

Colorado Technical University



**Technical Report
Computer Science**

Adding dynamic type binding to Java

Richard L. Hagen

rhagen@mastercodesmiths.com

**Technical Report Number
CTU-CS-2003-001**

Adding dynamic type binding to Java

Richard L. Hagen

Abstract

Java performs static type checking at compile time and dynamic binding at runtime. The Java Reflection API allows programmatic access to information about the fields, methods, and constructors of loaded classes. Through the Java Reflection API, programmers are able to dynamically invoke a method on an object at runtime. The Java Reflection API effectively allows the programmer to perform dynamic type binding while maintaining type safety. However, there is a fair amount of syntactical baggage that comes with using the Reflection API that obfuscates the code making it difficult to read and understand. This paper explores the possibility of adding native dynamic type binding to the Java language while maintaining the benefits of static type checking and eliminating the syntactical baggage of the Java Reflection API.

Introduction

The Java programming language is a statically typed language that provides facilities through the Reflection API to look up methods defined by a class or interface and execute those methods at runtime. The Java Reflection API is not part of the Java language specification. It is instead a class library that provides runtime introspection services. A programmer can use these services to find methods that a particular class implements and then dynamically call that method. This facility can be very useful in a variety of ways especially when the type of a class is not known at compile time and there is a need to operate on that class at runtime. However, the Reflection API carries with it a great deal of syntactical baggage. This paper explores the possibility of extending the Java language to include native dynamic type binding while maintaining type safety and much of the benefits of static type checking.

Defining Terminology

Terminology related to binding and typing have different meanings to different groups of people. For this paper, a language is type-safe if the only operations that can be performed on data in the language are those defined for the type of the data (Saraswat 1997). Static type

checking means that every variable and every expression has a type that can be determined at compile time (Lindholm and Yellin 1999) and checks are performed to ensure variable, expression, and operation compatibility. Static binding is a compile time operation where methods are selected before program execution and the references to methods remain unchanged throughout program execution. Dynamic binding is a runtime activity where methods are found and selected dynamically during program execution. Finally, dynamic type binding is a runtime activity where operations are type-checked just before they are performed (Bruce 1996).

Type Checking and Binding in Java

Java performs static type checking at compile time and dynamic binding at runtime. The following sections describe these features and also introduce the Java Reflection API.

Static Type Checking

The Java programming language is type-safe and requires both compile time and real-time type checking to assure that messages sent to objects have an implementation. In the Java programming language, every variable and every expression has a type that can be determined at compile time (Gosling et al. 2000, 52). Java uses this explicit or deduced type information to perform static type checking during the compilation phase. The compiler checks for compatibility between each variable and each expression being assigned to it. Checks are also performed to assure that messages sent to objects are implemented by the target object. A compilation error is generated when an incompatibility is detected.

Dynamic Binding

The java compiler performs static type checking and assures that there is type compatibility across all variables and expression. The compiler also selects a method signature that is used by the runtime system to help it determine the form of the method to be called. The method signature includes the method name and the types of each of the formal arguments. Because of polymorphism, the compiler cannot reliably select the exact method that will be called for a given object. Figure 1 shows an example. Class T implements a method m. Class S inherits from T and overrides method m. Another class creates an instance of S and assigns it to a variable of type T. Using the variable of type T, the message m is sent to the referenced object. We would expect to see the output “S::m” because we are sending the message to an instance of

class S. Indeed, this is what happens, but only because of runtime dynamic binding. At compile time, the compiler only knows that a message is being sent to a referenced object whose type is T. The compiler has no way of determining that the value referenced is a subclass of T whose implementation may have overridden method m. The only way to make this determination is to perform a runtime check. The runtime system searches the class hierarchy of the target object for implementations of the method signature generated by the compiler and calls the method that is furthest down the hierarchy. This is dynamic binding.

```
public class T {
    public void m() {
        System.out.println("T:m");
    }
}
public class S extends T {
    public void m() {
        System.out.println("S:m");
    }
}

S s = new S();
T t = s;
t.m();           // we want to see S::m printed because of polymorphism even
                 // though we are sending the message using a variable of type T
```

Figure 1 – Example of polymorphism and dynamic binding

Java Reflection API

The Java programming language has a facility called “reflection” that allows programmatic access to information about the fields, methods, and constructors of loaded classes. The Reflection API supports the ability to use the introspected information received from a class to dynamically invoke operations on instances of the class. Figure 2 shows an example of using the Java Reflection API. To use the API for invoking a method on an arbitrary object, three steps are required. The first step is to retrieve the class definition for a given object. The class definition keeps track of, among other things, methods that can be sent to an instance of the class. The next step is to find a particular method definition using the class definition. To find the method definition a method name and an array of formal arguments for the method need to be passed to the class definition. The last step is to invoke the method on a given object using an array of ob-

jects that represent the actual arguments to the method. In addition, the code described above needs to be surrounded with a try/catch block that captures all possible exceptions that the reflection API generates.

```
public class T {
    public void print(String s) { System.out.println(s); }
}
T t = new T();
Class c = t.getClass();
Method m;
try {
    m = c.getMethod("print", new Class[] { String.class, });
    try {
        m.invoke(t, new Object[] { "Hello world!" });
    } catch (IllegalAccessException e) {
    } catch (InvocationTargetException e) {
    }
} catch (NoSuchMethodException e) {
}
```

Figure 2 – Using the Java Reflection API

Introducing Dynamic Type Binding to Java

As can be seen in Figure 2, invoking a single message on an object using the Java Reflection API involves quite a bit of work. The following section proposes adding the ability to perform this type of operation in Java without the extra baggage required by the Java Reflection API.

The “Any” Type

Dynamic type binding could be introduced into the Java language by adding a single “hard-wired” type that the compiler does not perform static type checking upon. A new Java keyword is needed to represent this type, and for the purpose of this paper it will be called “Any”. Any Java variable, whether a class variables, instance variable, array component, method parameter, constructor parameter, exception-handler parameter, or local variable, may be given the type “Any”. A Java expression may also produce an “Any” type. The compiler will not perform static type checking on variables or expressions of the “Any” type. This means that an object pointed to by a variable of type “Any” may be sent any message and the compiler will not check to see if the message is appropriate. The type of the referenced object will not be

known, so the check would not be possible. Instead, the check will be performed at runtime in a fashion similar to the Java Reflection API. For all other “non-Any” variables, Java’s normal static type checking continues to be performed.

```
Any t = new T();  
t.print ( “Hello World!” );
```

Figure 3 – Simple example using keyword Any

Figure 3 shows an example of assigning a variable to an “Any” type and sending a message to an object referenced by an “Any” type. On the first line, a new instance of T is created and assigned to the variable, t, which has the declared type of “Any”. The compiler does not check for compatibility between the expression “new T()” and the variable t because a variable of type “Any” may take on any instance type. On the second line, the message “print”, is sent to the object referenced by t with parameter “Hello World!”. Again, the Java compiler will not check for compatibility between the operation “print” and the variable referenced by t, instead, compatibility will be checked at runtime.

Analyzing the Implications to the Java Language

The following sections discuss the impacts of dynamic type binding on the Java programming language.

Maintaining Type Safety

Currently, Java maintains type safety by performing static and dynamic type checks. These checks will remain, but the compiler will delegate all “Any” type checks to the runtime system and will assume that messages sent to a type “Any” are valid. The runtime system will be responsible for checking compatibility between “Any” types and operations.

When compiling a program that contains a method invocation, for example, object.method(param), the compiler has a step that determines which method in a given class or interface is the method that should be used at runtime for method dispatching. During this step, the compiler locates the class or interface of the target object, and then searches for all method

declarations that are both applicable and accessible. There may be more than one such method declaration, in which case the most specific one is chosen (Gosling et al. 2000, 347). With the introduction of the “Any” type this step remains and the “Any” type will be regarded as the “least specific” of all of the reference types. For example, given the choice between two methods that are applicable and accessible, one with a parameter “Any”, and another with any other reference type, say Object, the method with the Object parameter will be selected as it is the “most specific” method. Currently, Java handles the case where no single method is “more specific” than another method and generates a compile time error. The ambiguity check will continue and will need to include the “Any” type. Figure 4 shows an example of an ambiguous message send.

```
public Class T {
    public T () {}
    public void m (Any a, String s) {}
    public void m (String s, Any a) {}
}

T t = new T();
t.m("String1", "String2");           // compilation error – m (String, String) is ambiguous
t.m((Any)"String1", "String2");     // ok – m(Any, String) method used
```

Figure 4 – Ambiguous message send

Casting

The “Any” type can receive any value that inherits from the class Object without having to perform casting. However, casting must always be performed when converting a type “Any” to a “non-Any” type. This includes all seven of Java’s variable types: class variables, instance variables, array components, method parameters, constructor parameters, exception-handler parameters, and local variables. Of course casting is not necessary when assigning a variable of type “Any” to another variable of type “Any”. At runtime, Java will continue to check that the object being cast is compatible with the type of the cast. A ClassCastException will be thrown when an incompatibility is detected.

Exception Handling

As shown previously in Figure 2, the Java Reflection API requires that a dynamic dispatch be surrounded by a try/catch block that catches the `IllegalAccessException`, `InvocationTargetException`, and the `NoSuchMethodException` exceptions. To be consistent with the reflection API, these checked exceptions could remain. If they do remain, then Figure 5 shows an example of using the “Any” in a message send. Note that the meaning of the code is clearer than that shown in Figure 2 using the Java Reflection API. However, a fair amount of baggage is still required to call a single method. This baggage is due to the fact that the exceptions generated by the Java Reflection API are “checked” exceptions. This means that the calling code needs to make sure that the exception is handled or that the method calling the code throws the same exception. If these exceptions are not handled, then a compilation error occurs.

```
Any a = new String();
try {
    int index = a.hashCode();
} catch (IllegalAccessException e) {
} catch (InvocationTargetException e) {
} catch (NoSuchMethodException e) {
}
```

Figure 5 - Any using standard Java reflection exceptions

There is another class of exceptions in Java called unchecked exceptions. These exceptions are not required to be caught by the calling code or to be declared as being thrown. Instead of using checked exceptions for the `IllegalAccessException` and `NoSuchMethodException`, a new unchecked exception could be created called `DynamicTypeBindException`. The `DynamicTypeBindException` would be thrown by the Java runtime when it detects a compatibility problem between a target object and an operation. The `InvocationTargetException` would remain as a checked exception, as it is used to wrap any exception that the called method may generate. By having the runtime environment only throw one checked exception, the code from Figure 5 would be cleaned up considerably to look like what is shown in Figure 6. The code is now very understandable.

```
Any a = new T();
try {
    a.print("Hello World!");
} catch (InvocationTargetException e) {
}
```

Figure 6 – Using Any with an unchecked DynamicBindException

Impacts to Existing Code

Existing Java code should continue to compile correctly with the introduction of the “Any” type, unless the keyword “Any” is used as an identifier or as a class name somewhere in the code. When this occurs, it should be relatively minor to correct by simply changing the identifier or class name from the word “Any” to another word.

Performance Impacts

Code that does not use the “Any” keyword should exhibit the same runtime performance characteristics as seen today. This is because non-Any types will continue to be statically type checked at compile time and dynamically bound at runtime. Dynamic type binding will not need to occur at runtime for non-Any types. However, there will be a performance penalty for using the “Any” type. When the “Any” type is used, new runtime code will need to be called that finds the appropriate method to use based on the method name and formal arguments. This is not currently done in Java’s existing runtime system and will incur a performance penalty in a system implementing dynamic type binding using the “Any” type.

Benefits and Drawbacks

Adding the “Any” type into the Java programming language effectively introduces dynamic type binding into the language. The language remains type-safe and would be a hybrid language that supports both static type binding and dynamic type binding. The following sections describe the benefits and the disadvantages of this proposal.

Benefits

There are several advantages that dynamic type binding provides a programmer. Dynamic type binding is generally thought to be a more expressive way to program. Programmers

concentrate more on solving a domain problem rather than struggling with class hierarchy limitations and language syntactical issues. The Java Reflection API is a good example. Using this API, the programmer is required to write at least ten lines of code to perform a single message send. It would be much easier to write, read, and maintain if those ten lines of code were replaced with four lines of standard looking Java code.

With dynamic type binding, class hierarchies could be created that are more generic than they are today and much of the down-casting that is performed could be eliminated. For example, Java has a variety of reusable classes that implement the Collection interface. Collections can store and retrieve any type of Object. However, whenever an object is retrieved from a collection, the client code must cast the object into its appropriate type before a meaningful operation can be performed. In contrast, if the collection interface stored and retrieved “Any” types then a cast would not be required after an object is retrieved from the collection. This would eliminate the need for down-casting to the appropriate class. Figure 7 shows an example. The example using the AnyVector is cleaner and does not require the explicit cast. The example also points out how information is lost using an Object based collection. The client retrieving the object from the collection must know the types that are in the vector and then perform a cast before sending a message to the object. In the “Any” based vector, the client doesn’t need to know anything about the type of object that resides in the collection, only that it responds to the single message m. This form of reusability is very powerful and does not exist in Java today.

```
// ----- Casting without an Any
Vector v = new Vector ();
v.add(new T());

((T)v.elementAt(1)).m();

// ----- Using an Any
AnyVector av = new AnyVector();
av.add(new T());

v.elementAt(1).m();
```

Figure 7 – Object based versus “Any” based collections

Drawbacks

The introduction of the “Any” type does not require that programmers use the new type. Programmers who do not use the feature will not be impacted. Programmers that do use the feature need to be aware that compatibility between objects and operations are checked at runtime when an “Any” is involved. This means that type and operation incompatibilities will only be found when the code with an error is actually executed. Without good test coverage these types of errors may make it into a production environment.

Another drawback in using the “Any” type is one of performance. The runtime system will be required to perform some of the checking that was previously delegated to the compiler. Again, this only applies to the “Any” type, all other types will be checked by the compiler. This will slow down performance because the runtime system will have to determine that a type and operation are compatible.

Finally, introducing dynamic type binding to the Java language opens doors for abuse and misuse. Programmers may find it convenient to declare a variable of type “Any” to get around fundamental design flaws in their programs instead of correcting the basic design problem.

Conclusion

It appears that dynamic type binding can be added to the Java programming language rather easily by the introduction of single keyword that, for this paper, was named “Any”. Alterations would need to be made to the Java compiler as well as the Java runtime environment to support this feature. The performance of existing systems would not be impacted by this change and additional flexibility would be added to the Java language. The use of the feature would be optional. Programmers could choose to use the new feature or not, but they would have a choice that is not currently available to them today.

Works Cited

- Bruce, Kim B. 1996. *Typing in object-oriented languages: Achieving expressiveness and safety*. Accessed 2002. Available from <ftp://ftp.cs.williams.edu/pub/kim/Static.ps.gz>.
- Gosling, James, Bill Joy, Guy Steele, and Gilad Bracha. 2000. *The Java language specification*. Edited by Lisa Friendly: Addison-Wesley.
- Lindholm, Tim and Frank Yellin. 1999. *The Java virtual machine specification, second edition*. 2nd. Addison-Wesley. Accessed 2002. Available from <http://java.sun.com/docs/books/vmspec/2nd-edition/html/VMSpecTOC.doc.html>.
- Saraswat, Vijay. 1997. *Java is not type-safe*. Accessed 2002. Available from <http://www.research.att.com/~vj/bug.html>.