Opportunities and Challenges with J2SE 5 for Introductory Programming Teaching

Michael Kölling, Poul Henriksen
Maersk Institute
University of Southern Denmark
{mik, polle}@mip.sdu.dk

Davin MacCall, Bruce Quig, John Rosenberg
Deakin University
Melbourne, Australia
{davmac, bquig, johnr}@deakin.edu.au

ABSTRACT
The recent release of the Java version 5.0 "Tiger" introduces some significant language changes. For educators, some of these changes provide opportunities to improve teaching, while others pose additional problems that require awareness to avoid them. The authors have recently completed the inclusion of support for all new language features into a well-known educational IDE for Java – BlueJ – and in the course of doing so evaluated each of them for usefulness in education, and developed pedagogic strategies to handle the inherent opportunities and challenges. This has formed the basis of the design of the features in BlueJ which support the language changes. In this paper, we describe the results of our evaluation, provide recommendations on treatment of the new features in introductory courses and discuss how BlueJ may be used to illustrate important aspects.

Categories and Subject Descriptors
K.3.2 [**Computers & Education**]: Computer & Information Science Education - *Computer Science Education*

General Terms
Languages

Keywords
CS1, programming, Java, pedagogy, IDE.

1. INTRODUCTION
Recently a new version of Java, J2SE 5.0 "Tiger", has been released. This version includes some significant changes to the language, and thus will affect the way we teach our Java-based introductory courses.

An imposed language change such as this, made without a conscious decision by the teacher, could lead to unfortunate responses in two ways: teachers may resent having to change their courses and continue to use existing material, potentially missing out on valuable teaching opportunities, or they may jump in too deep, trying to discuss all new features, some of which clearly complicate the language and should be avoided in first year teaching.

We attempt to offer concrete information about potential benefits and problems with each feature, helping instructors find a middle path through the new challenges. Our comments are based on extensive functional design and interface design work done for the integration of J2SE 5.0 features in the educational IDE BlueJ [5]. During this design work, we carefully studied characteristics of each new language feature, analysed ways in which it could be included in a modern CS1 curriculum and designed the functionality and user interface in the environment to support the desired activities and illustrate the fundamental concepts.

The most relevant new language features are generic types, enumeration types, auto-boxing, a new for loop and variable argument lists. All of these have been described repeatedly and in detail elsewhere (see, for example, [2] [6]), and we will not repeat that exercise here. We assume that readers are familiar with the syntax and semantics of these constructs.

Instead, we want to discuss the impact of these features on introductory Java teaching. This has been partially covered elsewhere [4], and we will enhance and expand upon the discussion here.

We aim to identify which elements of the new language features represent important concepts that should be explicitly introduced, which parts can safely be ignored, and which parts form pitfalls that introduce new problems.

Furthermore, for those parts that warrant discussion in an introductory course, we give examples of use, guidance for introduction, and discuss how the visualisation elements of the BlueJ IDE help to illustrate the important aspects to students. We will also record several points where our assessment of the usefulness of the new constructs differs from that given in [4].

We start with a discussion of each of the new language features, followed by general discussion and our conclusion.

2. GENERIC TYPES
The inclusion of generic types is, without doubt, the most substantial and most important change to the Java language, both for developers and for education.

When making the switch to "Tiger", we recommend that teachers start using generic types in favour of *Object*-typed parameters and type casts wherever possible.

The most prominent use for generic types is in collection classes: all collections are now available in generic variants, and students are bound to encounter these in their course.

The main reason to embrace generic types is that they are able to completely replace an idiom (*Object*-typed collections) that was hard to explain, unsafe in its use, and seemed to contradict fundamental language principles.

The use of the classic heterogeneous collections is out of line with the spirit of the Java type rules, since it does not support

strong type checking on insertion of its elements. The necessary cast on retrieving elements is hard to explain to students at an early stage in the course, and presents nothing more than (necessary) syntactic noise. Furthermore, students could only fully understand collection semantics if they understood inheritance – a concept that is covered after collections in many courses.

With the availability of generic types, use of old-style collections can (and should) completely be avoided, thus gaining several educational advantages, including the complete decoupling of the topic of inheritance from the first use of collections, and avoiding premature discussion of type casts.

Not everything about genericity, however, is as rosy as this first judgement makes it look. The new language rules include constructs with subtle problems that are not always easy to understand.

The use of generic types in Java can be separated into four groups of tasks:

- using (instantiating and calling) generic types,
- writing simple generic types,
- writing generic types with (bounded or unbounded) wildcards,
- generic methods.

We propose that these four tasks are separated in a course and covered at different times.

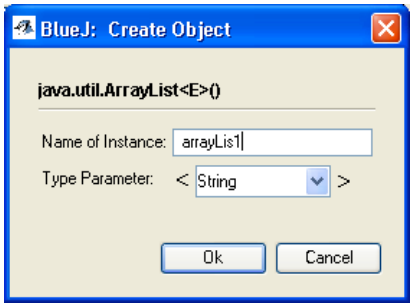

Figure 1: A constructor with a type parameter

2.1 Using generic classes

Using generic types is relatively straightforward, and can be covered quite early in the course. It should be covered when the first collections are needed in student programs.

In BlueJ, generic types are supported explicitly at all stages of interaction. When a generic class is instantiated, the constructor dialogue includes an explicit entry field for each type parameter, similar to the fields for constructor/method parameters. This illustrates the similarity of the concept of parameterisation of classes and methods (Figure 1).

When an object has been instantiated, its type on the object bench is displayed as its actual parameterised type (e.g. *ArrayList<String>* or *ArrayList<Person>* instead of *ArrayList<E>*) (Figure 2). This serves to illustrate the narrowing to an actual type during the instantiation process.

Similarly, calls to methods that were defined with formal generic type parameters show the actual type in the method invocation dialogue. All type checks are appropriately done in interactive invocations, just as they would be in equivalent calls in compiled source code, and type errors are reported when encountered. To ensure consistency, BlueJ uses the same error messages in reporting these errors that the compiler uses when it encounters errors in written source code.

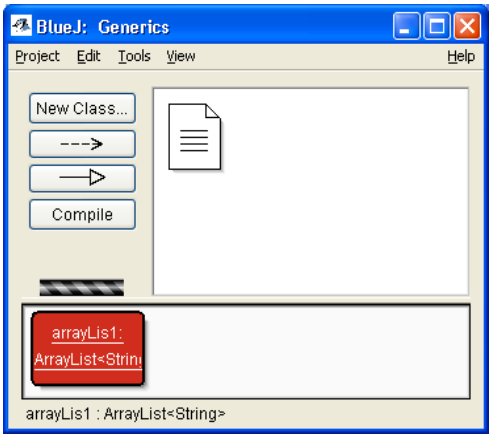

Figure 2: Parameterised objects on the object bench

In introducing generic types, it is helpful to interactively create two or more parameterised objects with different actual type parameters on the object bench. The different actual types can then be observed in their respective methods, and the effect of providing these different type parameters becomes obvious.

Another tool that can be used to examine and understand generic types is BlueJ's object inspector. When inspecting a generic object, fields declared with formal type parameters are displayed with their actual type. Again, contrasting two different instances serves to illustrate the semantics of generic instantiation.

2.2 Writing simple generic classes

Writing simple generic classes is also straightforward, and could be covered any time after use of generic classes has been understood. It is important to realise, however, that these two aspects are not necessarily connected – writing these classes may well be covered some considerable time later than using them. This leaves a great deal of freedom in course design. A similar approach has been used with BlueJ for some time with use of existing classes being introduced well before the design of a new class.

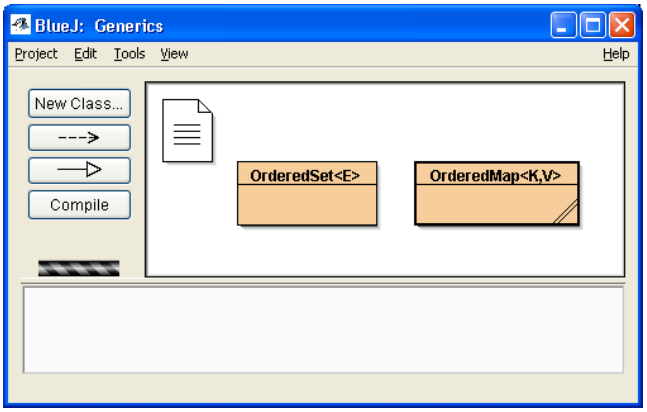

Figure 3: Generic classes in the class diagram

In BlueJ, generic classes in a project are marked with their full generic type name in the class diagram (Figure 3). Thus, they can easily be identified.

2.3 Wildcards and generic methods

The generic type mechanism in Java includes substantially more constructs than those covered above. These include wildcard type parameters (written in Java with a question mark symbol as the formal type parameter), wildcards with upper or lower bounds, and generic methods. In addition, the semantics of several constructs are not as easily understood as one might think at first glance. Among the more subtle issues are subclass relationships of generic collections and arrays of generic objects [3].

While some of these constructs – such as bounded type parameters – represent general principles, and should be discussed at some stage, it is not entirely clear whether this should be in a first year course or later in the curriculum.

If they are discussed in the first year course, the discussion should be separated from the coverage of simple generics. Understanding the need for wildcards and the semantics of bounded generic types requires detailed understanding of inheritance, and thus should be introduced fairly late in an introductory course, while general use of (generic) collections is so fundamental that it should not be delayed this long.

Some other generic type mechanisms have no obvious role in a first year course. One of these is generic methods. A programmer must, for example, decide whether to use bounded type parameters or generic methods in certain situations. I.e., should a method be written as

```
public void m(Collection<? extends E> a);
```

or as

```
public <T extends E> void m(Collection<T> a);
```

These two alternatives are similar in many regards, and the difference is subtle. Examples like this clearly go beyond an average first year course's material.

In other words: attempting to cover all aspects of Java's generic types would be a distraction from the fundamental programming concepts, and should consciously be avoided.

2.4 Generics and legacy code

This leaves, lastly, the question how to deal with old-style collections.

Once J2SE 5.0 is well established, there is no need to write new code using non-generic collections anymore. Students might, however, be confronted with those in reading older Java code from other sources.

We would suggest that old-style collections be avoided initially (while collections are introduced). They can then be discussed later as one example of using inheritance, presenting them as an outdated programming style, thus enabling students to read older code without encouraging them to write it.

3. ENUMERATION TYPES

The addition of type safe enumerations (known as *enum* types) to the Java language is a valuable improvement. The main reason, again, is that it completely replaces a code idiom that was potentially misleading to read and unsafe in operation.

The most common way to implement enumerations before was to use final static integer fields. The most obvious problems are that the declared type (*int*) does not represent the logical type, thus making interpretation harder; that the type does not correctly specify legal values; and that the use of illegal values goes undetected by the language type checker. These and other problems with *int*-values as enumerations have been discussed in more detail in [1].

Solutions to this problem prior to the introduction of explicit enum types include the type-safe enumeration pattern [1]. While this pattern solves many of the main problems, it remains largely unused in introductory courses, partly because teachers do not know about it, and partly because of its syntactic overhead.

These problems have now been completely removed. Using enums makes the code both more readable and type safe.

In introducing enums, teachers should initially concentrate on simple declaration without custom constructors or methods.

BlueJ now includes *enum* as one of the standard templates for creating new classes, and the default template provides an example of such a simple definition:

```
public enum Season
{
    spring, summer, autumn, winter
}
```

Using a simple declaration as above is relatively easy to understand and well worth the additional time needed to introduce an additional language construct. Enumeration types are a general programming language concept in their own right, and thus worth covering, and they support other concepts that we should try to convey, such as the need for type safety, readability and clarity of code.

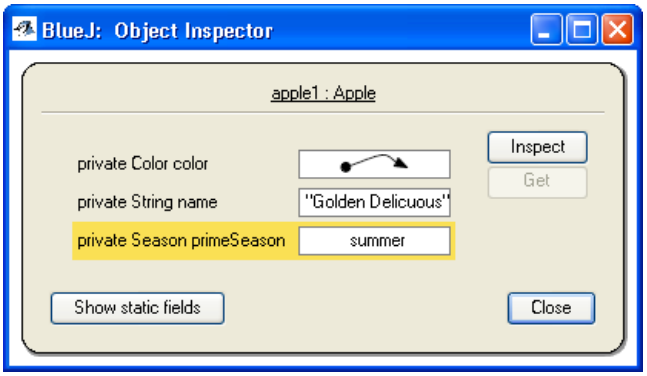

Figure 4: Presentation of enum fields in objects

Enums also allow an extended declaration which includes the definition of custom fields, constructors and methods. Using this extended format is much less important for an introductory course, since it does not touch as much on additional fundamental concepts. (It presents a fascinating example to examine for a programming language course in a later year, since the unification of enums and classes is well worth studying. This, however, would be out of the scope of many introductory programming courses.) We would recommend coverage of this only if a particular given

programming problem clearly benefits from this construct, but not setting aside time for introduction of this construct for its own sake.

One of the benefits of the new Java enums is that they provide readable default string representations.

BlueJ makes use of this when presenting enum values in object inspectors. Instead of presenting enum objects simply as object references, they are shown using their actual value strings (Figure 4). This reflects in the environment the spirit and purpose of enum values, and aids in conceptual understanding and debugging.

In the class diagram, enumeration classes are marked with an «enum» stereotype, so that they can easily be distinguished.

Important characteristics of enum types are that they cannot be explicitly instantiated, and that their implicitly generated instances can be accessed using the static *values()* method.

Both these important aspects are illustrated in the BlueJ interface. An enum class in BlueJ does not have a constructor, but access to the *values()* method is provided through the class popup menu.

From the class inspector, the *Get* button can be used to place the constants onto the object bench for further interactive use. Thus, interactive exploration of enum classes and objects is supported in similar style to interaction with other objects, and important characteristics of enums can be actively explored by students.

4. AUTO-BOXING

Auto-boxing – the automatic wrapping and unwrapping of primitive types into their corresponding object types when needed – has variously been commented on very positively or has been judged mildly useful or neutral in a teaching context [4]. Our view differs fundamentally: auto-boxing is a potentially confusing feature that makes our lives more difficult and will require great care and some extra time in teaching.

The feature is, without a doubt, useful for experienced programmers, since it allows more concise expression of a standard task. However, it is not helpful when trying to learn or teach programming and Java concepts.

The typical situation where beginners are confronted with this construct is the entry of primitive types into collection classes. The traditional way of writing this – the explicit creation of the wrapper object – is tedious and requires more careful thinking than we would like (since we have to write some code purely for technical reasons of the underlying programming language implementation, not for our logical task at hand).

But the new alternative – auto-boxing – would be beneficial only if it would replace the old notion. It does so, however, only syntactically, but not conceptually.

To understand the semantics of a statement such as

```
myList.add(5);
```

a student needs to fully understand the wrapping mechanism, including the creation of the wrapper object and the restriction of collections to object types. The shorter syntax serves to hide, and thus mystify, not clarify, this notion.

For students to acquire the necessary understanding, we have to discuss the wrapping mechanism – probably using and explaining the old-style syntax in the process – *and* the new syntax. We add a mechanism without adding functionality.

If students are left without this full understanding, they are in danger of forming misconceptions about important concepts, such as the distinction between primitive types and objects, and the nature and behaviour of collection elements. Seemingly simple rules, such as "*The type of elements added to a collection must match the declared parameter type in the collection declaration*" are apparently broken by the auto-boxing mechanism. However, understanding this rule is more important than saving the typing effort through auto-boxing – that is why this feature poses more problems than it helps.

In practice, this feature should largely be avoided. Collections of primitive types are usually found either in mathematics-based problems or artificial tasks. Much better examples are available. When programming examples are chosen from practical problem sets, we usually deal with collections of people, or records, or shapes, or diary entries, and many more such things. The emphasis here is on *things* – in other words *objects*. Collections of primitive types should be avoided in the first part of the course, and introduced later when students have the competence to easily understand the need for and notion of wrapper classes. Auto-boxing should then be introduced as the syntactic afterthought that it really is.

Auto-boxing in contexts other than collections – such as assignment – should be completely avoided.

BlueJ supports auto-boxing (it must, since it aims to be a fully compatible Java environment), but does not make a significant effort to illustrate it in sophisticated ways.

To illustrate auto-boxing using BlueJ, a generic collection object of Integers could be placed on the object bench. Primitive `int` types values can then be added using interactive invocation to demonstrate this feature, and the object inspector can be used to inspect the collection object and show the implicit conversion to the Integer class.

5. VARIABLE ARGUMENT LISTS

Variable argument lists (also known as *varargs*) provide a shorthand notation for passing an array of values to a method.

For the learning of programming principles, this construct does not provide substantial new material that would have a recognisable benefit in being covered. On the other hand, the syntax and semantics of this construct are relatively simple, and could be introduced without much time overhead.

We would recommend that the introduction of this construct very early in the course be avoided, since it seems to contradict the language rules about matching of formal and actual parameter lists. In the early weeks of the course, while students still struggle with these fundamental concepts, introducing such exceptional cases should be avoided.

Later in the course, this construct can be covered fairly easily, it does not, however, contribute much to the learning of general programming principles.

Again, we would recommend use of this construct if it happens to fit naturally in a programming example, but not set aside instruction time specifically to cover it for its own benefit.

If it is covered, the BlueJ method call dialogue can be used to illustrate its semantics. When calling a *vararg* method, the call dialogue will initially show one parameter for the variable argument list, together with two buttons labelled + and – that lets users add or remove parameters (Figure 5). This interface provides a strong hint as to the options provided by the *varargs* construct, and lets students actively explore the possibilities and limitations.

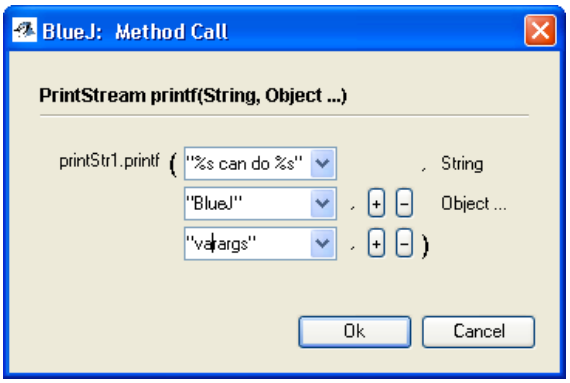

Figure 5: Method call dialogue with a variable argument list

6. NEW FOR LOOP

An additional for loop syntax has also been added to the "Tiger" version of Java. It provides an abbreviated format that embeds the iteration of collections and arrays into the for loop syntax. Its most visible effect is in removing lines of repetitive idiomatic code to access and manipulate an iterator for a data structure. It also complements other new features such as generic collection types.

The new loop makes the code needed to iterate and manipulate collections much shorter and more concise, without greatly compromising readability. We therefore advocate its use when iterating through collections and arrays.

The main benefit of this loop lies in the fact that the use of explicit iterators when introducing collections can now be completely avoided. This makes it significantly easier to introduce collections quite early in the course. The necessary iterator construct traditionally was the largest stumbling block in introducing use of collection classes early, and this has now been removed.

It should be noted that this does not altogether replace traditional for loops (not even for collection iteration) as it does not provide any capacity to use the iteration as a counting function, or to use any custom form of traversal.

BlueJ provides no special support for this feature; it is supported through the environment's support for parsing, compilation and the execution of JDK 5.0 compatible classes.

7. OTHER FEATURES

There are a number of other changes introduced in JDK 5.0, which are briefly discussed below.

Java now allows static imports. With this feature the use of the static keyword in an import statement allows static members and methods to be referenced without having to reference the class name. This would seem to add little overall benefit and possibly cause some confusion over where these methods and members are defined.

The meta-data facility is another new feature that aims to minimise the writing of boilerplate code. This feature would typically be beyond the scope of CS1.

Apart from language changes, there are also significant changes in the class libraries. One that may be of interest from an education standpoint is the new scanning and formatting classes. These simplify the code used to provide console input and output. This is of less interest to us since the interaction and inspection facilities of BlueJ eliminate the need for prematurely exposing students to I/O complexities.

8. CONCLUSION

The new language features of Java's "Tiger" release add a number of opportunities and potential problems in relation to the teaching of programming principles.

Teachers who choose to ignore those changes miss out on opportunities to teach some important concepts, and may encounter discrepancies as students discover some of those features on their own.

On the other hand, each of the new features adds more syntax to the language, thus reducing the simplicity of Java – a characteristic that is important for use of the language in teaching.

It is important to identify which of the new constructs represent a more general programming principle, and thus are worth the effort of teaching the added syntax, and which are changes merely at the syntactic level that add little to the understanding of general programming principles, and can even cause confusion amongst students. If selected carefully – including ignoring some parts – the new language constructs can contribute to raising the quality of our first year courses.

BlueJ, in its version 2.0, has been enhanced to illustrate those principles we have identified as relevant for beginners, and using it as a visualisation aid during program development can support the introduction of these new constructs.

9. REFERENCES

[1] Bloch, J., Effective Java: Programming Language Guide, Addison-Wesley, 2001.

[2] Bloch, J., New Language Features for Ease of Development in the Java 2 Platform, web document at http://java.sun.com/features/2003/05/bloch_qa.html, accessed August 2004.

[3] Bracha, G., Generics in the Java Programming Language, http://java.sun.com/j2se/1.5/pdf/generics-tutorial.pdf, accessed August 2004.

[4] Frens, J.D., Taming the Tiger: Teaching the Next Version of Java™. In Proceedings of the 35th SIGCSE symposium, Norfolk, Virginia, March 2004.

[5] Kölling, M., Quig, B., Patterson, A. and Rosenberg, J., The BlueJ system and its pedagogy, Journal of Computer Science Education, Special issue on Learning and Teaching Object Technology, Vol 13, No 4, 249-268, Dec 2003.

[6] Sun Microsystems, J2SE™ 5.0 "Tiger" Feature List. Available at http://jcp.org/aboutJava/communityprocess/pfd/jsr176/, accessed August 2004.