Static vs. Dynamic Method Resolution in Java: The Case For Argument-Based Method Selection

Bela Ban
Dept. of Computer Science
Cornell University
bba@cs.cornell.edu

Aug 27 1998

1 Static vs. Dynamic Method Resolution

In Java, the semantics of dynamic method resolution are not the same as those of compile-time method resolution with respect to argument type matching. This paper compares a static (compile-time) to a dynamic (runtime) example of method resolution. It shows that the default mechanism for method selection in Java is minimal with respect to flexibility and discussed how it could be replaced. As example, an algorithm is shown that ispatches methods based on both the type of the receiver object as well as the types of the actual arguments, mimicking method selection of CLOS [Ste90].

Consider the following compile-time method resolution example:

```
public class res {

    public class MyClass {
        public String toString() {return "MyClass";}
    }

    public void foo(Object obj) {
        System.out.println("Object is: " + obj +
                           " [" + obj.getClass().getName() + "]");
    }

    public static void main(String args[]) {
        res r=new res();
        r.foo(r.new MyClass());
        r.foo("Bela Ban");
    }
}
```

This example works because the method chosen at compile time is determined according to the type of the argument. Expression `r.new MyClass()` is of type `MyClass`, therefore method `foo(MyClass)` is chosen. Expression `"Bela Ban"` is of type `String` which can be converted to `Object`, therefore method `foo(Object)` is chosen.

However, the latter case does not apply to dynamic method resolution, as exemplified below.

Let's assume the client needs to execute a method of an instance dynamically, given the name of the method and a list of arguments. This would typically be done using Java's core reflection API [Sun96] which allows to lookup the `Method` object corresponding to the method's signature and then use it to invoke the method. This is shown in the sample code below:

```java
public class MethodCall implements java.io.Serializable {
    private String method_name;
    private Vector args=new Vector();

    public         MethodCall(String name) {method_name=name;}
    public void    AddArg(Object arg)      {args.addElement(arg);}

    public Object invoke(Object target) {
        Class   cl=target.getClass(), formal_args[];
        Method  m;
        Object  retval=null, obj_args[];
        try {
            formal_args=new Class[args.size()];
            for(int i=0; i < args.size(); i++)
                formal_args[i]=args.elementAt(i).getClass();
            try {m=cl.getMethod(method_name, formal_args);}
            catch(NoSuchMethodException no) {
                System.out.print("Found no method called " + method_name +
                                 " in class " + cl.getName());
                return null;
            }
            obj_args=new Object[args.size()];
            for(int i=0; i < args.size(); i++)
                obj_args[i]=args.elementAt(i);
            retval=m.invoke(target, obj_args);
            return retval;
        }
        catch(Exception e) {
            System.err.println(e);
        }
        return retval;
    }

    public void foo(Object o) {System.out.println("Object: " + o);}

    public static void main(String[] args) {
        MethodCall    m=new MethodCall("foo");
        Object        o=new String("Bela Ban");
        m.AddArg(o);
        m.invoke(m);
    }
}
```

Method `Class.getMethod` looks up the desired method (`foo` in our case) according to the method's name and number and types of arguments.[1] However, contrary to the compile-time example above, in which the formal types of the arguments are

[1]In the above example, the arguments to method `getMethod` must be the method's name and a list of formal parameters of the method. The formal parameters are derived from the actual arguments supplied (`getClass()`).

known (`Myclass` and `String`) at compile-time to select the correct method, the dynamic approach has to resolve it at runtime. The problem with the method that resolves a method dynamically (`Class.getMethod` and `Class.getDeclaredMethod` respectively) is that it only accepts exact matches between formal parameters and actual arguments, i.e. matches which are of exactly the same type. It does not allow subtype matching, e.g. an actual argument of type `String` which is a subclass of `Object` does not match with a formal parameter of `Object` !

2 Custom Method Resolution

This problem can be avoided by using a custom mechanism for method resolution. In fact, any function that accepts the target class, method name and a list of arguments and returns a `Method` object can be used to override the default `Class.getMethod` mechanism. Such a mechanism is presented below. Note, however, that due to the potential complexity of such a mechanism and - on the other hand - the *native* implementation of `Class.getMethod`, we may have to expect at least an order of magnitude more CPU cycles for a custom mechanism over the default one. However, where flexibility is desired over speed, custom method resolution offers powerful features such as selecting a method not only on the type of the receiver (as in C++ and Java), but based on the types of all arguments and the return type (as in CLOS [Ste90], Ada). The problem described above could be solved using a relatively simple, yet more powerful, approach to method resolution than the default one.

An algorithm for selecting a method based on all arguments is sketched below:
Input: target_class, method_name, list of arguments
Output: Method

1. Select all Methods of the target_class into a list (including parent classes). This is done using method `Class.getMethods`
2. Create an (empty) result list which will contain the matching methods
3. Iterate through the method list:
   (a) Match method_name. If match fails, continue with the next method
   (b) Match number of arguments with Method's formal parameters. If not the same arguments than defined in the formal paramater list, continue with the next method
   (c) Match arguments against formal parameters:
      i. Assign a number to each argument list (number is sum of all numbers for individual arguments). Tag method with rank, add method

to result list and continue with next method. The rank for each argument is:

   A. Exact match with formal parameter = rank 0. E.g. class of argument.equals(class of parameter).
   B. No match: continue with next method
   C. Argument is subtype of parameter: compute *distance* between subtyp and formal parameter's type. Direct subtypes = 1, indirect subtypes = ¡distance¿. Distance is the number of intermediate types between formal parameter and actual argument. E.g. in A – B – C – D, D would be assigned 3, and B 1.

4. If the resulting list is empty: error (no method found)

5. If there is exactly 1 Method object: return it

6. If there are multiple Method objects: return the one with the lowest assigned rank. Error if there are multiple methods with lowest rank.

A simpler and more efficient approach could be to match only method name and number of parameters first. If 1 `Method` object remains, it is returned. This may still result in a runtime error as no parameter type checking has been performed. However, it avoids the tedious (and slow) work of parameter type checking. One could go a step further after the initial round of method selection, and - if more than a single method has been found - add another round of type matching.
A slightly modified algorithm for class `MethodCall` is available at [Ban]. There is also a test program `MethodCallTest`.
The dynamic method resolution approach based on the types of arguments has been used in the context of the JavaGroups [Ban98] toolkit. It allows users of the toolkit to customize the semantics of remote method invocation by creating special `MethodInvoker` objects which are tailored to a specific method resolution mechanism through use of different constructors. There is a default mechanism in place (using Java's `Class.getMethod`), but programmers who want a more flexible approach may create an instance of `MethodCall` using a customized method resolution mechanism.

References

[Ban] Bela Ban. Sample code for MethodCall. `http://www.cs/cornell.edu/home/bba/papers.html`.

[Ban98] Bela Ban. Javagroups – group communication patterns in java. Technical report, Cornell University Computer Science Dept., 1998. `http://www.cs.cornell.edu/home/bba/Patterns.ps.gz`.

[Ste90] Guy L. Steele. *Common LISP. The Language*. Digital Press, 1990.

[Sun96] Sun Microsystems Inc. *Java Core Reflection. API and Specification*, October 1996.