Growing DrJava to cope with language extensions carried out in Java 5.0

by

Michael Dyrby Jensen

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE

Master of Science

APPROVED, THESIS COMMITTEE:

Robert Cartwright
Professor
Computer Science, Rice University

Walid Taha
Assistant Professor
Computer Science, Rice University

Dan Wallach
Associate Professor
Computer Science, Rice University

Houston, Texas
February, 2006
INFORMATION TO USERS

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleed-through, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.
Growing DrJava to cope with language extensions carried out in Java 5.0

Michael Dyrby Jensen
Advisor: Professor Robert Cartwright

Abstract

While Java continues to gain popularity in both industry and academia, few Java programming environments enable the user to directly interact with the code. Professional environments require developers to use a command line interface to execute Java code. In contrast, our DrJava programming environment supports a read-eval-print-loop (REPL) incorporating a Java interpreter called DynamicJava – enabling developers to execute arbitrary fragments of Java code on-the-fly.

DynamicJava was developed at the University of Nice, but development ceased in 2002 with DynamicJava supporting Java 1.3. In 2004, the Java language was extended in the Java 1.5 (renamed 5.0) release. To support Java 5.0 in DrJava, we were confronted with the task of extending DynamicJava to support the new language features.

This dissertation describes the new features in Java 5.0 – including generic types, auto-boxing, variable arguments, static import, foreach, enumeration types and metadata – and explains how we extended the existing DynamicJava code base to support them.
Acknowledgments

My greatest thanks and appreciation to my supervisor, Professor Robert Cartwright, for giving me the opportunity to work with him and his JavaPLT research group at Rice University, and for being an inspiration while working on this project.

Thank you, Professor Walid Taha and Professor Dan Wallach for inspiration throughout coursework, and for being on the committee for this project.

Thanks to Moez Abdel-Gawad for his patience working with me, without his diligence and passion for discussions, this project would not have come this far.

James Sasitorn, Adam Wulf, Johnathan Lugo, and Eric Cheng, thank you for our many discussions and conversations.

The students and teaching assistants of Comp312, for all their support, help and friendship which made this project possible.

Last, but certainly not least, thanks to my wife, Anne Elise Kennedy, for your love, and constant faith in me.
Contents

Abstract ii
Acknowledgments iii
List of Illustrations vii

1 Introduction 1
  1.1 Problem Definition 2

2 DrJava 5
  2.1 The file navigator 7
  2.2 The definitions pane 7
  2.3 The interactions pane 9
  2.4 Other recent extensions to DrJava 10

3 DynamicJava 12
  3.1 Differences between DynamicJava and Java 12
  3.2 Technical Description of DynamicJava 17
  3.3 DynamicJava’s link to DrJava 20

4 Generics 23
  4.1 Implementing generics in Java compilers 27
  4.2 Implementation of generics in DynamicJava 38

5 Static import 50
  5.1 Implementation of static import 51
6 Variable Arguments 56
   6.1 Implementation of Variable Arguments ......................... 57

7 Foreach 59
   7.1 Implementation of Foreach .................................... 60

8 Autoboxing 64
   8.1 Implementation of Autoboxing ................................ 67

9 Enum 71
   9.1 Implementation of Enum ..................................... 75

10 Metadata 80
   10.1 Implementing metadata in DynamicJava ...................... 81

11 Tools 82
   11.1 DrJava ......................................................... 82
   11.2 JUnit ......................................................... 82
   11.3 Ant ........................................................... 83
   11.4 JavaCC ......................................................... 83
   11.5 Clover ......................................................... 84
   11.6 CVS .......................................................... 85

12 Methodology 86
   12.1 On-site customer ............................................... 87
   12.2 User stories .................................................. 87
   12.3 Iterations ...................................................... 88
   12.4 Design sessions ................................................ 88
   12.5 Acceptance tests .............................................. 89
13 Related Work

13.1 Early Interactive Language Implementations ................................. 94
13.2 Java Interpreters ...................................................................... 95
13.3 BeanShell .............................................................................. 95
13.4 Groovy .................................................................................. 97
13.5 Why BeanShell and Groovy were rejected in favor of DynamicJava ... 98

14 Conclusion and future work

14.1 Enhancing DynamicJava was a Collaborative Effort ...................... 100
14.2 Remaining Work ..................................................................... 100
14.3 Disallow extension of the Enum Class ....................................... 101
14.4 Add generic type checker .......................................................... 102
14.5 Generate bridge methods .......................................................... 102
14.6 Solve the forward reference problem ......................................... 103
14.7 Reflections on Java, Extreme Programming, and Generics .......... 104

Bibliography .................................................................................. 107
Illustrations

2.1 Screenshot of DrJava .................................................. 6

3.1 A picture of the inner structure of DynamicJava .................. 18

4.1 Depicting LinkedList<? as super type of all LinkedList. ........ 33
4.2 Depicting an example hierarchy using upper bounds ............. 34
4.3 Depicting LinkedList<?> as super type of all LinkedList. ....... 35
4.4 Illustration of the extra overhead due to the preprocessor. ....... 39
4.5 Illustration without the extra overhead of the preprocessor. ....... 41
Chapter 1

Introduction

As the power of personal computers has grown, professional programming environments have become progressively more sophisticated. For the Java programming language, a variety of complex environments including Eclipse, NetBeans, IDEA, IntelliJ, and JBuilder have become well-established in the marketplace. Unfortunately, none of these environments is well-suited for students learning to program in Java or for developers who want a lightweight environment with a transparent, intuitive interface. The mainstream environments have a bewildering array of buttons, tool bars, and options – forcing users to learn many complex conventions before they can write and execute a trivial program.

For students and developers new to the Java language, a programming environment should support a simple obvious interface that requires little explanation – so that they can concentrate on learning the language and how to design Java programs. The DrJava programming environment, developed by the JavaPLT group at Rice University, is designed to fill this need. Besides providing students and beginning developers with a programming environment to meet their needs, DrJava offers the unique ability to interactively evaluate arbitrary fragments of Java code using a “read-eval-print-loop” (REPL), which is very useful in incrementally developing programs. Although DrJava is targeted primarily at students learning Java, it is capable of supporting large production projects including DrJava itself. Some professional developers use DrJava in preference to more elaborate environments because its REPL is so helpful in incrementally developing programs.

DrJava focuses on supporting the interactive evaluation of Java code – effectively transforming Java from a batch-oriented language to an interactive one. To achieve this functionality, DrJava supports a “read-eval-print-loop” incorporating a Java interpreter called
DynamicJava. DynamicJava was developed by Stephane Hillion from the university of Nice [11] in 1999 and actively supported until 2002. DynamicJava is described in detail in chapter 3. It supports the same syntax as regular Java and can execute Java class files (including the class libraries) using reflection as well as Java source code. In contrast to a Java compiler, DynamicJava can evaluate arbitrary code fragments that are Java expressions or sequences of Java statements.

Java 1.3 was the last version of the Java platform supported by DynamicJava. Since the cessation of support for DynamicJava, the Java platform – including the Java programming language – has continued to evolve. Java 1.4, released in 2002, only contained a few very small changes in the language, which DrJava easily accommodated by sub classing a few classes in DynamicJava. In contrast, Java 1.5, released in 2004, included major extensions to the Java programming language – including generic types,autoboxing,variable arguments,static import,foreach,enumeration types and metadata. The magnitude and significance of these extensions is evidenced by the fact that Java 1.5 was renamed as Java 5.0. Since the DrJava REPL critically depends on the DynamicJava interpreter, the Java Programming Languages Team decided to take responsibility for extending DynamicJava to support Java 5.0. This dissertation describes how the team addressed the problem of extending a large “foreign” code base to support the complex new behavior dictated by Java 5.0.

1.1 Problem Definition

This thesis focuses on the technical problems encountered in the extension of the DrJava REPL – the “interactions pane” in the DrJava interface – to support Java 5.0. Since the DrJava REPL simply delegates the task of interpreting Java code fragments to DynamicJava, essentially all of the technical issues involved in extending DrJava REPL to support Java 5.0 are encapsulated within extending the DynamicJava interpreter to support Java 5.0.
The new features of Java 5.0 consist of the following:

- **Generic types**, an enhancement to Java's type system by adding compile-time type safety and reduce the amount of castings significantly.

- **Foreach**, a construct used for looping over collections and arrays that subsumes common, error-prone usages of iterators.

- **Static import**, a facility which allows access to static members without having to qualify the members with class names.

- **Variable arguments**, a mechanism that allows methods and constructors to accept a variable number of arguments without having to wrap them into an array first.

- **Autoboxing**, a facility that automates wrapping and unwrapping primitives and wrapper classes, removing the tedious manual conversion.

- **Enumeration types**, an extension of the Java type system that supports the definition of enumerated types with arbitrary methods and fields in a type safe fashion.

- **Metadata**, a facility that allows the user or annotations in the code, allowing tools, such as JUnit, to know what members to use.

To support Java 5.0 in DrJava, the JavaPLT team had to extend DynamicJava to support for these new features. In the effort, the team treated DynamicJava as a separate, stand alone application, enabling the Java 5.0 version of DynamicJava to be run as a separate application and isolating the new functionality of Java 5.0 within a single component of DrJava.

The remainder of this thesis is organized as follows. Chapter 2 gives a brief overview of the DrJava programming environment. Chapter 3 describes the internals of DynamicJava. Chapters 4 through 10 describe the language extensions in Java 5.0 and how DynamicJava was extended to support them. The design of each of these features was based on it description in the Java Language Specification [9]. Chapter 11 and Chapter 12, describe the tools
and methodology that was used in developing the new version of DynamicJava. Chapter 13, discusses related work on interpreters that accommodate compiled code including the two major competing interpreters for Java. Finally, Chapter 14 summarizes what we did, what we learned, and what remains to be done with regard to extending DynamicJava.
Chapter 2

DrJava

DrJava is a lightweight programming environment created by the JavaPLT research group at Rice University*. The environment was developed to provide students with a friendly, easy-to-use programming environment that encourages “hands-on” experimentation with the Java language. DrJava supports a very simple interface, making the task of writing and running Java programs almost self-explanatory. The ability to execute arbitrary code fragments including library calls enables students to explore behavior of Java language constructs and library code.

Since its inception in the fall of 2001, DrJava has been extended to support the essential features required to support production programming—particularly the “test-driven” approach to software engineering favored by our research group. These features were added primarily because our research group believed it was important to develop DrJava in DrJava—ensuring that we were intimately familiar with all the “nooks and crannies” of the DrJava interface. During the evolution of DrJava, the development team has strived to preserve the transparent, intuitive character of the DrJava interface. The added functionality is generally invisible to users who choose not to use it.

The developers have also strived to preserve the lightweight character of DrJava. Although the size of the DrJava executable has grown from just over a megabyte to nearly six megabytes (primarily through the addition of metadata to class files in Java 5.0, the incorporation of comprehensive documentation in the executable, and the addition of some third-party open source libraries such as JUnit and BCEL), it has retained (and even improved) its lightweight responsiveness as the power of typical personal computers and Java

*Available for download at: http://www.drjava.org

5
optimization technology has improved.

DrJava has been adopted by many major colleges and universities and some high schools as the primary vehicle for teaching Java programming. Programming Languages Team (JavaPLT) at Rice University. In addition, two major textbooks [10][15] – with no connection to Rice JavaPLT – have adopted DrJava as the environment used in the text.

DrJava provides a graphical user interface consisting of a window subdivided into three panes. At the top of the window, there is a tool bar containing a button for each of the common commands executed by student programmers. Above the tool bar, there is a menu bar with menus containing configuration options and commands for more advanced users. See figure 2.1.

Figure 2.1 : Screenshot of DrJava
2.1 The file navigator

The narrow panel on the left side of the screen is called the (file) navigation pane, it contains the names of all open files in alphabetic order. Within this pane, the name of the current file is highlighted in light blue and its contents are displayed in the large definitions pane immediately to the right (See Section 2.2). When DrJava opens, there is an empty file shown in the navigator pane called Untitled and the corresponding blank document is shown in the definitions pane. This empty file is removed if another file is loaded. If the untitled document is edited, it is retained and renamed when saved. Changing the contents of a document forces an asterisk to appear next to the filename, indicating that the document has been altered since it was last saved.

The default mode for displaying file in the navigation pane is called “flat file mode”. This was the only mode for displaying collections of open files in early versions of DrJava. More recently, DrJava has been extended to support designated collections of files organized as “projects”. Using the Project menu, it is possible to create a new project or open an existing one. A new project can be populated with files simply by opening files in the tree rooted at the project root. By using the Open Folder command, the user can open all of the files in a folder (directory) and its sub folders. When a project is open, the navigation pane displays a tree structure showing all of the files in the project organized as a Java source tree. To ensure the lightweight character, DrJava maintains an open file cache deleting unmodified documents that can be restored from the file system when the cache gets full.

2.2 The definitions pane

The definitions pane is the main window of DrJava; all program source code (as opposed to transient interactions code used in increment program testing and development) is entered in this pane. To help the developer write syntactically correct code, DrJava presents an enhanced view of the structure of the program through:
• syntax coloring, where different types of text – such as keywords, numbers, strings, and comments – are given different colors, and

• bracket matching, where the text between matching brackets (such as {}, [], or ()) is highlighted when the cursor is placed after a bracket.

Syntax coloring and bracket matching are provided by most programming environments and intelligent editors (like Emacs), but the support for syntax coloring in many environments and editors – particularly lighter weight ones – is incomplete. In particular, both Emacs [8] and BlueJ [2], do not maintain accurate syntax coloring on every keystroke. In these environments, updating the coloring for a line often requires performing a specific event that updates the status of that line. In contrast, DrJava always displays accurate highlighting; it is part of updating the screen to show the inserted or modified text. In Java, changing a single character can affect the coloring of all subsequent text, but only DrJava – among lightweight environments known to JavaPLT – always displays these changes at the time the character is changed.

In addition, DrJava highlights matching brackets by shading all of the text enclosed by brackets in a designated highlight color while other lightweight environments simply enclose or highlight the matching bracket character. The latter convention can be very difficult to see if matching brackets are very far apart.

In the Edit menu, DrJava provides a command for changing the configuration of DrJava including all highlighting colors.

When inserting comments in the code, it is possible to comment or uncomment a whole block by using either a menu option or a key-shortcut. While writing multi-line comments, DrJava conveniently adds spaces and asterisks on every new line automatically.

Additionally, right-clicking the mouse, while over the definitions pane, opens a small utility menu. Besides being able to perform the simple tasks of comment, uncomment, copy, and paste, it is also in this menu that the developer can set breakpoints to be used in the built-in debugger.
The tool bar includes buttons for quick and easy access to commands involving the
definitions pane including operations such as Open, Close, and Save. More noteworthy
are the buttons named Compile All, Test, and Javadoc, which respectively perform
the actions of compiling all documents, running all JUnit tests, and generating Javadoc
reports for all open documents, respectively. DrJava has fully integrated JUnit for writing
and running tests, which makes life much easier for the developer.

2.3 The interactions pane

The interactions pane enables the developer to interact with his program by evaluating
arbitrary expressions and statements, including creating new instances of source program
or library classes and invoking methods on them. DrJava also enables the programmer
to define and use new classes within the interactions pane, but it is generally better to
define new classes as part of the source program so that the definitions persist between
programming sessions.

The evaluation of code in the interactions pane is cumulative so that statements and
expressions can refer to the results of preceding computations. Using this interactions
pane, students can execute program text without defining the infamous public static
void main method. In addition, they can explore the behavior of library classes simply
by creating instances of the classes and applying methods to them. This form of exper-
imentation is particularly helpful in learning how the Java GUI libraries (AWT and Swing)
work.

In contrast, other environments – with the partial exception of BlueJ – require students
to define a class including a main method, compile it, and run it via the main method from
a command line. BlueJ provides an object “bench” that is a weak visual analog of a REPL.
But the visual paradigm is extremely clumsy when dealing with non-trivial experiments
because it lacks the machinery for building interesting lexical environments (collections of
variable bindings). In BlueJ the only available objects are those that are visually displayed
on the workbench and they can only be used in computations by using a pull-down menu
to invoke methods on a displayed object.

From a pedagogic perspective, the interactions pane provides an attractive way to teach simple Java constructs without having to explain the major subset of the language required to support "public static void main" and command line execution. Furthermore, the interactions pane completely eliminates the need for file I/O (and the complex libraries supporting it) in simple exercises.

The interactions pane also provides a wonderful tool for demonstrating new language constructs or library API's in class. The teacher can show students simple interactions that involve the new construct or API. Similarly, students can experiment with the behavior of their own programs using interactions.

The interaction pane relies on DynamicJava to interpret expressions and statements. DynamicJava is described in detail in chapter 3. Of course, an interpreter is not essential to supporting the interactions pane. Two early prototypes for DrJava developed by JavaPLT relied on incremental compilation rather than interpretation to support arbitrary interactions. These prototypes demonstrated the pedagogic utility of the interactions pane but they were very sluggish in comparison to the interpreter based implementation used in the production version of DrJava. The overhead of compiling and loading a class – even a simple one that only performs trivial interaction – is significant.

What is noteworthy about the DynamicJava interpreter is that it adds very little overhead to the execution of compiled code. DynamicJava executes compiled code using reflection. As a result, once DynamicJava transfers control to compiled code, it runs with the same efficiency as conventional compiled Java executed from the command line. As a result, DrJava seamlessly transforms Java from a batch-oriented language to a fully interactive language.

2.4 Other recent extensions to DrJava

While this thesis concentrates on the extension of DrJava to support Java 5.0, this is not the only dimension along which DrJava has recently evolved. Although DrJava makes Java
programming more accessible to beginners than other environments, students still have to learn far more linguistic conventions before they can write simple programs than they do in a functional programming language like Scheme.

DrJava has recently addressed this issue by embracing the concept of *language levels* pioneered in DrScheme [6]. Recent editions of DrJava support a hierarchy of four progressively richer subsets of Java: Elementary, Intermediate, Advanced and Full Java. These levels are designed so that Java constructs can be introduced in a logical progression starting with simple programs that use immutable data. The Elementary and Intermediate language levels automatically generate the routine code associated with implementing classes representing immutable objects. The automatically generated code includes constructors; accessors; `toString`, `equals`, and `hashCode` methods; and access modifiers. With this augmentation, the size of programs that manipulate immutable data becomes competitive with functional languages like Scheme.
Chapter 3

DynamicJava

The primary feature that sets DrJava aside from other programming environments is the way developers can directly interact with their code by cumulatively interpreting arbitrary code fragments in the interactions pane. DynamicJava is the Java interpreter that enables DrJava to interpret arbitrary program text. DynamicJava incorporates the the Java class libraries and closely follows the Java Language Specification [9]. The fidelity to Java semantics makes the existence of DynamicJava almost transparent in the context of DrJava.

There are only a few differences between the semantics of DynamicJava and compiled Java code, most of which are intended to make DynamicJava more convenient to use.

3.1 Differences between DynamicJava and Java

To make it easier to type working code in the interactions pane, DynamicJava varies slightly from regular Java, by being less strict. This section explains the differences the developer can take exploit.

3.1.1 Leaving out the semicolon.

In DynamicJava it is legal to leave out the semicolon at the end of a statement that is an interaction. Doing so will result in the code being evaluated and the result printed out. This convention is much faster and more convenient than having to type out: System.out.println(...); every single time.
3.1.2 Using variables without declaration.

Variables can be used in DynamicJava without first declaring their types. The type of a variable is deduced from the value of which is assigned to the variable. An example of this usage can be seen in code example 1.

```
i = 5;
str = new String("DrJava");
list = new LinkedList();
```

**Code Example 1:** Example of how declarations in DynamicJava are less strict.

The code above, would cause regular Java to complain as the variables i, str, and list has not been defined. But it is legal in DynamicJava.

3.1.3 Non encapsulated code.

DynamicJava also allows the developer to write code segments that are not encapsulated within a class. This becomes very useful when the developer just needs to test a method or a syntax. This leaves more options for the structure of DynamicJava code.

3.1.4 Java code versus DynamicJava code.

A Java program can contain:

- An optional `package` statement
- Optional `import` statement
- One or more `class` or `interface` declarations

Code interpreted by DynamicJava can contain:

- `function` declarations
- `package` statements
- `import` statements
- `statements`
- `expressions`
- `class` and `interface` declarations
The first three of the items mentioned, needs a little further explanation. Java has methods contained within classes, but DynamicJava allows for methods to be declared without being encapsulated by a class, these methods in DynamicJava are called function declarations.

In Java, package and import statements have to appear on the top of the file, import statements following the package statement. DynamicJava allows for the package and import statement to appear anywhere on the top level. Each following package statement, will change the following code to be contained within the new package. Writing a package statement with an already existing package name, will get back to that package. Code Example 2 shows how it is possible to switch between packages as you create code segment.

```java
package p;
class A{
    A(){System.out.println(''Created instance of A'');}
}
package q;
class B{
    B(){System.out.println(''Created instance of B'');}
}
new B();

Output:Created instance of B
new A();

Output:Error: Undefined class 'A'
package p;
new A();

Output:Created instance of A
new B();

Output:Error: Undefined class 'B'
```

**Code Example 2**: An example using the package statement in DynamicJava.
A step by step explanation of example 2:

1) Changing the package to p.
2) Declaration of class A in package p.
3) Switching to package q.
4) Declaration of class B in package q.
5) Creates an instance of B, which calls the constructor.
6) Attempts to create an instance of A, which fails.
7) Switch back to package p.
8) Create an instance of A, which calls the constructor.
9) Attempts to create an instance of B, which fails.

In Java importing two classes with the same name will cause the compiler to report an ambiguity error, as it can not resolve which class to use. In DynamicJava this is allowed, and it is the last class to be imported that counts. Code example 3 shows an example of this.

```java
package p;
public class C{
    public C(){System.out.println("'C in package p'"};}
}
package q;
public class C{
    public C(){System.out.println("'C in package q'"};}
}
import p.C;
new C();

Output:C in p

import q.C;
new C();

Output:C in q
```

**Code Example 3:** An example using the `import` statement in DynamicJava.
A step by step explanation of example 3:

1) Declaring class C in package p, followed by declaration of class C in package q.
2) Importing class C from package p.
3) Creating new instance of class C, constructor is called.
4) Importing class C from package q.
5) Creating new instance of class C, constructor is called.

3.1.5 Dynamic casts are optional

When using collection classes, for example the Vector class, it is very tedious to add the casts when retrieving the objects from the collection. Although with the new feature “auto-boxing” this is being taken care of, DynamicJava originally allowed the assignment without any cast. Normally, leaving out the cast would cause an “incompatible types in assignment” error. If an assignment is made of two unmatching types, DynamicJava would give a ClassCastException anyway, just as if it had been cast to the wrong type. Code example 4 shows an example of leaving out the cast.

```java
import java.util.Vector;
Vector v = new Vector();
v.add(new String("'DrJava'"));
String s = v.get(0);
```

**Code Example 4:** Code segment where the dynamic cast has been left out.

3.1.6 Constant strings equality

When testing for equality of constant strings are treated different. For example:

```
"'DrJava'" == "'DrJava'
```
In DynamicJava this example will return false, where as regular Java will return true. The only way to compare Strings in DynamicJava is through the equals method, also for constant strings*.

3.1.7 Throws clauses are ignored

Throws clauses are ignored in DynamicJava. Again a design decision to make it more convenient for the developer. A large amount of "try-catch" blocks in the interaction pane would completely ruin the quick and swift interaction ability.

3.1.8 Non Java standard for comment allowed

The last difference addressed between DynamicJava and regular Java, is that DynamicJava allows the usage of the character "#" to denote the following text shall be treated as a comment.

It has been explained what DynamicJava is, and how it differs from regular Java. To be able to follow the implementation of the new features in the coming chapters, a little background information about how DynamicJava works is needed.

3.2 Technical Description of DynamicJava

DynamicJava consists of three major components. It consists of a Parser, a ClassLoader and an Interpreter. The interpreter consists of further three components, a NameVisitor, TypeChecker and EvaluationVisitor. The structure of these can be seen on figure 3.1.

*This is one of the future issues to fix for DynamicJava
3.2.1 The parser in DynamicJava

DynamicJava's parser scans and processes all the code that is input through the interactions pane in DrJava. The parser is a recursive descent parser, generated from a BNF-grammar† by the Java Compiler Compiler tool (also known as Javacc). Running Javacc on the grammar results in a Java file, Parser.java. Before Javacc can be run, a grammar to recognize the Java syntax is needed. Grammars are available for Java 1.4, but not for Java 1.5 (5.0). This meant that it was necessary to adapt the old grammar to be able to cope with the new syntax. A clear sign of what it is like to work with something that is on the cutting edge was encountered when working with the Javacc tool. Since Java 5.0 was not even released yet, the tool did not support the syntax for using Generics. To take full advantage of the static type checker that comes with Java 5.0, it was a large goal to find a way to resolve this problem. Raw types could have been used in the grammar file, but that would have led to unchecked warnings during compilation, thereby removing the type safety from the entire code base. Understanding how Generics work, the solution that was found to this problem was to insert comments surrounding the Generic tags, run the tool, and then remove the

†BNF stands for Backus-Naur Form, named from the inventor of this form.
comments again, see example 5 for a small segment of the grammar where this “trick” was used.

```java
//Production for a ReferenceType
ReferenceType ReferenceTypeName():
{
    Token id = null;
    List/*/<IdentifierToken>*/ list =
        new LinkedList/*/<IdentifierToken>*/();
    List/*/<Type>*/ typeArgs = new LinkedList/*/<Type>*/();
    List/*/List<? extends Type>*/ allTypeArgs =
        new LinkedList/*/List<? extends Type>*/();
}
....
```

**Code Example 5:** A small piece of the grammar to show how javacc was ”forced” to accept generics.

The changes made to the grammar to support the new features are explained in detail in chapter 4. The result of running the parser on the code written in the interactions pane in DrJava, is an abstract syntax tree, which the interpreter can traverse to find out if a given syntax is correct.

### 3.2.2 The Interpreter

The Java interpreter‡ contains a method, interpret, which will interpret an abstract syntax tree generated from parsed Java code. To perform this interpretation, three visitors are being used, each traversing the tree to perform different tasks. After the code has been parsed, the NameVisitor is called to traverse the tree. The NameVisitor makes sure that it can traverse the whole tree without any errors, and it defines all the variables in a context class, used to keep information about what variables are defined the which scopes. It also checks for redefinition of variables.

The type checker is also a visitor, and like the NameVisitor it traverses the abstract syntax tree. An instance of the TypeChecker is then created, using another context the

‡Named TreeInterpreter in the code base.
TypeChecker defines all variables and scopes as it traverse the tree in this context. The TypeChecker also checks that the typing rules of the Java language is being upheld and it loads the classes, fields and methods, and sets the properties in the node it is visiting for what type of node it is.

Last, the EvaluationVisitor is instantiated and uses a third context, into which it copies all the scope variables that the TypeChecker set up. The EvaluationVisitor traverse the tree and evaluated each node of the abstract syntax tree, and returns the appropriate value.

If some of the types referred to can not be found in the context, the class loader is used to look for it else where.

3.2.3 The class loader

The interpreter uses its own class loader to load and create classes. As mentioned in the interpreter section 3.2.2 the TypeChecker loads classes while traversing the tree. That is where the class loader is being used.

3.3 DynamicJava's link to DrJava

The last piece of the puzzle as to how DynamicJava works is how it interacts with DrJava. So far it has been explained that it is a stand alone Java interpreter that has been extended and incorporated in DrJava. This section describes how these two pieces interact with each other.

3.3.1 Creating a custom class loader

All computations performed in the interactions pane are solely executed by the DynamicJava interpreter. Where as the code written in the definitions pane is compiled by the Java compiler integrated in DrJava. To create a link between DynamicJava and DrJava, the class loader of DynamicJava was extended to allow class definitions in the definitions pane to be reloaded when they are compiled. A problem emerged since the Java system class loader takes precedence over other class loaders, if using default class loaders. This meant the
class loader extending DynamicJava's class loader would only be asked to load a class if the system class loader couldn't find it. This would indicate that even though a class A was defined in the definitions pane, it would not be the class referred to in the interactions pane if there at JVM start up existed a class A on the system class path. This posed a problem because classes loaded by the system class loader can not be unloaded.

A work around was found, implementing a class loader that doesn't follow the default delegation model for class loaders. Instead, this custom class loader will try to load the class itself, and only delegate to the system class loader if it can not find the class. In more detail, the custom class loader calls the ClassLoader.defineClass() method, which resides in the loaded Class object, but it doesn't perform the byte code reading itself, that is handled by the system class loader still. Since the defineClass() method resides by the Class object, the loaded Classes can be unloaded when the REPL is reset, and a new instance of the custom class loader is created. The loaded classes can now be garbage collected, and unload themselves when this happens [17].

3.3.2 Running DynamicJava in its own JVM

To prevent DrJava from entering a state where it cannot regain control, DrJava runs DynamicJava in a separate "slave" JVM. This means if the DrJava evaluates program text in the interactions pane that makes DynamicJava break or hang, e.g. executing an infinite loop, it is possible for DrJava to recover by killing the old slave JVM and restarting a new slave JVM. Using a separate JVM for interpreting code also enables DrJava to support a full-fledged debugger for compiled code that is executed by DynamicJava in the interactions pane.

DrJava uses Java Remote Method Invocation (RMI) to communicate between the the master JVM and the slave JVM. RMI relies on the TCP/IP protocol for communication between JVMs (even when they are running on the same computer); DrJava will not work if TCP/IP is not installed on the underlying machine. Since all modern operating systems have TCP/IP installed by default, this requirement has not proved to be an impediment to
using DrJava, even on low-end personal machines that are not connected to a network.

Using a slave JVM has not adversely affected the responsiveness of the DrJava interface because the amount of text transferred between the two JVMs is typically quite small. But it does lengthen startup times (since two JVMs must be started). In addition, the use of two JVMs increases the size of the memory footprint of DrJava. Fortunately, with the introduction of Java 5.0, the longer startup time and larger footprint have largely been eliminated because Java 5.0 shares the read only data structures across JVMs. We anticipate that future release of Java will further decrease the startup time and memory resources required for the second JVM. In addition, the power of computer hardware has increased significantly since DrJava was first introduced in the fall of 2001, making the current edition DrJava even more lightweight in practice than the original version with vastly less features (no projects, no language levels, no debugger) that ran in a single JVM.
Chapter 4

Generics

Shortly after the public release of Java in 1995, developers complained that Java did not have a mechanism supporting type abstraction (often called generic typing in the context of OO languages) akin to C++ templates [23]. Without type abstraction, developers were forced to write generic code using weak types (typically Object) instead of type parameters and casting weakly typed results to specific types, e.g. casting a string element of a Vector to type String. In addition, many Java APIs were cluttered with overloaded versions of methods merely to provide more accurate typing.

The incorporation of generic types in Java 5.0 was very well-received by developers familiar with C++ templates. Generic types facilitate writing reusable code – such as the container classes in java.util.* – without the insertion of problematic casts in client code. The generic type system catches type errors at compile time that would otherwise only be caught later in the development process (or perhaps not at all).

Java relies on type erasure to implement generic types. In type erasure, all parametric type information is removed in the compilation process. The generated code uses the same weak type idiom to implement generics that developers used to simulate generic code before Java 5.0. The weak type (common class or interface) bounding each type parameter is substituted for the parameter and casts are inserted where necessary to convert weak types to specific type instantiations. The static type checking performed by the compiler guarantees that the casts inserted by the compilation process can never fail.

Example 6 illustrates a scenario where the manual use of the weak typing idiom produces a runtime error when an element is removed from a collection and cast to an incompatible instantiation type.
```java
Vector v = new Vector();
String s = new String(’’DrJava’’);
v.add(s);
Integer i = (Integer)v.get(0);
```

**Code Example 6**: Without generics, some errors are caught at runtime.

In example 6, the compiler accepts the code without any problems. But at runtime, a `ClassCastException` will occur because a cast to type `Integer` is applied to a `String` with weak type `Object`. In Java 5.0, the `Vector` class is “replaced” by the generic class `Vector<T>` where the type parameter `T` is implicitly bounded by the weak type `Object`. The generic code corresponding to the 6 is given in example 7.

```java
Vector<String> v = new Vector<String>();
String s = new String(’’DrJava’’);
v.add(s);
Integer i = (Integer)v.get(0);
```

**Code Example 7**: The generically type equivalent of example 6.

In this example, the code produces a *compile-time* error because the expression `v.get(0)` has the instantiation type `String` rather than the weak type `Object`.

The byte code for the Java 5.0 generically typed `Vector<T>` class is identical to the byte code for weakly types `Vector` class in earlier versions of Java.* Hence, legacy code can use the new generic version of `Vector` in place of the old weakly typed version.

Java generics have been under development in the programming languages research community since the late 1990’s. In 1998 Martin Odersky released a prototype compiler for generic Java called GJ. Sun Microsystems bought the rights to GJ which became the basis for all subsequent Java compilers (including non-generic compilers prior to Java 5.0)

---

*This assertion is slightly overstated since the Java libraries are only required to preserve interfaces and behavior in new releases. The byte code can change but the changes must be invisible to legacy code. The author has not checked the byte code for `Vector` in Java 5.0 versus its predecessor, Java 1.4.*
starting with Java 1.3. Prior to the release of Java 5.0, Sun released a series of prototype JSR14\(^1\) compilers for generic Java presaging the release of Java 5.0.

In the following sections, the use of Generics is further explained. For example, one of the basic ways to observe the advantages of using Generics can be seen in the Java Collection classes, like the LinkedList, by comparing the code examples 8 and 9.

```
LinkedList myList = new LinkedList();
myList.add(new Integer(5));
Integer i = (Integer) myList.get(0);
```

**Code Example 8:** A code fragment without generics.

```
LinkedList<Integer> myList = new LinkedList<Integer>();
myList.add(new Integer(5));
Integer i = myList.get(0);
```

**Code Example 9:** A code fragment with generics.

In the first code example 8, an object of type Integer with the value 5 is added to the list myList. The object is then removed and cast to the appropriate type (Integer), before assigning it to the variable i. As in all collection classes in java.util.*, the elements in a LinkedList the collection have the weak type Object. When elements are removed from a collection, the developer has to type cast them to the appropriate instantiation type. Although it is generally considered bad programming style, it is possible to mix various subtypes of the weak type Object in these collections. In this case, the developer to perform run-time type checks to determine the instantiation type of each element before casting it to the appropriate one. Aside from the tedium of having to perform repeated casts, it is also unsafe because it places all responsibility on the developer to avoid casting exceptions.

\(^1\)JSR 14 refers to the “Java Specification Request 14” for adding generics under the Java Community Process for extending and revising the Java platform.
In code example 9, the brackets indicate that LinkedList is a generic class that takes a type parameter. Here, it is an Integer. It is read as "a LinkedList of Integers". Using this declaration allow only objects of the type Integer to be added to the list. This feature is a significant departure from the preceding example because the compiler is now able to check that the program is type correct; the source program does not contain any casts that can fault. In other words, it is possible to verify that the LinkedList collection has been used consistently and that correct casts will be inserted when necessary. Thus, the developer's task has been simplified and the source code is easier to read.

Generic Java supports type abstraction over methods as well as classes. A method that is abstracted over one or more type parameters is called a polymorphic method. The term polymorphic means that the same method can have many different instantiation signatures. For example, the following interface for lists includes a zip method that pairs corresponding elements of this and the method argument l to produce a list of pairs:

```java
interface List<T> {
    T first();
    List<T> rest();
    List<T> cons(T elt);
    <U> List<Pair<T,U>> zip(List<U> l);
}
```

**Code Example 10:** A generic List interface with a polymorphic zip method.

In contrast to generic classes, polymorphic (generic) methods do not have to be explicitly instantiated. The compiler can infer the types of the method type parameters for a given call site from the method arguments and the context in which the call site appears. See [4].

An important aspect of generic types in Java is that type abstraction is not covariant. In other words, List<String> is not a subtype of List<Object>. There is no subtyping relationship between different instantiations of the same generic class.

Example 11 below illustrates the relationship between different instantiations of the LinkedList class in the Java collections classes.

The sample program binds a LinkedList<String> to the variable list1 of type
LinkedList<String>. But the variable list2 of type LinkedList<Object> cannot be bound to the value of list1 because LinkedList<String> is not a subtype of LinkedList<Object>. Therefore, the second line in the same program will generate a compiler error.

```java
LinkedList<String> list1 = new LinkedList<String>();
LinkedList<Object> list2 = list1;
```

**Code Example 11:** Example focusing on the fact that List<String> is not a subtype of List<Object>.

In well-written generic Java code, all collection classes should be parameterized with respect to the element type. If a generic type is used without a type parameter, it is called a *raw type*. Raw types do not appear in well-typed generic Java programs but they are often necessary in programs that incorporate or link to legacy code that does not use generic types. Whenever an expression of raw type is used in a context where parametric type information is required for type-checking purposes, the compiler gives an unchecked warning to indicate that it cannot confirm that correctness of the instantiation type for expression. In the absence of legacy code compatibility requirements, unchecked warnings are type errors and should not occur in programs.

### 4.1 Implementing generics in Java compilers

To ensure the backward compatibility of generic Java, it was implemented on the same virtual machine (JVM) as earlier non-generic versions of Java. The GJ and JSR14 prototype compilers ran on Java 1.2, 1.3, and 1.4 platforms.

The GJ (generic Java) compiler developed by Martin Odersky maps generic Java code to conventional Java Virtual Machines (JVM) with no special provisions for generic types by erasing all of the parametric type information from generic types and erasing "naked" type parameters to their weak type bounds. After performing generic type checking, generic Java compilers erase the generic type information in the source code and generate con-
ventional JVM code for the erased source code. For example, every reference to the type `LinkedList<String>` is erased to `LinkedList`. Similarly, within a generic class `LinkedList<T>` where the type parameter `T` is bounded by the type `Object`, every reference to type `T` is erased to type `Object`. After erasure, the compiler inserts a cast around any expression that is not a subtype of the type required in its context [4].

In addition, a generic Java compiler must address the following pathology that arises from erasing the types of method arguments: When a non-generic class `B` extends an instantiated generic class `A<G>` where `G` is a type containing no type variables, the signatures of the inherited methods of `A<T>` (which is erased to `A`) are erased rather than instantiated. For example, a method with `T m()` in `A<T>` erases to `Object m()` in `A<T>`. But the static typing rules dictate that the return type for `m()` is `G`. The class `B` may implement an interface containing a method `G m()`, but the method `m()` inherited from the erased class does not have the proper return type even though the instantiation of the signature of `m()` exactly matches the interface method. Generic Java overcome this problem by generating a bridge method in `B` with the instantiated signature that does a super call on the inherited method `m()` and casts the result to type `G`. If `B` overrides method `m`, then the overriding code body is used instead of the super call and the erased version of `m()` is overridden to invoke the bridge method for `m()`. If the method in question takes arguments, the forwarding call may have to cast the arguments to the appropriate instantiation types [4].

Code example 12 shows a elided program fragment using generics as it might appear in program source text.

Example 13 shows how the type parameters are erased, and replaced by their bounds. `T` is replaced by `Number`, because `T` has `Number` as its bound. In addition, the erased source code shows a cast `(String)` in the body of method `m` so that an element is extracted from a `List` has the type required by its context.
class A<T extends Number> {
    T x;
    List<String> l = new LinkedList<String>();

    A(String s) { l.add(s); }

    <S, U extends List<T>> void m(S y, U aList){
        String str = l.get(0);
        ....
    }
}

**Code Example 12:** The code before erasure

class A {
    Number x;
    List l = new LinkedList();

    A(String s) { l.add(s); }

    void m(Object y, List aList){
        String str = (String) l.get(0);
        ....
    }
}

**Code Example 13:** The code after erasure

### 4.1.1 Specific generic Java extensions

The following subsections describe the specific grammatical constructions where generic Java extends conventional Java. In essence, each section deals with a different piece of syntax where generic (parameterized) types can be encountered.

#### 4.1.2 Class and interface declarations

When declaring a class or interface using formal type parameters, the developer can abstract the class with respect to a collection of type parameters and specify a bound for each type parameter. If no bound is given, `Object` is the default. For an example of a generic class declaration and a generic interface declaration, see code example 14.
class A<T> {
    T myVariable;
}
interface B<E>{
    void add(E x);
}

**Code Example 14:** Class and Interface declaration using generics.

All occurrences of the formal type parameters (T and E in the examples above) are replaced by the actual type argument when used. For example, in code example 14 above, the type parameter E in the declaration of method add is replaced by the actual type argument for E.

The following is a program statement that declares a new variable of type A<String> and binds it to a new instance of the class A:
A<String> a = new A<String>();

In the class instantiation A<String> all occurrences of the type T in the generic class A<T> are replaced by the type String.

It is important to note that no copies of the code for a generic class are generated when the class is instantiated. A generic type declaration is compiled only once, producing only one class file, namely the class file for the erasure of the generic class.

### 4.1.3 Polymorphic method declarations

In a polymorphic method declaration, the type parameters are introduced in a bracketed list that appears before the return type in the declaration. The scope of the parameters is the entire method declaration including the return type. See example 15.

What might seem unclear at first, is that it is not necessary to pass an actual type parameter when calling the method. Rather, the type can be inferred by the compiler, leaving it optional.
class A{
    <T> T myMethod(T x, int y){
        T myVariable = x;
        return myVariable;
    }
}

Usage:
A a1 = new A();
System.out.println(a1.myMethod("str", 5).getClass());

**Output:** class java.lang.String

A a2 = new A();
System.out.println(a2.<String>myMethod("str", 5).getClass());

**Output:** class java.lang.String

**Code Example 15:** An example of a polymorphic method declaration.

4.1.4 Polymorphic constructor declaration

Generic Java accommodates polymorphic constructors in essentially the same fashion as polymorphic methods, but they are rarely used because the genericity in constructors is typically due entirely to the genericity of the corresponding class. The constructor for such a class is not polymorphic because it takes no additional type parameters.

Polymorphic constructors are declared similarly to the polymorphic methods. The contrived example 16 shows how a polymorphic constructor could be declared in a class A<S>.
```java
class A<S> {
    <T> A(T x, S y) {
        System.out.println("x is " + x.getClass() + " and y is " + y.getClass());
    }
}

Usage:
A<String> a = new <Integer> A<String>(new Integer(5), "str");

Output:
```
x is class java.lang.Integer and y is class java.lang.String
```

**Code Example 16**: An example of a polymorphic constructor declaration.

### 4.1.5 Extends clause

The instantiation of a type parameter T can be restricted by following the binding (first) occurrence of a type parameter by the keyword extends followed by a type, possibly involving T. Then all instantiations of the parameter T must conform to this constraint. For example, a `Matrix` class might required all of its elements to be subtypes of `Number`. The header for such a class would have the form `Matrix<T extends Number>`. Such a `Matrix` class could be instantiated as `Matrix<Double>` but not `Matrix<String>`. Code example 17 shows a program fragment using bounded type abstraction.

In example 17, three variables are declared inside method `m` in class `E`. The first two declarations are legal, because `B` and `C` are subtypes of `A`. But the third declaration is illegal because `A` is not a subtype of `B`.

### 4.1.6 Wild cards

Wild cards are a result of a joint project between Sun Microsystems, and The University of Aarhus, Denmark [22]. Specifically, wild cards were designed to increase the flexibility of the type system with parameterized classes like Java has with Generics. Wild cards permit
class A{}
class B extends A{}
class C extends B{}
class D<T extends B>{}
class E{
    void m(){
        D<B> d1 = new D<B>();
        D<C> d2 = new D<C>();
        D<A> d3 = new D<A>();
    }
}

**Code Example 17:** An example of usage with the extends keyword.

the developer to leave the type parameter unbound. Thus, it is indicated that there is a type parameter, but that the type parameter is without a specific binding. Wild cards are a way to provide a type that abstracts over different parameterizations of the same class, where "?" is used to denote the unspecified type arguments.

From Generics, however, we know that it is allowed for a class LinkedList to be parameterized with different types. For example LinkedList<String> and LinkedList<Integer>. As in the example above, giving the LinkedList the type parameter "?", the wild card, is the special type argument that can be all possible type arguments. This means that LinkedList<?> is the type of all LinkedLists no matter what their element type is. See figure 4.1.

![Figure 4.1: Depicting LinkedList<?> as super type of all LinkedList.](image)
4.1.7 Wild card types

Wild card types are a recent addition to generic Java. They were not included in GJ or earlier JSR14 compilers. A wild card type is a type instantiation that uses the special symbol \( \square \) as a type argument followed by an optional extends or super clause. In the absence of a qualifying super or extends clause, a wildcard type means the union of all possible instantiations of the type parameter. In other words, \( \text{List}\langle \square \rangle \) means the union of all types \( \text{List}\langle E \rangle \) where \( E \) is any type (other than \( \square \) with no type variables. Wild card types give less information than conventional instantiation types because the binding of the corresponding type parameter is unknown.

Qualified wildcard arguments have the syntax \( \square \) extends \( T \) or \( \square \) super \( T \). The former refers to the union of all instantiations that are subtypes of \( T \) and the latter refers to the union of all instantiations that are supertypes of \( T \). See figures 4.2 and 4.3. Note that \( \text{LinkedList}\langle ? \rangle \) is equivalent with \( \text{LinkedList}\langle ? \text{ extends Object}\rangle \).

![Figure 4.2: Depicting an example hierarchy using upper bounds](image)

Although wild card types can be very useful in certain special situations, they have very strict type-checking rules which make them difficult to use. Code example 18 illustrates this problem.

In the example, the method \texttt{newDog} takes a \texttt{LinkedList} of some unknown subtype of \texttt{Dog} and adds a new Labrador to this list. Note that the input list must be a list of objects of type \texttt{Dog}. So it superficially looks reasonable to add a \texttt{Labrador} to such a list. But this conjecture is wrong. The attempt to add the \texttt{Labrador} to the list is wrong and generates
class Dog{}
class Poodle extends Dog{}
class Labrador extends Dog{}

class MyClass {
    void newDog(LinkedList<? extends Dog> list) {
        Dog d = new Labrador();
        list.add(d);
    }
}

**Code Example 18:** Example of bounded wildcards.

a compiler error. Why? If someone passes a LinkedList<Poodle> to newDog, then a Labrador will be added to the list breaking the type invariant for the list which asserts that all elements of the list are of type Poodle.

In short, it is illegal to write to a list of wildcard type with an upper bound. (Recall that ? in isolation abbreviates extends Object.) In essence, the list is “read only” while it is viewed as a member of a wildcard type with an upper bound.

Similarly, it is illegal to read from a list of wildcard type with a lower bound with one important exception. The type LinkedList<? super Integer> is union of all types LinkedList<T> where T is a supertype of Integer. If an element is extracted from such a list, what is its type? Nothing is known about such an object other than the fact that it is an Object. Hence, elements that are read from an object of super wildcard
type can only be typed as `Object` which is usually not very helpful. On the other hand, operations that write to such wildcard types are generally legal as long as the written values belong to the specified bounding type. For example the code in example 18 is legal and useful.

```java
class Dog{}
class Poodle extends Dog{}
class Labrador extends Dog{}
class MyClass {
    void newDog(LinkedList<? super Labrador> list) {
        Labrador d = new Labrador();
        list.add(d);
    }
}
```

**Code Example 19:** Example of bounded wildcards.

Figure 19 shows a usage of the `super` bounded wildcard type. The `newDog` method adds a `Labrador` to a list of any supertype of `Labrador` including `Dog`.

To date, `super` bounded wildcard types have not been used much in generic Java code. The observant reader might ask why we cannot use `super` in specifying the bounds of type parameters in generic class and polymorphic methods. For example, the `newDog` method could ostensibly be rewritten as a polymorphic method with the header

```java
<T super Labrador> void newDog(LinkedList<T> list) {
```

Neil Gafter, who was the primary developer of the Java 5.0 compiler and most of the JSR14 prototype compilers that preceded it gave the following short explanation for banning `super` in specifying the bounds of type parameters: "The short answer is that we didn’t add it to the syntax because we don’t know of a sound type inference algorithm for super-bounded type variables.” Recall that generic Java uses type inference to determine the instantiation of polymorphic method variables.
4.1.8 Multiple type bounds

Another late addition to generic Java was an option to specify multiple type bounds for a type parameter. All of the bounds after the first must be interfaces since Java does not support multiple class inheritance.

```
interface I{}
class A{}
class MyClass<T extends A & I>{}
```

Used like:

class SomeClass extends A implements I{}
MyClass<SomeClass> mClass = new MyClass<SomeClass>();
```

Code Example 20: Example where the upper bound is a type that extends two types.
```

Example 20 declares a class MyClass with a doubly bounded type parameter. The & character is used to separate types in the list of bounds. Note that this character is distinct from from the usual "," separator character. Otherwise it would be impossible to decide if a type was second bound for the preceding type parameter or another type parameter.

4.1.9 No genericity for primitive types

In all of the examples presented, type parameter have been bound to object types and never to primitive types. Instantiating type parameters with primitive type is incompatible with using type erasure to implement generics. This inclusion would also raise a number of issues concerning how to incorporate primitive types in the object type hierarchy so that the same parameterized is meaningful when a type parameter is bound to a primitive type and when it is bound to an object type.

The lack of support for generic primitive types is mitigated by another new feature in Java 1.5, namely autoboxing. By using autoboxing, developers can write code that appears to use primitive types as bindings for generic type parameters, but the corresponding boxed type (e.g., Integer) is actually bound to the type parameter “matching” the primitive
type. In such cases, primitive values are automatically boxed and unboxed when needed. For more about autoboxing, see Chapter 8.

4.2 Implementation of generics in DynamicJava

Before adding these new features to DynamicJava, we had to determine what functionality DrJava required of DynamicJava to support Java 5.0 generics. DrJava differed from other IDEs in providing limited support for the prototype generic Java compilers. In fact, DrJava was written from its inception (as an open source all Java application in 2001) in generic Java. The earliest versions of DrJava were compiled using GJ and subsequent versions were compiled using the JSR14 compilers prior to the release of beta versions of Java 5.0. For this reason, DrJava has always supported the editing and compiling of generic Java code. The interpreter was limited to executing erased code but that issue was finessed by adding a preprocessor that accepted generically typed code fragments, i.e., syntax including parameterized types, and simply erased the parametric type information. Hence, the interpreter did not perform generic type checking but that was not a major issue because DynamicJava does not perform any more static typing than is necessary to correctly resolve static overloading.

The design philosophy underlying both DynamicJava and the interactions pane is that disposable code – in contrast to persistent applications code – should be executed rather than rejected if any sense can be made of it. The code entered in the interactions pane is not archived for subsequent reuse. Hence, the underlying interpreter should be as forgiving as possible.

The preprocessor parsed generic code and built an abstract syntax tree including the

---

The principal argument against this practice in an instructional setting is that students may be misled about the behavior of compiled Java. In our experience, this has not proven to be a problem because the majority of the code that students write is compiled by a Java compiler so they learn the restrictions imposed by the compiler from day one. As long as the interactions pane interprets syntactically legal code correctly and treats nearly correct code sensibly, we do not see a pedagogic problem.
generic type information. After the code was parsed, an Eraser visitor walked the tree to erase all parametric type information. In the process, it detected many syntax error and generated better error diagnostics than the same code would have elicited from DynamicJava. If the erasure process did not detect any fatal errors, the preprocessor walked the tree again to produce a conventional string representation for the erased code, which DrJava subsequently passed to DynamicJava. DynamicJava reparsed the erased source code into a (different) abstract syntax tree, and executed it.

In short, the main function of the preprocessor was to convert generic Java code to equivalent conventional Java code. In addition, the preprocessor produced better syntax diagnostics than DynamicJava.

Figure 4.4: Illustration of the extra overhead due to the preprocessor.

Figure 4.4 depicts that sequence of steps involved in processing an interaction and vividly shows the unnecessary overhead involved in parsing and analyzing program text twice. Clearly, the responsiveness of the interactions pane in DrJava could be improved if these two parsing and analysis stages are collapsed into a single stage. There were two possible ways to accomplish this optimization:
The parser and analyzer from the preprocessor could be moved over to DynamicJava enabling DynamicJava to process generic Java program text and producing better error diagnostics.

The parser and syntax analyzer in DynamicJava could be extended to deal with generic Java code, in essence subsuming the functions of the preprocessor.

The first option was rejected because the AST built by the preprocessor was quite different from the AST built by DynamicJava. The AST in the preprocessor was created using a tool called ASTGEN developed by our research group. On the other hand, the AST in DynamicJava was created manually. To modify DynamicJava to use the AST format generated by ASTGEN would have required a massive revision of DynamicJava.

As a result, the DynamicJava parser was revised to cope with generic Java code and to produce better error diagnostics. To maintain compatibility with the rest of the DynamicJava interpreter, the original DynamicJava AST composite class hierarchy was extended by subclassing node types to add generic type information where necessary. Viewing these new node classes using the old superclass type interfaces implicitly erases the generic type information from the nodes. This use of inheritance to implicitly erase generic type information was most important design technique used in revising the architecture of DynamicJava.

4.2.1 Revising the DynamicJavaParser

Figure 4.5 shows how this revision of DrJava streamlined the interpretation of generic Java code.

The additional AST nodes that was created (as extensions of existing node classes) were GenericClassDeclaration, GenericInterfaceDeclaration, PolyMorphicMethodDeclaration and PolyMorphicConstructorDeclaration.
Generic Class Declarations

To illustrate this process, let us focus on the AST representation of generic class declarations. To represent such classes a `GenericClassDeclaration` node class extending the existing `ClassDeclaration` node class was created. Code example 21 outlines the structure of this new class.

```java
public class GenericClassDeclaration extends ClassDeclaration{
    private TypeParameter[] _typeParameters;

    public GenericClassDeclaration(...,
        TypeParameters[] typeParams) {
        super(...);
        _typeParameters = typeParams;
    }
}
```

**Code Example 21:** Defining the `GenericClassDeclaration` by extending the regular `ClassDeclaration` class.

The dots in code example 21 indicate where the formal parameters are the same as in
the ClassDeclaration. The only additional information in a GenericClassDeclaration was
the list of generic type parameters (including their bounds) for the class in question.

To support the new AST nodes, we had to change the parser grammar to generate them.
The DynamicJava parser is automatically generated from grammar and action tables by a
parser generator called JavaCC (short for Java Compiler Compiler). This tool is described
in more detail in the tools chapter.

See the code example 22 to see the core of the grammar used to parse generic class
declarations. The grammar is written in the form that JavaCC accepts.

Generic classes are declared as:

[modifiers] class 'identifier' [type parameters] [extends clause]
[implement clause] { [class_body] }

Where: [] indicate optional
      ' ' indicate that one must be present
      chars written without ' ' or [] around, means they are
      present and doesn't change.

```
t="class" id=<IDENTIFIER>
  [ typeParameters = TypeParameters() ]
  [ "extends" ext=ReferenceTypeName() ]
  [ "implements" impl=ReferenceTypeNameList() ]
  body=classBody()
  {
    if (typeParameters != null)
      return new GenericClassDeclaration(...., typeParameters);
  }
else {
  return new ClassDeclaration(....)
  }
```

**Code Example 22:** The grammar used in our Parser to handle the syntax for GenericClass-
Declarations.

There are a few additional checks in the grammar that are omitted in this example.
Those checks concern the modifier flags, but there are no differences in the way they are
handled when parsing a generic class declaration as compared to a regular class declaration. The handling of the modifier flags has therefore been omitted for clarity.

This part of the grammar can be read informally as follows: When the parser encounters the keyword `class`, followed by some "identifier" representing the class name, then check whether there will be any `typeParameters` following the class. The type parameters could for example be `<S, T>`. Also check if the class extends another class, and if the class implements some interfaces. After this check, parse the body of the class. If any `typeParameters` were encountered, return a new `GenericClassDeclarationNode`, otherwise just return a regular `ClassDeclarationNode`.

**Generic Interface Declaration**

The `GenericInterfaceDeclaration` node class extends the original `InterfaceDeclaration` class in almost exactly the same way that that the `GenericClassDeclaration` node class extends the original `ClassDeclaration` node class. Again the only noteworthy difference between the generic generalization and the original is the inclusion of type parameterization information as a node attribute. A code skeleton for this class is given in code example 23.

```java
public class GenericInterfaceDeclaration
    extends InterfaceDeclaration {

    private TypeParameter[] _typeParameters;

    public GenericInterfaceDeclaration(....,
            TypeParameter[] typeParams){
        super(....);
        _typeParameters = typeParams;
    }
}
```

**Code Example 23**: Defining the `GenericInterfaceDeclaration` by extending the regular `InterfaceDeclaration` class.

After the new node has been created, the grammar needs to be adapted. Adapting the
grammar for parsing a regular interface declarations, is done similarly to parsing a generic class declaration. The most interesting part of the grammar is shown in code example 24.

Generic interfaces are declared as:

```
[modifiers] interface 'identifier' [type parameters]
[extends clause] [implement clause] { [class_body] }
```

Where: [] indicate optional

'' indicate that one must be present

chars written without '' or [] around, means they are present and doesn't change.

```
t="interface" id=<IDENTIFIER>
[ typeParameters = TypeParameters() ]
[ "extends" impl=ReferenceTypeNameList() ]
"{"*
  (decl=interfaceMemberDeclaration(){
    list.addAll(decl);
  }
  )*
 e="*/
}{
  if (typeParameters == null){
    return new InterfaceDeclaration(....)
  }
  else {
    return new GenericInterfaceDeclaration(...., typeParameters);
  }
}
```

**Code Example 24:** The grammar used in our parser to handle the syntax for GenericInterfaceDeclarations.

The fragment from the grammar shown in code example 24 reads as: In case the parser encounters the keyword interface followed by some "identifier", which is the name of the interface. Then check whether or not there will be any typeParameters following the interface, or whether the interface extends other interfaces. Then parse all member declarations and add them to a list. This is done repeatedly until no more member declaration
exists. If the typeParameters were encountered, then return a new GenericInterfaceDeclaration. If not, just return a regular InterfaceDeclaration.

**PolyMorphicMethodDeclaration**

The PolymorphicMethodDeclaration node class extends the MethodDeclaration node class in essentially the same way as the two previous examples. The skeleton of the new class is shown in code example 25.

```java
public class PolymorphicMethodDeclaration
    extends MethodDeclaration {

    private TypeParameter[] _typeParameters;

    public PolymorphicMethodDeclaration(......,
        TypeParameter[] typeParams){
        super(......);
        _typeParameters = typeParams;
    }
}
```

**Code Example 25:** Defining the PolymorphicMethodDeclaration by extending the regular MethodDeclaration class.

Polymorphic Methods are declared as:

```
[modifiers] [type parameter] 'return type' 'identifier'
([formal parameters]) [throws clauses] { [method body] }
```

Where: [] indicate optional

' ' indicate that one must be present
chars written without ' ' or [] around, means they are present and doesn't change.

The PolymorphicMethodDeclaration node class involves a few interesting additional issues not present in two preceding generic AST node classes. The grammar in code example 26 informally reads as: (i) check for the optional type parameters, then parse the result type, followed by the name of the method; (ii) parse the list of the formal parameters. Since an array can be declared in two ways: int[] a; or int a[], a check for the brackets[,
Code Example 26: The grammar used in our parser to handle the syntax for PolymorphicMethodDeclarations.

needs to be done, to see if the return type of a method is an array type. Next the parser need to check if the throws clause is there. Since methods can throw several exceptions all of these are gathered in a list. The method body, which can possibly be empty, is then parsed. If the last parameter is a variable argument, indicate this by setting a flag, to be used later when dealing with variable arguments. Variable arguments is another of the new features of Java 1.5, which is described in detail in chapter 6. If no type parameters were found, return a regular MethodDeclaration node, otherwise return a polymorphic method declaration node.

Note that a special flag is set if the last type parameter is variable arguments, and that the provided method from Sun is not used. Sun provided the constant:
java.lang.reflect.Modifier.VARARGS == 0x00000080. Since the version of Java 1.5 was yet to be released in a full version, not everything was working quite right.

**PolyMorphicConstructorDeclaration**

The new PolyMorphicConstructorDeclaration node class extends the ConstructorDeclaration class in essentially the same way as the preceding generic AST classes extend their non-generic analogs.

A code skeleton for the PolyMorphicConstructorDeclaration class is given in code example 27.

```java
public class PolyMorphicConstructorDeclaration extends ConstructorDeclaration {
    private TypeParameter[] _typeParameters;

    public PolyMorphicConstructorDeclaration(...., TypeParameter[]
        typeParams) {
        super(....);
        _typeParameters = typeParams;
    }
}
```

**Code Example 27**: Defining the PolyMorphicConstructorDeclaration by extending the regular ConstructorDeclaration class.

Polymorphic constructors are declared as:

```
[modifiers] [type parameter] ‘identifier’ { [formal parameters] }
[throws clauses] { [constructor body]}
```

Where: [] indicate optional

'' indicate that one must be present
chars written without '' or [] around, means they are present and doesn’t change.

A polymorphic constructor is very similar to a polymorphic method declaration, with a few minor differences. First, it has no return type. Second, it can have explicit constructor invocations at the beginning of the body.
\{ 
  [ typeParameters = TypeParameters() ]
  id=IDENTIFIER
  params=formalParameters()
  [ "throws" exceptions=ReferenceTypeNameList() ]
  ""
  [ LOOKAHEAD( explicitConstructorInvocationLookahead() )
  ci=explicitConstructorInvocation() ]
  ( stmt=blockStatement() )
  
  stmts.addAll(stmt);
  
  }*
  
e="}"
  
  [ // Modifier.VARARGS == 0x00000080 /**/
  if(lastFormalParameterIsVarArgs) flag |= 0x00000080;
  if (typeParameters == null){
    return new ConstructorDeclaration(flag, params,
          exceptions, ci,
          stmts,....); 
  }
  else {
    return new PolymorphicConstructorDeclaration(flag, params,
          exceptions, ci, stmts,......, typeParameters);
  }
  ]
\}

**Code Example 28:** The grammar used in the parser to handle the syntax for PolymorphicConstructorDeclarations.

The grammar in example 27 informally reads as: check for the optional type parameters, and then parse the constructor name. The list of formal parameters is handled next, followed by any possible throws clauses. All exceptions encountered are stored in a list. The "LOOKAHEAD" is checking to see if there is any explicit constructor calls, such as "super()" or "this()" as it is known they should appear on the first line of the constructor body. All encountered statements in the constructor body is stored in a list. A check is then made for variable arguments, and a flag is set to be used when handling variable arguments. If no type parameters were encountered, a regular constructor declaration node is returned. If not, then return a polymorphic constructor declaration node.

To represent the difference between an ordinary ReferenceType (such as `Vector` and a
generic ReferenceType (such as Vector<String>), the we extended the ReferenceType class to define the GenericReferenceType class. Although GenericReferenceTypes are handled in the interpreter just like ordinary ReferenceTypes, the new class keeps track of the erased type information. Code example 29 shows a skeleton of the GenericReferenceType node class.

```java
public class GenericReferenceType extends ReferenceType{
    private List<List<? extends Type>> _typeArguments;
    public GenericReferenceType(....,
        List<List<? extends Type>> typeArgs){
        super(....);
        _typeArguments = typeArgs;
    }
    public List<List<? extends Type>> getTypeArguments(){
        return _typeArguments;
    }
}
```

**Code Example 29:** GenericReferenceTypes stores the erased type information.

The "..." in the code example showing the GenericReferenceType class(example 29), elides the formal parameters that a ordinary ReferenceType class takes. The parser could return a ReferenceType instead of returning a GenericReferenceType since the extra type information in the latter has no impact on interpretation. But such a design choice would have precluded extending DynamicJava to support a modicum of generic type propagation, which is required to implement method resolution (including operator overloading) with full fidelity. We anticipate that this information will be exploited in a subsequent extension of DynamicJava to address our implementations failure to propagate generic type information.
Chapter 5

Static import

The next new feature discussed in this dissertation is static import. Like classes and interfaces can be imported from packages, static import makes it possible to import static methods and fields. When using mathematical methods such as abs and sqrt, are no longer forced to type Math.abs() and Math.sqrt with every use. Code example 30 shows how the static import feature can be used.

```java
import static java.lang.Math.*;

class A {
    void printAbsOf(int x) {
        System.out.println(abs(x));
    }
}
```

**Code Example 30**: A small example of using static import.

It is possible to use static import to import a static member explicitly: import static java.lang.Math.abs;. Static import also provides the ability to import on demand with import static java.lang.Math.*. It is important to note that the class that the static members are imported from is not imported itself. In other words, when importing static methods from a specific class, it is not possible to make any reference to the class.
The static import imports all static members:

- static inner classes
- static methods
- static fields

It is even possible that the above members may all have the same name. This is no problem because it is possible to derive which of the encounters of the referenced name should be used. It all depends on what the developer is attempting to use the imported members for. Like with non static import, explicit importation takes precedence over an import on demand.

5.1 Implementation of static import

When it comes to implementation of these features for DynamicJava, it is important to remember that there are some differences from regular Java. There will therefore be a slight variation of the use of some of the features. When dealing with static import there is one small occurrence where it differs.

In regular Java, when two static import statements are importing something with the same name explicitly, the compiler gives an ambiguity error. The developer then must go back to his program and fix the error and recompile the application. But when using DynamicJava there is no compilation. It is therefore not possible to go back and change what was declared or imported previously. For this reason, it was decided that when such two import statements would have the ambiguity conflict, the most recent import statement is the one that counts.

To implement static import, the parser needed to be adapted, so it can deal with the new syntax. The most interesting part of the addition to the grammar to support this feature is shown in code example 31.
ImportDeclaration importDeclaration() :
{
    ReferenceType name = null;
    Token star = null;
    Token t1, t2;
    Token sttic = null;
}
try {
    t1="import" [sttic="static"] [name=ReferenceTypeName()]
    [^".* star="*"| t2=<EOF>)
    {
        if(name instanceof GenericReferenceType){
            _throwParseException("Import names cannot be parameterized.");
        }
        if(name == null) {
            _throwParseException("Missing name - Cannot import");
        }
        return new ImportDeclaration(name.getMethodRepresentation(),
                                    star != null,
                                    sttic != null, ....);
    }
    catch(ParseException pe) {
        _throwParseException(pe,"Invalid Import Declaration");
    }
}

Code Example 31: The grammar used in the parser to handle the syntax for static import.

5.1.1 Parser adaption

The option of the word static after the import statement, is an interesting part of the grammar shown in example 31. After the option of having the static keyword, a reference name should follow. It has been made an option in the grammar here to be able to provide a better error message. Without it, the message would just have been “Invalid Import Declaration”.

After the ReferenceTypeName, which is the name of the Class or the member to be imported there can either be a ‘.*’ which indicates importation from a class on demand. Alternatively, there can be a semicolon to signal the end of the import statement. In DynamicJava the semicolon at the end of the statement may be omitted. In such case, the program will merely continue on the next line. Another addition to the grammar, to handle
static import, is the check for a GenericReferenceType. This check is required since having type parameters in the import statement is not allowed. If the import statement is correctly parsed, an ImportDeclaration node will be returned, with one of the parameters indicating whether it was a static import or not.

Now that the parser is ready to accept static import statements the ImportationManager needs a small update. The ImportationManager is a class that, as the name indicates, handles import statements. Before implementing the static import, it contained two Lists. One list for the classes, which was used for the single type imports. For example `import java.util.LinkedList`. And a second list for packages, to be used for importation by demand. For example `import java.util.*`. When looking up a class, it will first look through the list used for classes, and then the list for packages.

To support the static imports, 3 new lists were added to the Importation manager:

- Import on Demand Static
- Single Type Import Static Field
- Single Type Import Static Method

The first list, is to hold all class names of those that were used with the import on demand. Each time this list is being gone through, reflection is used on each of the classes to see if they have any static members of the specified name.

```java
import static myPackage.MyClass.*/
import static java.lang.Math.*/

class A{
    void printAbs(int x){
        System.out.println(abs(x));
    }
}
```

**Code Example 32**: Static import gives allows nicer and easier syntax.

When the method `abs(x)` is called, refer to code example 32, the local context is checked for the method, in case it should be defined in the scope. If it is not found
in the context, the ImportationManager will look through its lists to find the fully qualified class name `java.lang.Math` and the class name `myPackage.MyClass` in the `ImportOnDemandStatic list`. Reflection will be used on each of the class names in the list, until the name of the method has been resolved or none is found, in which case a "non such method error" is thrown.

To ensure the small detail mentioned earlier, that the latest imported statement should count, in case two import statements imported a member with the same name. Each time a class name is added to the list, it is added to the beginning of the list. And the first encountered name that match is used, in this case the method call `abs(x)` will be replaced with `java.lang.Math.abs(x)`, since the `abs` method will be found in `java.lang.Math`.

The second list `SingleTypeImportStaticField`, holds field objects of the statically imported fields.

```java
package myPackage;
public class Color{

}

package myPackage;
public class MyClass{
    public static final Color GREEN = new Color();
}

import myPackage.Color;
import static myPackage.MyClass.GREEN;

class A{
    public Color someColor = GREEN;
}
```

**Code Example 33**: Example of importing a static field

In code example 33, the ImportationManager adds the field `GREEN` to the `SingleTypeImportStaticField` list. When the Field is used, it looks at each Field in the list, and uses the `getName()` method to see if the field has the name `GREEN`. If it does, `getDeclaringClass()` is called on the Field and it is thereby possible to obtain a
fully qualified name fo the Field, in this case `myPackage.MyClass.GREEN` which can replace the location in the code where `GREEN` appears.

The third of the newly added lists `SingleTypeImportStaticMethod`, works precisely like our list for Fields, the only difference is that it holds Methods instead of Fields. The reason for the two lists are seperated is to get faster lookups with shorter lists.

The order in which lookups will be performed is as follows:

- Look through the local context.
- Look through the `SingleTypeImportClauses` list (contains specific classes)
- Look through the Package List (import on demand list)
- Look through the `SingleTypeImportStaticField`
- Look through the `SingleTypeImportStaticMethod`
- Look through the Static Import on demand
Chapter 6

Variable Arguments

Adding a variable length of arguments of methods and constructors is in short what variable arguments is all about. This enables the opportunity to declare a method or constructor that takes any amount of parameters of the same type. They can be used in method declarations, including the main method, and constructors. Taking advantage of variable arguments it is possible for the developer in some cases to limit the number of methods and constructors. For example in the case where the developer needed a method to take one String as parameter, another method that takes two Strings, and a third method that takes three Strings. In this example it would be possible to settle for one method instead of three with the use of variable arguments. It is important to understand that the adding variable arguments does nothing beneath the surface, it is all handled at compile time. During compile time the variable arguments are actually converted into an array. This is precisely what the implementation of variable arguments are taking advantage of.

Code example 34 shows how an example of the syntax used for variable arguments.

Notice the special syntax for indicating the variable arguments “...”

Variable argument can be thought of as having an array, it works precisely the same as if you had passed in an array of ints containing those three numbers.

A rule for using variable arguments is that they always need to appear as the last argument of the parameters. This also means that there can not be several variable arguments in the same list of parameters of a method.

To implement this new feature, the parser needs to be updated where method declarations and constructor declarations are handled.
class A{
    void someMethod(String s, int ... n){
        for(int i=0; i<n.length(); i++){
            System.out.println(s+n);
        }
    }
}

Usage:
A a = new A();
a.someMethod('SomeString ',1, 2, 3);

Output:
SomeString 1
SomeString 2
SomeString 3

Code Example 34: A simple example using variable arguments.

6.1 Implementation of Variable Arguments

A part of the grammar, which handles the variable arguments when looking through the formal parameters in method and constructor declarations, is shown in example 35.

The grammar in example 35 performs a look-ahead to see if some formal parameters exist. If they do, they are all added to the list as formal parameters, except the last parameter. When the last formal parameter is reached the function varArgsFormalParameter() is called, which checks if the last parameter consists of variable arguments or if it is a regular formal parameter. If it consists of variable arguments, an ArrayType is passed back and added to the list of formal parameters, otherwise it is simply added to the list.

Besides altering the parser, a small method isVarArgs() was added to the Method-Declaration Node. When a MethodDeclaration is encountered and the isVarArgs() method can indicate there are any Variable Arguments. This indication is kept track of in the access flags, and by performing an "and" operation on it with an indication number (namely 0x000000080), it can be known if the method has variable arguments as parameter. The small method is shown in code example 36.
"{"  
    [ LOOKAHEAD(formalParameterLookahead())
    node=formalParameter()
    {
      list.add(node);
    }
    ( ",", node=formalParameter()
    {
      list.add(node);
    }
  }]*
} [lastVarArgsParam = varArgsFormalParameter() ] "}"
{
  lastFormalParameterIsVarArgs = (lastVarArgsParam != null);
  if(lastFormalParameterIsVarArgs) {
    list.add(lastVarArgsParam);
  }
  return list;
}

Code Example 35: Segment of the grammar to handle variable arguments syntax

public boolean isVarArgs(){
    return (accessFlags & 0x00000080) != 0;
}

Code Example 36: The method to check if a method has variable arguments as a parameter.

When Java 1.5 is no longer in its beta stage, it should be possible to use java.lang.reflect.Modifier.VARARGS instead, which will hold the indication number directly, but until then this work around is acceptable.

It is important to note that for evaluation, it is enough to build an array containing the variable arguments in each case they can be encountered*. Once the variable arguments has been converted into an array, the interpreter can perform the evaluation as it supports arrays already.

*For DrJava developers interested, this is done in the Evaluation Visitor when handling ObjectMethodCall, SuperMethodCall, StaticMethodCall, SimpleAllocation and ClassAllocation
Chapter 7

Foreach

Among the new features included in Java 1.5 is a new *for* loop. This new loop is specifically designed to iterate over collections and arrays. Previously when iterating over a collection the code would look like the code in code example 37.

```
For some given Collection c
Collection coll = .....
for(Iterator i = coll.iterator(); i.hasNext(); ){
    //
}
```

**Code Example 37:** Example of iterating without the new for loop.

Using the new *for* loop, it becomes easier, as shown in code example 38.

```
Collection coll = ..../some collection of strings
for (Object obj : coll){
    String s = (String) obj;
}
```

**Code Example 38:** Example of looping over a collection using the new for loop.

The new *for* loop is read like: “For each Object obj of the collection coll do”. The new “*for*” loop is known as “*foreach*” No new keywords are added to the Java language to support this feature, and no changes to the Java Virtual Machine was therefore necessary.

As code example 39 shows, combining the new for loop with generics, makes it even better.
Collection<String> coll = new LinkedList<String>();
coll.add("Str1");
coll.add("Str2");

for(String s: coll){
    System.out.println(s);
}

**Code Example 39:** Using the new for loop combined with generics.

There is some collection of Strings, which in this case is declared as a LinkedList. For each String, referred to as s, in the Collection referred to as coll, print out the String. In this case, “Str1” and “Str2” will be printed out.

### 7.1 Implementation of Foreach

To implement the “foreach” feature, changes to the Parser, Name Visitor, TypeChecker, and Evaluation Visitor was required. Additionally, a new node had to be added to the abstract syntax tree*.

#### 7.1.1 Parser

To accommodate the syntax for the new for loop, the grammar used to generate the parser needs a little twist. Since Sun didn’t add a new keyword for the foreach loop, it makes sense to handle the regular for loop together with the new. Code example 40 shows the extended grammar for handling both syntaxes of “for”.

This part taken out from our grammar is handling both the regular “for” loop and the new “foreach” loop. After encountering "for (" a look ahead is performed, to find out if a colon is present in the elements within the parenthesis. Remember the "foreach" looks like:

```java
for(FormalParameter:Expression){ Statements }
```

---

*The “Foreach” Node*
try {
    t="for" ( "(" 
    {
        LOOKAHEAD(formalParameterLookahead() ":")
        forEachInit=formalParameter() colon=":" exp=expression()
        |
        [ init=forInit() ";" ]
        [ exp=expression() ] ";" 
        [ update=statementExpressionList() ]
    }
    ")" stmt=statement()
    
    if(colon != null){
        return new ForEachStatement(forEachInit, exp, stmt,
                                   filename, t.beginLine, t.beginColumn,
                                   stmt.getEndElement(), stmt.getEndElement());
    }
    else{
        return new ForStatement(init, exp, update, stmt,
                                 filename, t.beginLine,
                                 t.beginColumn, stmt.getEndElement(),
                                 stmt.getEndElement());
    }
}

**Code Example 40:** The grammar used in the parser to handle the syntax for the new "for-loop".

If there is a colon, the variable "forEachInit" is set to contain the formal parameter, colon is set to the string ":", and exp is set to the expression. If there was no colon "init", "exp" and "update" is set, which is read from the well known regular for loop.

for(init; exp; update)

Following with the statements, kept in the variable stmt, a ForEachStatement or a ForStatement node is returned depending on a colon was encountered or not.

### 7.1.2 NameVisitor

When each "foreach" loop is transferred to a regular "for" loop, the name visitor has the responsibility to ensure that no variable used in the translated "for" loop will overshadow
any other variable name when dealing with nested loops. This is achieved by using a name
counter, adding numbers to variable names to make it less likely that the names chosen,
will be the same as what the user will pick. In section 7.1.4 the syntax for the special name
counter can be seen in the code examples.

7.1.3 TypeChecker

In the type checker there are two things to check. The first is whether the type of the right
side is a collection or an array. The second is whether the left side is assignable. To check if
the left side is assignable, the method "isAssignableFrom(Class <? > cls)" from the Class
"Class" in the Java Standard Library is utilized. This method returns true if the class object
it is called on, is either the same or a superclass of the class represented by the class given
in the parameter. As for primitives, for the method to return true it has to be precisely the
same type [19].

If the right side is a List of ints, then the left side has to be an int. To give another
example, the right side is an Array and without Generics, then the left side is an object.
Note that it is currently not possible to check if the type of the elements of the collection
is of the same type as the left side, because generics information is currently not used,
although it is kept. This will be resolved when a full TypeChecker is implemented. At the
present moment it is assumed that it is typed correct, if not a cast error will be thrown.

7.1.4 EvaluationVisitor

The evaluation visitor translates the "foreach" loops to regular "for" loops. There are three
different cases which needs to be handled, a generified collection (example 41), a raw col-
lection (example 42) and when dealing with an array of primitives or types (example 43).
The three cases are described in the following code. The "#i", indicates the variable name
generated to prevent the shadowing of variable names as previous mentioned.

Once the "foreach" loops has been translated into regular for loops, which the inter-
preter (DynamicJava) knows how to deal with.
Generified Collection

Collection<String> c = ... ;
for(String s: c){
    statements...
}

Translates to:

for(Iterator<E> \#i = Expression.iterator(); \#i.hasNext(); ){
    FormalParameter = \#i.next();
    statements...
}


Raw Collection

Collection c = ... ;
for(Object o: c){
    String s = (String) o;
    statements...
}

Translates to:

for(Iterator \#i = Expression.iterator(); \#i.hasNext(); ){
    FormalParameter = \#i.next();
    statements...
}

Code Example 42: Translation of a raw collection.

Array of primitives or types

int sum(int[] a){
    int sum = 0;
    for(int i:a){
        sum+=i;
        return sum
    }
}

The loop part Translates to:

for(int \#i=0; \#i<a.length; \#i++){
    FormalParameter=a[\#i];
    statement...
}

Code Example 43: Translation of array.
Chapter 8

Autoboxing

For a long time, Java developers have been annoyed with the constant wrapping and un-wrapping primitive values, especially when using them with Collection classes. All collection classes provided in the Java class library can only contain ReferenceTypes (Objects), and not primitives. For each primitive type there is a corresponding wrapper class, which can be used to wrap the primitive so it can be used in these collection classes.

An example of these wrapper classes is the Integer wrapper class. If the developer wants to place an number 10, inside a list, l, he will need to wrap it like this: l.add(new Integer(10));, and when he wants the value back out, he will have to do: ((Integer)l.get(0)).intValue();

All this wrapping and unwrapping is very tedious, and this is where autoboxing/unboxing comes to the rescue. The autoboxing feature automates the process of using the annoying wrapping and unwrapping. Table 8.1 shows the different conversations from primitives to their wrapper classes and visa versa.

<table>
<thead>
<tr>
<th>Integer to int</th>
<th>int to Integer</th>
</tr>
</thead>
<tbody>
<tr>
<td>to Double</td>
<td>to Double</td>
</tr>
<tr>
<td>Long to Long</td>
<td>Long to long</td>
</tr>
<tr>
<td>Float to Float</td>
<td>Float to float</td>
</tr>
<tr>
<td>char to Character</td>
<td>Character to char</td>
</tr>
<tr>
<td>byte to Byte</td>
<td>Byte to byte</td>
</tr>
<tr>
<td>Short to Short</td>
<td>Short to short</td>
</tr>
<tr>
<td>boolean to Boolean</td>
<td>Boolean to boolean</td>
</tr>
</tbody>
</table>

Table 8.1 : The automatic conversions by Autoboxing.
In previous version of Java, 1.4 and earlier, the equality operators == and != allowed for two booleans, two numbers, or two references to be compared, and everything else would result in a compilation error.

With Java 1.5's auto-unboxing, it becomes possible to compare a boolean with a Boolean, and any primitive with its corresponding wrapper class. This means that it is possible to have a primitive on one side of the equality operator and a reference type on the other.

When it comes to having a reference type on each side of the equality operators, they keep their same semantic as they had before. The object references are compared. They will not be unboxed and have their contents compared, see example 44.

```java
Boolean b1 = new Boolean(true);
boolean b2 = true;
Boolean b4 = new Boolean(true);
boolean b6 = true;

b1 == b2 is true
b2 == b6 is true
b1 == b4 is false
```

**Code Example 44: Overview of boolean comparison rules**

Notice that b1 is not equal to b4. Just because two objects can be unboxed to the same value, doesn't mean that they are equal.

For clarity, code example 45 shows a place to be aware of as well.

It is definately worth to note how the two equality checks inside the last output gives different results in example 45. The comparison will return true as long as the two variables are set to the same value, and the value ranges from -128 to 127 (both numbers included). The reason for this seemingly strange behavior is that Sun's Java Virtual Machine caches values from -128 to 127, and a new Integer will only be allocated if it is "necessary". If it finds one that represent the number 10 already in the cache, it will use the same one, instead of creating a new one.
Code example 45: Pitfall with equality check.

Code example 46 provides an example where autoboxing comes in handy.

Code Example 46: Autoboxing making life easy when inserting primitives into collections.

In example 46, autoboxing handles the wrapping in the line `v.add(5)`, so it gets wrapped to: `v.add(new Integer(5))`; automatically. Similarly when the first element is taken out of the Vector, the static typechecker provides the necessary cast to an Integer, and autoboxing then unbox the wrapped class to its primitive value.

Notice that the cost of wrapping primited to objects, and unwrap the objects to primitives still has the same cost as when the developer did it himself. When using the wrapper classes, it is much slower because the objects needs to be allocated on the heap.
8.1 Implementation of Autoboxing

Before the implementation is discussed, it is very important to make clear that the boxing and unboxing is done at compile time. When a box or unbox is needed, the compiler will wrap the primitives or unwrap the wrapper classes. This indicates that the right place to do the implementation is the TypeChecker, which is the part of the interpreter that traverse the abstract syntax tree and defines the variables. Almost all handling of autoboxing can be done in this specific visitor. Although most cases can be handled in the TypeChecker there are a few cases that can not be handled before evaluation. An example of one of these special cases is the PostIncrementExpression (x++).

8.1.1 Changes to the TypeChecker

In order to implement the autoboxing, every time a condition is encountered, a special concern has to be shown. This is for example in the “while loop”, and in the “if-then-else statement”. These places have to be dealt with because everytime a condition is encountered, it can be both a boolean and its wrapper class Boolean. The solution implemented is to convert all encountered wrapper class Booleans to the primitive boolean value. Every case of where these conditions can occur, is handled in the same way. Code example 47 shows how the conversion of the condition in a while loop is handled.

Example 47 shows the method that handles the while statement. The first step it takes, is to get the condition of the while statement. It checks if the condition is different from either the primitive boolean or the wrapper class Boolean, and throws an error if that is the case. It then checks to see if the condition was the wrapper class Boolean, and if so, it sets the condition to be the unboxed type, boolean. It then gets hold of the body of the while loop, and continues traversing the abstract syntax tree. This means that the nodes are being changed in the tree as it goes along, to ensure that all wrapper class Booleans in places of conditions are changed to the primitive boolean.

One other place where boxing or unboxing can take place is during assignments. If a variable of type Integer is declared and is being assigned to a primitive int, the
public Class<?> visit(WhileStatement whileStmt){
    //Get the condition
    Expression exp = whileStmt.getCondition();

    Class<?> type = exp.acceptVisitor(this);
    if(type != boolean.class && type != Boolean.class)
        throw new ExecutionError('condition.type', whileStmt);
    if(type == Boolean.class)
        whileStmt.setCondition(_unbox(exp, type));

    whileStmt.getBody().acceptVisitor(this);
    return null;
}

**Code Example 47:** This example shows how the condition conversion is done for the while loop

primitive int has to be boxed. Another example is that the variable of type Integer is being assigned to something a method invocation returns. The value being returned might have to be boxed. Example 48 shows this scenario.

```java
int abs(int num){
    if(num<0) return -1*num;
    return num;
}
Integer x = abs(-25);
```

**Code Example 48:** Example showing where boxing is needed.

In example 48 a method is declared to calculate the absolute value of a primitive integer. A variable is then declared and assigned the type of the wrapper class Integer. When the method abs is called, a primitive int is returned, and the return value needs to be boxed to the wrapper class type, Integer. The opposite scenario is shown in example 49.
Vector<Integer> v = new Vector<Integer>();
v.add(new Integer(3));
int x = v.get(0);

**Code Example 49:** Example showing where unboxing is needed.

In example 49 a collection Vector of Integer is declared, which then has a Integer
wrapping the value 3 added. When the variable x is declared as primitive type int, and
the value of the collection class is assigned to it. The return value, which is a Integer,
needs to be unboxed to an int.

Both cases are handled by doing a check of the type of the variable on the left hand
side. If necessary the value assigned to it, will be boxed or unboxed to match the type of
which is it being assigned to.

### 8.1.2 Changes to the EvaluationVisitor

The special cases where autoboxing has to wait until time of evaluation are when the ex-
pressions listed in table 8.2 are encountered. Previously, in Java 1.4, it was not allowed to
use these operations on wrapper classes such as Integer.

<table>
<thead>
<tr>
<th></th>
<th>AddAssignExpression</th>
<th>SubtractAssignExpression</th>
<th>MultiplyAssignExpression</th>
</tr>
</thead>
<tbody>
<tr>
<td>DivideAssignExpression</td>
<td>RemainderAssignExpression</td>
<td>PostIncrementExpression</td>
<td></td>
</tr>
<tr>
<td>PreIncrementExpression</td>
<td>PostDecrementExpression</td>
<td>PreDecrementExpression</td>
<td></td>
</tr>
<tr>
<td>BitAndAssignExpression</td>
<td>ExclusiveOrAssignExpression</td>
<td>BitOrAssignExpression</td>
<td></td>
</tr>
<tr>
<td>ShiftLeftExpression</td>
<td>ShiftRightExpression</td>
<td>UnsignedShiftRightExpression</td>
<td></td>
</tr>
</tbody>
</table>

**Table 8.2:** The special cases for handling Autoboxing.

Each of these special cases are handled precisely the same way. They are handled in the
evaluation visitor. When the visitor handles each of the cases, for example the PostInc-
crementExpression and ExclusiveOrAssignExpression, a check is added to ensure that boxing
is performed if the action is performed on a wrapper class. See code example 50.
```java
Integer x = new Integer(5);
x++;
x ^= 3;
```

**Code Example 50:** Examples where unbox or box can not be handled in the TypeChecker

In example 50 note that a variable `x` is declared as an `Integer`, wrapping the `int` value 5. When encountering the post incrementation of `x`, the evaluation visitor will unbox the value of `x`, do the post incrementation and box `x` to its wrapper class again. And the same follows in the exclusive or operation that follows in the example.

The TypeChecker can not handle the autoboxing because it is only unboxed to perform the evaluation and then boxed back up. That is why these special cases has to be handled in the evaluation visitor.

With these changes DynamicJava is now able to handle autoboxing.
Chapter 9

Enum

Another new feature with Java 1.5 is the enums. An enum is a special kind of class, and a declaration of a such class, will yield an enum type. An example of enum declaration can be seen in code example 51.

```java
public enum Suit(HEARTS, SPADES; int add(int x, int y)
    return x+y;
}
```

**Code Example 51:** An example of an enum declaration

The keyword enum is used instead of the class keyword which it used for declaring a regular class. Suit is the name of the enum-class defined. HEARTS and SPADES are enum-constants, which correspond to static final variables in a regular class, all of the same type as the class, here Suit. After the “;” separator any body declarations can be placed. In this example, a method declaration has been named add.

Each enum-class has an array called $Values$ which contain all the constants declared in the enum. Along with this variable, they have a corresponding get method, called values(), which returns a copy of the array of values. Last, enums also have a method called valueOf(String s), which returns an enum-constant if one exists matching the name given.

The very short code example 51 is equivalent to the code in example 52, where the new enum class is not used.

It is clear when comparing the code in example 51 and example 52 that the enums carry some power. It is easily noticable that they can represent something, in a very few lines, that a regular class would take many more lines to do.
public class Suit extends java.lang.Enum{
    public static final Suit HEARTS = new Suit(’HEARTS’, 0);
    public static final Suit SPADES = new Suit(’SPADES’, 1);

    private Suit(String name, int ordinal){
        super(name, ordinal);
    }

    private static final Suit[] $VALUES = new Suit[]{
        Suit.HEARTS, Suit.SPADES
    };

    public static final Suit[] values(){
        return (Suit[])Suit.$VALUES.clone();
    }

    public static final Suit valueOf(String s){
        for(int i = 0; i<$VALUES.length; i++){
            if($VALUES(i).name.equals(s)){
                return $VALUES(i);
            }
        }
        throw new IllegalArgumentException(s);
    }

    int add(int x, int y){
        return x+y;
    }
}

**Code Example 52:** Code equivalent to code example 51

It is essential to understand that an Enum class can not extend any classes. This is because the Enum class is already extending java.lang.Enum. In addition, an enum class is serializable, comparable and final, but no enum class is cloneable. The Enum class is “automatically” equipped with the appropriate toString, hashCode and equals methods, used for comparison operations.

Example 52 shows how an enum can be thought of in terms of a regular class, although in Java it is not allowed to extend the java.lang.Enum class directly.

It is important to note is that an enum class can **not** be instantiated, see example 53.
```java
enum Suit(HEARTS, SPADES)

public class A{
    Suit suit = new Suit();
}
```

**Code Example 53:** It is not legal to instantiate an Enum. This example is not allowed.

A more in depth explanation of Enums, is given by following example 54.

```java
enum Suit(HEARTS, SPADES;Suit pick(){return HEARTS;})

public class A{

    Suit[] suit = Suit.values();

    void doSomething(){
        for(Suit s : suit){
            System.out.println(s.pick());
        }
    }

    public static void main(String[] args){
        A a = new A();
        a.doSomething();
    }
}
```

**Code Example 54:** A short example of an enum.

First an enum class called Suit is declared, containing two enum constants named HEARTS and SPADES, both of the type Suit. A method is then declared called pick(), which returns an object of type Suit. In this case, it returns our constant HEARTS.

A class A is declared, where the variable suit now represent the array of all the enum constants from the enum class called Suit. In the doSomething() method, which is called, there is a "for-each" loop, which prints out each of the enum constants contained in the Suit enum class.

The enum constants can be used in a switch statement just like regular constants. This is shown in example 55.
class ColourUnknownException extends RuntimeException{}

enum Colour(RED, BLACK)

enum Suit{
    HEARTS, SPADES, CLUBS, DIAMONDS;

    Colour findColour(Suit s){
        switch(s){
            case HEARTS: return Colour.RED;
            case SPADES: return Colour.BLACK;
            case CLUBS: return Colour.BLACK;
            case DIAMONDS: return Colour.RED;
        }
        throw new ColourNotFoundException();
    }
}

**Code Example 55:** Using enum constants in a switch statement.

The java.util library of Java has been enlarged by two additional classes specific for support of enums. There two classes are EnumSet and EnumMap. Both are added to provide with an improvement in performance [18].

All members of an EnumSet must be of the same enum type, and the EnumSet offers a special static factory. The method "range" can be used to iterate over a set of enums, as seen in code example 56.

```java
enum Months{ JAN, FEB, MAR, APR, MAY, JUN, JUL, AUG, SEP, OCT, NOV, DEC
}

It is now possible to iterate over it like:

```for(Months m: EnumSet.range(Months.MAY, Months.NOV)){
    System.out.println(m);
}
```

**Code Example 56:** Example of using enums in the new for loop.

Dealing with EnumMaps, is similar to dealing with regular Maps, just altered to deal with enum keys, which are implemented as an array [18]. This makes it desirable to utilize an EnumMap instead of an array whenever an enum type needs to be mapped to a value.
9.1 Implementation of Enum

All the code for implementing Enums happens in the Parser, except the support for utilizing Enums in the switch statement. A new node was created, called EnumDeclaration. This node extends the original ClassDeclaration and not our GenericClassDeclaration, which is because it is not possible to utilize type parameters with enums.

The main purpose of the EnumDeclaration node, is to transform an Enum class into a regular class, after which the evaluation, through the evaluation visitor, can be performed just as with regular classes.

This "transformation" consists of three steps. Refer to the previous example 52 for clarity.

- Make Enum constants into public static field constants in a regular class.
- Handle the constructors
- Add Values.

The first step takes all the declared Enum constants, and represents the same information by making field constants in a regular class. Each constant is of the same type as the Enum was declared as. In the prior example 52, the type of the constants and the Enum class was Suit. The "0" passed as a parameter to the instantiation of the "new Suit", is called the ordinal and is used to indicate the position of the constant in its enum declaration, where the initial constant is assigned an ordinal of zero. All constants are made "public static final."

In the second step, a list containing the nodes from the Enum body, that being the enum constants(which is already handled) and the enum body declarations(anything following the ":" separator), is searched through to see if any of the enum body declarations is a constructor declaration. If no constructors are found, the only constructor creates is the private constructor, taking only the name and the ordinal as parameters, which pass them off to the constructor in the super class.
private Suit(String name, int ordinal) {
    super(name, ordinal);
}

If additional constructors are found, they will be added, and again with the additional parameters String name and int ordinal added in front of the list of parameters, illustrated in example 57.

```java
public enum Suit {HEARTS(2,3), Spaces(4,5); Suit(int x, int y){}}

Would have the constructors:

private Suit(String name, int ordinal) {
    super(name, ordinal);
}
public Suit(String name, int ordinal, int x, int y) {
    super(name, ordinal);
}
```

Code Example 57: Example of enums with several constructors

In the third and last step the private static field $VALUES$ is added. This field is an array of SUIT, where the constants are stored. The values() method and the valueOf(String s) method are added as well. They are shown in code example 52.

9.1.1 Parser

Unlike the new for loop construct, enum requires a new keyword, the keyword “enum” was therefore added to the parser.

In the parser, a look-a-head for the enumDeclaration was added, where it used to only contain look a heads for the “class” and “interface” keyword, see the small addition in code example 58. A similar look-a-head was added to the ClassBodyDeclaration and InterfaceBodyDeclaration, since inner enums can exist.
TypeDeclaration typeDeclaration() :
{
    TypeDeclaration node;
    int modifiers;
}
try {
    (LOOKAHEAD(classDeclarationLookahead())
        node=classDeclaration()
    | LOOKAHEAD(enumDeclarationLookahead())
        node=enumDeclaration()
    | LOOKAHEAD(interfaceDeclarationLookahead())
        node=interfaceDeclaration()
    }
}

**Code Example 58:** Small addition to the TypeDeclaration so enum declarations are recognized.

**For convenience, remember the enum looks like:**

[modifiers] enum 'identifier'{[enum body]}

Where:

[enum body]: [enum constant] [arguments]

    [enum body declaration]

Parsing of the enums, are devided into four parts: enumDeclaration, unmodifiedEnumDeclaration, enumConstants and enumBody.

EnumDeclaration handles the modifiers and then calls the method "unmodifiedEnumDeclaration", passing in already checked modifiers. In example 59, a small part of the grammar is shown which handles the enum declarations.

After having dealt with the modifiers, mf, the next token to encounter while parsing is the "enum" keyword. Here, it is stored in the variable "i1". Following the "enum" keyword comes the identifier, which is the name of the enum class declared, kept in the variable "id". An optional list of interfaces can be implemented. Here, it is kept in a list, called "impl". The curly brackets indicate the begining of the enumBody, which is parsed by a different method called enumBody(), further explanation of this method will be described in the
```java
EnumDeclaration unmodifiedEnumDeclaration(ModifierFlags mf) {
    t=<ENUM>
    id=<IDENTIFIER>
    "implements" impl=ReferenceTypeNameList()
    "{"
    body = enumBody()
    e="}"
    {
        if (mf != null) m = mf.accessFlags;
        return new EnumDeclaration(m, id.image, impl, body, ....);
    }
}
```

**Code Example 59:** The part of the grammar that handles the EnumDeclaration.

The following section. Ending the enumbody is the curly bracket, and a new EnumDeclaration node is returned with all the parsed parts as parameters.

The enumBody method is divided into the enum constants and the enum body declarations. The interesting part is in parsing an enum constant, which is being done for each constant encountered. They are all put in a list, as seen in example 60.

```java
EnumDeclaration.EnumConstant enumConstant() {
    id = <IDENTIFIER>
    [ args = arguments(null) ]
    [_classBody = classBody() ]
    {
        return new EnumDeclaration.EnumConstant(id.image,
            (args==null) ? null : args.arguments,
            (_classBody == null)? null : _classBody.list);
    }
}
```

**Code Example 60:** A subpart in the grammar to handle the enum body for the enum declaration.

The EnumConstant is merely an identifier with the possibility of the constant having some arguments (a parenthesis followed by the parameters). Then a class body, as it is possible to pass in methods to each of the enum constants, also known as "constant-specific
methods". Last a new object of the type EnumConstant is returned. It is then added to a list of EnumConstants as there can be several in an enum declaration.

When all the EnumConstants have been parsed it is time to parse the EnumBodyDeclarations. The interesting part of parsing those can be seen in example 61.

```java
List<Node> enumBodyDeclarations() :
{
    ";",
    (decl = classBodyDeclaration()) {
        body.addAll(decl);
    }*
    {
        return body;
    }
}
```

**Code Example 61**: Segment from the grammar to handle enum body declarations

What syntactically separates the enum constants and the enum body declarations is the semicolon, ";". Then everything encountered can be handled just like a bodyDeclaration used for classes. All these classBodyDeclarations are added to a list, and returned. Now the EnumDeclaration node can be created and returned because the parsing is now complete.
Chapter 10

Metadata

Another new feature of Java 1.5, is metadata, also known as annotations. The metadata provides an ability to add data to Java classes, interfaces, fields and methods. It allow developers to mark their code with a special tag beginning with @, for example: @test. The information are stored in the class files and are available during runtime. These annotations are created to offer support to tools, such as JUnit.

For example, before Java 1.5, JUnit was required to have its test cases start with the word test, like in testForEach. With metadata, tools no longer have to force special naming of methods upon developers. Code example 62 and 63 shows a before and after picture to illustrate the difference having this new feature and before.

```java
import junit.framework.TestCase;
public class EvaluationVisitorTest extends TestCase{
    protected void setUp(){
        ...
    }
    protected void tearDown(){
        ...
    }
    public void testPreDecrement(){
        ...
    }
}

Code Example 62: JUnit without metadata
```

Notice how in example 63 the developer can name his setUp, and tearDown method whatever he wants, and how the test method doesn’t need to begin with the word test anymore.
import junit.framework.TestCase;
public class EvaluationVisitorTest extends TestCase{
    @Before protected void initStuff(){
        ...
    }
    @After protected void closeDownStuff(){
        ...
    }
    @Test public void preDecrementTester(){
        ...
    }
}

**Code Example 63: JUnit with metadata**

Although typical developers will never find use for defining their own annotations, it is possible. Example 64 shows how it is possible to declare one of these annotations.

```java
public @interface RequestInformation{
    int id();
    String name();
    String location();
}

A method declaration using the annotation type.

@RequestInformation{
    id = 2910928,
    project = "DrJava",
    location = "Rice University"
} public static void registerFor312(String name){...}

**Code Example 64: JUnit with metadata**

### 10.1 Implementing metadata in DynamicJava

Metadata doesn’t provide with any real value in the interactions pane of DrJava, and it was therefore decided that it would not be implemented. The use of metadata is meant for classes that keep for a more permanent time, rather than those created in DynamicJava on-the-fly and discarded.
Chapter 11

Tools

This chapter describes the various tools that were using throughout the extension of DynamicJava.

11.1 DrJava

As any self-respecting IDE developer would do, DrJava developers use DrJava to develop DrJava. It is a great environment, with all the tools needed such as JUnit, a project facility, and debugger built-in. In particular, the Java interpreter provided in the interactions pane makes the environment superb. At the time that the new Java 1.5 features were implemented there were also no other programming environments available that would support the Java 1.5 code. DrJava has been able to do so, since the first versions of the JSR-14 compilers came out. DrJava was ready for them, as it has been supporting Generics through the GJ compiler that later turned into the JSR-14 compiler.

Using DrJava to develop DrJava also makes the developers “eat their own medicine” so to speak. If they make mistakes they will almost certainly notice, during their continued development. It has also proved to be a good way to come up with ideas for new features and improvements.

11.2 JUnit

From the author of many of the Extreme Programming books, Kent Beck, came this Unit test tool for Java. Kent Beck developed this tool together with Erich Gamma [3]. This Unit testing tool provides a way to code test cases for the entire code base, very easily, and setup
a test run with DrJava (or Ant). When test cases are provided, and the code is well covered with tests, any changes or new additions to the code that contains a bug, is meant to be caught by this tool. The biggest “flaw” is that the tool is only as good as those who writes the test cases. Only if the tests are written correctly will it be of any use.

When DynamicJava was adopted by JavaPLT at Rice University, it had no test cases at all. Most of the time in the beginning was spent covering as much code as possible with tests. A testing framework was set up so that before any code changes or additions could be added to the main source tree, all tests had to run and pass first. Avoiding having broken code when checking out from the source tree.

11.3 Ant

In 1998 James Duncan Davidson designed Ant [21], a Java based build tool. By using XML files, the developer can specify targets to build, and execute a variety of tasks for that target. Some of the tasks can be: compiling files, copying files, creating documentation, run tests, checkout files from CVS. Since Ant is developed in Java it works cross platform, and is therefore a highly attractive tool.

Throughout this project, Ant was used for checking out our code, compiling the code, committing code, and running our tests. Ant assumes that the build folder of the compiled java classes will be a copy of the structure that the source files have, and the automatic management of compiling all files to a separate folder is very convenient instead of placing the class files next to all the Java files. DrJava has this functionality too of course.

11.4 JavaCC

JavaCC stands for Java Compiler Compiler [20], and is a free commercial tool created at Sun. JavaCC reads from a BNF* grammar file and generates a recursive decent parser. In the implementation chapter, several examples of the grammar was shown. Most of the

---

*BNF stands for Backus-Naur Form
new Java 1.5 features implemented in DynamicJava, needed a tweak to the old available grammar, and it was this grammar that got used to create the parser that DynamicJava uses to parse the syntax developers write in the interactions pane. It is hard to find usable tools when working with the cutting edge technology and languages, such as Java 1.5. JavaCC did not allow usage of Generics in the code, and as such it was necessary to put the Generic tags in surrounded by comments. Then after the parser was generated, have a script take out all the comments.

11.5 Clover

In section 11.2 it was mentioned that in the beginning of this project, a lot of time was invested in developing an improved test coverage of the code base.† Clover is a code coverage analysis tool created by Cenqua [5] that helps determine where tests lack the most, and runs as an Ant task. Clover instruments the source code and then records in temporary files exactly what is executed when the tests are run, gathering all the data up in a data file. When the tests have completed a report, is generated. The report provides very detailed information as to which areas of the code were well covered, and which ones lacked coverage. The report is detailed down to the level of lines. It reports how many times each line in a class has been executed. Overall, Clover worked as a great tool, although with a few limitations. There is an <exclude> tag that can be used in Ant running the Clover task, which allows special file names to be excluded from the instrumenting process. Unfortunately, it is not possible to exclude static inner classes. Clover can also not measure coverage of a conditional expression if it contains an assignment operator. While running Clover, an “out of memory” error caused some delay. One of the tests forks several Java Virtual Machines, and during Clover tasks would run out of memory. Cenqua suggested several solutions, and in the end one was found that worked.‡

---

† Any tests would be an improvement to no tests.

‡ The solution to the failing test, GlobalModelJUnitTest, was to set maxmemory="1024m" in the “test-only” target, inside the <junit> tag in DrJava’s build-common.xml file.
11.6 CVS

Version control systems are almost assumed to be used in even smaller projects. It is a very useful tool, allowing shared code base without any big concerns regarding people messing up the main code base. CVS makes it possible to go back in history and recover files if the accident should happen [7]. The flow of the project was never slowed by questions regarding where the latest version of the source code is. It was easily accessible for everyone to get hold of and work with the latest source at all times.
Chapter 12

Methodology

Every developer is using some form of programming methodology. Even if it is not documented and seems ad hoc. The developer is in such case merely using an "ad hoc" methodology. A variety of methodologies exist to guide a developer through the work process. Each one claims to be better than the previous one. Some swear by a long analysis process, understanding a system, its context, and the conditions for its implementation, and determine the systems requirements. Only to follow with a just as long design process, producing a system design without any significant uncertainties. The implementation is to realize a design on a technical platform. And then finally testing that everything works. One of many places this proposed way of going about developing is done at [13].

At Rice University, and many other places, it has become natural to introduce tests as soon in the process as possible, as well as getting one small task done before going to the next.

The authors of the book "Extreme Programming Installed" - R. Jeffries - A. Anderson - and C. Hendrickson - claim that the Extreme Programming methodology is more useful, efficient, and faster than regular analysis and design processes [12]. The Java Programming Languages Team at Rice University, also favours this methodology, and it is the one followed in this project.

The main points from Extreme Programming is described in the following sections and how they were used in this project.
12.1 On-site customer

The customer is the key element to Extreme Programming. He is the one that writes user stories, and the one that steers the project with setting release dates. By having the customer on-site, it is possible to get quick feedback, which will limit the time wasted on misunderstandings. It makes it possible to go and ask him questions or run suggestions by him. When it is possible to talk to the customer so frequently, it also makes it possible for him to pick up small slips that found its way into the design documents. Communication with the customer and workers are of utmost importance, and it is therefore crucial that their workstations are located close to each other.

During this project, the customer has partly been Professor Robert Cartwright, regarding release dates, but not like a real customer since Professor Cartwright was actively working on some of the code base as well. Students at Rice University, especially in the Java class, as well as the Professor teaching them, have also acted in some way as customer, giving us feedbacks and coming up with feature requests.

12.2 User stories

Perhaps the idea came from Hollywood, as user stories are closely related to Story boards they use at the movies. Stories are written by the customer, which describes the different scenarios for usage of the product. The stories will at each meeting be altered, discarded or added to the next iteration. This will continue until the product reaches a satisfactory state to the customer. The stories are normally written down on small cards, and can quickly be reviewed to gain the big picture.

In this project, the user stories came from Professor Cartwright and the developers working on the project. They were not written down on small cards as the methodology suggests, but they were discussed verbally, and notes were taken. Since the project was steered by the Java 1.5 specification to some extent, the user stories were used as much as perhaps in other projects.
12.3 Iterations

An iteration in this process starts with a meeting, where the long term goals are discussed, and short term goals are identified. A short term goal is found by dividing bigger goals into smaller and smaller parts until the part to be done is small enough for someone to handle between now and the next meeting. Everyone is present at these meetings, and the user stories are reviewed or created. It is at these meetings that design questions are discussed, and to some extend design documentation or assignments for design documentation is made. After everyone has chosen a task, the developers go back to their workstations and work on achieving the short term goal before the next iteration. Having iterations such as these, makes it hard to steer the project far off the right track, and it gives everyone an opportunity to raise questions, and air ideas.

In this project the iterations had a duration of one week. There was a meeting once a week, where Professor Cartwright, students from the course 312 (Production Programming), and TAs from the 312 course, attended. The meetings were held for DrJava in general, but this project was one of the main topics. It was here that acceptance tests were set through combined efforts. The acceptance tests covered corner cases, and what the behavior should be on a large scale. Small scale tests was covered by unit tests. Although goals were broken down to smaller pieces, it happened that some of the tasks would take some weeks, although the progress of them were always discussed to ensure it moved forward.

12.4 Design sessions

During the weekly meeting design questions can be brought up and discussed. If a problem or question arises regarding the design of a certain aspect of the project, and it is not just before the weekly meeting, a design session should be called. Extreme Programming tries to emphasize the importance of getting answers to questions such as these out of the way as fast as possible, removing any roadblock that stands in the way of progress for the project. Again here it is very important to have the on-site customer as some of the questions can
only be answered by him.

Throughout this project design meetings were called now and again, to make sure that the design of the specification of the Java 1.5 features was understood correctly, or trying to figure out how to solve a specific corner case. The weekly meetings was more to get the overall picture, and detailed design discussions were then taken up in a smaller group afterwards.

12.5 Acceptance tests

The customer is the main person who comes up with acceptance tests. These tests cover what he wants the system to do in specific cases. More precisely what he wants to work in the system. For example he could say: When I hand the program the measurements of a wooden cube, it will give me results as to how I should cut a such wooden cube to get 4 pyramids of biggest possible size. A problem such as this is a high level problem. The customer doesn't care about how it is implemented, as long as he gets what he wants. When the program can provide him with his request, he will accept the program as working. The developer can use the acceptance test to write up a high level unit test, making the test automatic.

During this project many of the acceptance tests came from the Java 1.5 specifications. It was stated what a specific syntax should yield of result. Due to the differences between DynamicJava and regular Java, some acceptance tests were added, or some were even altered from the original specifications. For example declaring a List<String>, and then assigning an Integer to that list, is in DynamicJava accepted. Currently DynamicJava doesn't have a generic typechecker, so for this part of the project this part was accepted. The highest level acceptance test, was of course that when writing code in the interactions pane using the syntax for the new features in Java 1.5, they should be accepted, and the interpreter should handle the code appropriately.
12.6 Unit tests

Unlike acceptance tests, unit tests go much more into details, but the acceptance tests do get their own unit test too. Unit tests just also test in much more depth. Unit tests are automatic tests which has the purpose to help ensure new changes to the code doesn’t break any of the old code. When a developer is about to write a new method, for example to calculate the volume of a pyramid, he will first write a test case that checks this method. At first the test will fail, since he haven’t written the code yet, but after he has added the code for the method, the test should pass. Kent Beck in Extreme Programming states that “Writing tests before coding results in a Better Design” [12]. It is important to note that there is no assurance that having tests that pass indicate that everything works. The tests can only be as good as the one writing them. If the developer doesn’t know how to calculate the volume of a pyramid, he can use the wrong formula in both the method and in the test, and the test will pass. But it will still give a wrong result. The goal is to add tests cases for everything that can possibly break. Everytime a bug is found, a test case has to be added for that.

At Rice University and throughout this project, when addressing a new area, or introducing students to a code base, the first assignment is to add some test cases for the code. This improves the familiarity with the code, and improves the test coverage. A code coverage tool was used in this project* to help provide ideas of where the demand for tests were greatest. When the original code for DynamicJava was looked at for the first time, it was quite scarying to see that there were not even a single test case. Test cases was added as the project progressed, with a result of raising the test coverage to about 50%.

12.7 Small releases

One of the ways to make good use of the on-site customer is to often produce a release for the customer to evaluate. This will provide the developers with feedback, and it will

*Clover from Cenqua, descriped in the tools chapter.
let the customer know how work is progressing. In some methodologies, a prototype is provided to the customer far far into the project, and that can have two dangerous effects. The customers can think that since you have a graphical user interface to show them that seems to have all the right buttons and names, you are almost done. Or perhaps more often, the customers come back with a lot of changes, since it was not precisely what they meant when you talked to them. The dates for the releases are worked out between the developers and the customer.

During this project, it was very fortunate to have a Java class going on, where the students used each small release when they came out. But nothing could replace the value in developing with our own software that was being developed. If something had been done wrong, we would be the ones to first get a chance to experience it. Many of the bugs found were found during our excessive usage.

12.8 Pair programming

Pair programming has been one of the buzz words for a long time. As the name indicates the programming is done in pairs. Two people sit together at one computer, when they are developing code. One person writes the code, he “drives”, while the other observes, watch for mistakes and comment on the coding style and is there for discussions and help. The second person is called the “navigator”. After a period of time the two people switch, and not the first person gets to be the navigator. One of the main wins with pair programming is that it won’t just be one person who knows this specific part of the code. It is also a good way to introduce new people to the code base, teaming them up with a more experienced person. Code revision can also be done as you go along instead of having special time consuming sessions.

Pair programming was used almost at all times during this project. It was very helpful, as typos were caught a lot faster, and it felt easier to take on refactoring tasks, when some “ugly” code was encountered. It also made it very convinient to have someone to discuss decisions with, and debate the design decisions. Some of the design questions were often
resolved at this point, instead of taking it up as a design session.

12.9 Shared ownership of code

Everyone having ownership of the code base makes it easy to provide quick fixes to broken code, or add small improvements. The developers on a project gets to have more experience with the code in general instead of being focused on just one area. Having a shared ownership does require some discipline, especially regarding submitting the new code changes to the main source tree. If the new code changes are not submitted, it leaves to collisions with what other developers have done. That is why developers are encouraged to do quick submits. The best part of it, is that everyone get to work on the newest code.

CVS, Concurrent Versions System, was the tool used to allow everyone in this project to access the main code base. It worked really well, there were few times where one of us had forgotten to checkin the new changes and a collision occurred. It never was any big deal to handle though. Together with CVS, the tool Ant was used to set up tasks that could be performed. For example when a “cvs commit” was going to be executed, to submit the new code changes, a task in Ant was setup to handle this, instead of calling cvs directly, we would call “ant commit”. With Ant it was then possible to set up tasks to ensure that all unit tests were run before the code was submitted, and therefore avoid having broken code in the main source tree.
Chapter 13

Related Work

DrJava was created as part of an effort to develop a new, more accessible introductory computing curriculum focusing on enduring principles of program redesign. The principal forerunner of DrJava was DrScheme [6], developed by a different branch of the PLT research group at Rice University in the late 1990’s. The introductory computing curriculum at Rice University begins with functional programming in Scheme and subsequently migrates to object-oriented programming Scheme. The same principles of program design and development taught in the context of functional programming in Scheme are transferred and adapted to the context of object-oriented programming in Java.

DrScheme was the first sophisticated programming environment developed primarily for pedagogic purposes. It supports a programming interface very similar to that of DrJava except for the fact Scheme programs are generally constructed from a few large files (one for each “module”) rather than a myriad of smaller files like Java. As a result, DrScheme does not have a Navigation pane. A DrScheme session has a single file as its focus; external modules can be browsed but they are not part of the program being edited.

Based on its familiarity with DrScheme, the JavaPLT research group started work on DrJava in 1999 but the current implementation of DrJava based on DynamicJava did not emerge until the fall of 2001.

This doesn’t mean that it would be any replacement for the tests that are saved and kept with the projects. But these long term test suites can be easier to create having the interactions pane to write up small tests, before writing them up in the test file. Earlier prototype implementations relied on constructing a new class for each interaction, which was executed by compiling it, loading it into the Interactions Pane, and creating a new
instance of the class via reflection – executing the code required to perform the interaction.

The prototype implementations were useful but clumsy because evaluating even the most trivial interactions involved a delay of several seconds. The current implementation transformed DrJava from a promising concept to a production tool, which is now used at many colleges, universities, and secondary schools worldwide.

13.1 Early Interactive Language Implementations

The concept of interactive programming languages dates back to at least the early 1960's when the firstly implementations of explicitly interactive languages like Lisp and APL were developed [16]. These early implementations relied exclusively on interpretation. Sophisticated Lisp implementations subsequently added support for compilation but Lisp was plagued for decades by major semantic inconsistencies between compiled and interpreted code. Code that worked when interpreted often broke when compiled because the compiler allocated local variables on the stack, breaking Lisp's dynamic scoping of variables by making them invisible to other functions (either interpreted or compiled). The Scheme dialect of Lisp developed in the late 1970's finally addressed this issue, but semantic distinctions between compiled and interpreted Lisp persisted until the Common Lisp standard appeared in the mid 1980's.

The key insight in Scheme is that the semantics of Lisp became much simpler (and easier to optimize) if lexical scoping replaced the traditional dynamic scoping used in Lisp. Lisp compilers implicitly recognized the importance of lexical scoping for optimization purposes by converting dynamic scoping to lexical scoping (breaking the semantics of many – if not most – Lisp programs). Scheme also demonstrated the essential role that lexical scoping plays in functional abstraction (passing functions as data values).

Many interpreters have been developed for batch-oriented languages like Pascal and C, but they generally were incompatible with optimized production environments for executing these languages because the interpreters for these languages used incompatible execution run-times and data representations designed to facilitate debugging. Interpreters
generally use safe data representations so that the misuse of data is immediately detected at run-time while traditional batch-oriented languages (dating back to Fortran and COBOL) use unsafe representations in the interest of efficiency. Since interpreters typically add a factor of 5-10 in execution overhead versus compiled code, the extra cost of supporting safe data representations is negligible. Moreover, the utility of an interpreter using unsafe data representations is questionable since it cannot provide better run-time error diagnostics than a compiler.

The whole idea of incremental program development is generally a misnomer in the context of unsafe language implementations because the first occurrence of a run-time error generally breaks the current executable image and the only safe way to continue execution after correcting the source of the problem is to completely restart the entire program.

13.2 Java Interpreters

DynamicJava and its competitors, namely the BeanShell [14] and Groovy [24], broke new ground by supporting the interpretation of large subsets of Java without compromising the efficient execution of compiled code. Because the Java Virtual Machine

- supports the dynamic class loading and the reflective invocation of binary code; and

- relies on “just-in-time” (JIT) compilation to generate efficient native code from platform independent binaries (Java class files);

a Java interpreter can simply invoke compiled code using reflection to support the mixed execution of interpreted and compiled code without affecting the efficiency of compiled code. All three of these interpreters rely on this mechanism to support the efficient execution of compiled code.

13.3 BeanShell

BeanShell is a Java interpreter for Java expressions (excluding anonymous inner classes) and statements but not class definitions that has been extended to support several scripting
commands [14]. As example it holds several of the Unix-like shell commands, such as: `dir`, `cd`, `cat`, `pwd`, `rm`, and so on. We determined that the BeanShell was not suitable for use in DrJava because it provided no support for interpreting class definitions, most notably the anonymous inner classes that can appear within Java expressions.

Instead of class definitions, BeanShell supports the concept of nested methods which can be used to simulate simple class definitions by nesting the instance methods for a class inside a method corresponding to the constructor for the class. Unfortunately, this construct is a complete departure from both the syntax and semantics of Java.

The code example 65 shows a simple use of nested method to roughly simulate the Java class in code example 66.

```java
Student() {
    String firstName, lastName;

    teachingAssisant = null;

    _init() {
        super.firstName = "James";
    }

    return this;
}
```

**Code Example 65:** Example of Benshell “class”.

```java
public class Student() {
    private String firstName, lastName;

    private Student teachingAssistant;

    public Student(){
        firstName = "James";
    }
}
```

**Code Example 66:** The Java class corresponding to the BeanShell “class” in example 65

There are important semantics differences that should be noted. The use of `super` doesn’t mean the same thing in a BeanShell nested method and in Java. In BeanShell it
refers to the method that contains the current method. Hence, `super` inside the `_init` method refers to the `Student`—instance name space. The nested method simulation of a class obviously can only define a single constructor. In addition, since no class named `Student` is actually generated, compiled code cannot mention the simulated class. It is not clear how values of such a “class” appear to compiled code.

Note that the `teachingAssistant` variable in the BeanShell “class” is not typed, which means that it can be assigned to any type at a later time.

BellShell also uses `print` instead of `System.out.println`.

### 13.4 Groovy

Groovy is a dynamic language loosely based on Java that includes overwhelming amount of features adapted from languages such as Python, Ruby and Smalltalk [24]. A stated goal of the developers of Groovy was to adhere as closely as possible to standard Java syntax, but there are major differences.

Groovy has added a very extensive list of additional methods to the Java JDK [26]. It also has special support for regular expression, see code example 67*

```groovy
def checkSpelling(spellingAttempt, spellingRegularExpression){
  if(spellingAttempt == spellingRegularExpression){
    println("You spelled correctly.")
  }
  else{
    println("Sorry, try again.")
  }
}

theRegularExpression = /DrJava/
checkSpelling("DraJva", theRegularExpression)
checkSpelling("DrJava", theRegularExpression)
```

**Code Example 67:** Regular expression example with Groovy

The operator: `==` compares a string with a regular expression. Also note how the

---

*Adapted from Groovy’s homepage*
def keyword is used when defining a method, and how `System.out` has been left off of `System.out.println`, which is the Java syntax. Other differences to note is that Groovy uses `==` as the equality operator for both primitives and objects and doesn’t offer support for inner classes [25].

13.5 Why BeanShell and Groovy were rejected in favor of DynamicJava

BeanShell and Groovy are both designed to be scripting languages for programmers familiar with Java. Since their designers were interested in creating scripting languages with a Java-like feel, they are not faithful to the syntax and semantics of Java. Even some very simple Java code fragments (e.g., those attempting to print a string using `System.out.println`) break in these languages.

In contrast, DynamicJava is astonishingly faithful to Java even though it too was ostensibly designed as a scripting language. Perhaps, it has been less successful as a scripting language than either BeanShell or Groovy precisely because it adheres so faithfully to Java.
Chapter 14

Conclusion and future work

This dissertation has described the new features in Java 5.0 and how the JavaPLT research group extended DynamicJava to support them without only modest changes to the original DynamicJava source code. Most of the new functionality comes from adding new code – the proper way to extend programs an the object-oriented language. The extended DynamicJava now supports the interpretation of code fragments using generic types, including the use of wild card types. It also accommodates the new language constructs of Java 5.0 in the interactions pane including

- **Foreach**, the enhanced for-loop, enabling developers to loop directly over collections in the interactions pane;

- **Variable arguments** in method and constructor declarations significantly reducing the amount of static overloading required to produce flexible APIs.

- **Static import** so that static members of a class can all be imported into the interactions pane using a single declaration.

- **Autoboxing** so that primitive values can freely be used in place of the wrapped counterparts and vice-versa, eliminating annoying clutter from program text in the interactions pane.

- **Enum** types facilitating the declaration of a collection of constant objects and their use in the interactions pane.

   Now that all of these new features are implemented in DynamicJava, the DrJava programming environment supports interactions expressed in the full Java 5.0 language. Note
that the feature implementation was done as a joint effort.

14.1 Enhancing DynamicJava was a Collaborative Effort

The new features of DynamicJava were implemented in a joint effort between the author of this dissertation, a fellow graduate student (Moez Abdel-Gawad), Professor and Supervisor Robert Cartwright, and students attending the Production Programming course.

In particular Moez Abdel-Gawad pair programmed with me while implementing the changes to DynamicJava’s grammar to cope with the new syntax of generic types. Moez also pair programmed with me while we implemented support for enums, static import and variable arguments.

For adding support for the autoboxing feature, I pair programmed with Johnathan Lugo, and parts of the implementation was also done by Professor Robert Cartwright.

The new for loop construct foreach was implemented while pair programming with Adam Wulf.

The initiative generifying of the old DynamicJava grammar was mainly done by Professor Robert Cartwright.

For future extensions to DynamicJava I suggest a few that I have encountered throughout this project.

14.2 Remaining Work

Although DynamicJava now supports all of the new features from Java 5.0, there are some deviations from Java 5.0 semantics remaining in DynamicJava that need to be rectified.

14.2.1 Propagating generic types in type checking

DynamicJava does not propagate generic type information when it performs type checking. Since DynamicJava does not claim to catch all or even most type errors in code in the interactions pane before it is interpreted, this omission is understandable. But it in-
teracts negatively with autoboxing. When an element is extracted from a generically type
collection, it typically has erased type integer but a much more precise static type (based
on parametric type information) that is enforced in compiled code by performing a cast to
the more precise type. But Java autoboxing rules depend on such type information. For
example, an Integer can be automatically unboxed to an int but an Object cannot.
Hence, an expression (such as \texttt{v.get(0)} if \texttt{v} has generic type \texttt{Vector<Integer>})
with erased type \texttt{Object} and generic type \texttt{Integer} will not be unboxed when it appears
in a context requiring an \texttt{int} (such as \texttt{v.get(0) + 5}) in DynamicJava but it will be in
compiled Java code.

To complete support for Java 5.0, DynamicJava must be extended to propagate generic
types which requires some generic type checking. This is a significant undertaking be-
cause the generic type checker in \texttt{javac} is many thousands of lines of nearly inscrutable
code, but fortunately, should be considerably easier than writing a full fledged generic type
checker.

Of course, DynamicJava could in principle incorporate a full-fledged generic type checker
but we think that would be a mistake because the interactions pane should be much more
forgiving than a compiler. If a code fragment has a sensible interpretation, then Dynamic-
Java should interpret it.

### 14.3 Disallow extension of the Enum Class

In DynamicJava, it is presently possible to declare a class that extends the \texttt{java.lang.Enum}
class, which is not allowed in Java 1.5. A "classes cannot directly extend java.lang.Enum" error
is thrown. A check needs to be placed in the Class Declaration node to prevent this
occurrence.
14.4 Add generic type checker

Right now there are no generic type checker connected to the interactions pane. Without this kind of type checker, there is no way to enforce the restrictions that comes with Generic declarations, as shown in example 68.

```java
import java.util.Vector;

Vector<String> v = new Vector<String>();
v.add(new Integer(5));
```

**Code Example 68:** Example where the generic type checker is needed.

In example 68 a collection, Vector of String Objects are declared. This declaration means that only Strings are supposed to be allowed to be added to the collection. Without the generic type checker, it is possible in DynamicJava to add any type of Object to the collection. There is no way to enforce the type restriction without it.

14.5 Generate bridge methods

Bridge methods are, in Java, compiler generated methods used to implement covariant result types and other instances involving generic types. Bridge methods ensure that the correct methods are called when sub classing a class with type parameters. Example 69 illustrate this more clearly.

Observing example 69, note that without the bridge method, the call to the method `myMethod` will result in the String “In Super” being printed out, and not “In Sub”. This happens because after erasure, `myMethod` will not be overridden in the class `Sub`, because after erasure the method in the class `Super` gets erased to `Object`, and not `String`. The compiler must introduce the bridge method to emulate the method being overridden, to cause the expected results. Since Java support method overloading, the compiler can easily insert that extra method.
class Super<T> {
    public void myMethod(T t) {
        System.out.println("In Super");
    }
}

class Sub extends Super<String> {
    public void myMethod(String t) {
        System.out.println("In Sub");
    }
}

Compiler generated bridge method

public void myMethod(Object t) {
    myMethod((String)t);
}

Calling the method:

Super<String> s = new Sub();
s.myMethod("Hello");

Code Example 69: Example of how the bridge methods work.

DynamicJava currently does not generate bridge methods, and in scenarios such as the preceding example, will cause unexpected results.

14.6 Solve the forward reference problem

When classes are declared in DynamicJava (as opposed to compiled code that DynamicJava can execute because it is on the classpath), Dynamically Java incrementally generates a class file for the new class and loads it. As a result, DynamicJava does not treat forward references in classes that it loads correctly. The forward referenced class is not available to DynamicJava at the time that it loads the referencing class.

The program 70 illustrates this problem.

This code fragment works in regular Java, but it fails in DynamicJava. The interface AbstractShapeVisitor doesn’t know the class Box yet, and therefore it will cause
interface AbstractShapeVisitor<T>{
    T forBox(Box b);
    T forSphere(Sphere s);
}

interface AbstractShape{
    public <T> T accept(AbstractShapeVisitor<T> v);
}

class Box implements AbstractShape{
    public <T> T accept(AbstractShapeVisitor<T> v){
        return v.forBox(this);
    }
    ....
}

**Code Example 70:** Example of forward reference.

an error. If the class Box to the beginning of program the fragment, it will complain about not knowing the AbstractShape type yet. If the AbstractShape interface is then moved above location of where the class Box is declared, it will not be able to find the AbstractShapeVisitor type. And if the AbstractShapeVisitor interface is moved above the AbstractShape interface, they are all back where they started. In regular Java this is not a problem because the compiler has the entire program available when it compiles each class. In DynamicJava, the class declarations must be processed and loaded sequentially.

It may be possible for DynamicJava to support some forward references by recognizing including those in the preceding example by deferring all references to classes that do not yet exist (at the time a class declaration is translated to a class file) until absolutely necessary. Some indirection using reflection may be required.

### 14.7 Reflections on Java, Extreme Programming, and Generics

This extensions of DynamicJava to support Java 5.0 has been a very instructive software development project. We learned that an Extremely Programming team could cope with
extending a foreign code base with acceptable (if not elegant) OO structure and no supporting unit testing harness by retroactively writing some essential unit tests at reasonably large levels of granularity. Over time, we anticipate filling in more holes in the current unit testing coverage.

We also learned that well-written Java programs — even programs written by unfamiliar authors, little documentation and no unit tests — really are amenable to significant extension with only modest rewriting and refactoring of existing code. A much weaker version of the same proposition, namely that well-written OO code written by the development team backed by extensive unit tests is amenable to significant extension with only modest rewriting and refactoring. We were delighted that a much stronger version of this proposition held for the DynamicJava code base.

Even though we would have much preferred support for first-class generic types in Java 5.0*, we learned to live with and appreciate the value of Java 5.0 generics based on erasure. The DrJava and DynamicJava code bases make extensive use of generics. One of the major changes that we made to DynamicJava was to convert the program to use generic types. We found the inability to cast to type T where T is a “naked” type parameter annoying (an erasure-based implementation of generics cannot support this operation because the binding of T has been erased at runtime) because we needed this capability to glue our generic code to ungenerified libraries including some of Sun Microsystems’ own libraries. The Java 5.0 compiler weakly accommodates such casts by implementing them as no-ops and generating warning messages flagging them as unsafe.

If Java 5.0 supported first class generics, we might have been able to completely finesse the issue of propagating generic type information at runtime because the objects themselves carry it. Of course, runtime types can be narrower than the corresponding static types (which govern autoboxing/unboxing transformations and static overloading resolution in compiled code) but the parametric type information must match exactly. Hence, the static

*First-class generic types are available at run-time and can be used essentially anywhere than conventional non-generic types can.
types of elements stored in vectors will be revealed by the run-time type of the vector. The weak non-generic static type checking performed by the original DynamicJava conceivably might have sufficed for interpreting first-class generic Java as well. It depends on the details of how the first-class Java dialect was designed. If the design followed the JavaPLT research group's NextGen compiler [1] then the extant weak checking might have been enough because NextGen preserves a very high degree of compatibility with the erasure based semantics of Java 5.0; in fact, it handles autoboxing and static overload resolution exactly like Java 5.0.
Bibliography

   Efficient implementation of run-time generic types for Java.

   Objects first with Java.

   JUnit home.

   Making the future safe for the past: 'Adding Generics to the Java Programming
   Language'.

   Clover homepage.

   Journal of functional programming.
   Volume 12(Number 2), 2002.

[7] GNU.
   CVS - concurrent versions system.

[8] GNU.

Emacs.


*Introduction to Computing and Programming in Java.*


Dynamicjava homepage.

http://koala.ilog.fr/djava.


*Extreme Programming Installed.*


*Object Oriented Analysis & Design.*


Beanshell homepage.


*An introduction to Programming and Object-Oriented Design Using Java 2nd Edition*


programming in an interactive environment: the lisp experience.

A Framework for Building Pedagogic Java Programming Environments.

[18] SUN.
Enums.
http://java.sun.com/j2se/1.5.0/docs/guide/language/enums.html.

[19] SUN.
Java 1.5 api.
http://java.sun.com/j2se/1.5.0/docs/api/java/lang/Class.html.

[20] SUN.
Javacc home.
http://javacc.dev.java.net.

Ant - The Definitive Guide.

Adding wildcards to the java programming language.

C++ Templates - The Complete Guide.

    Groovy homepage.
    http://groovy.codehaus.org/Home.

    Groovy homepage, differences between groovy and java.
    http://groovy.codehaus.org/Differences+from+Java.

    Groovy homepage, methods added to the jdk.