false`:

```
a.equals(b)
a.toString().equals(b)
a.equals(b.toString())
```

A similar problem arises with hash codes. The hash codes of 'equal' string buffers are unequal: if a program uses the hash codes of string buffers, the programmer has to remember to convert them to strings first (where hash codes work correctly). By their nature, hash codes are probabilistic, so errors with hash codes are tricky to track down.

Although `String`s and `StringBuffer`s are closely related (Java automatically converts between them both in many contexts), they are unrelated classes. Why don't they both implement a common interface?

Concatenation of strings uniquely invokes the method `toString` automatically: in all other contexts in Java method invocation has to be explicit. Thus `"x"+o` is equivalent[¶] to `"x"+o.toString()` if `o` is not a `String` or a basic type. Even passing a non-`String`

[§] Strings have been around for a long time, so the definition of their class should be straightforward.

[¶] What Java does is much more complex. In fact, concatenation is defined for `StringBuffer`s, not for `String`s. Suppose we write `"x"+o`. This is effectively converted to `new StringBuffer("x").append(o).toString()`, with one of `StringBuffer`'s many `append` methods being selected appropriately to match the type of `o`.

actual parameter to a `String` formal parameter does not do this, so `+`'s operands are here doing something more powerful than parameters in any other context. Moreover, the programmer cannot define other methods (e.g. `toInt()`) with this privilege.

The operator `+` is overloaded, and means either addition on numbers, or concatenation of `StringBuffer` objects, with appropriate implicit use of the `toString` and the `StringBuffer` constructors. The alternatives seem to be either

(a) You believe operator overloading is a bad thing, and eliminating it leads to better code (in which case Java should not have overloaded `+`), or
(b) You believe that overloading is a good thing and improves expressiveness (in which case programmers should be able to define overloaded operators; in fact, Java programmers cannot define operators at all).

Is Java hypocritical, supporting overloading only where it suits Java itself?

In everyday use, addition is associative: `1+2+3` means the same whether `1+2` is added first or `2+3` is; that is, `(1+2)+3=1+(2+3)` – so brackets are unnecessary. Java's overloading of `+` loses associativity: `(x+1)+2` and `x+(1+2)` are not the same when `x` is a `String`. The first is '`x12`' and the second is '`x3`' (if `x` is the value of the `String`). Strictly there is no ambiguity because Java defines `+` to be *left* associative, so an unbracketed expression `a+b+c` is taken to mean `(a+b)+c`. (Note that `+` ceases to be associative when it is used for floating point numbers, and naïve programming in any language leads to unnecessary loss of significant digits.) In general, in any expression where `+` is overloaded, using brackets would be well advised.

The final point could be made of any class, not just strings. Suppose we have implemented a `String`-like class, and we realise that there is no need to have different objects represent the same `String`. Thus, when a program writes `new String("x")` we want it to return the same `String` as all other calls to `new String("x")`. In fact, because this is a recognised problem (see the quotes below), Java's own `String` class provides a method, `intern()`, to (almost) do this – but it has to be used *everywhere* a new `String` is constructed. The appropriate behaviour cannot be done centrally by the constructor. In other words, what should be the internal implementation of `String`s has to be explicit everywhere they are used. This is a language design feature that is unnecessary and dangerous because for most *but not all* purposes `new String("x")` and `new String("x").intern()` are equal and indistinguishable. Not only is the problem recognised, but *tricks* are suggested to cope with it – surely a clear indicator of a bad design choice?

> "Using equality operators on `String` objects does not work as expected." p. 133

> "For example, `==` probably works correctly in the following code: [. . .] But be careful – this trick works only if you are sure that all string references involved are references to string literals." p. 166

In summary, Java as a programming language requires strings (and string buffers) to behave in a particular way, so that the language is well-defined. But this requirement for strings is different from the programmer's need of strings in a program. On the one hand, it is useful that the compiler's idea of strings is explicitly defined, and can be defined in Java; on the other hand, the mechanisms in Java, as it is designed, are such that one cannot have the best of both worlds. Java does not provide both a well defined set of string operations for use *in* the language, and an 'open' string class for programmers to specialise *in* their own programs. The programmer comes off worse in Java's trade-off.

Arrays: Java's 'parameterised classes'

Classes in Java have no parameters. For example, if we have a class that defines lists, we cannot use it to help define classes of lists of integers, lists of strings, and so on – where we would be using 'integer' and 'string' as parameters to the list class to define more specialised classes. Instead, Java encourages the programmer to define lists of objects, which can be anything – integers, strings, or whatever. This approach does not stop the programmer accidentally putting an integer in what is supposed to be a string list. Alternatively, a programmer can define separate classes, one for lists of integers, one for lists of strings, and so on, making them specific to the element types. In this case, it is not easy to avoid rewriting all the class definitions from scratch. So, either the programmer has to use over-general code (then Java cannot detect type errors), or has to repeatedly write almost identical code (then there is a risk of typos). Both alternatives are risky, and both make rigorous testing harder.

Java has arrays, and interestingly they are treated a bit like parameterised classes, though with some special notation to make element access and initialisation easier. Arrays are like parameterised classes in the sense that they are collections of objects of specified type, and they are statically type checked. Arrays also 'inherit' a field, `length`, which is the number of elements.

We shall argue that arrays are anomalous (recapitulating the design issues raised over strings).

Arrays are built into Java. Java also provides a standard class `Vector` (that could be defined by any programmer) which behaves much like arrays, except that there is no static type checking that elements inserted into `Vector`s or retrieved from them are type correct. Moreover, programmer-defined general purpose collections (like `Vector`) *cannot* have static type checking: arrays do something special that, whilst similar to anything a programmer can do for themselves, is impossible to do exactly.

We can write an array element assignment like `a[i] = 4`, but if we decide to change `a` from an array to a `Vector`, then each use of `a[i]` in the program has to be examined before it can be changed: `a[i]` on the right side of an assignment has to be converted to `a.elementAt(i).intValue()`, whereas on the left of an assignment it has to be converted to, say, `a.setElementAt(new Integer(4), i)`. It is a shame Java cannot convert explicit assignments to method calls itself. Perhaps Java should have permitted programmers to define methods called `[]`, as is possible in C++.

An array can be initialised by an array initialiser, as in `int a[] = {1,x,3}`, which assigns the array denoted by `{1,x,3}` to the variable `a`. Although this looks like it includes an assignment, Java does *not* permit array literals in any context other than initialisation. Compare `int b = 2` which is equivalent to `int b; b = 2`, with `int a[] = {1,x,3}` which is not equivalent to `int a[]; a = {1,x,3}`, because the latter is illegal by design. It is not possible to use array literals in assignments or pass them as parameters, or to use them in any other expression.

It is easy to convert an illegal (but plausible) array assignment into correct Java. The previous example could be written out as `int a[]; int aa[] = {1,x,3}; a = aa; ...`, where `aa` is a variable the compiler introduces to solve the problem. Although some people might say in that case there is no problem, one wonders why Java could not do something so simple itself especially as the apparently arbitrary restriction makes the language more complex to explain.

Given that Java requires a superclass constructor to come first, how can a constructor call its superclass constructor giving it an array parameter? Not easily, since an array literal cannot be

passed as a parameter. Any array initialisation, say `int a[] = {1,x,3}`, so that `a` could be the constructor's parameter, must be evaluated *after* the constructor is invoked.

To solve the problem, the superclass could provide a method (`init`, say) that is invoked to initialise the object *after* it has been constructed. The code below shows what one would like to do, which is illegal, and two alternatives that are legal but are both less reliable code:

```
class Obj2 extends Obj
{ Obj2(int x) { super({1,x,3}); } // illegal

  Obj2(int x) // 'obvious' solution (note code follows super() call)
  { super();
    int a[] = {1,x,3};
    init(a);
  }

  Obj2(int x) { super(t(x)); } // contrived solution
  private int t(int x)[] { int a[] = {1,x,3}; return a; }
}
```

That the contrived solution successfully gets around Java's restriction that there should be no code before a super constructor begs the question why there needs to be a restriction in the first place. Certainly it proves that a sensible compiler could always generate code to allow array literals in any context, parameters or expressions, where array values are permitted.

Arrays are in Java because they are efficient and frequently useful. When they are compared with programmer-defined classes that 'do the same thing' (such as `Vector`), there seems to be very little correspondence. As a program is developed, it is likely that arrays and vectors be used interchangeably – yet their notations are strikingly different. If a programmer makes the conceptually trivial change from vectors to arrays, or *vice versa*, a lot of code must be rewritten. These differences (which could have been avoided by a different design) plus the curious rules about array literals, seem consistent with arrays being a hack – they provide an *ad hoc* solution when they could have been part of a general and consistent feature of the language.

JAVA AS 'STRONGLY TYPED'

Java is strongly typed, but the type correctness of a Java program is not known at compile time. In other words, Java is not statically typed (like ML). The intention of saying Java is strongly typed is to give the impression of robustness. Type robustness would be achieved by strong *static* typing, where the compiler detects type errors before a program is run. Like Java, both BASIC and LISP are strongly typed, but neither are statically typed; their type systems do not seem to be what the Java hype seems to imply! Our discussion, above, of the duck and lame duck's `equals` method illustrated some problems of Java not being statically typed.

The following example demonstrates Java's lack of strong static typing; it involves creating an `Object` and casting it to a `Character`. This is statically correct (i.e. it compiles without error), but at run time it throws an error (a `ClassCastException`), because `Object`s are not `Character`s. (Sophisticated compilers might detect the problem (though whether they – and they alone – should is an interesting question of compatibility), but they can easily be defeated by passing an `Object` as a parameter to a method expecting a `Character`: the result will be the same.)


```
Character c = (Character) new Object();
// always causes runtime java.lang.ClassCastException
```

Apparently the assignment is not questionable, for

> "Java prevents incompatible assignments by forbidding anything questionable." p. 121

Arithmetic raises issues of typing. A floating point number (i.e. a Java `float`) divided by zero is not a floating point number, yet Java prefers its principle of 'nonstop arithmetic' so no (type) error is raised. Nonstop must be a euphemism for fast.

Although these are simple examples, in practice serious type errors occur when objects are stored in collections such as vectors and hash tables, and are later incorrectly retrieved as objects of different type. Java exacerbates the problem, because there is no way to parameterise classes for particular types. Therefore there is no check that insertion and retrieval of objects is type-correct. A cautious programmer therefore has to define vectors for characters, vectors for integers, and so on – handling each type as a completely new definition. (They *could* subclass the standard generic definitions, but they would suffer risks, including ones like the lame duck's `equals` method.) Providing multiple definitions of essentially the same class encourages unnecessary error, and increases the work required for checking, testing and maintenance. Since doing so is both tedious and error-prone, it is easier to develop generic classes, that operate on `Object` values. As soon as the programmer does this, 'strong typing' becomes academic, because all objects of *any* class are instances of `Object`.

We saw above that attempts to make specifications safer (as in the lame duck's `equals` method) have unfortunate consequences: Java's rules will prefer a more general method, and therefore a programmer's cautious approach is undermined.

Although parameterised classes raise interesting design choices, C++, on which Java is partly based, does have parameterised classes, and there is a wealth of experience with them. Given the poor programming practices that are known to ensue from not having parameterised classes, it is surprising that Java does not offer them.

Java loses some opportunities to perform type checking on correct use of constant values. Had Java had enumerated types (either built-in or programmable using parameterised classes) incorrect use of values could have been detected at compile-time. The Java libraries themselves use constants (`final ints`) a great deal, which risks programmers accidentally passing the wrong *type* of constant. The following is an example, from the Java libraries, that is syntactically valid but effectively type incorrect.

```
new Event(target,when,0,0,Event.F1,Event.SHIFT_MASK,Event.GOT_FOCUS);
```

The parameters (after the first two) are all integers, and in this example they are put in the wrong order, which is acceptable to the type checking. If event identifiers, key names and modifier values were different types it would have made this example a compile-time error, rather than an obscure run-time error. Actually, this particular risk could have been avoided by defining 'enumeration' classes, such as a `Key` class with objects `F1`, `F2` and so on:‖

‖The solution shown achieves the same style of programming as the original, but does so more safely (recall, we are trying to criticise Java on its own terms). There are, however, many alternative approaches, which can be completely hidden in the class definition. First, simply define the `final int` flags to use independent bit patterns. Then adding the flags gives a unique and unambiguous value regardless of the order of the parameters. However, integers are rather too flexible: a programmer might provide unrelated integers (e.g. constants from another class) or accidentally combine them together using some operation other


```
class Key
{ final Key F1 = new Key(), F2 = new Key() ...
  private Key() {} // nobody else can construct Key objects
}
```

Then the `Event` constructor would take `Key` parameters, `Mask` parameters, and so on, and would thus be *effectively* strongly typed. However, as Java does not provide any way to construct parameterised classes (or, more specifically, enumerations), the Java designers presumably preferred the risk of type errors to the tedium of defining lots of small (and difficult to manage) classes like `Key`.

(Readers may note that this example is taken from a now obsolete Java package. That makes another point about the stability of Java; but whether or not the package is obsolete, the point being made about types still stands!)

Exceptions are parts of Java's types, but they are not all compile-time checked. Consider `Enumeration`s: the method `nextElement()` should throw a `NoSuchElementException` if it is called when there is no next element in the enumeration object. Unfortunately, `NoSuchElementException` is a subclass of `RuntimeException`, which the compiler does not check is handled (`RuntimeException`s are so common that no code is required to explicitly check for them). An `Enumeration` method therefore need not throw the exception – even though the `Enumeration` interface definition seems to require it. Interestingly, code give in the *Java Programming Language Book* makes just this mistake (pp. 222–223, 1st ed): their method returns `null` rather than throwing an exception; this problem was corrected in the second edition.

UNNECESSARY CONFUSION...

We've mentioned that Java makes a very inconspicuous distinction between different precisions: `71` is a `32` bit numeral, whereas `7l` is a 64 bit numeral (the first `71` was two digits, the second `7l` was a digit `7` followed by a letter `l`). The problem is compounded further, because Java *automatically* (i.e. paradigmatically) converts from one precision to another, depending on the context. It is therefore very easy to write a program which appears – in the mind of the programmer – to work at one precision, whereas, in fact, it is working at another. Thus, even where semantics are explicit, and there is no 'technical' problem, the programming language notation may encourage certain sorts of human error. Possibly the Java designers decided this was a sensible way of designing the language (though the Java language books written by the language designers themselves explicitly indicate otherwise), but if so, why is the way Java handles the 32/64 bit issue with integers and longs very different from the way it handles the 'same' issue with 32/64 bit floating point numbers?**

Another problem is that a language may be unnecessarily confusing. Java's use of the word `final` is a good example. It has many meanings. A `final` class cannot be subclassed (and, incidentally, that objects of that class can be compiled more efficiently since there is no need for some run time type checks). A `final` method cannot be overridden. A `final` field has a constant value but can be hidden. A `final` parameter has a constant value (though, if it is

than addition. Secondly, a better, object-oriented approach is for flag parameters to take flag sets (where flags are subclasses of flag sets, thus representing singleton sets directly). Each flag set class implements a union method, to combine flags of the correct type together. For a method where particular types and numbers of flags are required, parameters would simply be of the appropriate flag type, thus not permitting flag sets of unspecified size.

**With floats and doubles, *two* letters are available: `f` and `d` (as in `1.8e2f` or `1d`). The `d` is redundant if there is a decimal point, since the default is double precision. Thus the *necessary* single precision `f` suffix unlike the necessary long `l` suffix specifies the *lower* precision number.

an object, being `final` does not stop changing its fields). The *Java Programming Language* book itself seems to get confused:

> "Therefore, [local inner classes] cannot be private, protected, public, static or final, because these modifiers apply only to class members." p. 53

But a local inner class *can* be subclassed, and everywhere else classes *can* be `final` (to stop them being subclassed). Something seems to have gone wrong.

Final fields are constants, and *can* be hidden in a subclass, but a `final` class does not have subclasses, so its fields *cannot* be hidden. Whereas a method can be declared as `final` (so it cannot be overridden) there is no notation to so define a field. Thus, Java requires `String` to be a well-defined class since the compiler relies on it. Therefore, it is a final class. But that means no programmer can subclass `String` to define their own extensions of it. Surely it would have been preferable to allow programmers to subclass `String`, but use `"final"` to restrict what they can hide or override? Unfortunately, not – at least as Java is defined – since a `String` must have a `private` field, and that could in principle be hidden by a subclass.

Java is partly designed for embedded systems, such as consumer electronics. Much of an embedded system will be in read only memory. Java provides no explicit features for using read only memory. A `final` variable may have its value calculated at run time, so it cannot reside in ROM. A peculiar consequence of the `final` rules is illustrated by

```
final int i = 123, j = o.hashCode();

switch( x )
{ case i: // legal
  case j: // illegal
}
```

where, although both variables are `final`, `i` is a constant, and `j` is not; `i` can be used in a `case` statement, `j` cannot. *Apart* from `switch` statements, `i` and `j` can be used interchangeably. In other words, the design of `switch` creates a unique and confusing distinction – but at least it is a barrier and not a trap.

If only `switch` permitted more general case expressions, better programming would have been feasible. Above, we argued for the advantages of defining small classes, such as `Key`, instead of using `final` integers. However, testing object values is much more tedious than testing integer values. Anyone wishing to test values of 'enumerated types' has to use error-prone and difficult-to-read `if` statements: another discouragement to safe programming. Yet Java could have been designed to allow more general `case` expressions such as `case Key.F1` to test on (in this case) object equality. What is `switch` for but to be more readable, and to permit paradigmatic (i.e. hidden, correct and efficient) implementations of tests? The suggested generalisation of `switch` would affect neither criterion, would be upwards compatible, would remove the `final` peculiarity, and would be useful in encouraging better programming practice.

WHAT CAN WE LEARN?

Java is successful, and improving it in an 'objective' sense would be to forget the vast investment programmers have had in learning Java as it is. The conclusion, then, is not that Java should be changed, but that when designing a system, certainly one intended for a world

wide market, one should take – should have taken – great care to explore the design issues. Every minute each Java programmer wastes over an obscure feature is equivalent to the waste of a year of human life.

Although Java is now more robust than when it was first released, why wasn't it released with a test suite? My early attempts at just reading a byte from the keyboard obtained three different results with three different compilers! Java was quickly replaced by Java 1.1 – even though the designers *had* said they believed Java to be a 'mature language, ready for widespread use' [8]. All the revisions 1.1 (and 1.2, and so on) represent lost time to a huge number of programmers who must now learn and re-learn the extensions and variations, as well as the time they will waste recoding existing applications so that they still compile.

The best is the enemy of the good; there is no need to make Java the best it could be, just good enough – which it evidently is, if its popularity is taken as the indicator. But couldn't it *easily* have been a little better? How could the designers have better explored the design trade-offs, and made a better language? The *Java Programming Language* book 'defines' the language, and it was probably written *after* the language was firmed up. However, there are parts of the book (some quoted elsewhere in this paper) that seem to show the designers having second thoughts. A simple example is

> "[. . .] `L` is preferred over `l` because `l` (lowercase `L`) can easily be confused with `1` (the digit 1)." p. 108

But as this easy confusion is known, why design the language so the opportunity for confusion even arises? Indeed, as we pointed out, above, the `L` notation is not required in any case; it is just a confusing and unnecessary hang-over from C.

The notation for numbers should be (but isn't) a trivial design issue. More compelling examples (discussed in the paper, and which would be too tedious to summarise!) are where the book describes traps in the language, such as the purpose of fields, or the tricks (the *book's own* word) necessary to program with `String`s reliably.

Such thoughts, which are clearly expressed in the book – the worries, the long-winded explanations, and so on – should have been picked up as indicating areas of concern. What would have happened if the language design had been written up as carefully *before* the language was finished? What would have happened if the language design had been written up as carefully *concurrently* as the language was being designed? With little effort, the thoughts arising in the explanation would have been able to contribute positively to the design process, rather than being powerless hindsight [9]. Indeed, Wirth (designer of Pascal, Modula, etc.) in many of his programming language designs has explicitly tried to minimise explanations: his view [10] is that lengthy programming language definitions are a sure symptom of inadequacy. It seems likely that some better decisions might have been taken.

The problems we discussed with Java's object orientation arise because Java provides hiding, overriding and overloading, and that these features are combined in *ad hoc* ways, and in programs they combine in surprising and unexpected ways. Part of Java's hype is that it eliminated overloading (we saw it doesn't consistently, but it tried!): taking this good idea more seriously would have avoided many of the problems we illustrated.

The language specification itself is not the only explanation from which one can obtain design insights. Java's lack of parameterised classes means that programmers use `Object` instead of more specific types. This raises the possibility of there needlessly being undetected (and undetectable) errors, and it requires explicit use of casts (to convert `Object`s to more specific classes). Such casts require run time checks, so the code may raise run-time

exceptions – as well as being slow. These are all points that need explaining somewhere in an explanation of how to program in Java. Not surprisingly, the *Java Programming Language* book does not take much space to explain what is missing. Thus, some problems only become apparent when explaining how to program work-arounds in Java, not just when merely explaining the 'raw' use of the features of Java.

Some people reading this paper have commented that there were no amazing revelations, that the paper makes just a summary of weak spots in Java. They argue that some sort of engineering compromises must have been made in the design of Java, and in any case the Java community is actively discussing fixes to some of the problems discussed here. Surely, this paper argues back, if the problems are so well known, we should be even more amazed that they are there at all? It seems to be a rather regular occurrence in computing that failures in the current system are *excused* by the expectation that the next version will fix them. Thinking like this is typical of the fashion victim, and fashion is what much of the hype around Java celebrates. Tomorrow never comes, the fixes come with new features that still need fixing. Serious programmers should not have to rely on upgrades and fixes for problems that were understood in the first place.

In the early stages of a technology, consumers want more performance because the technology does not work well enough [11,12]. After a transition point, when the technology has become good enough, consumers want quality – and the technology can deliver it because it is good enough to fulfill the requirements. However manufacturers may find it easier to promote technology for its own sake, rather than solutions. Instead of moving past the transition point, it is easier to raise the stakes (e.g. using marketing initiatives) to encourage users to have higher performance thresholds rather than to deliver quality. Thus, the transition to quality recedes into the future and technology may never attain the raised requirements. The consumer becomes caught up in an upgrade race. They upgrade hardware, upgrade software, then upgrade again. From one point of view, when millions of people download upgrades to Java this is evidence of Java's popularity, but from another point of view this represents a lot of programmers still hoping for a better solution.

Java is history; people need stability to do anything. What can be done to design future systems better, even systems written in Java? Make writing the specification (or the user manuals and other documentation) much easier – in fact, automate it – so that it isn't put off until it is a retrospective, with no influence on the finished design. We need to find ways to make it easy to write the documentation early in the design process (e.g. by using appropriate tools), and doing so must be a central part of any design project. There are many ways to do this – literate programming is one [13]. It is a shame Java's own documentation system (so-called 'doc comments'), while better than nothing, hardly helps Java programmers adopt much better practice themselves in the systems they design.

Java is widely used, and this in itself is sufficient reason to teach it. It also has useful features, that in themselves are useful to learn, such as concurrency and object orientation; the convenience of these being available in a single language also make it a good choice of teaching language. This paper has shown it has another feature, which should be taught explicitly: Java can be used to teach programmers and programming language designers to be careful.

CONCLUSIONS

Good programming requires using a good language. The way to understand a language is a good indicator of how well it is designed; ideally, one should be able to learn incrementally,

building constructively on past learning. Simple things should be easy, complex things should not conflict with simple things. But with Java, one is always having to revise one's 'knowledge' of it as more is learnt.

The problem with a complex language like Java is that so much is unsaid. Sometimes this results in clearer and more compact programs. They don't need to mention garbage collection, and they can't get it wrong. But sometimes it leads to incredible but hidden complexity – such as the obscure rules for inheritance.

If we regret some aspects of Java's design, then we must ensure that future designers take better account of good design practice. Many Java programmers believe Java is a great success. Yet their programs are usually written in a Java-like subset of C. They surely gain by not having the risks of pointers and unchecked array subscripting. Thus, we cannot conclude it is just the design of Java to blame: much of the poor quality of programming (including the rough-and-ready implementations of Java and its packages) is due to lack of programming skill. 'Proper' computing science, including human computer interaction and software engineering, have been taught long enough to be well known; it is now time well-trained designers and programmers start to raise standards. They need to be taught not just what is, but what could be.

We started with a quote from Tony Hoare, and we end with one from the same 1980 Turing Award lecture [2]: "I conclude that there are two ways of constructing a software design: One way is to make it so simple there are *obviously* no deficiencies and the other way is to make it so complicated that there are no *obvious* deficiencies." The concern about Java is that it is a third way: it *looks* simple yet is complicated enough to conceal *obvious* deficiencies.

ACKNOWLEDGEMENTS

The referees made extensive comments which helped clarify many points, and it is really a shame that they were anonymous and cannot be acknowledged by name. The author is very grateful to Richard Bornat, Paul Cairns, George Coulouris, Jean Dollimore, Tony Hoare, Matt Jones, Gary Marsden, Hani Naguib, Bernard Sufrin and Ian Witten for their helpful comments. This project was supported by EPSRC grant GR/K79376. On the insistence of the ducks, we did not discuss any details of running or cloning.

It is possible that some readers of this paper will obtain different results on their Java systems. Whoever is 'really' right, the conclusion should be that some Java compilers are confused. And that confusion is mostly to do with Java's complexity.

REFERENCES

1. K. Arnold and J. Gosling, *The Java Programming Language*, Addison-Wesley, 2nd. ed., 1998.
2. C. A. R. Hoare, 'The Emperor's Old Clothes', 1980 Turing Award Lecture, reprinted in *ACM Turing Award Lectures*, ACM Press, 1987, pp. 143–161.
3. M. Abadi and L. Cardelli, *A Theory of Objects*, Springer-Verlag, 1996.
4. R. D. Tennent, *Principles of Programming Languages*, Prentice-Hall, 1981.
5. B. Stroustrup, *The Design and Evolution of C++*, Addison-Wesley, 1994.
6. H. W. Thimbleby, *User Interface Design*, Addison-Wesley, 1990.
7. F. S. Acton, *Real Computing Made Real*, Princeton University Press, 1996.
8. J. Gosling, B. Joy and G. Steele, *The Java Language Specification*, Addison-Wesley, 1996.
9. H. Thimbleby, 'Design aloud: A designer-centred design method', *HCI Letters*, **1**(1), 45–50, (1998).
10. N. Wirth, 'From Modula to Oberon', *Software—Practice & Experience*, **18**(7), 661–670, (1988).

11. C. M. Christensen, *The Innovator's Dilemma: When Technologies Cause Great Firms to Fail*, Harvard Business School Press, 1997.
12. D. A. Norman, *The Invisible Computer*, MIT Press, 1998.
13. D. E. Knuth, *Literate Programming*, CSLI Lecture Notes, **27**, Stanford: Center for the Study of Language and Information, 1992.

