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

School of Mathematical and Computing Sciences
Computer Science

PO Box 600
Wellington
New Zealand

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

Learning Java: Promises and Pitfalls

Robert Biddle and Ewan Tempero

Technical Report CS-TR-97/2
May 1997

Abstract

In this paper we will examine the Java language, and consider how easy it is for beginning programmers to learn . We address primarily the issues that arise directly from the language itself, and discuss whether the promises are compromised by the pitfalls. This analysis is the result of our teaching of Java to people in industry, our consideration of whether Java is suitable as a first programming language for university students, and our earlier work involving similar issues with regard to C++.

Publishing Information

This was originally published as: Robert Biddle and Ewan Tempero, "Learning Java: Promises and Pitfalls" In *Proceedings of the 14th Uniforum NZ Conference*, Uniforum NZ, 1997.

Learning Java: Promises and Pitfalls

Robert Biddle and Ewan Tempero
Computer Science
Victoria University of Wellington
{Robert.Biddle,Ewan.Tempero}@comp.vuw.ac.nz
http://www.comp.vuw.ac.nz/

Abstract

In this paper we will examine the Java language, and consider how easy it is for beginning programmers to learn . We address primarily the issues that arise directly from the language itself, and discuss whether the promises are compromised by the pitfalls. This analysis is the result of our teaching of Java to people in industry, our consideration of whether Java is suitable as a first programming language for university students, and our earlier work involving similar issues with regard to C++.

1 Introduction

The introduction of Java [AG96] to the programming world has meant that members of that community (once again) find themselves wondering what this new language can do for them, and whether it worth their learning it. Perhaps more than most new languages, Java is being considered for and by beginning programmers. For example, web designers with little programming experience are looking at Java as presenting a way to add dynamic and interactive elements to their web pages. On the other hand, Java is being considered by educators like us as a reasonable language for students beginning computer science degrees.

In this paper, we discuss the Java language as a language for beginning programmers, examining how the language features affect learning and teaching how to program. Our discussion is based on our examination of the language, our experience in teaching Java to industry professionals, and our experience in teaching other languages, notably C, Pascal, and especially C++ [BT94].

We are considering Java as the introductory programming language for our first year courses in computer science at our university. This paper documents a number of issues that have arisen in our discussions that concern the language itself. Our primary interest is in whether Java is a good language to teach beginning programmers, and so we are looking to see how easy the language is to learn, and whether the language supports principles that will serve the learners well in their future programming. We will compare Java to Pascal, C, and especially C++, and so we believe most of what we have to say will also be of interest to programmers familiar with those languages who are considering learning Java.

The rest of the paper is organised as follows: we begin by briefly presenting the promises of Java, as outlined by advocates of the language. We then discuss the details of the language itself, first the detail features for programming-in-the-small, and then the structural features more important for larger programs. We then outline several related issues extrinsic to the language, and finally address whether Java meets its promises relevant to beginning programmers.

2 Promises

The Java overview whitepaper [Sun94] summarises Java as:

> A simple, object-oriented, network-savvy, interpreted, robust, secure, architecture neutral, portable, high-performance, multithreaded, dynamic language.

Of these features, the most relevant to teaching Java to beginners is *simple*, *object-oriented*, and *robust*. The other features may impact *how* we teach Java, or even whether we teach Java, but are not directly related to how easy it is to teach (or learn) Java. We discuss these other issues in section 5.

2.1 Simple

The Java whitepaper [GM96] had this to say about why *simple* is important:

> Primary characteristics of Java include a simple language that can be programmed without extensive programmer training while being attuned to current software practices.

Reducing the time it takes to be effective in a language is important to all those associated with programmer training. Computer Science degrees can spend more time on actual computer science, rather than on basic skills; industry can get productive programmers more quickly. Java is based primarily on C++, a language that is not regarded as simple by any standards. However Java can be described as "C++ without the hard bits". The bits that have been left out include:

- Typedefs, Defines, Preprocessor directives — The C++ preprocessor can cause a seemingly innocent piece of code to behave in a very bizarre fashion, causing much frustration for the unwary beginner. Even Stroustrup would like to see `Cpp` abolished [Str94], but the design rules under which he was operating did not allow it.
- Functions, Structures, Unions, Enumerated types — Java removes the need for all of these by placing everything in a class. As a consequence, there are fewer features that a new programmer has to learn, thus simplying the learning process.
- Multiple Inheritance — Using multiple inheritance effectively requires a good understanding on what it is useful for, and so is not for beginners.
- Goto Statements — It is generally accepted that `goto` isn't needed and so Java removes it on the theory that it is one less thing that learners have to learn.
- Operator Overloading — The whitepaper comments "Eliminating operator overloading leads to great simplification of code."
- Pointers — Java does not allow any pointer arithmetic, and in fact hides the fact that pointers even exist. Thus beginners are spared from some of the more common sources of errors.

2.2 Object-oriented

In this paper we do not wish to address the issue of whether the object-oriented paradigm is the best approach for beginners or professionals. However, there is evidence that more beginners are learning to program this way: Levy [Lev95] presents a survey, and there are continuing efforts focussed on introducing the object-oriented paradigm to beginners [BMC96]. In *this* paper we will assume that it is at least reasonable for beginners to learn how to program in an object-oriented language.

From the teaching perspective, it is important that the object model be presented as "cleanly" as possible, that is, without any exceptions to the rule or distractions, otherwise the advantage of the principle under discussion will be lost.

Java provides forms of the expected features:

Classes and objects: Java allows the definition and instantiation of classes. In fact, this is about all that possible.

Encapsulation: Java allows the programmer to choose which parts of an object are accessible by other objects.

Inheritance: Java provides two ways of creating new classes based on old ones: the standard composition and the object-oriented inheritance. Inheritance is used to create "subclasses" of existing "superclasses". Java does restrict the use of inheritance so that it can only be applied to non "final" types.

Polymorphism: Java allows objects belonging to a subclass to be used where a superclass is expected.

Dynamic binding: When polymorphism is being used, the decision as to exactly what code is executed is delayed until run-time.

We discuss how well Java supports the principles of object-oriented programming later.

2.3 Robust

Inexperienced programmers produce programs that behave strangely because they do not completely understand what they are doing. Their understanding improves as they learn how what they are doing causes the behaviour they are seeing. The easier it is to see how the behaviour relates to what they do, the easier it is for them to learn. A good way to directly relate bad behaviour to the code responsible is to have the development system assist by warning about such code.

Java tries to improve the robustness of code produced over C++, and in doing so explicitly flags a number of common sources of programmer error. It improves robustness in two ways: improved type security and automatic memory management.

Java has generally tightened up how types work. It disallows some automatic type conversations that can take place in C++. A `boolean` type has been introduced and the `if` statement conditional must return a `boolean` value, rather than base the condition on a zero/non-zero value. It has also changed the meaning of "cast", so that instead of meaning "treat this sequence of bits as a value of this type", it means "insert a run-time type check to ensure this value is of this type". This means that the class of errors associated with incorrect casts at least produces a run-time error, rather than just unexpected behaviour.

Java removes control of memory management from the programmer through the introduction of an automatic memory management system (a "garbage collector"). This means beginning programmers do not have to learn about how memory is managed, thus simplifying the learning process. It also means that a large class of errors associated with pointers, such as "dangling pointers", is changed from silently producing weird program behavior, to raising a null-pointer exception.

3 Language Issues: Detail Level

In this section we will look at the detail level of the Java language, discussing elements of the language involving data and control within a method. This level of the language is quite significant for beginning programmers, because beginners really need to understand the fundamentals of data and control to understand how computing works.

3.1 Primitive Data Types and Operations

The primitive types of Java are similar to those of C. The numeric types are `int`, and `float`, and as with C there are several alternatives to these for greater and lesser precision. Some languages for beginners even avoid distinguishing integers and floating point numbers, but we are happy to make the distinction: some tricky issues of conversion arise otherwise, and it allows us to highlight the imprecision of floating point arithmetic.

The numeric operations also look like those of C, and for example include pre and post increment and decrement, as well as operations combined with assignment (e.g. `x += y` to add `y` to `x`). For beginners this is probably a richer set of opportunities than is desirable, but it need not all be introduced right away. Also like C, Java includes a range of bit oriented operations that work with numeric data. This can undermine the type concept, but beginners need not be introduced to bitwise operations at all.

While the types and syntax of type declaration initially look like C, there are also key differences. One is that the character type `char` is not a numeric type, another is that Java has and uses an explicit `boolean` type, whereas C programmers typically use integers for boolean values. The C practice undermines the important lessons of type safety and type checking, and so Java is an improvement.

The boolean operations have the same syntax as C, but the big difference is that the operations truly are boolean, and so return values of `true` or `false`. In C the operations return numeric values of zero or non-zero. The Java approach will be easier to explain, and better support understanding of types and type safety.

3.2 Statements and Control Flow

Like C, Java has no assignment statement. Instead, it has an assignment operator, and an expression is a valid statement. Beginners do not really have to appreciate this, however, and a simple assignment is typically treated as a common statement. It does mean that assignments in mid-expression are allowed (e.g. `x = b + (c = 5);`) but these do not have to be introduced early.

Beginning C programmers often become aware of these issues in the worst way, by confusing the assignment operator `=` with the boolean equality `==` operator. An `if` statement with the condition `(x = 0)` performs not a comparison but an assignment, and the value of the condition then depends on whether the assignment produced a zero or non-zero value. Beginners in Java make the same mistake, but in Java because the boolean operators return boolean values, the `if` statement requires a boolean value as well. Accordingly, this common mistake results in a compile-time error in Java, instead of the more problematic run-time incorrect behaviour that results in C and C++. (However, this can still happen when the assignment expression produces a boolean type!)

The Java statements for selection and iteration are almost exactly the same as those in C. The selection statements are the `if` and the `switch` statements. The `if` statement has the advantage of the boolean condition as described above. However, the beginning programmer must still remember that the two parts controlled by an `if` statement must just be single statements, unless the a sequence of statements is enclosed in braces so making a single block. The `switch` statement must still turn on a simple value, and cannot use more complex types. Moreover, Java retains the odd characteristic of the C `switch` statement: that cases drop through from one to the next unless stopped by a `break` statement.

The Java statements for iteration are the `while`, the `do`, and the `for` statement as in C. They all have the advantage of working with boolean conditions, and retain the disadvantage of requiring blocks or single statements. For beginners, the Java (or C) `for` statement is probably more difficult to understand than the Pascal `for` statement, but it has the advantage of allowing generality later on. Like C, Java allows control over iteration with `break` (immediate end of loop) and `continue` (immediate end of one iteration), but Java adds label matching to allow multi-level escape — probably not something to introduce early to beginners. Java has no direct branch statement (like C's `goto`) at all, but we won't miss it.

3.3 Using Objects

In the Java language there are important distinctions between primitive data, discussed above, and object data. We leave until a later section our discussion of creating classes. However, using objects is something relevant to the detail level of programming. Beginners need to use objects even at an early stage to access the world beyond their program. The library facilities, whether for input/output or mathematical functions, always involves using classes and objects. Moreover, there are pedagogical reasons for beginners to get experience in using objects before they go on to create their own classes.

Simple use of pre-existing classes and objects is reasonably straightforward, and uses a syntax similar to C++. Of course, beginners will have to be introduced to the underlying concepts, such as classes, methods, and fields. Moreover, it will also be necessary to explain that some methods can be used for a class, whereas others apply only to individual objects. Java uses the C++ term `static` for class-wide methods, which is not as helpful a term as might have been chosen.

To use a pre-existing class to declare and use objects of that class requires beginners understand in more detail the differences between primitive types and object types. The key difference is that a primitive type always has value semantics, and an object type always has reference semantics. There are two kinds of variables in Java: class fields, and local variables. Primitive type variables can be initialised or assigned from a literal value, or from data of the same type, and a *copy* is made. Object type variables can be initialised or assigned from a newly created object of the type, or from an existing object of the same type. However, in initialisation or assignment of objects, the variable does not receive a copy, but a *reference*. A variable of object type is a *reference* to an object. Assigning one such variable to another results in two variables that reference the same object, so that changes to one also affect the other.

Object variables can also be initialised or assigned from the special value `null`, and at this point the beginner must really understand the idea of reference semantics. In many ways, understanding object references is very similar to understanding pointers. As a result, although Java is claimed to be simpler than C++ because of the absence of *explicit* pointers, it is unfortunately true that it is more complex at initial stages because of the importance of the *implicit* pointers.

Understanding reference assignment also means understanding that when an object variable is assigned to, the original referenced object value is no longer accessible via that variable. This in turn means understanding that when that variable had been the *only* reference to the object value, the object value is now inaccessible. And this, at quite an early stage, may lead to the need to explain "garbage collection".

While Java looks at first more like C++ than anything else, this is one of the major differences. C++ object data works with value semantics like primitive data does, but C++ also has pointer types and dynamic allocation that allow reference semantics. Java has no explicit pointer types, and so avoids some difficult programming issues that arise in using C++. However, the exclusive use of reference semantics for object data means that some complex issues must be tackled at an early point by beginners. Moreover, the beginner must also keep in mind the there are two kinds of data, and that they work differently.

3.4 Arrays and Strings

Another reason that beginners must cope with understanding object data and reference semantics is that in Java both arrays and strings are objects. In both cases, there is special syntax involved, but understanding the reference semantics is still important. Moreover, strings and arrays are very common in programming, and cannot easily be avoided or postponed.

Arrays follow the basic syntax of C, and like in C are always indexed from zero. We do not ourselves believe this is a problem, although some educators prefer the Pascal approach where array indexes are subranges of the integers, which allows the matching subrange to be used for data involved in indexing the array.

While the syntax for using arrays does resemble C, there is an important difference. In C, array indexing is syntactic sugar disguising pointer dereferencing, and rigorous checks on array use are difficult. In Java, array use is syntactic sugar disguising object use, and checking is more more straightforward. For example, all array references are always bounds checked — this is a significant improvement for beginners.

The declaration of an array requires understanding of objects, and involves initialising with the `new` operator and the desired size of the array (e.g. `int[] marks = new int[10]`). While it may be a good thing that the syntax suggests the object nature, it is unfortunate that such a fundamental data construct requires such complication.

As objects, arrays can be assigned to each other, and of course it is reference assignment — which may lead to surprises as changes to one array will of course also change the other. Arrays are assignment compatible if their base type is the same, so arrays of different size can be assigned to each other. This is different from Pascal, for example, where arrays must be of the same type and dimension to be compatible. However, the comparison isn't straightforward because array assignment in Pascal involves copying the entire array. (Moreover some Pascal variants have loosened this rule.) While Pascal's approach does emphasise type safety, it does make it difficult to create reusable code. C gets around this problem relying on the pointer nature of arrays. The Java approach is probably as reasonable as can be achieved.

The situation of strings is broadly similar to that of arrays. However, unlike in C or Pascal, Strings in Java are not arrays. They are objects, albeit with special syntax. However, using them in a straightforward way is easy, and the literal string syntax resembles that of C. But of course variables of type `String` are *references* to objects, with the potential for problems. Perhaps the worst problem is that the equality operator `==` tests for the equality of the references — a pitfall almost as dangerous as the C `=` vs. `==` confusion. The `==` operator tests for reference equality for all objects, but this is likely to be especially problematic in the case of strings, which may seem more like primitive types, especially to beginners.

One positive aspect of the object nature of arrays and strings is that extra operations on arrays and strings are easily provided using the object method syntax. For example, finding the size of an array `myarray` is easy (use the field `myarray.length`), and to test whether two strings are the same one can use the method provided (e.g. `name1.equals(name2)`).

3.5 Summary of Detail Level Issues

The detail level of the Java language resembles C or C++ strongly. In doing so, it is conservative in supporting accepted and successful basics of imperative programming. The syntax resembles that of C even in some places where it seems weak (e.g. statement blocks, switch statement flow-through). However, there are important ways in which Java is superior, and addresses problems of C of particular difficulty to beginners (e.g boolean types, array bounds checking).

A major issue involves object data and reference semantics. Here Java again breaks with the C or C++ approach and avoids explicit pointers. However, the implicit pointers that it does involve are more pervasive, though less dangerous, and this complexity is a potential problem for beginners understanding the language.

4 Language Issues: Structural Level

4.1 Packages and Classes

The top structural element of a Java program is the package. A package is basically a set of classes, and has some reasonable advantages. Pragmatically, it allows the programmer freedom in naming when creating new code, without fear of making library or other programmer's code difficult to access. Moreover, the package structure also allows a high level partitioning that supports management of larger programs — although to really be effective this needs support from the development environment. Neither of these is a critical concern for beginners, but they are reasonable things for beginners to grow into, and Java makes this easy because a nameless package exists that beginners do not have to be aware of to use. There *is* a problem for beginners in using packages, but it concerns access control in classes, and is discussed below.

The principal structural element of a Java program is the class. The class is the main element of OO program design, allowing instances to be created and used, with state and behaviour wrapped up and accessible via an interface. The Java class strongly resembles the C++ class, but there are some important differences.

A Java class consists of fields and methods, which C++ would call data members and member functions. Like C++, each field and method can be designated `public` (access allowed to any class user), `private` (access allowed only to class methods), or `protected` (access allowed only to class methods or inheriting class methods). All this is reasonable, and properly emphasises the importance of encapsulation — and the more complicated concepts involved with `protected` do not need to be introduced right away.

However, there are two bothersome problems that do affect beginners. Firstly, whereas in C++ the absence of an explicit access control implies `private`, in Java it implies a nameless fourth access control category. This category allows access throughout the package, but denies access beyond the package. While such an access category is not itself unreasonable, it does create a problem for beginners. Because it is the default, it is the result if the programmer forgets to specify access control. And because beginners typically work with smaller programs that consist of classes within a single nameless package, the effect is that forgetting access control yields open access. So in this way, Java fails to reinforce its support for encapsulation, and beginners can easily miss the point.

The second problem with Java access control is more superficial, but also involves a failure to support the concept of encapsulation. In Java, the class declaration involves fields and methods, as with C++. However, in C++ the class can specify only the function prototypes for the methods, and the function implementations can be specified elsewhere. By grouping all `public` prototypes, a C++ class makes the interface to the class apparent, and the distinction between the interface and the implementation is made clear to beginners. A Java class, however, requires method implementations to be given in detail in the class declaration. As a result, it is no trivial matter to look at a Java class declaration and see what the interface is. For beginners, this is unfortunate, and the important idea about separation of interface and implementation is undermined. (The C++ class declaration must also contain the `private` details too — partly because of C++ value semantics — so C++ is hardly perfect in this regard either.)

Java supports exceptions and exception handling in a similar way to recent additions to C++, and features similar structures for `throw` and `catch`. Which exceptions methods throw is part of their interface, and this is another reason why making it difficult to clearly see the interface is problematic. Worse still is that we regard exception handling as a complicated topic, certainly unsuitable for very new programmers. Yet the Java style, and standard elements of the Java run-time environment, extensively uses exceptions that the programmer must handle. We discuss this problem further in section 5.

4.2 Class Methods

Like C++, Java supports "constructor" methods to initialise instances of the class, and they work in a similar way. However, there are several advantages with Java. Firstly, in C++ constructors are implicitly called whenever an instance is declared or dynamically created; in Java they are only called on dynamic creation. One of the problems in C++ is that beginners often forget that declaration involves a constructor call, but this is less common with dynamic creation. Moreover, because of Java's reference semantics, there is no need for the C++ copy-constructor, a common source of nightmares for C++ learners. And lastly, because of Java's garbage collection, there is no need for the C++ "destructors". Java does support a `finalize` method to be called for instances about to be garbage collected, but it is not necessary for dynamic memory management the way a C++ destructor is, and so beginners need not be introduced to these complexities.

Java methods resemble C++ member functions, and seem reasonable to understand. Like C and C++, Java does not distinguish functions and procedures, but has a `void` quasi-type to indicate the function does not return a value — beginners should have no real difficulty with this. There are some significant differences between Java and C++ concerning data. The way in which Java methods are passed parameters reflects assignment: primitive types are passed by value, and object types are passed by reference. This does mean that beginners who understand assignment should have little difficulty understanding parameter passing. However, it doesn't support a greater understanding of the possibilities of parameter passing that might be useful. Moreover, the restricted options mean that programmers will have difficulty if they need a method to return a primitive type other than via the method return value, or if they need to be sure a method will not change the value of an object parameter.

Another issue that may be problematic for beginners concerns a difference between class fields and local variables inside methods. Fields are initialised by default, primitive types to various "zero" values, and object types to `null`. Local variables, however, are not initialised by default at all. While there are some language implementation reasons for this distinction, it makes one more thing of which beginners must be aware.

4.3 Class Relationships

The most important way in which classes are used together in Java is by composition, whereby one class has a field that is of another class. In this way classes can be built using other classes, and this use can be a hidden implementation detail. As with C++ and most languages, this important relationship is not highlighted or explicit in any way, and it can be difficult for beginners to realise its great significance.

The relationship that is typically more celebrated and explicitly acknowledged is inheritance. Java supports inheritance in a similar way to C++. For example, inheriting classes automatically have the same fields and methods as the class they inherit from, but do not have access to `private` fields or methods. Java makes a helpful simplification by only supporting single inheritance, and not multiple inheritance. This avoids some difficult problems involving name resolution and data sharing: indeed there was controversy when multiple inheritance was being considered for C++ (see the papers by Waldo and by Cargill, for example [Wal94]). C++ eventually did include multiple inheritance, but Java probably does the right thing for beginning programmers by avoiding it.

Like C++, Java specifically acknowledges `abstract` inheritance, where the superclass cannot have any instances, but can only be used for inheriting from.

As well as inheritance, Java includes something similar involving the concept of "interface". An `interface` is like a class, but only has interface specifications. An interface can be used as a type specifier, and a class can be designated as *implementing* an interface, whereupon instances of that class conform to the type specification. This is very like inheritance, and indeed our approach to teaching about inheritance is based on emphasising interface conformance [BT96]. Whereas Java limits inheritance to single inheritance, it allows a class to conform to multiple interfaces. This allows something very similar to multiple inheritance, although it is less problematic because interfaces do not themselves have any data fields.

The Java support for inheritance — and interfaces — seems reasonable and justifiable. However, for beginners it has the problem that there are a number of concepts that must be understood before seeing how it all works together. In some ways it is simpler than with C++, but in other ways more complex, and great care is needed in introducing these relationships to beginning programmers.

Another important technique in structuring programs concerns genericity, especially the ability to parameterise a container class with respect to the type of what it contains. In this way, for example, one queue class can be used for a queue of events and a queue of characters. In C++ this is explicitly supported by the `template` construct. Java has no equivalent, and adopts the practice of many other objected-oriented languages in using inheritance to accomplish genericity.

Using inheritance to accomplish genericity is done by writing the container class so that the type of what it contains is a superclass from which other classes can inherit. For example, a `queue` class can be written to contain `queue_elements`, and then both character and event can be created to inherit from `queue_element`. In this way the queue can be used for either characters or events. The problem with this approach is that it can be used for *both* at the same time. With a generic container (such as a C++ template class) this kind of error can be detected at compile time, but the inheritance approach requires run-time checks. Of course, run-time checks can only detect such errors when they actually arise, and so programs may contain latent errors of this kind that may remain undetected despite testing. Another problem with using inheritance for genericity is that it can be difficult to arrange an inheritance structure so that the container will work for all types that may be necessary. Java provides a special class `Object` which is an unstated superclass of every class. This solves some problems, but not others; we consider this matter further in section 5.

While supporting genericity via inheritance does limit the concepts that a beginning programmer must learn, we feel this advantage came at a significant cost. Not only does it require run-time checks for type safety, and introduce difficulties in finding superclasses for effective generic containers, but the whole approach muddles two different ideas and programming strategies. We feel that the Java approach is less helpful to beginners than is desirable.

4.4 Summary of Structural Level Issues

As with the detail level, the main features Java offers for the larger structures in a program strongly resemble those of C++. And, as with the detail level, there are a number of ways in which Java is different to C++ that will help beginners. For example, object construction and destruction is simpler — a very positive change — and inheritance is limited to simple single inheritance. On the other hand, some of the ways in which Java is different to C++ make life more difficult for beginners: for example, the class interface is not so clear, access control is harder to work out, and as well as inheritance, there is the interface concept to master.

5 Extrinsic Factors

Our primary concerns as educators involve principles of computer science, and we have focussed on the specifics of the language because they will either help or hinder our efforts in explaining the principles. However, factors extrinsic to the language itself often play a big part in the learning experience for the student. Such factors include the library support, documentation, learning materials, development environment, and availability, and they are often interrelated.

In the case of Java, the Java designers have made a lot of effort to reduce the impact of these factors. By declaring (and providing) a standard library that provides a significant amount of the functionality required by most programmers, they have reduced the incompatibilities that arise through different libraries (Stroustrup regards the early lack of a standard library as the biggest mistake he made [Str94]). By making Java a byte-code interpreted language, they have increased its portability, since it is easier to port to multiple platforms, and so its availability. And by making the Java Development Kit (JDK) freely available, as well as increasing Java's availability, they have ensured a large population of potential developers, which has in turn created a large demand for books that a number of authors are trying to meet.

The impact of all of this on the beginning student has generally been positive. For example, while we do not intend to have beginners write multi-threaded, or network aware applications from scratch, the existence of the Java standard libraries does make it easier for *us* to produce such applications to demonstrate such concepts, and make available to the students to modify, thus increasing the learning experience for the student.

The availability of JDK means that students can have a cheap (albeit minimal) development environment on their home machines, which is beneficial for both those with home machines and those without, since it frees up seats in the department labs.

Despite all of these benefits, there are nevertheless some aggravating problems.

An unexpected problem comes from the requirement that most kinds of exceptions must be either explicitly caught, or explicitly passed through to any caller. Because a number of methods in the library throw exceptions, it is very difficult to write anything but the most simple of applications and not have to deal with exceptions. For example, a common early assignment is to read input and manipulate it in some way. Almost any kind of use of an input stream may throw an exception, and in some cases (such as when reading numbers) possibly more than one kind (from the input stream itself, and from the method that parses numbers). The exception mechanism is not something we want to talk about early in the course, but the design of both Java exceptions and the Java standard libraries makes this difficult. Our solution has been to wrap up such code in our own classes, but that somewhat defeats the lesson of "standard libraries".

As mentioned earlier, Java does not have template types as in C++, instead genericity is achieved by the use of the class `Object`. Thus the standard container classes of `Vector` and `Stack` allow values of any object type to be mixed freely. The usual convention when using these classes is to cast the value to the expected type when getting it out of these containers. This prevents weird behavior because the cast does a run-time check — it also takes time. It could be argued that it does not take *much* time, however it does mean that if values of the wrong type are put into the container, this error will not be caught until run-time. As commented earlier, it is always easier for beginners to learn if their errors are made explicit, and the earlier the better. But probably the most aggravating consequence of the lack of template types is the fact that beginners must now be exposed to the concept of *down-casting* if they want to use the container classes that are part of the Java standard libraries.

Most container classes have a method such as `contains`, which determines whether the specified value is in the container. In order for this to work, the values placed in the container must be able to be compared for equality. Recall that `==` tests for *reference* equality, not *value* equality. This means that classes such as `Vector` assume that there is a method on `Object` named `equals`. This method does not do anything interesting, it is expected that any actual value placed in a `Vector` will override `equals` to provide the correct behaviour. This causes two problems for beginners.

First, beginners often forget to override `equals`, but, because `Object` provides it, they get no warning as to what they have done — their program compiles fine but does not execute as they expect.

Second, if programmers do remember to create their own `equals`, they often prefer it to look like:

```
class Myclass {
  public boolean equals(Myclass e) {
  // ....
  }
}
```

but here Java's overloading mechanism regards this as a *different* operation than the one expected by `Vector`; it requires the argument to be of type `Object`. Understanding why it has to be this way (given the choices Java has made about generic types) is more than we normally expect from beginners.

The requirement that container types must take values of object types raises another problem. Recall that Java primitive types are *not* object types, and so they can not be used in the standard container classes. This is a serious problem for so-called reusable code. So, what the Java designers have had to do is provide ways to get "object" versions of primitive types. Thus, there is an `Integer` type, and `Boolean` type, and so on.

This solves the problem, but raises a troublesome issue: can the values of these objects be changed? With reference semantics, if the method body calls an operation that changes the state of the actual parameter, this change is reflected back to the calling context. This seems like a reasonable way to return integers, if it is not practical to return an integer as the return type. However, the object types provide by the Java standard libraries do not allow their value to be changed, something that beginners see as a perfectly reasonable thing to want to be able to do once they understand reference semantics. While it could be argued that the Java designers have made the right decision, it does mean the beginners often have face this before they are really ready to appreciate the argument.

Another problem is, while the libraries do have some documentation, it quite often is very terse to the point of being ambiguous. As a simple example, Java uses the 16 bit Unicode standard for its character type, however some uses of the word "character" in the library documentation actually refer to the 8-bit variant. The domain and range of many of the mathematical functions are often not specified. The documentation (or lack thereof) becomes more of a problem with the Java Abstract Window Toolkit (AWT), which provides GUI management. There a number of very subtle interactions between the different parts of the toolkit and are generally not documented at all. While this has spawned a growth industry in books on the Java AWT, none of them are (so far) complete in their treatment, and so it is often necessary to look through several to understand what is going on. While we do not expect beginners to have to deal with AWT, this is a symptom of the general problem with the Java standard libraries.

As mentioned earlier, there are a number of authors trying to fill the demand for information on Java, as is evidenced by the length of shelving devoted to Java books. However, until very recently, they all were targeted at people who already knew how to program, not the beginner. Even now there are only a few books targeted at students, however our experience is that it will probably be some time before there are any of good quality.

5.1 Summary of Extrinsic Issues

Java comes with a fairly complete and standard set of useful classes. Unfortunately, introducing them to beginners will result in exposing advanced features very early. Nevertheless, the existence of these classes, the fact that they are part of the standard definition of Java, and the fact that the whole Java system is available now on a wide variety of systems has a big impact on any decision to use Java.

6 Analysis: Does Java Keep Its Promises?

In section 2 we discussed the promises of Java that are relevant to teaching Java as a first language. In this section, we review those promises in the light of our discussion.

So, is Java really as simple as it is made out to be? The whitepaper offers the following quote as advice on design:

> *You know you've achieved perfection in design,*
> *Not when you have nothing more to add,*
> *But when you have nothing more to take away.*
>
> *Antoine de Saint Exupery.*

Unfortunately, while the Java designers did indeed take away many things from C++, they have also added many new things. They removed multiple inheritance, but added a different mechanism that appears to do something very similar: interfaces. Explaining the distinction to beginners is non-trivial. The Java encapsulation model is more complicated that C++, and interacts in unintuitive ways with the package construct. (In contrast, the equivalent of packages in C++, namespaces, are completely orthogonal to encapsulation.) Furthermore, as discussed above, the defaults they have chosen make teaching the concept of encapsulation more difficult.

They removed the concept of a pre-processor, but did not provide substitutes for all the mechanisms that `Cpp` fulfills in C++ [Str94]. This has led to there being no clear separation between interface and implementation. Interestingly, Java interfaces could be used in this role, like abstract types of Emerald [RTL$^+$91], although none of the Java literature suggests this.

They removed templates and operator overloading, and so had to provide object type versions of the primitive types, further complicating the primitive/object type distinction. This decision also exposes beginners to the cast construct very early.

Furthermore, it is not clear that it was appropriate to take away all of the things they did. For example, a beginner will never encounter the problems associated with multiple inheritance and operator overloading if they don't know these features exist, and there are arguments that these features are useful for more advanced programming. It is also not clear that completely hiding pointers is useful for teaching beginners. Because they are hidden, the problems associated with them, namely reference semantics, are also hidden, making them a nasty trap for the inexperienced. The reason given for the removal of enumerated types was that in C++ enumerated types inhabited a single namespace, meaning you cannot have the same name in two different enumerated types. The whitepaper favours using constructs like:

```
class Direction extends Object {
  public static final int North = 1;
  public static final int South = 2;
  public static final int East = 3;
  public static final int West = 4;
}
```

Try explaining *that* to beginners! This is an example where, rather than simplifying things, the Java choice makes a relatively straightforward concept much more difficult to describe.

Not having functions leads to the need for `static` (class-wide) methods. While `static` methods can be made to fit in the object-oriented paradigm, it does not fit cleanly, and creates confusion for those not completely comfortable with the general paradigm. For example, it is difficult to explain why the cosine function is a `static` method in the `Math` class, rather than an ordinary method in `Float`. (It has to be this way because the cosine function applies to the primitive `float`, which is not a class and so cannot have methods in the usual sense.)

Unfortunately, in Java there is no way to avoid talking about `static` methods early. For example, many useful methods, such as those for cosine, logarithm, or random number generation, appear as `static` methods in the `Math` class. Even more difficult to avoid is the fact that Java uses a static method to deal with the "startup" problem.

The "startup" problem concerns how an application should start: what happens first? According to the object-oriented model, the first thing that should happen is a message should get sent to an object. So how do you get an object? In Java, there are no global variables, so in order to get an object, it must be created inside of a class somehow, but then a message is needed to be sent to that class to get that object created. The Java solution is to call a `static` method, and in Java, that method must look like this:

```
public static void main(String[] args) {
  // ...
}
```

Thus, short of avoiding applications all together, there is no way to avoid introducing beginners to `static` methods. Note that avoiding applications all together is possible in Java — just use applets. Applets are the kind of application intended to provide a user interface embedded in a web page, and they are supported by structures in the standard library. However, new kinds of applets are created by inheritance, meaning that for the first program we show to beginners, they must see either `static` or inheritance — not a happy choice.

Another problem is that the Java designers have been inconsistent in their design rules. For example, they disallow both overloading of operators and implicit conversion between unrelated types, and yet the following is legal:

```
String s;
int i = 42;
s = "" + i; // Convert integer to string
```

The last statement involves an overloaded `+` operation (for Strings) as well as an implicit conversion of `int` to `String`. Giving the `String` class special status may seem a practical compromise. After all, values of type `int` are automatically converted to `float` and `+` works on both `int` and `float`. However those are primitive types, whereas `String` is an object type. Further complicating things is the fact that `==`, which will only do *reference* equality for object types, in fact does *value* equality for `String` values directly based on literal strings, just as if `String` was a primitive type. But even `String` values that result from concatenation do *not* work this way. The result is difficulty for beginners trying to build a consistent understanding of how the language works.

The attempt to clean up how types work has generally been good. The removal of the `=`/`==` confusion is possibly the most beneficial change that Java makes to C++. However, retaining the baroque "cast" syntax, even if it does something more safer than in C++, is a weird choice. Even C++, with the introduction of run-time type information, introduces better syntax.

In summary, while it is probably the case that Java is simpler than C++, it is not clear that it is that much simpler. What is interesting about many of these problems is that they involve features that also appear in the same or similar form in C++, and yet those features in C++ don't seem to cause the same problems to learners of C++. The reason why they don't seem to cause problems in C++ is that beginners just do not need to be taught those concepts, and they will never know they exist. The reason why the problems exist in Java is due to the *interaction* of different parts of the Java design. For example, the reason why beginners must know about `static`, exceptions, and down-casting is in part because the standard libraries make them very visible. This shows the importance of carefully considering the consequences of all decisions in language design.

7 Conclusions

We have presented a discussion on how good Java is for teaching as a first language to beginning programmers. Java makes a number of promises in this regard:

- object-oriented
- simple
- robust

To a great extent Java resembles C++. However, the principles above have resulted in Java being better than C++ for beginning programmers in several ways, for example:

- early detection of a number of common errors (such as through the use of the `boolean` type and array bounds checking)
- simplification of pointers
- simplification of constructors and destructors

But while Java is a better teaching language than C++ in several ways, there a number of pitfalls that educators must be aware of when teaching Java to beginning programmers:

- poor enforcement of encapsulation
- differences between primitive and object types
- implicit pointers
- poor separation between implementation and interface
- problems associated with genericity
- need to discuss complex topics early, such as `static` and exceptions
- not *all* weird stuff from C++ removed

In this paper we have not addressed whether the object-oriented paradigm is itself reasonable for teaching beginners; however, many educators have made the decision that it is. Having made this decision, the promises made for Java raise hopes that it might be an excellent teaching language: simplicity and robustness seem very attractive. And because Java resembles C++ so much, it seems reasonable to regard C++ as a benchmark for evaluating Java. And so, we expect that Java will be better as a teaching language than C++.

After examining Java, however, and on the basis of early experience, we are disappointed. As we have detailed in this paper, we do believe Java is a superior teaching language to C++ in many ways, but the promises are compromised by pitfalls.

Is Java a sensible language to use for teaching beginning programmers? This is a difficult question to answer, because a sensible answer cannot be based on the language alone. We do believe, on balance, Java to be marginally better for teaching than C++. However, if Java had remained in obscurity, this margin would not be enough for us to choose it to teach with — the popularity of C++ is a practical advantage in many ways. However, Java has itself become very popular, with its own practical advantages such as its widespread availability and the existence of standard libraries. Anyone making a decision must do so taking into account these practical considerations, which are still unfolding and which will differ in individual circumstances.

We believe that for teaching beginning programmers, the Java language itself is reasonable, but not compelling.

References

[AG96] Ken Arnold and James Gosling. *The Java Programming Language*. Addison-Wesley, 1996.

[BMC96] Robert Biddle, Rick Mercer, and Alistair Cockburn. Report on workshop on teaching object design in the first academic year. In *Addendum to Proceedings of OOPSLA 96*. ACM, 1996.

[BT94] Robert Biddle and Ewan Tempero. Teaching C++: Experience at Victoria University of Wellington. In *Proceedings of the Software Education Conference*, Dunedin, New Zealand, November 1994. IEEE Computer Society Press. Also available as Technical Report CS-TR-94/18.

[BT96] Robert Biddle and Ewan Tempero. Explaining inheritance: A code reusability perspective. In *Proceedings of the Twenty-Seventh ACM SIGCSE Technical Symposium*, February 1996. Also available as Technical Report CS-TR-95/18.

[GM96] James Gosling and Henry McGilton. The java language environment: A white paper, 1996. `http://www.javasoft.com/doc/language_environment/`.

[Lev95] Susan Pawlan Levy. Computer language usage in CS1: Survey results. *SIGCSE Bulletin*, 27(3):21–26, September 1995.

[RTL+91] Rajendra K. Raj, Ewan Tempero, Henry M. Levy, Andrew P. Black, Norman C. Hutchinson, and Eric Jul. Emerald: A general-purpose programming language. *Software—Practice & Experience*, 21(1):91–118, January 1991.

[Str94] Bjarne Stroustrup. *The Design and Evolution of C++*. Addison-Wesley, 1994.

[Sun94] Sun Microsystems. The Java Language: An Overview, 1994. `http://www.javasoft.com/doc/Overviews/java/java-overview-1.html`.

[Wal94] Jim Waldo, editor. *The Evolution of C++*. MIT Press, 1994.