Type Safety without Objects in Java

(Spine title: Type Safety without Objects in Java)

(Thesis format: Monograph)

by

Pavel Bourdykine

Graduate Program
in
Computer Science

A thesis submitted in partial fulfillment
of the requirements for the degree of
Master of Science

School of Graduate and Postdoctoral Studies Studies
The University of Western Ontario
London, Ontario, Canada

THE UNIVERSITY OF WESTERN ONTARIO
THE SCHOOL OF GRADUATE AND POSTDOCTORAL STUDIES

CERTIFICATE OF EXAMINATION

Supervisor:	Examination committee:
Dr. Stephen Watt	Dr. Daley
	Dr. Jeffrey
	Dr. Schost

The thesis by

Pavel Bourdykine

entitled:

Type Safety without Objects in Java

is accepted in partial fulfillment of the
requirements for the degree of
Master of Science

Date _____ _____
Chair of the Thesis Examination Board

Abstract

Many object-oriented programming languages provide type safety by allowing programmers to introduce distinct object types. In the case of Java this also introduces a considerable or even prohibitive cost, especially when dealing with small objects over primitive types. Consequently, Java library implementations typically abuse primitive types and are not type safe in practice. We present a solution that allows type safety in Java with little, if any, performance penalty, hence allowing for development of safe and efficicient applications and libraries.

We present a solution that provides the safety of object-oriented code, but avoids all overhead when the full generality and expressive capabilities of objects are not required. This is accomplished by treating named objects as primitive types during compilation. This allows for reusable and easily maintainable Java code that rivals natively compiled languages in efficiency. The proposed technique differs from the previous work in that distinction between objects is made by name, rather than implementation. Software implemented using the approach results in an order of magnitude improvement in execution speed and space use. It is likely that a native implementation and integration of our technique will also improve compilation time and ease of use, thus encouraging developers to use *opaque* object types in Java.

Keywords: Java Objects, Efficient Java, Type safety, Opaque types.

Acknowledgements

I would like to thank, first and foremost, my supervisor, Prof. Stephen M. Watt, for his support and guidance throughout the development of the ideas and software tools related to this work. Without his useful suggestions and our extended discussions about optimizing the techniques and concepts described in this thesis, the work could not have been completed.

I'd also like to acknowledge the valuable insight I received from speaking to several experts in the area of programming languages while attending conferences and joint lab meetings with my colleagues at ORCCA (Ontario Research Centre for Computer Algebra) and SCG (Symbolic Computation Group) laboratories. I received many useful innovations and beneficial ideas while discussing the subject with both my colleagues and invited speakers in the research area.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Setting

The goal of this thesis is to introduce, define, and implement an approach for a type-safe implementation of fast object types in Java. The new object types resemble all aspects of regular objects in regards to safety, clarity, and type distinction. However, their performance properties coincide more with performance of primitive Java types in practice. We give the new type category the name, *opaque types*, due to the primitive type representation hidden beneath an object-like name. The name was also chosen in part due to the similarity to the feature being developed in the C++ language - *opaque typedefs*[2, 3].

Mainstream object oriented languages such as Java and C++ take advantage of many years of research and programming practice to produce highly expressive, type-safe code, and generate efficient low level code. All of these properties are of great importance for a language to be successful. Despite this, there are still many instances where shortcomings of language design and definition result in code that is awkward and inefficient. To remain effective, the language must be adequately usable, maintainable, and effective at solving a variety of problems from a multitude of domains. Programming language research has been moving in the direction to accomplish these goals. One branch of this research focuses on the area of type systems in programming languages.

David A. Watt defines a *type* to be "a set of values, equipped with one or more operations that can be applied uniformly to all these values."[16] Typing in programming languages improves many aspects of the language quality. The language becomes more readable, and, some would say, more writable as variables with specific types make a better fit into programs as a whole. Checking variable types during com-

pilation increases the language safety and performance as resources can be allocated more effectively knowing the types of data expected.

However, it is sometimes the case that in order to abide by strict typing rules imposed by the language, sacrifices have to be made in performance and solution style. In the case of Java, strict type-checking limits the application performance levels due to excessive use of objects, requiring a large amount of memory allocation and garbage collection. On the other hand, implementations utilizing primarily primitive types are not expressive enough and are often error prone in practice. Such implementations typically make little type distinction between the data they represent and hence cause ambiguities and confusion in the code. Moreover, it is often difficult to encode complex ideas and concepts using predominantly primitive types.

The pathway of declaring many unnecessary objects is taken primarily. Use of full-sized objects to express ideas and data that can be represented using primitive types is prominent due to convenience using objects offers. Objects allow safer and more intuitive code that is more representative of the concepts being implemented. In general, code using objects takes a conceptually higher level form and is therefore, usually, easier to develop.

Our solution involves combining readability and safety of regular object types with efficient use of primitive types in order to exploit the best properties both have to offer. In our approach, we would like to capture the accessibility objects offer in Java code and take advantage of innately faster and more efficient primitive types.

Primitive types are types whose values cannot be decomposed into anything simpler in the programming language, i.e. they are *primitive values*[16]. Primitive values are, by their nature, usually faster to access and manipulate and therefore result in an overall efficiency improvement compared to composite data. Currently, there are some proposed mechanisms that allow developers to utilize efficiency of primitive types and also keep their code clean, and maintainable. A language well-suited for mathematical computing, Aldor, contains mechanisms for a similar kind of representation. Objects can have an underlying representation, indicated by the "Rep" type, that specifies how values of the type are really represented.[17] Another major example of this is the *opaque typedefs* feature being developed in C++. The characteristic feature of *opaque typedefs* that drives their development in C++ is overloading. The ability to overload functions and operators based on several newly-defined *opaque types* allows code flexibility and intuitive development without forfeiting performance by creating new object types[3]. However, *opaque typedefs* do not boast a very clear definition, and the types' substitutability is not always apparent.

Compilers for mainstream languages have become progressively smarter. Generated code is highly optimized and compilers often attempt to identify programs' "hot spots" (frequently executed areas) and focus most of the optimization on those sections. Many performance benchmarks for programming languages are widely available on the World Wide Web. The benchmarks become a valuble resource for selecting a particular compiler (and possibly programming language) before beginning development. Given the opportunity, this allows developers to choose the precise tools they are going to use to solve a particular problem. One example of such a collection of benchmarks exists on *The Computer Language Benchmarks Game* website. It presents a decent comparison of up-to-date versions of today's common production languages. The database is organized as a series of measurements of *execution time*, *memory use*, and *source code size* of various algorithms implemented in different languages and executed on different platforms[4]. This library of performance benchmarks has helped us gauge where the current version of Java stands in comparison to other languages, as well as, in developing performance tests for our approach.

Despite all these advances there are still situations where it is either too difficult/impossible to apply an implemented feature or the code is not fully optimized due to the presence of high level concepts that are obscure to the compiler. This is apparent in a diverse collection of problems ranging from computer mathematics to video games. In computer algebra, for example, efficient computation-heavy algorithms are required to provide accurate calculations and approximations for solutions to various algebraic problems, such as, solving differential equations or modelling complex behaviours using multivariate polynomials. A lot of effort is put into developing and optimizing the algorithms by hand due to non-trivial concepts that are difficult to express efficiently.

1.2 Our Contribution

To deal with these issues we propose to look further into enhancing the type system present in the Java programming language. Giving the developer more control over which variable types he or she can use to accomplish the task at hand can impact code efficiency. The goal then becomes to combine flexibility offered to the developer without obfuscating the language semantics and, at the same time, introducing a technique for generating high performance low level code. This does not have to come at a significant cost to usability and readability of the language and if used correctly can drastically improve performance. Thus we put forth a new kind of

object in Java. The new object type (*opaque type*) follows all standard Java object rules but is implemented as a low level primitive type when it is compiled.

The reasons behind introducing these mechanisms in Java become evident when one examines Java's abilities as a programming language. It is a well established, industry-standard language with thousands of libraries available for interaction with other software and implementation of a multitude of concepts. It is a language with clean semantics, and flexible constructs that can be combined and molded to represent solutions to problems from many domains. Java has also come a long way in being a relatively fast executing language. Well written programs perform near the speed of solutions implemented even in natively compiled languages such as C or C++[4]. Blending Java's versatility and its potential for efficiency results in a rich language applicable to a vast amount problems with implemented solutions that are useful in practice.

We implement the approach by augmenting the language type system with structures that mirror primitive types and objects simultaneously, taking the clarity and distinction typical of object types and affixing primitive type efficiency. We base object-type distinction on name rather than implementation resulting in the developer having a choice of how he or she wants to treat data that may look identical at the binary level. This introduces a "best of both worlds" advantage, where code efficiency is benefited due to use of primitive types, and code correctness does not suffer due to use of high level object "handles".

The approach focuses mainly on creating a framework where code correctness, reusability, and readability are not hindered by low level optimizations which are characteristic when working with primitive types. The techniques are aimed at applications that can use low level fine tuning in order to achieve better execution speed and lower memory use while maintaining the code's integrity.

The remainder of this thesis is organized as follows:

Chapter 2 provides an overview and examples of previous work related to improving programming language type systems for the sake of efficiency and clarity. Brief analysis and discussion of previously developed methods attempts to highlight features that have had moderate and high success at accomplishing the set out goals in other languages. Chapter 3 describes our approach to enhancing the Java type systems with *opaque types* and discusses the theory behind the specific properties chosen for the new types. The theory helps justify the particular choices that were made along the way to finalizing *opaque type* implementation. Chapter 4 is dedicated to detailing the implementation itself. It describes the technical design of *opaque*

types, what tools were developed in order to integrate them into the language, and how those tools were architectured. Chapter 5 presents a number of tests that were carried out to test what effect *opaque types* have on Java application performance. The experiments were chosen from a variety of areas and the reasoning behind the selections is discussed in this chapter. Chapter 6 draws the final conclusions regarding the practicality and general success of *opaque types* and proposes a number of topics to spark future work in the area of *opaque type* implementation and further possible improvements on the subject.

Chapter 2

Previous Work

2.1 The concept of abstraction

Abstraction is a powerful concept in software development. Concealing the details of a particular implementation behind an interface is a common practice and has been employed in different languages. Various implementations give rise to benefits in efficiency, portability and maintainability of large software projects[12]. Given the maturity of such concepts it is useful to differentiate between some previously developed methods that are similar and the approach proposed here. The common goal in the advancement of these concepts is to make the language in question more readable, and easily maintainable. Some techniques described below sacrifice efficiency in order to promote usability; the others, maintain efficiency at the cost of clarity.

It is important to illustrate how this and other concepts have been implemented in the past in various programming languages. Since it is useful to draw on the previous ideas, the detailed examples below help demonstrate the train of thought that was followed when *opaque types* were designed. In each case, the previous work brought something good to the table and it was a matter of reshaping the core idea in order to augment the guiding theory behind *opaque types*.

The discussion and examples are organized as follows:

Section 2.2 describes a simple mechanism for creation of new distinct type versions in C. In Section 2.3, a new, work-in-progress, mechanism for new type definition in C++ is discussed. Section 2.4 summarizes the use of "object branding" in the Modula-3 language. Section 2.5 presents an overview of the process of defining new types in Standard ML to illustrate contrast between approaches taken to accomplish this task in functional and imperative languages. Section 2.6 highlights differences and

similarities between the presented examples and the *opaque types* approach proposed for Java.

2.2 C structs

In C, there exists a simple mechanism for creation of basic structures (or records) in the form of *struct*s. Consider a primitive example such as two structs (named A and B) whose internal structure is exactly the same - namely, a single field of type *int*.

Figure 2.1: C structs

```
1 struct A {
2   int a;
3 };
```

```
1 struct B {
2   int b;
3 };
```

While this successfully distinguishes between the two types that both actually represent a single integer, optimization on structs is difficult for more complicated cases and does not always result in the most efficient code[10]. Moreover, as structures grow in size and become more complex, it is no longer convenient or beneficial to use them as new primitive types. Functions requiring access to the structure (as well as functions that need to modify its fields) must be passed a pointer to the *struct* type. Finally, while the identically constructed A and B *struct*s are distinct in name, the C standard provides no way for distinguishing between functions that can access the *struct*s' internal fields apart from explicitly specifying function parameter types[13]. A similar approach in Java, can be taken by creating different Object types whose underlying representation is identical. The nature of Java (being an Object-Oriented language) allows the creation of methods specific to either type but then we run the problem of dealing with "heavy" object types that need not be there at all after static type-checking phase is complete[1].

As an alternative to the *struct*s A and B, consider the use of a single variable type that is the same size as an *int*. The new type would resemble the *int* type in every way with the exception of being named differently. For clarity reasons, this might require two additional operations for converting to and from the new type and the regular *int* type. However, the use of a single built-in type compensates for the additional operations by being more optimizable and easier to use. In the long run, performance and usability trump the initial investment of extra operations.

2.3 C++ opaque typedefs

The *struct* approach can also be taken in C++ for creation of new types that are distinct in name from anything else. The low level implementation of the new types is irrelevant and therefore one can essentially redefine a primitive type by wrapping it in a *struct* construction.

The most recent work to implementing an analogous scheme and accomplishing similar results has been progress towards the introduction and development of *opaque typedefs* in C++. Despite the main motivation behind *opaque typedef* introduction being function and operation overloading, it is clear that *opaque typedef*s can also bring performance benefits to C++. The relation between the approach presented here and C++ *opaque typedef*s can be clearly seen with the aid of the next example. Figure 2.2 presents a possible use of *opaque typedef*s to define variables corresponding to Cartesian and Polar 3D coordinates. The coordinate behaviours and operations are different; however, they are both represented by the same underlying primitive type - *double*. This kind declaration essentially allows renaming of the primitive *double* type in order to abstract varying operations of the coordinates by using the new type name while maintaining efficient internal representation of the floating point number.

Figure 2.2: C++ opaque typedef example

```
1  opaque typedef double X, Y, Z;          // Cartesian 3D coordinate types
2  opaque typedef double Rho, Theta, Phi;    // Polar 3D coordinate types
3
4  class PhysicsVector
5  {
6  public:
7    PhysicsVector(X, Y, Z);
8    PhysicsVector(Rho, Theta, Phi);
9    ...
10 }; // PhysicsVector
```

The current progress towards *opaque typedef*s in C++ outlines two kinds of declarations. The two kinds of typdefs are *public* and *private*. The two flavours of *opaque typedef*s attempt to deal with a particularly important issue of implicit conversion between the newly created type and the underlying primitive type. The notion of *substitutability* then becomes vital to understanding which type of *typedef* is appropriate for which situation. *Public* and *private* keywords extend traditional *typedef*s with forms of transparency. The theory behind this is described as follows:

> Guided by well-understood *substitutability* principles as embodied in todays C++, we believe there is value in proposing to extend classical transparent `typedefs` with two forms of opacity. We have designated these new forms, respectively, as *public* and *private*. The former would permit *substitutability* in one (consistently specified) direction, the latter would permit no *substitutability* at all, while classical *typedef*s would continue to permit mutual substitutability. [3]

Unfortunately, the two kinds of opaque typedefs proposed (*public* and *private*) introduce some difficult to deal with complexity. The behaviour of *public* and *private* *typedef*s is modelled after the way the keywords are used with inheritance properties in C++. Extended by the previously mentioned notion of *substitutability*, *typedef* *public* is, perhaps, best summarized as:

> The semantics of the proposed `public typedef` would permit similar *substitutability* in that instances of a newly declared type (the *opaque-type*) may be used wherever an instance of the original type (the *underlying-type*) is expected. Unlike the mutual *substitutability* induced by a classical `typedef`, an instance of an *underlying-type* may not stand in where an instance of the opaque-type is expected. Further, an instance of a `public typedef` may never stand in for an instance of a second *opaque-type*, even when both have the identical underlying-type. [3]

Conversely, the *private typedef* is anticipated to fill the need for a non-*substitutable* type. However, the necessity of such a type is, in the end, debatable due to lack of a substantial application domain. [2, 3]

2.4 Modula-3 *BRANDED* Objects

A technique, similar to our approach, exists in the Modula-3 language, where the keyword *BRANDED* is used to distinguish between two structurally identical objects. Since the Modula-3 type system uses structural equivalence instead of name equivalence, it is often the case that simple objects that are the same in composition should be differentiated when static type-checking takes place. This contrasts with C and C++ *structs* where identically structured objects are already type-distinct as long as they are named different. The *BRANDED* keyword in Modula-3 is used exactly

for the purpose of differentiating between identically constructed objects. Explicit naming of objects by "labeling" or "branding" guarantees determinism when objects with identical format are type-checked. When the object is not "branded" explicitly, the compiler makes up a name for the object but does not assure distinction between structurally indistinguishable objects[5].

An example of this kind of construction follows. It is a simple implementation of a hierarchy of 2-Dimensional geometrical shapes beginning with a general *Shape* interface, subclasses of a generic 2-D shape - *Rectangle*, and *Circle*, and their respective implementations.

Type-checking and compiling the similarly constructed objects becomes dependent on the use of the *BRANDED* keyword. Modula-3's type distinction is based on the internal representation of the objects and therefore does not guarantee type contrast in the absence of the *BRANDED* keyword. This is important as the close relationship between *Rectangle* and *Ellipse* objects can result in similar and simultaneous use of their instances. This has the ability to result in confusion when determining substitutability properties during type resolution[11].

Figure 2.3: Modula-3 Shape Interface

```
1  INTERFACE Shape;
2    TYPE
3      T <: Public;
4         Public = ROOT OBJECT
5           METHODS
6              draw();
7              moveTo(newx: INTEGER; newy: INTEGER);
8              rMoveTo(deltax: INTEGER; deltay: INTEGER);
9              getX(): INTEGER;
10             getY(): INTEGER;
11     END;
12 END Shape.
```

The *Shape* interface introduces a generic 2-Dimensional shape that is to become the superclass of the possible shapes. It has general methods for drawing and manipulating the shape representation. Some of the method implementations can be specified in the *Shape* object (module) that follows, while others are left "abstract" to be implemented by the subclass. This allows subclass objects to resemble the general *Shape* and be classified as such, while adding their own, alternative, behaviours into the implementation without repeating code.

Figure 2.4: Modula-3 Shape Implementation

```
1  MODULE Shape;
2    REVEAL
3      T = Public BRANDED OBJECT
4        x: INTEGER;
5        y: INTEGER;
6        METHODS
7          setX(newx: INTEGER) := SetX;
8          setY(newy: INTEGER) := SetY;
9        OVERRIDES
10         moveTo := MoveTo;
11         rMoveTo := RMoveTo;
12         getX := GetX;
13         getY := GetY;
14     END;
15
16   (* Procedure implementations *)
17   ...
18 BEGIN
19 END Shape.
```

The implementation of the *Shape* module reveals that a general 2-D shape has just two fields of type *INTEGER*. These fields can correspond to location of the shape on a Cartesian grid. While it is unimportant what the fields of the object represent exactly, it is essential to note that these fields play an important role in how the object behaviours are implemented. The method implementation (which mainly consists of accessor and mutator methods for the object fields) is left out of the figure (2.4) as it is unimportant to demonstrating the hierarchichal system example here.

The *Rectangle* interface is a 2-D shape and therefore imports it as the first statement of the interface declaration. Along with 2-D *Shape* methods, *Rectangle* has the properties of width and height, which also happen to be of type *INTEGER*. This augments the 2-D shape representation and acts upon the actions possible involving the shape.

Implementation of the *Rectangle* type reveals that *Rectangle* is a "branded" object with *INTEGER width* and *height* fields. Along with the x and y *INTEGER* fields inherited from *Shape*. Thus the *Rectangle* object is represented by four integers and the actions corresponding to them.

The *Ellipse* interface also imports *Shape* as the first statement of its declaration. This implies that an *Ellipse* should at least have *INTEGER* type x and y coordi-

Figure 2.5: Modula-3 Rectangle Interface

```
1  INTERFACE Rectangle;
2    IMPORT Shape;
3    TYPE
4      T <: Public;
5      Public = Shape.T OBJECT
6        METHODS
7          init(x: INTEGER; y: INTEGER; width: INTEGER; height: INTEGER): T;
8          getWidth(): INTEGER;
9          getHeight(): INTEGER;
10         setWidth(newwidth: INTEGER);
11         setHeight(newheight: INTEGER);
12     END;
13 END Rectangle.
```

Figure 2.6: Modula-3 Rectangle Implementation

```
1  MODULE Rectangle;
2    IMPORT IO;
3    IMPORT Fmt;
4    REVEAL
5      T = Public BRANDED OBJECT
6        width: INTEGER;
7        height: INTEGER;
8        OVERRIDES
9          init := Init;
10         getWidth := GetWidth;
11         getHeight := GetHeight;
12         setWidth := SetWidth;
13         setHeight := SetHeight;
14         draw := Draw;
15     END;
16
17   (* Procedure implementations *)
18   ...
19 BEGIN
20 END Rectangle.
```

nates inherited from the *Shape* object. Along with the 2-D coordinates, the methods indicate the presence of a major and minor axes characteristic of a typical ellipse.

Eclipse module reveals the rest of the internal representation of a 2-D elliptical shape. Along with x and y coordinates from the *Shape* interface, *major* and *minor* *INTEGER* fields are declared. This makes the internal representation of an *Ellipse*

Figure 2.7: Modula-3 Ellipse Interface

```
1  INTERFACE Ellipse;
2    IMPORT Shape;
3    TYPE
4      T <: Public;
5      Public = Shape.T OBJECT
6        METHODS
7          init(x: INTEGER; y: INTEGER; major: INTEGER, minor: INTEGER): T;
8          getMajor(): INTEGER;
9          getMinor(): INTEGER;
10         setMajor(newMajor: INTEGER);
11         setMinor(newMinor: INTEGER);
12     END;
13 END Circle.
```

Figure 2.8: Modula-3 Ellipse Implementation

```
1  MODULE Ellipse;
2    IMPORT IO;
3    IMPORT Fmt;
4    REVEAL
5      T = Public BRANDED OBJECT
6        major: INTEGER;
7        minor: INTEGER;
8        OVERRIDES
9          init := Init;
10         getMajor := GetMajor;
11         getMinor := GetMinor;
12         setMajor := SetMajor;
13         setMinor := SetMinor;
14         draw := Draw;
15     END;
16
17   (* Procedure implementations *)
18   ...
19 BEGIN
20 END Circle.
```

module identical to the *Rectangle* representation. Four *INTEGER* type fields play the role of both an *Ellipse* and a *Rectangle* type internally. This is of no surprise given the innately close relationship and a rectangle and an ellipse have in the Cartesian coordinate system. The biggest ellipse that may be inscribed in a given rectangle is unique thus giving the ability to represent an ellipse in terms of a rectangle. The opposite is also true, a rectangle may be described by a given ellipse by either placing a

rectangle inside with the largest possible area, or inscribing the eclipse in the smallest possible rectangle. Both representations are again unique.

The use of "branding" allows the compiler to attach a permanent name to both structures, thereby eliminating the ambiguity that may arise from usage of the objects. Figure 2.9 illustrates using 2-D shapes in a simple example, it attempts to demonstrate the necessity of the *BRANDED* keyword in distinguishing between the similar structures.

Figure 2.9: Modula-3 Shape Use Example

```
1  MODULE Main EXPORTS Main;
2    IMPORT Shape;
3    IMPORT Rectangle;
4    IMPORT Ellipse;
5
6  VAR
7    test : ARRAY[1..2] OF Shape.T;
8    rect : Rectangle.T;
9
10 BEGIN
11   (* set up some shape instances *)
12   test[1] := NEW(Rectangle.T).init(10, 20, 5, 6);
13   test[2] := NEW(Ellipse.T).init(15, 25, 8, 4);
14
15   (* iterate through some shapes and handle polymorphically *)
16   FOR i := 1 TO 2 DO
17     test[i].draw();
18     test[i].rMoveTo(100, 100);
19     test[i].draw();
20   END;
21
22   (* access a rectangle specific function *)
23   rect := NEW(Rectangle.T).init(0, 0, 15, 15);
24   rect.setWidth(30);
25   rect.draw();
26 END Main.
```

The implementation of the *draw()* method is intentionally left out of the *Rectangle* and *Ellipse* examples as it is likely different. The use of the same method signature implies that successful type checking needs to take place prior to code generation at lines 17 and 19 of Figure 2.9 in order to determine the correct version of *draw()* to be used with the appropriate shape type.

Structural equivalence between the *Rectangle* and *Ellipse* representations requires distinction between the two objects to be made by name. This is where Modula-3's

use of the *BRANDED* keyword is most similar to the approach described in this work. Java's type-checking is name-based. Thus, in theory, it allows for renaming of structures as basic as primitive types for the purposes of readability, clarity, and usability. It is precisely this renaming that we intend to describe and implement in this work without loss of Java's regular language features.

2.5 Standard ML

Other languages also include constructs that introduce abstraction to types by hiding implementation details. Abstract data types in Standard ML, for example, also accomplish the same tasks as Java-like languages, although the language does not support sub-typing or implicit casting between these types. In general, the produced code is more modularized with high reuse potential, software complexity is reduced, and independence of the implementation is assured[6].

For example, new data-type declarations in Standard ML may look something as follows. The *suit* declaration represents a playing card suit and has a set of functions that rank the suits. Meanwhile the *tree* declaration can play the role of an abstract tree holding any kind of data at each node. The general tree operations such as *Empty* will work regardless of the data types held at the nodes.

Examples presented in this chapter serve as an introduction of previously implemented mechanisms and techniques for accomplishing a task that is similar to our goal. The goal of creating a language with enough expressive power to tackle many different problems and remain efficient at solving those problems. These examples also demonstrate concepts and ideas which we were able to compare and contrast in order to provide correct theory behind our approach.

The solution outlined and implemented by us is a type of abstraction that focuses on distinction of similar or identical underlying primitive types. That is to say, while many *opaque types* may have the same underlying primitive type they represent vastly different objects (not to be confused with actual Java object types) and therefore must type-check differently. For example, one may think of a multitude of objects that may be represented by the primitive built-in *int* type but in order to assure successful static type-checking, those objects should appear different to the compiler prior to compilation. Our approach accomplishes exactly that.

The approach is implemented by way of preprocessing the source code in order to transform regular object types into *opaque types* prior to compilation. A preprocessor

Figure 2.10: Standard ML - new type

```
datatype suit = Spades | Hearts | Diamonds | Clubs

fun outranks (Spades, Spades) = false
  | outranks (Spades, _) = true
  | outranks (Hearts, Spades) = false
  | outranks (Hearts, Hearts) = false
  | outranks (Hearts, _) = true
  | outranks (Diamonds, Clubs) = true
  | outranks (Diamonds, _) = false
  | outranks (Clubs, _) = false

...

datatype 'a tree = Empty | Node of 'a * 'a forest
and 'a forest = Nil | Cons of 'a tree * 'a forest

fun size_tree Empty = 0
  | size_tree (Node (_, f)) = 1 + size_forest f
and size_forest Nil = 0
  | size_forest (Cons (t, f')) = size_tree t + size_forest f'
```

for Java is mostly unnecessary and the language, by design, has strict rules that make using existing preprocessors (such as CPP - the C preprocessor) difficult. Thus we are forced to implement the preprocessing step as part of our multi-step building process rather than integrating it into the compilation step.

Chapter 3

Our Approach

3.1 Approach methodologies

We introduce the notion of *opaque types* in the Java programming language. These *opaque* types allow development of Java code that is reusable, elegant, and efficient. These types are meant to be used as regular `Object` types that can be represented by primitive built-in Java types, while still requiring to behave and act like regular `Object` types in the way they interact with the Java class hierarchy and static type-checking. An example of this kind of application may be an object that has a small finite number of different states that can intuitively be represented by a set of bit patterns. Although this can be implemented similarly to something written in assembly language, by using *int* types, resulting in code that is quite efficient, the code's extensibility would suffer. Moreover, like assembly, this type of code is difficult to maintain, and debug[7, 8]. This may lead to errors that could have been easily avoided if object types were used.

Opaque Java types are implemented by recognizing when a regular object may be directly implemented as a primitive type. This frequently occurs in practice when the data represented does not contain many pointers into memory. Special textual replacements take place prior to compilation in order to turn most of the object's representation to its now underlying primitive type. The only exception is the object's actual name. The object's name serves as a trace for the Java compiler to perform static type-checking and for our preprocessor to handle inheritance properly. The end result is a "thin" object that is type checked properly by the Java compiler and for which, the generated code is very efficient as it uses the underlying primitive type.

Additionally, the approach encompasses a core notion of *opacity*. High level Java objects do not necessarily have to be represented or compiled as such. Objects simply

serve as identification handles for static type-checking prior to compilation. The underlying type of these objects may be anything suitable for internally representing the construction. In this fashion, an alternative *String* object may be represented by a character array allowing for operations very similar to those on strings implemented in C or C++. In turn, a more complex object (e.g. an object representing a DNA sequence) may be represented by such an alternative *String* thus creating an artificial class hierarchy that remains consistant and type-safe. In this work, however, we are mostly concerned with objects that may be represented by primitive types in order to boost performance.

Along with the optimized version of the *opaque type* the regular unchanged version of the class is kept for reference and debugging purposes. Leaving the user code unchanged after compilation allows for more straight forward top-level design where good Object Oriented Design practices may be followed. The user may also choose to compile the *opaque-typed* code and run it as is, without conversion, in order to ensure correctness. Keeping both versions of the class also demonstrates the type safety of *opaque* Java types as either version of the project will produce identical results when executed.

In order to successfully implement *opaque types* in Java, we introduce several type rules that have to be followed in order to utilize safety and efficiency of such objects. These rules are followed by the preprocessor to transform the user's regular object into one for which the generated code will use the underlying primitive type.

3.2 Type rules

We use a Java code annotation (called *Opaque*) to identify classes as *opaque types*. Java annotations allow embedding of metadata directly into Java source code. "Annotations do not directly affect program semantics, but they do affect the way programs are treated by tools and libraries, which can in turn affect the semantics of the running program."[15] The annotation has a single *String* type field that denotes the primitive representation type of the opaque object. For example, the annotation `@Opaque(‘‘int’’)` indicates that the object is *opaque* and that its primitive representation is of type *int*. Currently, the annotation field serves as a way to quickly identify the underlying type and speed up *opaque type* file analysis but could be left out in later versions of the solution. The single annotation dictates all the required information to the preprocessor. The next restrictions/rules must be followed in order to guarantee successful conversion consistent with the Java language standard:

Figure 3.1: Java opaque type rules

1. object must have a single `protected` field of the underlying type unless it is a subclass of an *opaque type*
2. object constructor(s) must be declared `private`
3. all methods accessing or modifying the underlying type field representation must be declared `static` (or `final static` if no subclasses override the methods)

Rule 1 enforces *opaque type* representation and assures that it matches with the type suggested by the annotation. The primitively typed field (whose name is standardized to `rep`) takes place of the *opaque* object whenever it appears in user code. It is important that its uses are properly implemented and there are no compilation issues post-conversion.

If the new *opaque* object `extends` an *opaque type* (a property detected by the preprocessing utility), the object must not include a `rep` field in its declaration. The `rep` field is inherited from the superclass and bares the same primitive type. This ensures consistency in method inheritance and conversion.

Rule 2 follows the Java convention that only object types require a constructor. Since the new *opaque* object is to be converted to its underlying primitive type representation wherever it used, its constructor must remain `private`. Creating new instances of the opaque object is still possible through the use of a `static` method "New". This method should be implemented by the user as a means of converting from the underlying primitive type to the object type primarily for testing purposes and initial implementation of code that uses *opaque types*. The typical implementation is very simple:

Rule 3 places a restriction on the other methods possibly acting on the object representation. Default visibility `static` methods allow inheritance and class access to regularly used operations within the new object. At first glance this may seem limiting for using the object; however, since object instances are all converted to the underlying primitive type, only class methods remain as valid operations that can act upon the object `rep` field. This method declaration makes it easier for the preprocessor to handle extension quickly and efficiently and makes sure the opaque object is not inflated by non-static behaviors.

Following the *opaque* object restriction rules assures preprocessor compatibility

Figure 3.2: Opaque object, typical "New" implementation

```
1  @Opaque(``short'')
2  public class MyOpaqueObject {
3    protected short rep;
4    private MyOpaqueObject(short r){
5      rep = r;
6    }
7
8    ...
9
10   public static MyOpaqueObject New(short r){
11     return new MyOpaqueObject(r);
12   }
13 }
```

with the developed code. The restrictions also implicitly impose some type rules on *opaque types*. The use of a `protected` representation field and the field's absense in *opaque* object subclasses imply the presence of a concrete relationship between *opaque* subclasses and superclasses[14]. Namely, because they are represented by a primitive type, the inheritance relationship requires no special consideration as superclass behaviours are immediately applicable to the subclasses. Additionally, type distinction remains in tact, as type-checking is done prior to code generation. This solidifies the Object Oriented "is a" property when dealing with class hierarchy. Consider the example in Figure 3.3.

The *BaseClass* follows regular rules proposed in Section 3.2. The *@Opaque("int")* annotation suggests the object is represented internally by the *int* type. The object's single field is `protected int rep`. The constructor is left *private* as per rule 2. There is a single method called *operator* shown in the implementation while the rest is omitted as it is irrelevant to demonstrating the inheritance properties in *opaque* objects.

Both *ChildClassOne* and *ChildClassTwo* are subclasses of the base class and therefore share its internal representation type - *int*. Neither of these classes possess a *rep* field as it is inherited from *BaseClass*. Along with the `protected` field, both classes have access to the *operator* method. The class hierarchy is unchanged from the regular Java standard.

ChildClassThree `extends` *ChildClassTwo* and is therefore also represented by the integer type. Its *rep* field is too inherited from the initial *BaseClass*. The *operator*

Figure 3.3: Opaque object extension properties

```
@Opaque(``int'')
public class BaseClass {
  protected int rep;
  private BaseClass(short r){
    rep = (int) r;
  }

  public static void operator(BaseClass bc, short modifier){
    ...
  }

  ...
}
```

```
@Opaque(``int'')
public class ChildClassOne
      extends BaseClass {

  private ChildClassOne(short r){
    rep = (int) r;
  }

  ...
}
```

```
@Opaque(``int'')
public class ChildClassTwo
      extends BaseClass {

  private ChildClassTwo(long r){
    rep = (int) r;
  }

  ...
}
```

```
@Opaque(``int'')
public class ChildClassThree extends ChildClassTwo {

  private ChildClassThree(int r){
    rep = r;
  }

  public static ChildClassThree operator
    (ChildClassTwo modifier, short cc){
    ...
  }

  ...
}
```

method is overridden in *ChildClassThree* with new parameters and a new implementation. Polymorphism is a preserved paradigm in our approach as the overridden methods are `static` and are therefore always preceeded by the correct class name when they are invoked in user code.

Next, it is useful to see how these sample classes would look after being con-

verted to the representation type. The changes are minor but are essential to the approach. Figure 3.4 illustrates the mock-up objects of Figure 3.3 after invocation of the conversion utility.

The code is analyzed and the conversion takes place based on the *opaque type*'s representation (as per the *@Opaque* annotation). The field type is checked against the annotation and if there is no discrepancy, all `static` methods are converted to use the underlying primitive type. Parameters of these methods are also converted. Change in parameter and return types of the methods introduces new method signatures. Unfortunately this implies that if some of the unconverted objects were already compiled to `.class` files previously, a recompilation is required. This, however, is almost always a small price to pay for the benefits that the new, primitively typed, methods will bring.

It is essential to note that the class hierarchy is preserved after the conversion process. This is an important property that is unique to *opaque types* in Java and what distinguishes our approach from other implementations in various languages. Class name invariance before and after code conversion guarantees that a single recompilation will result in correct code granted *opaque type* rules are followed. Unambiguous and correct code, post converter invocation is a result of carefully selected rules for *opaque type* implementation.

3.3 Annotation Processing Example

To demonstrate exactly what happens to all aspects of a regular Java class when it is annotated with the *@Opaque* annotation and put through the conversion utility, a more typical example from one of the early performance tests is illustrated in Figures 3.5 and 3.6. *TopLevel* classes are also shown to reveal the single code annotation that exists in classes declaring *opaque types*; and show what conversions take place therein. The examples also serve as a prelude to the last section of this chapter in helping to identify which features are vital to *opaque type* implementation in Java.

Figure 3.5 contains an *opaque type* object called *CkPiece*. *CkPiece* is to be represented internally as an integer type and therefore has a `rep` field typed `protected int`. The rest of the object is presented in the usual way with a `static` *New* method as well as three other `static` methods. All three of the other methods immediately interact with the *CkPiece* type or its representative type in some way.

Method *insc* returns a *CkPiece* and takes an argument of the underlying primitive

Figure 3.4: Opaque objects converted

```
public class BaseClass {
  protected int rep;
  private BaseClass(short r){
    rep = (int) r;
  }

  public static void operator(int bc, short modifier){
    ...
  }

  ...
}
```

```
public class ChildClassOne
    extends BaseClass {

  private ChildClassOne(short r){
    rep = (int) r;
  }

  ...
}
```

```
public class ChildClassTwo
    extends BaseClass {

  private ChildClassTwo(long r){
    rep = (long) r;
  }

  ...
}
```

```
public class ChildClassThree extends ChildClassTwo {

  private ChildClassThree(int r){
    rep = r;
  }

  public static int operator(int modifier, short cc){
    ...
  }

  ...
}
```

type *int*. While both *flip* and *online* take a *CkPiece* type argument and interact with its `rep` field directly.

The precise meaning and purpose of this implementation is unimportant and is presented merely to emphasize how `static` behaviours (methods) are maintained through *opaque type* conversion.

Figure 3.6 shows the same *CkPiece* object converted to serve as just a label for the *int* type. The method signatures and implementations are preserved with the ex-

Figure 3.5: Unmodified annotated object class

```
1  @Opaque(``int'')
2  public class CkPiece {
3    protected int rep;
4
5    private CkPiece(int p){
6      rep = p;
7    }
8
9    public static CkPiece insc(int a){
10     int tot = -1;
11     int msk = 1;
12
13     if (a < 0)  a = -a;
14     if (a < 32) {
15       for(int i = 0; i < a; i++){
16         tot = tot ^ msk;
17         msk = msk << a;
18       }
19     }
20     return new CkPiece(tot);
21   }
22
23   public static int flip(CkPiece p){
24     return (p.rep ^ -1);
25   }
26
27   public static boolean online(CkPiece p){
28     return (p.rep != 0);
29   }
30
31   public static CkPiece New(int p){
32     return new CkPiece(p);
33   }
34 }
```

ception of class name use. In their place now are variables of the underlying primitive type - *int*. User code declaring objects of type *CkPiece* also requires small modifications to adhere to the new method signatures and variable types. Basic `main` methods are presented next to show exactly the small changes made by the conversion utility.

The first noteworthy item to mention is the annotation on the 1^{st} line of Figure 3.7. *@Opaque("user")* indicates that the user intends to declare or make use of *opaque type*(s) inside the class. This signals the conversion utility to scan the class for such declarations and uses, and change them to the underlying primitive type that

Figure 3.6: Processed object class

```
1  public class CkPiece {
2    protected int rep;
3
4    private CkPiece(int p){
5      rep = p;
6    }
7
8    public static int insc(int a){
9      int tot = -1;
10     int msk = 1;
11
12     if (a < 0)  a = -a;
13     if (a < 32) {
14       for(int i = 0; i < a; i++){
15         tot = tot ^ msk;
16         msk = msk << a;
17       }
18     }
19     return tot;
20   }
21
22   public static int flip(int p){
23     return (p ^ -1);
24   }
25
26   public static boolean online(int p){
27     return (p != 0);
28   }
29
30   public static int New(int p){
31     return p;
32   }
33 }
```

corresponds to the *opaque type*. Line 8 is the only line of code that declares an *opaque* object, namely *CkPiece*. Line 11 initializes every element of the *CkPiece* array; and line 12 assigns it a new value based on the existing value stored in each variable.

The `for` loop on lines 15 and 16 prints out some of the *CkPiece* array values. It also applies the *flip* function to each value prior to printing. This is meant to illustrate the use of *opaque type* functions in compound statements without assignment to assure correct type is resolved successfully before and after conversion.

Figure 3.8 contains the converted *opaque* user class. The annotation is removed and the data type of the `pieces` array is changed to the corresponding underlying

Figure 3.7: Regular main class

```
1  @Opaque(``user'')
2  public class TopLevel {
3    public static void main(String[] args){
4      final int DATA_SIZE = 100;
5      final int MAX_INT   = 1 << 31;
6      final int MOD       = 16;
7      final int INCR      = DATA_SIZE / MOD;
8      CkPiece[] pieces = new CkPiece[DATA_SIZE];
9
10     for(int i = 0; i < DATA_SIZE; i++){
11       pieces[i] = CkPiece.New(Math.pow(2, i) % MAX_INT);
12       pieces[i] = CkPiece.insc(pieces[i] % MOD);
13     }
14
15     for(int i = INCR; i < DATA_SIZE; i += INCR)
16       System.out.println(CkPiece.flip(pieces[i]));
17   }
18 }
```

Figure 3.8: Converted main class

```
1  public class TopLevel {
2    public static void main(String[] args){
3      final int DATA_SIZE = 100;
4      final int MAX_INT   = 1 << 31;
5      final int MOD     = 16;
6      final int INCR      = DATA_SIZE / MOD;
7      int[] pieces = new int[DATA_SIZE];
8
9      for(int i = 0; i < DATA_SIZE; i++){
10       pieces[i] = CkPiece.New(Math.pow(2, i) % MAX_INT);
11       pieces[i] = CkPiece.insc(pieces % MOD);
12     }
13
14     for(int i = INCR; i < DATA_SIZE; i += INCR)
15       System.out.println(CkPiece.flip(pieces[i]));
16   }
17 }
```

primitive type of *CkPiece*, namely *int*. The conversion of the user classes is done following analysis of *opaque* classes in order to build a list of *opaque types* and their corresponding underyling types. This assures correctness of the resulting user classes where *opaque types* are used.

Notice that both *New* and *insc* methods of the *CkPiece* class that previously had the *CkPiece* return type (see Figure 3.5 have both been converted to return a value of type *int*. Therefore lines 10 and 11 of code in Figure 3.8 do not cause a type mismatch between the types of the values returned and the data type of the `pieces` array.

3.4 Other Considerations

Prior to finalizing the type rules (see Figure 3.1) required by *opaque types* and devising a scheme for implementing a conversion utility, some considerations had to be made regarding features and behaviours the new special types would have. The precise structure of the *opaque types* was not arrived at immediately. Several features were considered, implemented, and tested in order to judge their feasibility and necessity.

3.4.1 Opaque class identification

@Opaque class annotation serves as the first identifier of an *opaque type*. The type following this code annotation represents the underlying primitive type to be used in place of the *opaque type*. In the case where the `String` "user" is used, the class is recognized as containing code that uses *opaque type*(s).

Initially, *opaque types* were identified by the use of a new keyword, `opaque`, in the class declaration. This approach was practical in easily distinguishing regular classes from declarations which involved *opaque* objects. However, introduction of a new keyword meant that without a deep implementation declared *opaque* classes did not adhere to Java language standards immediately. In other words, use of the `opaque` keyword was not valid Java prior to invoking the code conversion utility. Using the `opaque` keyword also meant that the code analyzer had to spend more time looking through the *opaque* class in order to determine its underlying primitive type as it was not clearly stated until the `rep` field was encountered.

Figure 3.9: Using `opaque` keyword

```
1 public opaque class SampleClass {
2   protected
3   ...
4 }
```

Figure 3.9 shows an example of a class declaration using the `opaque` keyword. Due to `opaque` being an unrecognized keyword in the Java language, the object could not be compiled and run prior to conversion and removal of the keyword. This was a poor choice since the unconverted code serves as a snapshop of the development process and aids in debugging and maintenance. The unconverted code is also essential in preserving the language readability as, ideally, the programmer should only read and make changes to the code that has not yet undergone transformation.

3.4.2 Required methods

The current version of *opaque types* requires just a single method to be implemented - `static` TypeName *New*, where "TypeName" is the name of the *opaque type* being defined. Initial work lead to a standard of *opaque types* that had two mandatory methods. The methods were called `toTypeName` and `fromTypeName` and served the purpose of converting the underlying type **to** the object type and retrieving a value of the underlying primitive type **from** a given *opaque* object type respectively.

The inclusion of two methods with varying names in the *opaque type* requirements turned out to be unnecessary for two reasons. First, the `toTypeName` method basically accomplishes the same task as the already reserved Java keyword `new`, which initializes an object. Initializing an object of an *opaque type* requires a call to the `New` method to take a value of the underlying primitive type as an argument and call the `private` constructor of the object type. Therefore, the name `New` was standardized for the purpose of converting from underlying type to object type. Secondly, implementation of the `fromTypeName` method turned out to serve no purpose as it never comes up in applications of *opaque types*.

The goal of implementing *opaque types* in Java is to abstract the use of underlying primitive types by object names. The use of `fromTypeName` implies explicit "unwrapping" of the object type to achieve access to its representative type. This is unnecessary, as every use of the object type is already substituted for the corresponding primitively typed field by the conversion utility.

Examples of classes with the deprecated methods are shown in Figures 3.10 and 3.11

Figure 3.10: Using deprecated methods

```
@Opaque(``long'')
public class SampleClass {
  protected long rep;
  private SampleClass(long arg){
    rep = arg;
  }

  public static SampleClass toSampleClass(long arg){
    return new SampleClass(arg);
  }

  public static long fromSampleClass(SampleClass arg){
    return arg.rep;
  }
  ...
}
```

Figure 3.11: Using deprecated methods

```
@Opaque(``long'')
public class SampleClass {
  private long sample;
  private SampleClass(long arg){
    sample = arg;
  }

  public static SampleClass New(long arg){
    return new SampleClass(arg);
  }
  ...
}
```

3.4.3 Representation field modifiers and names

Settling on the underlying representation of the *opaque types* was a process that had to take into consideration the properties of *opaque* objects that needed to be preserved and features that made these objects appealing to use. The standard of using `protected type rep` field in every *opaque type* was preceeded by use of `public`, `private`, and `static` fields, with field names that reflected *opaque* object name more.

Figure 3.11 illustrates one of the approaches previously taken to declaring the underlying representation field. `private long sample` is the primitive type representation field of the *SampleClass* and uses a different visibility modifier (`private`) and a different variable name (`sample`) for its declaration.

The first and main issue with the shown declaration is the use of the `private` visibility modifier. This implies that the underlying type field is only visible from within the class. This posed some issues with inheritance properties of *opaque types*. Each subclass of the *opaque type* would have to provide its own representative field (contrary to the current implementation of inheriting the `protected` field), which gave more flexibility to *opaque types*. With every class providing its own representative underlying primitive type field regardless of the underlying type of its parent, relationships between *opaque types* connected by class hierarchy were not as precise. For example, an *opaque* class whose underlying representation type was *int* could be extended by another *opaque* class whose underlying representation type was free to be a primitive type that is structurally different from *int*, e.g. *long*. This introduced unneeded confusion and broke the essential "is a" relationship of Object Oriented Programming.

The second problem was having a field with a name more resembling that of the *opaque* class name. Although not a major issue, a non-standardized name for the representation field of the *opaque type* meant more variance in class implementation. Fixing the name at `rep` means code that is more clear and where the use and modification of of the representation field is obvious.

3.4.4 Handling inheritance

Various schemes were discussed and developed for handling *opaque type* inheritance prior to settling with the current approach. Some of these were more straight forward than others. For example, keeping all field and method visibility `public` meant that *opaque* objects would be completely transparent to subclasses and the rest of the world. However, this posed a problem with a core notion that needed to be maintained in *opaque type* implementation. The notion that sibling classes, although represented by the same underlying primitive type, would be distinct in their operations before and after conversion. Figures 3.12 and 3.13 illustrate why `public` level of access creates a problem in this area.

Both *ChildOne* and *ChildTwo* are subclasses of the *SampleClass* and share its underlying primitive type representation field `rep`. They also inherit the *dble* method

Figure 3.12: Inheritance problems

```
1  @Opaque(``int'')
2  public class SampleClass {
3    public int rep;
4    private BaseClass(int arg){
5      rep = arg;
6    }
7
8    public static SampleClass dble(int sc){
9      return SampleClass.New(sc * 2);
10   }
11
12   ...
13 }
```

```
1  @Opaque(``int'')
2  public class ChildOne extends SampleClass {
3    private ChildOne(int arg){
4      rep = r;
5    }
6
7    ...
8  }
```

```
1  @Opaque(``int'');
2  public class ChildTwo extends BaseClass {
3    private ChildTwo(int arg){
4      rep = r;
5    }
6
7    ...
8  }
```

implemented in the *SampleClass* object. The problem of keeping the `rep` field `public` arises in the following main class:

Although the variables declared on lines 6 and 7 of Figure 3.13 are technically of different types, the ability to publically access the representation field of each allows the statement on line 8 to be legal. Thus both variables `co` of type *ChildOne* and `ct` of type *ChildTwo* have the same underlying representation value. This introduces confusion after conversion takes place as the converter eliminates the variable object types. The resulting "user" code is shown:

In Figure 3.14, the value of `co` is being assigned to a previously type-distinct variable `ct` without any sort of casting or type-checking. The example demonstrates

Figure 3.13: `public` rep field declarations

```
1  @Opaque(``user'')
2  public class Main {
3    public static void main(String[] args){
4      SampleClass sc1 = SampleClass.New(2);
5      SampleClass sc2 = SampleClass.dble(sc1.rep);
6      ChildOne co = ChildOne.New(sc2.rep);
7      ChildTwo ct = ChildTwo.New(0);
8      ct.rep = co.rep;
9    }
10 }
```

Figure 3.14: `public` rep field declarations converted

```
1  @Opaque(``user'')
2  public class Main {
3    public static void main(String[] args){
4      int sc1 = SampleClass.New(2);
5      int sc2 = SampleClass.dble(sc1.rep);
6      int co = ChildOne.New(sc2.rep);
7      int ct = ChildTwo.New(0);
8      ct = co;
9    }
10 }
```

the created ambiguity in distinguishing sibling types (both are immediate subclasses of *SampleClass*) in the case where the underlying representation field is declared `public` and is therefore visible between the siblings. This nullifies our initial goal of making distinct *opaque* objects that could be represented by identical primitive types and was therefore reworked.

Method inheritance was one of the later issues to be worked out in imeplementing *opaque types* in Java. In general, `public` methods are visible to the world and therefore can be accessed by subclasses directly. However, one of the initial ideas was to implement a separate mechanism for *opaque type* object hierarchy, which would simulate regular inheritance with some restrictions. In part, method inheritance and visibility by subclasses would be assured by copying the inherited methods verbatim into the subclass implementation in order preserve visibility. Partial polymorphism support would be accomplished by simply implementing a similarly named method with different parameter types.

Figure 3.15: Method propagation through *private extension*

```
@Opaque(``int'')
public class SampleClass {
  protected int rep;
  private BaseClass(int arg){
    rep = arg;
  }
  public static int dble(int sc){
    return sc * 2;
  }
  ...
}
```

```
@Opaque(``int'')
public class ChildOne privately extends SampleClass {
  private ChildOne(int arg){
    rep = r;
  }
  public static int dble(int sc){
    return sc.rep * 2;
  }
  public static int dble(int sc, int v){
    return sc * v;
  }
  ...
}
```

Figure 3.15 illustrates the deprecated approach. The problems that arose when applying this technique for *opaque type* extension included the previously mentioned introduction of a new keyword (see Section 3.3.1), namely `privately` (to be used along with the `extends` keyword to identify "special" private extension). As well as, code size bloating as the class hierarchy grew. As all inherited methods were copied verbatim into their subclass implementation, deeply located *opaque* classes became unnecessarily big without adding much, if any, behaviour themselves. The redundancy of copying code with every extension was simply inefficient.

This chapter summarizes the theories behind our approach and reasons for the particular steps taken while designing *opaque types* in Java. The code samples highlight the major features of *opaque types* that were chosen for the reasons of clarity and efficiency. Justification for selection of features is also presented in the form of explanations following all the accepted features and rejection explanations following the previously considered constructs.

Chapter 4

Implementation

4.1 Implementation techniques

As a means of quickly and easily testing our type rules and the newly introduced *opaque* object types we have tackled the implementation step using a code preprocessor approach. The first preprocessor was written as a Unix shell script using the Unix stream editor utility: `sed`. Due to `sed`'s similarity to `perl`, an easy conversion was made into a `perl` script. The script was a compact way to apply a number of basic Java type substitutions and test the preliminaries of the approach without spending a long time on a complex conversion utility.

The shell script had to be invoked by hand via the command line and applied some essential transformations to the supplied *opaque* class. The script was able to determine the underlying type of the declared object and identify whether the object was a subclass of another named type. Based on this information textual substitutions were made to replace object name by the underlying type in all applicable constructors and methods. The file was then rewritten verbatim with the exception of the type changes.

The part of the script that did most of the transformation was only about 10 lines long and is shown in Figure 4.1.

Unfortunately, both the shell script and perl script implementations were insufficient for the full conversion, difficult to maintain, and hard to apply to programs with multiple classes. They were used for a short period of time to automate some of the code conversion and do quick prototyping during the infancy of the approach.

Figure 4.1: Conversion shell script

```
#!/bin/sh
TYPE=`sed -n -r 's/@Opaque\(\"[a-z]*\"\);/\1/p' <$1`
PARENT=`sed -n -r 's/([A-Z][a-zA-Z0-9]*) extends
          ([A-Z][a-zA-Z0-9]*) \{/\2/p' < $1`
sed -r -e '
s/@Opaque\(\"[a-z]*\"\)//
s/extends ([A-Z][a-zA-Z0-9]*)/extends \1/
s/protected [a-z]* [a-zA-Z][a-zA-Z0-9]*;/\/\/ \1/
/private [A-Z][a-zA-Z0-9]*\(.*\).*\{/,/\}/ d
s/static [A-Z][a-zA-Z0-9]*/static int/
s/return to[A-Z][a-zA-Z0-9]*\((.*)\)/return \1/
s/return new [A-Z][a-zA-Z0-9]*\((.*)\)/return \1/
s/\([A-Z][a-zA-Z0-9]* ([a-z]*)\)/(int \1)/
s/return ([a-z]*)\.[a-z]*/return \1/' <$1 >new
```

4.2 Preprocessor approach

The current version of the preprocessor is implemented in Java using utilities in the *java.util.regex* package to apply necessary changes to the source code. The preprocessor analyzes all the source files present in the specified source directories looking for the *@Opaque* annotation. It builds a list of all the source files that need to be converted in order to correctly process complicated class dependencies.

The *Opaque* annotation is a special Java class declared in the converter project. It is a particular class type for creating new Java code annotations and is declared according to the guidelines in [15]. The implementation turns to be very simple but slightly different from the traditional Java class declaration. Figure 4.2 depicts this class:

Figure 4.2: Opaque code annotation

```
public @interface Opaque {
  String value();
}
```

Each annotated class has two source file versions associated with it. The first version is as is, where the type rules of Figure 3.1 apply. The second, is the version stripped of most of its object "wrapper" qualities. That is, the object is now used according to its underlying primitive type representation. User classes containing

@Opaque("user") annotation are analyzed for instances of *opaque* object types. Declarations and instantiations are adjusted according to the primitive type associated with the object by consulting the list of *opaque types* built earlier.

Inheritance of *opaque* object types is supervised by the preprocessor. If the preprocessor finds a subclass of an *opaque type*, it checks the class for validity in satisfying the correct *opaque type* object extension. The criterion that must be met in order to maintain consistency with the introduced type rules are listed in Figure 4.3:

Figure 4.3: Subclass restrictions

- `protected` "primitive representation" field is absent
- all other *opaque* class restrictions apply

Methods inherited from the superclass are recognized by the compiler and do not cause conflict due to the restriction that the primitive representation of a subclass of an *opaque* object type is the same as that of the superclass. This also assures no discrepancy between converted methods of sub and superclasses.

The goal of the conversion utility is to be light-weight and efficient at analyzing the Java project structure and building a list of *opaque types* and their respective underlying representations. After creating an internal representation of the project, *opaque types* are converted to their underlying primitive representation in a single pass in accordance with the types found in the source tree. Construction of an internal representation in order to encapsulate all the interdependencies of a complex Java project is the first step the converter undertakes. Some of the fields of the *Converter*, as well as, its constructor are shown in Figure 4.4.

The key methods called in the *Converter* consturctor are *findSrcFiles* and *buildFileLists*. These methods, as the names suggest, locate all the source files relevant to the Java project and then classify them into three categories. All source files are divided into opaque files (`Vector`$<File>$ `opaqueFiles`), opaque user files (`Vector`$<File>$ `opaqueUserFiles`), and source files which are neither. The groups represent *opaque types* that have an underlying primitive representation, source files that make use of *opaque types*, and source files that are regular Java files respectively. The entire classification is done by looking for the *@Opaque* annotation in each class.

During the first pass through all the source files in order to classify them into three groups, the converter also identifies the primitive type representation of each

Figure 4.4: Conversion utility - identification of *opaque types*

```
1  public class Converter {
2    Vector<File>       srcFiles;
3    Vector<File>       opaqueFiles;
4    Vector<File>       opaqueUserFiles = new Vector<File>();
5    Vector<OpaqueType>  opaques;
6
7    ...
8
9    public Converter(File dir) {
10     srcFiles = new Vector<File>();
11     opaqueFiles = new Vector<File>();
12     opaques = new Vector<OpaqueType>();
13
14     findSrcFiles(dir);
15     buildFileLists();
16     ...
17   }
```

opaque type. It is stored as a `Vector` of `OpaqueTypes`, which are just an object with two `String` fields. One stores the *opaque type* name, and the other, its primitive representation type. Figure 4.5 shows this class.

Figure 4.5: Conversion utility - identification of *opaque types*

```
1  public class OpaqueType {
2    String name;
3    String type;
4    public OpaqueType(String n, String t){
5      name = n;
6      type = t;
7    }
8    public String toString(){
9      return name + "␣" + type;
10   }
11 }
```

The *findSrcFiles* and *buildFileLists* are instrumental in correctly restructuring the source files and into the aforementioned categories. They do this in a single pass, recursively traversing the directory structure of the Java project, if one exists. The helper method, *processDir*, analyzes a single directory within the project and isolates all files (files with extension ".java") to add to a common list of project source files.

This list of files then becomes the working domain for further analysis and conversion of *opaque types*.

The conversion tool utilizes *java.io* classes *BufferedReader*, *BufferedWriter*, *FileReader*, *FileWriter* for processing of the source files. Exceptions are caught and handled appropriately in order to catch errors that may occur during the IO-heavy operations. Final file classification is done on the basis of classes containing the *@Opaque* annotation and having the proper underlying primitive type specified there. Their implementation (as well as that of some helper methods) is shown in Figure 4.6.

The precise actions taken by the code converter can summarized as follows:

1. find all source (.java) files in project/source directories
2. identify annotated source files and build internal representation of the structure
3. convert all annotated opaque classes that have an underlying representation
4. convert all annotated classes that make use of opaque classes

The converter substitutes all uses of the declared object name and its field for variables of the underlying type and removes the `protected` field from the final version of the *opaque* class as it is no longer required.

The preprocessor is compiled into and distributed as a `jar` file that can be imported by any project or referenced on the *classpath*. Single file distribution makes it easy to develop applications that use *opaque types*. The `jar` file contains all the necessary classes for code annotation and conversion without the need for separate compilation every time changes are made to the application or project.

Figure 4.6: Conversion utility - classification of *opaque types*

```
1   private void findSrcFiles(File dir){
2     processDir(dir);
3     if (dir.isDirectory()) {
4       String[] children = dir.list();
5       for (int i=0; i<children.length; i++)
6         findSrcFiles(new File(dir, children[i]));
7     }
8   }
9   // PROCESS SINGLE DIRECTORY LOCATING .java FILES
10  private void processDir(File dir){
11    FilenameFilter filter = new FilenameFilter() {
12      public boolean accept(File d, String n) {
13        return n.endsWith({\tt".java{\tt"});
14      }
15    };
16    File[] fs = dir.listFiles(filter);
17    if (fs != null){
18      ArrayList<File> f = new ArrayList<File>(Arrays.asList(fs));
19      srcFiles.addAll(f);
20    }
21  }
22  private void buildFileLists(){
23    String line = null;
24    // find files that qualify for conversion
25    for (File f : srcFiles){
26      try {
27        br = new BufferedReader(new FileReader(f));
28        while (( line = br.readLine()) != null){
29          if (line.contains({\tt"}@Opaque{\tt"})){
30            String name = f.getName().replace(".java", "");
31            p = Pattern.compile("@Opaque[(]\"("+javaType")\"[)]");
32            m = p.matcher(line);
33            if (m.find()){
34              String type = m.group(1);
35              opaques.add(new OpaqueType(name, type));
36              opaqueFiles.add(f);
37            }
38            ...
39          }
40        br.close();
41      }
42      catch (IOException e) { e.printStackTrace(); }
43    }
44  }
```

4.3 Ant build script

Introducing the conversion utility for code analysis meant that the Java code could no longer be compiled as usual. The code had to be converted prior to compilation in order to substitute the use of *opaque type* variables for their respective primitive type representations. To automate the new building process an Ant build script was developed. The script can be executed directly from the Eclipse IDE provided the necessary plugin is installed. The default script calls the Java code preprocessor on all the current project and identifies all *opaque types* therein. The preprocessor applies the necessary conversion prior to compilation. A summary of the steps the Ant build

Figure 4.7: Ant build script steps

1. back up original source files
2. invoke converter on current project
3. compile newly converted files

script takes are outlined in Figure 4.7.

The first step, backing up of original source files, is done for the purposes of debugging and maintenance. The original code is left in tact for further development by the programmer as it is our goal to have the converted code remain in the background and be generally unnoticable. Backing up of the original code also allows for compilation without conversion and therefore allows the project to be run and verified prior to converting *opaque types* to their underlying primitive representation.

Invokation of the conversion utility in the second step of the Ant build script is self explanatory. The script uses a small configuration file called `conv.properties` to identify which main project directory should be analyzed. The structure of the configuration file is illustrated in Figure 4.8:

Figure 4.8: Ant build script properties file

```
project-name=MainProject
main-class=MainTopLevelClass
src.dir=src
```

Here, `project-name` serves as the placeholder for the Java project name/-main directory that is analyzed. If a main source file directory exists (such as in

Eclipse projects), it should be specified under the `src.dir` attribute, otherwise the `project-name` value is used as the main directory. Finally, the script allows the project to be run automatically following conversion and compilation, for this to succeed, the `main-class` attribute must be specified.

The Ant script invokes a compiler of the user's choice as the third step of the automatic building process. By default, the Sun's `javac` compiler is used but is easily changed as one of the script's properties. Files compiled are the newly created converted *opaque* type classes with *opaque types* replaced by their underlying representative types. Optionally, the original source files can be compiled using the same compiler and the two resulting projects can be run sequentially to check the correctness of the developed *opaque types*.

Therefore the build script is able to run the compilation process twice, one time without any preprocessing and once with it. The user can then choose which project to run, the one with *opaque types* converted or the regular untouched project, or run them both. The two versions of source files are kept as identical as possible for debugging purposes. If the Java compiler encounters an error during compilation and indicates the line number at which this error occurred, the converter works in such a way as to guarantee that the number is correct for both versions of the source file.

Most of the script is shown in Figure 4.9 at the end of this chapter as an example of what the user might see if he or she wanted to change some properties of the build script to suit their needs.

4.4 Native implementation

The ultimate goal of our method is to provide the developer with robust static type checking that exists in Java at the Object level and efficient generated code that uses primitive types wherever possible. Combining powerful static type checking techniques that Java offers with utilization of primitive types should yield code that is fast and correct. Introduction of *opaque type* extension (or *opaque type* inheritance) allows use of well developed and understood Object Oriented Design principles to be applied without loss of efficiency (when Object representation permits, i.e., Object can really just be an *int*).

The intention is to introduce a process for making the currently necessary code adjustments either completely automatic or altogether unnecessary. The former can be accomplished by augmenting a high level Java code representation (such as the parse trees used by the Eclipse IDE) to be able to recognize *opaque types* as valid Java code. The conversion would then take place at the intermediate code level instead of applying conversions immediately to the code as it is currently done. This would solidify the abstraction of the approach and ease the use of *opaque types* in Java by indirectly implementing them into the language standard. The solution could consist of a modified JFlex grammar that will make the necessary substitutions and additions to the code prior to passing it to the parser generator. Integrating the features into an IDE such as Eclipse for Java should make the process simple enough to encourage many developers to use our *opaque* object types for performance sensitive applications.

Eliminating code conversion completely would mean integrating of *opaque types* directly into the Java language standard. A possible implementation of this, without official adoption by the language standard, consists of introducing a special flag for the command line Sun Java compiler. The code would then be compiled as usual with the "javac" command and when the flag is given, the specified classes will be transformed according to the rules outlined in Chapter 3. A more general implementation such as this one will further encourage use of the proposed technique by accommodating a greater variety of development environments.

Both of the routes require more careful consideration and a possible reshaping of some rules described in Chapter 3. As the integration into an existing compiler is complex, concrete rules are needed for *opaque* object type creation which may change over the course of future work. Several possible solutions need to be evaluated and tested extensively in order to justify fundamentally modifying implementation of the proposed approach. Some uncertainties have various solutions that at first

glance seem equally viable. Research and testing should allow us to discern which alternatives will yield the best results for the solution.

The benefits a native implementation will introduce are numerous. Firstly, ease of use of *opaque types* should encourage developers to utilize their efficiency and safety. Secondly, integration into an existing compiler will allow for further optimization techniques by combining implemented optimization methods with those that are possible only in the case of *opaque types* (i.e. eventual primitive types). Such optimizations may shed more light on how to improve compilation and optimization of structures similar to C-type structs as a byproduct.

Finally, introducing this approach into the world of mainstream compilers will greatly increase its exposure allowing further improvements and fixes by other researchers and developers working with the Java programming language. As the techniques are further developed, the approach suggested here must become more robust and versatile, able to handle a variety of situations that may occur as Java is used for development. The foundation for success of this development undoubtedly lies in a concrete native implementation that increases the method's availability across different audiences. The subject of native implementaion is touched on in more detail in Chapter 6.

Figure 4.9: Ant build script

```
<project basedir="." default="main">
  <property file="conv.properties"/>
  <property name="converter-class" value="MainConverter"/>
  <property name="converter-jar" value="converter.jar"/>

  <property name="fastsrc.dir" value="../${project-name}/fastsrc"/>
  <property name="fastbin.dir" value="../${project-name}/fastbin"/>

  <!-- Clean targets -->
  <target name="clean">
      <delete dir="${fastbin.dir}"/>
    <delete dir="${fastsrc.dir}"/>
  </target>

  <!-- Convert targets -->
  <target name="convert-compile">

    <!-- Copy source -->
    <mkdir dir="${fastsrc.dir}"/>
    <copy todir="${fastsrc.dir}" overwrite="true">
      <fileset dir="../${project-name}/${src.dir}"/>
    </copy>

    <!-- Convert -->
    <java jar="${converter-jar}" fork="true"/>

    <!-- Fast -->
    <!-- Compile -->
    <mkdir dir="${fastbin.dir}"/>
    <javac srcdir="${fastsrc.dir}" destdir="${fastbin.dir}">
      <classpath>
        <pathelement path="${fastsrc.dir}"/>
        <pathelement path="${fastbin.dir}"/>
      </classpath>
    </javac>
  </target>

  <!-- Run targets -->
    <target name="run">
      <java classname="${main-class}">
        <classpath>
        <pathelement path="${fastsrc.dir}"/>
          <pathelement path="${fastbin.dir}"/>
        </classpath>
      </java>
    </target>

    <target name="main" depends="clean,convert-compile,run"/>
</project>
```

Chapter 5

Performance

5.1 Performance benchmarks

One of the main goals for introducing *opaque* object types is to improve application performance. Here, we define the overall application performance by the time it takes the program to execute, and memory consumed during its execution. We compare performance of Java code using regular objects and code which has been converted to use *opaque* objects. The applications accomplish identical tasks and have minimal implementation differences aside from the use of *opaque types*. The test cases range from simple classes implementing only a few methods with shallow class hierarchy to classes with a large internal representation (e.g. a large integer array), several constructors, and a large number of methods. All types of tests were run enough times until the average results stabilized and consequent runs did not create significant variation in the data. The tests were executed with varying total data size in order to maximize the accuracy of results and demonstate generally applicable benefits. The types of tests were chosen to reflect varying uses and applications a developer may encounter when writing Java code for a typical project.

Implementations have been tested on varying platforms (including different CPUs, RAM, and operating system types). The platform used for a particular test is mentioned in each table depicting the results. This was done mostly due to different availability of computing environments during testing, as tests were developed throughout the evolution process of *opaque types*. However, running of experiments on different platforms has also given us a an opportunity to look at the variance in underlying software and hardware that affects the performance of Java applications using *opaque types*.

Execution time and memory use were measured using built-in Java tools for deter-

mining system time - `System.currentTimeMillis()`, and tools for determining how memory is currently used by the Java Virtual Machine - `Runtime` methods named `totalMemory()` and `freeMemory()`. All tests measuring memory use were carefully designed to avoid involuntary garbage collection and execution time tests were averaged to account for varying CPU load during the experiments.

5.2 Direct testing of compound statements, declarations, instantiations

We conducted several kinds of tests comparing regular and *opaque* objects. The goal of the tests is to determine whether objects that had undergone the transformation to the underlying primitive type performed significantly faster and took up considerably less memory space than regular objects. Usage tests varied by type of declaration and calling convention for the objects. For example, some tests measured only how regular objects perform during declaration and instantiation versus *opaque* objects, while others used the objects in longer compound statements without explicit declaration or instantiation. A compound statement may involve passing function values directly as arguments to another function or object constructor.

The contrast between the two compared sets of statements is illustrated in Figure 5.1. The first set of statements involves using a regular object method and passing the result to another method inside of a longer compound statement. The second set, shows a similar outer method being applied to an argument which is computed by a `public static` method of an *opaque type*, whose underlying primitive type is *int*.

Figure 5.1: Instantiation followed by compound statement

```
1 for (int i = 0; i < DATA_SIZE; i++){
2   objInstance[i] = new RegObject(i);
3   testMethod(objInstance[i].objMethod());
4 }
```

```
1 for (int i = 0; i < DATA_SIZE; i++){
2   primitiveInstance[i] = OpaqueObject.New(i);
3   testMethod(OpaqueObject.method(primitiveInstance[i]));
4 }
```

The first difference which can be noted between code using regular objects and

that using converted *opaque* classes is the constrasting instantiation of the variables. In the case of a regular object, `objInstance[i]` is initialized using the `RegObject`'s constructor. The `primitiveInstance[i]` variable, post conversion, bares the underlying primitive type of `OpaqueObject`, namely, *int*. The call to *New* returns a value of type *int* to be assigned to `primitiveInstance[i]`. However, due to primitive types being initialized to zero by default, this initialization is unnecessary.

Java utilities for measuring memory use are not as sensitive as we would have ideally liked, therefore, for small data sizes, the difference in memory use is not apparent. The tools measure memory use on the heap and therefore fail altogether for sufficiently small sample sizes of primitively typed variables. For small total data size regular objects are still allocated on the heap while *opaque* objects reside completely on the stack. This is due to a much smaller foot print that *opaque* objects (sized no bigger than the largest primitive type) have in memory. In fact, because *opaque* objects are compiled to primitive types, they are given preference to reside on the stack just like ordinary primitive types.

This result is summarized in Table 5.1 and shown on plot Figures 5.2 and 5.3. Data allocation on the heap is characteristic of traditional object types during both, conventional declaration as well as use in compound statements. In contrast, use of the stack is prominent for *opaque* objects as expected. In fact, for compound statements, *opaque* objects are not allocated on the heap at all, even for large data sizes. Meanwhile, heap space use begins to be noticable enough once the total size of regular objects reaches about 500KB.

Worth noting is the fact that for large data sizes (1,000,000 and higher), it was necessary to add the `-Xmx` *`N`* flag to the application launch command where *`N`* is the maximum Java heap size the JVM is allowed to use in megabytes.

Declaration and instantiation of regular objects versus *opaque* objects yielded expected results.[9] Due to *opaque* objects being converted to primitive types prior to compilation, the memory taken by them is identical to that of their representative type. For example, Table 5.2 shows that declaring two arrays of *opaque* objects both represented by the primitive type *int*, is similar to just declaring two integer

Table 5.1: Instatiation and compound statements
Platform: Intel C2D E4600 @ 2.4GHz, 2GB RAM, Ubuntu Linux 9.04

	Regular object call		**Opaque object call**		**Improvement ratio**	
	Time(ms)	**Memory(bytes)**	**Time(ms)**	**Memory(bytes)**	**Time**	**Memory**
Array size:						
100	2	0	2	0	1.0	1.0
1,000	6	36,848	3	0	2.0	n/a
10,000	13	334,112	4	0	3.3	n/a
20,000	14	660,600	4	0	3.5	n/a
50,000	19	1,981,816	6	0	3.2	n/a
100,000	28	3,303,000	10	0	2.8	n/a
1,000,000	171	8,775,680	39	0	4.4	n/a
10,000,000	501	22,909,496	265	0	1.9	n/a

Figure 5.2: Compound statement memory use

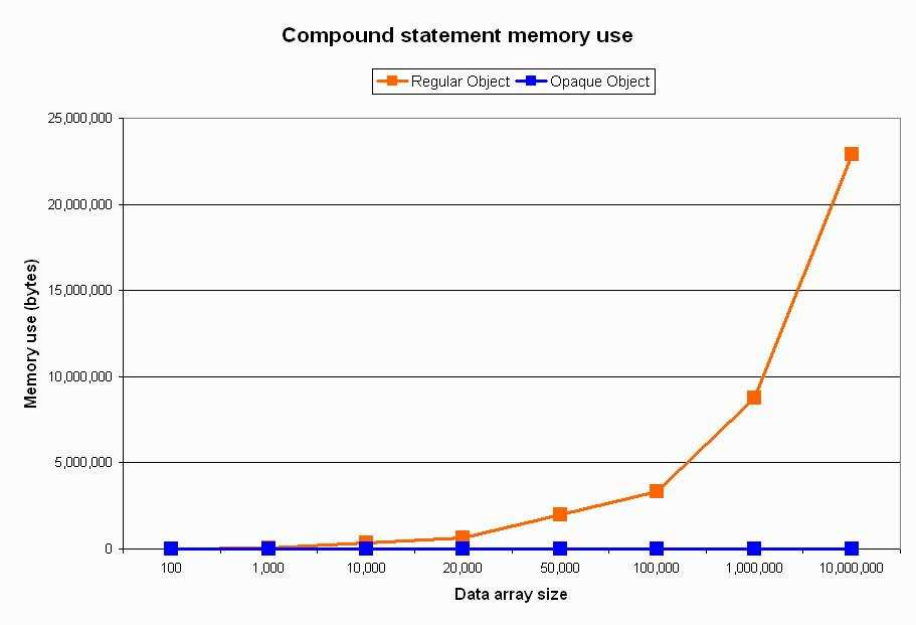
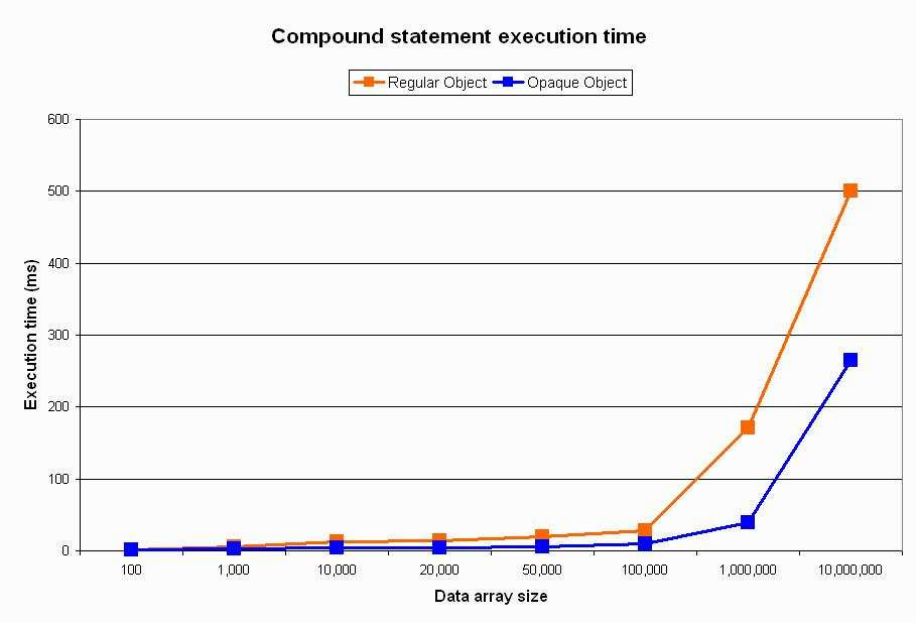


Figure 5.3: Compound statement execution time



arrays. The memory space taken up by each object is exactly the same as an *int* type (i.e. 4 bytes). Unconverted object types with the same functionality as the corresponding *opaque* types took roughly 5 times more space for straight declaration and instantiation statements. Results of Table 5.2 are plotted on 5.4 and 5.5.

Figure 5.4: Declaration and instantiation memory use

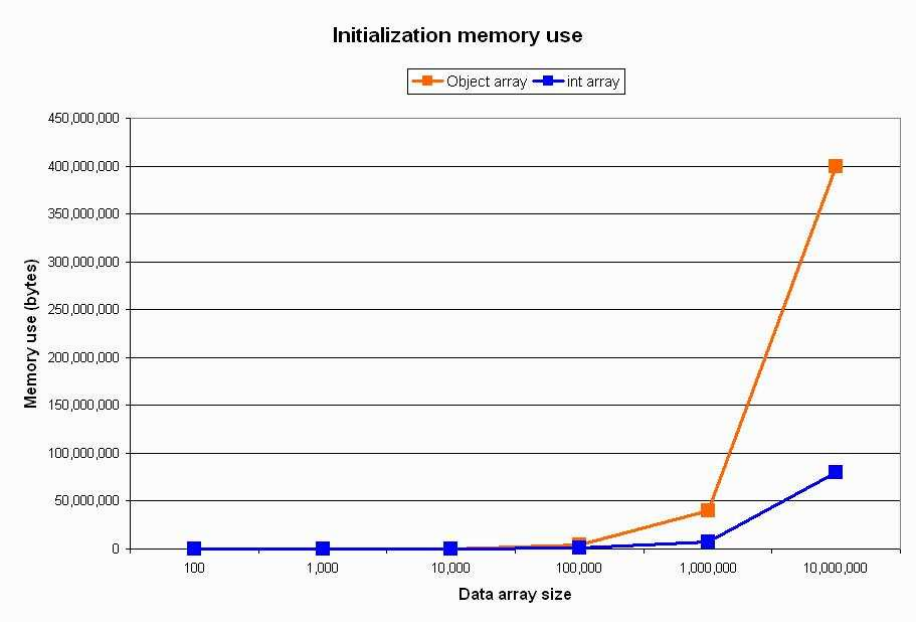
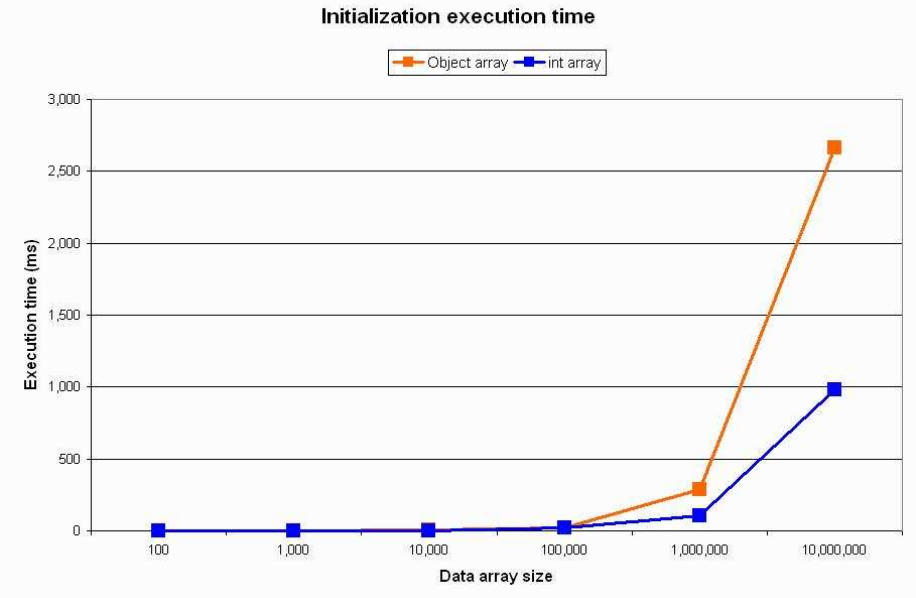


Figure 5.5: Declaration and instantiation execution time



Once again, larger data sizes required manually increasing the allowed Java heap size by using the `-Xmx` *N* when launching the tests.

Table 5.2: Declaration and instantiation
Platform: Intel C2D E4600 @ 2.4GHz, 2GB RAM, Ubuntu Linux 9.04

	Object array		**int array**		**Improvement ratio**	
	Time(ms)	**Memory(bytes)**	**Time(ms)**	**Memory(bytes)**	**Time**	**Memory**
Array size:						
100	1	0	0	0	n/a	1.0
1,000	2	36,456	1	0	2.0	n/a
10,000	5	337,744	3	0	1.7	n/a
100,000	23	3,962,848	19	800,032	1.2	5.0
1,000,000	289	39,921,800	104	7,778,096	2.8	5.1
10,000,000	2,667	399,972,016	981	80,000,032	2.7	5.0

5.3 *Opaque types* with a more complex representation

Similarly to the *opaque types* used through this work, it is possible to represent object types by single primitively typed array fields of fixed size. For example, an *opaque type* object may be represented by 256 bits, or an array of size 4 of type `long[]`. The next set of tests deals with objects represented by different sized arrays of primitively typed variables. The tests attempt to use the same, previously shown, metrics in measuring execution speed and memory use. Implementation of the actual accomplished operation is kept as identical as possible to avoid performance differences due to algorithm efficiency. This assures that we compare directly the speed and size of regular objects versus *opaque* objects without introducing unnecessary bias.

Figure 5.6 illustrates an *OpaqueObject* represented by the `long[]` type and a *RegObject* that has a field of type `long[]`. Both objects have the similarly implemented method called *setBit*. Method *setBit* takes an argument of type `int` that corresponds to the bit number that must be set to 1 in the internal representation of *OpaqueObject* `o` or the field of the *RegObject* with 0 being the least significant bit. Imagine arranging either the internal `long[]` representation of *OpaqueObject* or the field of *RegObject* as sets of back-to-back 64 bit sets (each set represented by a `long` type value) where significance of the bits increases with the array index of the respective field. Thus operation *setBit* is potentially able to turn on a single bit in a bit set of size over 2,000,000,000. For the test, however, we limit the size of the `long` array to 4.

Results comparing performance of these two objects are presented in Table 5.3 and plotted on 5.7 and 5.8.

Test results shown in Table 5.3 are very important as they show that extending *opaque types* to primitive type array representation preserves lower memory use and faster execution speed, to some degree. This is a powerful augmentation of the *opaque* object approach as it allows representations which are much more complex to be applied with the same ease.

Table 5.3: Instantiation of complex objects
Platform: Intel C2Q Q6600 @ 3.3GHz, 4GB RAM, Windows XP SP3

	Regular object array		**Opaque object array**		**Improvement ratio**	
	Time(ms)	**Memory(bytes)**	**Time(ms)**	**Memory(bytes)**	**Time**	**Memory**
Array size:						
100	0	18,400	0	0	1.0	n/a
1,000	9	73,600	0	0	n/a	n/a
10,000	15	682,480	6	0	2.5	n/a
100,000	47	6,732,848	23	1,159,200	2.0	5.8
1,000,000	203	66,954,048	110	14,722,616	1.8	4.5
10,000,000	2015	679,930,056	1181	129,657,024	1.7	5.2

Figure 5.6: Regular and *opaque* objects with array typed fields

```
@Opaque(``long[]'')
public class MyOpaqueObject {
  protected long[] rep;
  private OpaqueObject(long[] arg){
    rep = new long[arg.length];
    for (int i = 0; i < arg.length; i++)
      rep[i] = arg[i];
  }
  public static OpaqueObject New(long[] arg){
    return new OpaqueObject(arg);
  }
  public static OpaqueObject setBit(OpaqueObject o, int i){
    long mask = (long) (1 << (i % 64));
    o.rep[i / 64] |= mask;
    return o;
  }
  ...
}
```

```
public class RegObject {
  private long[] rep;
  public RegObject(long[] arg){
    rep = new long[arg.length];
    for (int i = 0; i < arg.length; i++)
      rep[i] = arg[i];
  }
  public void setBit(int i){
    long mask = (long) (1 << (i % 64));
    rep[i / 64] |= mask;
  }
  ...
}
```

Analyzing the data in Table 5.3 shows that the performance gains are somewhat diminished as *opaque type* underlying primitive representation becomes more complex. The diminished performance improvement is due to the way Java handles primitively typed arrays. In every representation, an array is automatically converted to an object in Java, thus deteriorating execution speed and increasing memory use. Nevertheless, by using *opaque* objects in place of regular objects, the conversion utility takes off a "layer" in the object hierarchy, replacing it by a primitive type. This accounts for the, somewhat smaller, performance gains. *Opaque types* still outperform regular object types in both execution speed (roughly by a factor of 2.0) and in memory use

Figure 5.7: Instantiation of complex objects memory use

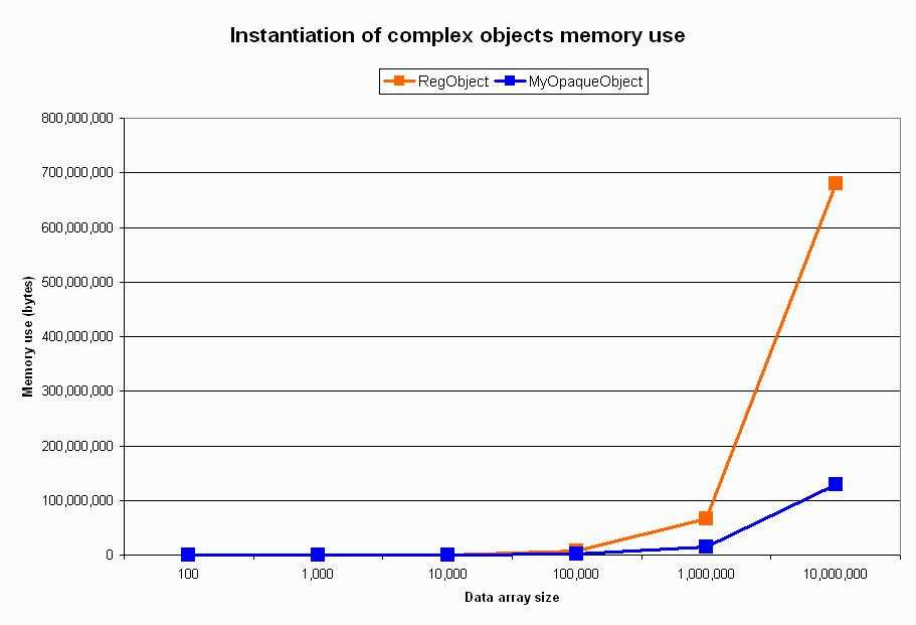
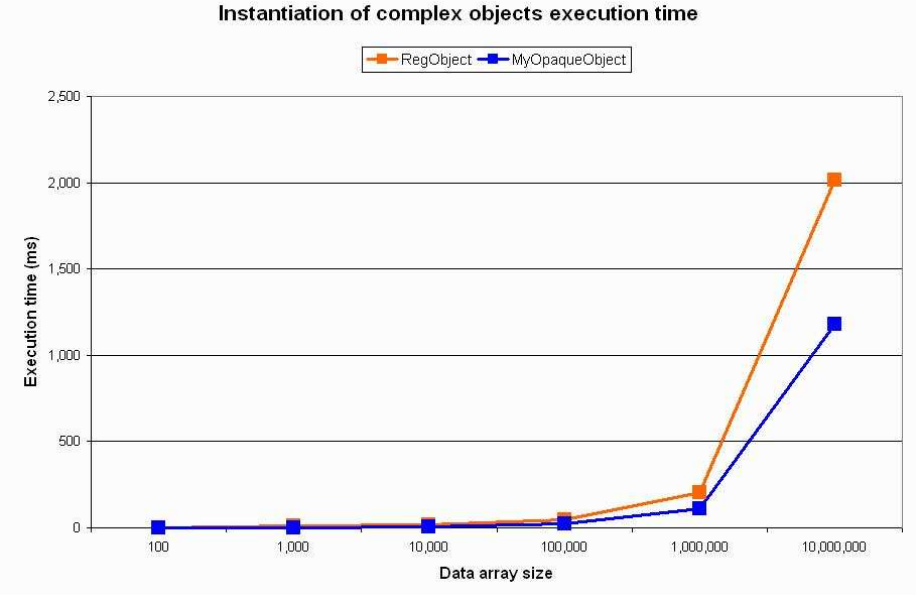


Figure 5.8: Instantiation of complex objects execution time



(roughly by a factor of 5). As program size and memory footprint increase, these gains become more significant in real world applications.

5.4 A more involved experiment

The next set of performance comparison tests consists of two implementations of a typical Chess board. A conventional chess board is an 8 by 8 board with black and white squares with at most 32 pieces on the board at any given time. The game of chess is rich in theory and has been used for computing performance measurement for several decades. The most prominent approach to implementing chess playing computer programs can be classified as a combination of brute force and heuristic algorithms. As the game is turn based, there is a finite number of consequent moves for any possible position. By constructing a game tree, and analyzing each path in this tree (representing a sequence of moves) heuristically, today's chess playing programs are able to play at a very high level. Since the number of possibilities at each level in the tree grows nearly exponentially, computational speed becomes of high importance and the measure of quality of play is reduced to the number of tree paths and path depth that is analyzed successfully in a limited amount of time.

The first implementation is a traditional approach to actualizing chess using an Object-Oriented language. It makes use of Java *enum* type and has clear distinctions between elements involved in the game of chess. The implementation consists of four main classes: *ChessPosition*, *ChessSquare*, *ChessPiece*, and *ChessMove*. Intuitively, the class *ChessPiece* represents any chess piece available to be placed on the board. *ChessSquare* represents a square on the chess board identified by two cartesian style coordinates (but called by their chess names: *file* [a letter from *a* to *h*], and *rank* [a number from *1* to *8*]) and a ChessPiece that possibly occupies that square. A *ChessPosition* consists of an array of *ChessSquare*'s of size 64 - representing a full chess board and the pieces present at any given time during the game. Finally, *ChessMove* assists in implementing a single chess move (or, more accurately, a ply - i.e. a move completed by a single player) possible in a game of chess. The move is represented simply by an initial and a destination square and all validity checking, as well as, move application is done given a *ChessPosition*.

Shown next are some relevant parts of the traditional implementation. These code samples help illustrate the difference between traditional and *opaque type* style implementations. The explainations that follow attempt to highlight parts of the code where the *opaque type* approach gains its performance benefits without losing approach clarity and abstraction. The implementation is simplified and is meant to minimally demonstrate how a realistic application can be represented in traditional Java versus Java augmented with *opaque types*.

Figure 5.9: Traditional implentation: `ChessSquare`

```
1  public class ChessSquare {
2    char       file;
3    int        rank;
4    ChessPiece piece;
5
6    public ChessSquare(char f, int r, ChessPiece p){
7      file = f;
8      rank = r;
9      piece = p;
10   }
11
12   public ChessSquare(char f, int r, ChessPosition cp){
13     file = f;
14     rank = r;
15     piece = cp.board[getLocation(file, rank) - 1].piece;
16   }
17
18   public int getLocation(){
19     int col = (int)(file - 'a') + 1;
20     int row = 8 - rank;
21     return row * 8 + col;
22   }
23
24   public static int getLocation(char f, int r){
25     int col = (int)(f - 'a') + 1;
26     int row = 8 - r;
27     return row * 8 + col;
28   }
29 }
```

The *ChessSquare* class provides an intuitive implementation of a chess square. Each cell is labeled by a file and rank and has a *ChessPiece* associated with it. In the case of an empty square, the *ChessPiece* field is set to **null**. The *ChessPiece* class provides a basic implementation of the various chess pieces, which can be encountered in the game. In total, there are 12 possible pieces (6 different pieces times 2 different colours). These are implemented using Java `enum` types, thus adding two fields to the *ChessPiece* type, namely *Color* and *Type* members.

ChessSquare provides a utility method *getLocation* for converting from the conventional chess system of files and ranks to an integer corresponding to the array index in the board's internal representation. In this case, the convention is to start at 1 (corresponding to square *a8* on the board) and end with 64 (corresponding to square *h1*).

Figure 5.10: Traditional implentation: `ChessPosition`

```
1  public class ChessPosition {
2    ChessSquare[] board;
3    /* Main constructor:
4     * Initialize the board with
5     * 64 new (empty) ChessSquares
6    */
7    public ChessPosition(){
8      board = new ChessSquare[64];
9      for (int i = 0; i < board.length; i++){
10       char file = (char) (i % 8 + 'a');
11       int rank = 8 - i / 8;
12       board[i] = new ChessSquare(
13             file, rank, (ChessPiece)null);
14     }
15
16     // Initialize board to starting position
17     ...
18   }
19
20   ...
21   public boolean applyMove(ChessMove cm){
22     if(!cm.isValid(this))
23       return false;
24
25     // initial square becomes empty
26     int ind = cm.init.getLocation() - 1;
27     Color c;
28     if (((int) (cm.init.file - 'a')
29             + cm.init.rank) % 2 == 0)
30       c = Color.WHITE;
31     else
32       c = Color.BLACK;
33     board[ind].piece = new ChessPiece(
34                   c, Type.EMPTY);
35
36     // final square is occupied by piece
37     ind = cm.fin.getLocation() - 1;
38     board[ind].piece = cm.init.piece;
39
40     // move succeeded
41     return true;
42   }
43 }
```

The *ChessPosition* class, shown in 5.10, serves as a representation of a chess position at any point in time during the game. It includes all 64 *ChessSquare*'s and provides methods that correspond to possible events that can happen during transi-

tion from one position to the next. The essential code that supplies this functionality is located in the *applyMove* method. Given a *ChessMove*, a decision is made whether the move is legal for the current position and if it is; the move is carried out. Some implementation details of this class are omitted here.

The second implementation, using *opaque-typed* code, is an attempt to represent the game of chess in a more compact manner. In this approach we take advantage of the fact that there are only 12 different pieces present at any time during the game (*pawn*, *rook*, *knight*, *bishop*, *queen*, *king* of black and white colours). Since a square on the chess board may also be unoccupied, we can add the "empty square" as a possibility to the already present 12 chess pieces, bringing the total to 13. 13 patterns may be represented in binary as four bits. There are 64 squares on the chess board, therefore the total number of bits needed to represent a board in binary is 256. Although this is not the best we can do, it is easier to work with this representation and it is sufficient to demonstrate a low level implementation abstracted by Java classes.

The goal of this implementation is to provide identical functionality to the traditional approach, leaving calling conventions and ease of use in tact, and improve code performance through use of primitive types. To further optimize the low-level implementation, the use of *opaque types* eliminates "object-wrapper" code from the primitively-typed fields prior to compliation. This effectively transforms the end application into a series of bit shuffling operations that execute quickly and leave a small footprint in memory.

Figures 5.11 and 5.12 show partial implementations of the *ChessBoard* and *ChessGame opaque* classes. The algorithms developed in both *opaque* objects rely heavily on the underlying primitive representation of a *ChessBoard*. The `long[]` type field contained in *ChessBoard* is conventionally initialized to size 4 when the object's *New* method (omitted in the figure) is called, thus representing an entire chess position in just 256 bits (64 squares, at 4 bits each). Despite this low, bit-level, representation, all operations provided by the *ChessBoard* and *ChessGame* are intuitive and straight forward to use. The actions which can be applied to these objects correspond to the operations implemented by the traditional approach. This is accomplished via abstracting low level bit-shifting operations behind `static` methods in the *opaque* classes and allows code using these objects to be written in the traditional Java style. *ChessBoard* and *ChessGame* objects are declared and analyed normally while maintaining their primitive type representation after the code conversion utility is invoked.

Figure 5.11: *Opaque-typed* representation: `ChessBoard`

```
@Opaque(``long[]'')
public class ChessBoard {
  static final long EMPTY     = 0;
  static final long W_PAWN    = 1;
  static final long W_KNIGHT  = 2;
  static final long W_BISHOP  = 3;
  static final long W_ROOK    = 4;
  static final long W_QUEEN   = 5;
  static final long W_KING    = 6;
  static final long B_PAWN    = 8;
  static final long B_KNIGHT  = 9;
  static final long B_BISHOP  = 10;
  static final long B_ROOK    = 11;
  static final long B_QUEEN   = 12;
  static final long B_KING    = 13;

  protected long[] rep;

  private ChessBoard(){
    rep = new long[4];
    ChessBoard.fillStarting(this);
  }

  public static void fillStarting(ChessBoard b){
    // fill black pieces
    b.rep[0] |= B_ROOK | (B_KNIGHT << 4)
             | (B_BISHOP << 8) | (B_QUEEN << 12);
    b.rep[0] |= (B_KING << 16) | (B_BISHOP << 20)
             | (B_KNIGHT << 24) | (B_ROOK << 28);
    for (long i = 32; i < 64; i += 4)
      b.rep[0] |= (B_PAWN << i);

    // fill empty squares
    b.rep[1] = 0L;
    b.rep[2] = 0L;

    // fill white pieces
    for (long i = 0; i < 32; i += 4)
      b.rep[3] |= (W_PAWN << i);
    b.rep[3] |= (W_ROOK << 32) | (W_KNIGHT << 36)
             | (W_BISHOP << 40) | (W_QUEEN << 44);
    b.rep[3] |= (W_KING << 48) | (W_BISHOP << 52)
             | (W_KNIGHT << 56) | (W_ROOK << 60);
  }
  ...
}
```

Figure 5.12: *Opaque-typed* representation: `ChessGame`

```
@Opaque(``long[]'')
public class ChessGame {
  protected long[] rep;

  public static ChessBoard applyMove(ChessBoard b, short move){
    int init = move & 0x3F;
    int fin = move & 0xFC0;
    long[] board = b.rep;

    long mask = 15L;
    int initInd = init / 16;
    int finInd = fin / 16;
    long initPiece = board[initInd]
          & (mask << ((init % 16) * 4));
    // zero destination square
    board[finInd] &= ~(mask << ((fin % 16) * 4));
    // zero initial square
    board[initInd] &= ~(mask << ((init % 16) * 4));
    if (initInd == finInd) {
      if (init > fin)
        initPiece <<= (init - fin);
      else if (init < fin)
        initPiece >>= (fin - init);
    }
    else {
      initPiece >>= ((init % 16) * 4);
      initPiece <<= ((fin % 16) * 4);
    }
    board[finInd] |= initPiece;
    b.rep = board;
    return b;
  }
  ...
}
```

The *ChessGame* class in Figure 5.12 is also represented by the primitive `long` type array. The class plays the role of a sequence of consecutive moves in a single chess game. Therefore it implements a particular important method, *applyMove*. The method takes a *ChessBoard* and a `short move` as parameters and attempts to apply the said move to the given *ChessBoard* representation. Each chess move is represented by a `short` type value where the bits signify the initial and final squares of the move. This is discussed further when move generation for the two types of implementations is compared.

Given this implementation we can carry out several important performance tests

comparing traditional and *opaque-typed* approaches. Once again, the main criteria tested are execution time and memory usage. The tests consist of creating and manipulating a large number of chess positions. This simulates realistic operations that happen during a chess program's execution. During the engine's "thinking" phase, new positions in the game tree are constantly created and evaluated against some set of heuristics and are then discarded or saved accordingly. The speeds of retrieval, manipulation, and storage directly affect the overall performance of the application.

Performance is compared in three stages. First, the regular implementation of *ChessPosition* is used to initialize a number of positions. Some moves are applied to the positions to simulate varying states of the game. The time taken to allocate the required memory for these positions is recorded and averaged over several runs. Memory use is also averaged to avoid skewed test results due to differences in representation of equally likely board positions.

A similar set of tests is then executed on the *opaque-typed* code using the *ChessBoard* class. The tests are performed without invoking the code conversion utility in order to accurately measure gains achieved simply by moving to an alternative internal representation of the game board. Measuring the execution speed and memory use of the alternate representation also allows us to gauge the difference that the code conversion utility makes. This particular measure is important because it comes at no cost to the developer. The code requires no changes prior to invoking the conversion utility as long as it was properly annotated from the start. Results are shown in Table 5.4 and plotted on 5.13 and 5.14.

These results are very impressive. They indicate that the structure of *opaque types* alone plays an important role in performance improvements they allow. Execution time is decreased by a factor of about **5-10** when *opaque ChessBoard* class is used instead of the regular implementation of of *ChessPosition* even prior to converting *ChessBoard* to its underlying `long[]` type. Meanwhile, memory use is decreased by a factor of **3** demonstrating how much difference `static` methods used by the *opaque ChessBoard* make.

Next we compare the same *ChessPosition* implementation with the converted implementation of *opaque ChessBoard*. Therefore, Table 5.5 essentially shows the

Table 5.4: Initialization and storage of chess positions
Platform: Intel C2Q Q6600 @ 3.3GHz, 4GB RAM, Windows XP SP3

	Regular ChessPosition array		**Opaque ChessBoard array (unconverted)**		**Improvement ratio**	
	Time(ms)	**Memory(bytes)**	**Time(ms)**	**Memory(bytes)**	**Time**	**Memory**
Array size:						
100	0	224,184	0	112,640	1.0	2.0
1,000	14	2,579,504	3	892,108	4.7	2.9
10,000	35	28,374,544	7	9,242,336	5.0	3.1
100,000	929	284,956,880	83	91,962,292	11.2	3.1
500,000	4691	1,423,907,944	489	443,360,416	9.6	3.2

Figure 5.13: Initialization and storage of chess positions memory use

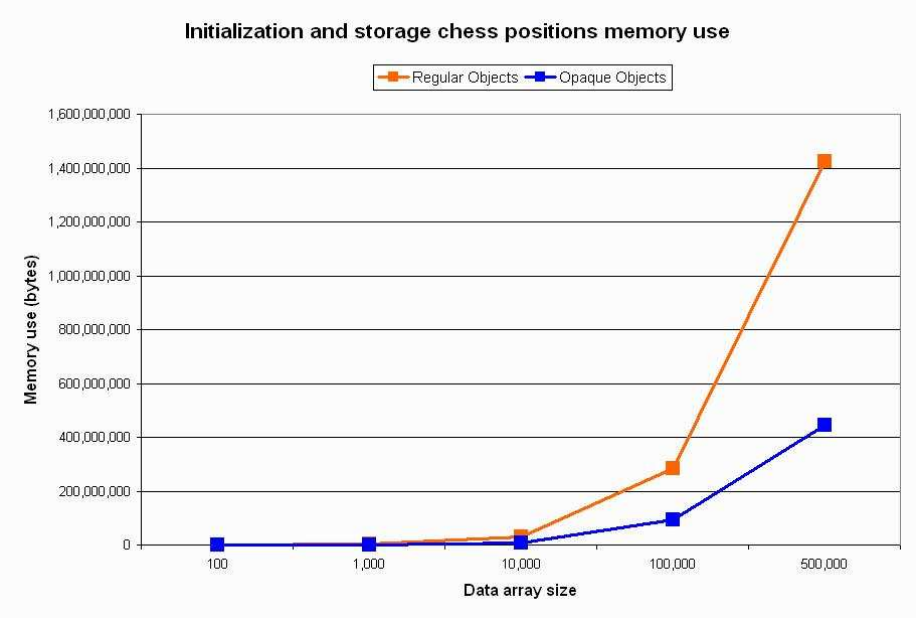
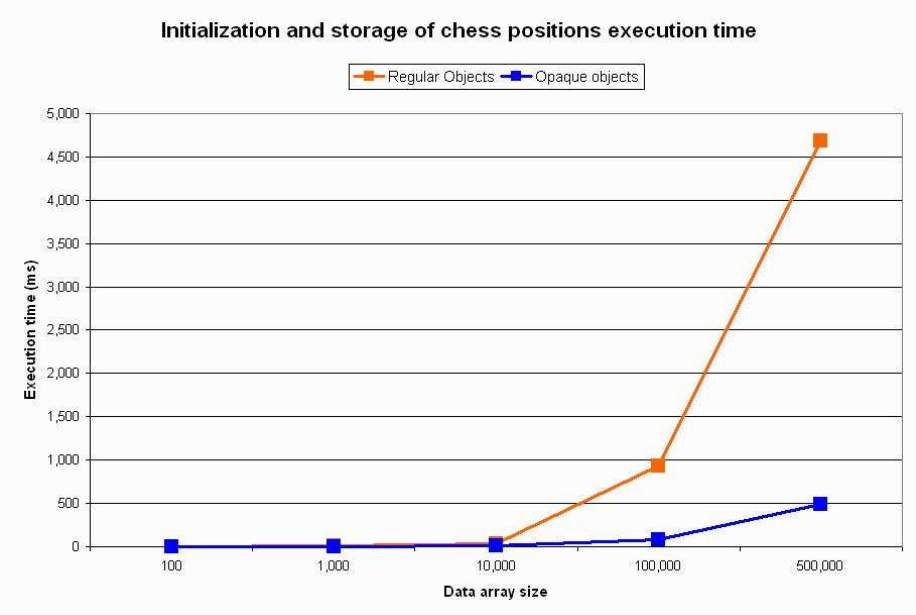


Figure 5.14: Initialization and storage of chess positions execution time



comparison in execution speed and memory use of the *ChessPosition* object and the primitively typed `long` array. Results are plotted on 5.15 and 5.16.

Table 5.5: Initialization and storage of chess positions
Platform: Intel C2Q Q6600 @ 3.3GHz, 4GB RAM, Windows XP SP3

	Regular ChessPosition array		**Opaque ChessBoard array (converted)**		**Improvement ratio**	
	Time(ms)	**Memory(bytes)**	**Time(ms)**	**Memory(bytes)**	**Time**	**Memory**
Array size:						
100	0	224,128	0	18,416	1.0	12.2
1,000	15	2,484,092	0	55,200	n/a	45.0
10,000	32	28,920,860	0	642,656	n/a	45.0
100,000	953	282,616,116	47	6,331,488	20.3	44.6
500,000	4704	1,432,704,640	250	31,924,488	18.8	44.9

Figure 5.15: Initialization and storage of chess positions memory use

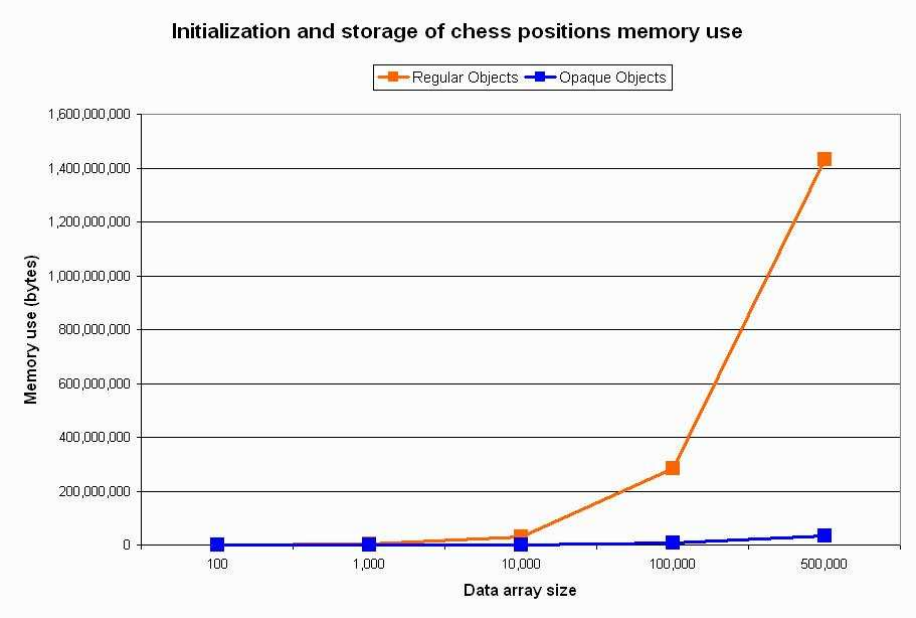
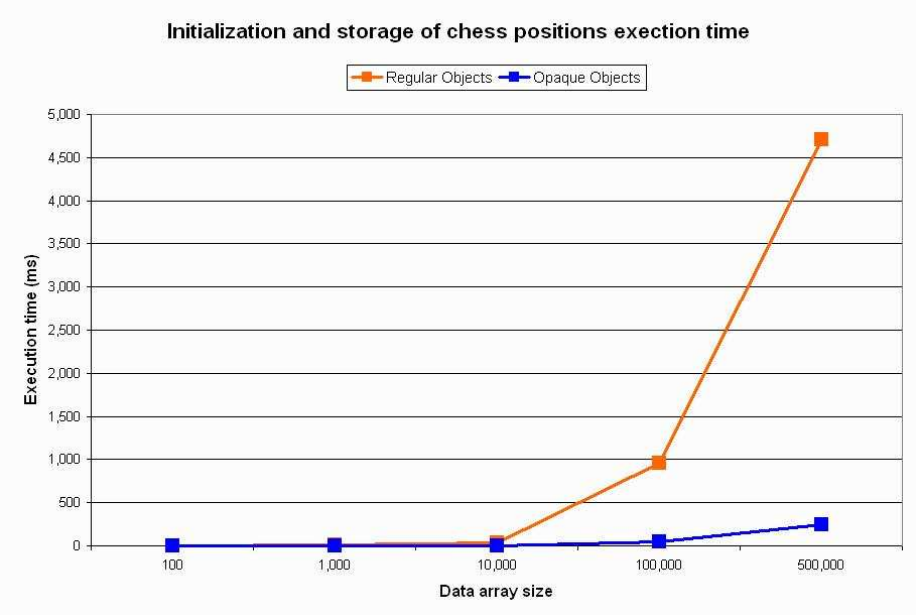


Figure 5.16: Initialization and storage of chess positions execution time



In order to complete these tests, the available memory for JVM heap space had to be raised to approximately 1600MB (`-Xms1600M -Xmx1600M`).

The testing of more realistic objects shows just how beneficial *opaque types* can be in Java applications in terms of gaining performance. Non-trivial implementations demonstrate that primitive types vastly outperform object types when it comes to execution time and memory use. The classes which were declared and initialized are

able to perform exactly the same operations; however, the *opaque type* objects (later converted to their respective primitive type representation) performed an astonishing **15-20** times faster and took up an amazing **40-45** times less heap space. This indicates the realistic potential of the approach presented in this work. The trade off of working out a somewhat more complex implementation for *opaque type* methods yields an enormous performance improvement in an application where space and speed are of vital importance.

It is important to highlight that going just from *opaque* objects to converted *opaque types* (which comes at a very low cost and practically no effort) improved execution speed **2-5** times, and reduced memory use by a factor of about **15**. This is a very satisfying result. It demonstrates that, not only does the *opaque types* approach enforce general code structure that is more efficient but it allows for further performance improvements without sacrficing readability and abstraction. Since the code changes the conversion utility applies are not major, in combination with *opaque type* declaration and use conventions, the two-step process yields significant performance benefits without severely hindering the expressive power of the language.

5.4.1 Measuring move generation and evaluation.

The following experiments build on the previous representations of the chess board (*ChessPosition* and *ChessBoard*) by implementing procedures for move generation and evaluation given a particular board position. This kind of test represents another realistic gauge of performance for an application manipulating and accessing large sets of data.

Delving further into more complex operations allows for a better picture of the genuine potential that *opaque types* entail; and while it is no secret that primitive types outperform object types in practice, working with built-in types quickly becomes a chore without a more sophisticated mechanism. With *opaque types*, we make an attempt at exactly that, a well developed technique for manipulating and implementing complicated concepts while using primitive types and reaping the performance benefits this brings about.

Move generation is an important part of any chess playing software. It simulates thorough access of the underlying data. A given chess position must be fully examined in order to return a structure containing every possible move allowed by one or both of the sides. This provides opportunity to compare performance of regular objects and *opaque types* not only in data storage but also, and primarily here, in data

retrieval, which is also vital in majority of realistic applications. Measuring the speed of data retrieval and its manipulation allows us to make preditictions regarding the overall performance of the application as all major areas of interaction with data are covered. The chess board is first declared, initialized, and then stored in memory. The internal representation is then retrieved, analyzed, and changed according to some rules. Finally, the resulting data is once again stored, thus completing the cycle and supplying us with a good estimate on the total application performance.

Figure 5.17: Regular MoveGenerator - Representation

```
1  public class MoveGenerator {
2
3    /*
4     * Checking squares in different directions
5     * given an initial square, this enumeration
6     * provides a different implementation of the
7     * move method depending on direction of
8     * and piece type being moved
9     */
10   public enum Direction {
11     E {
12       ChessSquare move(ChessSquare s, ChessPosition cp) {
13         if (s.file < 'h')
14           return new ChessSquare((char)(s.file + 1), s.rank, cp);
15         else
16           return null;
17       }
18     };
19
20     /*
21      * All other relevant directions implemented here
22      */
23     ...
24
25     abstract ChessSquare move(ChessSquare s, ChessPosition cp);
26   }
27
28   ...
```

The regular object-based *MoveGenerator* class conveniently uses a Java `enum` to represent all directions that a chess piece can move on the board. The enumeration provides a straight forward implementation of the *move* method for each appropriate direction (and where necessary, appropriate piece). The *move* method returns a new

ChessSquare where the current piece would finish as a successful result of the move the given *Direction*.

In order to utilize the different implementaions of the *move* method, we would like to generate all possible moves that a particular *ChessPosition* contains. The following continuation of the *MoveGenerator* class illustrates the construction of the *generate* method.

Figure 5.18: Regular MoveGenerator - generate method

```
1   ...
2   /*
3    * Given a ChessPosition, generate all legal moves
4    * allowed for the position.
5    */
6   public static Vector<ChessMove> generate(ChessPosition pos){
7     Vector<ChessMove> moves = new Vector<ChessMove>();
8
9     // look at every square in pos
10    for (ChessSquare curSquare: pos.board){
11      if (curSquare.piece.type != Type.EMPTY){
12        Vector<ChessSquare> legalSquares = new Vector<ChessSquare>();
13
14        if (curSquare.piece.type == Type.ROOK)
15          for (Direction d:
16           EnumSet.of(Direction.E, Direction.N, Direction.W, Direction.S))
17            getLegalSquares(curSquare, curSquare.piece, pos, legalSquares, d);
18
19        else if (curSquare.piece.type == Type.KNIGHT)
20          for (Direction d: EnumSet.range(Direction.NEE, Direction.SEE))
21            getLegalSquares(curSquare, curSquare.piece, pos, legalSquares, d);
22        ...
23        }
24
25        for (ChessSquare sq: legalSquares)
26          moves.add(new ChessMove(curSquare, sq));
27      }
28    }
29    return moves;
30  }
```

The *generate* method returns a `Vector`$<ChessMove>$ of legal *ChessMove*'s given a particular *ChessPosition*. The use of a Java `enum` allows application of some of the *EnumSet* methods that limit which subsets of the enumeration are counted. This presents an opportunity to reduce the code size somewhat and artificially implement different chess piece behaviour by using only the legal directions of the particular

pieces. `for` loops on lines 15 and 19 demonstrate this particularly well for the *rook* and *knight* pieces.

Generation of possible moves in this, regular object based implementation, occurs when the `static` *generate* method is called and passed, as a parameter, a particular *ChessPosition*. Every square in the given position is then analyzed, and the occupied squares are passed to the helper method *getLegalSquares* along with the piece type occupying them. The method returns all legal squares that the current piece may occupy as a result of a move on the pertinent board position. Finally, all the legal moves are packed together into *ChessMove* objects (which are represented by an initial and final squares) and returned as a vector.

The helper *getLegalSquares* is illustrated in Figure 5.19 to emphasize how the object based implementation is accessed through the move generation task.

The implementation attempts to use Object-Oriented Design paradigms where possible. The code tries to be clear and self-explanatory with regards to checking various properties pertaining to legality of chess moves.

Figure 5.20 depicts part of the *opaque type* based *MoveGenerator* class. The class is annotated with *@Opaque("user")* type as does not have a primitive type representation and serves only as a method facility for accessing, manipulating, and storing chess positions.

genMoves method accomplishes the same task that the *generate* method did in the regular object implementation of the *MoveGenerator* (Figure 5.18, line 6). This method, however, returns an array of `Short`s ("boxing" class for the primitive `short` in Java) each of which represents a separate possible chess move. The method *getSquares* is this object's equivalent of the *getLegalSquares* method presented in Figure 5.21. It scans the current position and returns all possible moves that can be applied from a given square (granted it is occupied by a chess piece).

The method undertakes a similar linear analysis of the board squares relevant to the current square being examined and the chess piece occupying it. The major difference lies in the lack of a Java `enum` type to aid with changing the direction of movement along the 2-dimensional board. This puts a strain on the readability of the code but not to the point of making the code completely cryptic. Most of the analysis is done by representing and numbering the board squares linearly, i.e., the squares are numbered 0 through 63, with the top left square (`a8`) being 0 and bottom right (`h1`) being 63. The examination of various squares can then be done without the need

Figure 5.19: Regular MoveGenerator - helper method

```
1  private static void getLegalSquares(ChessSquare s,
2                      ChessPiece curPiece,
3                      ChessPosition pos,
4                      Vector<ChessSquare> sqs,
5                      Direction dir) {
6
7    ChessSquare potential = dir.move(s, pos);
8    // pawn move
9    if (EnumSet.range(
10     Direction.W_PAWN_ONE, Direction.B_PAWN_TWO).contains(dir))
11     if (potential != null)
12       if (potential.piece.type == Type.EMPTY)
13         sqs.add(potential);
14   // pawn capture
15   if (EnumSet.range(
16     Direction.W_PAWN_CAP_NW, Direction.B_PAWN_CAP_SE).contains(dir))
17     if (potential != null)
18       if (potential.piece.type != Type.EMPTY
19          && potential.piece.color != curPiece.color)
20         sqs.add(potential);
21   // we're still on the board
22   if (potential != null && curPiece.type != Type.PAWN){
23
24     if (potential.piece.type != Type.EMPTY) {
25       // piece of opposite color,
26       // so can capture, but can't go any further
27       if (potential.piece.color != curPiece.color)
28         sqs.add(potential);
29     }
30     // it's an empty square
31     else {
32       sqs.add(potential);
33       if (curPiece.type == Type.ROOK
34           || curPiece.type == Type.BISHOP
35           || curPiece.type == Type.QUEEN)
36         getLegalSquares(potential, curPiece, pos, sqs, dir);
37     }
38   }
39 }
```

for nested loops, which are usually natural for accessing and mutatting 2-dimensional data structures. Instead, the board is "flattened" and relevant squares simply become carefully chosen subsets of the {0 .. 63} set.

Comparison of the two implementations (regular object and *opaque* object) consists of declaring a number of chess positions using the respective representations; then applying a number of moves to the boards in order to create a small amount of

Figure 5.20: *Opaque* MoveGenerator

```
1  @Opaque(``user'')
2  public class MoveGenerator {
3    /*
4     * Generate a list of all possible moves given
5     * a board position. Each move is represented by
6     * a short (ChessMove). Bits 0-5 represent the
7     * initial square, bits 6-11 represent the final
8     * square.
9     */
10   public static Short[] genMoves(ChessBoard pos){
11     int currentPiece = 0;
12     int currentSquare = 0;
13     long mask = 15L;
14     Vector<Short> moves = new Vector<Short>();
15
16     for (long l: pos.rep){
17       long sqs = l;
18       currentPiece = (int) (sqs & mask);
19       Integer[] relevantSquares =
20            getSquares(currentSquare, currentPiece, pos);
21       for (Integer i: relevantSquares)
22         moves.add((short)(currentSquare & (i << 6)));
23     }
24
25     return (Short[]) moves.toArray();
26   }
27   ...
```

variation in the positions. Finally, the possible moves for each position are generated using the respective methods shown in Figures 5.18 and 5.20 and if the number of possible moves returned is greater than 30, one of the moves is chosen at random and applied to the current position. Once again, two sets of tests are performed to gauge the significance and impact both *opaque type* structure and conversion of *opaque types* to underlying primitive types have on application performance.

The data sizes have been decreased and brought closer together in this set of tests due to the complexity of the undergoing operations. Move generation is a time and space consuming task for both types used in Table 5.6. However, even prior to conversion,

Table 5.6: Move generation
Platform: Intel C2Q Q6600 @ 3.3GHz, 4GB RAM, Windows XP SP3

	Regular ChessPosition array		**Opaque ChessBoard array (unconverted)**		**Improvement ratio**	
	Time(ms)	**Memory(bytes)**	**Time(ms)**	**Memory(bytes)**	**Time**	**Memory**
Array size:						
100	109	2,067,008	0	266,008	∞	7.8
500	188	12,402,424	15	1,172,760	12.5	10.6
1,000	282	24,804,096	31	2,343,952	9.1	10.6
5,000	1,047	124,974,120	87	11,452,992	12.0	10.9
10,000	2,083	249,314,872	124	22,716,448	16.8	11.0
50,000	9,406	1,258,011,204	590	103,925,952	15.9	12.1

Figure 5.21: *Opaque* MoveGenerator

```
1  public static Integer[] getSquares
2        (int square, int piece, ChessBoard position){
3
4      Vector<Integer> sq = new Vector<Integer>();
5
6      if (piece == ChessBoard.B_ROOK || piece == ChessBoard.W_ROOK)
7        sq.addAll(getRookMoves(square));
8
9      else if (piece == ChessBoard.B_KNIGHT || piece == ChessBoard.W_KNIGHT){
10       int[] diff = {17, 15, 10, 6};
11       for (int d: diff){
12         if (square - d >= 0)
13           sq.add(square - d);
14         if (square + d <= 63)
15           sq.add(square + d);
16       }
17     }
18
19     ...
20     return (Integer[])sq.toArray();
21   }
```

we once again see that use of the *opaque ChessBoard* class yields much better results than the regular object *ChessPosition*. Since both board declarations were initialized to starting positions and only several moves were applied to each position prior to move generation; majority of positions generated a number of moves that is greater than 30. Therefore, the move generation tests were not only heavy in position access but also in position manipulation as a lot of move application occurred.

The next set of tests conducted, once again involved move generation given a number of positions. This time, however, the *opaque ChessBoard* objects were converted to their underlying `long[]` type. The conversion of *opaque* classes *ChessBoard*, *ChessGame*, and *MoveGenerator* took a very small amount of time for the code size of the respective objects demonstrating that code conversion is an insignificant factor in measuring execution speed. This is due to the number of changes that the converter makes to the code being very small and the conversion utility completing the transformation in a single pass.

Table 5.7: Move generation
Platform: Intel C2Q Q6600 @ 3.3GHz, 4GB RAM, Windows XP SP3

	Regular ChessPosition array		**Opaque ChessBoard array (converted)**		**Improvement ratio**	
	Time(ms)	**Memory(bytes)**	**Time(ms)**	**Memory(bytes)**	**Time**	**Memory**
Array size:						
100	101	2,062,496	0	161,192	∞	12.8
500	209	12,112,224	8	773,800	26.1	15.7
1,000	285	24,046,440	14	1,507,960	20.4	15.9
5,000	997	118,972,620	62	7,665,664	16.1	15.5
10,000	2,013	244,394,860	97	15,849,600	20.8	15.4
50,000	10,002	1,238,998,804	298	78,907,312	33.6	15.7

Figure 5.22: Move generation memory use

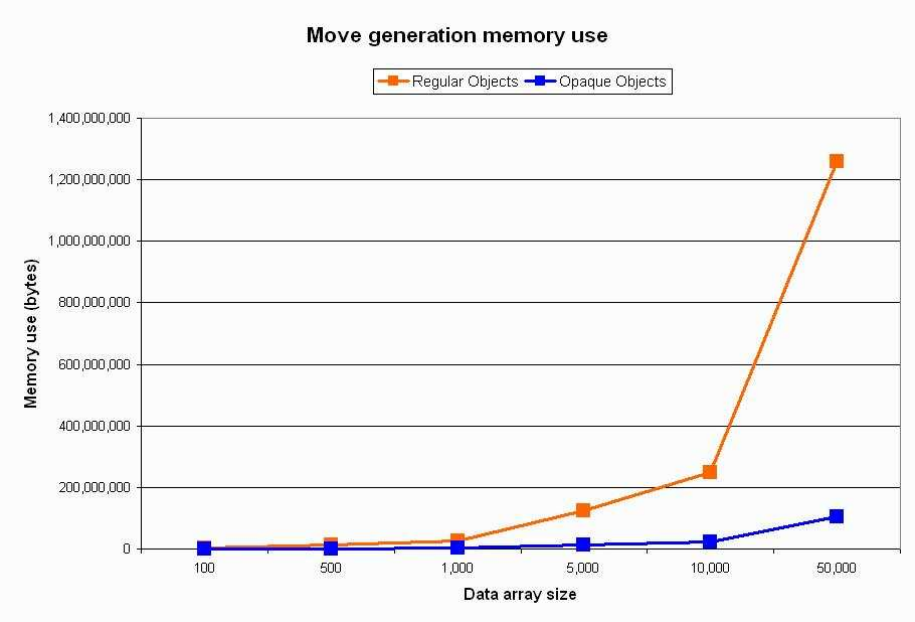
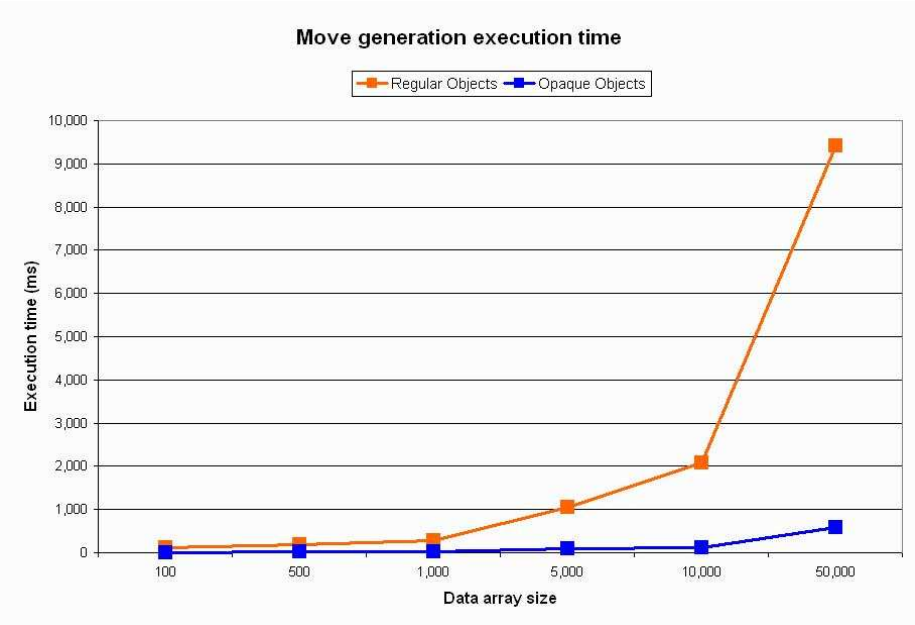


Figure 5.23: Move generation execution time



Once again, converted *ChessBoard* objects and `static` methods acting on them in the *ChessBoard* and *ChessGame opaque* classes show a significant improvement over the unconverted *opaque types* and regular object implementations. The converted *ChessBoard* objects (or `long` type arrays) performed on average **18-20** times faster than the regular object types and showed an small improvement (about **1.5-2** times) in execution speed over unconverted *opaque types*. The `long` type arrays also used

Figure 5.24: Move generation memory use

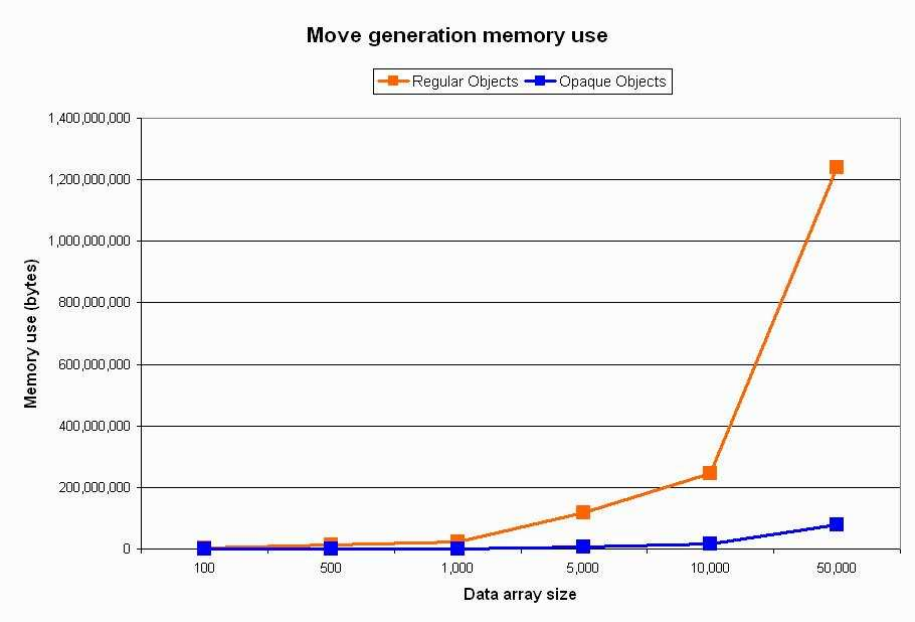
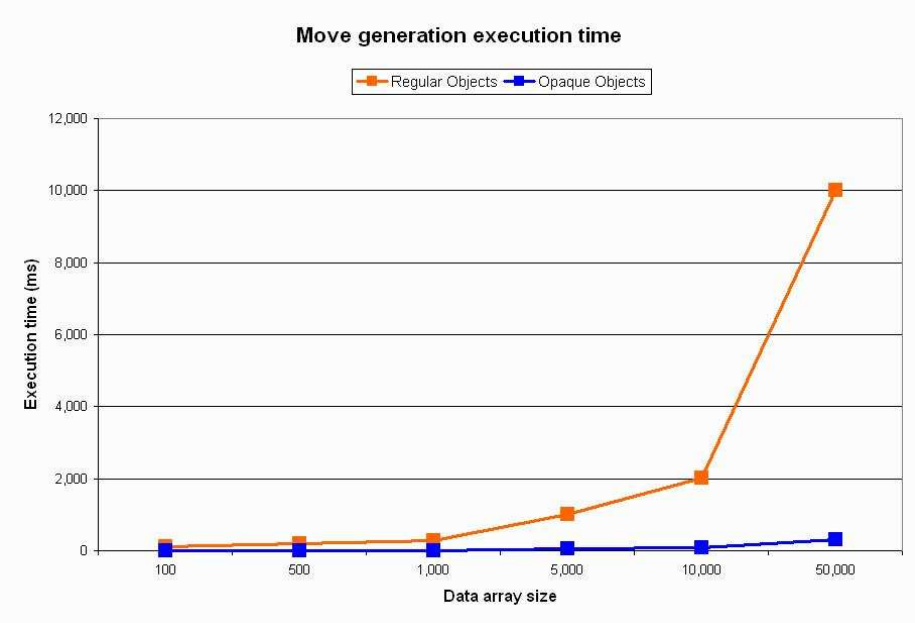


Figure 5.25: Move generation execution time



approximately **13-18** times less memory in order to accomplish the same tasks as the regular object *ChessPosition* counter-parts; and **1.3-1.5** times less memory than the unconverted *ChessBoard* and *ChessGame opaque types*.

Finally, it is useful to compare the performance results of the "move generation" tests visually, together on a single graph. The execution time and memory use values of Tables 5.6 and 5.7 are plotted in Figures 5.26 and 5.27 respectively. The figures

demonstrate the improvement code conversion creates without any additional work other than invoking the conversion utility on the already efficient implementation of the opaque typed code.

Figure 5.26: Move generation memory use

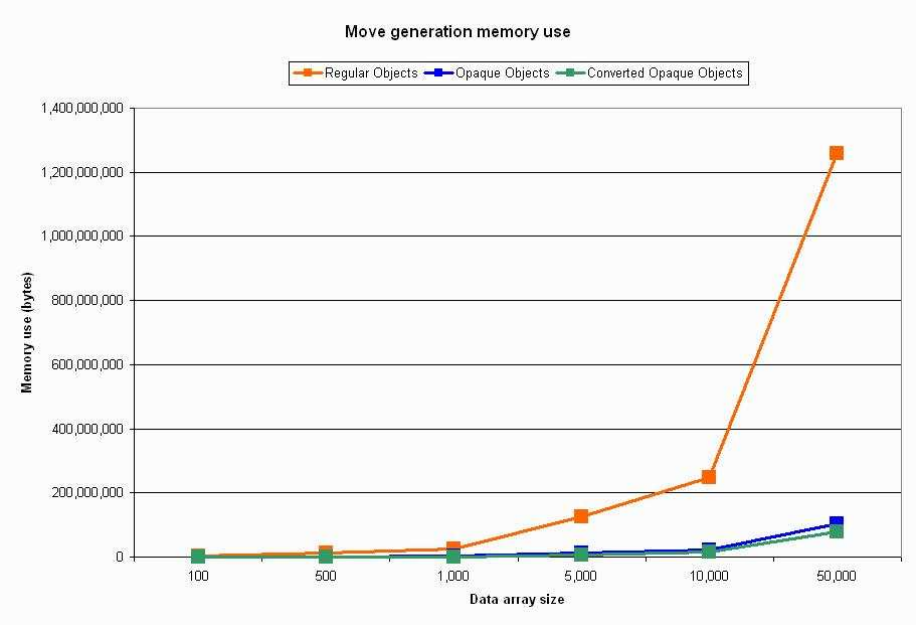
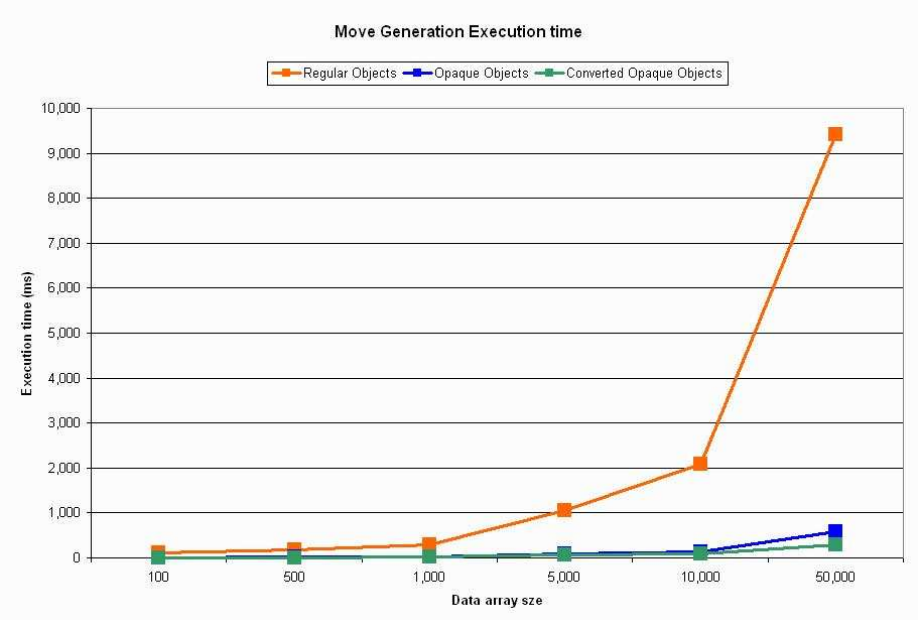


Figure 5.27: Move generation execution time



The performance tests conducted and summarized in this section all but confirm *opaque type* superiority to regular object types in execution speed and memory use. The results shown here are just a fraction of the experiments that were carried out to measure and maximize performance of *opaque types* in Java. The majority of the performance experiments were done using small to medium sized classes in order to preserve the one-to-one correspondence between regular and *opaque* classes. Conducting the tests in this manner allowed us to focus on precisely comparing the implemented *opaque type* features versus the constructs already available in standard Java. Isolating and comparing the features directly gives a more detailed overview of exactly why *opaque types* perform that much better than regular object types. Hence, the tests were conducted throughout the theory development process in order to shape, guide, and when necessary, correct the ideas suggested throughout the lifetime of the approach development.

Testing of application performance using different programming languages is an acknowledged area of research with many articles and results published on the subject. Therefore, we considered it vital to include the more involved application of *opaque types* (section 5.4.1) in the performance chapter in order to give the reader the satisfactory feeling of seeing realistic results that have come out of implementing the theory behind *opaque types*. It was also useful in showing that code conversion plays a significant enough role in performance improvement without any noticable overhead; implying that even lexical code conversion is a viable approach to *opaque type* implementation and realization.

Chapter 6

Conclusions and Future Work

6.1 Conclusions

The approach described in this work is intended to be an easy-to-use tool to allow fast type safe Java code while preserving existing features of abstraction and robustness. Scarsity of restrictions and ease of use makes possible for flexible and straight forward implementations of a vast variety of concepts without a steep learning curve. The use of metadata in the form of code annotations allows for efficient code transformation that yields immediate and noticable performance improvements.

Performance is vital in software; *opaque* object types should allow for Java to become a more versatile language and make it possible to be used for large scale projects where speed and resource use are of great concern. Performance experiments that were carried out demonstrate the flexibility of our approach and its ability to improve resource use with sacrificing code integrity.

During our experiments we were able to achieve performance that is **15-20** (depending on application) times faster in execution speed and took up **40-45** less heap space when using *opaque types* instead of regular object types. The goal of the performance investigations was to measure exactly how much more efficient *opaque types* are compared to object types. Achieving these figures shows the true potential of our approach in making Java a more desirable language for performance-essential applications. The approach attempts to put Java at the forefront of high performance general programming languages while maintaining its power to express a high variety of problems.

Our method complements the research and implementations that have been done in the past. Techniques that have had varying degrees of success may benefit from

the approach and become more commonplace. The proposed concept deals with an industry standard language thus establishing its importance immediately.

It is the hope of the authors that this work demonstrates a viable tool for expanding Java's general application domain as well as, a valuable step along the way to general purpose programming languages becoming more efficient and able to solve a greater variety of problems.

6.2 Native implementation

Current implementation of the approach relies heavily on the use of external tools. The code converter is a utility written in Java and distributed as a `jar` file that has to be referenced during compilation. Despite the converter having been shown to be a very efficient tool, it is easy to see how a more immediate, low-level, implementation can benefit the *opaque types* approach.

The build script, which automates the conversion and compilation process is written in Ant and requires `ant` to run. Meanwhile integration into Eclipse is already partially completed with the Ant build script approach, it is not as intuitive and versatile as regular Java language features that are implemented at a deeper level.

Usability and versatility are not the only factors that must be considered when popular languages are augmented with useful features. In order to promote adoption by the language standard, the new language features should seamlessly integrate into the language without causing any awkwardness in their application. The new constructs (*opaque types*) should also not hinder the long-standing good performance and reputation of the language with bloated external tools and alternate building processes.

Seamless implementation into an existing compiler would speed up the building process somewhat and, more importantly, make it more convenient for the developers to employ the power of *opaque types*. Making *opaque types* easier to use will help with adoption for applications where performance may not be vital. Introduction into the language standard will also be aided by this as the use for external tools would cease to exist and experimental use by a wider audience would have a greater opportunity to begin.

In turn, distribution of the implementation would no longer be a difficult step in encouraging Java developers to try out *opaque types*. Thus the benefits of using the new types could be more easily used to convince the general Java developer to find applications for *opaque types* where he or she can get the most out of them.

There are several approaches to providing a more concrete implementation for the building process. A “convert” or “opaque” flag for an existing compiler is perhaps the most straight forward. Given the flag, any properly annotated code can be converted prior to compilation. This does not alter the existing building process much except for speeding it up and making it more invisible. In addition, a more efficient implementation of the converter utility can be employed to speed up the process further.

Another alternative to improving opaque type intergration into Java is implementing the approach into an existing Java IDE (e.g. Eclipse). Since some IDE's keep an up-to-date internal representation of the code, compilation time is significantly improved. Adding the capability to recognize and represent opaque types using the internal syntax helps abstract the feature implementation and speeds up the building process.

Although not as attractive as direct compiler and JVM implementation, augmenting a high level Java code representation is a natural next step to take for the *opaque types* approach. Implementing *opaque types* in an alternative representation should shed light on further developements the approach could take as more factors come into play.

6.3 Computer algebra applications

There are many areas where the Java programming language lacks expressive power and efficiency for a reasonable implementation of a particular application. One of such areas is computer algebra applications where problems require high level language features to express but grow quickly in size requiring very efficient implementations to perform reasonably.

Allowing use of primitive types while preserving the high level Java language features should prove valuable in designing solutions to mathematical problems. Problems that previously required algorithms to be optimized mostly by hand, due to complex internal structure, could be handled easier by general optimizing compilers. While Java's expressive power might still lag behind some of the languages specifically designed for representing mathematical problems, Java would become a viable choice for subroutine and algorithm implementations that in turn become parts of a larger application.

This is the general aim of a native implementation of *opaque types* as it broadens the spectrum of possible uses and audiences for Java; meanwhile maintaining its status

as a general purpose language that can solve a variety of problems from different domains elegantly and efficiently.

6.4 Digital ink applications

Compact and efficient representation of digital ink is another application where the *opaque types* approach could be an important step forward. Ink can be represented as a series of points and strokes on the 2-Dimensional plane. As the amount of points and strokes increases, the size of the data required to store the ink quickly grows. *Opaque types* would allow efficient storage and manipulation of ink data without obfuscating the already complex algorithms used for its analysis and recognition.

For instance, points in the X-Y plane are currently represented by regular objects with two floating point fields standing for X and Y coordinates. The coordinates could potentially be converted into `int` types and "packed" into a single `long` type object field. This is a very natural application of the *opaque types* approach that could be implemented naturally as only "packing" and "unpacking" operations would need to be developed from scratch in order to start utilizing the benefts of *opaque types*.

We aim to make use of *opaque types* in existing digital ink software in the near future. We expect this to introduce the observed performance benefits of *opaque types* to the analysis and recognition software currently used researchers in the area.

6.5 Java generics

A relatively recent important feature of Java is *generics*. It is an important branch of the language that the *opaque types* approach has not touched upon. As part of the future work, it is very important to put in place theory and implementation to incorporate Java *generics* into the techniques introduced in this thesis.

Development of general algorithms that work on many data types is a valuable byproduct of Java *generics*. Combining *generics* and *opaque types* will take Java further in the direction of being a language with great expressive power and efficient resource use.

Bibliography

[1] G. Bracha. Generics in the java programming language. 2004.

[2] Walter E. Brown. Toward opaque typedefs in c++0x. 2004.

[3] Walter E. Brown. Progress toward opaque typedefs for c++0x. 2005.

[4] B. Fulgham. The computer language benchmarks game, 2009. `http://shootout.alioth.debian.org`.

[5] Samuel P. Harbinson. *Modula-3*. Prentice-Hall, Inc., Englewood Cliffs, NJ 07632, 1992.

[6] Rachel Harrison. *Abstract Data Types in Standard ML*. John Wiley and Sons Ltd, Baffins Lane, Chichester, West Sussex PO19 1UD, England, 1993.

[7] B. Johnston. *Java programming today*. Upper Saddle River, NJ; Pearson Prentice Hall, c2004., 2004.

[8] Elliot B. Koffman. *Objects, abstraction, data structures and design using Java*. John Wiley and Sons, c2005., Hoboken, NJ., 2005.

[9] G.; Magnusson E.; Ekman T. Nilsson-Nyman, E.; Hedin. Declarative intraprocedural flow analysis of java source code. *Electronic Notes in Theoretical Computer Science*, 2009.

[10] Larry R. Nyhoff. *C++ An Introduction to Data Structures*. Prentice-Hall, Inc, Upper Saddle River, NJ 07458, 1999.

[11] C. Rathman. Shapes oo example: Modula-3 code. `http://www.angelfire.com/tx4/cus/shapes/modula3.html`.

[12] A. Rossberg. Generativity and dynamic opacity for abstract types. *ACM*, 2003.

[13] B. Stroustrup. *The C++ programming language*. Reading, Mass. Addison-Wesley, c1997., 1997.

[14] Inc. Sun Microsystems. Trail: Learning the java language, 1995-2009. `http://java.sun.com/docs/books/tutorial/java`.

[15] Inc. Sun Microsystems. Annotations, 2004. `http://java.sun.com/j2se/1.5.0/docs/guide/language/annotations.html`.

[16] David A. Watt. *Programming Language Design Concepts*. John Wiley and Sons Ltd, The Atrium, Southern Gate, Chichester, West Sussex P019 8SQ, England, 2004.

[17] S.M. Watt. Aldor. In E.;Weispfenning V. Grabmeier, J.;Kaltofen, editor, *Handbook of Computer Algebra*, pages 265–270. Springer Verlag, Heidelberg, 2003.

VITA

NAME: Pavel Bourdykine

EDUCATION:

The University of Western Ontario
London, Ontario, Canada
September 2008 to December 2009
M.Sc. (Computer Science)
Supervisor: Dr. Stephen M. Watt

The University of Western Ontario
London, Ontario, Canada
September 2004 to April 2008
B.Sc. with Honours (Computer Science)

EXPERIENCE:

Teaching and Research Assistant
The University of Western Ontario
Dept. of Computer Science
September 2008 to December 2009

Lab and Network Administrator
The University of Western Ontario
ORCCA Lab
November 2008 to December 2009