Prepare Yourself... The Tiger Will Be Unleashed Soon

María Lucía Barrón Estrada[1], Ryan Stansifer[2], and Ramón Zatarain Cabada[1]

[1]Instituto Tecnológico de Culiacán, Av. Juan de Dios Batíz s/n Col. Guadalupe, Culiacán, Sin. 80220 México Tel. 667-713 3804
mbarron@fit.edu, rzc777@hotmail.com
[2]Florida Institute of Technology, 150 W. University Blvd. Melbourne, FL, 32901 USA
ryan@cs.fit.edu

Abstract. The Java programming language has been, for the last several years, in the process to adding many features. The newest version of Java is called Tiger and will include new features to simplify software development. The watchwords of the Java 2 Standard Edition (J2SE) version 5 are *"easy of development"* and *"easy to use"*. The most significant addition to the language is the ability to create and manipulate generic types. Despite major changes the language syntax had to be changed only slightly and the Java virtual machine remains the same, thus insuring compatibility with existing code. Others features in Tiger allow developers to annotate programs and reduce the amount of code they need to write. The new features of Java will have a major impact in the use of Java and in the curricula of Computer Sciences and Informatics. In particular, Java's generics will put parametric polymorphism back into the toolkit of students, teachers, and practicioners. In this paper, we describe several of the changes in Java and provide examples illustrating their use.

Keywords: Java, generic types, typesafe enums ,foreach, programming, data structures.

1 Introduction

Java [AG 98] is one of the most popular programming languages. The balance of simplicity and power that Java offers allows developers to simplify the construction of robust and maintainable programs. However there is room for improvement. Java has been constantly evolving since it catapulted in the programming arena in 1995. After its first release, nested classes were added, event handling in GUIs was revised, numerous speciality APIs were added, and the collection class hierarchy was designed. Many changes to Java are requested by users through the Java Community Process. One of the most requested features is some sort of parametric polymorphism (generics). A formal document [JSR014] outlines the proposal. With the next release of Java, Tiger, the proposal becomes reality.

Adding parametric polymorphism to Java has been the focus of many researchers as well [BCK+ 03, BOSW 98a, BOSW 98b, CS 98, MBL 97, and OW 97]. After many proposals

were analyzed, a new version of the language including this and other features is now under review process. Java 1.5 is named Tiger, several beta versions have been released since last year; the latest version is beta 3, which is available for developers since July 19, 2004. Sun plans to release the final version later this year.

Many universities all over the world have adopted Java as the programming language to be taught in several courses of different curricula. The new features included in Java 1.5 will impact this process in several ways. Some of these additions will release the programmer of writing repetitive code that could be automatically generated. Other features will support the *"easy of development"* theme providing enumerations, autoboxing, enhanced *for* loops, and static import. All these changes are described in [JSR 201].

In this paper we present a list of features that are part of the new version of Java. We show some examples of code in the current version y contrast them with code in Tiger. The paper is organized as follows: section 2 explains the benefits of static import, section 3 describes the automatic conversion of primitive types to they corresponding reference types (wrapper classes), this process is known as autoboxing. Section 4 describes the new form of *for* loop designed to iterate over collections. Section 5 explains the enumeration facility provided in the language and its differences with respect to other languages. The most important feature added in this version is presented in section 6, it contains three subsections describing how to use generics, how to implement generics and the problems associated to them. Metadata, a declarative way of programming using annotations is shown in section 7. Section 8 shows an example of a procedure with a variable number of arguments. API's to get formatted input/output are presented in section 9. Finally conclusions are offered in section 10.

2 Static Import

In the current version of Java, class members defined as static can be imported by other classes that want to use them but it is necessary to refer to them using fully qualified names (*ClassName.member* or *ClassName.member()*). In other languages like Pascal or C, imported members can be directly referred to without using qualifications.
Java 1.5 includes variants of the import statements to allow importation of static members (fields and methods) in the same way classes and interfaces can be imported.

Example of the two new import static declarations

```
import static TypeName.Identifier;
import static TypeName.*;
```

In the example shown above the first declaration imports into the current unit the specific *Identifier*, which must be a static member of the class or interface named *TypeName*. The second declaration should import into the current unit all the static members (fields and methods) of the class or interface named *TypeName*.

Next we show an example of the use of static import declarations in Java 1.5 and contrast it with the current version of Java.

Current version

```
import java.lang.Math;
y = Math.sqr(r) * Math.PI;
```

Tiger Java 1.5

```
import static java.lang.Math.*;
y = sqr(r) * PI;
```

3 Automatic Boxing and Unboxing

Frequently programmers need to wrap primitive types into objects. Java provides wrapper classes that can be used by the programmer to convert primitive types into reference types. However this is a process that could be automatically done by the compiler and it is commonly known as *boxing*. Java 1.5 offers this feature facilitating the integration of generic types into the language.
The rules for boxing are straightforward. A value of a primitive type is converted into its corresponding reference type. Corresponding types are shown in table 1. An opposite process, known as unboxing, converts a reference type into a value of its corresponding primitive type.

Table 1. Correspondence between reference types and primitive

Primitive type	boolean	byte	double	short	int	long	float
Reference type	Boolean	Byte	Double	Short	Integer	Long	Float

An example of the use of automatic boxing and unboxing is presented in the code fragment below. A code fragment with the same functionality using the actual version of Java is presented before.

Current version. Wrapping primitive type into reference types

```
public static void main (String [] args) {
      Stack x = new Stack();
        x.push(new Integer(17));.       // wrap 17
        Integer y = (Integer) x.pop();
        int num = y.value();  // get value
    }
```

Tiger Java 1.5. Using automatic boxing/unboxing with parameterized classes

```
public static void main(String args[ ] ) {
     Stack<Integer> x = new Stack<Integer>();
      x.push(17);                    // boxing
      Integer y =  x.pop();
      int num = y;                   // unboxing
   }
```

4 Foreach

An iterator provides a mechanism to navigate sequentially through a collection of elements. Using the current version of Java, boilerplate code is needed to iterate through a collection defining explicitly an iterator. This is shown in the first code fragment below. Tiger provides a new form of *for* loop commonly known as *foreach* loop. The *foreach* loop reduces the need of boilerplate iteration code, and simplifies the code reducing the chances of errors. The code presented in the right part of figure 4 is using a generic type to define a type argument of a Collection received as argument. This code is based on a similar example presented in [BG 03]. As we can appreciate iterating over the elements of the collection is simpler shorter and safer in the code using Tiger than the code using the current version. Both code fragments are shown below. The enhanced for loop can be used to iterate through array elements also.

Current version. An example of iterating over a collection.

```
void cancelAllElements(Collection c) {
      for( Iterator Ii = c.iterator(); i.hasNext() ; ) {
           TimerTask timer = (TimerTask) i.next();
           timer.cancel();
      }
}
```

Tiger Java 1.5. An example of foreach using generics and collections

```
void cancelAllElements(Collection<TimerTask> c) {
      for( TimerTask  timer : c)
            timer.cancel();
}
```

5 Typesafe Enumerated Types

Enumerations were widely known in Pascal and C/C++. This construct allows the definition of names to help document and clarify code. Java didn't have an enumeration

construct and programmers were forced to workaround to create this pattern. This is not an easy task if type safety is required and many programmers fail to implement the correct pattern. In his book [B 01] Bloch discuss the importance of implement a correct enumeration but the amount of code needed increases the chances of errors. Tiger offers a typesafe enum facility that combines power, performance and it is easy to use. Enums have all the advantages of the typesafe enum patter described in [B 01]. Enums are a special kind of classes. They introduce a new keyword into the language. Its simplest definition looks like a C/C++ enum declaration, but it doesn't have its disadvantages. Figure 5 shows two examples of enum declarations.

Example of the typesafe enumeration pattern.

```
public class City {
   private final String name;
   private Suit(String name) { this.name = name; }
   public String toString() { return name; }
   public static final City MIAMI = new City("Miami");
   public static final City ORLANDO = new City("Orlando");
   public static final City MELBOURNE = new
City("Melbourne");
}
```

Two examples of the type safe enumeration facility provided in Tiger.

```
public enum City { Miami, Orlando, Melbourne; }
public enum Ticket {
    Plateau(1), General(5), Number(10), VIP(100);
    Ticket(int value) { this.value = value; }
    private final int value;
    public int value() { return value; }
}
```

Class modifiers in enum declarations contain restrictions. Some of them are: all enum declarations are implicitly **final** and can not be **abstract** unless they contain constant-specific class bodies for every constant, and members of enum classes are implicitly static.

6 Generic Types

The absence of generics in Java forces the programmer to use subtype polymorphism to workaround writing code with cast operations that could fail at runtime. Tiger provides generics enhancing the expressiveness of the language and improving safety because more errors can be detected at compile time. A detailed description of generics can be found in

[B 04]. Generic types in Java are not like templates in C++. They are compiled once and for all, they are type checked at compile time and they are translated into a homogenous piece of code.

6.1 Generic Libraries

Generic types are widely used in the Collections API. In Tiger, collection types like List or ArrayList, are part of the Collection API and can be parameterized by a type to specify the type of elements the collection will contain. An example of the List class is presented next in both version.

Current version

```
List xs = new LinkedList();
xs.add(new Integer(0));                    // wrapping needed
Integer x = (Integer) xs.iterator().next(); // cast needed
```

Example of an instantiation of a generic class in Tiger Java 1.5

```
List<Integer> xs = new LinkedList<Integer>();
xs.add(0);              // automatic boxing
Integer x = xs.iterator().next();      // no cast
```

6.2 Implementing Generic Types

It is easy to define generic classes and interfaces. A generic class definition contains a list of type parameters after the class identifier. The code fragment below shows an example of a generic class with two type parameters. The main function of this code fragment shows an example of how instantiate and use generic classes.

A generic class declaration and instantiation.

```
public class Pair<T,U> {
   private T first;
   private U second;

   public  Pair(T f, U s) {
      first = f;
      second = s;
   }

   T getfirst () { return first; }
   U getsecond() { return second; }
   String toString() {...}
   ...
```

```
    public static void main(String args[ ] ) {
        Pair<Integer, String>  x = new Pair<Integer, String>
(1,"Aaron");

        List<String> l = new List<String>();

        int tot = x.getfirst() + 1;
        l.add(x.getsecond());
    }

}
```

6.3 Problems with generics

The translation approach used for generic classes requires that the type parameters must be reference types. Primitive types cannot be used to instantiate generic classes or interfaces. Tiger provides a feature to ameliorate this problem, automatic boxing and unboxing of primitive types. F-bounded polymorphism is used to define constrained type parameters but it cannot be combined smoothly with inheritance in the presence of binary methods as noted in [BS 03].

7 Metadata

This feature allows annotating classes, interfaces, fields, and methods as having particular attributes. The current version of Java has a limited implementation of metadata using tags. The new version of Java contains six built-in annotations and users can define custom made annotations to decorate their types controlling their availability. With this new feature developers are going to avoid writing boilerplate code that may be automatically generated and updated by tools maintaining all the information is the source file.

7.1 Built-in annotations

The built-in annotations available in the new version of Java are:

a) java.lang.Overrides - indicates that a method declaration in a class intent to override a method form its superclas.
b) java.lang.annotation.Documented - indicates that javadoc documents the annotated element. It can be ignored by the tool.
c) java.lang.annotation.Deprecated - The java compiler can warns the user if he/she uses the annotated element.
d) java.lang.annotation.Inherited - A class decorated with an annotation that contains the annotation Inherit, will inherit the annotation to all derived classes.

e) java.lang.annotation.Retention - used to determine the annotation availability.
f) java.lang.annotation.Target -indicates to which kind of element (class, method, or field) the annotation is applicable.

An example of an annotation is shown in the code fragment below.

Current version

```
public interface PingIF extends Remote {
   public void ping() throws RemoteException;
}

public class Ping implements PingIF{
   public void ping() { … }
}
```

An example of metadata annotations using Tiger Java 1.5

```
public class Ping {
   public @Remote void ping() { … }
}
```

8 Variable Arguments

In the actual version of Java, the number of parameters in a method is fixed. Generally when a method needs an arbitrary number of parameters, it uses an array to store all the arguments. The new version of Java allows defining methods with a variable number of arguments without using arrays to store them. The code fragments below show an example of this in both the current version and Tiger.

Current version

```
Object [ ] arguments = {
   new Integer(7),
   new Date(),
   " a disturbance in the force"
}

String result = MessageFormat.format (
     "At {1,time} on {1,date}, there was {2} on planet" +
"{0,number,integer}.",    arguments);
```

reused without recompilation. In this regard the big problem was with the collection classes. The new generic collection classes are translated in a way that produces the same Java bytecode as the old (non-generic) collection classes. This is not without its own drawbacks, however.

The collections classes: lists, sets, dictionaries, and so on, are by their nature polymorphic. With generics in Java it will be possible to program these data structures naturally without attention to the type of the elements. This should be a great help in the instruction of data structures using the Java programming language.

Automatic boxing and unboxing also contributes to a more natural view of data structures by erasing the difference between, say, stacks of integers and stacks of strings.

Input and ouput has been greatly improved. Although quite logically designed, I/O has been cumbersome in Java. It as been the cause of much frustration to developers, students, and instructors alike. The new I/O API with it C style formatted output will make common tasks like printing numbers in columns simple again.

References

1. [AG 98] Ken Arnold and James Gosling. *The Java™ Programming Language*. Addison Wesley. 1998
2. [B 01] Joshua Bloch. Effective Java™ Addison Wesley. 2001.
3. [B 04] Gilad Bracha. Generics in the Java Programming Language. March 9, 2004. Available online at http://java.sun.com/j2se/1.5.0/lang.html
4. [BCK+ 03] Gilad Bracha, Norman Cohen, Christian Kemper, Martin Odersky, David Stoutamire, Kresten Thorup, and Philip Wadler. Adding generics to the Java™ Programming Language: Public Draft Specification, Version 2.0. June 23, 2003. Available online at http://java.sun.com
5. [BG 03] Joshua Bloch and Neal Gafter. Forthcoming Java™ Programming Language Features. Presentation in JavaOne Conference. June 2003.
6. [BOSW 98a] Gilad Bracha, Martin Odersky, David Stoutamire, and Philip Wadler. GJ: Extending the Java Programming Language with type parameters. Manuscript, March 1998, revised August 1998.
7. [BOSW 98b] Gilad Bracha, Martin Odersky, David Stoutamire and Philip Wadler. Making the future safe for the past: Adding Genericity to the Java Programming Language (GJ) In *Proceedings of OOPSLA '98, Conference on Object-Oriented Programming, Systems, Languages and Applications*, Vancouver, British Columbia, Canada. October 1998.
8. [BS 03] Maria Lucia Barron Estrada and Ryan Stansifer. Inheritance, Genericity and Binary Methods in Java. In *Computacion y Sistemas*, Volumen VII, numero 2. Mexico, DF. December 2003.
9. [CS 98] Robert Cartwright and Guy L. Steele. Compatible Genericity with Run-Time Types for the Java™ Programming Language. (NextGen) In *Proceedings of OOPSLA '98, Conference on Object-Oriented Programming, Systems, Languages and Applications*, Vancouver, British Columbia, Canada. October 1998.

reused without recompilation. In this regard the big problem was with the collection classes. The new generic collection classes are translated in a way that produces the same Java bytecode as the old (non-generic) collection classes. This is not without its own drawbacks, however.

The collections classes: lists, sets, dictionaries, and so on, are by their nature polymorphic. With generics in Java it will be possible to program these data structures naturally without attention to the type of the elements. This should be a great help in the instruction of data structures using the Java programming language.

Automatic boxing and unboxing also contributes to a more natural view of data structures by erasing the difference between, say, stacks of integers and stacks of strings.

Input and ouput has been greatly improved. Although quite logically designed, I/O has been cumbersome in Java. It as been the cause of much frustration to developers, students, and instructors alike. The new I/O API with it C style formatted output will make common tasks like printing numbers in columns simple again.

References

1. [AG 98] Ken Arnold and James Gosling. *The Java™ Programming Language*. Addison Wesley. 1998
2. [B 01] Joshua Bloch. Effective Java™ Addison Wesley. 2001.
3. [B 04] Gilad Bracha. Generics in the Java Programming Language. March 9, 2004. Available online at http://java.sun.com/j2se/1.5.0/lang.html
4. [BCK+ 03] Gilad Bracha, Norman Cohen, Christian Kemper, Martin Odersky, David Stoutamire, Kresten Thorup, and Philip Wadler. Adding generics to the Java™ Programming Language: Public Draft Specification, Version 2.0. June 23, 2003. Available online at http://java.sun.com
5. [BG 03] Joshua Bloch and Neal Gafter. Forthcoming Java™ Programming Language Features. Presentation in JavaOne Conference. June 2003.
6. [BOSW 98a] Gilad Bracha, Martin Odersky, David Stoutamire, and Philip Wadler. GJ: Extending the Java Programming Language with type parameters. Manuscript, March 1998, revised August 1998.
7. [BOSW 98b] Gilad Bracha, Martin Odersky, David Stoutamire and Philip Wadler. Making the future safe for the past: Adding Genericity to the Java Programming Language (GJ) In *Proceedings of OOPSLA '98, Conference on Object-Oriented Programming, Systems, Languages and Applications*, Vancouver, British Columbia, Canada. October 1998.
8. [BS 03] Maria Lucia Barron Estrada and Ryan Stansifer. Inheritance, Genericity and Binary Methods in Java. In *Computacion y Sistemas*, Volumen VII, numero 2. Mexico, DF. December 2003.
9. [CS 98] Robert Cartwright and Guy L. Steele. Compatible Genericity with Run-Time Types for the Java™ Programming Language. (NextGen) In *Proceedings of OOPSLA '98, Conference on Object-Oriented Programming, Systems, Languages and Applications*, Vancouver, British Columbia, Canada. October 1998.

10. [JSR 014] Sun Microsystems. Adding Generic Types to the Java™ Programming Language. Java Specification Request JSR-000014, 1998. [Online] URL http://www.jcp.org/en/jsr/detail?id=14 Approved in 1999.
11. [JSR 176] J2SE™ 5.0 (Tiger) Release Contents http://www.jcp.org/en/jsr/detail?id=176
12. [JSR 201] Sun Microsystems. Extending the Java Programming Language with Enumerations, Autoboxing, Enhanced for loops and Static Import. Java Specification Request JSR 201. Available online at http://jcp.org/en/jsr/detail?id=201
13. [MBL 97] Andrew C. Myers, Joseph A. Bank, and Barbara Liskov Parameterized Types for Java (PolyJ). In *Proceedings 24th ACM Symposium on Principles of Programming Languages*, pages 132-145, Paris, France, January 1997.
14. [OW 97] Martin Odersky, Philip Wadler. Pizza into Java: Translating theory into practice. In *Proceedings 24th ACM Symposium on Principles of Programming Languages*, pages 146–159, Paris, France, January 1997.